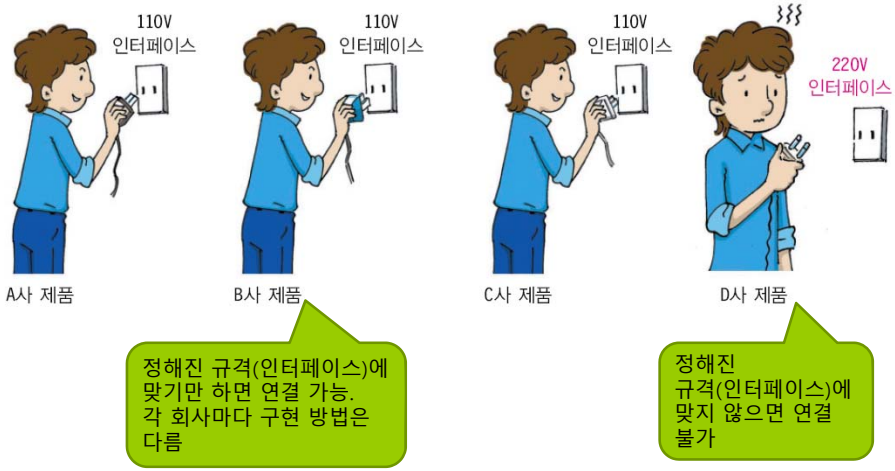
인터페이스, 람다식

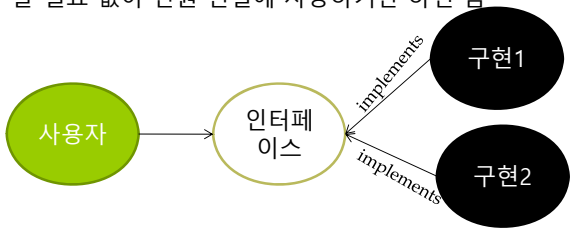
514760-1
2017년 가을학기
10/2/2017
박경신

실세계의 인터페이스와 인터페이스의 필요성



인터페이스의 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 아울렛처럼 규격에 맞기만 하면, 어떻게 만들어졌는지 알 필요 없이 전원 연결에 사용하기만 하면 됨



자바의 인터페이스

- 인터페이스(interface)
  - 모든 메소드가 추상 메소드인 클래스
- 인터페이스 선언
  - **interface** 키워드로 선언
  - ex) public **interface** SerialDriver {...}
- 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

인터페이스 선언

- 인터페이스 구성멤버
  - 상수필드 (constant field)
  - 추상메소드 (abstract method)
  - 디폴트메소드 (default method)
  - 정적메소드 (static method)

```
public interface 인터페이스명 {
    //상수(constant fields)
    타입 상수명 = 값;

    //추상 메소드(abstract method)
    리턴타입 메소드명(매개변수,..);

    //디폴트 메소드(default method)
    default 리턴타입 메소드명(매개변수,..) {
        ..내부구현..
    }

    //정적 메소드(static method)
    static 리턴타입 메소드명(매개변수,..) {
        ..내부구현..
    }
}
```

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

```
interface Days {
    public static final int SUNDAY = 1, MONDAY = 2, TUESDAY = 3,
    WEDNESDAY = 4, THURSDAY = 5, FRIDAY = 6,
    SATURDAY = 7;
}
```

인터페이스 선언

- **추상 메소드(abstract method)** 선언
  - 인터페이스 통해 호출된 메소드는 최종적으로 객체에서 실행
  - 인터페이스의 메소드는 기본적으로 실행 블록이 없는 추상 메소드로 선언
  - **public abstract**를 생략하더라도 자동적으로 컴파일 과정에서 붙게 됨

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

인터페이스 선언

- **디폴트 메소드(default method)** 선언
  - 자바8에서 추가된 인터페이스의 새로운 멤버
  - 인스턴스 메소드와 동일하게 실행 블록을 가지고 있는 메소드
  - **default** 키워드를 반드시 붙여야 함
  - 기본적으로 public 접근 제한
    - 생략하더라도 컴파일 과정에서 자동 붙음

```
interface MyInterface {
  public void myMethod1();
  default void myMethod2() {
        System.out.println("myMethod2()");
  }
}
```

인터페이스 선언

- **정적 메소드(static method)** 선언
  - 자바8에서 추가된 인터페이스의 새로운 멤버
  - static 키워드를 반드시 붙여야 함

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

```
interface MyInterface {
  static void print(String msg) {
        System.out.println(msg + ": 인터페이스의 정적 메소드 호출");
  }
}
public class StaticMethodTest {
  public static void main(String[] args) {
  MyInterface.print("Java 8");
  }
}
```

인터페이스 선언 사례

```
public interface RemoteControl {
    int MAX_VOLUME = 10; //상수(constant fields)
    int MIN_VOLUME = 0; //상수(constant fields)

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

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

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

인터페이스 상속

- 인터페이스 간에도 상속 가능
  - 인터페이스를 상속하여 확장된 인터페이스 작성 가능
- 인터페이스 다중 상속 허용

```
interface MobilePhone {
    boolean sendCall();
    boolean receiveCall();
    boolean sendSMS();
    boolean receiveSMS();
}
interface MP3 {
    void play();
    void stop();
}
interface MusicPhone extends MobilePhone, MP3 {
    void playMP3RingTone();
}
```

```
public class MyPhone implements MusicPhone {
    public boolean sendCall() { ……; }
    public boolean receiveCall() { ……; }
    public boolean sendSMS() { ……; }
    public boolean receiveSMS() { ……; }
    public void play() { ……; }
    public void stop() { ……; }
    void playMP3RingTone() { ……; }
}
```

인터페이스 구현

- 구현 클래스 정의
  - 자신의 객체가 인터페이스 타입으로 사용할 수 있음
  - **implements** 키워드 사용
  - 여러 개의 인터페이스 동시 구현 가능
  - 상속과 구현이 동시에 가능

```
public class 클래스명 implements 인터페이스명 {
    //인터페이스에 선언된 추상 메소드의 실체 메소드 구현

}
```

- 추상 메소드의 실체 메소드를 작성하는 방법
  - 메소드의 선언부가 정확히 일치해야 함
  - 인터페이스의 모든 추상 메소드를 재정의하는 실체 메소드를 작성해야 함
    - 일부 추상메소드만 재정의할 경우, 구현 클래스는 추상 클래스

인터페이스 구현 예제

```
public class Television implements RemoteControl {
    private int volume; //필드
    //turnOn() 추상 메소드의 구현
    public void turnOn() {
        System.out.println( " TV를 켭니다. " );
    }
    //turnOff() 추상 메소드의 구현
    public void turnOff() {
        System.out.println( " TV를 끕니다. " );
    }
    //setVolume() 추상 메소드의 구현
    public void setVolume(int volume) {
        if(volume>RemoteControl.MAX_VOLUME) {
            this.volume = RemoteControl.MAX_VOLUME;
        } else if(volume<RemoteControl.MIN_VOLUME) {
            this.volume = RemoteControl.MIN_VOLUME;
        } else {
            this.volume = volume;
            }
        System.out.println("현재 TV 볼륨: " + volume);
    }
}
```

```
public class RemoteControlExample {
    public static void main(String[] args) {
        RemoteControl rc;
        rc = new Television();      // 구현 객체
        rc = new Audio();           // 구현 객체
    }
}
```

인터페이스 구현 및 사용 예제

```
public class Audio implements RemoteControl {
    private boolean mute; //필드
    @Override // 필요시 디폴트메소드 재정의
    public void setMute(boolean mute) {
        this.mute = mute;
        if(mute) {
            System.out.println("Audio 무음 처리합니다.");
        } else {
            System.out.println("Audio 무음 해제합니다.");
        }
    }
}
```

```
public class RemoteControlExample {
    public static void main(String[] args) {
        RemoteControl rc = null;
        rc = new Television();
        rc.turnOn(); // use abstract method
        rc.setMute(true); // use default method
        rc = new Audio();
        rc.turnOn(); // use abstract method
        rc.setMute(true); // use default method
        RemoteControl.changeBattery(); // use static method
    }
}
```

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

```
public class RemoteControlExample {
    public static void main(String[] args) {
        RemoteControl rc = new RemoteControl() { // anonymous class
            public void turnOn() { /*실행문*/ }
            public void turnOff() { /*실행문*/ }
            public void setVolume(int volume) { /*실행문*/ }
        };
    }
}
```

인터페이스의 다중 구현

- **다중 인터페이스(multi-interface) 구현 클래스**
  - 구현 클래스는 다수의 인터페이스를 모두 구현
  - 객체는 다수의 인터페이스 타입으로 사용

```
public class 클래스명 implements 인터페이스명A, 인터페이스명B {
    //인터페이스A에 선언된 추상 메소드의 실체 메소드 구현
    //인터페이스B에 선언된 추상 메소드의 실체 메소드 구현
}
```

인터페이스의 다중 구현

```
interface USBMouseInterface {
  void mouseMove();
  void mouseClick();
}
interface RollMouseInterface {
  void roll();
}
public class MouseDriver implements RollMouseInterface, USBMouseInterface
{
  public void mouseMove() { .... }
  public void mouseClick() { ... }
  public void roll() { ... }

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

인터페이스 사용

- 인터페이스의 사용
  - 클래스의 **필드(field)**
  - 생성자 또는 메소드의 **매개변수(parameter)**
  - 생성자 또는 메소드의 **로컬 변수(local variable)**

```
MyClass mc = new MyClass( new Television() );

public class MyClass {
      // field
      RemoteControl rc = new Television();
      // constructor – parameter
      MyClass(RemoteControl rc) {
            this.rc = rc;
      }
      // method
      public method() {
            // local variable
            RemoteControl rc = new Audio();
      }
}
```

추상 클래스와 인터페이스 비교

비교	내용
추상 클래스	• 일반 메소드 포함 가능 • 상수, 변수 필드 포함 가능 • 모든 서브 클래스에 공통된 메소드가 있는 경우에는 추상 클래스가 적합
인터페이스	• 모든 메소드가 추상 메소드 • 상수 필드만 포함 가능 • 다중 상속 지원

Comparable 인터페이스

- Comparable 인터페이스는 객체의 비교를 위한 인터페이스로 객체 간의 순서나 정렬을 하기 위해서 사용

```
public interface Comparable {
      // 이 객체가 다른 객체보다 크면 1, 같으면 0, 작으면 -1을 반환한다.
      int compareTo(Object other);
}
```

```
class Person implements Comparable {
    public int compareTo(Object other) {
        Person p = (Person)other;
        if (this.age == p.age) return 0;
        else if (this.age > p.age) return 1;
        else   return -1;
    }
}
```

Comparable 인터페이스

```
public Object findLargest(Object object1, Object object2) {
   Comparable obj1 = (Comparable)object1;
   Comparable obj2 = (Comparable)object2;
   if ((obj1).compareTo(obj2) > 0)
      return object1;
   else
      return object2;
}
```

예제 : Comparable 인터페이스

```
public class Rectangle implements Comparable {
  public int width = 0;       public int height = 0;
  @Override
  public String toString() { return "Rect [w=" + width + ", h=" + height + "]"; }
  public Rectangle(int w, int h) { width = w; height = h; System.out.println(this); }
  public int getArea() { return width * height; }
  @Override
  public int compareTo(Object other) {
     Rectangle otherRect = (Rectangle)other;
     if (this.getArea() < otherRect.getArea())
          return -1;
     else if (this.getArea() > otherRect.getArea())
          return 1;
     else
          return 0;
  }
}
```

예제 : Comparable 인터페이스

```
public class RectangleTest {
  public static void main(String[] args) {
  Rectangle r1 = new Rectangle(100, 30);
  Rectangle r2 = new Rectangle(200, 10);
  int result = r1.compareTo(r2);
  if (result == 1)
          System.out.println(r1 + "가 더 큽니다.");
  else if (result == 0)
          System.out.println("같습니다");
  else
          System.out.println(r2 + "가 더 큽니다.");
  }
}
```

Comparator 인터페이스

- Comparator 인터페이스는 다른 두 개의 객체를 비교하기 위한 인터페이스

```
public interface Comparator {
      // o1가 o2보다 크면 1, 같으면 0, 작으면 -1을 반환한다.
      int compare(Object o1, Object o2);
}
```

```
class AgeComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        Person p1 = (Person)o1;
        Person p2 = (Person)o2;
        if (p1.age == p2.age) return 0;
        else if (p1.age > p2.age) return 1;
        else   return -1;
    }
}
```

Enumeration 인터페이스

- Enumeration 인터페이스는 객체들의 집합(Vector)에서 각각의 객체들을 순차적으로 처리할 수 있는 메소드를 제공하는 인터페이스
  - Enumeration 객체는 new 연사자로 생성할 수 없으며, Vector를 이용하여 생성할 수 있다.
  - Vector 클래스의 elements()라는 메소드는 객체의 모든 요소들을 Enumeration 객체로 반환한다.

```
public interface Enumeration {
      // Vector로부터 생성된 Enumeration 요소가 있으면 true, 아니면 false
      boolean hasMoreElements();
      // 다음 Enumeration 요소를 반환
      Object nextElement();
}
```

예제 : Enumeration 인터페이스

```
public class EnumerationTest {
   public static void main(String[] args) {
        Vector<String> v1 = new Vector<String>();
        v1.addElement("ABC");
        v1.addElement("DEF");
        v1.addElement("GHI");
        for(int i=0; i<v1.size(); i++){
            System.out.println("v1["+i+"]="+v1.elementAt(i));
        }
        Enumeration<String> e = v1.elements();
        while(e.hasMoreElements()){
            System.out.println("e="+e.nextElement());
        }
   }
}
```

ActionListener 인터페이스

- ActionListener 인터페이스는 자바의 이벤트 리스너 (이벤트를 처리하는 인터페이스) 중 하나로, 사용자가 Action을 했을 시 Action 이벤트를 발생시키는 인터페이스

```
public interface ActionListener {
      // 액션이벤트가 발생할 때 호출된다.
      void actionPerformed(ActionEvent other);
}
```

예제 : ActionListener 인터페이스

```
class MyClass implements ActionListener {
  public void actionPerformed(ActionEvent event) {
          System.out.println("beep");
  }
}
public class CallbackTest {
  public static void main(String[] args) {
          ActionListener listener = new MyClass();
          Timer t = new Timer(1000, listener);
          t.start();
          for (int i = 0; i < 1000; i++) {
                  try {
                          Thread.sleep(1000);
                  } catch (InterruptedException e) {
                  }
          }
  }
}
```

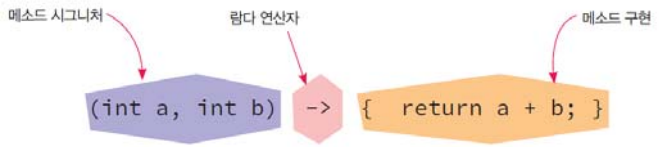
```
beep
beep
beep
...
// 1초에 한번씩 "beep" 출력
```

람다식

- 람다식(lambda expression)은 나중에 실행될 목적으로 다른 곳에 전달될 수 있는 코드 블록이다.
- 람다식을 이용하면 메소드가 필요한 곳에 간단히 메소드를 보낼 수 있다.

람다식의 구문

- 람다식은 **(argument-list) -> {body}** 구문 사용하여 작성



- 람다식의 예
  - **() -> System.out.println("Hello World");**
  - **(String s) -> { System.out.println(s); }**
  - **() -> 69**
  - **() -> { return 3.141592; };**
  - **(String s) -> { return "Hello, " + s; };**

예제 : 람다식 사용

```
import javax.swing.Timer;
public class CallbackTest {
  public static void main(String[] args) {
  Timer t = new Timer(1000, event -> System.out.println("beep"));
  t.start();
  for (int i = 0; i < 1000; i++) {
          try {
                  Thread.sleep(1000);
          } catch (InterruptedException e) {
          }
  }
  }
}
```

```
beep
beep
beep
...
// 1초에 한번씩 "beep" 출력
```

함수 인터페이스와 람다식

- 함수 인터페이스는 하나의 추상 메서드만 선언된 인터페이스, 예: java.lang.Runnable
- 람다식은 함수 인터페이스에 대입할 수 있다.
  - Runnable r = () -> System.out.println("스레드가 실행되고 있습니다.");

예제 : 함수 인터페이스

```
@FunctionalInterface
interface MyInterface {
  void sayHello();
}

public class LambdaTest1 {
  public static void main(String[] args) {
     MyInterface hello = () -> System.out.println("Hello Lambda!");
     hello.sayHello();
  }
}
```

```
Hello Lambda!
```

람다식을 사용한 sort

- 무명클래스를 사용한 방식

```
Comparator<Person> byName = new Comparator<Person>() {
   @Override
   public int compare(Person o1, Person o2) {
      return o1.getName().compareTo(o2.getName());
   }
};
```

- 람다식를 사용한 방식

```
Comparator<Person> byName = (Person o1, Person o2) ->
                                    o1.getName().compareTo(o2.getName());
};
```

람다식을 사용한 sort

- 무명클래스를 사용한 방식

```
List<Person> plist = getPersonList();
// sort by age
plist.sort(new Comparator<Person>() {
   @Override
   public int compare(Person o1, Person o2) {
      return o1.getAge() - o2.getAge();
   }
});
```

- 람다식을 사용한 방식

```
// sort by age
plist.sort( (Person o1, Person o2) -> o1.getAge() - o2.getAge() );
plist.forEach( (person) -> System.out.println(person) );
```