

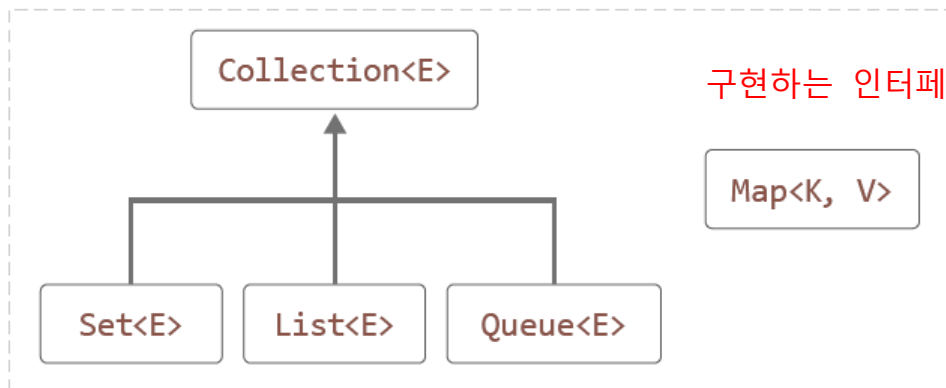
# 컬렉션 프레임워크 1

23-1.

컬렉션 프레임워크의 이해

# 컬렉션 프레임워크

컬렉션 프레임워크의 골격에 해당하는 인터페이스들



구현하는 인터페이스에 따라 사용방법과 특성이 결정된다.

자료구조 및 알고리즘을 구현해 놓은 일종의 라이브러리!  
제네릭 기반으로 구현이 되어 있다.

## 23-2. List<E> 인터페이스를 구현하는 컬렉션 클래스들

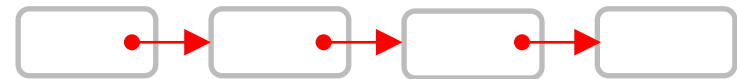
# List<E> 인터페이스

List<E> 인터페이스를 구현하는 대표적인 컬렉션 클래스 둘은 다음과 같다.

- `ArrayList<E>`                      배열 기반 자료구조, 배열을 이용하여 인스턴스 저장
- `LinkedList<E>`                  리스트 기반 자료구조, 리스트를 구성하여 인스턴스 저장

List<E> 인터페이스를 구현하는 컬렉션 클래스들의 공통 특성

- 인스턴스의 저장 순서 유지
- 동일 인스턴스의 중복 저장을 허용한다.



리스트 자료구조의 구성

# ArrayList<E> 클래스

```
public static void main(String[] args) {  
    List<String> list = new ArrayList<>(); // 컬렉션 인스턴스 생성
```

```
    // 컬렉션 인스턴스에 문자열 인스턴스 저장  
    list.add("Toy");  
    list.add("Box");  
    list.add("Robot");
```

```
    // 저장된 문자열 인스턴스의 참조  
    for(int i = 0; i < list.size(); i++)  
        System.out.print(list.get(i) + '\t');  
    System.out.println();
```

```
    list.remove(0); // 첫 번째 인스턴스 삭제
```

```
    // 첫 번째 인스턴스 삭제 후 나머지 인스턴스들을 참조  
    for(int i = 0; i < list.size(); i++)  
        System.out.print(list.get(i) + '\t');  
    System.out.println();  
}
```

배열 기반 자료구조이지만 공간의 확보 및 확장은  
ArrayList 인스턴스가 스스로 처리한다.



```
C:\JavaStudy>java ArrayListCollection  
Toy      Box      Robot  
Box      Robot  
C:\JavaStudy>
```

# LinkedList<E> 클래스

```
public static void main(String[] args) {  
    List<String> list = new LinkedList<>();    // 유일한 변화!!!
```

```
    // 컬렉션 인스턴스에 문자열 인스턴스 저장  
    list.add("Toy");  
    list.add("Box");  
    list.add("Robot");
```

리스트 기반 자료구조는 열차 칸을 더하고 빼는 형태의 자료구조이다.

인스턴스 저장

열차 칸을 하나 더한다.

인스턴스 삭제

해당 열차 칸을 삭제한다.

```
    // 저장된 문자열 인스턴스의 참조  
    for(int i = 0; i < list.size(); i++)  
        System.out.print(list.get(i) + '\t');  
    System.out.println();
```

```
    list.remove(0); // 첫 번째 인스턴스 삭제
```

```
    // 첫 번째 인스턴스 삭제 후 나머지 인스턴스들을 참조  
    for(int i = 0; i < list.size(); i++)  
        System.out.print(list.get(i) + '\t');  
    System.out.println();
```

```
}
```

# ArrayList<E> vs LinkedList<E>

## ArrayList<E>의 단점

- 저장 공간을 늘리는 과정에서 시간이 비교적 많이 소요된다.
- 인스턴스의 삭제 과정에서 많은 연산이 필요할 수 있다. 따라서 느릴 수 있다.

## ArrayList<E>의 장점

- 저장된 인스턴스의 참조가 빠르다.

## LinkedList<E>의 단점

- 저장된 인스턴스의 참조 과정이 배열에 비해 복잡하다. 따라서 느릴 수 있다.

## LinkedList<E>의 장점

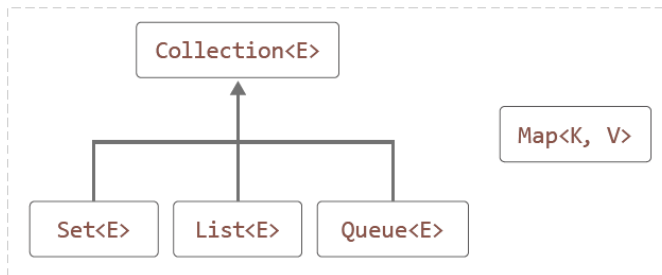
- 저장 공간을 늘리는 과정이 간단하다.
- 저장된 인스턴스의 삭제 과정이 단순하다.



# 저장된 인스턴스의 순차적 접근 방법 1: enhancec for문의 사용

```
public static void main(String[] args) {  
    List<String> list = new LinkedList<>();  
  
    // 인스턴스 저장  
    list.add("Toy");  
    list.add("Box");  
    list.add("Robot");  
  
    // 전체 인스턴스 참조  
    for(String s : list)  
        System.out.print(s + '\t');  
    . . . .  
}
```

for-each문의 대상이 되기 위한 조건  
**Iterable<T>** 인터페이스의 구현



```
public interface Collection<E> extends Iterable<E>
```

# 저장된 인스턴스의 순차적 접근 방법 2

```
public static void main(String[] args) {  
    List<String> list = new LinkedList<>();  
    ....  
    Iterator<String> itr = list.iterator();    // 반복자 획득, itr이 지팡이를 참조한다.  
    ....  
    // 반복자를 이용한 순차적 참조  
    while(itr.hasNext()) {    // next 메소드가 반환할 대상이 있다면,  
        str = itr.next();    // next 메소드를 호출한다.  
        ....  
    }  
}
```

```
public interface Iterable<T> {  
    Iterator<T> iterator();  
    . . . .  
}
```

# Iterator 반복자의 세 가지 메소드

E `next()`

다음 인스턴스의 참조 값을 반환

boolean `hasNext()`

`next` 메소드 호출 시 참조 값 반환 가능 여부 확인

void `remove()`

`next` 메소드 호출을 통해 반환했던 인스턴스 삭제

```
// 반복자를 이용한 참조 과정 중 인스턴스의 삭제
while(itr.hasNext()) {
    str = itr.next();
    if(str.equals("Box"))
        itr.remove(); // 위에서 next 메소드가 반환한 인스턴스 삭제
}
```

# 배열보다는 컬렉션 인스턴스가 좋다.: 컬렉션 변환

다음 두 가지 이유로 배열보다 `ArrayList<E>`가 더 좋다.

인스턴스의 저장과 삭제가 편하다.

반복자를 쓸 수 있다.

단, 배열처럼 선언과 동시에 초기화가 불가능하다. 그러나 다음 방법을 쓸 수 있다.

```
List<String> list = Arrays.asList("Toy", "Robot", "Box");
```

- 인자로 전달된 인스턴스들을 저장한 컬렉션 인스턴스의 생성 및 반환
- 이렇게 생성된 리스트 인스턴스는 `Immutable` 인스턴스이다.

# 배열보다는 컬렉션 인스턴스가 좋다.: 이어서

다음 생성자를 통해서 새로운 ArrayList 인스턴스 생성 가능

```
public ArrayList(Collection<? extends E> c) {...}
```

- Collection<E>를 구현한 컬렉션 인스턴스를 인자로 전달받는다.
- 그리고 E는 인스턴스 생성 과정에서 결정되므로 무엇이든 될 수 있다.
- 덧붙여서 매개변수 c로 전달된 컬렉션 인스턴스에서는 참조만(꺼내기만) 가능하다.

```
public static void main(String[] args) {
```

```
    List<String> list = Arrays.asList("Toy", "Box", "Robot", "Box");
```

```
    // 생성자 public ArrayList(Collection<? extends E> c)를 통한 인스턴스 생성
```

```
    list = new ArrayList<>(list);
```

```
    ....
```

```
}
```

# 배열 기반 리스트를 연결 기반 리스트로...

```
public ArrayList(Collection<? extends E> c)    // ArrayList<E> 생성자 중 하나
```

→ 인자로 전달된 컬렉션 인스턴스로부터 ArrayList<E> 인스턴스 생성

```
public LinkedList(Collection<? extends E> c)    // LinkedList<E> 생성자 중 하나
```

→ 인자로 전달된 인스턴스로부터 LinkedList<E> 인스턴스 생성

```
public static void main(String[] args) {  
    List<String> list = Arrays.asList("Toy", "Box", "Robot", "Box");  
    list = new ArrayList<>(list);  
    . . .  
    // ArrayList<E> 인스턴스 기반으로 LinkedList<E> 인스턴스 생성  
    list = new LinkedList<>(list);  
    . . .  
}
```

# 기본 자료형 데이터의 저장과 참조

```
public static void main(String[] args) {  
    LinkedList<Integer> list = new LinkedList<>();  
    list.add(10);    // 저장 과정에서 오토 박싱 진행  
    list.add(20);  
    list.add(30);
```

오토 박싱과 오토 언박싱 덕분에 컬렉션 인스턴스에  
기본 자료형의 값도 저장 가능하다.

```
    int n;  
    for(Iterator<Integer> itr = list.iterator(); itr.hasNext(); ) {  
        n = itr.next();    // 오토 언박싱 진행  
        System.out.print(n + "\t");  
    }  
    System.out.println();  
}
```

# 리스트만 갖는 양방향 반복자

```
public ListIterator<E> listIterator()    // List<E> 인터페이스의 메소드
```

→ ListIterator<E>는 Iterator<E>을 상속한다.

```
E next()
```

다음 인스턴스의 참조 값을 반환

```
boolean hasNext()
```

next 메소드 호출 시 참조 값 반환 가능 여부 확인

```
void remove()
```

next 메소드 호출을 통해 반환했던 인스턴스를 삭제

```
E previous()
```

next 메소드와 기능은 같고 방향만 반대

```
boolean hasPrevious()
```

hasNext 메소드와 기능은 같고 방향만 반대

```
void add(E e)
```

인스턴스의 추가

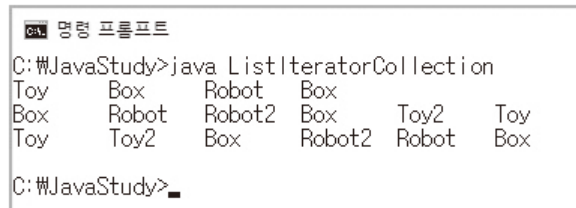
```
void set(E e)
```

인스턴스의 변경



# 양방향 반복자의 예

```
public static void main(String[] args) {  
    List<String> list = Arrays.asList("Toy", "Box", "Robot", "Box");  
    list = new ArrayList<>(list);  
  
    ListIterator<String> litr = list.listIterator(); // 양방향 반복자 획득  
  
    String str;  
    while(litr.hasNext()) { // 왼쪽에서 오른쪽으로 이동을 위한 반복문  
        str = litr.next();  
        System.out.print(str + '\t');  
        if(str.equals("Toy")) // "Toy" 만나면 "Toy2" 저장  
            litr.add("Toy2");  
    }  
    System.out.println();  
  
    while(litr.hasPrevious()) { // 오른쪽에서 왼쪽으로 이동을 위한 반복문  
        str = litr.previous();  
        System.out.print(str + '\t');  
        if(str.equals("Robot")) // "Robot" 만나면 "Robot2" 저장  
            litr.add("Robot2");  
    }  
    . . . .  
}
```

명령 프롬프트 창에서 실행된 Java 코드의 출력 결과. 첫 줄은 'C:\JavaStudy>java ListIteratorCollection'이다. 그 다음에 3줄의 출력 결과가 나오는데, 각각 'Toy Box Robot Box', 'Box Robot Robot2 Box Toy2 Toy', 'Toy Toy2 Box Robot2 Robot Box'이다. 마지막 줄은 'C:\JavaStudy>'로 끝난다.

```
C:\JavaStudy>java ListIteratorCollection  
Toy    Box    Robot  Box  
Box    Robot  Robot2 Box    Toy2    Toy  
Toy    Toy2   Box    Robot2 Robot  Box  
  
C:\JavaStudy>
```

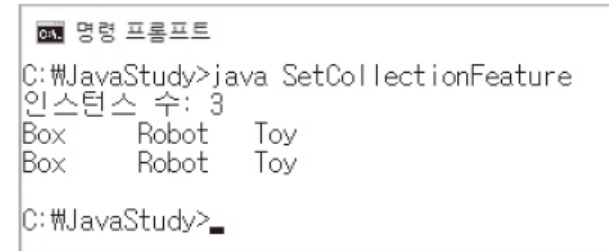
## 23-3. Set<E> 인터페이스를 구현하는 컬렉션 클래스들

# Set<E>을 구현하는 클래스의 특성과 HashSet<E> 클래스

Set<E> 인터페이스를 구현하는 제네릭 클래스들은 다음 두 가지 특성을 갖는다.

- 저장 순서가 유지되지 않는다.
- 데이터의 중복 저장을 허용하지 않는다.

```
public static void main(String[] args) {  
    Set<String> set = new HashSet<>();  
    set.add("Toy");    set.add("Box");  
    set.add("Robot");    set.add("Box");  
    System.out.println("인스턴스 수: " + set.size());  
  
    // 반복자를 이용한 전체 출력  
    for(Iterator<String> itr = set.iterator(); itr.hasNext(); )  
        System.out.print(itr.next() + '\t');  
    System.out.println();  
  
    // for-each문을 이용한 전체 출력  
    for(String s : set)  
        System.out.print(s + '\t');  
    System.out.println();  
}
```



```
CA 명령 프롬프트  
C:\JavaStudy>java SetCollectionFeature  
인스턴스 수: 3  
Box    Robot    Toy  
Box    Robot    Toy  
C:\JavaStudy>
```

출력 결과를 통해 동일 인스턴스가 저장되지 않음을 알 수 있다.

그렇다면 동일 인스턴스의 기준은?

# 동일 인스턴스에 대한 기준은?

```
public boolean equals(Object obj)
```

Object 클래스의 equals 메소드 호출 결과를 근거로 동일 인스턴스를 판단한다.

```
public int hashCode()
```

그런데 그에 앞서 Object 클래스의 hashCode 메소드 호출 결과가 같아야 한다.

정리하면,

두 인스턴스가 hashCode 메소드 호출 결과로 반환하는 값이 동일해야 한다.

그리고 이어서 두 인스턴스를 대상으로 equals 메소드의 호출 결과 true가 반환되면 동일 인스턴스로 간주한다.

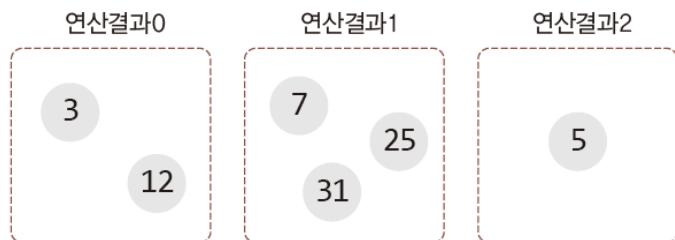
# 해수 알고리즘의 이해

분류 대상: 3, 5, 7, 12, 25, 31

적용 해쉬 알고리즘:  $\text{num} \% 3$

이렇듯 분류를 해 놓으면 탐색의 속도가 매우 빨라진다.  
즉 존재 유무 확인이 매우 빠르다.

분류 결과:



Object 클래스의 hashCode 메소드는 이렇듯 인스턴스들을 분류하는 역할을 한다.

# HashSet<E>의 인스턴스 동등 비교

- 탐색 1단계

Object 클래스에 정의된 hashCode 메소드의 반환 값을 기반으로 부류 결정

- 탐색 2단계

선택된 부류 내에서 equals 메소드를 호출하여 동등 비교

따라서 동등 비교의 과정에서 hashCode 메소드의 반환 값을 근거로 탐색의 대상이 확! 줄어든다.

# HashSet<E> 인스턴스에 저장할 클래스 정의 예

```
class Num {  
    private int num;  
    public Num(int n) { num = n; }  
  
    @Override  
    public String toString() { return String.valueOf(num); }  
  
    @Override  
    public int hashCode() {  
        return num % 3; // num의 값이 같으면 부류도 같다.  
    }  
  
    @Override  
    public boolean equals(Object obj) { // num의 값이 같으면 true 반환  
        if(num == ((Num)obj).num)  
            return true;  
        else  
            return false;  
    }  
}
```

# hashCode 메소드의 다양한 정의의 예

```
class Car {  
    private String model;  
    private String color;  
    . . . .  
  
    @Override  
    public int hashCode() {  
        return (model.hashCode() + color.hashCode()) / 2;  
    }  
    . . . .  
}
```

모든 인스턴스 변수의 정보를 다 반영하여 해쉬 값을 얻으려는 노력이 깃든 문장.  
결과적으로 더 세밀하게 나뉘고, 따라서 그만큼 탐색 속도가 높아진다.



# 해쉬 알고리즘 일일이 정의하기 조금 그렇다면...

```
public static int hash(Object...values)
```

→ java.util.Objects에 정의된 메소드, 전달된 인자 기반의 해쉬 값 반환

```
@Override
```

```
public int hashCode() {
```

```
    return Objects.hash(model, color);    // 전달인자 model, color 기반 해쉬 값 반환
```

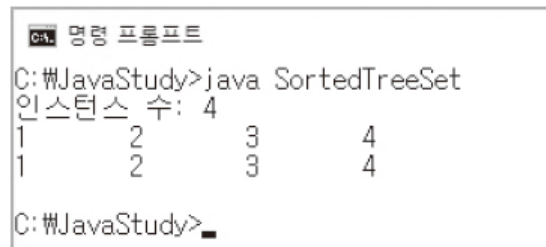
```
}        전달된 인자를 모두 반영한 해쉬 값을 반환한다.
```

# TreeSet<E> 클래스의 이해와 활용

Set<E> 인터페이스를 구현하는 TreeSet<E> 클래스

트리(Tree) 자료구조를 기반으로 인스턴스를 저장, 이는 정렬 상태가 유지되면서 인스턴스가 저장됨을 의미

```
public static void main(String[] args) {  
    TreeSet<Integer> tree = new TreeSet<Integer>();  
    tree.add(3); tree.add(1);  
    tree.add(2); tree.add(4);  
    System.out.println("인스턴스 수: " + tree.size());  
  
    // for-each문에 의한 반복  
    for(Integer n : tree)  
        System.out.print(n.toString() + '\t');  
    System.out.println();  
  
    // Iterator 반복자에 의한 반복  
    for(Iterator<Integer> itr = tree.iterator(); itr.hasNext(); )  
        System.out.print(itr.next().toString() + '\t');  
    System.out.println();  
}
```



```
C:\JavaStudy>java SortedTreeSet  
인스턴스 수: 4  
1      2      3      4  
1      2      3      4  
C:\JavaStudy>
```

반복자의 인스턴스 참조 순서는 오름차순을 기준으로 한다는 특징이 있다.

# TreeSet<E> 클래스의 오름차순 출력이란?

```
interface Comparable
```

```
→ int compareTo(Object o)
```

인자로 전달된 o가 작다면 양의 정수 반환

인자로 전달된 o가 크다면 음의 정수 반환

인자로 전달된 o와 같다면 0을 반환

```
interface Comparable<T>
```

```
→ int compareTo(T o)
```

인자로 전달된 o가 작다면 양의 정수 반환

인자로 전달된 o가 크다면 음의 정수 반환

인자로 전달된 o와 같다면 0을 반환

Chapter 20에서 설명한 내용

:Arrays 클래스의 sort 메소드 언급하면서 설명한 내용

제네릭 등장 이후로 추가된 인터페이스

# TreeSet 인스턴스에 저장될 것을 고려한 클래스의 예

```
class Person implements Comparable<Person> {  
    private String name;  
    private int age;  
    . . .  
    @Override  
    public int compareTo(Person p) {  
        return this.age - p.age;  
    }  
}
```

Comparable<T> 인터페이스의 구현 결과를 근거로 저장의 이뤄지고 또 참조가 진행이 된다.

따라서 TreeSet<T>에 저장할 인스턴스들은 모두 Comparable<T> 인터페이스를 구현한 클래스의 인스턴스이어야 함. 아니면 예외 발생!!!

# Comparator<T> 인터페이스 기반으로 TreeSet<E>의 정렬 기준제기하기

```
public interface Comparator<T>
```

→ `int compare(T o1, T o2)` 의 구현을 통해 정렬 기준을 결정할 수 있다.

- `o1`이 `o2`보다 크면 양의 정수 반환
- `o1`이 `o2`보다 작으면 음의 정수 반환
- `o1`과 `o2`가 같다면 `0` 반환

위 인터페이스를 구현한 클래스의 인스턴스를 `TreeSet<E>`의 다음 생성자를 통해 전달!

```
public TreeSet(Comparator<? super E> comparator)
```

# Comparator<T> 인터페이스 기반 TreeSet<E>의 예

```
class Person implements Comparable<Person> {
    String name;
    int age;
    . . .

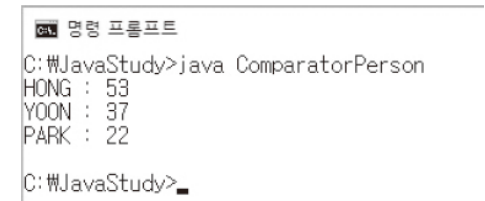
    @Override
    public int compareTo(Person p) {
        return this.age - p.age;
    }
}
```

- p1이 p2보다 크면 양의 정수 반환
- p1이 p2보다 작으면 음의 정수 반환
- p1과 p2가 같다면 0 반환

```
class PersonComparator implements Comparator<Person> {
    public int compare(Person p1, Person p2) {
        return p2.age - p1.age;
    }
}
```

```
public static void main(String[] args) {
    TreeSet<Person> tree = new TreeSet<>(new PersonComparator());
    tree.add(new Person("YOON", 37));
    tree.add(new Person("HONG", 53));
    tree.add(new Person("PARK", 22));

    for(Person p : tree)
        System.out.println(p);
}
```



```
명령 프롬프트
C:\JavaStudy>java ComparatorPerson
HONG : 53
YOON : 37
PARK : 22
C:\JavaStudy>
```

Person 클래스에 TreeSet을 위한 정렬 기준이

마련되어 있으나 Comparator 구현 인스턴스를 전달하여 새로운 기준을 제공!

# Comparator<T> 인터페이스 기반 TreeSet<E>의 예 하나 더

```
class StringComparator implements Comparator<String> {  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
}
```

```
public static void main(String[] args) {  
    TreeSet<String> tree = new TreeSet<>(new StringComparator());  
    tree.add("Box");  
    tree.add("Rabbit");  
    tree.add("Robot");  
}
```

String 클래스의 정렬 기준은 사전 편찬순이다.  
이를 길이 순으로 바꾸는 문장.

```
for(String s : tree)  
    System.out.print(s.toString() + '\t');  
System.out.println();
```

cmd 명령 프롬프트

```
C:\JavaStudy>java ComparatorString  
Box      Robot  Rabbit  
C:\JavaStudy>
```

# 중복된 인스턴스의 삭제!

```
public static void main(String[] args) {
    List<String> lst = Arrays.asList("Box", "Toy", "Box", "Toy");
    ArrayList<String> list = new ArrayList<>(lst);

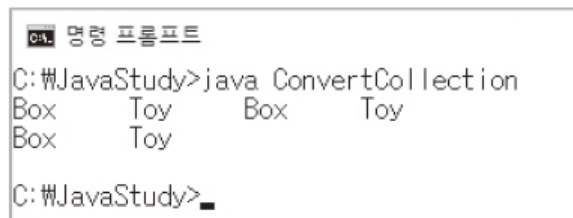
    for(String s : list)
        System.out.print(s.toString() + '\t');
    System.out.println();

    // 중복된 인스턴스를 걸러 내기 위한 작업
    HashSet<String> set = new HashSet<>(list);
    // 원래대로 ArrayList<String> 인스턴스로 저장물을 옮긴다.
    list = new ArrayList<>(set);

    for(String s : list)
        System.out.print(s.toString() + '\t');
    System.out.println();
}
```

**중복을 허용하는 리스트**  
**중복을 허용 않는 집합**

`public HashSet(Collection<? extends E> c)`  
→ 다른 컬렉션 인스턴스로부터 `HashSet<E>` 인스턴스 생성

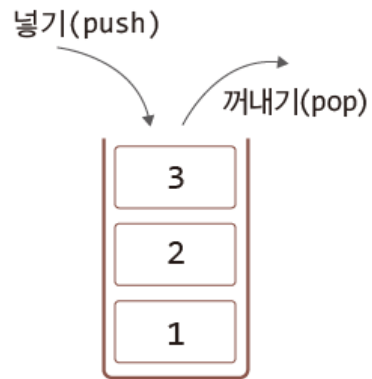


```
C:\JavaStudy>java ConvertCollection
Box Toy Box Toy
Box Toy
C:\JavaStudy>
```



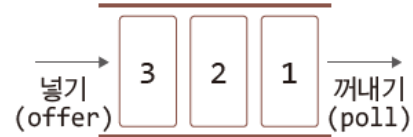
## 23-4. Queue<E> 인터페이스를 구현하는 컬렉션 클래스들

# 스택과 큐의 이해



LIFO(last-in-first-out)

→ 먼저 저장된 데이터가 마지막에 빠져나간다.



FIFO(first-in-first-out)

→ 먼저 저장된 데이터가 먼저 빠져나간다.

# 큐 인터페이스

## Queue<E> 인터페이스의 메소드들

`boolean add(E e)`

넣기

`E remove()`

꺼내기

`E element()`

확인하기

`boolean offer(E e)`

넣기, 넣을 공간이 부족하면 `false` 반환

`E poll()`

꺼내기, 꺼낼 대상 없으면 `null` 반환

`E peek()`

확인하기, 확인할 대상이 없으면 `null` 반환

# 큐의 구현

```
public static void main(String[] args) {  
    Queue<String> que = new LinkedList<>(); // LinkedList<E> 인스턴스 생성!  
    que.offer("Box");  
    que.offer("Toy");  
    que.offer("Robot");
```

LinkedList<E>는 List<E>와 동시에 Queue<E>를 구현하는 컬렉션 클래스이다.

따라서 어떠한 타입의 참조변수로 참조하느냐에 따라 '리스트'로도 '큐'로도 동작한다.

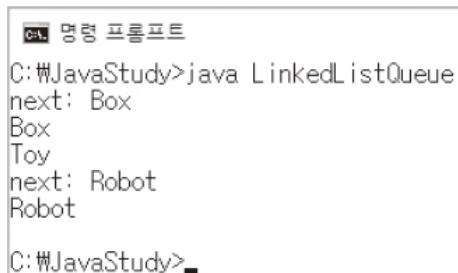
```
// 무엇이 다음에 나올지 확인  
System.out.println("next: " + que.peek());
```

```
// 첫 번째, 두 번째 인스턴스 꺼내기  
System.out.println(que.poll());  
System.out.println(que.poll());
```

```
// 무엇이 다음에 나올지 확인  
System.out.println("next: " + que.peek());
```

```
// 마지막 인스턴스 꺼내기  
System.out.println(que.poll());
```

```
}
```



```
명령 프롬프트  
C:\JavaStudy>java LinkedListQueue  
next: Box  
Box  
Toy  
next: Robot  
Robot  
C:\JavaStudy>
```

# 스택(Stack)의 구현

Deque을 기준으로 스택을 구현하는 것이 자바에서의 원칙!

Deque<E> 인터페이스의 메소드들

- 앞으로 넣고, 꺼내고, 확인하기

boolean **offerFirst**(E e)

E **pollFirst**()

E **peekFirst**()

넣기, 공간 부족하면 false 반환

꺼내기, 꺼낼 대상 없으면 null 반환

확인하기, 확인할 대상 없으면 null 반환

- 뒤로 넣고, 꺼내고, 확인하기

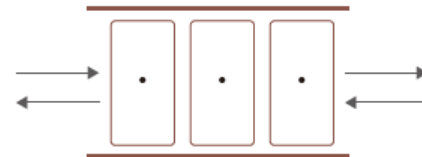
boolean **offerLast**(E e) 넣기, 공간이 부족하면 false 반환

E **pollLast**()

E **peekLast**()

꺼내기, 꺼낼 대상 없으면 null 반환

확인하기, 확인할 대상 없으면 null 반환



- 앞으로 넣고, 꺼내고, 확인하기

void **addFirst**(E e)      넣기

E **removeFirst**()      꺼내기

E **getFirst**()      확인하기

- 뒤로 넣고, 꺼내고, 확인하기

void **addLast**(E e)      넣기

E **removeLast**()      꺼내기

E **getLast**()      확인하기

# 스택의 예

```
public static void main(String[] args) {  
    Deque<String> deq = new ArrayDeque<>();  
  
    // 앞으로 넣고  
    deq.offerFirst("1.Box");  
    deq.offerFirst("2.Toy");  
    deq.offerFirst("3.Robot");  
  
    // 앞에서 꺼내기  
    System.out.println(deq.pollFirst());  
    System.out.println(deq.pollFirst());  
    System.out.println(deq.pollFirst());  
}
```



```
명령 프롬프트  
C:\JavaStudy>java ArrayDequeCollection  
3.Robot  
2.Toy  
1.Box  
C:\JavaStudy>
```

다음 문장도 구성 가능

```
Deque<String> deq = new LinkedList<>();
```

LinkedList<E>가 구현하는 인터페이스들  
Deque<E>, List<E>, Queue<E>

## 23-5. Map<K, V> 인터페이스를 구현하는 컬렉션 클래스들

# Key-Value 방식의 데이터 저장과 HashMap<K, V> 클래스

```
public static void main(String[] args) {  
    HashMap<Integer, String> map = new HashMap<>();  
  
    // Key-Value 기반 데이터 저장  
    map.put(45, "Brown");  
    map.put(37, "James");  
    map.put(23, "Martin");  
  
    // 데이터 탐색  
    System.out.println("23번: " + map.get(23));  
    System.out.println("37번: " + map.get(37));  
    System.out.println("45번: " + map.get(45));  
    System.out.println();  
  
    // 데이터 삭제  
    map.remove(37);  
  
    // 데이터 삭제 확인  
    System.out.println("37번: " + map.get(37));  
}
```

cmd 명령 프롬프트

```
C:\JavaStudy>java HashMapCollection  
23번: Martin  
37번: James  
45번: Brown  
  
37번: null  
C:\JavaStudy>_
```



# HashMap<K, V>의 순차적 접근 방법

HashMap<K, V> 클래스는 Iterable<T> 인터페이스를 구현하지 않으니 for-each문을 통해서, 혹은 ‘반복자’를 얻어서 순차적 접근을 진행할 수 없다.

대신 다음 메소드 호출을 통해서 key를 따로 모아 놓은 컬렉션 인스턴스를 얻을 수 있다. 그리고 이때 반환된 컬렉션 인스턴스를 대상으로 반복자를 얻을 수 있다.

```
public Set<K> keySet()
```

# HashMap<K, V>의 순차적 접근의 예

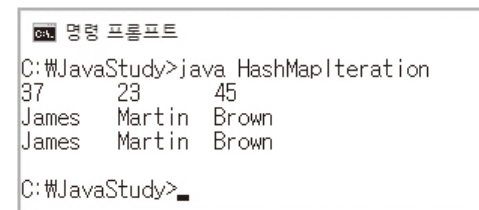
```
public static void main(String[] args) {
    HashMap<Integer, String> map = new HashMap<>();
    map.put(45, "Brown");
    map.put(37, "James");
    map.put(23, "Martin");

    // Key만 담고 있는 컬렉션 인스턴스 생성
    Set<Integer> ks = map.keySet();

    // 전체 Key 출력 (for-each문 기반)
    for(Integer n : ks)
        System.out.print(n.toString() + '\t');
    System.out.println();

    // 전체 Value 출력 (for-each문 기반)
    for(Integer n : ks)
        System.out.print(map.get(n).toString() + '\t');
    System.out.println();

    // 전체 Value 출력 (반복자 기반)
    for(Iterator<Integer> itr = ks.iterator(); itr.hasNext(); )
        System.out.print(map.get(itr.next()) + '\t');
    System.out.println();
}
```



```
명령 프롬프트
C:\WJavaStudy>java HashMapIteration
37      23      45
James   Martin  Brown
James   Martin  Brown
C:\WJavaStudy>
```

# TreeMap<K, V>의 순차적 접근의 예

```
public static void main(String[] args) {
    TreeMap<Integer, String> map = new TreeMap<>();
    map.put(45, "Brown");
    map.put(37, "James");
    map.put(23, "Martin");

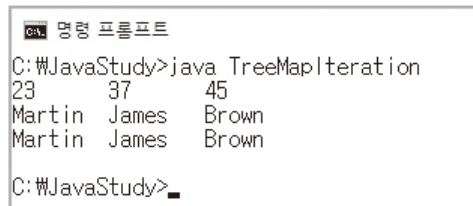
    // Key만 담고 있는 컬렉션 인스턴스 생성
    Set<Integer> ks = map.keySet();

    // 전체 Key 출력 (for-each문 기반)
    for(Integer n : ks)
        System.out.print(n.toString() + '\t');
    System.out.println();

    // 전체 Value 출력 (for-each문 기반)
    for(Integer n : ks)
        System.out.print(map.get(n).toString() + '\t');
    System.out.println();

    // 전체 Value 출력 (반복자 기반)
    for(Iterator<Integer> itr = ks.iterator(); itr.hasNext(); )
        System.out.print(map.get(itr.next()) + '\t');
    System.out.println();
}
```

Tree 자료구조의 특성상 반복자가 정렬된 순서대로 key들에 접근을 하고 있다. 이렇듯 반복자의 접근 순서는 컬렉션 인스턴스에 따라 달라질 수 있다.



```
C:\JavaStudy>java TreeMapIteration
23      37      45
Martin  James   Brown
Martin  James   Brown
C:\JavaStudy>
```