**다형성 구현**

2017-09-13

이승진

**학습목표**

다형성의 사용 목적과 구현 방법

interface 문법

abstract class & abstract method 사용 목적과 구현 방법

**목차**

[1. 배경지식 2](#_Toc462417212)

[1) 다형성 2](#_Toc462417213)

[2) interface 4](#_Toc462417214)

[3) interface의 상속 5](#_Toc462417215)

[4) 메소드 재정의 6](#_Toc462417216)

[5) @Override 어노테이션 6](#_Toc462417217)

[6) abstract method & abstract class 7](#_Toc462417218)

[2. 예제코드 8](#_Toc462417219)

[1) 다형성 호출 예제 #1 8](#_Toc462417220)

[2) 다형성 호출 예제 #2 9](#_Toc462417221)

[3) abstract 클래스 예제 #1 10](#_Toc462417222)

[4) abstract 클래스 예제 #2 11](#_Toc462417223)

[5) abstract 클래스 예제 #3 12](#_Toc462417224)

[6) interface 예제 코드 #1 13](#_Toc462417225)

[7) interface 예제 코드 #2 14](#_Toc462417226)

# 배경지식

## 다형성

### 다형성의 목적

사용 목적이 유사한 클래스들의 사용 방법이 통일되어 있다면 편리할 것이다.

예를 들어 Java에서 객체 목록을 저장할 때, ArrayList 클래스나 LinkedList 클래스를 사용한다.

객체 목록을 0,1,2,3... 인덱스 순서대로 저장하고 싶을 때 이 클래스들을 사용한다.

이 클래스들은 배열과 사용 방법이 비슷하면서 배열보다 편리한 기능을 제공한다.

ArrayList 클래스와 LinkedList 클래스의 내부 구현은 서로 다르지만, 이들의 사용 목적은 같다.

사용 목적이 같으므로 사용 방법도 같으면 편리할 것이다.

ArrayList 클래스와 LinkedList 클래스에는 다형성이 구현되어 있어서 사용 방법이 같다.

즉 ArrayList 클래스와 LinkedList 클래스는 서로 호환된다.

그래서 ArrayList 클래스를 사용하는 소스 코드를 LinkedList 클래스를 사용하도록 수정할 때,

수정해야 하는 소스 코드가 거의 없다.

사용 목적이 유사한 클래스들의 사용 방법을 통일하여 서로 호환되도록 만드는 것이 다형성의 목적이다.

### 다형성의 활용

예를 들어 프린터기를 사용하여 문서를 인쇄하는 작업에 다형성이 구현되어 있다.

아래한글로 작성한 문서를 인쇄하는 경우에, 그 컴퓨터에 잉크젯 프린터가 연결되어 있다면, 잉크젯 프린터로 인쇄될 것이고, 그 컴퓨터에 레이저 프린터가 연결되어 있다면, 레이저 프린터로 인쇄될 것이다.

잉크젯 프린터의 내부 구조와 레이저 프린터의 내부 구조는 매우 다르다.

아래한글은 현재 프린터가 잉크젯 프린터인지 레이저 프린터인지 알지 못한다.

아래한글에서 프린터에 전달하는 인쇄 명령은 프린터에 무관하다.

프린터가 잉크젯이든 레이저든 아래한글은 동일한 인쇄 명령을 전달한다.

인쇄 명령은 동일한데, 잉크젯 프린터가 그 명령을 받은 경우에는 잉크젯 프린터에 인쇄되고, 레이저 프린터가 받은 경우에는 레이저 프린터에 인쇄된다.

프린터를 교체해도 아래한글은 수정할 필요가 없다.

즉 작업 명령을 지시하는 쪽에서는 동일한 명령으로 작업을 지시했는데, 그 명령을 받아서 실행하는 부품에 따라 다른 방식으로 작업이 실행되는 것이 다형성이다.

### 다형성 구현

다형성을 구현하려면 다음 사항들을 구현해야 한다.

● 다형성을 구현하려는 클래스들이 공통 부모 클래스를 상속(extends) 받거나, 아니면 공통 interface를 구현(implements) 해야 한다.



● 공통 부모 클래스를 상속(extends) 받은 자식 클래스들은 다형성을 구현하려는 부모 클래스의 메소드들을 재정의(override) 해야 한다.



● 공통 interface를 구현(implements) 자식 클래스들은 interface의 모든 메소드를 구현해야 한다.

● 공통 부모 클래스 타입(type)의 변수나, 공통 interface 타입의 변수를 사용하여 메소드를 호출해야 다형성 호출이다. 자식 클래스 타입의 변수를 사용하여 메소드를 호출하면 언제나 그 자식 클래스의 메소드가 호출될 것이기 때문에 다형성 호출이 아니다.

|  |
| --- |
| ChildOne obj1 = new ChildOne();  ChildTwo obj2 = new ChildTwo();  Parent p = obj1;  p.method1(); // 다형성 호출. ChildOne 클래스의 method1 메소드가 호출된다.  p = obj2;  p.method1(); // 다형성 호출. ChildTwo 클래스의 method1 메소드가 호출된다.  obj1.method1(); // 다형성 호출이 아님.  obj2.method1(); // 다형성 호출이 아님. |

## interface

Java 언어의 interface는 메소드 구현 없이 메소드 선언만 포함해야 한다. 그리고 멤버 변수도 없어야 한다.

서로 호환되는 클래스들의 규격을 정의하기 위해 Java interface를 사용한다.

예를 들어 Java의 ArrayList 클래스와 LinkedList 클래스는 사용 방법이 같다. 즉 public method 목록이 같다.

그래서 ArrayList 클래스를 사용하는 소스코드를, ArrayList 대신 LinkedList 클래스를 사용하도록 수정할 때, 별로 수정할 부분이 없다. 즉 ArrayList와 LinkedList는 서로 호환된다.

ArrayList와 LinkedList가 서로 호환되는 이유는, 둘 다 List interface를 구현(implements)했기 때문이다.

List interface를 구현했다는 말은, List interface에 선언된 메소드들을 전부 구현했다는 말이다.

그렇기 때문에 ArrayList와 LinkedList의 public method 목록이 같고, 사용 방법도 같고, 그래서 서로 호환된다.

List interface의 존재 이유는, List interface를 구현한 ArrayList와 LinkedList 클래스들이, 사용 방법이 같고, 그래서 서로 호환이 되도록 하는데 있다.

사용 방법이 같고 서로 호환이 되는 클래스들을 여러 개 만들어야 할 때, interface를 활용한다.

① 미리 그 사용 방법(public method 목록)을 interface로 정의한다.

② 그 interface에 정의된 public method를 구현(implements)하는 형태로 클래스들을 구현한다.

interface를 구현(implements)하는 클래스는 그 interface의 모든 메소드를 구현해야 한다.

interface를 구현한 클래스가 abstract 클래스이라면, interfac의 모든 메소드를 구현하지 않아도 된다.



ArrayList 클래스와 LinkedList 클래스에는 다형성이 구현되어 있다.

### public abstract 메소드

인터페이스의 메소드는 모두 public abstact 메소드이다.

public abstract 키워드를 메소드 앞에 붙여주어도 되고 생략해도 된다.

생략하더라도 인터페이스의 메소드는 모두 public abstact 메소드이다.

### public static final 멤버 변수

인터페이스의 멤버 변수는 모두 public static final 멤버 변수이다.

public static final 키워드를 멤버 변수 앞에 붙여주어도 되고 생략해도 된다.

생략하더라도 인터페이스의 멤버 변수는 모두 public static final 멤버 변수이다.

-

## interface의 상속

interface들 끼리도 상속 관계를 가질 수 있다.

자식 interface는 부모 interface의 모든 메소드를 상속 받는다.

예를 들어 List interface는 Collection interface를 상속 받는다.

자식 interface를 implements한 클래스는

자식 interface의 메소드 뿐만 아니라 부모 interface의 메소드도 전부 구현해야 한다.



예를 들어 ArrayList 클래스는 List interface에 정의된 메소드들 뿐만 아니라, Collection interface에 정의된 메소드들도 전부 구현했다.

ArrayList 클래스와 LinkedList 클래스는 둘 다 List interface를 구현했기 때문에, List interface의 메소드들을 기준으로 호환된다.

java 표준 라이브러리의 ArrayDeque 클래스는 List interface를 구현하지 않았고, Deque interface를 구현했다. Deque interface도 Collection interface를 상속 받는다. 따라서 ArrayDeque 클래스도 Collection interface에 정의된 메소드들을 전부 구현했다.

ArrayList 클래스와 ArrayDeque 클래스는 둘 다 Collection interface를 구현했기 때문에, Collection interface 메소드들을 기준으로 호환된다.

List interface는 Collection interface의 모든 메소드를 상속 받기 때문에, List interface를 기준으로 호환되는 것은 Collection interface를 기준으로 호환되는 것 보다 좀 더 많은 메소드들이 호환된다.



사람과 원숭이는 영장목에 속한다. 그래서 영장목에 정의된 속성을 사람과 원숭이 모두 가지고 있다.

영장목은 포유류에 속한다. 그래서 포유류에 정의된 속성을 사람, 원숭이 모두 가지고 있다.

얼룩말은 말목에 속한다. 말목도 포유류에 속한다. 그래서 얼룩말도 포유류에 정의된 속성을 모두 가지고 있다.

### 인터페이스 상속의 예

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | interface A {  void method1();  }  interface B extends A {  void method2();  }  class S1 implements A {  public void method1() { System.out.println("S1 method1()"); }  }  class S2 implements B {  public void method1() { System.out.println("S2 method1()"); }  public void method2() { System.out.println("S2 method2()"); }  } |

interface A에는 method1() 메소드가 선언되어 있다.

그래서 interface A를 implements한 S1 클래스는 method1() 메소드를 재정의 해야 한다.

interface B는 interface A를 상속 받는다.

즉 interface B는 interface A의 method1()을 상속 받는다.

그래서 interface B를 implements한 S2 클래스는 method1()과 method2() 메소드를 재정의 해야 한다.

## 메소드 재정의

부모 클래스 메소드를 자식 클래스에서 다시 구현하는 것을 메소드 재정의(override)라고 부른다.

부모 클래스의 메소드를 재정의하려면, 다음 규칙을 지켜야 한다.

\* 메소드명이 정확하게 일치해야 한다.

\* 파라미터의 수가 정확하게 일치해야 한다.

\* 파라미터의 타입이 정확하게 일치해야 한다.

\* 리턴 타입이 일치해야 한다.

\* 자식 메소드의 리턴 타입이 부모 메소드의 리턴 타입의 자식 클래스이어도 된다.

\* 파라미터 변수 명은 일치하지 않아도 된다.

## @Override 어노테이션

재정의 메소드에 @Override 어노테이션을 붙이면,

부모 클래스의 메소드를 정확하게 재정의(override)했는지 컴파일러가 검사해준다.

혹시 실수로 메소드 이름을 다르게 입력했다면, 컴파일러 에러가 발생한다.

메소드를 재정의할 때 반드시 @Override 어노테이션을 붙여야 하는 것은 아니고, 생략해도 된다.

가급적 붙여주는 것이 좋다.

## abstract method & abstract class

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | abstract class Parent { // abstract class  public void method1() {  System.out.println("Parent method1()");  }    public abstract void method2(); // abstract method  } |

method2()는 메소드 본문이 없다. 이런 메소드를 abstract method 라고 부른다.

abstract 메소드가 아닌 메소드를 concrete 메소드라고 부른다.

abstract 메소드가 한 개라도 들어있으면 그 클래스는 abstract 클래스이다.

abstract 클래스가 아닌 클래스를 concreate 클래스라고 부른다.

abstract 클래스의 인스턴스 객체를 생성할 수 없다.

오직 concrete 클래스의 인스턴스 객체만 생성할 수 있다.

abstract 메소드, abstract 클래스 앞에 abstract 키워드를 붙여주지 않으면 컴파일 에러가 발생한다.

abstract 클래스를 상속한 자식 클래스가 conrete 클래스라면, conrete 자식 클래스는 상속 받은 abstract 메소드를 전부 재정의 해야 한다. 그렇지 않으면 컴파일 에러가 발생한다.

abstract 클래스를 상속한 자식 클래스가 abstract 클래스라면, abstract 자식 클래스는 상속 받은 abstract 메소드를 전부 재정의 하지 않아도 되고 해도 된다.

# 예제코드

## 다형성 호출 예제 #1

Polymorphism1.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | package net.skhu.lecture05.polymorphism1;  class Parent {  public void method1() {  System.out.println("Parent method1()");  }  }  class ChildOne extends Parent {  @Override  public void method1() {  System.out.println("ChildOne method1()");  }  }  class ChildTwo extends Parent {  @Override  public void method1() {  System.out.println("ChildTwo method1()");  }  }  class ChildThree extends Parent {  }  public class Polymorphism1 {    public static void main(String[] args) {  Parent parent;  ChildOne obj1 = new ChildOne();  ChildTwo obj2 = new ChildTwo();  ChildThree obj3 = new ChildThree();    obj1.method1(); // 다형성 호출이 아님    parent = obj1;  parent.method1(); // 다형성 호출    parent = obj2;  parent.method1(); // 다형성 호출    parent = obj3;  parent.method1(); // 다형성 호출  }  } |

(줄34) 다형성 호출이 아닌 이유는 무엇인가?

위 코드의 method1 메소드 호출은 각각 어떤 메소드를 호출하나?

## 다형성 호출 예제 #2

Polymorphism2.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | package net.skhu.lecture05.polymorphism2;  class Parent {  public void method1() {  System.out.println("Parent method1()");  }  }  class ChildOne extends Parent {  @Override  public void method1() {  System.out.println("ChildOne method1()");  }  }  class ChildTwo extends Parent {  @Override  public void method1() {  System.out.println("ChildTwo method1()");  }  }  class ChildThree extends Parent {  }  public class Polymorphism2 {    static void doSomething(Parent parent) {  parent.method1(); // 다형성 호출  }    public static void main(String[] args) {  ChildOne obj1 = new ChildOne();  ChildTwo obj2 = new ChildTwo();  ChildThree obj3 = new ChildThree();    obj1.method1(); // 다형성 호출이 아님  doSomething(obj1);  doSomething(obj2);  doSomething(obj3);  }  } |

(줄37) 다형성 호출이 아닌 이유는 무엇인가?

위 코드의 실행 결과 출력은 무엇인가?

## abstract 클래스 예제 #1

Abstract1.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59 | package net.skhu.lecture05.abstract1;  abstract class Parent { // abstract class  public void method1() {  System.out.println("Parent method1()");  }    public abstract void method2(); // abstract method  }  class ChildOne extends Parent {  @Override  public void method1() {  System.out.println("ChildOne method1()");  }  @Override  public void method2() { // abstract method override는 필수  System.out.println("ChildOne method2()");  }  }  class ChildTwo extends Parent {  @Override  public void method1() {  System.out.println("ChildTwo method1()");  }  @Override  public void method2() { // abstract method override는 필수  System.out.println("ChildTwo method2()");  }  }  class ChildThree extends Parent {  @Override  public void method2() { // abstract method override는 필수  System.out.println("ChildThree method2()");  }  }  public class Abstract1 {    static void doSomething(Parent parent) {  parent.method1(); // 다형성 호출  parent.method2(); // 다형성 호출  }    public static void main(String[] args) {  ChildOne obj1 = new ChildOne();  ChildTwo obj2 = new ChildTwo();  ChildThree obj3 = new ChildThree();    doSomething (obj1);  doSomething (obj2);  doSomething (obj3);  }  } |

위 코드의 실행 결과 출력은 무엇인가?

## abstract 클래스 예제 #2

Abstract2.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | package net.skhu.lecture05.abstract2;  abstract class Parent {  public static final double PI = 3.141592;    public abstract void method1();  public abstract void method2();  }  class ChildOne extends Parent {  @Override  public void method1() {  System.out.println("ChildOne method1()");  }  @Override  public void method2() {  System.out.println("ChildOne method2()");  }  }  class ChildTwo extends Parent {  @Override  public void method1() {  System.out.println("ChildTwo method1()");  }  @Override  public void method2() {  System.out.println("ChildTwo method2()");  }  }  public class Abstract2 {    static void doSomething(Parent parent) {  parent.method1();  parent.method2();  }    public static void main(String[] args) {  ChildOne obj1 = new ChildOne();  ChildTwo obj2 = new ChildTwo();  doSomething(obj1);  doSomething(obj2);  }  } |

실행 결과 출력은 무엇인가?

## abstract 클래스 예제 #3

Abstract3.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | package net.skhu.lecture05.abstract3;  abstract class Parent {  public static final double PI = 3.141592;    public abstract void method1();  public abstract void method2();  }  abstract class Child extends Parent {  @Override  public void method1() {  System.out.println("Child method1()");  }  }  class GrandChild extends Child {  @Override  public void method2() {  System.out.println("GrandChild method2()");  }  }  public class Abstract3 {    static void doSomething(Parent parent) {  parent.method1();  parent.method2();  }    public static void main(String[] args) {  doSomething(new GrandChild());  }  } |

Child 클래스는 method2 메소드를 재정의하지 않았기 때문에 abstract 클래스이어야 한다.

method1은 이미 Child 클래스에서 재정의하였으므로, GrandChild 클래스에서는 method2만 재정의하면 된다.

## interface 예제 코드 #1

Interface1.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | package net.skhu.lecture05.interface1;  interface Parent {  double PI = 3.141592;    void method1();  void method2();  }  class ChildOne implements Parent {  @Override  public void method1() {  System.out.println("ChildOne method1()");  }  @Override  public void method2() {  System.out.println("ChildOne method2()");  }  }  class ChildTwo implements Parent {  @Override  public void method1() {  System.out.println("ChildTwo method1()");  }  @Override  public void method2() {  System.out.println("ChildTwo method2()");  }  }  public class Interface1 {  static void doSomething(Parent parent) {  parent.method1();  parent.method2();  }  public static void main(String[] args) {  ChildOne obj1 = new ChildOne();  ChildTwo obj2 = new ChildTwo();  doSomething(obj1);  doSomething(obj2);  }  } |

abstract 클래스 예제 #2 (Abstract2.java)와 비교하여 살펴보자.

(줄4) public static final 키워드를 생략해도 되고 붙여도 된다. 생략해도 public static final 이다.

(줄6) public abstract 키워드를 생략해도 된고 붙여도 된다. 생략해도 public abstract 이다.

## interface 예제 코드 #2

Interface2.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50 | package net.skhu.lecture05.interface2;  interface A {  void method1();  }  interface B extends A {  void method2();  }  class S1 implements A {  @Override  public void method1() {  System.out.println("S1 method1()");  }  }  class S2 implements B {  @Override  public void method1() {  System.out.println("S2 method1()");  }  @Override  public void method2() {  System.out.println("S2 method2()");  }  }  public class Interface2 {  static void print1(A obj) {  obj.method1();  }  static void print2(B obj) {  obj.method1();  obj.method2();  }  public static void main(String[] args) {  S1 o1 = new S1();  S2 o2 = new S2();  print1(o1);  print1(o2);  print2(o2);  }  } |

인터페이스 상속을 구현한 예제 코드이다.

인터페이스 B를 구현한 concrete 클래스에서 반드시 method1() 메소드도 재정의 해야 하는 이유는 무엇인가?

## 호환되는 부품 예제 코드

Polymorphism3.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | package net.skhu.lecture05;  import java.util.ArrayList;  import java.util.Iterator;  import java.util.LinkedList;  import java.util.List;  public class Polymorphism3 {    static void addIntegersToList(List<Integer> list, int count) {  for (int i=0; i < count; ++i)  list.add(i);  }    static void removeOddNumbers(List<Integer> list) {  Iterator<Integer> iterator = list.iterator();  while (iterator.hasNext()) {  int i = iterator.next();  if (i % 2 == 1) iterator.remove();  }  }    static void printList(List<Integer> list) {  for (int i : list)  System.out.printf("%d ", i);  System.out.println();  }    static void doSomething(List<Integer> list) {  addIntegersToList(list, 20);  removeOddNumbers(list);  printList(list);  }    public static void main(String[] args) {  doSomething(new ArrayList<Integer>());  doSomething(new LinkedList<Integer>());  }  } |

## 탐색 & 삭제

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | package net.skhu.iterator;  import java.util.ArrayList;  import java.util.Iterator;  import java.util.List;  public class Exampl01 {  static void addIntegersToList(List<Integer> list, int count) {  for (int i = 0; i < count; ++i)  list.add(i);  }  static void remove1(List<Integer> list) {  Iterator<Integer> iterator = list.iterator();  while (iterator.hasNext()) {  int value = iterator.next();  if (3 <= value && value <= 6)  iterator.remove();  }  }  static void remove2(List<Integer> list) {  for (int i = 0; i < list.size(); ++i) {  int value = list.get(i);  if (3 <= value && value <= 6)  list.remove(i);  }  }  public static void main(String[] args) {  List<Integer> list1 = new ArrayList<Integer>();  addIntegersToList(list1, 10);  remove1(list1);  System.out.println(list1.toString());  List<Integer> list2 = new ArrayList<Integer>();  addIntegersToList(list2, 10);  remove2(list2);  System.out.println(list2.toString());  }  } |

실행 결과 출력

|  |
| --- |
| [0, 1, 2, 7, 8, 9]  [0, 1, 2, 4, 6, 7, 8, 9] |

remove2 메소드는 3~6 값들을 삭제하지 못했고, 3과 5 값만 삭제되었다. 이유가 무엇일까?