

# 14강 컬렉션 프레임워크

# 목차

- 컬렉션 프레임워크란?
- List
- Set
- hash
- 이진 트리구조
- Map
- 그외 컬렉션

## 컬렉션프레임워크란

- 데이터를 다루는 핵심 기술은

- 어떻게 저장할 것인가?
- 어떻게 꺼낼 쓸 것인가?
- 어떻게 변경할 것인가?
- 어떻게 삭제 할 것인가?

를 다루게 된다 이것을 CRUD라고 한다.

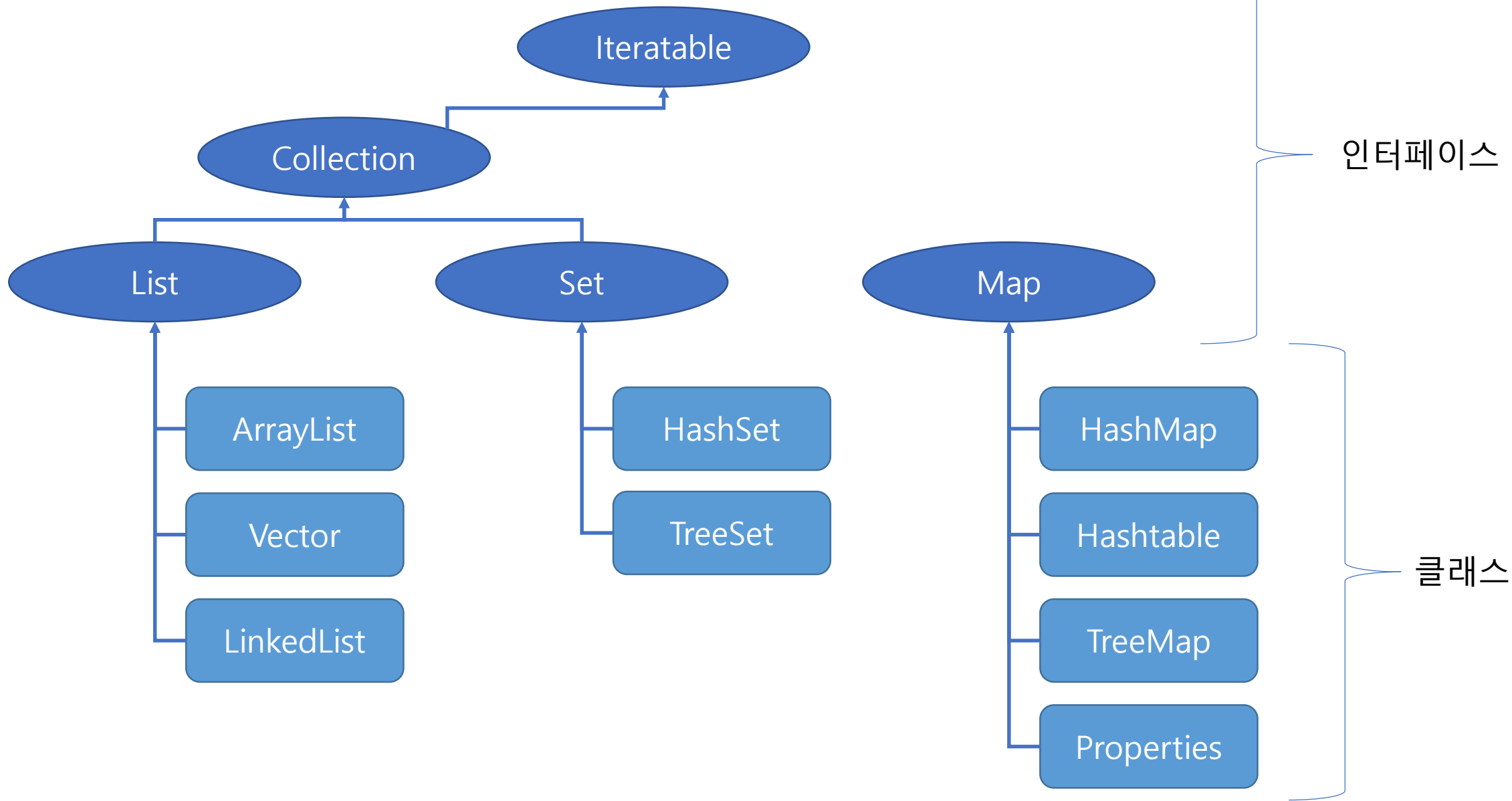
## 컬렉션프레임워크란

- 이미 우리는 자료를 적절하게 저장하기 위한 기술인 변수를 배웠다. 그리고 다수의 데이터를 한 변수에 담아서 관리하는 기술인 배열도 배웠다
- 그러나 배열에도 단점이 있다.
  - 길이의 제한
  - null => 중간에 데이터를 삭제하면 중간중간 데이터의 공백이 생긴다. 이럴 경우 새로운 데이터를 추가하고자 하면 매번 공백여부를 체크해야 하는 상황이 벌어진다.

## 컬렉션프레임워크란

- 데이터를 다루기 위한 핵심 기술은 소프트웨어 공학적으로 어느정도 완성이 되어있고 자바는 그러한 기술을 바탕으로 이미 적절한 클래스들이 제작되어 있다.
- 이러한 인터페이스와 클래스들의 집합체를 총칭하여 컬렉션 프레임워크라고 부른다.

# 컬렉션프레임워크란



## 컬렉션프레임워크란

- 주요 인터페이스로는 List, Set, Map등이 있다.
- 특히 같은 Collection 인터페이스로 묶여 있는 List와 Set은 내부 구조가 다르나 사용방법은 비슷하다.
  - List 특징
    - 순서가 존재, 중복데이터 허용
  - Set 특징
    - 순서가 없고, 중복데이터 불허

## List

- List는 객체를 일렬로 늘어놓은 구조를 가지고 있다.
- 리스트 내부에 데이터를 사용하는 메소드
  - 조회 : `get(idx)`; => 해당 인덱스의 데이터 조회
  - 추가 : `add(데이터)`; => 리스트 마지막에 데이터 넣기
  - 변경 : `set(idx,데이터)`; => 해당 인덱스의 데이터를 변경
  - 삭제 : `remove(idx)`; => 해당 인덱스의 데이터 삭제
- 추가 기능
  - `add(idx,데이터)` => 해당 인덱스에 데이터 끼워 넣기
  - `size()`; => 리스트 전체 길이 반환
  - `isEmpty()`;=> 리스트가 비어있는지 확인(참, 거짓)
  - `clear()`; => 객체 전체 삭제



## List

- 리스트를 구현받는 클래스 : ArrayList, LinkedList
- ArrayList 특징 : 배열의 구조를 가진다.
  - 인덱스 번호를 통해 빠른 조회가 가능하다.
  - 객체의 추가 삭제가 빈번한 경우 속도가 느려진다.
- LinkedList 특징 : 체인의 구조를 가진다.
  - 객체의 추가 삭제가 빈번해도 체인연결만 조절하면 되므로 속도가 빠르다.
  - 인덱스가 있지만 해당 위치를 찾기 위해서 처음부터 찾아가야 하므로 조회가 느리다.

# ArrayList

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

    list.add("고길동");
    list.add("김길동");
    list.add("박길동");
    list.add("홍길동");
    list.add("김길동");
    System.out.println("리스트의 길이 : "+list.size());
    System.out.println("-----");
    String name = list.get(2);
    System.out.println("2번째 객체 이름 : "+name);
    System.out.println("-----");
    for(int i=0;i<list.size();i++) {
        System.out.println(i+"번째 이름 : "+list.get(i));
    }
    System.out.println("리스트의 길이 : "+list.size());
    System.out.println("-----");
    list.add(3,"이길동");
    for(int i=0;i<list.size();i++) {
        System.out.println(i+"번째 이름 : "+list.get(i));
    }
    System.out.println("리스트의 길이 : "+list.size());
    System.out.println("-----");
    list.remove(1);
    for(int i=0;i<list.size();i++) {
        System.out.println(i+"번째 이름 : "+list.get(i));
    }
    System.out.println("리스트의 길이 : "+list.size());
    System.out.println("-----");
}
```

리스트의 길이 : 5

2번째 객체 이름 : 박길동

0번째 이름 : 고길동

1번째 이름 : 김길동

2번째 이름 : 박길동

3번째 이름 : 홍길동

4번째 이름 : 김길동

리스트의 길이 : 5

0번째 이름 : 고길동

1번째 이름 : 김길동

2번째 이름 : 박길동

3번째 이름 : 이길동

4번째 이름 : 홍길동

5번째 이름 : 김길동

리스트의 길이 : 6

0번째 이름 : 고길동

1번째 이름 : 박길동

2번째 이름 : 이길동

3번째 이름 : 홍길동

4번째 이름 : 김길동

리스트의 길이 : 5

# LinkedList

```
public static void main(String[] args) {  
    List<String> list = new LinkedList<>();  
  
    list.add("고길동");  
    list.add("김길동");  
    list.add("박길동");  
    list.add("홍길동");  
    list.add("김길동");  
    System.out.println("리스트의 길이 : "+list.size());  
    System.out.println("-----");  
    String name = list.get(2);  
    System.out.println("2번째 객체 이름 : "+name);  
    System.out.println("-----");  
    for(int i=0;i<list.size();i++) {  
        System.out.println(i+"번째 이름 : "+list.get(i));  
    }  
    System.out.println("리스트의 길이 : "+list.size());  
    System.out.println("-----");  
    list.add(3,"이길동");  
    for(int i=0;i<list.size();i++) {  
        System.out.println(i+"번째 이름 : "+list.get(i));  
    }  
    System.out.println("리스트의 길이 : "+list.size());  
    System.out.println("-----");  
    list.remove(1);  
    for(int i=0;i<list.size();i++) {  
        System.out.println(i+"번째 이름 : "+list.get(i));  
    }  
    System.out.println("리스트의 길이 : "+list.size());  
    System.out.println("-----");  
}
```

리스트의 길이 : 5

-----

2번째 객체 이름 : 박길동

-----

0번째 이름 : 고길동

1번째 이름 : 김길동

2번째 이름 : 박길동

3번째 이름 : 홍길동

4번째 이름 : 김길동

리스트의 길이 : 5

-----

0번째 이름 : 고길동

1번째 이름 : 김길동

2번째 이름 : 박길동

3번째 이름 : 이길동

4번째 이름 : 홍길동

5번째 이름 : 김길동

리스트의 길이 : 6

-----

0번째 이름 : 고길동

1번째 이름 : 박길동

2번째 이름 : 이길동

3번째 이름 : 홍길동

4번째 이름 : 김길동

리스트의 길이 : 5

-----

## Set

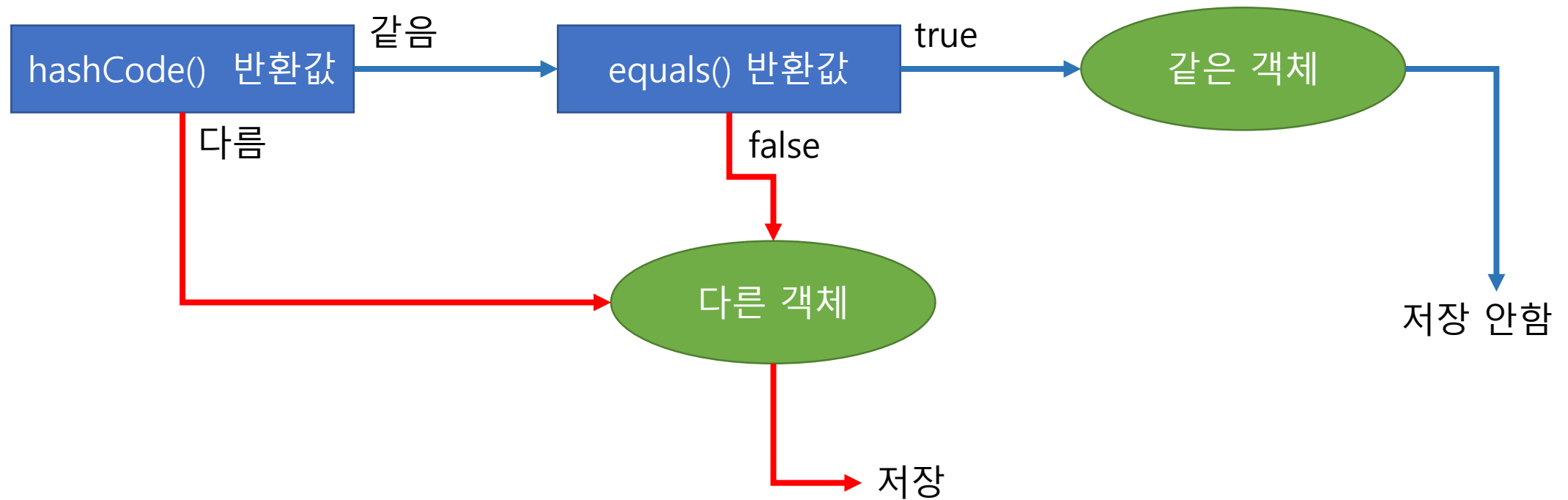
- Set은 집합의 형태를 가진다.
- Set 내부에 데이터를 사용하는 메소드
  - 추가 : `add(데이터);` => Set 마지막에 데이터 넣기
  - 삭제 : `remove(객체);` => 해당 데이터 삭제
  - 조회 => 전체 조회를 위한 반복자 사용
    - `iterator();`
- 추가 기능
  - `size();` => Set 전체 길이 반환
  - `isEmpty();` => Set이 비어있는 지 확인(참, 거짓)
  - `clear();` => 객체 전체 삭제

## Set

- Set을 구현받는 클래스 : HashSet, TreeSet
- HashSet 특징 : 객체가 가진 hashCode메소드와 equals 메소드를 통해서 중복을 방지
- TreeSet 특징 : 이진트리를 기반으로 한 구조를 가진다.
  - 객체가 크기순으로 정렬이 된다.
  - 크기 비교를 위해서 Comparable인터페이스를 직접 구현한 클래스나 Comparator인터페이스를 구현한 클래스를 도구삼아서 정렬이 가능하다.

# Set

- 중복 방지를 위한 검사방법



## Set

- 대부분 기본타입이나 String 클래스는 중복 제거를 위한 오버라이딩이 되어있음
- 사용자 정의 클래스도 중복을 제거하기 위해서는 hashCode와 equals메소드를 적절하게 오버라이딩할 필요가 있다.

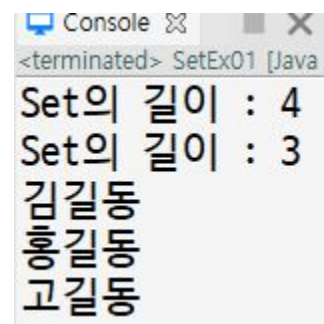
## Set

- Set은 순서없이 마구잡이로 소속되어 있으므로 일반적인 방법으로 데이터를 조회할 수 없다.
- 전체 데이터를 조회할 수밖에 없는데 이때 사용하는 객체가 반복자(iterator)이다.
- Set에 직접 조회는 안되지만 반복자를 통해 간접 접근 형태로 데이터를 조회한다.
  - 단, 반복자는 한번 꺼내 온 데이터를 다시 사용할 수 없으므로 데이터를 재 조회하기 위해서는 반복자를 다시 호출해야 한다.



## hashSet 예제

```
public static void main(String[] args) {  
    Set<String> set= new HashSet<>();  
  
    set.add("고길동");  
    set.add("김길동");  
    set.add("박길동");  
    set.add("홍길동");  
    set.add("김길동");  
  
    System.out.println("Set의 길이 : "+set.size());  
  
    set.remove("박길동");  
    System.out.println("Set의 길이 : "+set.size());  
  
    Iterator<String> itr = set.iterator();  
    while(itr.hasNext()) {  
        String str = itr.next();  
        System.out.println(str);  
    }  
}
```



```
Console  
<terminated> SetEx01 [Java]  
Set의 길이 : 4  
Set의 길이 : 3  
김길동  
홍길동  
고길동
```

## hashSet 예제

```
public static void main(String[] args) {
    Set<String> set= new HashSet<>();

    set.add("고길동");
    set.add("김길동");
    set.add("박길동");
    set.add("홍길동");
    set.add("김길동");

    Iterator<String> itr = set.iterator();
    while(itr.hasNext()) {
        String str = itr.next();
        System.out.println(str);
    }

    String str = itr.next();
    System.out.println(str);
}
```

반복자는 next를 통해서 한번 데이터를 조회하면 다시 뒤로 가서 데이터를 가져올 수 없다.

박길동  
김길동  
홍길동  
고길동

Exception in thread "main" [java.util.NoSuchElementException](#)  
at java.util.HashMap\$HashIterator.nextNode(Unknown Source)  
at java.util.HashMap\$KeyIterator.next(Unknown Source)  
at chapter14.SetEx01\_1.main([SetEx01\\_1.java:24](#))

# hashCode 예제

```
public class SetEx02 {  
  
    public static void main(String[] args) {  
        Set<Student> set = new HashSet<>();  
  
        set.add(new Student("고길동",16));  
        set.add(new Student("박길동",20));  
        set.add(new Student("김길동",13));  
        set.add(new Student("홍길동",16));  
        set.add(new Student("김길동",13));  
  
        System.out.println("Set의 길이 : "+set.size());  
  
        set.add(new Student("박길동",20));  
        System.out.println("Set의 길이 : "+set.size());  
    }  
}
```

```
Set의 길이 : 4  
Set의 길이 : 4
```

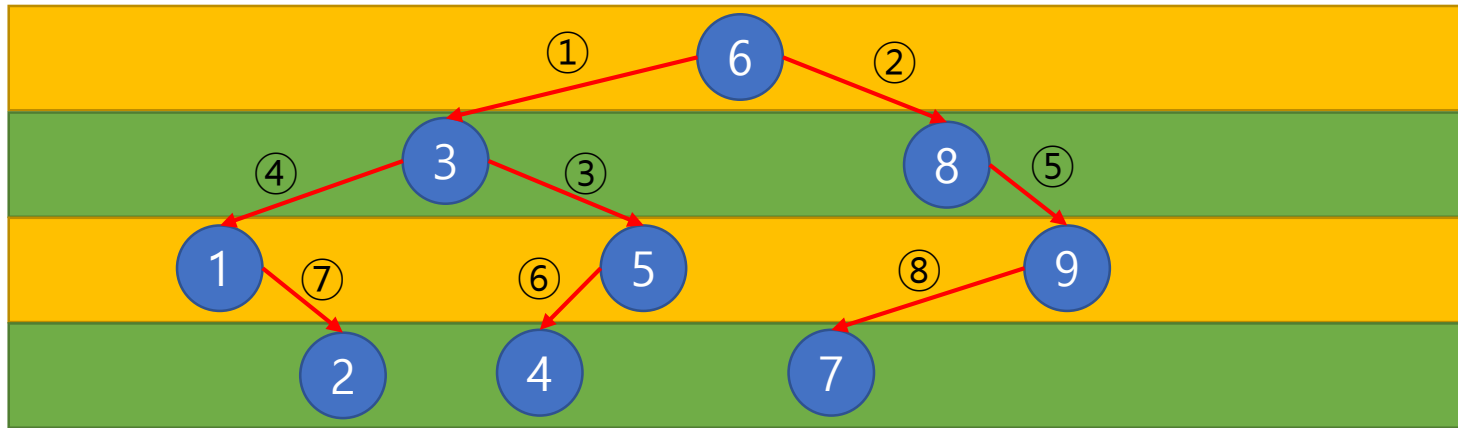
```
public class Student {  
    String name;  
    int StudentNumber;  
  
    public Student(String name, int studentNumber) {  
        this.name = name;  
        StudentNumber = studentNumber;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public int getStudentNumber() {  
        return StudentNumber;  
    }  
  
    public void setStudentNumber(int studentNumber) {  
        StudentNumber = studentNumber;  
    }  
  
    @Override  
    public boolean equals(Object obj) {  
        if(obj instanceof Student) {  
            Student stu = (Student)obj;  
            return (stu.getName().equals(this.name)) &&  
                (stu.getStudentNumber()==this.StudentNumber);  
        }else {  
            return false;  
        }  
    }  
  
    @Override  
    public int hashCode() {  
        return StudentNumber;  
    }  
}
```

# TreeSet

- 이진 트리구조
  - 기본적으로 크기 비교를 통해서 계층적으로 데이터를 배치하는 형태이다.
  - 자연적으로 데이터가 오름차순으로 정렬이 진행되며 데이터 검색 시 보다 빠르게 검색이 가능하다.

# TreeSet

- 이진 트리구조



## TreeSet

- TreeSet은 Set의 기본 구조를 가지므로 Set의 메소드와 반복자 등을 사용할 수 있다
- 추가로 TreeSet은 내부에 정렬된 구조를 가지므로 추가적인 메소드를 가지고 있다.
  - first() => 첫번째 객체 반환
  - last() => 마지막 객체 반환
  - lower(객체) => 주어진 객체보다 바로 아래 객체 반환
  - higher(객체) => 주어진 객체보다 바로 위 객체 반환
  - floor(객체) => 주어진 객체와 같은 객체가 있다면 반환  
없다면 바로 아래 객체 반환
  - ceiling(객체) => 주어진 객체와 같은 객체가 있다면 반환  
없다면 바로 위 객체 반환
  - pollFirst() => 첫번째 객체를 반환하고 Set에서 삭제
  - pollLast() => 마지막 객체를 반환하고 Set에서 삭제



# TreeSet

```
public static void main(String[] args) {  
    TreeSet<String> set = new TreeSet<>();  
  
    set.add("박길동");  
    set.add("고길동");  
    set.add("홍길동");  
    set.add("송길동");  
    set.add("이길동");  
    set.add("최길동");  
  
    System.out.println("첫번째 이름 : " + set.first());  
    System.out.println("마지막 이름 : " + set.last());  
    System.out.println("신길동 아래 이름 : " + set.lower("신길동"));  
    System.out.println("신길동 위 이름 : " + set.higher("신길동"));  
    System.out.println("최길동 이거나 바로 아래 이름 : " + set.floor("최길동"));  
    System.out.println("배길동 이거나 바로 위 이름 : " + set.ceiling("배길동"));  
  
    System.out.println();  
  
    Iterator<String> itr = set.iterator();  
    while(itr.hasNext()) {  
        System.out.println(itr.next());  
    }  
}
```

```
첫번째 이름 : 고길동  
마지막 이름 : 홍길동  
신길동 아래 이름 : 송길동  
신길동 위 이름 : 이길동  
최길동 이거나 바로 아래 이름 : 최길동  
배길동 이거나 바로 위 이름 : 송길동
```

```
고길동  
박길동  
송길동  
이길동  
최길동  
홍길동
```

## Map

- Map은 키와 값으로 이루어진 데이터 집합체(엔트리)를 저장하는 구조를 가진다.
- List의 특징과 Set의 특징을 모두 가지고 있다
  - 키는 중복이 될 수 없다.(set의 특징)
  - 키를 통해서 값을 조회하는 형태로 구성된다.(List의 특징)
- Map의 키도 중복된 데이터를 가질 수 없으므로 중복을 체크해야 한다.



# HashMap

- Map 내부에 데이터를 사용하는 메소드
  - 추가 : put(키,값); => 주어진 키로 값을 저장
  - 조회 : get(키); => 키를 통해 값을 조회
  - 삭제 : remove(키); => 해당 키에 속하는 값을 삭제
  - 변경 : put(키,값); => 주어진 키와 동일한 키가 있을 경우 새로운 값으로 대체
- 추가 기능
  - size(); => 전체 키의 개수를 반환
  - isEmpty(); => Map이 비어 있는지 확인(참, 거짓)
  - clear(); => 모든 엔트리 전체 삭제
  - entrySet(); => 키와 값의 쌍으로 구성된 Map.Entry 객체를 Set에 담아서 반환
  - keyset(); => 모든 키를 Set에 담아서 반환

# HashMap

```
public static void main(String[] args) {  
    Map<String,Integer> map = new HashMap<>();  
  
    map.put("박길동",78);  
    map.put("고길동",57);  
    map.put("홍길동",81);  
    map.put("송길동",66);  
    map.put("이길동",91);  
    map.put("최길동",57);  
  
    System.out.println("전체 map 개수 : "+map.size());  
  
    map.put("송길동",77);  
    System.out.println("전체 map 개수 : "+map.size());  
    System.out.println("송길동의 점수 : "+map.get("송길동"));  
  
    map.remove("이길동");  
    System.out.println("전체 map 개수 : "+map.size());  
}
```

```
<terminated> mapex01 Java Applicat  
전체 map 개수 : 6  
전체 map 개수 : 6  
송길동의 점수 : 77  
전체 map 개수 : 5
```

## HashMap

- Map은 Collection을 구현받지 않으므로 반복자(iterator)을 사용할 수 없다.
- 그러므로 Map 내부의 데이터를 한꺼번에 조회하려면 Set으로 전환후 반복자를 사용해야한다.
  - 모든 key만으로 Set을 구성하기 keyset
  - 모든 key와 값을 가진 Entry로 Set을 구성하기

# HashMap

```
public static void main(String[] args) {  
    Map<String,Integer> map = new HashMap<>();  
  
    map.put("박길동",78);  
    map.put("고길동",57);  
    map.put("홍길동",81);  
    map.put("송길동",66);  
    map.put("이길동",91);  
    map.put("최길동",57);  
  
    // keySet  
    Set<String> keySet = map.keySet();  
  
    Iterator<String> keyItr = keySet.iterator();  
    while(keyItr.hasNext()) {  
        String key = keyItr.next();  
        Integer value = map.get(key);  
  
        System.out.println("이름 : "+key+", 점수 : "+value);  
    }  
}
```

이름	: 박길동	, 점수	: 78
이름	: 홍길동	, 점수	: 81
이름	: 최길동	, 점수	: 57
이름	: 이길동	, 점수	: 91
이름	: 송길동	, 점수	: 66
이름	: 고길동	, 점수	: 57

# HashMap

```
public static void main(String[] args) {  
    Map<String,Integer> map = new HashMap<>();  
  
    map.put("박길동",78);  
    map.put("고길동",57);  
    map.put("홍길동",81);  
    map.put("송길동",66);  
    map.put("이길동",91);  
    map.put("최길동",57);  
  
    //EntrySet  
    Set<Map.Entry<String, Integer>> entrySet = map.entrySet();  
  
    Iterator<Map.Entry<String, Integer>> entryItr = entrySet.iterator();  
    while(entryItr.hasNext()) {  
        Map.Entry<String, Integer> entry = entryItr.next();  
  
        String key = entry.getKey();  
        Integer value = entry.getValue();  
  
        System.out.println("이름 : "+key+", 점수 :"+value);  
    }  
}
```

이름	:	박길동	,	점수	:	78
이름	:	홍길동	,	점수	:	81
이름	:	최길동	,	점수	:	57
이름	:	이길동	,	점수	:	91
이름	:	송길동	,	점수	:	66
이름	:	고길동	,	점수	:	57

## TreeMap

- TreeMap도 TreeSet과 마찬가지로 이진 트리구조를 가진다.
- key를 기준으로 정렬을 하되 Map.entry를 정렬 시킨다.

## TreeMap

- TreeMap도 TreeSet과 마찬가지로 이진 트리구조를 가진다.
- key를 기준으로 정렬을 하되 Map.entry를 정렬 시킨다.
- TreeMap은 내부에 정렬된 구조를 가지므로 추가적인 메소드를 가지고 있다.

## TreeMap

- TreeSet메소드에 Entry를 붙이면 된다.
  - firstEntry() => 첫번째 Map.Entry 반환
  - lastEntry () => 마지막 Map.Entry 반환
  - lowerEntry (key) => 주어진 키보다 바로 아래 Map.Entry 반환
  - higherEntry (key) => 주어진 키보다 바로 위 Map.Entry 반환
  - floorEntry (key) => 주어진 key와 같은 key가 있다면 Map.Entry 반환  
없다면 바로 아래 Map.Entry 반환
  - ceilingEntry (key) => 주어진 key와 같은 key가 있다면 Map.Entry 반환  
없다면 바로 위 Map.Entry 반환
  - pollFirstEntry () => 첫번째 Map.Entry를 반환하고 Map에서 삭제
  - pollLastEntry () => 마지막 Map.Entry를 반환하고 Map에서 삭제



# TreeMap

```
public static void main(String[] args) {
    TreeMap<String,Integer> map = new TreeMap<>();

    map.put("박길동",78);
    map.put("고길동",57);
    map.put("홍길동",81);
    map.put("송길동",66);
    map.put("이길동",91);
    map.put("최길동",57);

    System.out.println("첫번째 이름 : "+ map.firstEntry().getKey()
        +", 점수 : "+map.firstEntry().getValue());
    System.out.println("마지막 이름 : "+ map.lastEntry().getKey()
        +", 점수 : "+map.lastEntry().getValue());
    System.out.println("신길동 아래 이름 : "+ map.lowerEntry("신길동").getKey()
        +", 점수 : "+map.lowerEntry("신길동").getValue());
    System.out.println("신길동 위 이름 : "+ map.higherEntry("신길동").getKey()
        +", 점수 : "+map.higherEntry("신길동").getValue());
    System.out.println("최길동 이거나 바로 아래 이름 : "+ map.floorEntry("최길동").getKey()
        +", 점수 : "+map.floorEntry("최길동").getValue());
    System.out.println("배길동 이거나 바로 위 이름 : "+ map.ceilingEntry("배길동").getKey()
        +", 점수 : "+map.ceilingEntry("배길동").getValue());
    System.out.println();
    Set<String> keySet = map.keySet();

    Iterator<String> keyItr = keySet.iterator();
    while(keyItr.hasNext()) {
        String key = keyItr.next();
        Integer value = map.get(key);

        System.out.println("이름 : "+key+", 점수 : "+value);
    }
}
```

```
첫번째 이름 : 고길동, 점수 : 57
마지막 이름 : 홍길동, 점수 : 81
신길동 아래 이름 : 송길동, 점수 : 66
신길동 위 이름 : 이길동, 점수 : 91
최길동 이거나 바로 아래 이름 : 최길동, 점수 : 57
배길동 이거나 바로 위 이름 : 송길동, 점수 : 66
```

```
이름 : 고길동, 점수 :57
이름 : 박길동, 점수 :78
이름 : 송길동, 점수 :66
이름 : 이길동, 점수 :91
이름 : 최길동, 점수 :57
이름 : 홍길동, 점수 :81
```

## 객체의 정렬

- 숫자는 크기 순으로 정렬이 된다
- 문자열도 역시 크기순으로 정렬이 된다.
- 그런데 우리가 임의로 만든 객체는 어떤 기준으로 정렬이 되는 것일까?
- 객체의 비교 기준을 지정해 주어야 한다.

## 객체의 정렬

- 객체의 비교 기준을 지정하는 방법
  - 1) 비교를 직접적으로 할 클래스에게 비교 메소드를 만들어주는법
  - 2) 제 3의 클래스에게 비교 메소드를 만들어 주는법

## 객체의 정렬

- 1) 비교를 직접적으로 할 클래스에게 비교 메소드를 만들어주는 법
  - comparable 인터페이스를 구현한 후 compareTo()메소드를 오버라이딩한다.
  - A.compareTo(B) :
    - B가 작다면 양수 반환(A가 크다면)
    - B가 크다면 음수 반환(A가 작다면)
    - A와 B가 같다면 0을 반환

## 객체의 정렬

- 1) 비교를 직접적으로 할 클래스에게 비교 메소드를 만들어주는 법

- 예제

```
public class Coffee implements Comparable<Coffee>{
    private String name;
    private int price;

    public Coffee(String name, int price) {
        this.name = name;
        this.price = price;
    }

    public void prt() {
        System.out.println(name+" : "+price);
    }

    @Override
    public int compareTo(Coffee o) {
        if(this.price < o.getPrice()) {
            return -1;
        }else if(this.price > o.getPrice()){
            return 1;
        }else {
            return 0;
        }
    }

    public int getPrice() {
        return price;
    }
}
```

```
public static void main(String[] args) {
    TreeSet<Coffee> tree = new TreeSet<>();

    tree.add(new Coffee("카라멜 프라푸치노",5300));
    tree.add(new Coffee("카페라떼",4100));
    tree.add(new Coffee("카페 아메리카노",3600));
    tree.add(new Coffee("에스프레소 칩",5400));
    tree.add(new Coffee("바닐라라떼",4600));

    Iterator<Coffee> itr = tree.iterator();
    while(itr.hasNext()) {
        itr.next().prt();
    }
}
```

```
카페 아메리카노 : 3600
카페라떼 : 4100
바닐라라떼 : 4600
카라멜 프라푸치노 : 5300
에스프레소 칩 : 5400
```

## 객체의 정렬

- 2) 제 3의 클래스에게 비교 메소드를 만들어 주는 법
  - 비교에 사용할 제3의 클래스에 Comparator 인터페이스를 구현한 후 compare 메소드를 오버라이딩한다.
  - compare(A a, B b) :
    - $a > b$ 라면 양수를 반환
    - $a < b$ 라면 음수를 반환
    - $a == b$ 라면 0을 반환
  - TreeSet 인스턴스를 생성할 때 위 클래스의 인스턴스를 매개로 제공한다.

## 객체의 정렬

- 2) 제 3의 클래스에게 비교 메소드를 만들어 주는 법
  - 예제

```
public class PokeMon {
    private String name;
    private int level;

    public PokeMon(String name, int level) {
        this.name = name;
        this.level = level;
    }
    public void prt() {
        System.out.println(name+" : "+level);
    }
    public int getLevel() {
        return level;
    }
    public void setLevel(int level) {
        this.level = level;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

```
public class Ranking implements Comparator<PokeMon>{

    @Override
    public int compare(PokeMon o1, PokeMon o2) {
        if(o1.getLevel() > o2.getLevel()) {
            return 1;
        }else if(o1.getLevel() < o2.getLevel()) {
            return -1;
        }else {
            return 0;
        }
    }
}
```



## 객체의 정렬

- 2) 제 3의 클래스에게 비교 메소드를 만들어 주는 법
  - 예제

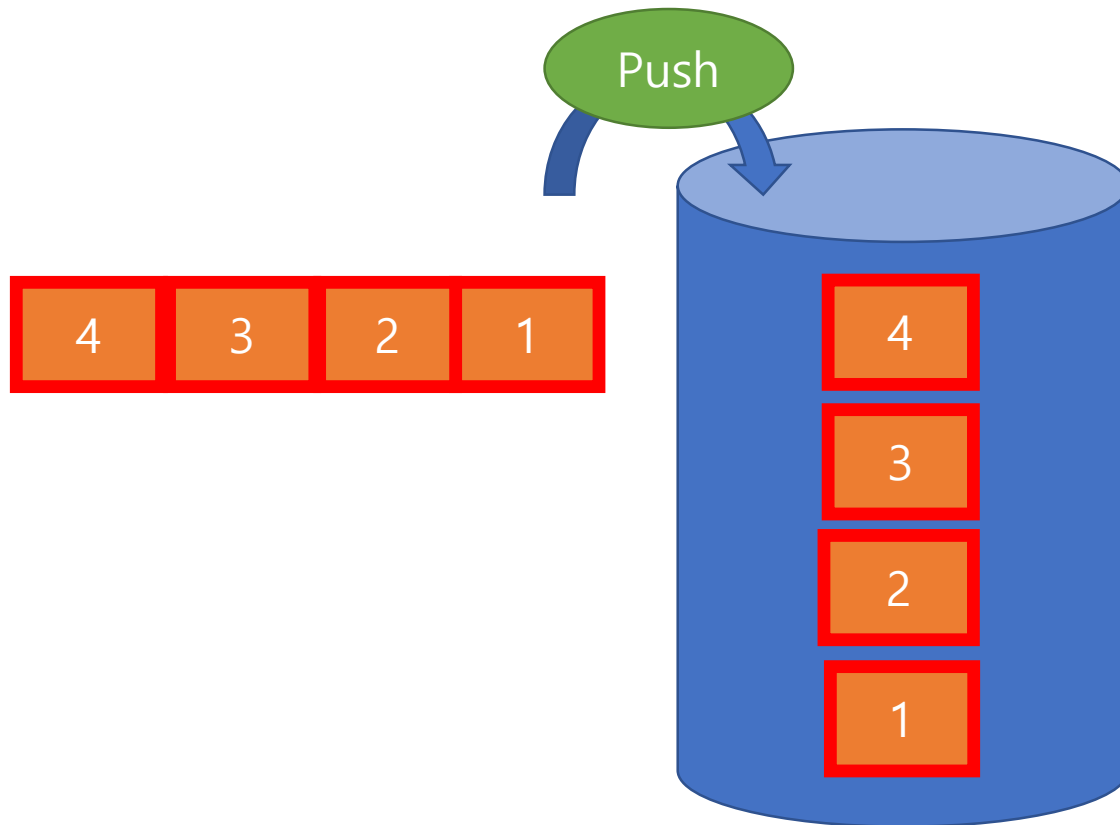
```
public static void main(String[] args) {  
    TreeSet<PokeMon> tree = new TreeSet<>(new Ranking());  
  
    tree.add(new PokeMon("야도란",14));  
    tree.add(new PokeMon("잠만보",21));  
    tree.add(new PokeMon("파이리",17));  
    tree.add(new PokeMon("피카츄",29));  
    tree.add(new PokeMon("파오리",11));  
    tree.add(new PokeMon("꼬부기",25));  
  
    Iterator<PokeMon> itr = tree.iterator();  
    while(itr.hasNext()) {  
        itr.next().prt();  
    }  
}
```

```
파오리 : 11  
야도란 : 14  
파이리 : 17  
잠만보 : 21  
꼬부기 : 25  
피카츄 : 29
```



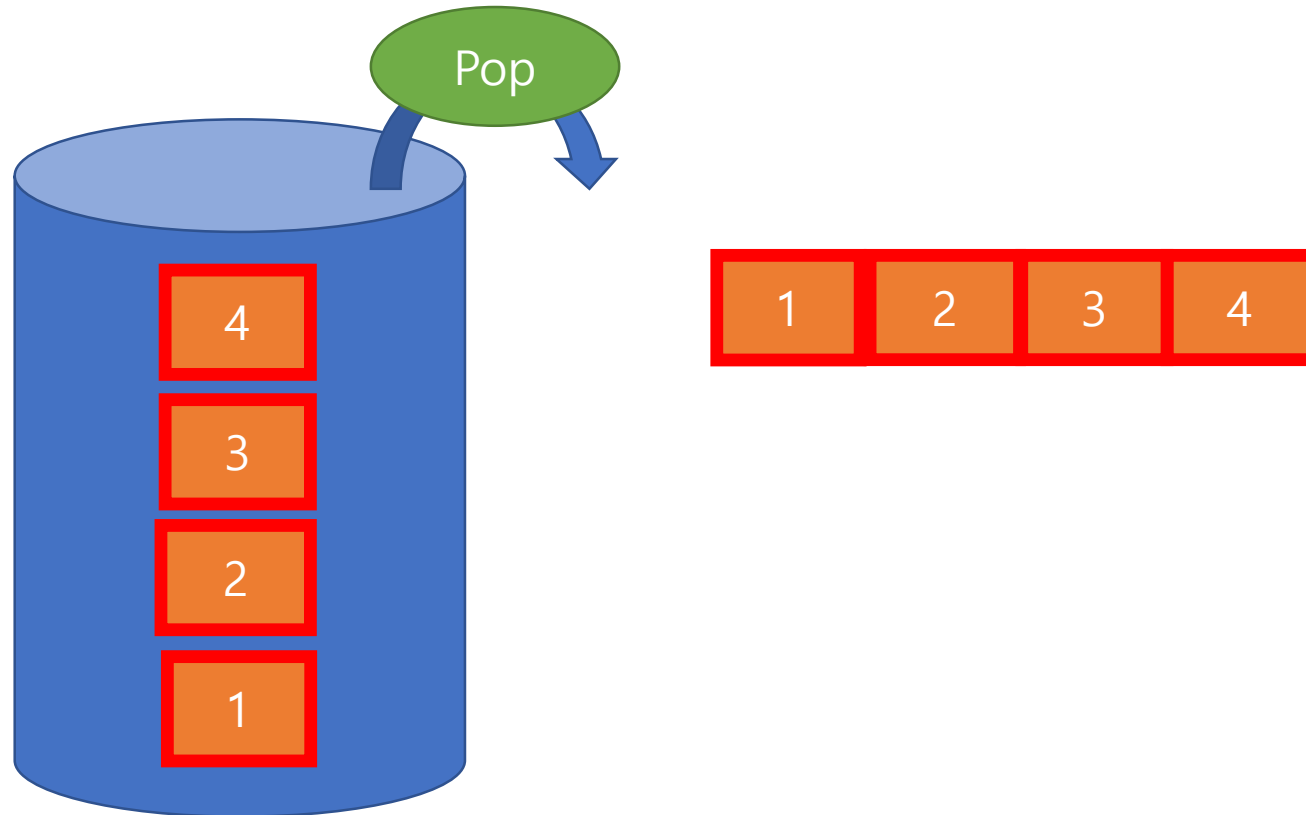
# Stack

- Stack 구조는 LIFO 자료구조를 구현한 클래스



# Stack

- Stack 구조는 LIFO 자료구조를 구현한 클래스



## Stack

- `push(객체);` => 데이터 추가 메소드
- `isEmpty();` => 스택 내부가 비었는가 체크하는 메소드
- `pop();` => 스택의 마지막의 데이터를 가져오는 메소드(삭제)
- `peek();` => 스택의 마지막의 데이터를 가져오는 메소드(미삭제)

# Stack

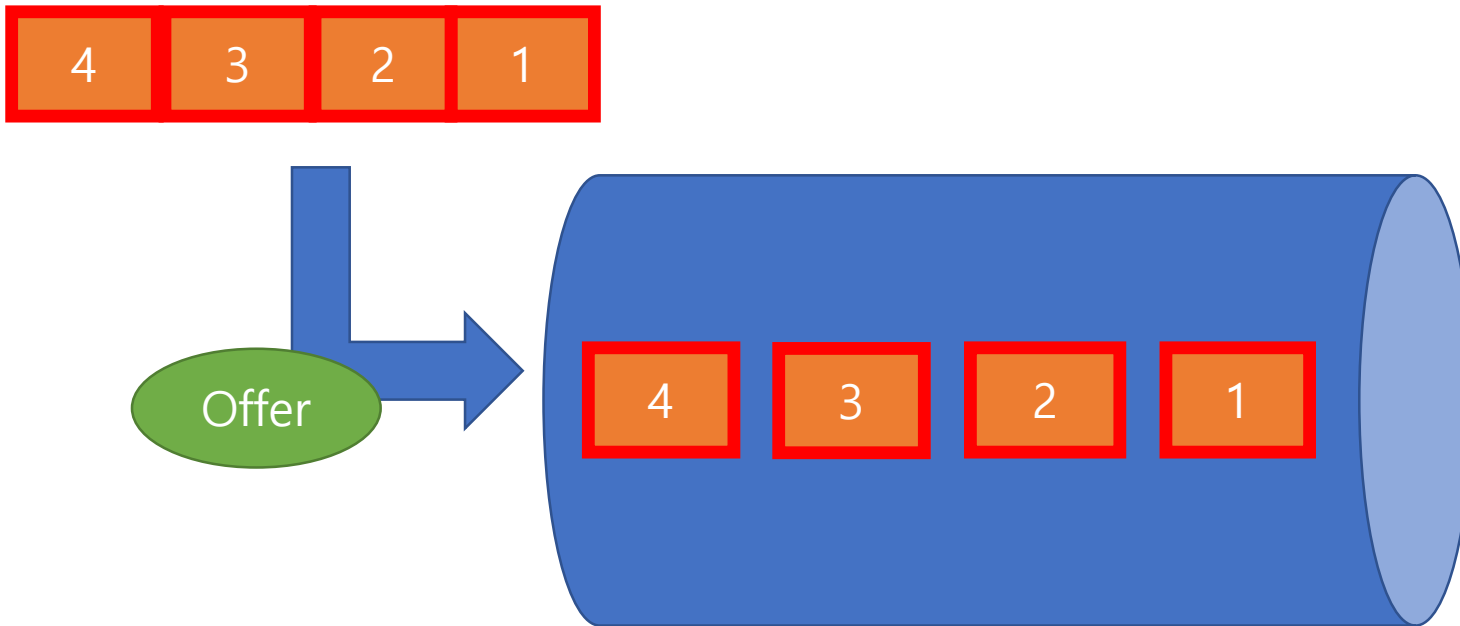
- 예제

```
public static void main(String[] args) {  
    Stack<String> stack = new Stack<>();  
  
    stack.push("A");  
    System.out.println(stack.peek());  
    stack.push("B");  
    System.out.println(stack.peek());  
    stack.push("C");  
    System.out.println(stack.peek());  
    stack.push("D");  
    System.out.println(stack.peek());  
    stack.push("E");  
    System.out.println(stack.peek());  
  
    System.out.println("-----");  
    while(!stack.isEmpty()) {  
        System.out.println(stack.pop());  
    }  
}
```

The diagram illustrates the stack's behavior. It shows a vertical container with a dashed horizontal line representing the stack's boundary. Above the line, the elements A, B, C, D, and E are listed from top to bottom, representing the push sequence. Below the line, the elements E, D, C, B, and A are listed from top to bottom, representing the pop sequence. This demonstrates the Last In, First Out (LIFO) principle of a stack.

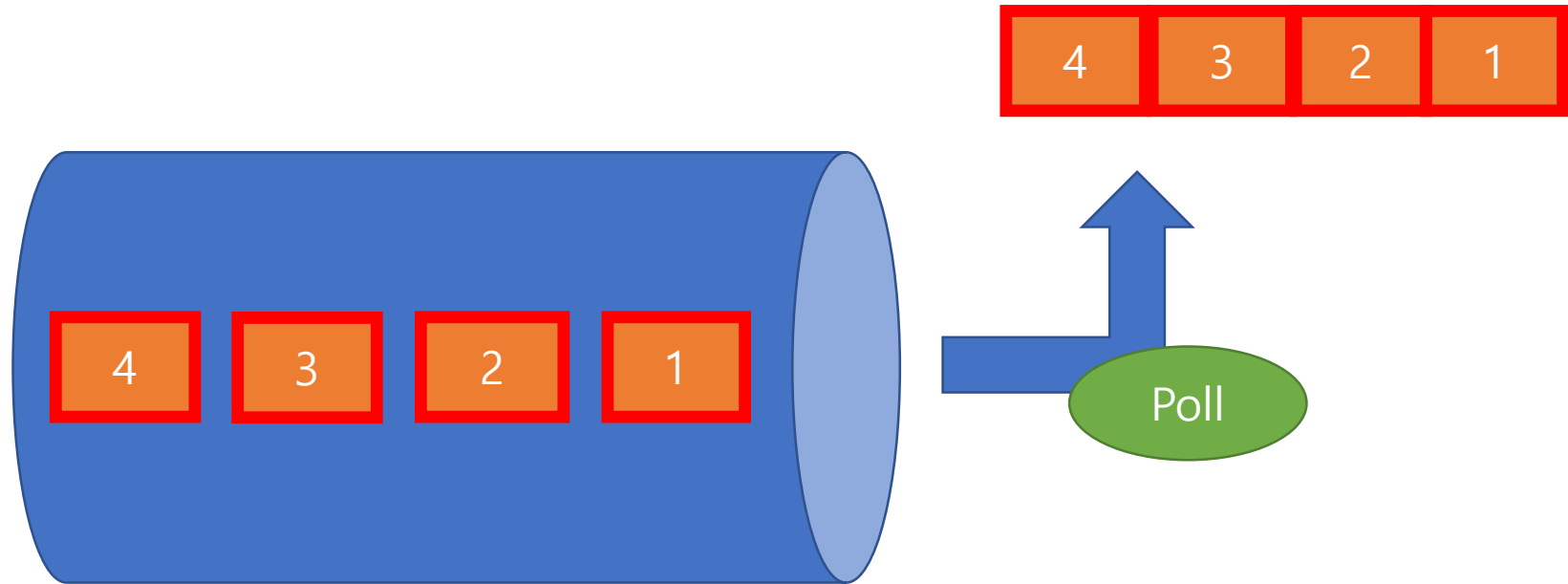
## Queue

- Queue구조는 FIFO 자료구조에서 사용되는 메소드를 정의한다.
  - Queue 인터페이스를 구현한 대표적인 클래스가 LinkedList이다.



# Queue

- Queue구조는 FIFO 자료구조에서 사용되는 메소드를 정의한다.
  - Queue 인터페이스를 구현한 대표적인 클래스가 LinkedList이다.



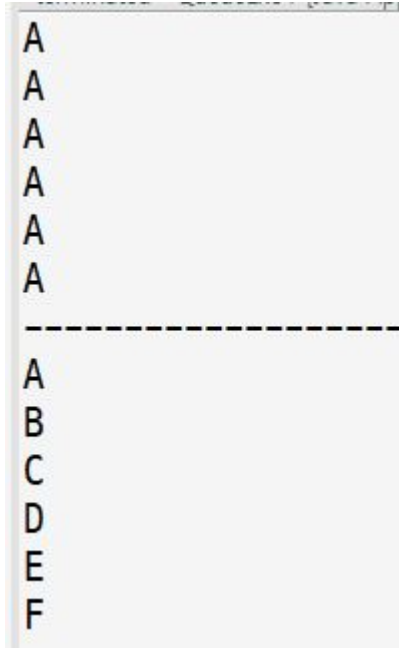
## Queue

- offer(객체); => 데이터 추가 메소드
- isEmpty(); => 스택 내부가 비었는가 체크하는 메소드
- poll(); => 큐의 첫번째의 데이터를 가져오는 메소드(삭제)
- peek(); => 큐에서의 첫번째의 데이터를 가져오는 메소드(미삭제)

# Queue

- 예제

```
public static void main(String[] args) {  
    Queue<String> queue = new LinkedList<>();  
  
    queue.offer("A");  
    System.out.println(queue.peek());  
    queue.offer("B");  
    System.out.println(queue.peek());  
    queue.offer("C");  
    System.out.println(queue.peek());  
    queue.offer("D");  
    System.out.println(queue.peek());  
    queue.offer("E");  
    System.out.println(queue.peek());  
    queue.offer("F");  
    System.out.println(queue.peek());  
  
    System.out.println("-----");  
  
    while(!queue.isEmpty()) {  
        System.out.println(queue.poll());  
    }  
}
```



```
A  
A  
A  
A  
A  
A  
-----  
A  
B  
C  
D  
E  
F
```