**List 인터페이스**

2017-09-29

이승진

**학습목표**

List 인터페이스 메소드 사용법을 학습한다.

**목차**

[1. List interface 2](#_Toc37258925)

[1) List interface 메소드 2](#_Toc37258926)

[2) List 인터페이스를 구현한 클래스 5](#_Toc37258927)

[3) 부모 타입의 변수 선언과 다형성 6](#_Toc37258928)

[2. List 예제 7](#_Toc37258929)

[1) 예제1 7](#_Toc37258930)

[2) 예제2 8](#_Toc37258931)

[3) 예제3 9](#_Toc37258932)

[4) 예제4 10](#_Toc37258933)

[5) 예제5 11](#_Toc37258934)

[6) 예제6 12](#_Toc37258935)

[7) 예제7 13](#_Toc37258936)

[8) 예제8 14](#_Toc37258937)

[3. 다형성 활용 15](#_Toc37258938)

[1) Example1 15](#_Toc37258939)

[2) Example2 16](#_Toc37258940)

[3) 다형성 활용 17](#_Toc37258941)

[4) Example3 17](#_Toc37258942)

[4. 구현 실습 18](#_Toc37258943)

[1) 홀수만 남기기 18](#_Toc37258944)

[2) 정렬 구현 18](#_Toc37258945)

# List interface

List 인터페이스를 구현한 대표적인 클래스가 ArrayList 클래스이다.

즉 List 인터페이스의 대표적인 자식 클래스가 ArrayList 클래스이다.

배열과 비슷한 방법으로 사용할 수 있는 클래스들을 대표하는 것이 List 인터페이스이다.

List 인터페이스의 자식 클래스들은, 배열처럼 인덱스(index) 위치에 값을 저장한다.

인덱스는 0부터 시작한다.

## List interface 메소드

List 인터페이스는 Collection 인터페이스를 상속한다.

따라서 Collection 인터페이스에 정의된 모든 메소드가 List 인터페이스에 상속된다.

|  |
| --- |
| boolean add(E e)  boolean addAll(Collection<E> c)  void clear()  boolean contains(Object o)  boolean containsAll(Collection<E> c)  boolean equals(Object o)  int hashCode()  boolean isEmpty()  Iterator<E> iterator()  boolean remove(Object o)  boolean removeAll(Collection c)  boolean retainAll(Collection c)  int size()  Object[] toArray()  E[] toArray(E[] a) |

위 메소드들은 Collection 인터페이스에 선언된 메소드들이다.

이 메소드들이 List 인터페이스에 상속된다.

### Collection 인터페이스로부터 상속 받은 메소드

|  |
| --- |
| **boolean add(E e)**  객체 e를 내부 목록에 추가한다.  추가 작업이 성공했으면 true를, 실패면 false를 리턴한다. |
| **boolean addAll(Collection<E> c)**  목록 객체 c에 들어있는 항목들을 전부 내부 목록에 추가한다.  추가 작업이 성공했으면 true를, 실패면 false를 리턴한다. |
| **void clear()**  내부 목록에 들어있는 항목 전체를 제거한다. 그 결과 내부 목록은 비어 있는 상태가 된다. |
| **boolean contains(Object o)**  파라미터 o와 동일한 값이 내부 목록에 들어있다면 true를 리턴한다.  동일한 값이 있는지 찾을 때 equals 메소드를 호출해서 비교한다. |
| **boolean containsAll(Collection<E> c)**  컬렉션 객체 c에 들어있는 항목들이 전부 this 객체의 내부 목록에 들어있는지 확인한다.  전부 들어있다면 true를 리턴한다.  동일한 값이 있는지 찾을 때 equals 메소드를 호출해서 비교한다. |
| **boolean equals(Object o)**  객체 o와 this 객체의 equality를 비교한다.  즉 this 객체와 o 객체의 클래스가 같고,  목록 o에 들어있는 항목의 수가, this의 그것과 동일하고,  양쪽 목록에 들어있는 항목들 각각이 서로 동일한 경우에 true 를 리턴한다.  항목들 각각이 서로 동일한지 비교할 때 equals 메소드를 호출해서 비교한다. |
| **int hashCode()**  내부 객체 목록까지 고려하여 hash code 값을 계산해서 리턴한다. |
| **boolean isEmpty()**  목록이 비어 있으면 true 를 리턴한다. |
| **Iterator<E> iterator()**  내부 목록에 들어있는 항목을 하나씩 탐색하기 위한 Iterator 객체를 생성하여 리턴한다. |
| **boolean remove(Object o)**  파라미터 o와 동일한 값(equals)을 목록에서 찾아서 제거한다.  파라미터 값이 null 이면, null을 찾아서 제거한다.  제거에 성공했으면 true를, 제거 실패면 false를 리턴한다.  예를 들어 파라미터와 동일한 값이 목록에 없어서 제거할 것이 없는 경우에 false를 리턴한다. |
| **boolean removeAll(Collection c)**  파라미터로 주어진 목록 객체 c에 들어있는 항목들을 this 객체에서 찾아서 전부 제거한다.  제거된 항목이 한 개 이상이면 true를 리턴한다. |
| **boolean retainAll(Collection c)**  파라미터로 주어진 목록 객체 c에 들어있지 않은 항목들을 this 객체에서 찾아서 전부 제거한다.  그래서 c에 들어있는 항목들만 this 객체에 남아있게 된다.  제거된 항목이 한 개 이상이면 true를 리턴한다. |
| **int size()**  내부 목록에 들어있는 항목의 수를 리턴한다. |
| **Object[] toArray()**  목록 객체에 들어있는 항목들을 배열에 채워서 리턴한다.  리턴되는 배열의 타입은 Object[] 이다. |
| **E[] toArray(E[] a)**  목록 객체에 들어있는 항목들을 배열에 채워서 리턴한다.  리턴되는 배열의 타입은 파라미터로 주어진 배열의 타입과 같다. |

### List interface 에 선언된 메소드

|  |
| --- |
| **void add(int index, E e)**  객체 e를 내부 목록의 index 위치에 끼워 넣는다. |
| **boolean addAll(int index, Collection<E> c)**  목록 객체 c에 들어있는 항목들을 전부 내부 목록의 index 위치에 끼워 넣는다.  추가 작업이 성공했으면 true를, 실패면 false를 리턴한다. |
| **E get(int index)**  내무 목록에서 index 위치의 객체를 리턴한다. |
| **int indexOf(Object o)**  내부 목록에서 객체 o를 찾아서 위치를, 즉 index 값을 리턴한다.  찾지 못한 경우 -1을 리턴한다. |
| **ListIterator<E> listIterator()**  내부 목록에 들어있는 항목을 하나씩 탐색하기 위한 ListIterator 객체를 생성하여 리턴한다. |
| **E remove(int index)**  내부 목록에서 index 위치의 객체를 제거한다.  리턴 값은 제거된 객체이다. |
| **void set(int index, E e)**  내부 목록에서 index 위치에 객체 e를 대입한다.  index 위치의 기존의 값을 덮어쓴다. |
| **List<E> subList(int fromIndex, int toIndex)**  내부 목록에서 formIndex 부터 시작해서, toIndex 직전까지의 항목을 포함하는  새 List 객체를 생성하여 리턴한다. |

### List 인터페이스의 자식 클래스의 메소드

인터페이스이 메소드는 전부 abstract methopd 이다.

인터페이스의 자식 클래스는 부모 인터페이스의 메소드들을 전부 재정의(override)해야 한다.

List 인터페이스의 자식 클래스들은,

List 인터페이스의 메소드들과,

List 인터페이스의 부모인 Collection 인터페이스의 메소드들을

전부 재정의해야 한다.

부모로부터 상속된 abstract method를 전부 재정의 하지 않으면, 그 클래스는 abstract class가 된다.

abstract class는 객체를 생성할 수 없다.

부모로부터 상속된 abstract method들을 전부 재정의한 concrete class는 객체를 생성할 수 있다.

## List 인터페이스를 구현한 클래스

ArrayList 클래스 배열 형태의 자료구조

LinkedList 클래스 링크드 리스트 형태의 자료구조

Vector 클래스 배열 형태의 자료구조 + synchronized object

ArrayList 클래스, LinkedList 클래스, Vector 클래스는

List 인터페이스를 구현하였다.

따라서 Collection 인터페이스와 List 인터페이스에 선언된 메소드들이

이 클래스들에 모두 구현되어 있다.

이렇게 같은 인터페이스를 구현한 자식 클래스들 사이에는 다형성이 적용된다.

### synchronized object

여러 스레드(thread)가 객체 하나를 공유하며 동시에 공유 객체의 메소드를 호출해도

문제가 발생하지 않도록 구현한 객체를 synchronized object 라고 부른다.

sychronized obejct는 다음과 같이 구현된다.

- 스레드가 객체의 메소드를 실행하기 직전에 객체를 잠근다 (lock)

- 스레드가 메소드에서 리턴하기 직전에 잠금을 해제한다 (unlock).

잠겨있는(locked) 객체의 메소드를 다른 스레드가 호출하면,

객체의 잠금이 풀릴 때까지(unlocked), 그 스레드는 자동으로 대기(sleep) 상태가 된다.

객체의 잠금이 풀리면 대기 상태에 있던 스레드가 자동으로 깨어나서

아까 호출했던 메소드를 실행하게 된다.

이때에도 객체를 먼저 잠그고 실행한다.

그래서 어느 한 순간 하나의 스레드만 객체의 메소드를 실행할 수 있다.

이렇게 구현된 객체를 synchronized object 라고 부른다.

비유를 하자면, 공공 화장실 한 칸을 생각하면 된다.

잠겨 있지 않는 칸에만 사람(thread)이 들어갈 수 있고,

이미 잠겨 있는 칸에 들어가려면, 그 칸이 열릴 때까지 대기 해야 한다.

lock, unlock 하는 작업이 무거운 연산이기 때문에

synchronized object는 그렇지 않은 객체보다 느리다.

따라서, 여러 스레드에 의해서 공유되는 경우에는 synchronized obejct를 사용한다.

## 부모 타입의 변수 선언과 다형성

### 구현1

|  |
| --- |
| ArrayList<String> list = new ArrayList<String>(); |

ArrayList<String> 타입 변수에 ArrayList<String> 객체를 대입한다.

### 구현2

|  |
| --- |
| List<String> list = new ArrayList<String>(); |

List<String> 타입 변수에 ArrayList<String> 객체를 대입한다.

구현1과 구현2 모두 합법이다.

구현2가 바람직하다.

예를들어, 남자 클래스와 여자 클래스는, 사람 클래스의 자식 클래스이다.

사람 객체가 들어갈 수 있는 방에, 남자 객체가 들어가는 것은 정상이다.

List 객체가 대입될 수 있는 변수에, ArrayList 객체를 대입하는 것은 정상이다.

어떤 방을 선언할 때, 남자 방이라고 선언해야 하는 특별한 이유가 없다면,

가급적 사람 방이라고 선언하는 것이 융통성이 있다.

가급적 List 타입의 변수라고 선언하는 것이 융통성이 있다.

남자 방이라서 선언하면, 여자 객체는 들어갈 수 없기 때문에, 제한적이 된다.

ArrayList 타입의 변수라고 선언하면, LinkedList 객체는 대입할 수 없기 때문에, 제한적이 된다.

사람 방이라고 선언하면, 남자도 여자도 들어갈 수 있다.

List 타입의 변수라고 선언하면, ArrayList 객체도 LinkedList 객체도 대입할 수 있다.

### 다형성

부모 타입의 변수로 메소드를 호출해야, 다형성이 실현된다.

남자 방에서 누가 노래를 부르면, 남자가 노래하는 것이다.

사람 방에서 누가 노래를 부르면, 남자가 노래할 수도 있고, 여자가 노래할 수도 있다.

ArrayList 타입의 변수로 메소드를 호출하면, ArrayList 메소드가 호출된다.

List 타입의 변수로 메소드를 호출하면, ArrayList 메소드가 호출될 수도 있고, LikedList 메소드가 호출될 수도 있다. 이것이 다형성이다.

위와 같은 이유로,

지역 변수나 멤버 변수를 선언할 때, 메소드 리턴 타입을 선언할 때, 파라미터 변수를 선언할 때,

특별한 이유가 없다면, 가급적 부모 타입으로 선언하는 것이 바람직하다.

# List 예제

## 예제1

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | package net.skhu.list;  import java.util.ArrayList;  import java.util.List;  public class Example01 {    public static void main(String[] args) {  List<String> list = new ArrayList<String>();    list.add("one");  list.add("three");  list.add(0, "zero");  list.add(2, "two");  for (String s : list)  System.out.printf("%s ", s);    }  } |

실행 결과 출력

|  |
| --- |
| zero one two three |

## 예제2

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | package net.skhu.list;  import java.util.ArrayList;  import java.util.LinkedList;  import java.util.List;  public class Example02 {  public static void main(String[] args) {  List<String> list1 = new ArrayList<String>();  List<String> list2 = new LinkedList<String>();  list1.add("one");  list1.add("two");  list2.add("zero");  list2.add("three");  list2.addAll(1, list1);  for (String s : list2)  System.out.printf("%s ", s);  }  } |

실행 결과 출력

|  |
| --- |
| zero one two three |

## 예제3

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | package net.skhu.list;  import java.util.ArrayList;  import java.util.List;  public class Example03 {  public static void main(String[] args) {  List<String> list = new ArrayList<String>();  list.add("one");  list.add("three");  list.add(0, "zero");  list.add(2, "two");  for (String s : list)  System.out.printf("%s ", s);  for (int i = 0; i < list.size(); ++i)  System.out.printf("%s ", list.get(i));  }  } |

실행 결과 출력

|  |
| --- |
| zero one two three zero one two three |

## 예제4

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | package net.skhu.list;  import java.util.ArrayList;  import java.util.List;  public class Example04 {  public static void main(String[] args) {  List<String> list = new ArrayList<String>();  list.add("zero");  list.add("one");  list.add("two");  list.add("three");  System.out.println(list.indexOf("two"));  System.out.println(list.indexOf("three"));  System.out.println(list.indexOf("four"));  }  } |

실행 결과 출력

|  |
| --- |
| 2  3  -1 |

## 예제5

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | package net.skhu.list;  import java.util.ArrayList;  import java.util.List;  public class Example05 {  public static void main(String[] args) {  List<String> list = new ArrayList<String>();  list.add("zero");  list.add("one");  list.add("two");  list.add("three");  list.remove("two");  for (String s : list)  System.out.printf("%s ", s);  }  } |

실행 결과 출력

|  |
| --- |
| zero one three |

## 예제6

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | package net.skhu.list;  import java.util.ArrayList;  import java.util.List;  public class Example06 {  public static void main(String[] args) {  List<String> list = new ArrayList<String>();  list.add("zero");  list.add("one");  list.add("two");  list.add("three");  list.set(2, "TWO");  for (String s : list)  System.out.printf("%s ", s);  }  } |

실행 결과 출력

|  |
| --- |
| zero one TWO three |

## 예제7

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | package net.skhu.list;  import java.util.ArrayList;  import java.util.List;  public class Example07 {  public static void main(String[] args) {  List<String> list = new ArrayList<String>();  list.add("zero");  list.add("one");  list.add("two");  list.add("three");  list.add("four");  List<String> list2 = list.subList(1, 4);  for (String s : list2)  System.out.printf("%s ", s);  }  } |

실행 결과 출력

|  |
| --- |
| one two three |

## 예제8

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | package net.skhu.list;  import java.util.ArrayList;  import java.util.List;  public class Example08 {  public static void main(String[] args) {  List<String> list = new ArrayList<String>();  list.add("zero");  list.add("one");  list.add("two");  list.add("three");  list.set(1, list.get(0));  list.set(3, list.get(2));  for (String s : list)  System.out.printf("%s ", s);  }  } |

실행 결과 출력

|  |
| --- |
| zero zero two two |

# 다형성 활용

인터페이스(interface)를 만드는 이유는 다형성을 활용하기 위해서다.

다형성 활용의 장점을 알아보자.

## Example1

|  |  |
| --- | --- |
| 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 | package net.skhu.list.sum;  import java.util.ArrayList;  import java.util.LinkedList;  import java.util.Random;  public class Example1 {  static int getSum(ArrayList<Integer> list) {  int sum = 0;  for (Integer i : list)  sum = sum + i;  return sum;  }  public static void main(String[] args) {  ArrayList<Integer> list1 = new ArrayList<Integer>();  LinkedList<Integer> list2 = new LinkedList<Integer>();  Random random = new Random();  for (int i = 0; i < 10; ++i) {  list1.add(random.nextInt(20));  list2.add(random.nextInt(20));  }  System.out.printf("%s 합계: %d\n", list1.toString(), getSum(list1));  // int sum = getSum(list2); 컴파일 에러 발생  }  } |

실행 결과 출력

|  |
| --- |
| [1, 8, 17, 14, 8, 4, 3, 16, 7, 7] 합계: 85 |

// int sum = getSum(list2); 컴파일 에러 발생

getSum 메소드의 파라미터 변수는 ArrayList<Integer> 타입이고

list2 변수는 LinkedList<Integer> 타입이기 때문에

서로 일치하지 않아서 에러가 발생한다.

ArrayList<Integer> list 파라미터 변수에 list2 를 대입할 수 없기 때문이다.

## Example2

|  |  |
| --- | --- |
| 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 | package net.skhu.list.sum;  import java.util.ArrayList;  import java.util.LinkedList;  import java.util.List;  import java.util.Random;  public class Example2 {  static int getSum(List<Integer> list) {  int sum = 0;  for (Integer i : list)  sum = sum + i;  return sum;  }  public static void main(String[] args) {  ArrayList<Integer> list1 = new ArrayList<Integer>();  LinkedList<Integer> list2 = new LinkedList<Integer>();  Random random = new Random();  for (int i = 0; i < 10; ++i) {  list1.add(random.nextInt(20));  list2.add(random.nextInt(20));  }  System.out.printf("%s 합계: %d\n", list1.toString(), getSum(list1));  System.out.printf("%s 합계: %d\n", list2.toString(), getSum(list2));  }  } |

실행 결과 출력

|  |
| --- |
| [11, 14, 14, 16, 9, 4, 11, 0, 16, 5] 합계: 100  [5, 8, 14, 17, 3, 0, 6, 17, 14, 5] 합계: 89 |

int getSum(List<Integer> list)

메소드 파라미터 타입이 List<Integer> 타입이기 때문에

int sum = getSum(list1);

int sum = getSum(list2);

둘 다 가능하다. 그 이유는,

List<Integer> list = list1;

List<Integer> list = list2;

둘 다 가능하기 때문이다.

## 다형성 활용

int getSum(ArrayList<Integer> list)

int getSum(LinkedList<Integer> list)

이 두 메소드를 다 구현하는 것은 낭비다.

int getSum(List<Integer> list)

메소드만 구현하는 것이 정답이다.

이 메소드의 파라미터 변수 타입이 List<Integer> 이기 때문에,

ArrayList<Integer>, LinkedList<Integer>, Vector<Integer> 객체들을 사용할 수 있다.

파라미터 변수 타입을 가급적 부모 클래스나, 부모 인터페이스 타입으로 선언하여

다형성을 활용하자.

## Example3

|  |  |
| --- | --- |
| 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 | package net.skhu.list.sum;  import java.util.ArrayList;  import java.util.LinkedList;  import java.util.List;  import java.util.Random;  import java.util.Vector;  public class Example3 {  static int getSum(List<Integer> list) {  int sum = 0;  for (Integer i : list)  sum = sum + i;  return sum;  }  static void addRandomValue(List<Integer> list, int count) {  Random random = new Random();  for (int i = 0; i < count; ++i)  list.add(random.nextInt(20));  }  public static void main(String[] args) {  ArrayList<Integer> list1 = new ArrayList<>();  LinkedList<Integer> list2 = new LinkedList<>();  Vector<Integer> list3 = new Vector<>();  addRandomValue(list1, 10);  addRandomValue(list2, 10);  addRandomValue(list3, 10);  System.out.printf("%s 합계: %d\n", list1.toString(), getSum(list1));  System.out.printf("%s 합계: %d\n", list2.toString(), getSum(list2));  System.out.printf("%s 합계: %d\n", list3.toString(), getSum(list3));  }  } |

실행 결과 출력

|  |
| --- |
| [11, 2, 14, 8, 18, 11, 18, 16, 19, 19] 합계: 136  [11, 8, 5, 12, 14, 6, 19, 5, 5, 4] 합계: 89  [2, 8, 4, 14, 1, 5, 1, 16, 3, 18] 합계: 72 |

# 구현 실습

## 홀수만 남기기

LinkedList<Integer> 타입의 목록 객체를 하나 생성하고

0 .. 99 까지 정수를 저장하라.

목록 객체의 size 메소드와 get 메소드를 사용하여 목록 객체를 탐색하면서

홀수 값을 모두 제거하라.

## 정렬 구현

ArrayList<Integer> 타입의 목록 객체를 하나 생성하고 정수 난수를 200 개 추가하라.

(1 ~ 100 사이의 random integer 200 개)

ArrayList<Integer> 타입의 목록 객체를 정렬하는 메소드를 구현하라.

size, get, set 메소드 사용 해서 구현.

### bubble sort 구현 참고

|  |  |
| --- | --- |
| 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 | package net.skhu.sort.bubble;  import java.util.Arrays;  public class Example1 {  // 배열 a에서 i 위치와 j 위치의 값을 서로 바꾼다  static void swap(int[] a, int i, int j) {  int temp = a[i];  a[i] = a[j];  a[j] = temp;  }  // bubble sort  static void bubbleSort(int[] a) {  for (int i = a.length - 1; i >= 1; --i) {  for (int j = 0; j < i; ++j) {  if (a[j] > a[j + 1])  swap(a, j, j + 1);  }  }  }  public static void main(String[] args) {  int[] a = { 17, 14, 11, 19, 13, 15, 20, 12, 16, 18 };  bubbleSort(a);  System.out.println(Arrays.toString(a));  }  } |