

# 상속과 다형성

# 1. 상속(is a)

- ❖ 상위 클래스의 모든 멤버를 하위클래스가 물려 받는 것
- ❖ 상속은 객체지향언어의 주요한 특징 중 하나인 재 사용성(CBD로 많이 이전)과 코드의 간결성(중복된 코드를 하나의 클래스에 표현)을 제공
- ❖ 상속의 장점
  - ✓ 코드를 재활용함으로써 간소화된 클래스 구조 이용
  - ✓ 클래스의 기능 테스트에 대한 생산성 및 정확성이 증가
  - ✓ 클래스의 수정/추가에 대한 유연성 및 확장성이 증가

# 1. 상속(is a)

- ❖ 자신이 만든 클래스로부터 상속(코드의 중복 제거나 다형성 구현이 목적) 받을 수 있고 프레임워크 (ex JDK, Spring..)가 제공해주는 클래스로부터 상속(기능 확장이 목적)을 받을 수 있음
- ❖ Java의 모든 클래스는 상위클래스로부터 상속을 받아야 되는데 명시적으로 상위 클래스를 기재하지 않는다면 상위 클래스는 `java.lang.Object`
- ❖ 상위 클래스를 지정하기 위해 자바에서는 `extends`라는 키워드를 사용
- ❖ 상속의 종류
  - ✓ 단일 상속: 하나의 클래스로부터 상속받는 경우
  - ✓ 다중 상속: 2개 이상의 클래스로부터 상속받는 경우
- ❖ 상속의 용어
  - ✓ 상속하는 클래스 - Base, Super, Parent Class
  - ✓ 상속받는 클래스 - Derivation, Sub, Child Class

# 1. 상속(is a)

## ❖상속 형식

```
[public/final/abstract] class 클래스이름 extends 상위클래스이름
{
    멤버 변수;
    멤버 메소드;
}
```

❖상위 클래스의 모든 멤버는 상속되지만 private 멤버는 하위 클래스에서 직접 접근할 수 없음

❖초기화 블록과 생성자는 상속되지 않음

❖동일한 이름의 변수가 상위클래스와 하위클래스에 존재하는 경우 상위클래스의 변수는 가려지게 되  
서 public이라 하더라도 인스턴스를 이용해서는 직접 접근이 불가능

❖static 변수도 상속

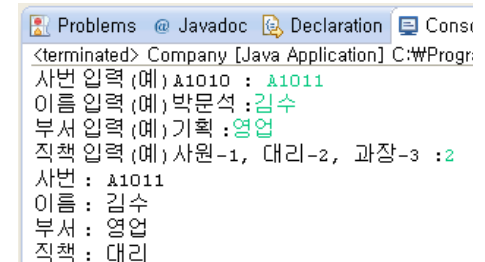
❖하위 클래스 객체가 생성될 때 상위 클래스의 생성자가 먼저 호출되서 상위 클래스의 멤버 필드를 먼저 만들고 하위 클래스의 멤버 필드를 생성

상위 클래스 멤버 필드
하위 클래스 멤버 필드

# 실습(Company-Employee.java)

```
package company;
```

```
public class Employee {  
    //사원번호, 이름, 부서를 저장할 멤버 변수를 선언  
    //protected는 상속받은 클래스에서 사용할 수 있고  
    //객체가 사용하지는 못합니다.  
    protected int empNo;  
    protected String name;  
    protected String part;  
  
    //일련번호 생성을 위한 static 변수 선언  
    protected static int autoIncrement;  
  
    public Employee() {  
        super();  
        empNo = ++autoIncrement;  
    }  
    public Employee(String name, String part) {  
        super();  
        empNo = ++autoIncrement;  
        this.name = name;  
        this.part = part;  
    }  
}
```



```
Problems Javadoc Declaration Console  
<terminated> Company [Java Application] C:\WPProgr  
사번 입력 (예) 1010 : 1011  
이름 입력 (예) 박문석 : 김수  
부서 입력 (예) 기획 : 영업  
직책 입력 (예) 사원-1, 대리-2, 과장-3 : 2  
사번 : 1011  
이름 : 김수  
부서 : 영업  
직책 : 대리
```

# 실습(Company-Employee.java)

//접근자 메소드

```
public int getEmpNo() {  
    return empNo;  
}  
public void setEmpNo(int empNo) {  
    this.empNo = empNo;  
}  
public String getName() {  
    return name;  
}  
public void setName(String name) {  
    this.name = name;  
}  
public String getPart() {  
    return part;  
}  
public void setPart(String part) {  
    this.part = part;  
}
```

# 실습(Company-Employee.java)

//멤버 변수의 값들을 문자열로 만들어서 리턴해주는 메소드

```
public String resultStr(){  
    String str="";  
    str += "사번:" + empNo + "Wn";  
    str += "이름:" + name + "Wn";  
    str += "부서:" + part + "Wn";  
    str += "=====Wn";  
    return str;  
}
```

```
}
```

# 실습(Company-Manager.java)

```
package company;
//Employee 클래스로 부터 상속받는 클래스
public class Manager extends Employee {
    private String position;

    public Manager() {
        empNo = ++autoIncrement;
    }
    public Manager(String name, String part, String position) {
        empNo = ++autoIncrement;
        this.name = name;
        this.part = part;
        this.position = position;
    }
    public String getPosition() {
        return position;
    }
    public void setPosition(String position) {
        this.position = position;
    }
    //position을 문자열로 만들어서 리턴해주는 메소드
    public String addStr(){
        return "직책:" + position + "Wn";
    }
}
```



# 실습(Company-Company.java)

```
package company;
import java.util.*;
public class Company {

    public static void main(String[] args) {
        //출력할 때 사용할 임시변수
        String result = "";

        //Employee와 Manager 객체의 참조를 저장할 변수
        Employee emp = null;
        Manager man = null;

        //3가지 정보를 입력받아서 저장할 임시 변수
        String name="";
        String part="";
        String position = "";

        Scanner scanner = new Scanner(System.in);

        System.out.print("이름을 입력하세요:");
        name = scanner.next();
        System.out.print("부서를 입력하세요:");
        part = scanner.next();
        System.out.print("직책을 입력하세요(1.사원 2.관리자:");
        position = scanner.next();
```

# 실습(Company-Company.java)

```
//position 1이면 Employee 객체를 만들어서 출력
//position 2이면 Manager 객체를 만들어서 출력
if(position.equals("1")){
    emp = new Employee(name, part);
    result = emp.resultStr();
}
else{
    man = new Manager(name, part, "관리자");
    result = man.resultStr() + man.addStr();
}
System.out.println(result);
scanner.close();
}
}
```

## 2. super

- ❖ 하위 클래스의 인스턴스 메소드 안에서 상위 클래스 객체를 가리키는 포인터
- ❖ super는 상위 클래스와 하위클래스에 동일한 멤버 필드나 멤버 메소드가 있을 때 상위 클래스의 멤버를 명시적으로 지시하기 위해 사용
- ❖ super()는 상위 클래스의 생성자를 호출하는 것으로 상위 클래스의 생성자는 하위 클래스에서는 생성자에서만 호출이 가능하며 다른 문장보다 무조건 먼저 와야 함
- ❖ 사용방법
  - ✓ 멤버 필드 접근: super.멤버필드
  - ✓ 멤버 메소드 접근: super.멤버메소드(매개변수)
  - ✓ 생성자 접근: super(매개변수) – 생성자 메소드에서만 사용이 가능

# 실습(company1-Manager.java)

```
package Company1;

public class Manager extends Employee {
    private String position; // 직책

    public Manager() {
        super();
    }

    public Manager(String name, String part, String position) {
        super(name, part);
        this.position = position;
    }

    public String addStr() {
        String result = super.resultStr();
        result += "직책 : " + position + "Wn";
        return result;
    }
}
```

# 실습(company1-Company.java)

```
package company1;
import java.util.*;
public class Company {

    public static void main(String[] args) {
        //출력할 때 사용할 임시변수
        String result = "";

        //Employee와 Manager 객체의 참조를 저장할 변수
        Employee emp = null;
        Manager man = null;

        //3가지 정보를 입력받아서 저장할 임시 변수
        String name="";
        String part="";
        String position = "";

        Scanner scanner = new Scanner(System.in);

        System.out.print("이름을 입력하세요:");
        name = scanner.next();
        System.out.print("부서를 입력하세요:");
        part = scanner.next();
        System.out.print("직책을 입력하세요(1.사원 2.관리자:");
        position = scanner.next();
```

# 실습(Company1-Company.java)

```
//position 1이면 Employee 객체를 만들어서 출력
//position 2이면 Manager 객체를 만들어서 출력
if(position.equals("1")){
    emp = new Employee(name, part);
    result = emp.resultStr();
}
else{
    man = new Manager(name, part, "관리자");
    result += man.addStr();
}
System.out.println(result);
scanner.close();
}
}
```

## 2. super

- ❖ 다른 클래스로부터 상속을 받는 하위 클래스에서 생성자를 반드시 만들어야 하는 경우가 있는데 이러한 경우는 상위 클래스에 디폴트 생성자(매개변수가 없는 생성자)가 없는 경우
- ❖ 상위 클래스에 디폴트 생성자가 없으면 하위 클래스의 생성자에서 상위 클래스의 생성자를 명시적으로 호출해야 함
- ❖ 하위 클래스에서 생성자를 만들지 않으면 아래와 같은 생성자가 있는 것으로 간주

```
public 생성자(){  
    super();  
}
```

- ❖ 해결책은 2가지
  - ✓ 상위 클래스에 디폴트 생성자를 생성 - 프레임워크가 제공하는 클래스에서는 불가능
  - ✓ 하위 클래스의 생성자에서 상위 클래스의 생성자를 호출 - super()

# 실습(company2-Employee.java)

```
package company2;
public class Employee {
    private String empNo; // 사원번호
    private String name; // 이름
    private String part; // 부서

    // 매개변수3개를 갖는 생성자
    public Employee(String empNo, String name, String part) {
        this.empNo = empNo;
        this.name = name;
        this.part = part;
    }

    public String getEmpNo() {
        return empNo;
    };

    public String getName() {
        return name;
    };
};
```



# 실습(company2-Employee.java)

```
public String getPart() {
    return part;
};
public void setEmpNo(String empNo) {
    this.empNo = empNo;
};
public void setName(String name) {
    this.name = name;
};

public void setPart(String part) {
    this.part = part;
};

// 멤버 필드의 값을 결과 문자열로 결합
public String resultStr() {
    String result = "";

    result += "사번 : " + empNo + "Wn";
    result += "이름 : " + name + "Wn";
    result += "부서 : " + part + "Wn";

    return result;
}
}
```

# 실습(company2-Manager.java)

```
Manager.java
1 package Company2;
2
3 public class Manager extends Employee {
4     private String position; // 직책
5
6     public Manager() {
7
8     }
9     // 매개변수 4개를 갖는 생성자
10    public Manager(String empNo, String name, String part, String position) {
11        super(empNo, name, part);
12        this.position = position;
13    }
14
15    public String addStr() {
16        String result = super.resultStr();
17        result += "직책 : " + position + "\n";
18        return result;
19    }
20 }
21
```

# 실습(company2-Manager.java)

```
package company2;

public class Manager extends Employee {
    private String position; // 직책

    public Manager() {
        super("A0000", "noname", "대기발령");
    }
    // 매개변수 4개를 갖는 생성자
    public Manager(String empNo, String name, String part, String position) {
        super(empNo, name, part);
        this.position = position;
    }

    public String addStr() {
        String result = super.resultStr();
        result += "직책 : " + position + "\n";
        return result;
    }
}
```

### 3. MethodOverriding(재정의)

- ❖ 상속 관계에 있는 클래스들에 원형이 동일한 메소드가 존재하는 경우
- ❖ 오버라이딩 된 메소드는 오브젝트가 메소드를 호출하면 객체의 타입(메모리 할당 시 호출한 생성자)에 따라 메소드를 호출
- ❖ 상위 클래스와 하위 클래스에 동일한 동일한 원형의 메소드가 존재하는 경우는 하위 클래스에 존재하는 메소드의 접근 지정자가 상위 클래스의 메소드의 접근 지정자보다 더 크거나 같아야 함
- ❖ public > protected > default > private

# 실습(overriding-Employee.java)

```
package overriding;
```

```
public class Employee {  
    private String empNo; // 사원번호  
    private String name; // 이름  
    private String part; // 부서  
  
    public Employee() {  
        super();  
    }  
  
    // 매개변수3개를 갖는 생성자  
    public Employee(String empNo, String name, String part) {  
        this.empNo = empNo;  
        this.name = name;  
        this.part = part;  
    }  
}
```

# 실습(overriding-Employee.java)

```
public String getEmpNo() {  
    return empNo;  
};  
  
public String getName() {  
    return name;  
};  
public String getPart() {  
    return part;  
};  
public void setEmpNo(String empNo) {  
    this.empNo = empNo;  
};  
public void setName(String name) {  
    this.name = name;  
};  
  
public void setPart(String part) {  
    this.part = part;  
};
```

# 실습(overriding-Employee.java)

// 멤버 필드의 값을 결과 문자열로 결합

```
public String resultStr() {  
    String result = "";  
  
    result += "사번 : " + empNo + "Wn";  
    result += "이름 : " + name + "Wn";  
    result += "부서 : " + part + "Wn";  
  
    return result;  
}  
}
```

# 실습(overriding-Manager.java)

```
package overriding;
```

```
public class Manager extends Employee {  
    private String position; // 직책  
  
    public Manager() {  
  
    }  
    // 매개변수 4개를 갖는 생성자  
    public Manager(String empNo, String name, String part, String position) {  
        super(empNo, name, part);  
        this.position = position;  
    }  
  
    // 관리자에서 추가되는 정보를 결과 문자열로 결합  
    public String resultStr() {  
        String result = super.resultStr();  
        result += "직책 : " + position + "Wn";  
        return result;  
    }  
}
```



# 실습(overriding-Company.java)

```
import java.util.Scanner;

public class Company {
    public static void main(String args[]) {
        String result = ""; // 결과 문자열
        Employee emp = null; // Employee객체의 레퍼런스 변수
        Manager mng = null; // Manager객체의 레퍼런스 변수
        String empNo = null; // 사원번호를 입력받는 변수
        String name = null; // 이름을 입력받는 변수
        String part = null; // 부서를 입력받는 변수
        String position = null; // 직책을 입력받는 변수
        Scanner sc = new Scanner(System.in);

        System.out.print("사번 입력(예)A1010 : ");
        empNo = sc.nextLine();
        System.out.print("이름 입력(예)박문석 :");
        name = sc.nextLine();
        System.out.print("부서 입력(예)기획 :");
        part = sc.nextLine();
        System.out.print("직책 입력(예)사원-1, 대리-2, 과장-3 :");
        position = sc.nextLine();
    }
}
```

# 실습(overriding-Company.class)

```
// position의 값이 "1"이면 사원, 그외이면 관리자
    if (position.equals("1")) { // 사원
        // Employee클래스의 객체emp를 생성
        emp = new Employee(empNo, name, part);
        result += emp.resultStr();
    } else { // 관리자
        // position의 값이 "2"이면 대리, 그외이면 과장으로 대체
        position = (position.equals("2")) ? "대리" : "과장";
        // Manager클래스의 객체 mng를 생성
        mng = new Manager(empNo, name, part, position);
        result += mng.resultStr();
    }

    // 결과 문자열을 콘솔에 출력
    System.out.println(result);
    sc.close();
}
}
```

## 4. 참조형 데이터의 대입

- ❖ 상위 클래스 타입으로 선언된 참조형 변수에는 하위 클래스 타입으로 만들어진 인스턴스의 참조를 대입할 수 있음
- ❖ 상위 클래스 타입에서 요구하는 것을 하위 클래스 타입에서는 모두 제공 가능하기 때문
- ❖ 하위 클래스 타입으로 선언된 참조형 변수에 상위 클래스 타입의 객체의 참조를 대입할 때는 강제형 변환을 통해서만 가능
- ❖ 원본의 타입이 하위 클래스 타입 인 경우에만 해야 하는데 그렇지 않으면 에러는 아니지만 실행 시 예외 발생
- ❖ 기본적으로 멤버에 접근할 때는 참조형 변수를 선언할 때의 멤버에만 접근이 가능하지만 오버라이딩된 메소드는 참조형 변수를 선언할 때 사용한 타입이 아니고 메모리를 할당할 때 사용한(생성자) 타입을 가지고 오버라이딩된 메소드를 호출
- ❖ 동일한 메시지에 대하여 다르게 반응하는 다형성(Polymorphism)을 구현할 목적으로 많이 사용

# 실습(convert-Company.java)

```
package chap07.convert;
import java.util.Scanner;
import overriding.Employee;
import overriding.Manager;

public class Company {
    public static void main(String args[]) {
        String result = ""; // 결과 문자열
        Employee emp = null; // Employee객체의 레퍼런스 변수

        String empNo = null; // 사원번호를 입력받는 변수
        String name = null; // 이름을 입력받는 변수
        String part = null; // 부서를 입력받는 변수
        String position = null; // 직책을 입력받는 변수
        Scanner sc = new Scanner(System.in);

        System.out.print("사번 입력(예)A1010 : ");
        empNo = sc.nextLine();
        System.out.print("이름 입력(예)박문석 :");
        name = sc.nextLine();
        System.out.print("부서 입력(예)기획 :");
        part = sc.nextLine();
        System.out.print("직책 입력(예)사원-1, 대리-2, 과장-3 :");
        position = sc.nextLine();
    }
}
```

# 실습(convert-Company.java)

```
// position의 값이 "1"이면 사원, 그외이면 관리자
    if (position.equals("1")) { // 사원
        // Employee클래스의 객체 emp를 생성
        emp = new Employee(empNo, name, part);
    } else { // 관리자
        // position의 값이 "2"이면 대리, 그외이면 과장으로 대치
        position = (position.equals("2")) ? "대리" : "과장";
        // Manager클래스의 객체 mng를 생성
        emp = new Manager(empNo, name, part, position);
    }
    result += emp.resultStr();
    // 결과 문자열을 콘솔에 출력
    System.out.println(result);
    sc.close();
}
}
```

## 5. Abstract Class

- ❖ 클래스 이름 앞에 abstract라는 예약어가 사용된 클래스
- ❖ 하위 클래스에 구현될 기능을 추상 메소드로 선언한 미완성 클래스
- ❖ 추상 메소드는 선언만 되어 있고 구현 부분이 없는 메소드로 메소드의 결과형 앞에 abstract를 기재해서 선언
- ❖ 추상 메소드는 추상 클래스나 인터페이스에만 존재해야 함
- ❖ 추상 메소드는 반드시 하위 클래스에서 오버라이딩 되어야 함
- ❖ 추상 클래스는 추상 메소드와 일반적인 멤버변수 그리고 메소드를 가질 수 있음
- ❖ 추상클래스는 자신의 객체(new를 이용한 생성)를 생성하지 못하며 상속을 통해서만 이용
- ❖ 추상클래스 타입으로 참조형 변수는 선언할 수 있음

# 실습(Starcraft.java)

```
abstract public class Starcraft {  
    abstract void attack();  
}
```



# 실습(Protoss.class)

```
public class Protoss extends Starcraft {  
  
    @Override  
    public void attack() {  
        System.out.println("프로토스의 공격");  
    }  
  
}
```



# 실습(Terran.class)

```
public class Terran extends Starcraft {  
  
    @Override  
    public void attack() {  
        System.out.println("테란의 공격");  
    }  
}
```

# 실습(Zerg.class)

```
public class Zerg extends Starcraft {  
  
    @Override  
    public void attack() {  
        System.out.println("저그의 공격");  
    }  
  
}
```



# 실습(StarcraftMain.class)

```
public class StarcraftMain {  
    public static void main(String[] args) {  
        Terran marine = new Terran();  
        marine.attack();  
        Protoss dragoon = new Protoss();  
        dragoon.attack();  
        Zerg hydralisk = new Zerg();  
        hydralisk.attack();  
        System.out.println("====객체 형변환과 오버라이딩을 이용====");  
        Starcraft obj = new Terran();  
        // 상위 클래스의 참조형 변수에 하위 클래스의 객체를 대입  
        obj.attack(); // shot() 메소드 호출  
        obj = new Protoss();  
        obj.attack(); // shot() 메소드 호출  
        obj = new Zerg();  
        obj.attack(); // shot() 메소드 호출  
        /*  
        //error 추상 클래스는 자신의 생성자를 이용해서 객체를 생성할 수 없음  
        obj = new Starcraft(); obj.shot();  
        */  
    }  
}
```

## 6. 인터페이스

### ❖ 인터페이스

- ✓ 다중 구현과 메시지 사용의 일관성을 위해서 나온 개념
- ✓ static final 변수와 추상 메소드와 default 메소드를 소유할 수 있음
- ✓ 인터페이스에 정의된 모든 메소드는 추상 메소드이므로 상속 받는 클래스에서는 전부 Overriding(재정의)
- ✓ 인터페이스로부터 클래스가 상속을 받을 때(구현)는 extends 키워드 대신에 implements 키워드를 사용
- ✓ JDK는 인터페이스 이름에 able을 추가하는 경우가 많음

### ❖ 인터페이스 정의

```
public interface 인터페이스이름 [extends 인터페이스이름, ...]
```

```
{
```

```
    상수 선언
```

```
    //static final이 붙지 않아도 무조건 static 상수가 됩니다.
```

```
    메소드 선언
```

```
    // 메소드 앞에 abstract 가 나오지 않아도 메소드는 무조건 추상메소드 입니다.
```

```
}
```

## 6. 인터페이스

### ❖ 인터페이스 선언 - 예

```
public interface Inter
{
    public int x = 10;
    public int y = 100;
    public void wake();
}
```

### ❖ 인터페이스 구현

- ✓ 클래스 생성 시 인터페이스를 구현하기 위해서는 implements 예약어를 사용
- ✓ 여러 개의 인터페이스로부터 상속을 받을 때는 ,로 구분해서 인터페이스 이름을 나열
- ✓ 클래스로부터 상속받고 인터페이스를 구현할 때는 상속받는 문장이 먼저 나와야 함

```
[public/final/abstract] class 클래스이름 extends 상위클래스이름
implements 인터페이스이름 [,인터페이스이름]
{
    멤버 변수;
    멤버 메소드;
    인터페이스에 속한 메소드 { }
}
```

# 실습(Inter.java)

```
public interface Inter {  
    public int a=100;  
    public final int b = 100;  
    public abstract void method1();  
    public void method2();  
}
```



# 실습(InterImpl.java)

```
public class InterImpl implements Inter
{
    public void method1()
    {
        System.out.println("abstract가 있으면 당연히 추상메소드");
    }

    public void method2()
    {
        System.out.println("abstract가 없어도 추상메소드");
    }
}
```

# 실습(InterMain1.java)

```
public class InterfaceMain1 {  
    public static void main(String args[])  
    {  
        InterImpl obj = new InterImpl();  
        //obj.a = 200;  
        obj.method1();  
        System.out.println("인터페이스에서는 final을 사용하지 않아도 변수는 final");  
        System.out.println("인터페이스에서는 abstract을 사용하지 않아도 메소드는 abstract");  
    }  
}
```



# 실습(Inter1.java)

```
public interface Inter1 {  
    public int a=100;  
    public abstract void method1();  
}
```



# 실습(Inter2.java)

```
public interface Inter2 {  
    public int b=100;  
    public abstract void method2();  
}
```



## 실습(MultiImpl.java)

```
public class MultiImpl implements Inter1, Inter2 {  
    public void method1() {  
        System.out.println("Inter1의 메소드");  
    }  
  
    public void method2() {  
        System.out.println("Inter2의 메소드");  
    }  
}
```

## 실습(InterfaceMain2.java)

```
public class InterfaceMain2 {  
    public static void main(String args[])  
    {  
        Multimpl obj = new Multimpl();  
        System.out.println("Inter1의 a=" + Multimpl.a);  
        System.out.println("Inter2의 b=" + Multimpl.b);  
        obj.method1();  
        obj.method2();  
    }  
}
```

## 6. 인터페이스

### ❖ default method

- ✓ 인터페이스에 만들 수 있는 내용이 있는 메소드로 상속받은 클래스에서 바로 사용이 가능
- ✓ 접근 지정자로 default를 사용
- ✓ default method는 추상 메소드가 아니므로 하위 클래스에서 재정의 하지 않아도 됨
- ✓ 하위 클래스에서 재정의하지 않으면 원래의 기능 그대로 이용이 가능하고 재정의하면 재정의한 기능을 이용
- ✓ 인터페이스에 메소드를 추가해야 하는 경우에 추상 메소드를 추가하게 되면 상속받은 하위 클래스에서 재정의 해야 하기 때문에 기존의 코드들이 에러가 발생
- ✓ default method를 추가하게 되면 에러가 발생하지 않음
- ✓ 하나의 인터페이스로부터 상속받은 클래스가 여러 개 일 때 동일한 기능을 추가하고자 할 때 도 전부 재정의하지 않고 인터페이스에 default method 만 추가하면 되기 때문에 편리하게 사용



## 실습(MyInterface.java)

```
public interface MyInterface {  
    //추상 메소드 이므로 하위 클래스에서 재정의 해야 합니다.  
    public void method1();  
    //추상 메소드가 아니므로 하위 클래스에서 재정의하지 않아도 됩니다.  
    public default void method2(){  
        System.out.println("default method");  
    }  
}
```

## 실습(MyClass.java)

```
public class MyClass implements MyInterface {  
  
    @Override  
    public void method1() {  
        System.out.println("메소드 재정의");  
    }  
}
```



## 실습(DefaultMain.java)

```
public class DefaultMain {  
    public static void main(String[] args) {  
        MyClass obj = new MyClass();  
        obj.method2();  
    }  
}
```






## 6. 인터페이스

- ❖ 인터페이스도 상속(확장) 가능
- ❖ 인터페이스를 인터페이스에 상속할 때는 extends 키워드 사용

```
public interface 인터페이스이름  
    extends 인터페이스이름[, 인터페이스 이름,...] {  
    상수 선언  
    메소드 선언  
}
```

- ❖ 인터페이스의 상속 - 예

```
public interface Sleeper {  
    public long ONE_SECOND = 1000;  
    public long ONE_MINUTE = 60000;  
    public void wakeup();  
}  
  
public interface People extends Sleeper{  
    public int MAX = 24;  
    public int MIN = 0;  
    public void work();  
}
```



## 실습(C1.java)

```
public interface C1 {  
    void method1();  
    void method2();  
}
```



## 실습(C2.java)

```
public interface C2 {  
    void method3();  
}
```



## 실습(C3.java)

```
public interface C3 extends C1, C2 {  
    void method4();  
}
```



# 실습(C.java)

```
public class C implements C3 {  
    @Override  
    public void method1() {  
        System.out.println("method1() 메소드의 구현");  
    }  
  
    @Override  
    public void method2() {  
        System.out.println("method2() 메소드의 구현");  
    }  
  
    @Override  
    public void method3() {  
        System.out.println("method3() 메소드의 구현");  
    }  
  
    @Override  
    public void method4() {  
        System.out.println("method4() 메소드의 구현");  
    }  
}
```

## 실습(InterfaceMain3.java)

```
public class InterfaceMain3 {  
    public static void main(String arg[]) {  
        C obj = new C();  
        obj.method1();  
        obj.method2();  
        obj.method3();  
        obj.method4();  
    }  
}
```

## 6. 인터페이스

- ❖ 인터페이스 자료형으로 참조형 변수를 선언할 수 있고 사용은 클래스를 자료형으로 선언한 변수와 동일하게 사용할 수 있음
- ❖ 인터페이스 타입의 객체 참조 변수에 인터페이스를 구현한 클래스의 객체를 할당하는 것이 가능
- ❖ 클래스의 참조형 변수와 마찬가지로 인터페이스 타입의 참조형 변수를 이용해서는 인터페이스에 선언된 속성과 메소드만 접근 가능



# 실습(Starcraftable.java)

```
public interface Starcraftable {  
    abstract void attack();  
}
```





# 실습(ProtossImpl.java)

```
public class ProtossImpl implements Starcraftable {  
  
    @Override  
    public void attack() {  
        System.out.println("프로토스의 공격");  
    }  
}
```



# 실습(TerranImpl.java)

```
public class TerranImpl implements Starcraftable {  
  
    @Override  
    public void attack() {  
        System.out.println("테란의 공격");  
    }  
}
```



# 실습(ZergImpl.java)

```
public class ZergImpl implements Starcraftable {  
  
    @Override  
    public void attack() {  
        System.out.println("저그의 공격");  
    }  
  
}
```



# 실습(StarcraftableMain.java)

```
public class StarcraftAbleMain {  
    public static void main(String[] args) {  
        TerranImpl marine = new TerranImpl();  
        marine.attack();  
        ProtossImpl dragoon = new ProtossImpl();  
        dragoon.attack();  
        ZergImpl hydralisk = new ZergImpl();  
        hydralisk.attack();  
        System.out.println("====객체 형변환과 오버라이딩을 이용====");  
        Starcraft obj = new Terran(); // 상위 클래스의 객체에 하위 클래스의 객체를 형변환  
        obj.attack();  
        obj = new Protoss();  
        obj.attack();  
        obj = new Zerg();  
        obj.attack();  
    }  
}
```

## 7. final

- ❖ final 클래스: 상속할 수 없는 클래스
- ❖ final 클래스를 제외하고도 생성자 메소드의 접근 지정자가 private으로 되어 있는 경우도 상속이 불가능한데 싱글톤 클래스나 static 멤버 만을 가진 클래스의 경우 생성자를 이용해서 객체를 생성할 수 없도록 하기 위해서 생성자를 private으로 만들어두는 경우가 있음
- ❖ final 메소드: 재정의(Overriding) 할 수 없는 메소드
- ❖ final 필드: 값을 변경할 수 없는 필드

## 8. 클래스 사이의 관계

- ❖ has a 관계: 객체가 자신의 클래스 타입이 아닌 객체를 포함하는 구조(포함 관계)
  - ✓ Aggregation Relation: 포함하는 객체와 포함되는 객체의 수명이 다른 경우 - 다른 클래스의 객체가 소유한 메소드를 호출해서 사용
  - ✓ Composition Relation: 포함하는 객체와 포함되는 객체의 수명이 같은 경우 - 다른 클래스의 객체들을 이용해서 구성된 경우
- ❖ is a 관계: 상속 관계

