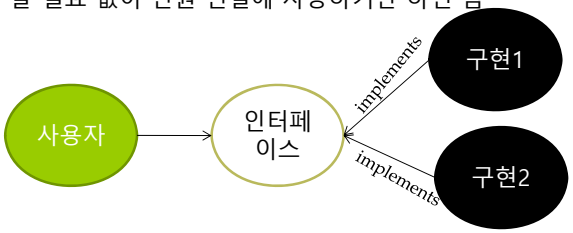
Interface, Collections, Lambda

514770-1
2017년 가을학기
3/29/2017
박경신

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

```
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 및 사용

```
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

```
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

```
인터페이스명 변수 = new 인터페이스명() {
    //인터페이스에 선언된 추상 메소드의 실체 메소드 구현

}
```

    - 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();
        }
}
```

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;
    }
}
```

Ex : 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;
  }
}
```

Ex : 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

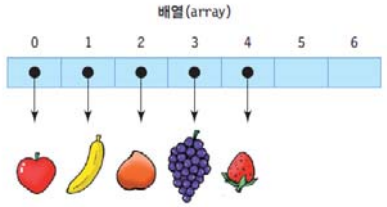
- Comparator 인터페이스는 다른 두 개의 객체를 비교하기 위한 인터페이스

```
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 (s1.age == s2.age) return 0;
        else if (s1.age > s2.age) return 1;
        else    return -1;
    }
}
```

컬렉션(collection)의 개념

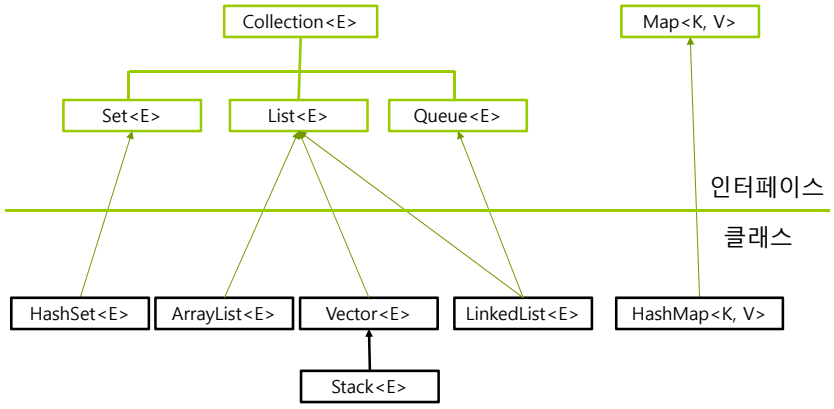
- 컬렉션
  - 요소(element)라고 불리는 가변 개수의 객체들의 저장소
    - 객체들의 컨테이너라고도 불림
    - 요소의 개수에 따라 크기 자동 조절
    - 요소의 삽입, 삭제에 따른 요소의 위치 자동 이동
  - 고정 크기의 배열을 다루는 어려움 해소
  - 다양한 객체들의 삽입, 삭제, 검색 등의 관리 용이



- 고정 크기 이상의 객체를 관리할 수 없다.
- 배열의 중간에 객체가 삭제되면 응용프로그램에서 자리를 옮겨야 한다.

- 가변 크기로서 객체의 개수를 염려할 필요 없다.
- 컬렉션 내의 한 객체가 삭제되면 컬렉션이 자동으로 자리를 옮겨준다.

컬렉션 인터페이스와 클래스



Collection 인터페이스 메소드

Method	Description
boolean add(Object element)	is used to insert an element in this collection.
boolean addAll(Collection c)	is used to insert the specified collection elements in the collection.
boolean remove(Object element)	is used to delete an element from this collection.
boolean removeAll(Collection c)	is used to delete all the elements of specified collection from the collection.
boolean retainAll(Collection c)	is used to delete all the elements of invoking collection except the specified collection.
int size()	return the total number of elements in the collection.
void clear()	removes the total no of element from the collection.
boolean contains(Object element)	is used to search an element.
boolean containsAll(Collection c)	is used to search the specified collection in this collection.
Iterator iterator()	returns an iterator.
Object[] toArray()	converts collection into array.
boolean isEmpty()	checks if collection is empty.
boolean equals(Object element)	matches two collection.
int hashCode()	returns the hashcode number for collection.

List 인터페이스 메소드

Method	Description
void add(int index, Object elemen t)	is used to insert an element into the list at index passed in t he index.
boolean addAll(int index, Collecti on c)	is used to insert all elements of c into the list at the index p assed in the index.
object get(int index)	is used to return the object stored at the specified index wit hin the collection.
object set(int index, Object eleme nt)	is used to assign element to the location specified by index within the list.
object remove(int index)	is used to delete all the elements of invoking collection exce pt the specified collection.
ListIterator listIterator()	is used to return an iterator to the start of the list.
ListIterator listIterator(int index)	is used to return an iterator to the list that begins at the sp ecified index.

컬렉션과 제네릭

- 컬렉션은 제네릭(Generics) 기법으로 구현됨
- 컬렉션의 요소는 객체만 가능
  - 기본적으로 int, char, double 등의 기본 타입 사용 불가
    - JDK 1.5부터 자동 박싱/언박싱 기능으로 기본 타입 사용 가능
- 제네릭(Generics)
  - 특정 타입만 다루지 않고, 여러 종류의 타입으로 변신할 수 있도록 클래스나 메소드를 일반화시키는 기법
    - <E>, <K>, <V> : 타입 매개 변수
      - 요소 타입을 일반화한 타입
  - 제네릭 클래스 사례
    - 제네릭 벡터 : Vector<E>
    - E에 특정 타입으로 구체화
    - 정수만 다루는 벡터 Vector<Integer>
    - 문자열만 다루는 벡터 Vector<String>

컬렉션과 자동 박싱/언박싱

- JDK 1.5 이전
  - 기본 타입 데이터를 Wrapper 클래스를 이용하여 객체로 만들어 사용

```
Vector<Integer> v = new Vector<Integer>();
v.add(new Integer(4));
```

  - 컬렉션으로부터 요소를 얻어올 때, Wrapper 클래스로 캐스팅 필요

```
Integer n = (Integer)v.get(0);
int k = n.intValue(); // k = 4
```

- JDK 1.5부터
  - 자동 박싱/언박싱이 작동하여 기본 타입 값 사용 가능

```
Vector<Integer> v = new Vector<Integer> ();
v.add(4); // 4 → new Integer(4)로 자동 박싱
int k = v.get(0); // Integer 타입이 int 타입으로 자동 언박싱, k = 4
```

    - 제네릭의 타입 매개 변수를 기본 타입으로 구체화할 수는 없음

```
Vector<int> v = new Vector<int> (); // 오류
```

Vector<E>

- Vector<E>의 특성
  - java.util.Vector
    - <E>에서 E 대신 요소로 사용할 특정 타입으로 구체화
  - 여러 객체들을 삽입, 삭제, 검색하는 컨테이너 클래스
    - 배열의 길이 제한 극복
    - 원소의 개수가 넘쳐나면 자동으로 길이 조절
  - Vector에 삽입 가능한 것
    - 객체, null
    - 기본 타입은 박싱/언박싱으로 Wrapper 객체로 만들어 저장
  - Vector에 객체 삽입
    - 벡터의 맨 뒤에 객체 추가
    - 벡터 중간에 객체 삽입
  - Vector에서 객체 삭제
    - 임의의 위치에 있는 객체 삭제 가능 : 객체 삭제 후 자동 자리 이동

Vector<E> 클래스의 주요 메소드

메소드	설명
boolean add(E element)	벡터의 맨 뒤에 element 추가
void add(int index, E element)	인덱스 index에 element를 삽입
int capacity()	벡터의 현재 용량 리턴
boolean addAll(Collection<? extends E> c)	컬렉션 c의 모든 요소를 벡터의 맨 뒤에 추가
void clear()	벡터의 모든 요소 삭제
boolean contains(Object o)	벡터가 지정된 객체 o를 포함하고 있으면 true 리턴
E elementAt(int index)	인덱스 index의 요소 리턴
E get(int index)	인덱스 index의 요소 리턴
int indexOf(Object o)	o와 같은 첫 번째 요소의 인덱스 리턴. 없으면 -1 리턴
boolean isEmpty()	벡터가 비어 있으면 true 리턴
E remove(int index)	인덱스 index의 요소 삭제
boolean remove(Object o)	객체 o와 같은 첫 번째 요소를 벡터에서 삭제
void removeAllElements()	벡터의 모든 요소를 삭제하고 크기를 0으로 만듦
int size()	벡터가 포함하는 요소의 개수 리턴
Object[] toArray()	벡터의 모든 요소를 포함하는 배열 리턴

ArrayList<E>

- ArrayList<E>의 특성
  - java.util.ArrayList, 가변 크기 배열을 구현한 클래스
    - <E>에서 E 대신 요소로 사용할 특정 타입으로 구체화
  - ArrayList에 삽입 가능한 것
    - 객체, null
    - 기본 타입은 박싱/언박싱으로 Wrapper 객체로 만들어 저장
  - ArrayList에 객체 삽입/삭제
    - 리스트의 맨 뒤에 객체 추가
    - 리스트의 중간에 객체 삽입
    - 임의의 위치에 있는 객체 삭제 가능
  - 벡터와 달리 스레드 동기화 기능 없음
    - 다수 스레드가 동시에 ArrayList에 접근할 때 동기화되지 않음
    - 개발자가 스레드 동기화 코드 작성

ArrayList<E> 클래스의 주요 메소드

메소드	설명
boolean add(E element)	ArrayList의 맨 뒤에 element 추가
void add(int index, E element)	인덱스 index에 지정된 element 삽입
boolean addAll(Collection<? extends E> c)	컬렉션 c의 모든 요소를 ArrayList의 맨 뒤에 추가
void clear()	ArrayList의 모든 요소 삭제
boolean contains(Object o)	ArrayList가 지정된 객체를 포함하고 있으면 true 리턴
E elementAt(int index)	index 인덱스의 요소 리턴
E get(int index)	index 인덱스의 요소 리턴
int indexOf(Object o)	o와 같은 첫 번째 요소의 인덱스 리턴. 없으면 -1 리턴
boolean isEmpty()	ArrayList가 비어 있으면 true 리턴
E remove(int index)	index 인덱스의 요소 삭제
boolean remove(Object o)	o와 같은 첫 번째 요소를 ArrayList에서 삭제
int size()	ArrayList가 포함하는 요소의 개수 리턴
Object[] toArray()	ArrayList의 모든 요소를 포함하는 배열 리턴

HashMap<K,V>

- HashMap<K,V>
  - 키(key)와 값(value)의 쌍으로 구성되는 요소를 다루는 컬렉션
    - java.util.HashMap
    - K는 키로 사용할 요소의 타입, V는 값으로 사용할 요소의 타입 지정
    - 키와 값이 한 쌍으로 삽입
    - 키는 해시맵에 삽입되는 위치 결정에 사용
    - 값을 검색하기 위해서는 반드시 키 이용
  - 삽입 및 검색이 빠른 특징
    - 요소 삽입 : put() 메소드
    - 요소 검색 : get() 메소드
  - 예) HashMap<String, String> 생성, 요소 삽입, 요소 검색

```
HashMap<String, String> h = new HashMap<String, String>();
h.put("apple", "사과"); // "apple" 키와 "사과" 값의 쌍을 해시맵에 삽입
String kor = h.get("apple"); // "apple" 키로 값 검색. kor는 "사과"
```

HashMap<K,V>의 주요 메소드

메소드	설명
void clear()	HashMap의 모든 요소 삭제
boolean containsKey(Object key)	지정된 키(key)를 포함하고 있으면 true 리턴
boolean containsValue(Object value)	하나 이상의 키를 지정된 값(value)에 매핑시킬 수 있으면 true 리턴
V get(Object key)	지정된 키(key)에 매핑되는 값 리턴. 키에 매핑되는 어떤 값도 없으면 null 리턴
boolean isEmpty()	HashMap이 비어 있으면 true 리턴
Set<K> keySet()	HashMap에 있는 모든 키를 담은 Set<K> 컬렉션 리턴
V put(K key, V value)	key와 value를 매핑하여 HashMap에 저장
V remove(Object key)	지정된 키(key)와 이에 매핑된 값을 HashMap에서 삭제
int size()	HashMap에 포함된 요소의 개수 리턴

LinkedList<E>

- LinkedList<E>의 특성
  - java.util.LinkedList
    - E에 요소로 사용할 타입 지정하여 구체와
  - List 인터페이스를 구현한 컬렉션 클래스
  - Vector, ArrayList 클래스와 매우 유사하게 작동
  - 요소 객체들은 양방향으로 연결되어 관리됨
  - 요소 객체는 맨 앞, 맨 뒤에 추가 가능
  - 요소 객체는 인덱스를 이용하여 중간에 삽입 가능
  - 맨 앞이나 맨 뒤에 요소를 추가하거나 삭제할 수 있어 스택이나 큐로 사용 가능

컬렉션 순차 검색을 위한 Iterator 인터페이스

- Iterator<E> 인터페이스
  - Vector<E>, ArrayList<E>, LinkedList<E>가 상속받는 인터페이스
    - 리스트 구조의 컬렉션에서 요소의 순차 검색을 위한 메소드 포함
  - Iterator<E> 인터페이스 메소드

메소드	설명
boolean hasNext()	다음 반복에서 사용될 요소가 있으면 true 리턴
E next()	다음 요소 리턴
void remove()	마지막으로 리턴된 요소 제거

  - iterator() 메소드
    - iterator()를 호출하면 Iterator 객체 반환
    - Iterator 객체를 이용하여 인덱스 없이 순차적 검색 가능

컬렉션의 순차 검색을 위한 Iterator

```
Vector<Integer> v = new Vector<Integer>();
Iterator<Integer> it = v.iterator();
while(it.hasNext()) { // 모든 요소 방문
        int n = it.next(); // 다음 요소 리턴
        …
}

또는

for (int n : v) {
    …
}
```

Custom 클래스에 대한 sort 메소드 사용

- 개인적으로 만든 클래스에 대해서 컬렉션에 추가하고, Collections.sort 기능을 이용해서 정렬하고 싶다면 java.lang.Comparable 인터페이스를 구현해주어야 함

```
public interface Comparable<T> {
   int compareTo(T o);
}
```

- CompareTo(T o) 메소드는 현 객체를 인자로 주어진 o와 비교해서 순서를 정한 후에 정수(int) 값을 반환함
  - 만약 현 객체가 주어진 인자보다 작다면 음수를 반환
  - 만약 현 객체가 주어진 인자와 동일하다면 0을 반환
  - 만약 현 객체가 주어진 인자보다 크다면 양수를 반환

Custom 클래스에 대한 sort 메소드 사용

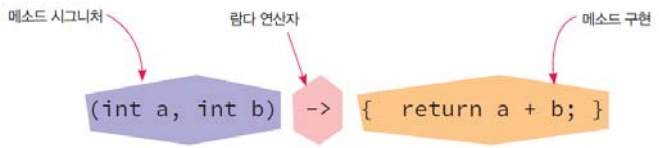
```
import java.util.Collections;
class A implements java.lang.Comparable<A> {
    int num;    String s;
    public A(String s, int n) {
        this.s = s; num = n;
    }
    public int compareTo(A a) {
        if (s.compareTo(a.s) == 0) {
            if (num > a.num)
                return 1;
            else if (num < a.num)
                return -1;
            else
                return 0;
        }
        else {
            return s.compareTo(a.s);
        }
    }
    public String toString() { return "String: " + s + "₩t num = " + num;    }
}
```

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; };**

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) );
```