다형성
추상 메서드, Interface

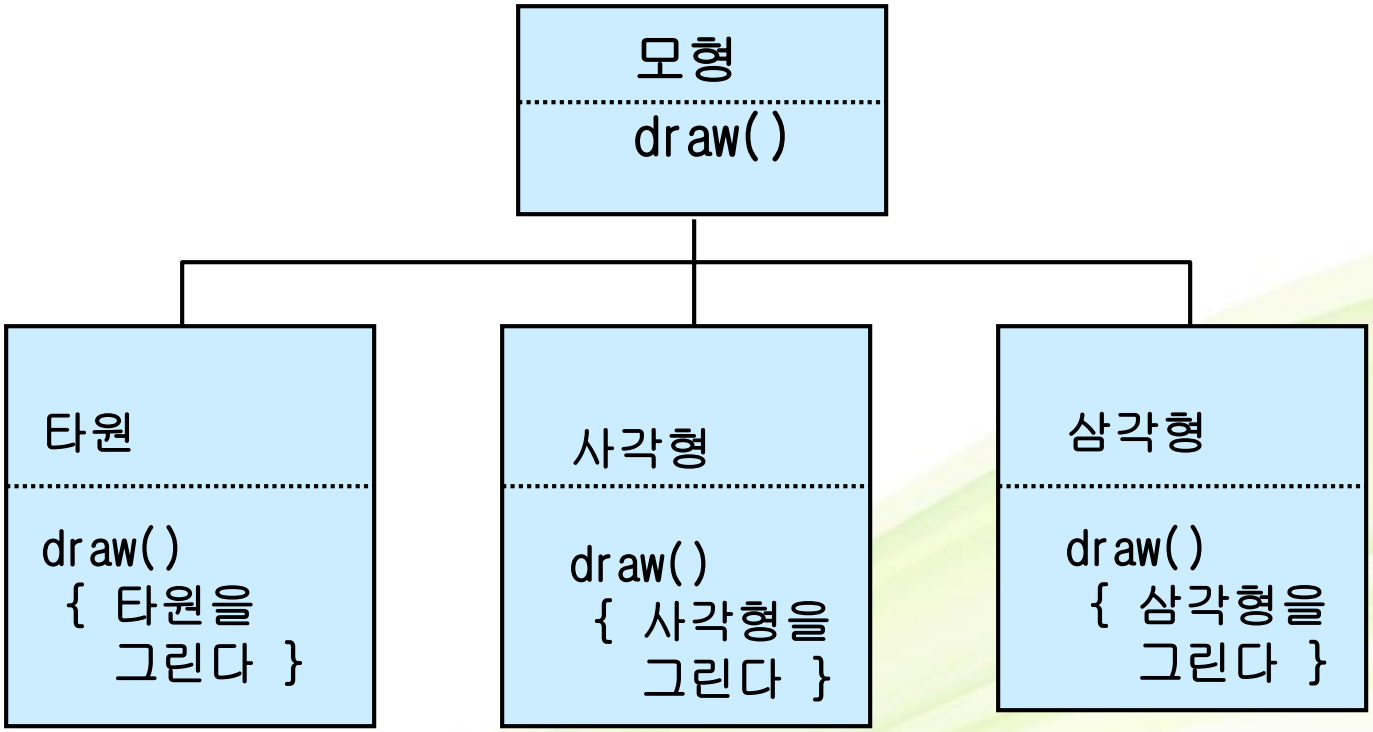
강사 : 강병준

다형성이란?

"one interface, multiple implementation"

하나의 인터페이스를 사용하여 다양한 구현 방법을 제공

하나의 클래스나 함수가 다양하게 동작하는 것



다형성(polymorphism)이란?

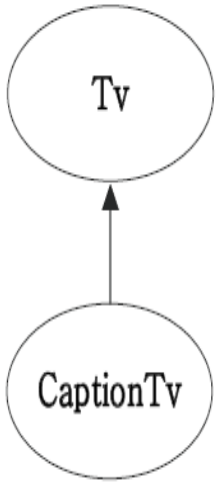
-"여러 가지 형태를 가질 수 있는 능력"

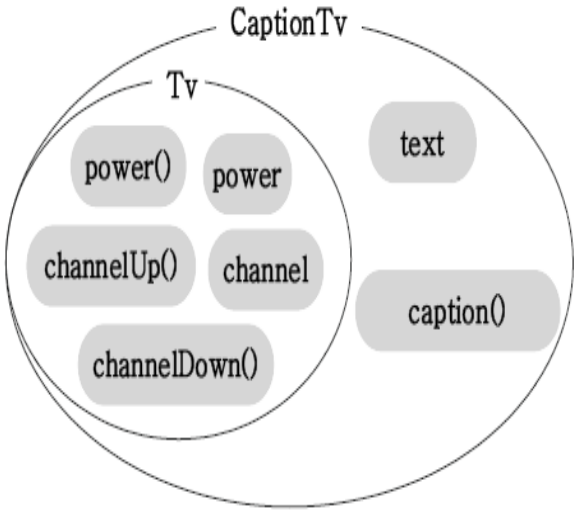
-"하나의 참조변수로 여러 타입의 객체를 참조할 수 있는 것"
즉, 조상타입의 참조변수로 자손타입의 객체를 다룰 수 있는 것이 다형성.

```
class Tv {
    boolean power;  // 전원상태(on/off)
    int channel;    // 채널

    void power(){   power = !power;}
    void channelUp(){   ++channel; }
    void channelDown(){ --channel; }
}

class CaptionTv extends Tv {
    String text;    // 캡션내용
    void caption() { /* 내용생략 */}
}
```






```
Tv        t = new Tv();
CaptionTv c = new CaptionTv();
```

↓

```
Tv        t = new CaptionTv();
```

```
CaptionTv c = new CaptionTv();

Tv        t = new CaptionTv();
```

오버라이딩(overriding)

오버라이딩(overriding)이란?

"조상클래스로부터 상속받은 메서드의 내용을 상속받는 클래스에 맞게 변경하는 것을 오버라이딩이라고 한다."

* override - vt. '~위에 덮어쓰다(overwrite).', '~에 우선하다.'

```
class Point {
      int x;
      int y;

      String getLocation() {
            return "x :" + x + ", y :"+ y;
      }
}

class Point3D extends Point {
      int z;
      String getLocation() {        // 오버라이딩
            return "x :" + x + ", y :"+ y + ", z :" + z;
      }
}
```

오버라이딩(overriding)

오버라이딩의 조건

1. 선언부가 같아야 한다.(이름, 매개변수, 리턴타입)

2. 접근제어자를 좁은 범위로 변경할 수 없다.
   - 조상의 메서드가 protected라면, 범위가 같거나 넓은 protected나 public으로만 변경할 수 있다.

3. 조상클래스의 메서드보다 많은 수의 예외를 선언할 수 없다.

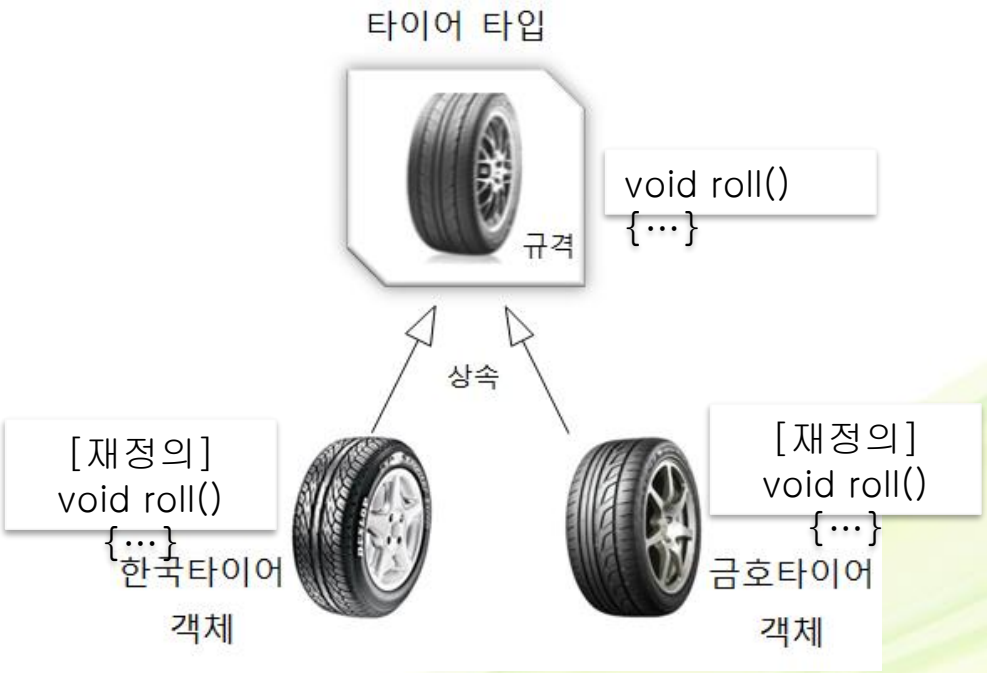
```
class Parent {
    void parentMethod() throws IOException, SQLException {
        // ...
    }
}

class Child extends Parent {
    void parentMethod() throws IOException {
        //..
    }
}

class Child2 extends Parent {
    void parentMethod() throws Exception {
        //..
    }
}
```

타입변환과 다형성(polymorphism)

- 필드의 다형성
  - 다형성을 구현하는 기술적 방법
    - 부모 타입으로 자동 변환
    - 재정의된 메소드(오버라이딩)



```
public class OverridingSuper {
	public void method(){
		System.out.println("난 super에서정의한메쏘드");
	}
}
public class OverridingSub extends OverridingSuper {
	public void method(){
		System.out.println("난 sub에서재정의한메쏘드");
	}
}
public class OverridingSuperSubMain {
	public static void main(String[] args) {
		OverridingSuper oSuper=new OverridingSuper();
		oSuper.method();
		OverridingSub oSub=new OverridingSub();
		oSub.method();
	}
}
```

오버라이딩(overriding)

오버로딩 vs. 오버라이딩

오버로딩(overloading) - 기존에 없는 새로운 메서드를 정의하는 것(new)
오버라이딩(overriding) - 상속받은 메서드의 내용을 변경하는 것(change, modify)

```
class Parent {
    void parentMethod() {}
}

class Child extends Parent {
    void parentMethod() {}          // 오버라이딩
    void parentMethod(int i) {}     // 오버로딩

    void childMethod() {}
    void childMethod(int i) {}      // 오버로딩
    void childMethod() {}           // 에러!!! 중복정의임
}
```

오버라이딩(overriding)

- 오버로딩(Overloading)과 오버라이딩(overriding)

구분	Method Overloading	Method Overriding
해석	메소드 다중정의	메소드 재정의
방법	같은 Class에서 동일한 Method 여러 개 존재	Super Class와 Sub Class에 동일한 Method 존재(틀만 가져와서 재정의)
매개변수	매개변수의 개수, 타입	매개변수 일치, 리턴 타입 일치

오버라이딩(overriding)

- 실습예제 – 메소드 다중정의(Overloading) 예

OverridingTest1.java

```
class Da {
      void show(String str) {
           System.out.println("상위  클래스의 " + str);
     }
}
class Db extends Da {
    void show( ) {
         System.out.println("하위클래스의  메소드 내용");
   }
}
public class OverridingTest1  {
      public static void main(String args[]) {
           Db over = new Db(); over.show(“메소드 내용”);
           over.show();
    }
}
```

제한자(final)

1. 클래스 앞에 붙일경우

   상속금지

   ex> public final class Test{}

2. 멤버 메소드 앞에 붙일경우

   오버라이딩 금지

   ex> public final void print(){}

3. 멤버변수 앞에 붙일 경우--> 상수

   ex> public final int PORT_NUMBER=80;

   상수화된다.

   변경금지

```
public class Parent {
	public static final int PORT_NUMBER=80;

	public void parent1(){
		System.out.println("난 일반메쏘드..");
		//this.PORT_NUMBER=90;
		//The final field Parent.PORT_NUMBER cannot be assigned

	System.out.println("PORT_NUMBER:"+Parent.PORT_NUMBER);
	}
	public final void parent2(){
		System.out.println("난 final 메쏘드 [재정의금지]");
	}

}
/*
public class StringChild  extends String{
}
*/
```

```
public class Child extends Parent {
	public void parent1() {
		System.out.println("parent1 재정의");
	}
	/*
	public void parent2(){
		//Cannot override the final method from Parent
	}
	*/
}
public class FinalMain {
    public static void main(String[] args) {
	Child c=new Child();
	//Child.PORT_NUMBER=87;
	System.out.println("Child.PORT_NUMBER:"+Child.PORT_NUMBER);
	c.parent1();
	c.parent2();
    }
}
```

```
public class Employee {
        public final double INCENTIVE_RATE=0.1;
        private String name;
        public Employee() {         }
        public Employee(String name){      this.name=name;            }
        public String getName() {          return name;       }
        public void setName(String name) {  this.name = name;        }
        //월급계산메쏘드[모든 자식들이 재정의 해야하는 메쏘드]
        public int computePay(){           return 0; }
        //incentive 계산메쏘드(회장님의 방침)
        public final int computeIncentive(){
                int result=0;              int pay=this.computePay();
                if(pay>=100000){
                        double temp = pay*INCENTIVE_RATE;
                        result=(int)temp;
                }else{    result=0;          }
                return result;
        }
}
```

```
public class SalaryEmployee extends Employee {
        private int annualSalary;
        public SalaryEmployee() {     }
        public SalaryEmployee(String name,int annualSalary) {
                super(name);                    this.annualSalary=annualSalary;
        }
        public int computePay() {
                int result = annualSalary/12;
                return result;
        }
        /*
        public final int computeIncentive(){  // override가 안됨
        }
         */
        public int getAnnualSalary() {
                return annualSalary;
        }
        public void setAnnualSalary(int annualSalary) {
                this.annualSalary = annualSalary;
        }
}
```

```
public class HourlyEmployee  extends Employee{
    private int hoursWorked;//일한시간
   private int moneyPerHour;//시간당받는돈
   public HourlyEmployee() {          }
   public HourlyEmployee(String name,int hoursWorked,int moneyPerHour) {
            super(name);
            this.hoursWorked=hoursWorked;     this.moneyPerHour=moneyPerHour;
   }
   public int computePay() {            return hoursWorked*moneyPerHour;          }
   public int getHoursWorked() {      return hoursWorked;          }
   public void setHoursWorked(int hoursWorked) {
                        this.hoursWorked = hoursWorked;
   }
   public int getMoneyPerHour() {
                        return moneyPerHour;
   }
   public void setMoneyPerHour(int moneyPerHour) {
                        this.moneyPerHour = moneyPerHour;
   }
}
```

```
public class EmployeeMain {
   public static void main(String[] args) {
            Employee[] emps=new Employee[5];
            emps[0] = new SalaryEmployee("김경호", 2000000);
            emps[1] = new SalaryEmployee("김경수", 1000000);
            emps[2] = new HourlyEmployee("김경미", 100, 800);
            emps[3] = new HourlyEmployee("김경록", 100,2000);
            emps[4] = new HourlyEmployee("김기록", 100,3000);

            for (int i = 0; i < emps.length; i++) {
                int pay=emps[i].computePay();
                int incentive =emps[i].computeIncentive();
                System.out.println("*****"+emps[i].getName()+"님 명세서*******");
                System.out.println("pay:"+pay);
                System.out.println("incentive:"+incentive);
            }
   }
}
```

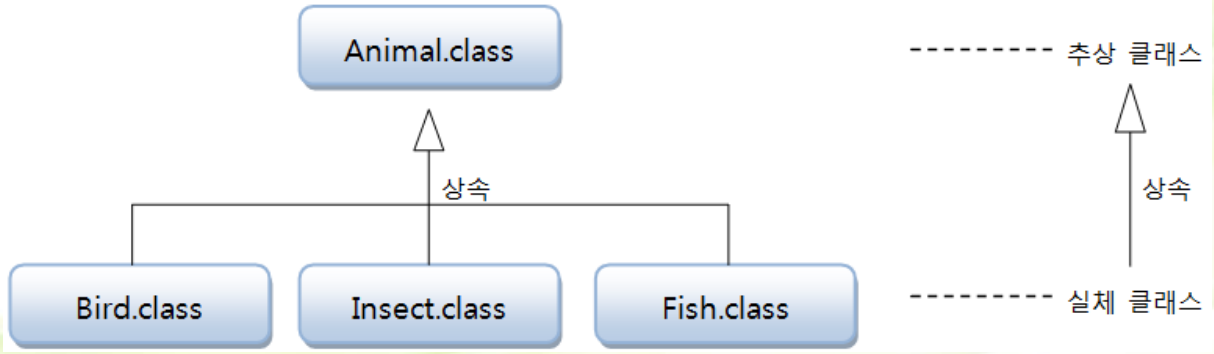
추상 클래스(Abstract Class)

❖ 추상 클래스 개념

- 추상(abstract)
  - 실체들 간에 공통되는 특성을 추출한 것
    - ⇨예1: 새, 곤충, 물고기→ 동물 (추상)
    - ⇨예2: 삼성, 현대, LG → 회사 (추상)

- 추상 클래스(abstract class)
  - 실체 클래스들의 공통되는 필드와 메소드 정의한 클래스
  - 추상 클래스는 실체 클래스의 부모 클래스 역할 (단독 객체 X)

*실체 클래스: 객체를 만들어 사용할 수 있는 클래스

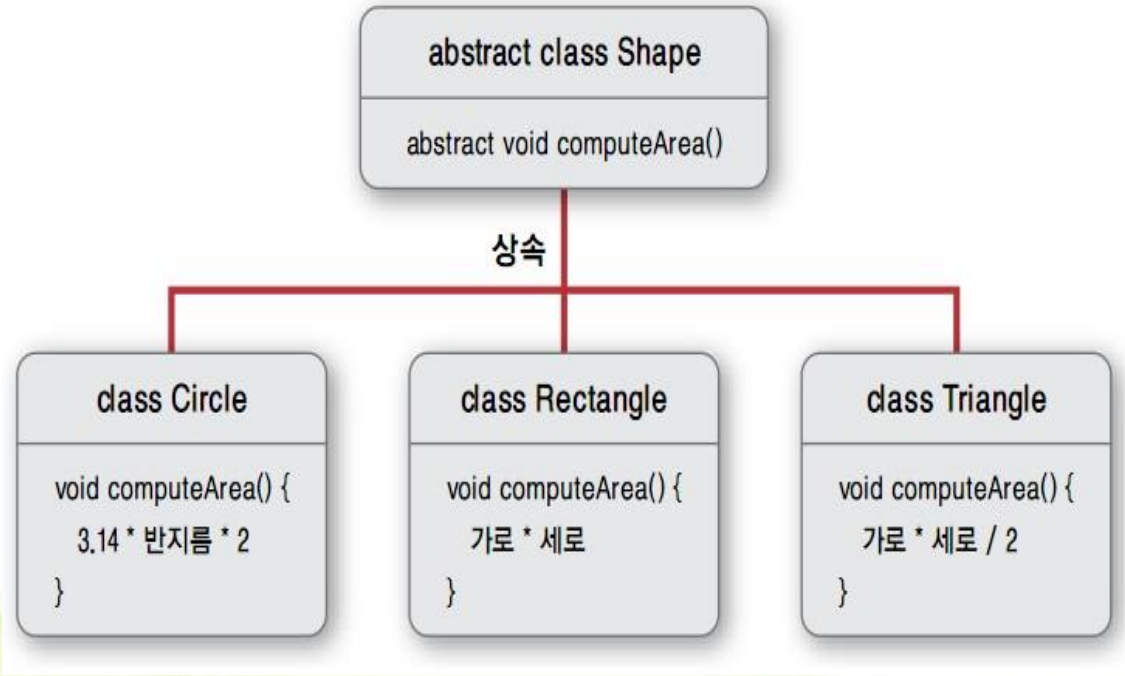


추상 클래스(Abstract Class)

- 추상 클래스의 용도
  - 실체 클래스의 공통된 필드와 메소드의 이름 통일할 목적
    - 실체 클래스를 설계자가 여러 사람일 경우,
    - 실체 클래스마다 필드와 메소드가 제각기 다른 이름을 가질 수 있음
  - 실체 클래스를 작성할 때 시간 절약
    - 실체 클래스는 추가적인 필드와 메소드만 선언
  - 실체 클래스 설계 규격을 만들고자 할 때
    - 실체 클래스가 가져야 할 필드와 메소드를 추상 클래스에 미리 정의
    - 실체 클래스는 추상 클래스를 무조건 상속 받아 작성

추상클래스(abstract class)

- **추상 클래스**
  - 클래스의 프레임만 구성
  - 하나 이상의 추상 메소드 포함
  - 직접 객체 생성 불가능
  - 추상 클래스에서 정의된 추상적인 기능은 하위 클래스에서 상세 구현



추상클래스(abstract class)

- **추상 메소드**
  - 추상 클래스 내에 정의되는 메소드로써 선언 부분만 있고 구현 부분이 없는 메소드
  - 하위 클래스는 상위 클래스에서 추상 메소드로 정의된 메소드를 재정의하여 사용

- **추상 클래스와 추상 메소드**

형식

```
abstract class 클래스이름 {
    ..........
    클래스에 기술할 수 있는 일반적인 멤버 변수와 메소드
    ..........
    abstract void 추상메소드이름();
}
```

추상 메소드는 선언 부분만 있다.
하위 클래스에서 오버라이딩하여 사용

추상클래스(abstract class)

- 추상 클래스와 추상 메소드

```
abstract class Shape {
    .....
    abstract void draw();
    .....
}
public class Circle extends Shape {
    .....
    void draw()
    {
    실제 원을 그리는 기능이 기술됨;
    }
}
public class Triangle extends Shape {
    .....
    void draw()
    {
    실제 삼각형을 그리는 기능이 기술됨;
    }
}
```

같은 이름의 메소드를 선언함으로서 오버라이딩

추상메서드(abstract method)란?

- 선언부만 있고 구현부(몸통, body)가 없는 메서드

```
/* 주석을 통해 어떤 기능을 수행할 목적으로 작성하였는지 설명한다. */
abstract 리턴타입 메서드이름();


Ex)
/* 지정된 위치(pos)에서 재생을 시작하는 기능이 수행되도록 작성한다.*/
abstract void play(int pos);
```

- 꼭 필요하지만 자손마다 다르게 구현될 것으로 예상되는 경우에 사용
- 추상클래스를 상속받는 자손클래스에서 추상메서드의 구현부를 완성

```
abstract class Player {
    ...
     abstract void play(int pos);      // 추상메서드
     abstract void stop();             // 추상메서드
    ...
}

class AudioPlayer extends Player {
    void play(int pos) { /* 내용 생략 */ }
    void stop() { /* 내용 생략 */ }
}

abstract class AbstractPlayer extends Player {
    void play(int pos) { /* 내용 생략 */ }
}
```

추상클래스의 작성

- 여러 클래스에 공통적으로 사용될 수 있는 추상클래스를 바로 작성하거나 기존클래스의 공통 부분을 뽑아서 추상클래스를 만든다.

```
class Marine {    // 보병
    int x, y;     // 현재 위치
    void move(int x, int y) { /* 지정된 위치로 이동 */ }
    void stop()             { /* 현재 위치에 정지 */ }
    void stimPack()         { /* 스팀팩을 사용한다.*/}
}

class Tank {      // 탱크
    int x, y;     // 현재 위치
    void move(int x, int y) { /* 지정된 위치로 이동 */ }
    void stop()             { /* 현재 위치에 정지 */ }
    void changeMode()       { /* 공격모드를 변환한다. */}
}

class Dropship {  // 수송선
    int x, y;     // 현재 위치
    void move(int x, int y) { /* 지정된 위치로 이동 */ }
    void stop()             { /* 현재 위치에 정지 */ }
    void load()             { /* 선택된 대상을 태운다.*/ }
    void unload()           { /* 선택된 대상을 내린다.*/ }
}
```

```
abstract class Unit {
    int x, y;
    abstract void move(int x, int y);
    void stop() {      /* 현재 위치에 정지 */ }
}

class Marine extends Unit {  // 보병
    void move(int x, int y) { /* 지정된 위치로 이동 */ }
    void stimPack()         { /* 스팀팩을 사용한다.*/}
}

class Tank extends Unit {    // 탱크
    void move(int x, int y) { /* 지정된 위치로 이동 */ }
    void changeMode()       { /* 공격모드를 변환한다. */}
}

class Dropship extends Unit { // 수송선
    void move(int x, int y) { /* 지정된 위치로 이동 */ }
    void load()             { /* 선택된 대상을 태운다.*/ }
    void unload()           { /* 선택된 대상을 내린다.*/ }
```

```
Unit[] group = new Unit[4];
group[0] = new Marine();
group[1] = new Tank();
group[2] = new Marine();
group[3] = new Dropship();

for(int i=0;i< group.length;i++) {
    group[i].move(100, 200);
}
```

추상메서드가 호출되는 것이 아니라 각 자손들에 실제로 구현된 move(int x, int y)가 호출된다.

추상화 형식

1.정의: 하나이상의 추상 메소드가 정의되어있는 클래스

```
ex> public abstract class Test{
        //추상 메소드();
        public abstarct int print(int i);
        //일반 메소드();
        public void test(){
        }
    }
```

==> 추상메소드: 메소드의 구현부분이 없고 원형(prototype) 만 존재하는 메소드

ex> public abstract int print(int i);

2. 추상클래스는 불완전한 추상 메소드를 가지므로 객체생성이 불가능하다. Test t=new Test();(X)

3. 추상클래스는 추상클래스를 상속받아서 추상 메소드를 구현(오버라이딩)하는 자식 클래스를 만들어 사용(객체생성)해야 한다

▶ class 를 추상화 시키는 형식

abstract class class-name{…..} ☞ abstract class Person{…..}

▶ method를 추상화 시키는 형식

abstract 접근 제어자 반환 자료형 method-name(인자);

☞ abstract public void setName(String name);

☞ abstract public String getName();

☞ abstract public void setAge(int age);

```
public abstract class AbstractClass {
	public void method1(){	System.out.println("난 일반 메쏘드");	}
	public abstract void method2();
}
public class ChildAbstractClass extends AbstractClass{
	public void method2() {
		System.out.println("난부모메쏘드의 구현(재정의)");
	}
}
public class AbstractMain {
	public static void main(String[] args) {
		//AbstractClass abc=new AbstractClass();
		ChildAbstractClass childAbs=new ChildAbstractClass();
		childAbs.method1();	childAbs.method2();
		AbstractClass abc=new ChildAbstractClass();
		abc.method1();
		abc.method2();
	}
}
```

```
abstract class One{
  int su1;
  abstract void suPrint();
  abstract int setSu(int i);
  }
class Three extends One{
  int count;
  void suPrint(){       System.out.println("su1=>"+su1);   }
  int setSu(int i){
    su1=i;
    System.out.println("su1_setting=>"+su1);
    return su1;
  }
  public static void main(String[] args){
    Three t= new Three();
    t.suPrint();
    System.out.println("setting value =>"+t.setSu(35));
    if(t instanceof One){
      System.out.println("ok");
    }else{
      System.out.println("no!");
    }
  }
}
```

```
abstract class Shape {
 abstract void draw();
}
class Circle extends Shape {
  void draw() {
    System.out.println("원그리는 기능");
  }
}
class Rectangle extends Shape {
  void draw() {
    System.out.println("사각형그리는 기능");
  }
}
class Triangle extends Shape {
  void draw() {
    System.out.println("삼각형그리는 기능");
  }
}
```

```
class AbstractClass {
  public static void main(String args[]) {
    Circle c = new Circle();
    c.draw();
    Rectangle r = new Rectangle();
    r.draw();
    Triangle t = new Triangle();
    t.draw();
   System.out.println("객체형변환과 오버라이딩을이용");
    Shape s = new Circle();
    s.draw();              // draw() 메소드 호출
    s = new Rectangle();
    s.draw();              // draw() 메소드 호출
    s = new Triangle();
    s.draw();              // draw() 메소드 호출
  }
}
```

추상화 실습예제

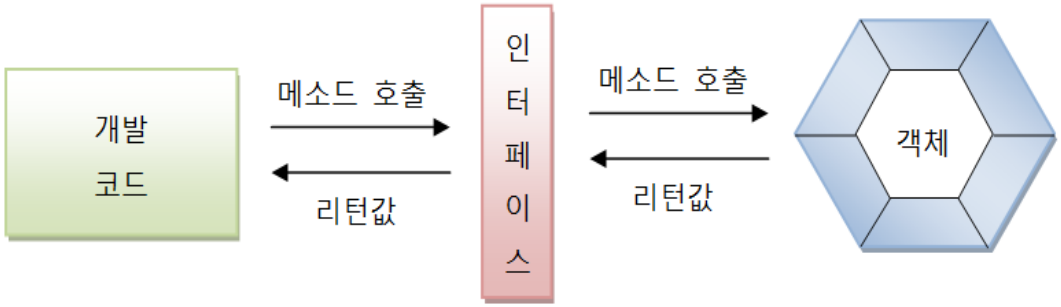
```
abstract class Person {
        String name="홍길동";      int age=14;
        abstract public void setName(String name);
        abstract public String getName();
        abstract public void setAge(int age);
        abstract public int getAge();
}
```

```
class Student extends Person{
  private int sno=123456;
  public void setName(String name){ this.name=name;}
  public void setAge(int age){ this.age=age;}
  public String getName(){ return name;}
  public int getAge(){ return age;}
  public void setSno(int sno){ this.sno = sno;}
  public int getSno(){ return sno; }
  public void printAll(){
   System.out.println("Name="+getName()+"Age="+getAge()+"Sno="+sno);    }
  public static void main(String[] args){
          Student s = new Student();     s.setName("kim");  s.printAll();
  }
}
```

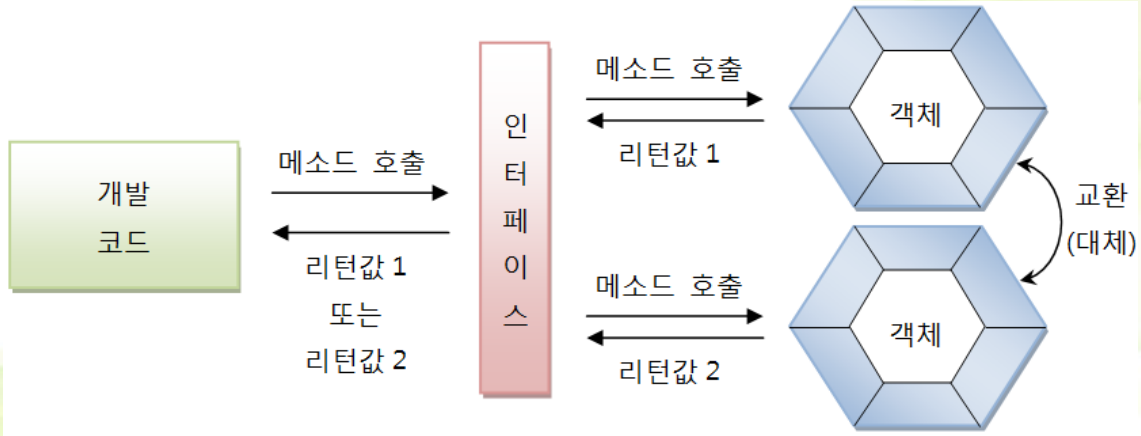
인터페이스의 역할

인터페이스란?

- 개발 코드와 객체가 서로 통신하는 접점
  - 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)



인터페이스 선언

❖ 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 { ... }
```

인터페이스 선언

❖ 인터페이스 선언

- 인터페이스의 구성 멤버

```
interface 인터페이스명 {
  //상수
  타입 상수명 = 값;
  //추상 메소드
  타입 메소드명(매개변수,...);
  //디폴트 메소드
  default 타입 메소드명(매개변수,...) {...}
  //정적 메소드
  static 타입 메소드명(매개변수) {...}
}
```

인터페이스 선언

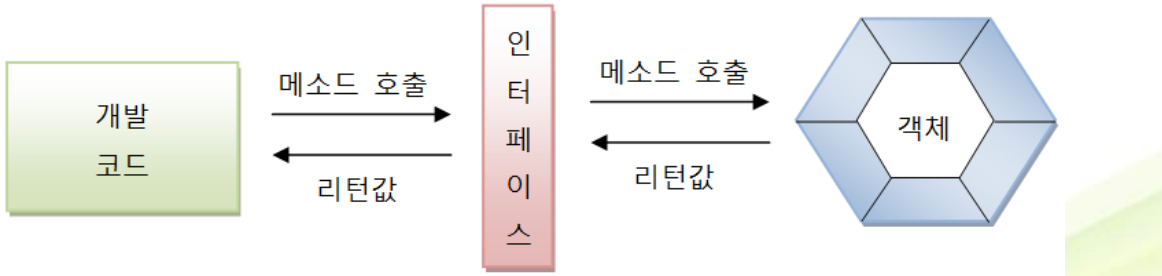
❖ 상수 필드 선언

- 인터페이스는 상수 필드만 선언 가능
  - ⇨ 데이터 저장하지 않음
- 인터페이스에 선언된 필드는 모두 public static final
  - ⇨자동적으로 컴파일 과정에서 붙음
- 상수명은 대문자로 작성
  - ⇨서로 다른 단어로 구성되어 있을 경우에는 언더 바(_)로 연결
- 선언과 동시에 초기값 지정
  - ⇨ static { } 블록 작성 불가 - static {} 으로 초기화 불가

인터페이스 선언

❖ 추상 메소드 선언

- 인터페이스 통해 호출된 메소드는 최종적으로 객체에서 실행
  - 인터페이스의 메소드는 기본적으로 실행 블록이 없는 추상 메소드로 선언
  - public abstract를 생략하더라도 자동적으로 컴파일 과정에서 붙게 됨



추상메소드
(메소드 선언부)
(호출 방법만 기술)

재정의된 메소드
(실제 실행 메소드)

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

인터페이스의 작성

- 'class'대신 'interface'를 사용한다는 것 외에는 클래스 작성과 동일하다.

```
interface 인터페이스이름 {
        public static final 타입 상수이름 = 값;
        public abstract 메서드이름(매개변수목록);
}
```

- 하지만, 구성요소(멤버)는 추상메서드와 상수만 가능하다.

- 모든 멤버변수는 public static final 이어야 하며, 이를 생략할 수 있다.
- 모든 메서드는 public abstract 이어야 하며, 이를 생략할 수 있다.

```
interface PlayingCard {
        public static final int SPADE = 4;
        final int DIAMOND = 3;      // public static final int DIAMOND = 3;
        static int HEART = 2;       // public static final int HEART = 2;
        int CLOVER = 1;             // public static final int CLOVER = 1;

        public abstract String getCardNumber();
        String getCardKind();  // public abstract String getCardKind();
}
```

인터페이스의 상속

- 인터페이스도 클래스처럼 상속이 가능하다.(클래스와 달리 다중상속 허용)

```
interface Movable {
    /** 지정된 위치(x, y)로 이동하는 기능의 메서드 */
    void move(int x, int y);
}

interface Attackable {
    /** 지정된 대상(u)을 공격하는 기능의 메서드 */
    void attack(Unit u);
}

interface Fightable extends Movable, Attackable { }
```

- 인터페이스는 Object클래스와 같은 최고 조상이 없다.

인터페이스의 구현

- 인터페이스를 구현하는 것은 클래스를 상속받는 것과 같다.

다만, 'extends' 대신 'implements'를 사용한다.

```
class 클래스이름 implements 인터페이스이름 {
    // 인터페이스에 정의된 추상메서드를 구현해야한다.
}
```

- 인터페이스에 정의된 추상메서드를 완성해야 한다.

```
class Fighter implements Fightable {
    public void move() { /* 내용 생략*/ }
    public void attack() { /* 내용 생략*/ }
}
```

```
interface Fightable {
    void move(int x, int y);
    void attack(Unit u);
}
```

```
abstract class Fighter implements Fightable {
    public void move() { /* 내용 생략*/ }
}
```

- 상속과 구현이 동시에 가능하다.

```
class Fighter extends Unit implements Fightable {
    public void move(int x, int y) { /* 내용 생략 */}
    public void attack(Unit u) { /* 내용 생략 */}
}
```

인터페이스를 이용한 다형성

- 인터페이스 타입의 변수로 인터페이스를 구현한 클래스의 인스턴스를 참조할 수 있다.

```
class Fighter extends Unit implements Fightable {
    public void move(int x, int y)  { /* 내용 생략 */ }
    public void attack(Fightable f) { /* 내용 생략 */ }
}
```

```
Fighter   f = new Fighter();
Fightable f = new Fighter();
```

- 인터페이스를 메서드의 매개변수 타입으로 지정할 수 있다.

```
void attack(Fightable f) {  // Fightable인터페이스를 구현한 클래스의 인스턴스를
    //...                   // 매개변수로 받는 메서드
}
```

- 인터페이스를 메서드의 리턴타입으로 지정할 수 있다.

```
Fightable method() { // Fightable인터페이스를 구현한 클래스의 인스턴스를 반환
    // ...
    return new Fighter();
}
```

타입변환과 다형성

❖ 필드의 다형성

[다형성은 객체를 부품화시킨다]


```
public interface Tire {
    public void roll();
}
```

```
public class HankookTire implements Tire {
    @Override
    public void roll() {
        System.out.println("한국 타이어가 굴러갑니다.");
    }
}
```

```
public class Car {
    Tire frontLeftTire = new HankookTire();
    Tire frontRightTire = new HankookTire();
    Tire backLeftTire = new HankookTire();
    Tire backRightTire = new HankookTire();

    void run() {
        frontLeftTire.roll();
        frontRightTire.roll();
        backLeftTire.roll();
        backRightTire.roll();
    }
```

```
Car myCar = new Car();
myCar.frontLeftTire = new KumhoTire();
myCar.frontRightTire = new KumhoTire();
```

```
myCar.run();
```

interface

1. 클래스가가진 모든 메쏘드가 추상메쏘드 형식임
2. 다중상속의 효과를 냄 (클래스는 불가능, 인터페이스는가능)

-형식 * class keyword 대신에 interface 라는 keyword를사용
* 추상메쏘드앞에 abstract 를 붙히지않는다.

```
ex>public interface Test{
        public void method1();
        public void method2();
   }
```

- 사용 : 상속(implements)받아서 재정의(구현)한후 사용한다.

```
ex> public class TestImpl implements Test{
                public void method1(){}
                public void method2(){}
    }
```

interface : 연결, 양식

- interface의 포함 멤버
  - ⇨public static final 멤버 필드
  - ⇨public abstract 멤버 메서드

```
interface AA {
        int aa();
}
class BB implements AA {     //오버라이딩
        public int aa()         {
                return 100;
        }
}
class CC implements AA {     //오버라이딩
        public int aa()         {
                return 200;
        }
}

class InterfaceTest2 {
        public static void     main(String args[]) {
                System.out.println(new BB().aa());
                System.out.println(new CC().aa());
        }
}
```

인터페이스(interface)

- 클래스에서 인터페이스 상속

형식1

```
public  class  클래스명  implements  인터페이스1, 인터페이스2 .....] {

..... 메소드 재정의
}
```

- 인터페이스에서 인터페이스 상속

형식2

```
public  interface  인터페이스  extends  인터페이스1,  인터페이스2.....] {
..... 상수선언
..... 메소드선언
}
```

```
interface AA  {
       int MAX=100; // final static 이다
      int aa(); // 생략해도 public 접근 지정자이다
}
interface BB {    double bb();    }
interface CC extends AA,BB {
            // MAX=700; // final staic 변수 (즉 상수형 변수)는 값변경 불가능
            boolean  cc();
}
class DD implements CC  {        // 오버라이딩
            //MAX=700; // final staic 변수 (즉 상수형 변수)는 값변경 불가능
            public int aa()          {   return 123;          }
            public double bb()     {   return 23.5;         }
            public boolean cc()    {   return true;         }
}
class InterfaceTest {
            public static void main(String args[])          {
                        DD D=new DD();
                        System.out.println("필드값 :"+D.MAX);
                        System.out.println("aa() 호출 :"+D.aa());
                        System.out.println("bb() 호출 :"+D.bb());
                        System.out.println("cc() 호출 :"+D.cc());
            }
}
```

```
interface AA{        int MAX=100;       int aa();  }
interface BB extends AA{      double PI=3.14;      double bb();   }
interface CC extends BB{
        char ch='A';          char cc();   }
class InterfaceTest3 implements CC{    // 오버라이딩
        public int aa() { return 500; }
        public double bb(){ return 188.8; }
        public char cc(){ return 'B';}
        public static void main(String args[])    {// 상위 클래스 변수로 하위 객체 처리
                CC C=new InterfaceTest3();
                if(C instanceof AA)             {
                        System.out.println("필드값 : "+C.MAX);
                        System.out.println("필드값 : "+C.PI);
                        System.out.println("필드값 : "+C.ch);
                        System.out.println();
                        System.out.println("메서드 호출  : "+C.aa());
                        System.out.println("메서드 호출  : "+C.bb());
                        System.out.println("메서드 호출  : "+C.cc());
                }
        }
}
```

```
public interface Volume{
        public void volumeUp(int level);
        public void volumeDown(int level);
}
```

```
public class Radio implements Volume{
  private   int          VolLevel;
  public Radio()       {
     VolLevel = 0;
  }
  public void volumeUp(int level)  {
    System.out.println("라디오 볼륨을
올려요");
    VolLevel += level;
 }
 public void volumeDown(int level) {
    System.out.println("라디오 볼륨을
내려요");
    VolLevel -= level;
 }
}
```

```
public class TV implements Volume {
  private   int          VolLevel;
  public TV() {       VolLevel = 0;   }
  public void volumeUp(int level)  {
      System.out.println("TV 볼륨을 올려요");
      VolLevel += level;                }
  public void volumeDown(int level)  {
      System.out.println("TV 볼륨을 내려요");
      VolLevel -= level;
      if(VolLevel < 0)  VolLevel = 0;
  }
}
```

```
public class Speaker implements Volume{
   private   int          VolLevel;
   public Speaker()     {  VolLevel = 0; }
   public void volumeUp(int level)  {
       System.out.println("스피커 볼륨을 올려요");
       VolLevel += level;
       if(VolLevel > 100)  VolLevel = 100;
   }
    public void volumeDown(int level)   {
      System.out.println("스피커 볼륨을 내려요");
      VolLevel -= level;
      if(VolLevel < 0)  VolLevel = 0;
  }
}
```

```
public class VolTest  {
   public static void main(String args[])                {
            Radio      radio = new Radio();
            TV          tv = new TV();
            Speaker    speaker = new Speaker();
            Volume    vol[] = new Volume[3];
            // 인터페이스에 개체 할당
            vol[0] = radio;
            vol[1] = tv;
            vol[2] = speaker;
            // 개체를 이용한 메서드 호출
            radio.volumeUp(1);
            tv.volumeUp(1);
            speaker.volumeUp(1);
            System.out.println("");
         System.out.println("이제부터는 인터페이스를 이용한 호출입니다요");
            // 인터페이스를 이용한 메서드 호출
            for(int i=0;i<3;i++)
                        vol[i].volumeUp(1);
            }
}
```

인터페이스의 장점

1. 개발시간을 단축시킬 수 있다.

일단 인터페이스가 작성되면, 이를 사용해서 프로그램을 작성하는 것이 가능하다. 메서드를 호출하는 쪽에서는 메서드의 내용에 관계없이 선언부만 알면 되기 때문이다.
그리고 동시에 다른 한 쪽에서는 인터페이스를 구현하는 클래스를 작성하도록 하여, 인터페이스를 구현하는 클래스가 작성될 때까지 기다리지 않고도 양쪽에서 동시에 개발을 진행할 수 있다.

2. 표준화가 가능하다.

프로젝트에 사용되는 기본 틀을 인터페이스로 작성한 다음, 개발자들에게 인터페이스를 구현하여 프로그램을 작성하도록 함으로써 보다 일관되고 정형화된 프로그램의 개발이 가능하다.

3. 서로 관계없는 클래스들에게 관계를 맺어 줄 수 있다.

서로 상속관계에 있지도 않고, 같은 조상클래스를 가지고 있지 않은 서로 아무런 관계도 없는 클래스들에게 하나의 인터페이스를 공통적으로 구현하도록 함으로써 관계를 맺어 줄 수 있다.

4. 독립적인 프로그래밍이 가능하다.

인터페이스를 이용하면 클래스의 선언과 구현을 분리시킬 수 있기 때문에 실제구현에 독립적인 프로그램을 작성하는 것이 가능하다.
클래스와 클래스간의 직접적인 관계를 인터페이스를 이용해서 간접적인 관계로 변경하면, 한 클래스의 변경이 관련된 다른 클래스에 영향을 미치지 않는 독립적인 프로그래밍이 가능하다.

인터페이스(interface)

● 인터페이스(interface)

- 추상 클래스와 유사(상수와 추상 메소드만 구성)
- 추상 클래스보다 더 완벽한 추상화 제공
- 다중 상속(Multiple Inheritance) 지원

구분	추상클래스	인터페이스

선언	abstract class 클래스명{
	변수;
	메소드(){ .....}
	abstract 메소드( );
	}
		interface 인터페이스명{
		상수;
		메소드( ); // 추상 메소드
		}

상속	class Sub **extends** Super{
	메소드 재정의(Overriding)
	}
		class Sub **implements** Interface1, Interface2{
		메소드 재정의(Overriding)
		}

장점	프레임 제공	프레임 제공, 다중 상속

```
public interface Lenderable {
        int BORROWED = 1; // 대출 중
        int NORMAL = 0; // 대출되지 않은 상태
        void checkOut(String name, String date);
        void checkIn();
}

class SeperateVolume implements Lenderable {
        String title, writer, name, date;
        int state;
        SeperateVolume(String title, String writer) {
                this.title = title; this.writer = writer;
        }
```

```
    public void checkOut(String name, String date) {
            if (state == SeperateVolume.BORROWED) return;
            this.name = name;   this.date = date;
            state = SeperateVolume.BORROWED;
            System.out.println("제목 : " + title);
            System.out.println("저자 : " + writer);
            System.out.println("대출자 : " + name);
            System.out.println("대출일 : " + date);
            System.out.println("====================");
    }
    public void checkIn() {
            if (state == SeperateVolume.NORMAL) return;
            state = SeperateVolume.NORMAL;
            System.out.println("제목:"+title+"이 반납 되었습니다");
            name = "";    date = "";
    }
    boolean stateInfo() {
            if (state == SeperateVolume.BORROWED) {
                    System.out.println("대출된 책입니다");
                    return false;
            } else { System.out.println("대출 가능 합니다");
                    return true;            }
    }
}
```

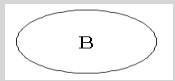
```
public class LenderableEx {
	public static void main(String[] args) {
		SeperateVolume sv = new SeperateVolume("윈피스","걔");
		if(sv.stateInfo()) sv.checkOut("홍길동","170120");

		if (!sv.stateInfo()) sv.checkIn();
	}
}
```

인터페이스의 이해

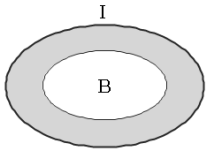
- 인터페이스는…
  - 두 대상(객체) 간의 '연결, 대화, 소통'을 돕는 '중간 역할'을 한다.
  - 선언(설계)와 구현을 분리시키는 것을 가능하게 한다.

```
class B {
   public void method() {
      System.out.println("methodInB");
   }
}
```




```
interface I {
   public void method();
}

class B implements I {
   public void method() {
      System.out.println("methodInB");
   }
}
```



- 인터페이스를 이해하려면 먼저 두 가지를 기억하자.
  - 클래스를 사용하는 쪽(User)과 클래스를 제공하는 쪽(Provider)이 있다.
  - 메서드를 사용(호출)하는 쪽(User)에서는 사용하려는 메서드(Provider)의 선언부만 알면 된다.

인터페이스(interface)

▶ 직접적인 관계의 두 클래스(A-B)

```
class A {
	public void methodA(B b) {
		b.methodB();
	}
}
```

```
class B {
	public void methodB() {
		System.out.println("methodB()");
	}
}
```

```
class InterfaceTest {
	public static void main(String args[]) {
		A a = new A();
		a.methodA(new B());
	}
}
```

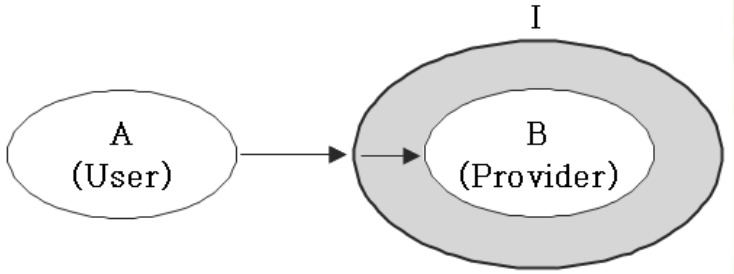
▶ 간접적인 관계의 두 클래스(A-I-B)

```
class A {
	public void methodA(I i) {
		i.methodB();
	}
}
```

```
interface I { void methodB(); }

class B implements I {
	public void methodB() {
		System.out.println("methodB()");
	}
}
```

```
class C implements I {
	public void methodB() {
		System.out.println("methodB() in C");
	}
}
```



인터페이스(interface)

```
class F1 {
	// void a(F2 f) {	f.b();	}
	void a(F3 f) {	f.b();	}
}
class F2 {
	void b() {		System.out.println("대박");		}
}
class F3 {
	void b() {		System.out.println("쪽박");		}
}

public class InteEx1 {
	public static void main(String[] args) {
	//		F2 f2 = new F2();
			F3 f3 = new F3();
			F1 f1 = new F1();
			f1.a(f3); // f1.a(f2);
	}
}
```

인터페이스(interface)

```
interface I {         void b();}
class G1 {
        void a(I f) {         f.b();      }
}
class G2 implements I{
        public void b() {  System.out.println("대박");          }
}
class G3 implements I {
        public void b() {  System.out.println("쪽박");          }
}
public class InteEx2 {
        public static void main(String[] args) {
         //   G2 f2 = new G2();
            G3 f2 = new G3();
                    G1 f1 = new G1();
                    f1.a(f2);
        }
}
```

```
 interface I {
     public abstract void play();
 }
class A {
   void autoPlay(I i) {          i.play();     }
 }
class B implements I {
    public void play() {          System.out.println("play in B class");     }
 }
 class C implements I {
    public void play() {
        System.out.println("play in C class");
    }
 }
class InterfaceTest2 {
            public static void main(String[] args) {
                        A a = new A();
                        a.autoPlay(new B());
                        a.autoPlay(new C());
            }
}
```

```
class RepairableTest {
    public static void main(String[] args)              {
            Tank tank = new Tank();
            Dropship dropship = new Dropship();
            Marine marine = new Marine();
            SCV       scv = new SCV();
            scv.repair(tank);                  // SCV가 Tank를 수리하도록 한다.
            scv.repair(dropship);
           //  scv.repair(marine);
           // 에러! repair(Repairable)  in SCV cannot be applied to (Marine)
            }
}

interface Repairable {}
class GroundUnit extends Unit {
      GroundUnit(int hp) {            super(hp);     }
}
class AirUnit extends Unit {
      AirUnit(int hp) {           super(hp);     }
}
```

```
class Unit {
    int hitPoint;
    final int MAX_HP;
    Unit(int hp) {          MAX_HP = hp;    }
}
class Tank extends GroundUnit implements Repairable {
        Tank() {
                super(150);                     // Tank의 HP는 150이다.
                hitPoint = MAX_HP;
        }
        public String toString() {
            return "Tank";
        }
}
class Dropship extends AirUnit implements Repairable {
        Dropship() {
                super(125);                     // Dropship의 HP는 125이다.
                hitPoint = MAX_HP;
        }
        public String toString() {      return "Dropship“;  }
}
```

```
class Marine extends GroundUnit {
	Marine() {
		super(40);
		hitPoint = MAX_HP;
	}
}
class SCV extends GroundUnit implements Repairable{
	SCV() {
		super(60);
		hitPoint = MAX_HP;
	}
	void repair(Repairable r) {
		if (r instanceof Unit) {
			Unit u = (Unit)r;
			while(u.hitPoint!=u.MAX_HP) {
				/* Unit의 HP를 증가시킨다. */
				u.hitPoint++;
			}
			System.out.println( u.toString() + "의 수리가 끝났습니다.");
		}
	}
}
```

인터페이스 선언

❖ 디폴트 메소드 선언

- 자바8에서 추가된 인터페이스의 새로운 멤버

```
[public] default 리턴타입 메소드명(매개변수, ...) { ... }
```

- 실행 블록을 가지고 있는 메소드
- default 키워드를 반드시 붙여야
- 기본적으로 public 접근 제한

  ⇨생략하더라도 컴파일 과정에서 자동 붙음

❖ 모든 구현 객체가 가지고 있는 기본 메소드

디폴트 메소드 선언

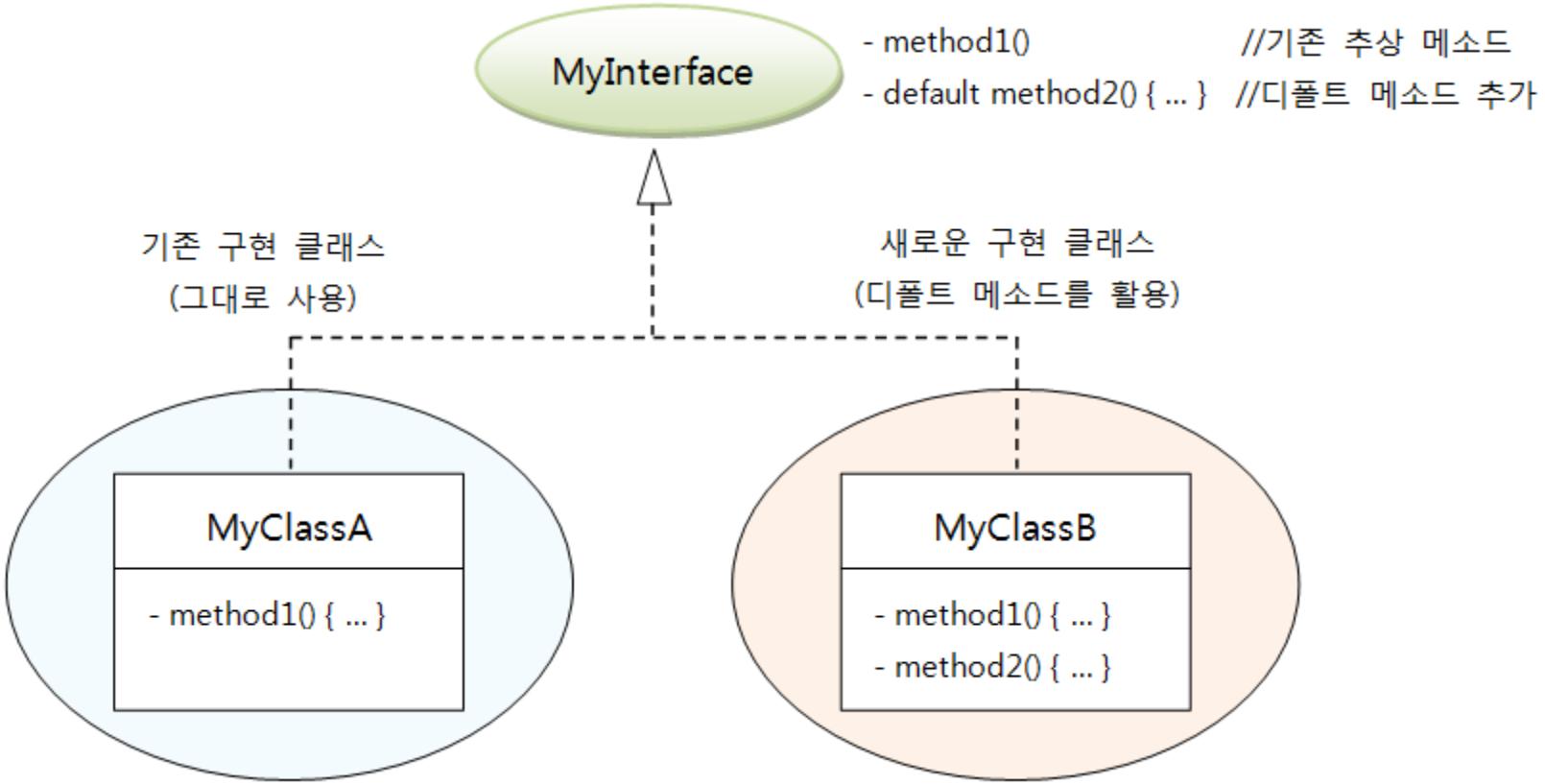
```
public interface RemoteControl {
        //상수
        int MAX_VOLUME = 10;
        int MIN_VOLUME = 0;

        //추상 메소드
        void turnOn();
        void turnOff();
        void setVolume(int volume);

        //디폴트 메소드
        default void setMute(boolean mute) {
                if(mute) {
                        System.out.println("무음 처리합니다.");
                } else {
                        System.out.println("무음 해제합니다.");
                }
        }
}
```

디폴트 메소드와 인터페이스 확장

❖ 디폴트 메소드와 확장 메소드 사용하기



인터페이스 선언

- 정적 메소드 선언
  - 자바8에서 추가된 인터페이스의 새로운 멤버

```
[public] static 리턴타입 메소드명(매개변수, ...) { ... }
```

```
public interface RemoteControl {
    //정적 메소드
    static void changeBattery() {
        System.out.println("건전지를 교환합니다.");
    }
}

public class RemoteControlExample {
        public static void main(String[] args) {
                RemoteControl.changeBattery();
        }
}
```

인터페이스 선언

```
public interface RemoteControl {
        //상수
        int MAX_VOLUME = 10;
        int MIN_VOLUME = 0;
        //추상 메소드
        void turnOn();
        void turnOff();
        void setVolume(int volume);
        //디폴트 메소드
        default void setMute(boolean mute) {
                if(mute) {
                        System.out.println("무음 처리합니다.");
                } else {
                        System.out.println("무음 해제합니다.");
                }
        }
        //정적 메소드
        static void changeBattery() {
                System.out.println("건전지를 교환합니다.");
        }
}
```

인터페이스 선언

```
public class RemoteControlExample {
        public static void main(String[] args) {
                RemoteControl[] rc = new RemoteControl[2];
                rc[0] = new Television();
                rc[1] = new Audio();
                for(RemoteControl r: rc) {
                        r.turnOn();
                        r.turnOff();
                        r.setVolume(7);
                        r.setMute(true);
                        RemoteControl.changeBattery();
                }
        }
}
```

인터페이스 구현

❖ 익명 구현 객체

- 명시적인 구현 클래스 작성 생략하고 바로 구현 객체를 얻는 방법
  - 이름 없는 구현 클래스 선언과 동시에 객체 생성

```
인터페이스 변수 = new 인터페이스() {
  //인터페이스에 선언된 추상 메소드의 실체 메소드 선언
};
```

  - 인터페이스의 추상 메소드들을 모두 재정의하는 실체 메소드가 있어야
  - 추가적으로 필드와 메소드 선언 가능하나 익명 객체 안에서만 사용
  - 인터페이스 변수로 접근 불가

익명 구현객체

```
public class RemoteEX2 {
    public static void main(String[] args) {
        RemoteControl rc = new RemoteControl() {
            public void turnOn()  { System.out.println("스피커 켜");  }
            public void turnOff() {  System.out.println("스피커 꺼");  }
            public void setVolume(int volume) {
                    if(volume>RemoteControl.MAX_VOLUME) {
                            volume = RemoteControl.MAX_VOLUME;
                    } else if(volume<RemoteControl.MIN_VOLUME) {
                            volume = RemoteControl.MIN_VOLUME;
                    }
                    System.out.println("현재 TV 볼륨: " + volume);
            }
        };
        rc.turnOn();
        rc.turnOff();
        rc.setMute(true);
        rc.setVolume(15);
        RemoteControl.changeBattery();
    }
}
```

연습문제

1. 다음과 같은 실행결과를 얻도록 코드를 완성하시오.

[Hint] instanceof연산자를 사용해서 형변환한다.

- 메서드명 : action
- 기 능 : 주어진 객체의 메서드를 호출한다.
  DanceRobot인 경우, dance()를 호출하고, SingRobot인 경우, sing()을 호출하고, DrawRobot인 경우, draw()를 호출한다.
- 반환타입 : 없음
- 매개변수 : Robot r - Robot인스턴스 또는 Robot의 자손 인스턴스

1. 답

```
Class Ex01 {
  public static void action(Robot r) {
      if (r instanceof DanceRobot) {  DanceRobot dr = (DanceRobot)r;     dr.dance();
      } else if(r instanceof SingRobot) {  SingRobot sr = (SingRobot)r;      sr.sing();
      } else if(r instanceof DrawRobot)  { DrawRobot dr = (DrawRobot)r; dr.draw();       }
   }
   public static void main(String[] args) {
        Robot[] arr = { new DanceRobot(), new SingRobot(), new DrawRobot()};
        for(int i=0; i< arr.length;i++)  action(arr[i]);
   } // main
}
class Robot {}

class DanceRobot extends Robot {
    void dance() {  System.out.println("춤을 춥니다.");   }
}
class SingRobot extends Robot {
    void sing() {  System.out.println("노래를 합니다.");  }
}
class DrawRobot extends Robot {
    void draw() {  System.out.println("그림을 그립니다.");  }
)
```

어노테이션(Annotation)

- 어노테이션(Annotation)이란?
  - 프로그램에게 추가적인 정보를 제공해주는 메타데이터(metadata)
  - 어노테이션 용도
    - 컴파일러에게 코드 작성 문법 에러 체크하도록 정보 제공
    - 소프트웨어 개발 툴이 빌드나 배치 시 코드를 자동 생성하게 정보 제공
    - 실행 시(런타임시) 특정 기능 실행하도록 정보 제공

어노테이션(Annotation)

- 어노테이션 타입 정의와 적용
  - 어노테이션 타입 정의
    - 소스 파일 생성: AnnotatoinName.java
    - 소스 파일 내용

```
public @interface AnnotationName {
}
```

  - 어노테이션 타입 적용

```
@AnnotationName
```

```
@Override
public void toString() { ... }
```

어노테이션(Annotation)

❖ 기본 엘리먼트 value

```
public @interface AnnotationName {
  String value();
  int elementName() default 5;
}
```

기본 엘리먼트 선언

```
@AnnotationName("값");
```

- 어노테이션 적용할 때 엘리먼트 이름 생략 가능

```
@AnnotationName(value="값", elementName=3);
```

- 두 개 이상의 속성을 기술할 때에는 value=값 형태로 기술