10. 컬렉션 프레임워크 - 순회, 정렬, 전체 정리

#1.인강/0.자바/4.자바-중급2편

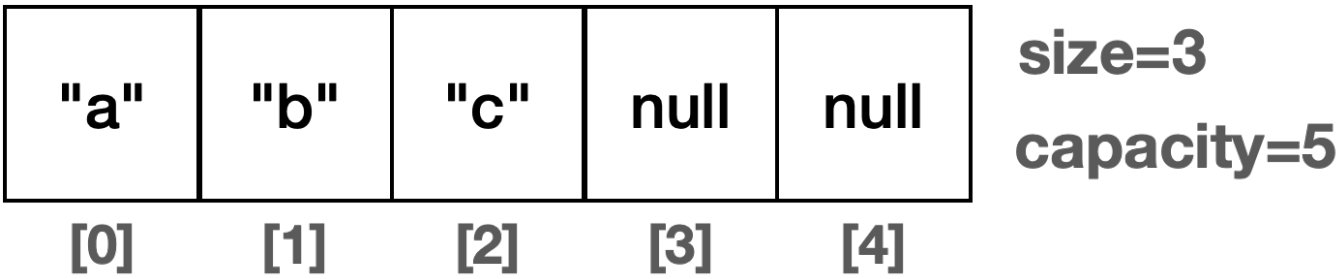
순회1 - 직접 구현하는 Iterable, Iterator

순회라는 단어는 여러 곳을 돌아다닌다는 뜻이다.

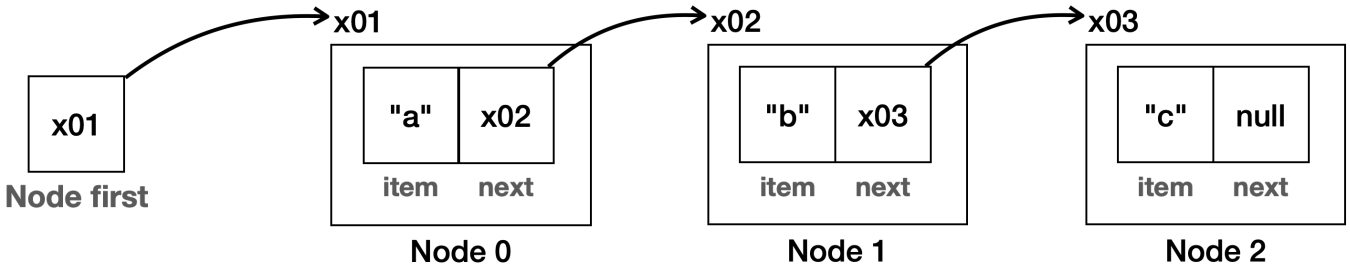
자료 구조에 순회는 자료 구조에 들어있는 데이터를 차례대로 접근해서 처리하는 것을 순회라 한다.

그런데 다양한 자료 구조가 있고, 각각의 자료 구조마다 데이터를 접근하는 방법이 모두 다르다.

배열 리스트



연결 리스트



예를 들어서 배열 리스트는 `index`를 `size`까지 차례로 증가하면서 순회해야 하고, 연결 리스트는 `node.next`를 사용해서 `node`의 끝이 `null` 일 때 까지 순회해야 한다. 이렇듯 각 자료 구조의 순회 방법이 서로 다르다.

배열 리스트, 연결 리스트, 해시 셋, 연결 해시 셋, 트리 셋 등등 다양한 자료 구조가 있다. 각각의 자료 구조마다 순회하는 방법이 서로 다르기 때문에, 각 자료 구조의 순회 방법을 배워야 한다. 그리고 순회 방법을 배우려면 자료 구조의 내부 구조도 알아야 한다. 결과적으로 너무 많은 내용을 알아야 하는 것이다. 하지만 자료 구조를 사용하는 개발자 입장에서 보면 단순히 자료 구조에 들어있는 모든 데이터에 순서대로 접근해서 출력하거나 계산하고 싶을 뿐이다.

자료 구조의 구현과 관계 없이 모든 자료 구조를 동일한 방법으로 순회할 수 있는 일관성 있는 방법이 있다면, 자료 구조를 사용하는 개발자 입장에서 매우 편리할 것이다.

자바는 이런 문제를 해결하기 위해 `Iterable`과 `Iterator` 인터페이스를 제공한다.

Iterable, Iterator

`Iterable`: "반복 가능한"이라는 뜻이다.
`Iterator`: "반복자"라는 뜻이다.

Iterable 인터페이스의 주요 메서드

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

- 단순히 `Iterator` 반복자를 반환한다.

Iterator 인터페이스의 주요 메서드

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
}
```

- `hasNext()`: 다음 요소가 있는지 확인한다. 다음 요소가 없으면 `false`를 반환한다.
- `next()`: 다음 요소를 반환한다. 내부에 있는 위치를 다음으로 이동한다.

자료 구조에 들어있는 데이터를 처음부터 끝까지 순회하는 방법은 단순하다. 자료 구조에 다음 요소가 있는지 물어보고, 있으면 다음 요소를 꺼내는 과정을 반복하면 된다. 만약 다음 요소가 없다면 종료하면 된다. 이렇게 하면 자료 구조에 있는 모든 데이터를 순회할 수 있다.

`Iterable`, `Iterator`를 사용하는 자료 구조를 하나 만들어보자. 둘 다 인터페이스여서 구현체가 필요하다.

먼저 `Iterator`의 구현체를 만들자.

```
package collection.iterable;

import java.util.Iterator;

public class MyArrayIterator implements Iterator<Integer> {

    private int currentIndex = -1;
    private int[] targetArr;

    public MyArrayIterator(int[] targetArr) {
        this.targetArr = targetArr;
    }

    @Override
    public boolean hasNext() {
        return currentIndex < targetArr.length - 1;
    }

    @Override
    public Integer next() {
        return targetArr[++currentIndex];
    }
}
```

- 생성자를 통해 반복자가 사용할 배열을 참조한다. 여기서 참조한 배열을 순회할 것이다.
- `currentIndex`: 현재 인덱스, `next()`를 호출할 때마다 하나씩 증가한다.
- `hasNext()`: 다음 항목이 있는지 검사한다. 배열의 끝에 다다르면 순회가 끝났으므로 `false`를 반환한다.
  - 참고로 인덱스의 길이는 `0`부터 시작하므로 배열의 길이에 1을 빼야 마지막 인덱스가 나온다.
- `next()`: 다음 항목을 반환한다.
  - `currentIndex`를 하나 증가하고 항목을 반환한다.
  - 인덱스는 `0`부터 시작하기 때문에 `currentIndex`는 처음에는 `-1`을 가진다. 이렇게 하면 다음 항목을 조회했을 때 `0`이 된다. 따라서 처음 `next()`를 호출하면 `0`번 인덱스를 가리킨다.

`Iterator`는 단독으로 사용할 수 없다. `Iterator`를 통해 순회의 대상이 되는 자료 구조를 만들어보자.
여기서는 매우 간단한 자료 구조를 하나 만들자. 내부에는 숫자 배열을 보관한다.

```java
package collection.iterable;

import java.util.Iterator;

public class MyArray implements Iterable<Integer> {

    private int[] numbers;

    public MyArray(int[] numbers) {
        this.numbers = numbers;
    }

    @Override
    public Iterator<Integer> iterator() {
        return new MyArrayIterator(numbers);
    }

}
```

- 배열을 가지는 매우 단순한 자료 구조이다.
- `Iterable` 인터페이스를 구현한다.
  - 이 인터페이스는 이 자료 구조에 사용할 반복자(`Iterator`)를 반환하면 된다.
  - 앞서 만든 반복자인 `MyArrayIterator`를 반환한다.
  - 이때 `MyArrayIterator`는 생상자를 통해 `MyArray`의 내부 배열인 `numbers`를 참조한다.

```java
package collection.iterable;

import java.util.Iterator;

public class MyArrayMain {

    public static void main(String[] args) {
        MyArray myArray = new MyArray(new int[]{1, 2, 3, 4});

        Iterator<Integer> iterator = myArray.iterator();
        System.out.println("iterator 사용");
        while (iterator.hasNext()) {
            Integer value = iterator.next();
            System.out.println("value = " + value);
        }
```

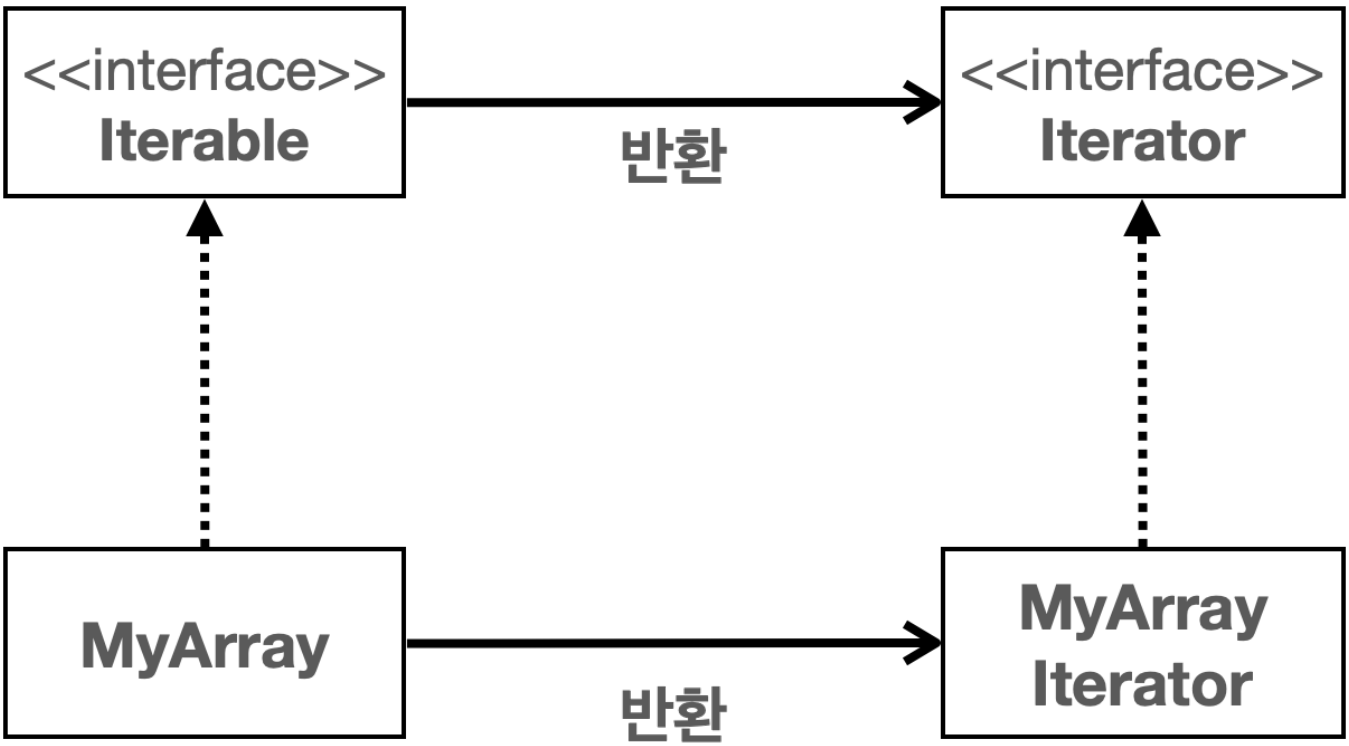
```
    }

}
```

실행 결과

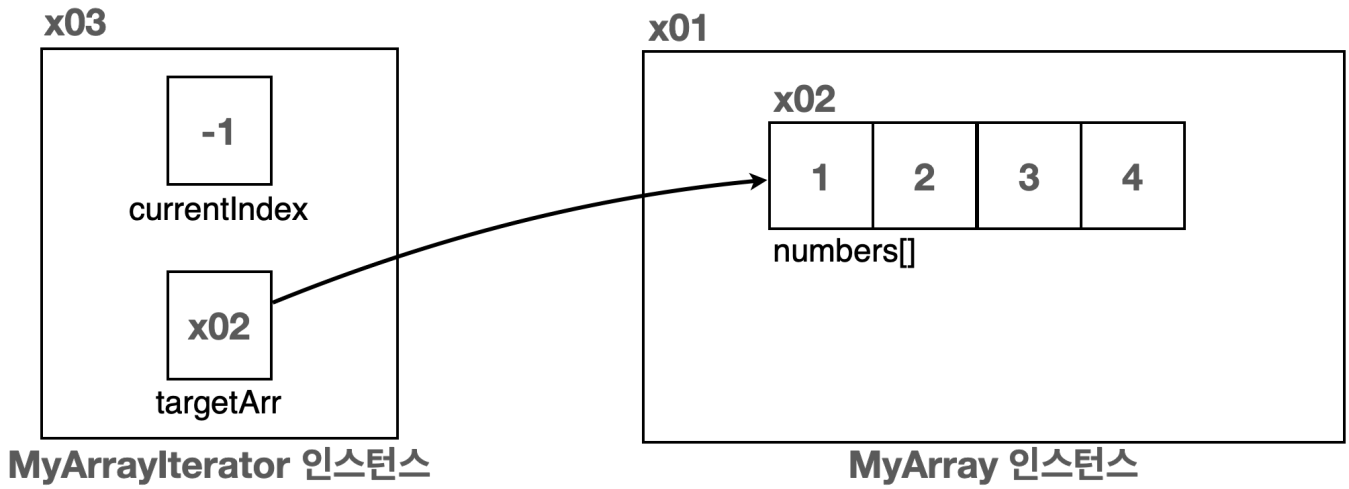
```
iterator 사용
value = 1
value = 2
value = 3
value = 4
```

클래스 구조도



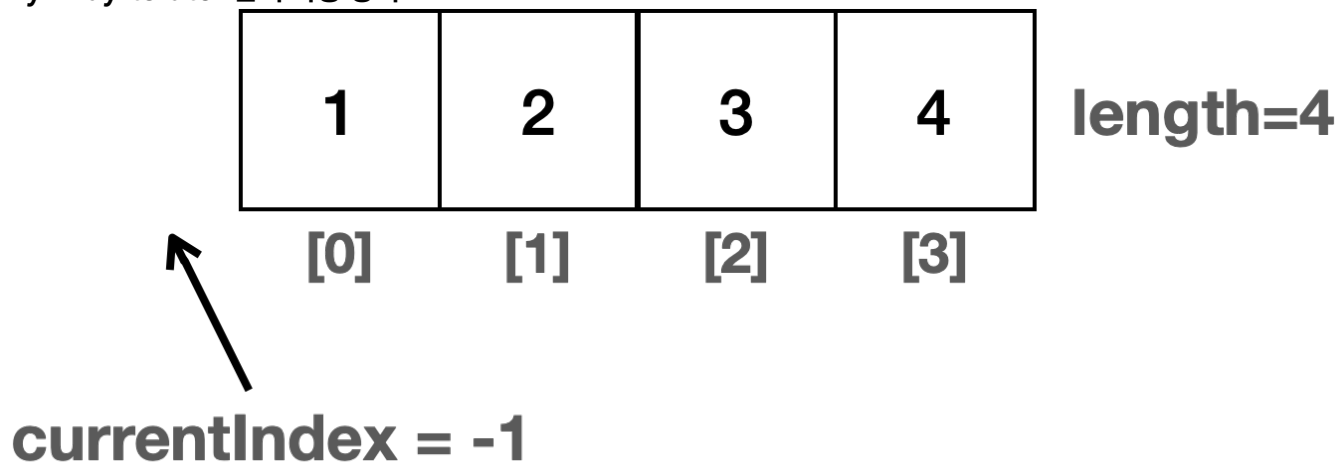
- `MyArray`는 `Iterable`(반복할 수 있는) 인터페이스를 구현한다. 따라서 `MyArray`는 반복할 수 있다는 의미가 된다.
- `Iterable` 인터페이스를 구현하면 `iterator()` 메서드를 구현해야 한다. 이 메서드는 `Iterator` 인터페이스를 구현한 반복자를 반환한다. 여기서는 `MyArrayIterator`를 생성해서 반환했다.

런타임 메모리 구조도

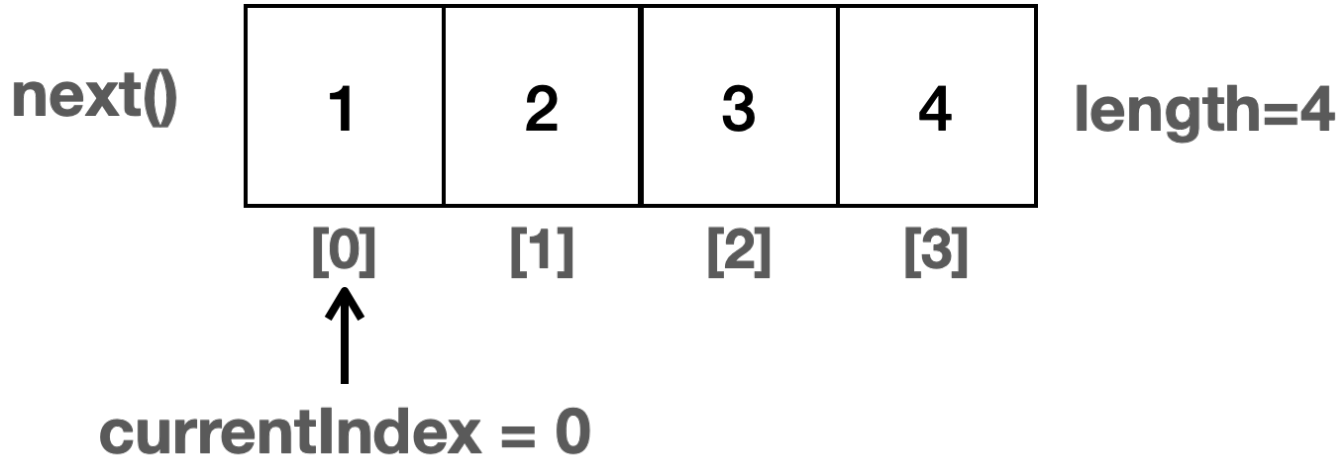


- `MyArrayIterator`의 인스턴스를 생성할 때 순회할 대상을 지정해야 한다. 여기서는 `MyArray`의 배열을 지정했다.
- `MyArrayIterator` 인스턴스는 내부에서 `MyArray`의 배열을 참조한다.
- 이제 `MyArrayIterator`를 통해 `MyArray`가 가진 내부 데이터를 순회할 수 있다.

MyArrayIterator 순회 작동 방식

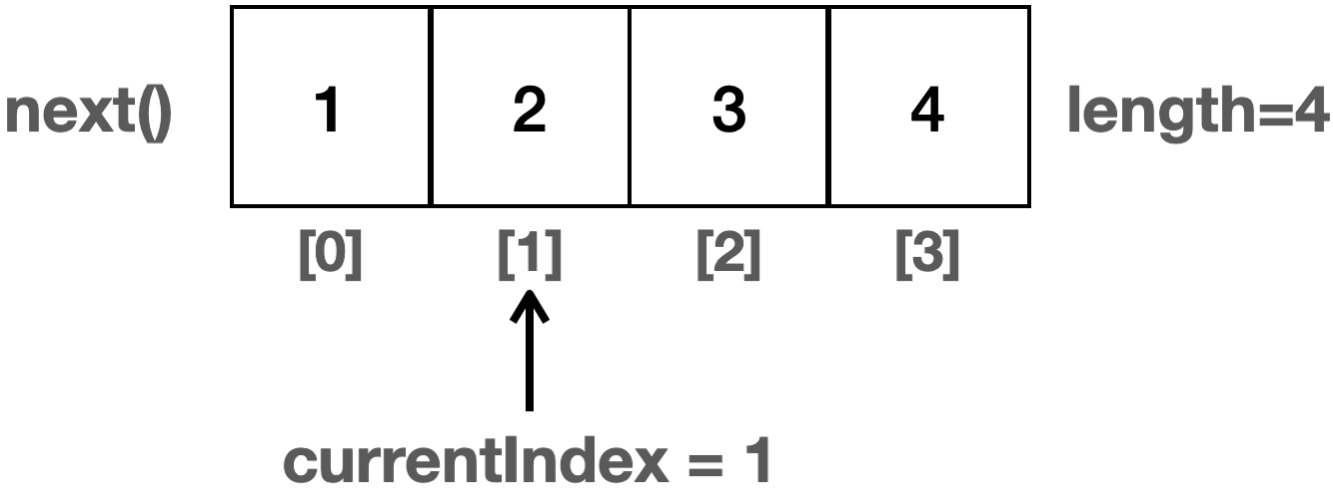


- 처음에 `currentIndex = -1`이다.

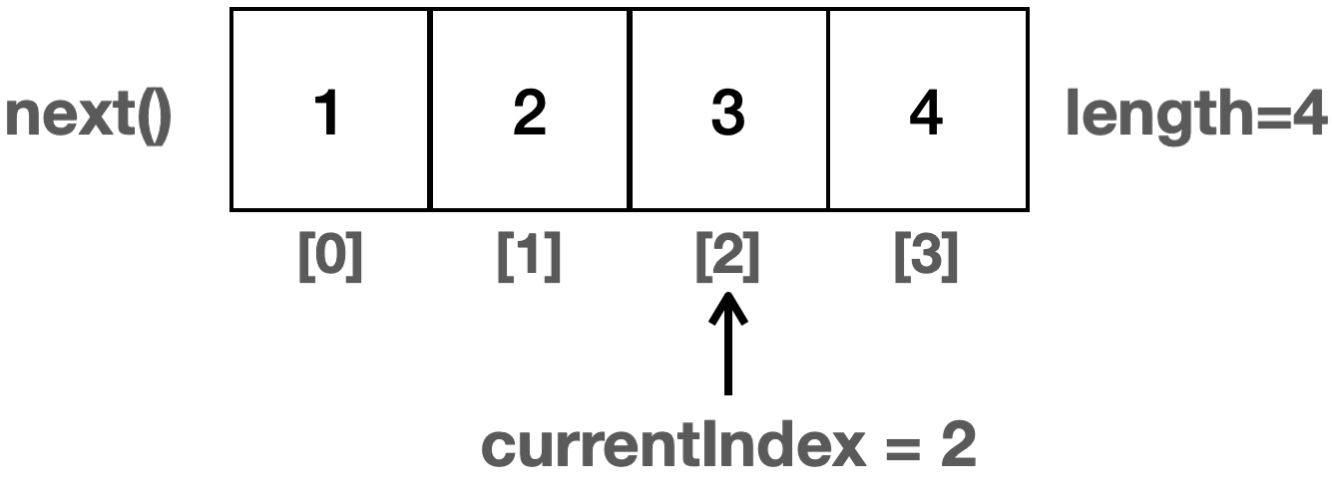


- `next()`를 처음 호출

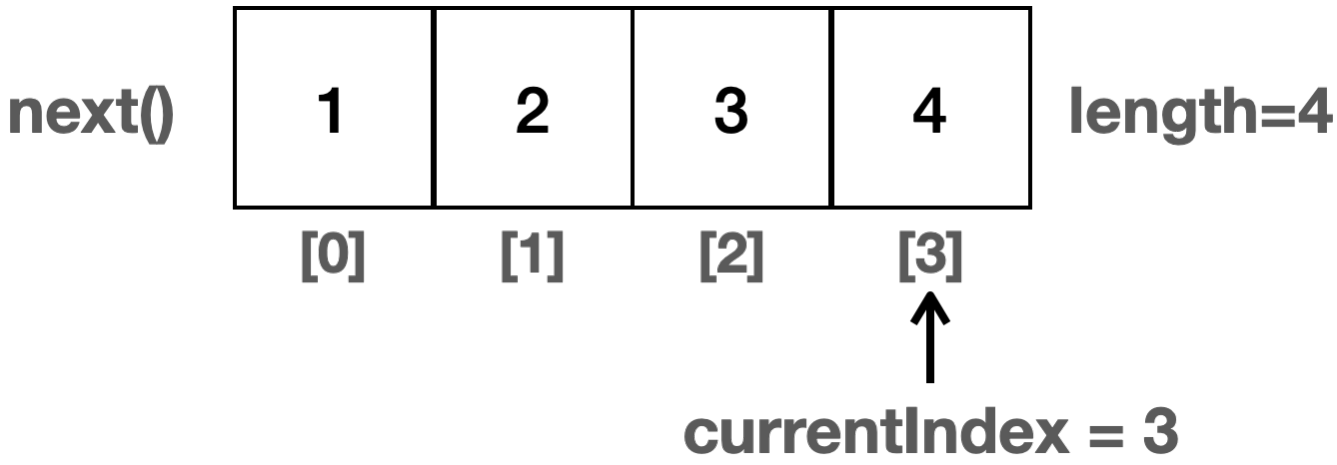
- `currentIndex = 0`으로 증가
- 1 반환



- `next()` 호출
- `currentIndex = 1`로 증가
- 2 반환



- `next()` 호출
- `currentIndex = 2`로 증가
- 3 반환



- `next()` 호출
- `currentIndex = 3`으로 증가
- 4반환
- 이후에 `hasNext()`를 호출하면 `currentIndex(3) < length(4) -1`에 의해서 종료

순회2 - 향상된 for문

Iterable과 향상된 for문(Enhanced For Loop)

`Iterable`, `Iterator`를 사용하면 또 하나의 큰 장점을 얻을 수 있다. 다음 코드를 보자.

MyArrayMain.main()에 다음 코드를 추가하고 실행해보자.

```
//추가
System.out.println("for-each 사용");
for (int value : myArray) {
    System.out.println("value = " + value);
}
```

실행 결과

```
...
for-each 사용
value = 1
value = 2
value = 3
```

```
value = 4
```

for-each문으로 불리는 향상된 for문은 자료 구조를 순회하는 것이 목적이다.
자바는 `Iterable` 인터페이스를 구현한 객체에 대해서 향상된 for문을 사용할 수 있게 해준다.

```
for (int value : myArray) {
    System.out.println("value = " + value);
}
```

이렇게 하면 자바는 컴파일 시점에 다음과 같이 코드를 변경한다.

```
while (iterator.hasNext()) {
    Integer value = iterator.next();
    System.out.println("value = " + value);
}
```

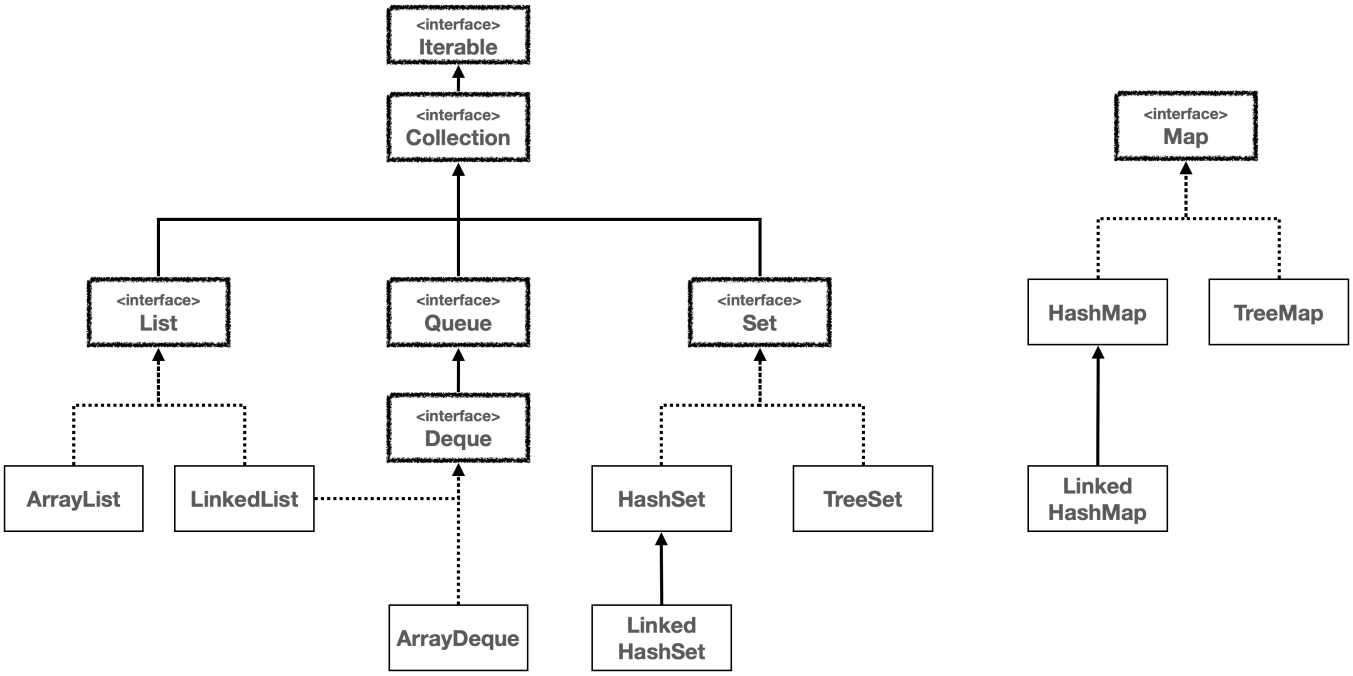
따라서 두 코드는 같은 코드이다. 물론 모든 데이터를 순회한다면 둘 중에 깔끔한 향상된 for문을 사용하는 것이 좋다.

참고 - 쉽게 이해하기
`Iterable` : "반복 가능한"이라는 뜻이다.
`Iterator` : "반복자"라는 뜻이다.

용어를 잘 보면 `Iterable`은 반복 가능한이라는 뜻이다. 우리가 만든 `MyArray`는 `Iterable`을 구현했다. 따라서 `MyArray`는 반복 가능하다는 뜻이다. `MyArray`가 반복 가능하기 때문에 `iterator`를 반환하고, for-each문도 작동한다.

정리
만드는 사람이 수고로우면 쓰는 사람이 편하고, 만드는 사람이 편하면 쓰는 사람이 수고롭다.
특정 자료 구조가 `Iterable`, `Iterator`를 구현한다면, 해당 자료 구조를 사용하는 개발자는 단순히 `hasNext()`, `next()` 또는 for-each 문을 사용해서 순회할 수 있다. 자료 구조가 아무리 복잡해도 해당 자료 구조를 사용하는 개발자는 동일한 방법으로 매우 쉽게 자료 구조를 순회할 수 있다. 이것이 인터페이스가 주는 큰 장점이다.
물론 자료 구조를 만드는 개발자 입장에서는 `Iterable`, `Iterator`를 구현해야 하니 수고롭겠지만, 해당 자료 구조를 사용하는 개발자 입장에서는 매우 편리하다.

순회3 - 자바가 제공하는 Iterable, Iterator



- 자바 컬렉션 프레임워크는 배열 리스트, 연결 리스트, 해시 셋, 연결 해시 셋, 트리 셋 등등 다양한 자료 구조를 제공한다.
- 자바는 컬렉션 프레임워크를 사용하는 개발자가 편리하고 일관된 방법으로 자료 구조를 순회할 수 있도록 `Iterable` 인터페이스를 제공하고, 이미 각각의 구현체에 맞는 `Iterator`도 다 구현해두었다.
- 자바 `Collection` 인터페이스의 상위에 `Iterable`이 있다는 것은 모든 컬렉션을 `Iterable`과 `Iterator`를 사용해서 순회할 수 있다는 뜻이다.
- `Map`의 경우 `Key`뿐만 아니라 `Value`까지 있기 때문에 바로 순회를 할 수는 없다. 대신에 `Key`나 `Value`를 정해서 순회할 수 있는데, `keySet()`, `values()`를 호출하면 `Set`, `Collection`을 반환하기 때문에 `Key`나 `Value`를 정해서 순회할 수 있다. 물론 `Entry`를 `Set` 구조로 반환하는 `entrySet()`도 순회가 가능하다.

정리하면 자바가 제공하는 컬렉션 프레임워크의 모든 자료 구조는 `Iterable`과 `Iterator`를 사용해서 편리하고 일관된 방법으로 순회할 수 있다. 물론 `Iterable`을 구현하기 때문에 향상된 for문도 사용할 수 있다.

예제 코드로 자세히 알아보자.

```
package collection.iterable;

import java.util.*;

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

```
        List<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(2);
        list.add(3);

        Set<Integer> set = new HashSet<>();
        set.add(1);
        set.add(2);
        set.add(3);

        printAll(list.iterator());
        printAll(set.iterator());

        foreach(list);
        foreach(set);
    }

    private static void printAll(Iterator<Integer> iterator) {
        System.out.println("iterator = " + iterator.getClass());
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }

    private static void foreach(Iterable<Integer> iterable) {
        System.out.println("iterable = " + iterable.getClass());
        for (Integer i : iterable) {
            System.out.println(i);
        }
    }
}
```

실행 결과

```
iterator = class java.util.ArrayList$Itr
1
2
3
iterator = class java.util.HashMap$KeyIterator
1
2
3
```

```
iterable = class java.util.ArrayList
1
2
3
iterable = class java.util.HashSet
1
2
3
```

- `Iterator`, `Iterable`은 인터페이스이다. 따라서 다형성을 적극 활용할 수 있다.
- `printAll()`, `foreach()` 메서드는 새로운 자료 구조가 추가되어도 해당 자료 구조가 `Iterator`, `Iterable`만 구현하고 있다면 코드 변경 없이 사용할 수 있다.
- `java.util.ArrayList$Itr`: `ArrayList`의 `Iterator`는 `ArrayList`의 중첩 클래스이다.
- `java.util.HashMap$KeyIterator`: `HashSet` 자료 구조는 사실은 내부에서 `HashMap` 자료 구조를 사용한다. `HashMap` 자료 구조에서 `Value`를 사용하지 않으면 `HashSet`과 같다.

> **참고**: **Iterator (반복자) 디자인 패턴**은 객체 지향 프로그래밍에서 컬렉션의 요소들을 순회할 때 사용되는 디자인 패턴이다. 이 패턴은 컬렉션의 내부 표현 방식을 노출시키지 않으면서도 그 안의 각 요소에 순차적으로 접근할 수 있게 해준다. Iterator 패턴은 컬렉션의 구현과는 독립적으로 요소들을 탐색할 수 있는 방법을 제공하며, 이로 인해 코드의 복잡성을 줄이고 재사용성을 높일 수 있다.

정렬1 - Comparable, Comparator

데이터를 정렬하는 방법을 알아보자.

예제를 통해서 배열에 들어있는 데이터를 순서대로 정렬해보자.

```
package collection.compare;

import java.util.Arrays;

public class SortMain1 {

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

```
        Integer[] array = {3, 2, 1};
        System.out.println(Arrays.toString(array));

        System.out.println("기본 정렬 후");
        Arrays.sort(array);
        System.out.println(Arrays.toString(array));
    }
}
```

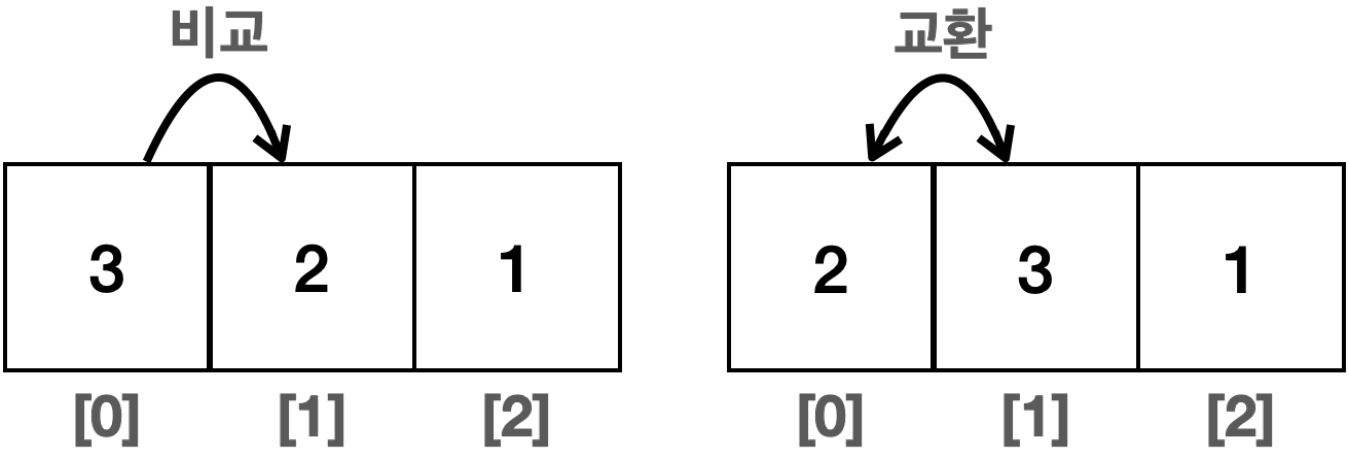
실행 결과

```
[3, 2, 1]
기본 정렬 후
[1, 2, 3]
```

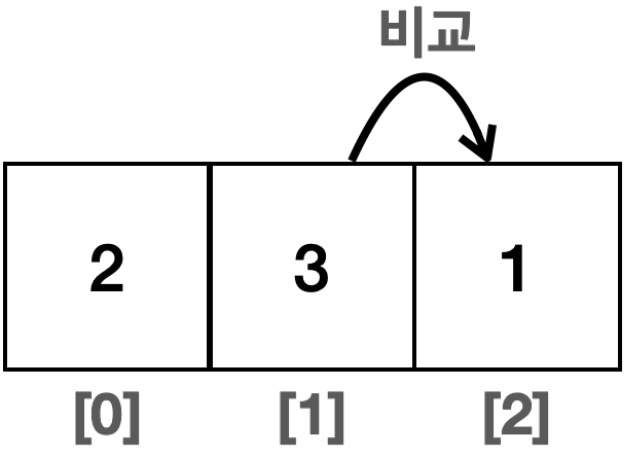
`Arrays.sort()`를 사용하면 배열에 들어있는 데이터를 순서대로 정렬할 수 있다.
원래 3, 2, 1 순서로 데이터가 들어있었는데, 정렬 후에는 1, 2, 3의 순서로 데이터가 정렬된 것을 확인할 수 있다.

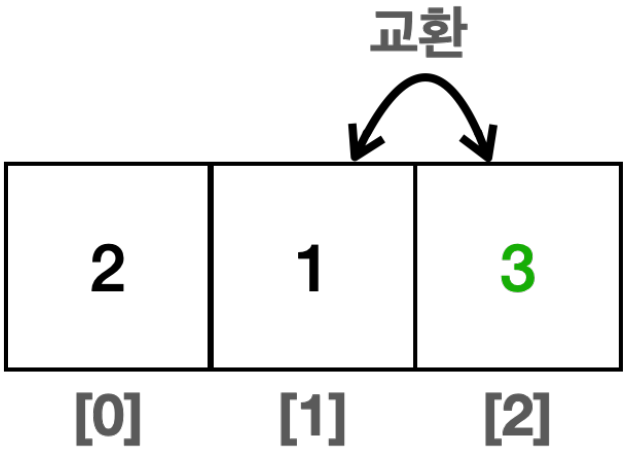
정렬 알고리즘

정렬은 대략 다음과 같은 방식으로 이루어진다.

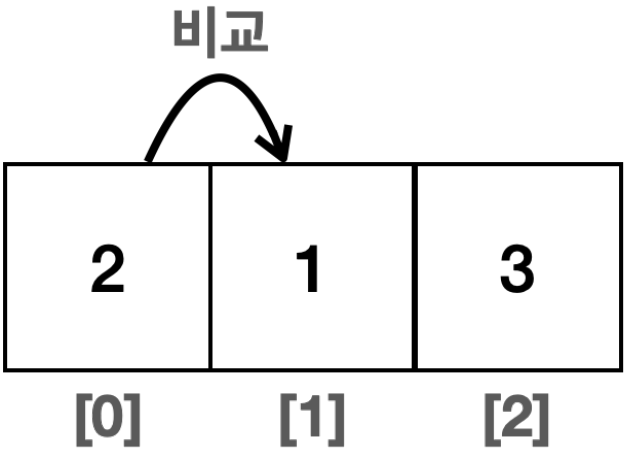


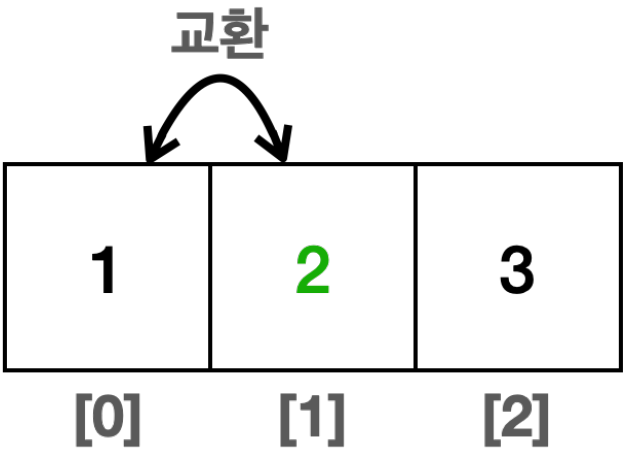
- 먼저 가장 왼쪽에 있는 데이터와 그 다음 데이터를 비교한다.
- 3과 2를 비교했을 때 3이 더 크기 때문에 둘을 교환한다.





- 다음 차례의 둘을 비교한다.
- 3과 1를 비교했을 때 3이 더 크기 때문에 둘을 교환한다.
- 이렇게 처음부터 끝까지 비교하면 마지막 항목은 가장 큰 값이 된다. 여기서는 3이다.





- 처음으로 돌아와서 다시 비교를 시작한다.
- 2와 1을 비교했을 때 2가 더 크기 때문에 둘을 교환한다.
- 최종적으로 1, 2, 3으로 정렬된다.

지금 설명한 정렬은 가장 단순한 정렬의 예시이다. 실제로는 정렬 성능을 높이기 위한 다양한 정렬 알고리즘이 존재한다. 자바는 초기에는 퀵소트를 사용했다가 ~~지금은 데이터가 작을 때(32개 이하)는 듀얼 피벗 퀵소트(Dual-Pivot QuickSort)를 사용하고, 데이터가 많을 때는 팀소트(TimSort)를 사용한다.~~ 지금은 기본형 배열의 경우 듀얼 피벗 퀵소트(Dual-Pivot QuickSort)를 사용하고, 객체 배열의 경우 팀소트(TimeSort)를 사용한다. 이런 알고리즘은 평균 O(n log n)의 성능을 제공한다.

참고: 정렬 알고리즘에 대한 이론적인 내용은 여기서 다루지 않는다. 정렬 알고리즘에 대해서 자세히 알고 싶다면 자료구조와 알고리즘을 학습하자.

비교자 - Comparator

그런데 정렬을 할 때 1, 2, 3 순서가 아니라 반대로 3, 2, 1로 정렬하고 싶다면 어떻게 해야할까?

이때는 비교자(`Comparator`)를 사용하면 된다. 이름 그대로 두 값을 비교할 때 비교 기준을 직접 제공할 수 있다.

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

- 두 인수를 비교해서 결과 값을 반환하면 된다.
  - 첫 번째 인수가 더 작으면 음수, 예(`-1`)
  - 두 값이 같으면 `0`
  - 첫 번째 인수가 더 크면 양수, 예(`1`)

```
package collection.compare;

import java.util.Arrays;
import java.util.Comparator;

public class SortMain2 {

    public static void main(String[] args) {
        Integer[] array = {3, 2, 1};
        System.out.println(Arrays.toString(array));
        System.out.println("Comparator 비교");
        Arrays.sort(array, new AscComparator());
        System.out.println("AscComparator:" + Arrays.toString(array));

        Arrays.sort(array, new DescComparator());
        System.out.println("DescComparator:" + Arrays.toString(array));
        Arrays.sort(array, new AscComparator().reversed()); //DescComparator와
같다.
        System.out.println("AscComparator.reversed:" +
Arrays.toString(array));
    }

    static class AscComparator implements Comparator<Integer> {
        @Override
        public int compare(Integer o1, Integer o2) {
            System.out.println("o1=" + o1 + " o2=" + o2);
```

```
            return (o1 < o2) ? -1 : ((o1 == o2) ? 0 : 1);
        }
    }

    static class DescComparator implements Comparator<Integer> {
        @Override
        public int compare(Integer o1, Integer o2) {
            System.out.println("o1=" + o1 + " o2=" + o2);
            return ((o1 < o2) ? -1 : ((o1 == o2) ? 0 : 1)) * -1;
        }
    }
}
```

참고 - 영상에서 문제가 있는 부분

`DescComparator`에서 `return` 부분에 -1을 곱하기 전에 괄호가 하나 더 들어가야 합니다. 메뉴얼 코드와 같이 고쳐주세요.

영상 코드

```
return (o1 < o2) ? -1 : ((o1 == o2) ? 0 : 1) * -1;
```

메뉴얼 코드 - 이렇게 수정해야 합니다.

```
return ((o1 < o2) ? -1 : ((o1 == o2) ? 0 : 1)) * -1;
```

실행 결과

```
[3, 2, 1]
Comparator 비교
o1=2 o2=3
o1=1 o2=2
AscComparator:[1, 2, 3]

o1=2 o2=1
o1=3 o2=2
DescComparator:[3, 2, 1]

o1=3 o2=2
o1=2 o2=1
```

```
AscComparator.reversed:[3, 2, 1]
```

`Arrays.sort()`를 사용할 때 비교자(`Comparator`)를 넘겨주면 알고리즘에서 어떤 값이 더 큰지 두 값을 비교할 때, 비교자를 사용한다.

```
Arrays.sort(array, new AscComparator())
Arrays.sort(array, new DescComparator())
```

- `AscComparator`를 사용하면 숫자가 점점 올라가는 오름차순으로 정렬된다.
- `DescComparator`를 사용하면 숫자가 점점 내려가는 내림차순으로 정렬된다. 왜냐하면 `DescComparator` 구현의 마지막에 `-1`을 곱해주었기 때문에 이렇게 하면 양수는 음수로, 음수는 양수로 반환된다. 쉽게 이야기해서 계산의 결과가 반대로 된다. 따라서 정렬의 결과도 반대가 된다.

정렬을 반대로

```
new AscComparator().reversed()
```

- 정렬을 반대로 하고 싶으면 `reversed()` 메서드를 사용하면 된다. 이렇게 하면 비교의 결과를 반대로 변경한다. 앞서 설명한 `-1`을 곱한 것과 같은 결과가 나온다.

비교자(`Comparator`)를 사용하면 정렬의 기준을 자유롭게 변경할 수 있다.

정렬2 - Comparable, Comparator

자바가 기본으로 제공하는 `Integer`, `String` 같은 객체를 제외하고 `MyUser`와 같이 직접 만든 객체를 정렬하려면 어떻게 해야 할까? 내가 만든 객체이기 때문에 정렬을 할 때 내가 만든 두 객체 중에 어떤 객체가 더 큰지 알려줄 방법이 있어야 한다.
이때는 `Comparable` 인터페이스를 구현하면 된다. 이 인터페이스는 이름 그대로 비교 가능한, 비교할 수 있는 이라는 뜻으로, 객체에 비교 기능을 추가해 준다.

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

- 자기 자신과 인수로 넘어온 객체를 비교해서 반환하면 된다.
  - 현재 객체가 인수로 주어진 객체보다 더 작으면 음수, 예(`-1`)
  - 두 객체의 크기가 같으면 `0`
  - 현재 객체가 인수로 주어진 객체보다 더 크면 양수, 예(`1`)

```
package collection.compare;

public class MyUser implements Comparable<MyUser> {

    private String id;
    private int age;

    public MyUser(String id, int age) {
        this.id = id;
        this.age = age;
    }

    public String getId() {
        return id;
    }

    public int getAge() {
        return age;
    }

    @Override
    public int compareTo(MyUser o) {
        return this.age < o.age ? -1 : (this.age == o.age ? 0 : 1);
    }

    @Override
    public String toString() {
        return "MyUser{" +
                "id='" + id + '\'' +
                ", age=" + age +
                '}';
    }
}
```

- `MyUser`가 `Comparable` 인터페이스를 구현한 것을 확인할 수 있다.

- `compareTo()` 구현을 보면 여기서는 정렬의 기준을 나이(`age`)로 정했다.
- `MyUser` 클래스의 기본 정렬 방식을 나이 오름차순으로 정한 것이다.
- `Comparable`을 통해 구현한 순서를 자연 순서(Natural Ordering)라 한다.

```
package collection.compare;

import java.util.Arrays;

public class SortMain3 {

    public static void main(String[] args) {
        MyUser myUser1 = new MyUser("a", 30);
        MyUser myUser2 = new MyUser("b", 20);
        MyUser myUser3 = new MyUser("c", 10);

        MyUser[] array = {myUser1, myUser2, myUser3};
        System.out.println("기본 데이터");
        System.out.println(Arrays.toString(array));

        System.out.println("Comparable 기본 정렬");
        Arrays.sort(array);
        System.out.println(Arrays.toString(array));
    }

}
```

실행 결과

```
기본 데이터
[MyUser{id='a', age=30}, MyUser{id='b', age=20}, MyUser{id='c', age=10}]

Comparable 기본 정렬
[MyUser{id='c', age=10}, MyUser{id='b', age=20}, MyUser{id='a', age=30}]
```

Arrays.sort(array)

기본 정렬을 시도한다. 이때는 객체가 스스로 가지고 있는 `Comparable` 인터페이스를 사용해서 비교한다.

`MyUser`가 구현한 대로 나이(`age`) 오름차순으로 정렬된 것을 확인할 수 있다. `MyUser`의 자연적인 순서를 사용했

다.

다른 방식으로 정렬

만약 객체가 가지고 있는 `Comparable` 기본 정렬이 아니라 다른 정렬을 사용하고 싶다면 어떻게 해야할까?
나이가 아니라 아이디로 비교하는 예제를 추가로 만들어보자.

아이디로 비교할 수 있는 `IdComparator`를 하나 만들자.

```
package collection.compare;

import java.util.Comparator;

public class IdComparator implements Comparator<MyUser> {

    @Override
    public int compare(MyUser o1, MyUser o2) {
        return o1.getId().compareTo(o2.getId());
    }
}
```

- 아이디를 기준으로 정렬할 때 사용한다.

```
package collection.compare;

import java.util.Arrays;

public class SortMain3 {

    public static void main(String[] args) {
        MyUser myUser1 = new MyUser("a", 30);
        MyUser myUser2 = new MyUser("b", 20);
        MyUser myUser3 = new MyUser("c", 10);

        MyUser[] array = {myUser1, myUser2, myUser3};
        System.out.println("기본 데이터");
        System.out.println(Arrays.toString(array));

        System.out.println("Comparable 기본 정렬");
        Arrays.sort(array);
```

```
        System.out.println(Arrays.toString(array));

        //추가
        System.out.println("IdComparator 정렬");
        Arrays.sort(array, new IdComparator());
        System.out.println(Arrays.toString(array));

        System.out.println("IdComparator().reversed() 정렬");
        Arrays.sort(array, new IdComparator().reversed());
        System.out.println(Arrays.toString(array));
    }

}
```

- 추가 부분을 확인해서 코드를 추가하자.

실행 결과

```
기본 데이터
[MyUser{id='a', age=30}, MyUser{id='b', age=20}, MyUser{id='c', age=10}]

Comparable 기본 정렬
[MyUser{id='c', age=10}, MyUser{id='b', age=20}, MyUser{id='a', age=30}]

IdComparator 정렬
[MyUser{id='a', age=30}, MyUser{id='b', age=20}, MyUser{id='c', age=10}]

IdComparator().reversed() 정렬
[MyUser{id='c', age=10}, MyUser{id='b', age=20}, MyUser{id='a', age=30}]
```

Arrays.sort(array, Comparator)

기본 정렬이 아니라 정렬 방식을 지정하고 싶다면 `Arrays.sort`의 인수로 비교자(`Comparator`)를 만들어서 넘겨주면 된다. 이렇게 비교자를 따로 전달하면 객체가 기본으로 가지고 있는 `Comparable`을 무시하고, 별도로 전달한 비교자를 사용해서 정렬한다.

여기서는 기본으로 나이를 기준으로 정렬하지만, 아이디로 정렬하고 싶다면 `IdComparator`를 넘겨주면 된다.
결과를 보면 아이디(`id`) 순으로 정렬된 것을 확인 할 수 있다.

주의!

만약 `Comparable`도 구현하지 않고, `Comparator`도 제공하지 않으면 다음과 같은 런타임 오류가 발생한다.

```
java.lang.ClassCastException: class collection.compare.MyUser cannot be cast
to class java.lang.Comparable
```

`Comparator`가 없으니, 객체가 가지고 있는 기본 정렬을 사용해야 한다. 이때 `Comparable`을 사용한다. 그런데 `Comparable`을 찾는데 없으니, 예외가 발생한다.

Comparable, Comparator 정리

객체의 기본 정렬 방법은 객체에 `Comparable`를 구현해서 정의한다. 이렇게 하면 객체는 이름 그대로 비교할 수 있는 객체가 되고 기본 정렬 방법을 가진다. 그런데 기본 정렬 외에 다른 정렬 방법을 사용해야 하는 경우 비교자(`Comparator`)를 별도로 구현해서 정렬 메서드에 전달하면 된다. 이 경우 전달한 `Comparator`가 항상 우선권을 가진다.

자바가 제공하는 `Integer`, `String` 같은 기본 객체들은 대부분 `Comparable`을 구현해 두었다.

정렬3 - Comparable, Comparator

정렬은 배열 뿐만 아니라 순서가 있는 `List` 같은 자료 구조에도 사용할 수 있다.

List와 정렬

```
package collection.compare;

import java.util.*;

public class SortMain4 {

    public static void main(String[] args) {
        MyUser myUser1 = new MyUser("a", 30);
        MyUser myUser2 = new MyUser("b", 20);
        MyUser myUser3 = new MyUser("c", 10);

        List<MyUser> list = new LinkedList<>();
        list.add(myUser1);
        list.add(myUser2);
        list.add(myUser3);
```

```
        System.out.println("기본 데이터");
        System.out.println(list);

        System.out.println("Comparable 기본 정렬");
        list.sort(null);
        //Collections.sort(list);
        System.out.println(list);

        System.out.println("IdComparator 정렬");
        list.sort(new IdComparator());
        //Collections.sort(list, new IdComparator());
        System.out.println(list);
    }

}
```

실행 결과

```
기본 데이터
[MyUser{id='a', age=30}, MyUser{id='b', age=20}, MyUser{id='c', age=10}]

Comparable 기본 정렬
[MyUser{id='c', age=10}, MyUser{id='b', age=20}, MyUser{id='a', age=30}]

IdComparator 정렬
[MyUser{id='a', age=30}, MyUser{id='b', age=20}, MyUser{id='c', age=10}]
```

Collections.sort(list)

- 리스트는 순서가 있는 컬렉션이므로 정렬할 수 있다.
- 이 메서드를 사용하면 기본 정렬이 적용된다.
- 하지만 이 방식보다는 객체 스스로 정렬 메서드를 가지고 있는 `list.sort()` 사용을 더 권장한다. 참고로 둘의 결과는 같다.

list.sort(null)

- 별도의 비교자가 없으므로 `Comparable`로 비교해서 정렬한다.
- 자연적인 순서로 비교한다.
- 자바 1.8 부터 사용

Collections.sort(list, new IdComparator())

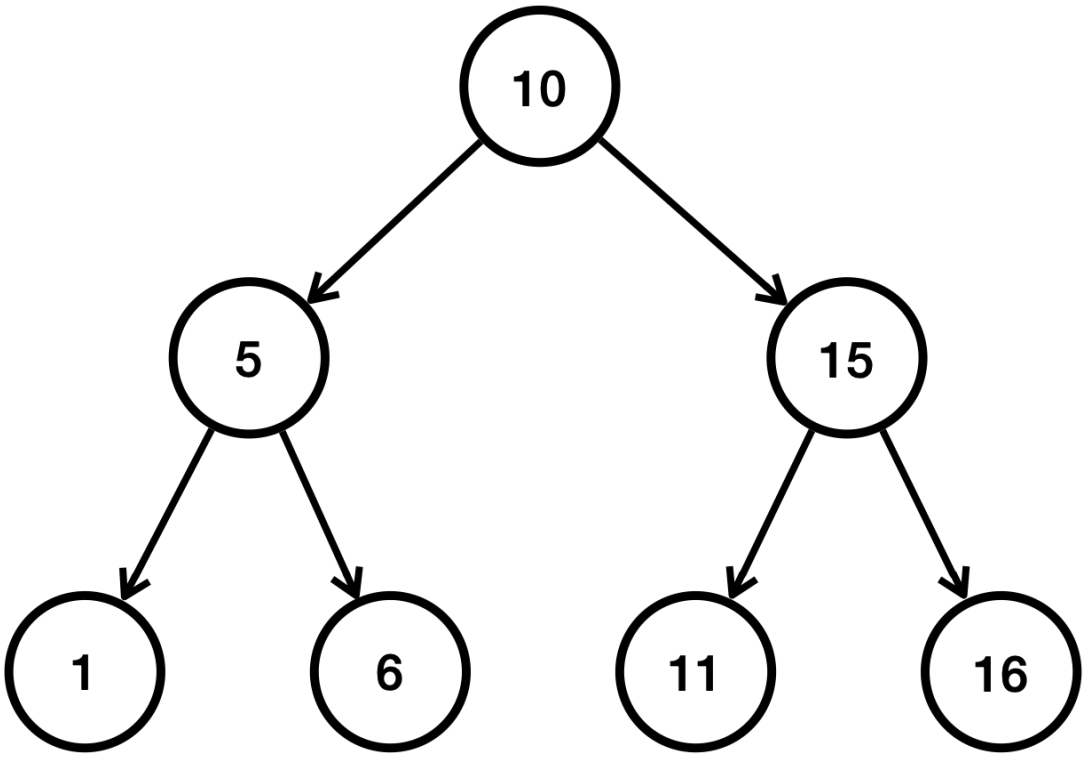
- 별도의 비교자로 비교하고 싶다면 다음 인자에 비교자를 넘기면 된다.
- 하지만 이 방식보다는 객체 스스로 정렬 메서드를 가지고 있는 `list.sort()` 사용을 더 권장한다. 참고로 둘의 결과는 같다.

list.sort(new IdComparator())

- 전달한 비교자로 비교한다.
- 자바 1.8 부터 사용

Tree 구조와 정렬

`TreeSet`과 같은 이진 탐색 트리 구조는 데이터를 보관할 때, 데이터를 정렬하면서 보관한다. 따라서 정렬 기준을 제공하는 것이 필수다.



이진 탐색 트리는 데이터를 저장할 때 왼쪽 노드에 저장해야 할 지, 오른쪽 노드에 저장해야 할 지 비교가 필요하다. 따라서 `TreeSet`, `TreeMap`은 `Comparable` 또는 `Comparator`가 필수이다.

```
package collection.compare;

import java.util.TreeSet;

public class SortMain5 {
```

```
    public static void main(String[] args) {
        MyUser myUser1 = new MyUser("a", 30);
        MyUser myUser2 = new MyUser("b", 20);
        MyUser myUser3 = new MyUser("c", 10);

        TreeSet<MyUser> treeSet1 = new TreeSet<>();
        treeSet1.add(myUser1);
        treeSet1.add(myUser2);
        treeSet1.add(myUser3);
        System.out.println("Comparable 기본 정렬");
        System.out.println(treeSet1);

        TreeSet<MyUser> treeSet2 = new TreeSet<>(new IdComparator());
        treeSet2.add(myUser1);
        treeSet2.add(myUser2);
        treeSet2.add(myUser3);
        System.out.println("IdComparator 정렬");
        System.out.println(treeSet2);
    }

}
```

실행 결과

```
Comparable 기본 정렬
[MyUser{id='c', age=10}, MyUser{id='b', age=20}, MyUser{id='a', age=30}]

IdComparator 정렬
[MyUser{id='a', age=30}, MyUser{id='b', age=20}, MyUser{id='c', age=10}]
```

```
new TreeSet<>()
```

- `TreeSet`을 생성할 때 별도의 비교자를 제공하지 않으면 객체가 구현한 `Comparable`을 사용한다.

```
new TreeSet<>(new IdComparator())
```

- `TreeSet`을 생성할 때 별도의 비교자를 제공하면 `Comparable` 대신 비교자(`Comparator`)를 사용해서 정

렬한다.

주의!

만약 `Comparable`도 구현하지 않고, `Comparator`도 제공하지 않으면 다음과 같은 런타임 오류가 발생한다.

```
java.lang.ClassCastException: class collection.compare.MyUser cannot be cast
to class java.lang.Comparable
```

정리

자바의 정렬 알고리즘은 매우 복잡하고, 또 거의 완성형에 가깝다.

자바는 **개발자가 복잡한 정렬 알고리즘은 신경 쓰지 않으면서** 정렬의 기준만 간단히 변경할 수 있도록, 정렬의 기준을 `Comparable`, `Comparator` 인터페이스를 통해 추상화해 두었다.

객체의 정렬이 필요한 경우 `Comparable`을 통해 기본 자연 순서를 제공하자. 자연 순서 외에 다른 정렬 기준이 추가로 필요하면 `Comparator`를 제공하자.

컬렉션 유틸

컬렉션을 편리하게 다룰 수 있는 다양한 기능을 알아보자.

정렬

```
package collection.utils;

import java.util.ArrayList;
import java.util.Collections;

public class CollectionsSortMain {

    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(4);
```

```
        list.add(5);

        Integer max = Collections.max(list);
        Integer min = Collections.min(list);

        System.out.println("max = " + max);
        System.out.println("min = " + min);

        System.out.println("list = " + list);
        Collections.shuffle(list);
        System.out.println("shuffle list = " + list);
        Collections.sort(list);
        System.out.println("sort list = " + list);
        Collections.reverse(list);
        System.out.println("reverse list = " + list);
    }
}
```

실행 결과

```
max = 5
min = 1
list = [1, 2, 3, 4, 5]
shuffle list = [2, 4, 1, 5, 3] //랜덤
sort list = [1, 2, 3, 4, 5]
reverse list = [5, 4, 3, 2, 1]
```

Collections 정렬 관련 메서드

- `max`: 정렬 기준으로 최대 값을 찾아서 반환한다.
- `min`: 정렬 기준으로 최소 값을 찾아서 반환한다.
- `shuffle`: 컬렉션을 랜덤하게 섞는다.
- `sort`: 정렬 기준으로 컬렉션을 정렬한다.
- `reverse`: 정렬 기준의 반대로 컬렉션을 정렬한다. (컬렉션에 들어있는 결과를 반대로 정렬한다.)

편리한 컬렉션 생성

```
package collection.utils;
```

```
import java.util.List;
import java.util.Map;
import java.util.Set;

public class OfMain {

    public static void main(String[] args) {
        // 편리한 불변 컬렉션 생성
        List<Integer> list = List.of(1, 2, 3);
        Set<Integer> set = Set.of(1, 2, 3);
        Map<Integer, String> map = Map.of(1, "one", 2, "two");

        System.out.println("list = " + list);
        System.out.println("set = " + set);
        System.out.println("map = " + map);
        System.out.println("list class = " + list.getClass());

        //java.lang.UnsupportedOperationException 예외 발생
        //list.add(4);
    }
}
```

실행 결과

```
list = [1, 2, 3]
set = [1, 2, 3]
map = {1=one, 2=two}
list class = class java.util.ImmutableCollections$ListN
```

- `List.of(...)` : 를 사용하면 컬렉션을 편리하게 생성할 수 있다. 단 이때는 가변이 아니라 불변 컬렉션이 생성된다.
  - `List`, `Set`, `Map` 모두 `of()` 메서드를 지원한다.
- 불변 컬렉션은 변경할 수 없다. 변경 메서드를 호출하면 `UnsupportedOperationException` 예외가 발생한다.

불변 컬렉션과 가변 컬렉션 전환

```
package collection.utils;
```

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class ImmutableMain {

    public static void main(String[] args) {
        //불변 리스트 생성
        List<Integer> list = List.of(1, 2, 3);

        //가변 리스트
        ArrayList<Integer> mutableList = new ArrayList<>(list);
        mutableList.add(4);
        System.out.println("mutableList = " + mutableList);
        System.out.println("mutableList class = " + mutableList.getClass());

        //불변 리스트
        List<Integer> unmodifiableList =
Collections.unmodifiableList(mutableList);
        System.out.println("unmodifiableList class = " +
unmodifiableList.getClass());

        //예외 발생 java.lang.UnsupportedOperationException
        // unmodifiableList.add(5);
    }
}
```

실행 결과

```
mutableList = [1, 2, 3, 4]
mutableList class = class java.util.ArrayList
unmodifiableList class = class
java.util.Collections$UnmodifiableRandomAccessList
```

- 불변 리스트를 가변 리스트로 전환하려면 `new ArrayList<>()`를 사용하면 된다.
- 가변 리스트를 불변 리스트로 전환하려면 `Collections.unmodifiableList()`를 사용하면 된다.
  - 물론 다양한 `unmodifiableXxx()`가 존재한다.

빈 리스트 생성

```
package collection.utils;

import java.util.ArrayList;
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;

public class EmptyListMain {

    public static void main(String[] args) {
        //빈 가변 리스트 생성
        List<Integer> list1 = new ArrayList<>();
        List<Integer> list2 = new LinkedList<>();

        //빈 불변 리스트 생성
        List<Integer> list3 = Collections.emptyList(); //자바5
        List<Integer> list4 = List.of(); //자바9

        System.out.println("list3 = " + list3.getClass());
        System.out.println("list4 = " + list4.getClass());
    }
}
```

실행 결과

```
list3 = class java.util.Collections$EmptyList
list4 = class java.util.ImmutableCollections$ListN
```

- 빈 가변 리스트는 원하는 컬렉션의 구현체를 직접 생성하면 된다.
- 빈 불변 리스트는 2가지 생성 방법이 있다.
  - `Collections.emptyList()` : 자바5부터 제공되는 기능이다.
  - `List.of()` : 자바9부터 제공되는 최신 기능이다.
  - `List.of()` 가 더 간결하고, `List.of(1,2,3)` 도 불변이기 때문에 사용법에 일관성이 있다. 자바 9 이상을 사용한다면 이 기능을 권장한다.

Arrays.asList()

`Arrays.asList` 메서드를 사용해도 다음과 같이 리스트를 생성할 수 있다.
참고로 이 메서드는 자바 1.2부터 존재했다. 자바 9를 사용한다면 `List.of()` 를 권장한다.

```
List<Integer> list = Arrays.asList(1, 2, 3);
List<Integer> list = List.of(1, 2, 3);
```

- `Arrays.asList()` 로 생성된 리스트는 **고정된 크기를 가지지만, 요소들은 변경할 수 있다.** 즉, 리스트의 길이는 변경할 수 없지만, 기존 위치에 있는 요소들을 다른 요소로 교체할 수 있다.
  - `set()` 을 통해 요소를 변경할 수 있다.
  - `add()` , `remove()` 같은 메서드를 호출하면 예외가 발생한다. 크기를 변경할 수 없다.
    - `java.lang.UnsupportedOperationException` 발생
- 고정도 가변도 아닌 애매한 리스트이다.

정리하면 일반적으로 `List.of()` 를 사용하는 것을 권장한다. 다음과 같은 경우 `Arrays.asList()` 를 선택할 수 있다.

- **변경 가능한 요소**: 리스트 내부의 요소를 변경해야 하는 경우(단, 리스트의 크기는 변경할 수 없음).
- **하위 호환성**: Java 9 이전 버전에서 작업해야 하는 경우

멀티스레드 동기화

```
package collection.utils;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class SyncMain {

    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(2);
        list.add(3);

        System.out.println("list class = " + list.getClass());
```

```
        List<Integer> synchronizedList = Collections.synchronizedList(list);
        System.out.println("synchronizedList class = " +
synchronizedList.getClass());
    }
}
```

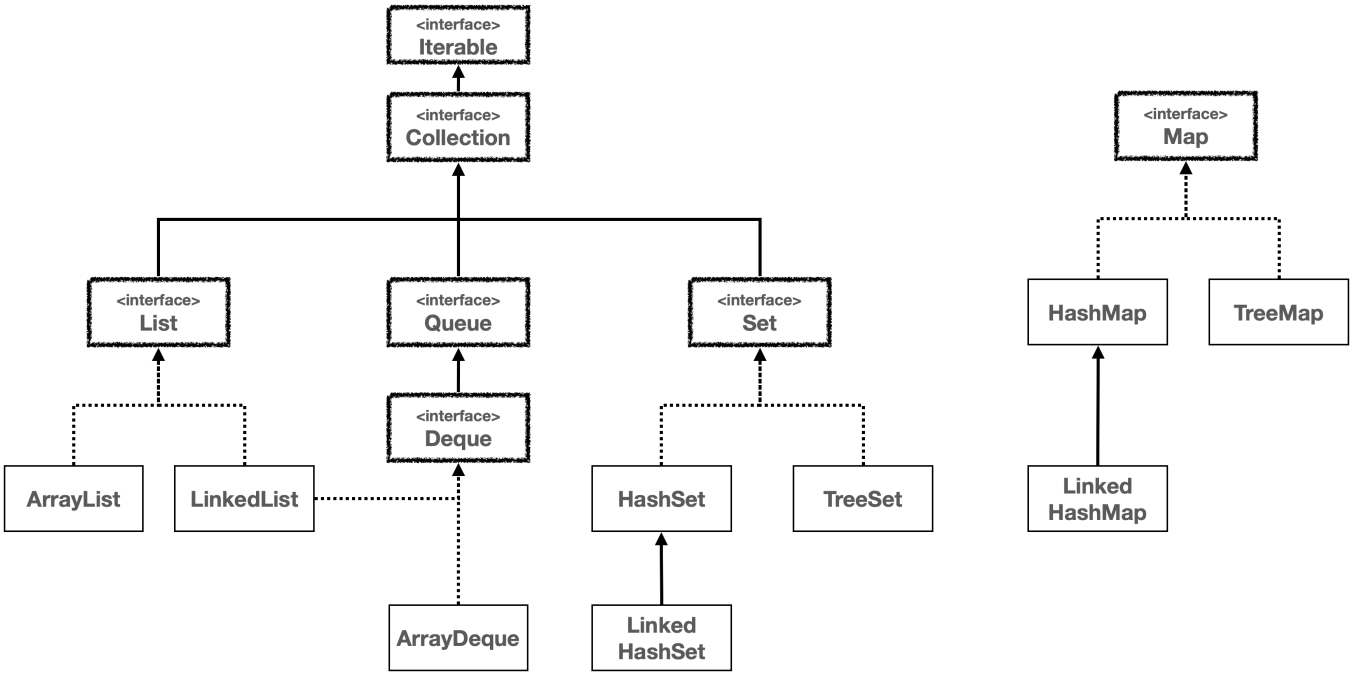
실행 결과

```
list class = class java.util.ArrayList
synchronizedList class = class
java.util.Collections$SynchronizedRandomAccessList
```

- `Collections.synchronizedList`를 사용하면 일반 리스트를 멀티스레드 상황에서 동기화 문제가 발생하지 않는 안전한 리스트로 만들 수 있다.
- 동기화 작업으로 인해 일반 리스트보다 성능은 더 느리다.
- 이 부분은 멀티스레드를 학습해야 이해할 수 있으므로 지금은 이런 것이 있다 정도만 참고하고 넘어가자.

컬렉션 프레임워크 전체 정리

자바 컬렉션 프레임워크 주요 기능



자바 컬렉션 프레임워크는 데이터 그룹을 저장하고 처리하기 위한 통합 아키텍처를 제공한다. 이 프레임워크는 인터페이스, 구현, 알고리즘으로 구성되어 있으며, 다양한 타입의 컬렉션을 효율적으로 처리할 수 있게 해준다. 여기서 컬렉션이란 객체의 그룹이나 집합을 의미한다.

Collection 인터페이스의 필요성

`Collection` 인터페이스는 자바 컬렉션 프레임워크의 가장 기본적인 인터페이스로, 자바에서 데이터 그룹을 다루는 데 필요한 가장 기본적인 메서드들을 정의한다. 그리고 다양한 컬렉션 타입들이 공통적으로 따라야 하는 기본 규약을 정의한다. `List`, `Set`, `Queue`와 같은 더 구체적인 컬렉션 인터페이스들은 모두 `Collection` 인터페이스를 확장(extend)하여, 공통된 메서드들을 상속받고 추가적인 기능이나 특성을 제공한다. 이러한 설계는 자바 컬렉션 프레임워크의 일관성과 재사용성을 높여준다.

- **일관성**: 모든 컬렉션 타입들이 `Collection` 인터페이스를 구현함으로써, 모든 컬렉션들이 기본적인 동작을 공유한다는 것을 보장한다. 이는 개발자가 다양한 타입의 컬렉션을 다룰 때 일관된 방식으로 접근할 수 있게 해준다.
- **재사용성**: `Collection` 인터페이스에 정의된 메서드들은 다양한 컬렉션 타입들에 공통으로 적용된다. 이는 코드의 재사용성을 높이고, 유지 보수를 용이하게 한다.
- **확장성**: 새로운 컬렉션 타입을 만들 때 `Collection` 인터페이스를 구현함으로써, 기존에 정의된 알고리즘과 도구를 사용할 수 있게 된다. 이는 프레임워크의 확장성을 향상시킨다.
- **다형성**: `Collection` 인터페이스를 사용함으로써, 다양한 컬렉션 타입들을 같은 타입으로 다룰 수 있다. 이는 다형성을 활용해서 유연한 코드를 작성할 수 있게 해준다.

Collection 인터페이스의 주요 메서드

`Collection` 인터페이스에는 다음과 같은 주요 메서드들이 포함된다.

- `add(E e)`: 컬렉션에 요소를 추가한다.
- `remove(Object o)`: 주어진 객체를 컬렉션에서 제거한다.
- `size()`: 컬렉션에 포함된 요소의 수를 반환한다.
- `isEmpty()`: 컬렉션이 비어 있는지 확인한다.
- `contains(Object o)`: 컬렉션이 특정 요소를 포함하고 있는지 확인한다.
- `iterator()`: 컬렉션의 요소에 접근하기 위한 반복자를 반환한다.
- `clear()`: 컬렉션의 모든 요소를 제거한다.

`Collection`은 `Map`을 제외한 모든 컬렉션 타입의 부모이다. 따라서 모든 컬렉션을 받아서 유연하게 처리할 수 있다.
대표적으로 컬렉션 인터페이스는 `iterator`를 제공한다. 따라서 데이터를 단순히 순회할 목적이라면 `Collection`을 사용하면 모든 컬렉션 타입의 데이터를 순회할 수 있다.

컬렉션 프레임워크는 크게 인터페이스, 구현, 알고리즘을 제공한다.

인터페이스

자바 컬렉션 프레임워크의 핵심 인터페이스는 다음과 같다:

- **Collection**: 단일 루트 인터페이스로, 모든 컬렉션 클래스가 이 인터페이스를 상속받는다.
  - `List`, `Set`, `Queue` 등의 인터페이스가 여기에 포함된다.
- **List**: 순서가 있는 컬렉션을 나타내며, 중복 요소를 허용한다. 인덱스를 통해 요소에 접근할 수 있다.
  - 예: `ArrayList`, `LinkedList`
- **Set**: 중복 요소를 허용하지 않는 컬렉션을 나타낸다. 특정 위치가 없기 때문에 인덱스를 통해 요소에 접근할 수 없다.
  - 예: `HashSet`, `LinkedHashSet`, `TreeSet`
- **Queue**: 요소가 처리되기 전에 보관되는 컬렉션을 나타낸다.
  - 예: `ArrayDeque`, `LinkedList`, `PriorityQueue`
- **Map**: 키와 값 쌍으로 요소를 저장하는 객체이다. `Map`은 `Collection` 인터페이스를 상속받지 않는다.
  - 예: `HashMap`, `LinkedHashMap`, `TreeMap`

구현

자바는 각 인터페이스의 여러 구현을 제공한다:

- **List**: `ArrayList`는 내부적으로 배열을 사용하며, `LinkedList`는 연결 리스트를 사용한다.
- **Set**: `HashSet`은 해시 테이블을, `LinkedHashSet`은 해시 테이블과 연결 리스트를, `TreeSet`은 레드-블랙 트리를 사용한다.
- **Map**: `HashMap`은 해시 테이블을, `LinkedHashMap`은 해시 테이블과 연결 리스트를, `TreeMap`은 레드-블랙 트리를 사용한다.
- **Queue**: `LinkedList`는 연결 리스트를 사용한다. `ArrayDeque`는 배열 기반의 원형 큐를 사용한다. 대부분의 경우 `ArrayDeque`가 빠르다.

알고리즘

컬렉션 프레임워크는 데이터를 처리하고 조작하기 위한 다양한 알고리즘을 제공한다. 이러한 알고리즘은 각각의 자료 구조 자체적으로 기능을 제공하기도 하고 또 `Collections`와 `Arrays` 클래스에 정적 메소드 형태로도 구현되어 있다. 이를 통해 정렬, 검색, 순환, 변환 등의 작업을 수행할 수 있다.

선택 가이드

- **순서가 중요하고 중복이 허용되는 경우**: `List` 인터페이스를 사용하자. `ArrayList`가 일반적인 선택이지만, 추가/삭제 작업이 앞쪽에서 빈번한 경우에는 `LinkedList`가 성능상 더 좋은 선택이다.
- **중복을 허용하지 않고 순서가 중요하지 않은 경우**: `HashSet`을 사용하자. 순서를 유지해야 하면

`LinkedHashSet`을, 정렬된 순서가 필요하면 `TreeSet`을 사용하자

- **요소를 키-값 쌍으로 저장하려는 경우**: `Map` 인터페이스를 사용하자. 순서가 중요하지 않다면 `HashMap`을, 순서를 유지해야 한다면 `LinkedHashMap`을, 정렬된 순서가 필요하면 `TreeMap`을 사용하자
- **요소를 처리하기 전에 보관해야 하는 경우**: `Queue`, `Deque` 인터페이스를 사용하자. 스택, 큐 구조 모두 `ArrayDeque`를 사용하는 것이 가장 빠르다. 만약 우선순위에 따라 요소를 처리해야 한다면 `PriorityQueue`를 고려하자.

참고: `PriorityQueue`는 자주 사용하지 않아서 따로 설명하지 않았다. 큐에 입력하는 요소에 우선순위를 부여할 수 있다.

실무 선택 가이드

- `List`의 경우 대부분 `ArrayList`를 사용한다.
- `Set`의 경우 대부분 `HashSet`을 사용한다.
- `Map`의 경우 대부분 `HashMap`을 사용한다.
- `Queue`의 경우 대부분 `ArrayDeque`를 사용한다.

문제와 풀이

카드 게임을 만들어보자.

요구사항

- 카드(`Card`)는 1 ~ 13까지있다. 각 번호당 다음 4개의 문양이 있다.
  - ♠: 스페이드
  - ♥: 하트
  - ♦: 다이아
  - ♣: 클로버
- 예) 1(♠), 1(♥), 1(♦), 1(♣), 2(♠), 2(♥), 2(♦), 2(♣) ... 13(♠), 13(♥), 13(♦), 13(♣)
- 따라서 13 * 4 = 총 52장의 카드가 있다.
- 52장의 카드가 있는 카드 뭉치를 덱(`Deck`)이라 한다.
- 2명의 플레이어(`Player`)가 게임을 진행한다.

게임을 시작하면 다음 순서를 따른다.

1. 덱에 있는 카드를 랜덤하게 섞는다.

2. 각 플레이어는 덱에서 카드를 5장씩 뽑는다.
3. 각 플레이어는 5장의 카드를 정렬된 순서대로 보여준다. 정렬 기준은 다음과 같다.
    - 작은 숫자가 먼저 나온다.
    - 같은 숫자의 경우 ♠, ♥, ♦, ♣ 순으로 정렬한다. ♠가 가장 먼저 나온다.
    - 예) 1(♠), 1(♥), 2(♦), 3(♣) 순서로 출력된다.
4. 카드 숫자의 합계가 큰 플레이어가 승리한다.
    - 게임을 단순화 하기 위해 숫자만 출력한다.
    - 합계가 같으면 무승부이다.

실행 결과 예시

```
플레이어1의 카드: [2(♠), 7(♥), 7(♦), 8(♣), 13(♠)], 합계: 37
플레이어2의 카드: [1(♠), 1(♣), 6(♠), 9(♠), 9(♣)], 합계: 26
플레이어1 승리
```

```
플레이어1의 카드: [2(♦), 3(♠), 6(♥), 10(♣), 13(♦)], 합계: 34
플레이어2의 카드: [2(♠), 4(♣), 5(♠), 11(♣), 12(♥)], 합계: 34
무승부
```

참고

스페이드, 하트 같은 아이콘을 직접 사용하기 어려운 경우 다음과 같이 `\`(백슬래시 backslash)와 함께 다음 코드를 적어주면 아이콘을 출력할 수 있다.

- `"\u2660"` : 스페이드(♠)
- `"\u2665"` : 하트(♥)
- `"\u2666"` : 다이아몬드(♦)
- `"\u2663"` : 클로버(♣)

예) `System.out.println("\u2660")`

이 문제는 정해진 정답이 없다. 실행 결과 예시를 참고하되, 자유롭게 풀면 된다.

`CardGameMain`에 `main()` 메서드를 만들고 시작하자. 필요하면 클래스를 추가해도 된다.

CardGameMain - 코드 작성

```
package collection.compare.test;
```

```
public class CardGameMain {

    public static void main(String[] args) {
        // 코드 작성
    }
}
```

정답

```
package collection.compare.test;

public enum Suit {
    SPADE("♠"),  // 스페이드(♠)
    HEART("♥"),// 하트(♥)
    DIAMOND("♦"),// 다이아몬드(♦)
    CLUB("♣");// 클로버(♣)

    private String icon;

    Suit(String icon) {
        this.icon = icon;
    }

    public String getIcon() {
        return icon;
    }

}
```

```
package collection.compare.test;

public class Card implements Comparable<Card> {
    private final int rank; // 카드의 숫자
    private final Suit suit; // 카드의 마크

    public Card(int rank, Suit suit) {
        this.rank = rank;
        this.suit = suit;
```

```
    }

    public int getRank() {
        return rank;
    }

    public Suit getSuit() {
        return suit;
    }

    @Override
    public int compareTo(Card anotherCard) {
        // 숫자를 먼저 비교하고, 숫자가 같으면 마크를 비교
        if (this.rank != anotherCard.rank) {
            return Integer.compare(this.rank, anotherCard.rank);
        } else {
            return this.suit.compareTo(anotherCard.suit);
        }
    }

    @Override
    public String toString() {
        return rank + "(" + suit.getIcon() + ")";
    }
}
```

- `Suit`는 ENUM 타입이다. 스페이드, 하트 등의 문양의 순서는 변하지 않는다고 가정하고, ENUM의 기본 순서를 사용한다.
- ENUM 타입은 `compareTo()`가 열거형의 순서인 `ordinal`로 구현되어 있다. 그리고 ENUM의 `compareTo()` 메서드는 `final` 선언되어 있어서 재정의 할 수 없다.

```
package collection.compare.test;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Deck {

    private List<Card> cards = new ArrayList<>();
```

```
    public Deck() {
        initCard();
        shuffle();
    }

    private void initCard() {
        for (int i = 1; i <= 13; i++) {
            for (Suit suit : Suit.values()) {
                cards.add(new Card(i, suit));
            }
        }
    }

    private void shuffle() {
        Collections.shuffle(cards);
    }

    public Card drawCard() {
        return cards.remove(0);
    }
}
```

- `List`에서 데이터를 앞에서 부터 꺼내고 있다. 지금처럼 데이터의 수가 작다면 `ArrayList`를 사용해도 괜찮지만, 데이터의 수가 많다면 `LinkedList`를 고려하자.

```
package collection.compare.test;

import java.util.ArrayList;
import java.util.List;

public class Player {
    private String name;
    private List<Card> hand;

    public Player(String name) {
        this.name = name;
        this.hand = new ArrayList<>();
    }

    public void drawCard(Deck deck) {
        hand.add(deck.drawCard());
```

```
    }

    public int rankSum() {
        int value = 0;
        for (Card card : hand) {
            value += card.getRank();
        }
        return value;
    }

    public void showHand() {
        hand.sort(null);
        System.out.println(name + "의 카드: " + hand + ", 합계: " + rankSum());
    }

    public String getName() {
        return name;
    }

}
```

```
package collection.compare.test;

public class CardGameMain {

    public static void main(String[] args) {
        Deck deck = new Deck();
        Player player1 = new Player("플레이어1");
        Player player2 = new Player("플레이어2");

        for (int i = 0; i < 5; i++) {
            player1.drawCard(deck);
            player2.drawCard(deck);
        }

        player1.showHand();
        player2.showHand();

        Player winner = getWinner(player1, player2);
        if (winner != null) {
            System.out.println(winner.getName() + " 승리");
```

```
        } else {
            System.out.println("무승부");
        }

    }

    private static Player getWinner(Player player1, Player player2) {
        int sum1 = player1.rankSum();
        int sum2 = player2.rankSum();

        if (sum1 > sum2) {
            return player1;
        } else if (sum1 == sum2) {
            return null;
        } else {
            return player2;
        }
    }
}
```

정리