Java 자료구조

1. 자료구조

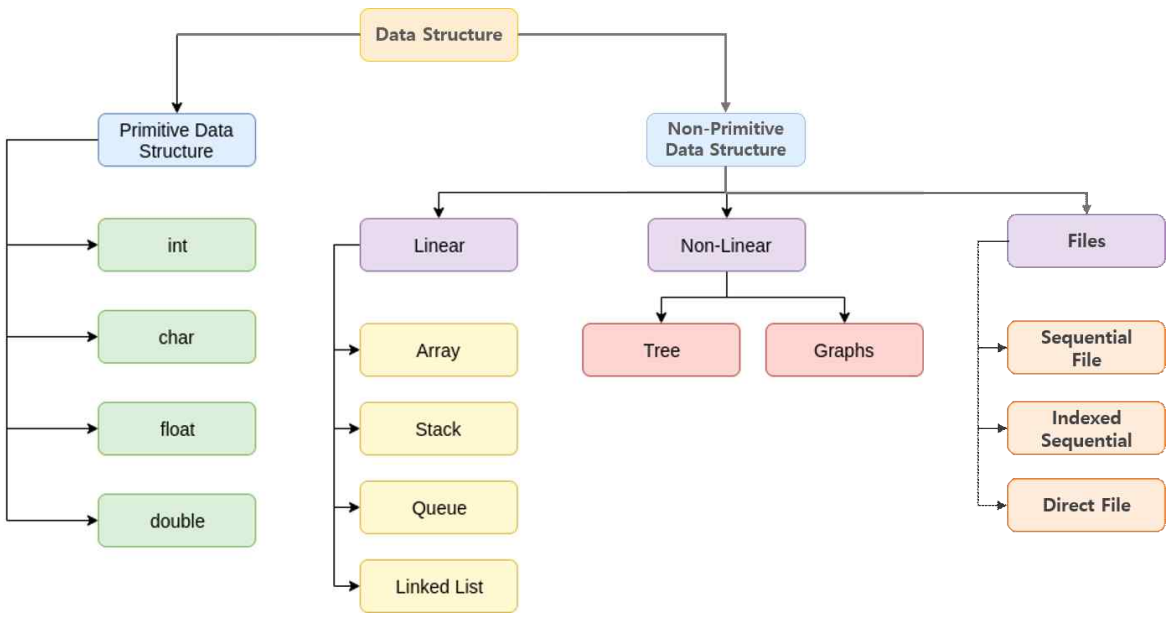
컴퓨터에서 다량의 데이터를 효율적으로 관리하기 위한 데이터의 구조를 의미하며 데이터를 효율적으로 추가, 수정, 삭제하고 검색하는 일련의 구조화된 방식을 자료구조(Data Structure)라고 할 수 있다. 자료구조에는 다양한 방식이 있는데 예를 들면 데이터를 작은 것에서 큰 순으로 저장해 관리하는 자료구조, 자료에 이름표를 붙여서 저장해 관리하는 자료구조, 데이터가 입력되는 순서를 유지해 저장하고 관리하는 자료구조 등 아주 다양한 방식의 자료구조가 존재한다.

자료구조가 이렇게 다양한 이유는 사용자 요구사항에 따라서 프로그램을 개발하는 목적이 매우 다양하기 때문이며 어떤 자료구조를 사용하느냐에 따라서 프로그램의 속도와 적용하는 알고리즘이 달라질 수 있다

1) 자료구조의 종류

자료구조는 다음 그림에서와 같이 단순 데이터 구조(기본 데이터 구조, Primitive Data Structure)와 비단순 데이터 구조(Non-Pri mitive Data Structure)로 나눌 수 있다. 비단순 데이터 구조는 다시 선형 구조(Linear), 비선형 구조(Non-Linear), 파일 구조(Files) 등으로 나뉜다.

자바 프로그래밍 언어는 아래 그림에서 선형 구조의 자료 구조를 구현한 ArrayList, Stack, Queue, LinkedList 클래스와 비선영 구조의 Tree 구조의 알고리즘을 구현한 클래스를 컬렉션 프레임워크를 통해 제공하고 있다.



2) 자료구조의 장점

프로그램을 구현할 때 적절한 자료구조를 사용하면 다음과 같은 장점을 얻을 수 있다.

▶ 효율성 향상

자료구조의 사용 목적은 데이터를 효율적으로 관리하고 사용하기 위한 것으로 주어진 문제에 알맞은 자료구조를 선택하여 사용한다면 업무 효율성이 증가하며 문제를 빠르고 정확하게 해결할 수 있다.

▶ 추상화로 처리 과정 단순화

자료구조는 복잡한 자료, 모듈, 시스템 등으로부터 핵심적인 개념이나 기능을 간추려 낸 것으로 데이터 처리 과정을 단순화하면서 처리 속도를 향상시킬 수 있다.

▶ 재사용성 증가

자료구조는 특정 시스템에서만 사용할 수 있도록 설계하지 않기 때문에 여러 시스템과 프로그래밍 언어에서 사용할 수 있어서 매우 다양한 프로젝트에서 활용될 수 있다.

2. 알고리즘

알고리즘(Algorithm)은 컴퓨터가 주어진 문제를 빠르고 효율적으로 해결하기 위해서 실행하는 절차나 방법을 자세히 표현한 것으로 컴퓨터를 활용해 주어진 문제를 해결하기 위한 일련의 방법 또는 절차이며 문제를 해결하는 과정을 순서나 절차에 따라서 나열한 것이다.
알고리즘의 예로 내비게이션 앱을 한 번 생각해 보자.
내비게이션 앱에서 길 찾기 기능은 최단 시간에 목적지까지 도착할 수 있는 최적의 방법을 알려주는 것이 핵심 알고리즘이 될 것이다. 이런 내비게이션 앱의 길 찾기에서 사용하는 패스파인더(pathfinder) 알고리즘이 있다. 또 다른 예로 컴퓨터에서 이미지를 효율적으로 사용하기 위해서 여러 가지 이미지 압축 알고리즘이 사용되는데 이 이미지 압축 알고리즘의 핵심은 이미지가 덜 손상되면서 용량을 효율적으로 줄일 수 있는 알고리즘이 될 것이며 JPG, PNG와 같은 이미지 포맷을 만드는 알고리즘이 많이 사용된다.
자료구조와 알고리즘을 체계적으로 공부하지 않아도 프로그래밍을 할 수 있지만 주어진 문제를 해결하기 위해서 자료구조와 알고리즘을 적절히 활용하면 실행 속도가 빠르고 더 효율적인 프로그램을 작성할 수 있다.
우리는 세부적인 알고리즘 이론 보다는 알고리즘의 기본 개념과 알고리즘 성능 평가에서 사용하는 용어를 간단히 알아보고 다음 단계에서 학습하는 각각의 자료구조에 대해서 알아 볼 때 해당 자료구조와 함께 자주 사용되는 알고리즘에 대해서 같이 알아 볼 것이다.

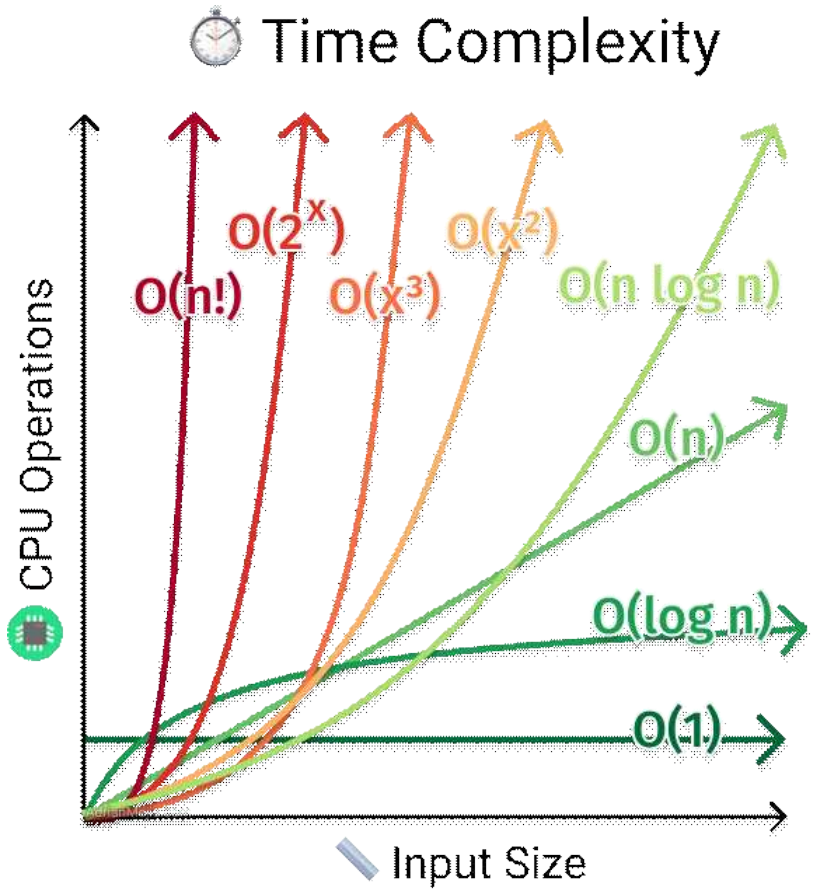
1) 복잡도

복잡도(Complexity)는 알고리즘의 효율성을 이야기 할 때 빠지지 않는 개념이며 평가 지표이다.
복잡도에는 알고리즘이 실행을 완료하는 시간 즉 실행 속도에 따른 시간 복잡도(Time Complexity)와 알고리즘이 실행될 때 사용하는 메모리 공간에 따른 공간 복잡도(Memory Complexity)가 있다.
요즘에는 메모리 가격이 저렴하고 대용량이라서 공간 복잡도의 중요도가 낮아졌기 때문에 공간 복잡도는 잘 사용되지 않고 시간 복잡도가 주로 사용되고 있으며 코딩 테스트에서 사용되는 알고리즘의 효율성 평가 지표도 시간 복잡도를 주로 사용하고 한다.
알고리즘이 주어진 문제를 해결하는데 걸리는 시간을 시간 복잡도라고 하는데 실제 주어진 문제를 해결하는 시간을 명확하게 알면 좋겠지만 같은 알고리즘이라 하더라도 컴퓨터 성능에 따라서 처리하는 시간이 각각 달라질 수 있다. 그래서 주로 점근적 분석법을 사용해 시간 복잡도를 나타낸다.
점근적 분석법이란 입력되는 데이터의 크기에 따라 주어진 문제를 해결 하는 시간과 공간이 얼마나 필요한지 알아보는 탐색법으로 다음과 같이 시간 복잡도를 표기하는 3가지 표기법을 사용한다.

- **빅-오메가 표기법(Ω(N), Big-Omega Notation) : 최선일 때(best case)의 연산 횟수 표기법**
- **빅-세타 표기법(θ(N), Big-Theta Notation) : 평균일 때(average case)의 연산 횟수 표기법**
- **빅-오 표기법(O(N), Big-O Notation) : 최악일 때(worst case)의 연산 횟수를 나타내는 표기법**

위의 3가지 표기법 중에서 알고리즘의 시간 복잡도를 표기할 때 가장 일반적으로 사용되는 것은 빅오 표기법이다. 빅-오 표기법은 알고리즘의 최악의 경우를 고려하여 프로그램이 실행되는 과정에서 소요되는 최악의 시간까지 고려할 수 있기 때문에 가장 많이 사용된다.
빅-오 표기법은 “주어진 문제를 해결할 때 입력 값 N의 변화에 따라 연산 횟수가 얼마나 많이 걸리

는지"를 표기하는 방법으로 아래 그래프와 같이 입력에 따른 시간의 증가도를 표시한다.
다음은 빅-오 표기법(O(n))으로 표현한 시간 복잡도를 나타내는 그래프 이다. 이 그래프에서 각각의 시간 복잡도를 살펴보면 X축의 데이터의 크기(n)가 증가함에 따라서 컴퓨터 명령의 수(연산 횟수, 수행시간)가 변화되는 것을 확인할 수 있다. 이때 입력 n은 작은 것이 아니라 매우 크다고 가정하고 알고리즘에서 연산 횟수는 1초에 1억 번 연산하는 것을 기본으로 하고 있다.



https://studiousguy.com/big-o-notation-definition-examples/

2) 빅-오 표기법의 시간 복잡도

빅-오 표기법으로 시간 복잡도를 나타낼 때 아래 표에서와 같이 다양한 연산 횟수를 표기하는 시간 복잡도 표기법이 있다.

이 름	시간 복잡도 표기 및 읽는 법	알고리즘 예제
상수 시간 (Constant Time)	O(1) [Order One]	입력 N의 수와 관계없이 일정한 실행 시간을 갖는 알고리즘, 배열의 인덱스로 데이터 검색, 해시 테이블에 추가, 스택에 Push, Pop

로그 시간 (Logarithmic Time)	O(logn) [Order Log N]	초반에는 빠르지만 뒤로 갈수록 시간이 증가, 이진트리 탐색
선형 시간 (Linear Time)	O(n) [Order N]	입력이 N개일 경우 N번의 수행 시간을 가짐, 연결 리스트 순회, 최댓값 찾지, for문, 정렬되지 않은 배열에서 데이터 검색 등
선형 로그 시간 (Linearithmic Time)	O(nlogn) [Order N Log N]	입력이 2배 늘 때, 연산 횟수는 2배 조금 넘게 증가, 퀵 정렬, 병합정렬, 힙 정렬 등
2차 시간 (Quadratic Time)	O(n^2) [Order N Squared, Order square of N]	중첩 for 문을 이용해 조합 가능한 모든 쌍을 탐색 할 때, 버블정렬, 삽입정렬, 선택정렬 등
3차 시간 (Cubic Time)	O(n^3) [Order cube of N]	편상관관계 계산 등
지수 시간 (Exponential Time)	O(2^n) [Order 2 to the power of N]	지수의 배수 연산은 연산 횟수를 폭발적으로 증가 시킴, 피보나치, Brutal Force 등
팩토리얼 시간 (Factorial Time)	O(n!) [Order N Factorial]	완전탐색(Brutal Force)무작위 대입

▶ 연산 횟수 계산 방법

연산 횟수 = 알고리즘 시간 복잡도 x 데이터의 크기

▶ 알고리즘 적합성 평가

알고리즘 문제에 입력의 개수가 n(1 ≤ n ≤ 1,000,000)이며 시간제한이 2초로 지정된 경우 다음과 같이 알고리즘 적합성을 평가할 수 있다. 기본적으로 1초의 연산 횟수를 1억 번을 기준으로 시간제한이 2초이므로 2억 번 이내에서 문제를 해결하는 알고리즘을 적용해야 한다.

버블 정렬 = (1,000,000)2 = 1,000,000,000,000 > 200,000,000 - 부적합 알고리즘
병합 정렬 = (1,000,000 log(1,000,000)) = 약 20,000,000 < 200,000,000 - 적합 알고리즘

▶ 시간 복잡도 산출 기준(소스 코드 참고)

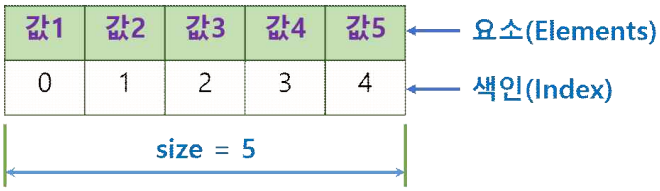
상수는 시간 복잡도 계산에서 제외한다.
코드에서 제일 많이 중첩된 반복문의 수행 횟수가 시간 복잡도의 기준이 된다.

3. 배열과 리스트

선형(Linear) 자료구조인 배열과 리스트는 비슷한 점도 많지만 다른 점도 많다. 두 자료구조의 특징을 정확히 이해하고 주어진 문제를 해결하기 위해서 적절한 자료구조를 선택해 사용해야 한다.

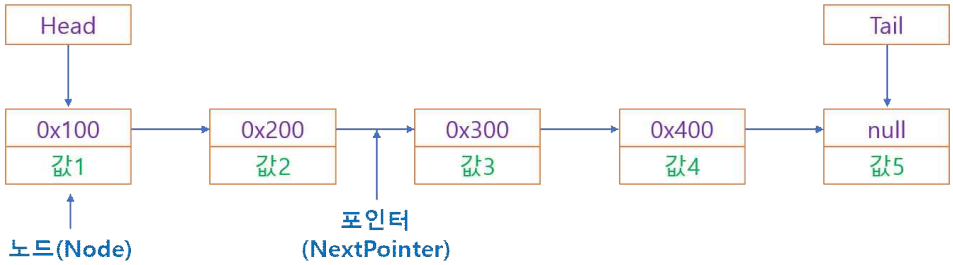
1) 배열

배열(Array)은 메모리상에 같은 데이터 타입의 자료를 연속적으로 저장하는 구조를 가지며 배열 생성시 설정한 요소의 수로 길이가 고정되는 자료구조 이다. 각 요소는 인덱스 번호가 부여되며 배열의 요소에 접근할 때 인덱스 번호를 이용해 접근할 수 있다.
배열은 중간의 인덱스 위치에 데이터를 삽입하거나 삭제하기가 어렵다. 중간 위치의 인덱스에 값을 삽입하거나 삭제하려면 해당 인덱스 주변에 있는 값들을 이동시키는 작업이 필요하다. 반면에 데이터를 탐색하거나 읽어올 때는 인덱스를 통해 빠르게 접근할 수 있다.
배열 자료구조는 버블 정렬, 삽입 정렬, 병합 정렬과 같은 다양한 정렬 알고리즘에서 사용된다.
자바에서는 컬렉션 프레임워크에서 Vector와 ArrayList를 가변 배열 형태로 제공하고 있다. ArrayList는 배열을 기반으로 자료를 저장하고 관리하며 요소를 저장할 수 있는 크기가 고정되어 있지 않고 가변적으로 동작하도록 설계되어 있다.



2) 연결 리스트

연결 리스트(Linked List)는 배열과는 다르게 인덱스나 순서에 따라서 데이터의 물리적 배치를 사용하지 않지만 순서를 가지고 연결된 순차적인 구조를 가지며 노드의 수는 고정되어 있지 않아서 필요에 따라서 노드를 늘리거나 줄일 수 있는 자료구조 이다.



컴퓨터 과학에서 노드라는 말은 값과 포인터를 쌍으로 가지는 기초적인 단위를 의미한다.
연결 리스트의 노드는 다음 노드를 가리키는 포인터와 데이터가 저장되는 곳이며 모든 노드가 연결될 때 까지 반복되어 순차적인 노드의 연결로 이루어진 자료구조 이다.
연결 리스트는 순차적인 자료구조를 가지고 있지만 배열과 다르게 데이터가 메모리에 연속적으로 위치하지 않기 때문에 Head 포인터부터 마지막까지 순서대로 데이터에 접근해야 하므로 데이터를 검색하여 읽어 올 때는 비효율적이다. 하지만 데이터를 삽입하거나 삭제할 때 포인터를 사용하기 때문에 매우 빠르게 데이터를 삽입하거나 삭제할 수 있는 자료구조이다. 그러므로 데이터 조회보다는 삽입과 삭제가 빈번하게 일어나는 데이터를 관리할 때 유용하게 사용할 수 있는 자료구조 이다. 또한 동적으로 메모리 크기를 관리하며 포인터를 사용해 메모리를 더 효율적으로 사용할 수 있기 때문에 대용량 데이터 처리에 적합하다.
연결 리스트가 사용되는 곳은 이미지 뷰어나 갤러리, 음악 플레이어 등에서 사용되며 메모리 크기가 정해져 있지 않고 데이터를 연속적으로 빠르게 삽입하거나 삭제하는 기능을 구현할 때 유용하게 사용할 수 있는 자료구조 이다.
연결 리스트에는 단일 연결 리스트(Singly-Linked List), 이중 연결 리스트(Doubly-Linked List) 등이 있다.

▶ 배열과 가변 배열 구조인 ArrayList

- com.javastudy.ds.arrayandlist

```
public class ArrayAndList01 {

  public static void main(String[] args) {

    int[] nums = {1, 3, 5, 7, 9};
    String[] books = {"자바 프로그래밍 입문", "자바 완전정복", "자바의 정석"};
    List<String> bookList = new ArrayList<>(Arrays.asList(books));

    /* 배열은 index를 통해 접근하기 때문에 데이터 검색은 빠르다. 하지만 데이터를
     * 추가, 삭제할 때는 기존에 저장되어 있는 데이터를 이동해야하기 때문에 느리다.
     **/
    for(int i = 0; i < nums.length; i++) {
      System.out.printf("%s", i < nums.length - 1 ? nums[i] + ", " : nums[i] + "\n");
    }

    /* 배열의 길이는 한 번 정해지면 변경할 수 없기 때문에 새로운 데이터를 기존
     * 데이터 뒤에 추가하려면 기존 배열보다 더 큰 배열을 새로 만들어 사용해야 한다.
     **/
    int[] nums2 = new int[nums.length + 10];
    for(int i = 0; i < nums.length; i++) {
      nums2[i] = nums[i];
    }
    nums2[5] = 20;
```

```
        System.out.println(Arrays.toString(nums2));

        /* 자바에서는 컬렉션 프레임워크에서 Vector와 ArrayList를 가변 배열 형태로
         * 제공하고 있다. ArrayList는 배열을 기반으로 자료를 저장하고 관리하며 요소를
         * 저장할 수 있는 크기가 고정되어 있지 않고 가변적으로 동작하도록 설계되어 있다.
         **/
        bookList.add("열혈 자바");
        System.out.println(bookList);

        System.out.println(nums.length + " : " + Arrays.toString(nums));
        System.out.println(bookList.size() + " : " + bookList);
    }
}
```

▶ ArrayList와 LinkedList 메서드

- com.javastudy.ds.arrayandlist

```
public class ArrayAndList02 {

    public static void main(String[] args) {

        ArrayList<Integer> aList = new ArrayList<>();

        // ArrayaList에 데이터 추가
        aList.add(1);
        aList.add(0, 2);
        aList.add(3);
        System.out.println(aList.size() + " : " + aList);
        System.out.println(aList.get(2));

        aList.remove(0);
        System.out.println(aList);

        /* 연결 리스트(Linked aList)는 배열과는 다르게 인덱스나 순서에 따라서
         * 데이터의 물리적 배치를 사용하지 않지만 순서를 가지고 연결된 순차적인
         * 구조를 가지며 노드의 수는 고정되어 있지 않아서 필요에 따라서 노드를
         * 늘리거나 줄일 수 있는 자료구조 이다.
         **/
        LinkedList<Integer> lList = new LinkedList<>();

        // lList에 데이터 추가하기
        lList.add(1);
        lList.addFirst(2);
        lList.addLast(3);
```

```
        System.out.println(lList.size() + " : " + lList);
        System.out.println(lList.get(2));
        System.out.println(lList.getLast());

        /* LinkedList는 Dequeue 인터페이스를 구현한 클래스로 Dequeue
         * 인터페이스에는 Queue와 Stack의 메서드가 정의되어 있기 때문에
         * 이들 메서드를 사용하면 큐나 스택의 자료구조로 사용할 수도 있다.
         *
         * lList의 목록이 나타내는 큐의 꼬리(마지막 노드, rear)에 데이터 추가
         **/
        lList.offer(4);
        lList.offerLast(5);
        System.out.println(lList);

         // lList의 목록이 나타내는 스택의 맨 위에 데이터를 추가
        lList.push(6);
        lList.push(7);
        System.out.println(lList);

        // lList에서 처음과 마지막 노드 삭제
        lList.remove(0);
        lList.removeLast();
        System.out.println(lList);

        // lList의 목록이 나타내는 큐의 헤드(처음 노드, front) 데이터를 읽어오면서 제거
        System.out.println(lList.poll());
        System.out.println(lList);

        // lList의 목록이 나타내는 스택에서 맨 위 데이터를 읽어오면서 제거
        System.out.println(lList.pop());
        System.out.println(lList);

        // lList의 목록이 나타내는 큐의 헤드(처음 노드, front)에서 데이터를 읽어오고 제거하지 않음
        System.out.println(lList.peek());
        System.out.println(lList.peekLast());
    }
}
```

▶ ArrayList와 LinkedList 속도 비교

- com.javastudy.ds.arrayandlist

```
public class ArrayAndList03 {

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

```
List<Integer> aList = new ArrayList<>();
List<Integer> lList = new LinkedList<>();
long start = 0, end = 0;

// ArrayList의 특정 위치에 데이터 추가하기
start = System.currentTimeMillis();
for(int i = 0; i < 100000; i++) {
  aList.add(0, i);
}
end = System.currentTimeMillis();
System.out.printf("ArrayList 데이터 추가 : %d\n", end - start);

// LinkedList의 특정 위치에 데이터 추가하기
start = System.currentTimeMillis();
for(int i = 0; i < 100000; i++) {
  lList.add(0, i);
}
end = System.currentTimeMillis();
System.out.printf("LinkedList 데이터 추가 : %d\n", end - start);

// ArrayList에서 데이터 검색하기
start = System.currentTimeMillis();
for(int i = 0; i < aList.size(); i++) {
  aList.get(i);
}
end = System.currentTimeMillis();
System.out.println("ArrayList 데이터 검색 : " + (end - start));

// LinkedList에서 데이터 검색하기
start = System.currentTimeMillis();
for(int i = 0; i < lList.size(); i++) {
  lList.get(i);
}
end = System.currentTimeMillis();
System.out.println("LinkedList 데이터 검색 : " + (end - start));

// ArrayList에서 데이터 삭제하기
start = System.currentTimeMillis();
for(int i = 0; i < aList.size(); i++) {
  aList.remove(0);
}
end = System.currentTimeMillis();
System.out.println("ArrayList 데이터 삭제 : " + (end - start));
```

```
    // LinkedList에서 데이터 삭제하기
    start = System.currentTimeMillis();
    for(int i = 0; i < lList.size(); i++) {
      lList.remove(0);
    }
    end = System.currentTimeMillis();
    System.out.println("LinkedList 데이터 삭제 : " + (end - start));
  }
}
```

▶ 재귀함수(Recursive Function) 사용하기

- com.javastudy.ds.arrayandlist

```
public class RecursionFunction01 {

  public static void main(String[] args) {

    /* 반복문보다 시간이 더 많이 걸리긴 하지만 논리적인 간결함으로
     * 복잡한 문제를 간단하게 해결 할 수 있는 장점이 있다.
     **/
    Hi(10);
    System.out.println("1 ~ 10 합 : " + sum(10));
    System.out.println("============================");

    /* 재귀함수의 기능은 반복문으로도 구현할 수 있으며 반복문은 단순히 블럭 안의
     * 코드를 반복 수행하면 되지만 함수의 재귀호출은 매개 변수 복사, 함수 종료 후
     * 복귀할 주소 저장 등의 컴퓨터 내부적인 추가 작업이 필요하기 때문에 반복문
     * 보다 재귀함수의 수행 시간이 더 많이 걸린다.
     **/
    int sum = 0;
    for(int i = 1; i <= 10; i++) {
      sum += i;
    }
    System.out.println("1 ~ 10 합 : " + sum);
  }

  /* 함수 안에서 자기 자신을 호출하는 함수를 재귀함수(Recursive Function)라고 한다.
   * 재귀함수는 자신 안에서 계속해서 자신을 호출하기 때문에 반복문과 마찬가지로
   * 종료 지점을 제대로 생각하고 구현해야 한다. 그렇지 않으면 무한 반복문을 도는 것
   * 처럼 호출 스택에 계속 쌓이면서 스택 오버플로우(Stack overflow)가 발생한다.
   **/
  public static void Hi(int n) {
    // n이 0일 때 함수 종료
    if(n == 0) return;
```

```
    System.out.println("안녕 재귀함수...");
    Hi(n - 1);
  }

  public static int sum(int n) {
    // n이 0일 경우 함수 종료
    if(n == 0) return 0;
    return n += sum(n-1);
  }
}
```

▶ 피보나치수열(Fibonacci Sequence) 생성기

- com.javastudy.ds.arrayandlist

```
/* 피보나치수열은 12세기에 이탈리아의 수학자 레오나르도 피보나치가 소개한 수열이다.
 * 당시 토끼의 번식 과정을 설명하는 문제를 통해 이 수열을 발견하였다고 한다. 피보나치
 * 수열은 각 항이 바로 앞의 두 항의 합으로 이루어진 수열로, 처음 두 항은 0과 1이다.
 * 이를 제0항 제1항이라고 하며 제2항부터는 바로 앞의 두 수를 더한 수를 나열한다.
 * 예를 들어 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 등으로 이루어진다. 피보나치수열은
 * 일반적으로 1번 항부터 1, 1, 2, 3, 5,  8, 13, 21, 34, ...와 같이 많이 표현된다.
 *
 * 피보나치수열은 다양한 수학적 성질을 가지고 있다고 알려져 있다. 가장 잘 알려진
 * 성질 중 하나는 연속되는 두 항의 비율이 황금비(약 1.618)에 근접한다는 것이다.
 * 이 비율은 수학, 예술, 자연 등 실 세계의 다양한 분야에서 중요한 역할을 하는데
 * 실제로 피보나치수열은 건축 및 디자인, 자연의 최적화 과정, 음악 구조와 리듬,
 * 경제학의 경제 주기 예측 등의 다양한 분야에서 응용되고 있다.
 **/
public class FibonacciSequence01 {

  public static void main(String[] args) {

    // 피보나치수열의 제1항 ~ 제20항까지 구한다.
    long[] fiboSeq = fibonacciSequence(20);

    for(long num : fiboSeq) {
      System.out.println(num);
    }

    System.out.println("============================");
    System.out.println(fibonacciNum(20));
    System.out.println("============================");

    for(int i = 1; i <= 20; i++) {
      System.out.println(fibonacciNum(i));
```

```
    }
  }

  // 매개변수로 받은 n 번째 항까지 피보나치 수를 배열로 반환하는 메서드
  public static long[] fibonacciSequence(int n) {
    long[] fiboSeq = new long[n];

    // 제1항과 제2항을 배열에 저장
    fiboSeq[0] = 1;
    fiboSeq[1] = 1;

    for(int i = 2; i < fiboSeq.length; i++) {
      /* 피보나치수열의 n번째 항은 바로 이전 항과 전전 항을 더한 값이
       * 되므로  F(n) = F(n-1) + F(n-2) 와 같이 구할 수 있다.
       **/
      fiboSeq[i] = fiboSeq[i - 1] + fiboSeq[i - 2];
    }
    return fiboSeq;
  }

  // 매개변수로 받은 n 번째 항의 피보나치 수를 구하는 재귀 함수
  public static long fibonacciNum(int n) {
    long num;
    if(n < 2) { // 제1항, 제2항은 1로 설정
      num = n;
    } else {
      num =  fibonacciNum(n - 1) + fibonacciNum(n - 2);
    }
    return num;
  }
}
```

[연습문제 3-1]

다음 main() 메서드 안에 생성한 배열에서 최댓값을 찾는 메서드와 재귀 함수를 정의하고 아래의 실행 결과와 같이 출력되는 프로그램을 작성하시오.

```
  public static void main(String[] args) {
    int arr[] = {20, 70, 90, 50, 60, 100, 40, 30};

  }
```

[실행결과]
배열에서 가장 큰 값(메서드) : 150
배열에서 가장 큰 값(재귀함수) : 150

▶ 팩토리얼(Factorial) 생성기

- com.javastudy.ds.arrayandlist

```
/* 계승(Factorial)은 1부터 어떤 양의 정수 n까지의 정수를 모두 곱한 것을 말하며
 * n!(n factorial로 읽음)로 나타낸다. 1부터 n개의 양의 정수를 모두 곱한 것을
 * n계승이라고 하며 n!로 나타낸다. 즉 n! = 1 x 2 x 3 x ... x (n-1) x n 이다.
 *
 * 0! = 1,  1! = 1,  2! = 2,  3! = 6,  4! = 24, 5! = 120,  6! = 720...
 **/
public class Factorial01 {

  public static void main(String[] args) {

    System.out.println(factorial(6));
    System.out.println(factorial0(6));

  }

  // n! 패토리얼을 구하는 재귀함수
  public static long factorial(int n) {
    if(n == 1) return 1;
    else return n * factorial(n - 1);
  }

  // n! 팩토리얼을 구하여 반환하는 메서드
  public static long factorial0(int n) {
    long factorial = 1;
    for(int i = 1; i <= n; i++) {
      factorial *= i;
    }
    return factorial;
  }
}
```

▶ 숫자의 합 구하기

우리나라의 컴퓨터 프로그래밍 알고리즘 트레이닝 사이트 중 하나인 백준 온라인 저지(Baekjoon Online Judge) 11720번에 올라와 있는 숫자 합 구하기 문제를 풀어 보자.

문제 출처 :

https://www.acmicpc.net/problem/11720

문제 :

N개의 숫자가 공백 없이 쓰여 있다. 이 숫자를 모두 합해서 출력하는 프로그램을 작성하시오.

입력 :

첫째 줄에 숫자의 개수 N (1 ≤ N ≤ 100)이 주어진다. 둘째 줄에 숫자 N개가 공백 없이 주어진다.

출력 :

입력으로 주어진 숫자 N개의 합을 출력한다.

입력 예시1 : 1 1	**출력 예시1 :** 1
입력 예시2 : 5 54321	**출력 예시2 :** 15
입력 예시3 : 25 7000000000000000000000000	**출력 예시3 :** 7
입력 예시4 : 11 10987654321	**출력 예시4 :** 46

문제분석 :

문제에서 N개의 숫자가 공백 없이 쓰여 있는데 이 숫자의 개수 N (1 ≤ N ≤ 100)이므로 100까지 입력 될 수 있다는 것인데, 이는 입력 값을 바로 숫자로 변환해 int나 long 타입의 변수로 담을 수 없다. 그러므로 문자열로 받아서 한 자씩 숫자로 변환하여 모두 더하는 방법을 사용해야 한다. 하지만 문자열에서 한 자씩 읽어 올 때 char 타입으로 읽어야 하므로 바로 숫자로 변환할 수 없고 문자 값의 아스키코드 값과 숫자와 차이를 확인하고 코드에 적용해야 한다. 숫자 1은 아스키코드 값으로 49이며 이는 숫자 1과 48의 차이가 있다.

가상코드 :

본격적으로 코드를 작성하기 전에 코드의 흐름을 생각해 보고 논리적인 가상 코드를 작성해 보자.

숫자의 개수 N 값 입력 받기
길이 N에 해당 하는 숫자를 문자열로 입력 받아 strNums에 저장
strNums의 문자열을 하나씩 분해해서 charNums 배열에 저장

입력 받은 숫자의 합을 더할 int 형 변수 sum 선언

for(charNums 배열 길이만큼 반복) {
  배열의 각 요소를 정수로 변환 후 sum에 더해서 누적하기
}
sum 출력하기

- com.javastudy.ds.arrayandlist

```
public class NumbersSum01 {

  public static void main(String[] args) {

    // 사용자 입력을 받기 위한 스캐너 객체 생성
    Scanner sc = new Scanner(System.in);

    // 숫자의 개수 N값 입력 받기
    int N = sc.nextInt();

    // 길이 N에 해당 하는 숫자를 문자열로 입력 받아 strNums에 저장
    String strNums = sc.next();

    // strNums의 문자열을 하나씩 분해해서 charNums 배열에 저장
    char[] charNums = strNums.toCharArray();

    // 입력 받은 숫자의 합을 더할 int 형 변수 sum 선언
    int sum = 0;
    for(int i = 0; i < charNums.length; i++) {
      /* 입력된 문자의 아스키 값에 문자 '0'의 아스키 값을 빼면 해당 숫자가
       * 되며 이를 다시 sum에 더하여 누적 합계를 구한다.
       **/
      sum += charNums[i] - '0';
    }

    // sum 출력하기
    System.out.println(sum);
  }
}
```

[연습문제 3-2]

백준 온라인 저지(Baekjoon Online Judge) 1546번에 올라와 있는 평균 구하기 문제를 풀어 보자.

문제 출처 :

https://www.acmicpc.net/problem/1546

▶ 구간 합 구하기

백준 온라인 저지(Baekjoon Online Judge) 11659번에 올라온 구간 합 구하기 문제를 풀어 보자.

문제 출처 :

https://www.acmicpc.net/problem/11659

문제분석 :

문제에서 N개의 숫자가 주어졌을 때 구해야 하는 수의 개수와 횟수가 최대 100,000번이다. 거기에 M개의 줄에 합을 구해야 하는 i와 j가 주어지는 이 M개의 중에 주어진 i와 j의 구간의 합을 구해서 출력해야 하므로 주어진 시간 1초 안에 구간 합을 계산할 수 없다. 이런 경우에는 합 배열을 이용한 구간 합을 구하는 알고리즘을 활용해야 한다.

구간 합 알고리즘 :

구간 합(Prefix Sum)은 배열을 활용해 시간 복잡도를 더 줄이기 위해 사용하는 특수한 목적의 알고리즘 이다. 이 구간 합은 합 배열을 활용하면 쉽고 간편하게 구할 수 있다. 어떤 배열 A가 있을 때 합 배열 S는 다음과 같이 배열 A[0] ~ A[i] 요소까지 모두 더하여 정의한다.

S[i] = A[0] + A[1] + A[2] + … A[i-1] + A[i]

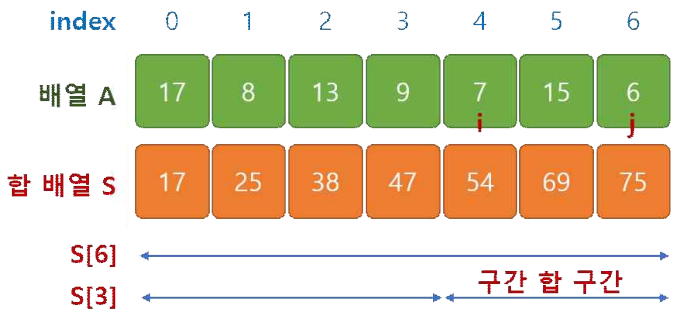
합 배열을 미리 구해 놓으면 기존 배열의 일정 범위의 합을 구하는 시간 복잡도가 O(N)에서 O(1)로 감소한다.

만약 A[i]부터 A[j]까지 배열 합을 합 배열 없이 구하게 된다면 최악의 경우 i가 0이고 j가 N인 경우로 N번의 연산이 필요하므로 시간 복잡도는 O(N)이 된다. 이런 경우 합 배열을 활용하면 O(1) 안에 답을 찾을 수 있다. 합 배열 S를 구하는 공식은 아래와 같다.

합 배열 S를 구하는 공식 : S[i] = S[i-1] + A[i]

위와 같이 합 배열이 구해지면 이 합 배열을 이용해 구간 합을 쉽게 구할 수 있다. 다음은 i부터 j까지 구간 합을 구하는 공식이다.

구간 합을 구하는 공식 : S[j] - S[i-1]



위의 그림을 보면 합 배열과 구간 합이 연관되어 있다는 것을 알 수 있다. A[0] ~ A[6]까지의 합에서 A[0] ~ A[3]까지의 합을 빼면 A[4] ~ A[6]까지의 구간 합을 구할 수 있다. 그러므로 구간 합을 구하는 공식에서와 같이 합 배열 S[6](S[j])에서 S[3](S[i-1])을 빼면 구간 합을 쉽게 구할 수 있다. 이와 같이 합 배열을 잘 만들어 두면 구간 합은 한 번의 계산으로 쉽게 구할 수 있으며 합 배열과 구간 합을 구하는 방식을 잘 익혀두면 알고리즘에서 시간 복잡도를 줄이는데 활용할 수 있다.

가상코드 :

본격적으로 코드를 작성하기 전에 코드의 흐름을 생각해 보고 논리적인 가상 코드를 작성해 보자.

```
숫자의 개수 N과, 구하는 횟수 M 입력 받기
합 배열을 만들 long형 배열 변수 S 선언
for(숫자의 개수 N만큼 반복) {
  합 배열 만들기(S[i] = S[i-1] + A[i])
}

for(구하는 횟수 M만큼 반복) {
  합을 구해야 하는 구간(i, j) 입력 받기
  입력 받은 i, j 구간 합을 구해 출력하기
}
```

- com.javastudy.ds.arrayandlist

```
public class PrefixSum01 {

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

```
        // 사용자 입력을 받기 위한 스캐너 객체 생성
        Scanner sc = new Scanner(System.in);

        // 숫자의 개수 N과, 구하는 횟수 M 입력 받기
        int N = sc.nextInt();
        int M = sc.nextInt();

        // 합 배열을 만들 long형 배열 변수 S 선언
        long[] S = new long[N + 1];

        for(int i = 1; i <= N; i++) {
          // 합 배열 만들기(S[i] = S[i - 1] + A[i])
          S[i] = S[i - 1] + sc.nextInt();
        }

        for(int m = 0; m < M; m++) {
          //* 합을 구해야 하는 구간(i, j) 입력 받기
          int i = sc.nextInt();
          int j = sc.nextInt();

          /* 입력 받은 i, j 구간 합을 구해 출력하기
           * 구간 합을 구하는 공식 : S[j] - S[i - 1] 활용
           **/
          System.out.println(S[j] - S[i - 1]);
        }

        sc.close();
    }
}
```

[연습문제 3-3]

백준 온라인 저지(Baekjoon Online Judge) 10986번에 올라와 있는 나머지 합을 구하는 문제를 풀어 보자.

문제 출처 :

https://www.acmicpc.net/problem/10986

▶ 주몽의 명령

백준 온라인 저지(Baekjoon Online Judge) 1940번에 올라온 주몽 문제를 풀어 보자.

문제 출처 :

https://www.acmicpc.net/problem/1940

문제분석 :

두 가지 재료의 고유한 번호의 합이 M과 같으면 갑옷을 만들 수 있는 재료가 되므로 전체 재료에서 두 가지를 선택해 M과 비교하기 위해서 데이터를 정렬해 놓고 두 가지 재료의 고유 번호가 M과 같은지 투 포인트 알고리즘을 사용해서 양쪽 끝에서 이동하면서 문제를 풀면 된다.
문제에서 N의 최대 범위는 15,000이므로 시간 복잡도는 크게 부담되지 않고 O(nlogn)의 시간 복잡도 알고리즘을 사용해서 문제를 해결하면 된다. 참고로 일반적인 정렬 알고리즘의 시간 복잡도가 O(nlogn)이다.

투 포인터 알고리즘 :

투 포인터는 2개의 포인터로 알고리즘의 시간 복잡도를 최적화 하는 알고리즘 이다. 투 포인터 알고리즘은 문제에 따라서 데이터의 특정 위치에 포인터 두 개를 설정하고 이 포인트를 이동해 가면서 조건에 맞는지 비교하면서 데이터를 탐색하는 알고리즘 이다. 이 때 조건에 따라서 포인터를 이동해야 하는 이동 원칙을 정의해 사용하면 된다.
주몽의 명령에 대해서 투 포인트 알고리즘을 적용하는 사례를 살펴보자.

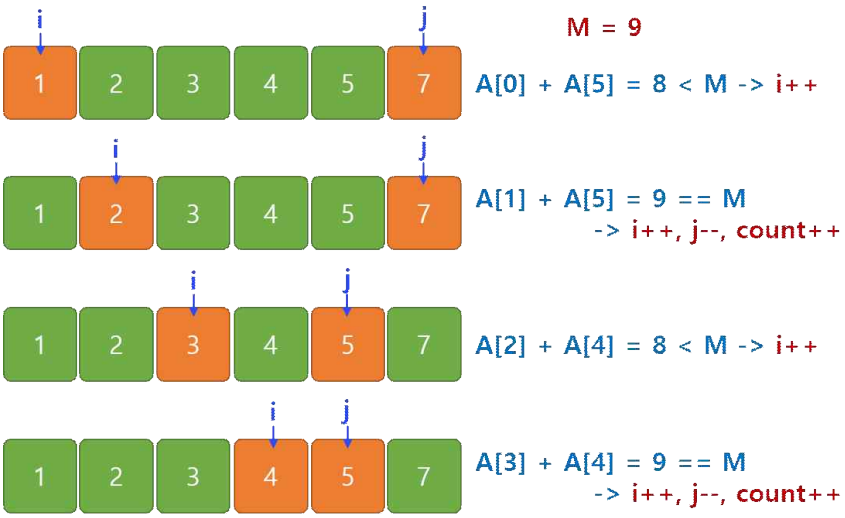
1. 먼저 입력 받은 재료들의 고유 번호 N을 배열에 저장하고 오름차순 정렬한다.

2. 두 개의 포인터를 정렬된 배열 양쪽(i, j)에 위치 시켜서 해당 위치의 재료의 고유 번호 값을 더해서 갑옷을 만들 수 있는 수 M과 같은지 비교하고 포인터 이동 원칙을 적용해 탐색을 수행한다. 이 때 갑옷을 만들 수 있는 조건인지 확인해 만들 수 있는 갑옷의 수(카운트)를 증가시킨다.

▶ 투 포인터 이동 원칙

- **투 포인터 위치의 합이 M보다 작으면 작은 쪽 index를 증가** : A[i] + A[j] < M -> i++
- **투 포인터 위치의 합이 M보다 크면 큰 쪽 index를 감소** : A[i] + A[j] > M -> j--
- **투 포인터 위치의 합이 M과 같으면 카운트 증가, 양쪽 index 이동** :
  A[i] + A[j] == M -> i++, j--, count++

3. 투 포인터의 위치 i, j가 만날 때까지 반복해서 투 포인터 이동 원칙을 적용하고 최종적으로 만들 수 있는 갑옷의 수를 출력한다.



가상코드 :

본격적으로 코드를 작성하기 전에 코드의 흐름을 생각해 보고 논리적인 가상 코드를 작성해 보자.

재료의 개수 N과, 갑옷을 만드는데 필요한 수 M 입력 받기
for(재료의 개수 N만큼 반복) {
  재료를 배열에 저장
}

재료가 저장된 배열 정렬

while(i < j) {
  if(재료 합 < M) 작은 번호 index i++
  else if(재료 합 > M) 큰 번호 index j--
  else 카운트 증가, 양쪽 index i, j 이동
}
count 출력

- com.javastudy.ds.arrayandlist

```
public class JuMongsOrder01 {

  public static void main(String[] args) {

    // 사용자 입력을 받기 위한 스캐너 객체 생성
    Scanner sc = new Scanner(System.in);

    // 재료의 개수 N과, 갑옷을 만드는데 필요한 수 M 입력 받기
    int N = sc.nextInt();
    int M = sc.nextInt();
```

```
        // 재료 정보를 담을 배열 생성
        int[]A = new int[N];

        // 재료의 개수 N만큼 반복하면서 재료를 배열에 저장
        for(int i = 0; i < N; i++) {
          A[i] = sc.nextInt();
        }

        // 재료가 저장된 배열 정렬
        Arrays.sort(A);

        // 만들 수 있는 갑옷의 수를 저장할 카운트 변수
        int count = 0;

        // 양 끝의 index 변수 i, j
        int i = 0, j = N - 1;

        // 투 포인터 이동 원칙을 적용해 포인터를 이동하면서 처리
        while(i < j) {
          // if(재료 합 < M) 작은 번호 index i++
          if(A[i] + A[j] < M) {
            i++;
          }
          // else if(재료 합 > M) 큰 번호 index j--
          else if(A[i] + A[j] > M) {
            j--;
          }
          //else 카운트 증가, 양쪽 index i, j 이동
          else {
            count++;
            i++;
            j--;
          }
        }
        // count 출력
        System.out.println(count);
        sc.close();
      }
    }
```

[연습문제 3-4]

백준 온라인 저지(Baekjoon Online Judge) 2018번에 올라와 있는 수들의 합 5(연속된 자연수의

합) 문제를 풀어 보자.

문제 출처 :

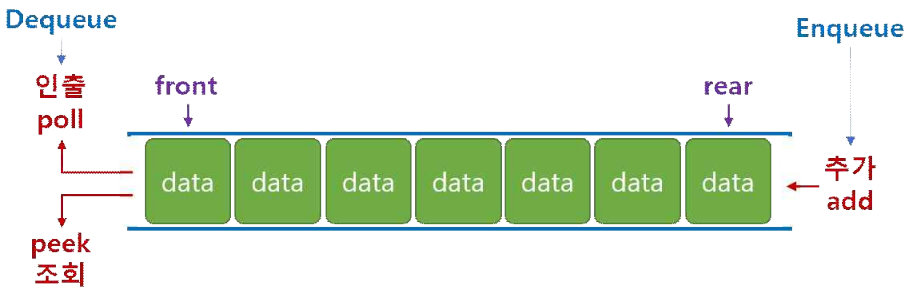
https://www.acmicpc.net/problem/2018

4. 큐와 스택

큐(Queue)와 스택(Stack)은 배열의 자료구조를 조금 더 발전시킨 자료 구조이며 리스트와 같이 저장 순서를 유지해 주는 선형 데이터 구조이다.

1) 큐

큐는 리스트와 같은 선형 구조를 가지며 음식점에서 대기 줄을 서는 것과 같이 먼저 온 데이터가 먼저 들어가서 나올 때도 먼저 나오는 선입선출(First In First Out)의 자료구조 이다.
큐 또한 선형 구조를 가지고 있어서 저장 순서를 유지하며 메모리를 동적으로 관리하고 빠른 런타임이 특징이지만 가장 오래된 데이터만 가져올 수 있고 한 번에 하나의 데이터만 처리할 수 있다는 단점이 있다.
큐는 다음 그림과 같이 한 쪽에서 데이터가 추가되고 다른 쪽에서 데이터를 삭제하는 구조를 가지고 있으며 front는 큐에서 가장 앞쪽의 데이터를 가리키고 rear는 큐에서 맨 끝의 데이터를 가리키는 포인터 이다. 또한 큐의 대기열에 데이터를 추가하는 동작을 인큐(Enqueue)라 하며 큐의 대기열에서 데이터를 꺼내는 동작을 디큐(Dequeue)라고 부른다.

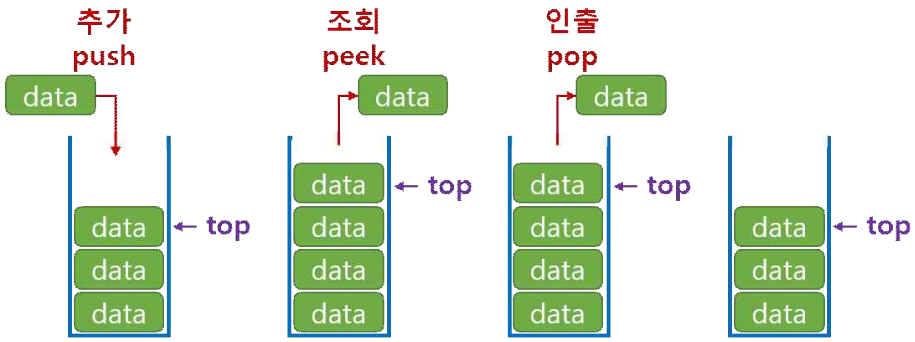


큐의 자료구조가 사용되는 곳은 프린트 대기열, 캐시 구현 등에서 사용되며 음성 데이터처럼 순서에 민감한 데이터를 처리하거나 반복적이고 자주 받는 데이터를 비동기적으로 처리해야 할 때 유용하게 사용할 수 있는 자료구조 이다.
자바에서는 Queue 인터페이스를 상속해 우선순위 큐를 구현한 PriorityQueue 클래스를 제공하고 있다. 우선순위 큐는 큐의 선입선출(FIFO, First In First Out) 구조를 가지면서 데이터가 들어온 순서대로 데이터가 나가는 것이 아니라 우선순위를 정하여 우선순위가 높은 데이터가 먼저 나가는 자료구조 이다.

2) 스택

스택은 데이터를 아래에서부터 위쪽으로 차곡차곡 쌓아 저장하며 저장되는 순서를 유지해주는 자료구조이다. 새로운 데이터가 들어오면 기존에 쌓은 위쪽에 추가하고 데이터를 꺼내려면 맨 위에서부

터 차례대로 꺼내야 한다. 그래서 이런 자료구조를 선입후출(FILO, First In Last Out) 또는 후입선출(LIFO, Last In First Out) 구조라고 한다.
스택은 다음 그림과 같이 데이터의 추가와 삭제가 한 쪽에서만 수행하는 구조이며 스택에 저장된 맨 위의 데이터를 top이 가리키고 있다. 새로운 데이터가 추가되면 top은 다시 새로 추가된 데이터를 가리키게 되며 스택에서 top이 가리키는 값을 조회하거나 인출(값을 가져오고 스택에서 삭제) 하게 된다.



스택은 메모리를 동적으로 관리하고 저장 순서를 유지하며 빠른 런타임이 특징이지만 마지막에 저장되어 있는 요소만 가져올 수 있고 한 번에 하나의 데이터만 처리할 수 있다는 단점이 있다.
스택 자료구조가 사용되는 곳은 브라우저의 뒤로 가기 기능, 프로그램의 실행 취소 기능 등에서 사용되며 가장 마지막에 입력된 데이터를 순차적으로 꺼내서 처리할 때 유용하게 사용할 수 있는 구조이다.
자바에서 스택은 디큐(Dequeue)를 기준으로 구현되어 있어서 디큐의 메서드를 사용하면 큐의 자료구조로 사용할 수도 있다.

▶ 선입선출(First In First Out) 구조를 가진 큐(Queue)

- com.javastudy.ds.queueandstack

```
public class QueueMethods01 {

  public static void main(String[] args) {

    /* LinkedList는 Dequeue 인터페이스를 구현한 클래스로 Dequeue
     * 인터페이스에는 Queue와 Stack의 메서드가 정의되어 있기 때문에
     * 이들 메서드를 사용하면 큐나 스택의 자료구조를 사용할 수도 있다.
     *
     * 큐의 자료구조는 선입선출(FIFO, First In First Out) 구조를 가지고 있어서
     * 데이터가 들어온 순서대로 데이터가 나가는 구조로 다음과 같이 큐의 대기열에 추가된다.
     *
     * --front-----------rear--
     *    70  30  50  90  60
```

```
 * ------------------------
 **/
Queue<Integer> queue1 = new LinkedList<>();
queue1.add(70);
queue1.add(30);
queue1.add(50);
queue1.add(90);
queue1.add(60);

// peek() 메서드는 큐의 front에 있는 데이터를 삭제하지 않고 읽어만 온다.
System.out.println(queue1.peek());

/* elememt() 메서드는 peek() 메서드와 동일하게 큐의 front에 있는 데이터를
 * 삭제하지 않고 읽어오지만 큐가 비어 있으면 NoSuchElementException을
 * 발생시킨다.
 **/
System.out.println(queue1.element());
System.out.println(queue1);

// poll() 메서드는 큐의 front에 있는 데이터를 읽어오면서 삭제한다.
System.out.println(queue1.poll());

/* remove() 메서드는 poll() 메서드와 동일하게 큐의 front에 있는 데이터를
 * 읽어오면서 삭제하지만 큐가 비어 있으면  NoSuchElementException을
 * 발생시킨다.
 **/
System.out.println(queue1.remove());
System.out.println(queue1);
System.out.println("==============================");

/* Queue 인터페이스를 기반으로 PriorityQueue 클래스가 구현되어 있다.
 * 우선순위 큐는 큐의 선입선출(FIFO, First In First Out) 구조를 가지면서
 * 데이터가 들어온 순서대로 데이터가 나가는 것이 아니라 우선순위를 정하여
 * 우선순위가 높은 데이터가 먼저 나가는 자료구조 이다. 우선순위 큐는 내부적으로
 * 힙(Heap)으로 구성되어 있으며 힙은 이진트리 구조를 사용해 구현된다.
 *
 * 숫자는 기본적으로 우선순위 큐에서 오름차순으로 정렬되어 큐의 대기열에 추가된다.
 *
 * --front-----------rear--
 *    30  50  60  70  90
 * ------------------------
 *
 *
 * 우선순위 큐의 기본 우선순위가 오름차순이므로 데이터를 이진트리에 저장하면 큐에서
 * 제일 작은 수가 맨 위의 루트 노드에 저장되고 트리 아래로 내려갈수록 큰 수가
```

```
 * 저장되는 구조를 가지게 되는데 이런 구조의 힙을 최소 힙(Min Heap)이라고 한다.
 * 최소 힙은 이진트리의 맨 위쪽인 루트 노드에 제일 작은 수가 저장될 수 있도록
 * 데이터가 저장될 때 수의 크기를 비교해 작은 수가 위쪽으로 큰 수가 아래쪽으로 저장될
 * 수 있도록 자리바꿈하면서 저장되며 이렇게 저장된 데이터는 아래와 같은 구조가 된다.
 *
 *          30
 *         /   \
 *       60     50
 *      /  \
 *    90   70
 **/
Queue<Integer> queue2 = new PriorityQueue<>();
queue2.offer(70);
queue2.offer(30);
queue2.offer(50);
queue2.offer(90);
queue2.offer(60);

// 이진트리의 내용 출력 - [30, 60, 50, 90, 70]
System.out.println(queue2);

// 큐의 대기열 출력 - 우선순위 순으로 출력 : 30, 50, 60, 70, 90
/*
while(true) {
  System.out.printf("%s", queue2.poll() + ", ");
  if(queue2.peek() == null) {
    break;
  }
}
System.out.println();
*/

// peek() 메서드는 큐의 front에 있는 데이터를 삭제하지 않고 읽어만 온다.
System.out.println(queue2.peek());

/* elememt() 메서드는 peek() 메서드와 동일하게 큐의 front에 있는 데이터를
 * 삭제하지 않고 읽어오지만 큐가 비어 있으면 NoSuchElementException을
 * 발생시킨다.
 **/
System.out.println(queue2.element());
System.out.println(queue2);

// poll() 메서드는 큐의 front에 있는 데이터를 읽어오면서 삭제한다.
System.out.println(queue2.poll());
System.out.println(queue2);
```

```
        /* remove() 메서드는 poll() 메서드와 동일하게 큐의 front에 있는 데이터를
         * 읽어오면서 삭제하지만 큐가 비어 있으면  NoSuchElementException을
         * 발생시킨다.
         **/
        System.out.println(queue2.remove());
        System.out.println(queue2);
    }
}
```

▶ 선입후출(First In Last Out) 구조를 가진 스택(Stack)

- com.javastudy.ds.queueandstack

```
public class StackMethods01 {

    public static void main(String[] args) {

        /* 스택의 자료구조는 선입후출(FILO, First In Last Out) 구조를 가지고 있어서
         * 맨 처음 들어온 데이터가 맨 아래에 저장되어 제일 마지막에 나갈 수 있는 자료구조이다.
         *
         *      | 30 |  <- top
         *      | 70 |
         *      | 90 |
         *      | 20 |
         *      | 50 |
         *      -----
         **/
        Stack<Integer> stack = new Stack<Integer>();
        stack.push(50);
        stack.push(20);
        stack.push(90);
        stack.push(70);
        stack.push(30);
        System.out.println(stack + " - " + stack.size());

        // peek() 메서드는 스택의 top에 있는 데이터를 삭제하지 않고 읽어만 온다.
        System.out.println(stack.peek());
        System.out.println(stack);

        // pop() 메서드는 스택의 top에 있는 데이터를 읽어오면서 삭제한다.
        System.out.println(stack.pop());
        System.out.println(stack);
```

```
        // search() 메서드는 지정한 값이 맨 위를 기준으로 몇 번째에 있는지 알려준다.
        System.out.println(stack.search(20));
        System.out.println(stack.isEmpty());
    }
}
```

▶ 쌓여있는 카드

백준 온라인 저지(Baekjoon Online Judge) 2164번에 올라온 카드2(쌓여있는 카드) 문제를 풀어 보자.

문제 출처 :

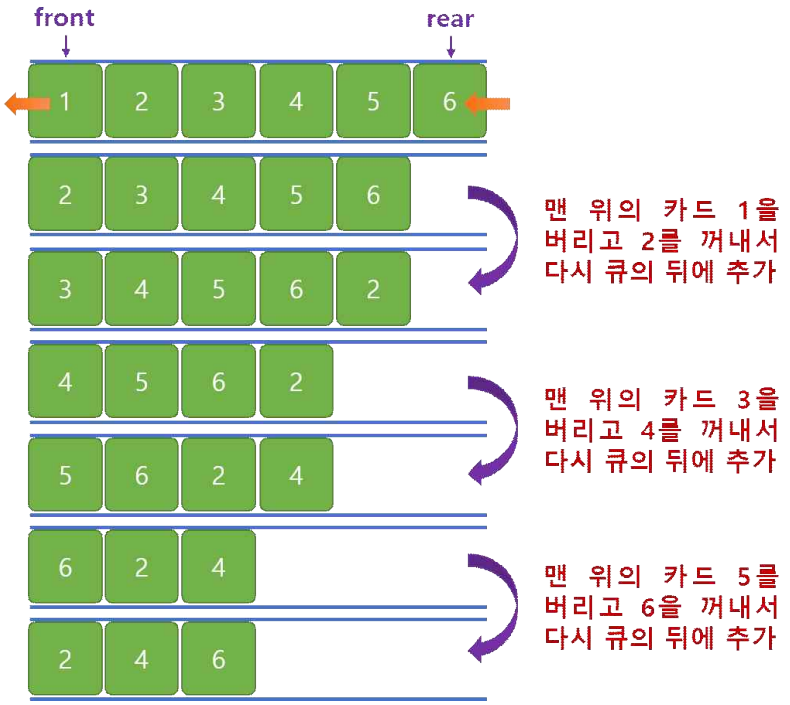
https://www.acmicpc.net/problem/2164

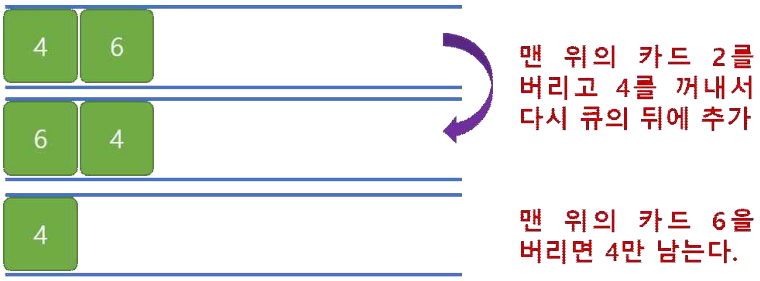
문제분석 :

문제에서 쌓여있는 카드를 예를 들어서 설명하고 있어서 스택으로 생각할 수 있지만 실제 쌓여있는 카드에서 위쪽으로 카드가 나가고 다시 위에 있는 카드를 아래쪽으로 추가하는 전형적인 큐(Queue) 자료구조를 설명하고 있다. 이 문제는 큐의 자료구조를 활용해 풀 수 있는 문제로 카드의 개수 N의 크기도 50만개 이므로 시간 복잡도의 제약사항도 크게 신경 쓸 사항은 아니다.

큐 자료구조를 적용한 알고리즘 :

카드를 맨 위에서 꺼내서 버리고 다시 맨 위의 카드를 아래에 추가하는 동작에서 알 수 있듯이 뒤쪽에서 들어와 앞으로 나가는 전형적인 선입선출(FIFO, First In First Out) 구조로 동작한다. 이는 큐(Queue)의 자료구조와 동일하다.





가상코드 :

본격적으로 코드를 작성하기 전에 코드의 흐름을 생각해 보고 논리적인 가상 코드를 작성해 보자.

```
카드의 개수 N 입력 받기
카드를 저장할 큐 생성
for(카드 개수 N만큼 반복) {
  큐에 카드 저장
}

while(카드가 1장남을 때까지 반복) {
  front에 있는 카드를 버리고 - poll()
  다시 front에 있는 카드를 가져와 큐의 뒤에 추가 - poll(), add()
}
마지막 남을 카드 숫자 출력
```

- com.javastudy.ds.queueandstack

```java
public class StackOfCards01 {

  public static void main(String[] args) {

    // 사용자 입력을 받기 위한 스캐너 객체 생성
    Scanner sc = new Scanner(System.in);

    // 카드의 개수 N값 입력 받기
    int N = sc.nextInt();

    // 카드를 저장할 큐 생성
    Queue<Integer> cardQueue = new LinkedList<>();

    // 카드 개수 N만큼 반복하며 큐에 카드 저장
    for(int i = 1; i <= N; i++) {
      cardQueue.add(i);
    }
```

```
        /* 카드가 1장 남을 때까지 반복하여 front에 있는 카드를 버리고 - poll()
         * 다시 front에 있는 카드를 가져와 큐의 뒤에 추가 - poll(), add()
         **/
        while(cardQueue.size() > 1) {
          cardQueue.poll();
          cardQueue.add(cardQueue.poll());
        }

        // 마지막 남을 카드 숫자 출력
        System.out.println(cardQueue.peek());
        sc.close();
    }
}
```

[연습문제 4-1]

백준 온라인 저지(Baekjoon Online Judge) 1874번에 올라와 있는 스택 수열을 만드는 문제를 풀어 보자.

문제 출처 :

https://www.acmicpc.net/problem/1874

5. 정렬

정렬(Sort)은 데이터를 정해진 기준에 따라서 필요한 구조로 다시 배치하는 것을 말한다.
정렬 알고리즘은 버블 정렬(Bubble Sort), 선택 정렬(Selection Sort), 삽입 정렬(Insertion Sort), 퀵 정렬(Quick Sort), 병합 정렬(Merge Sort), 기수 정렬(Radix Sort) 등 그 종류가 매우 다양하다.

1) 버블 정렬

버블 정렬(Bubble Sort)은 첫 번째 요소부터 마지막 번째 직전 요소까지 접근하면서 두 인접한 데이터의 크기를 비교해 정렬하는 방식으로 시간 복잡도가 $O(n^2)$으로 속도는 느린 편이지만 구현이 쉽고 간단히 종종 사용되는 알고리즘 이다.

▶ 버블 정렬

- com.javastudy.ds.sort

```
public class BubbleSort01 {

  public static void main(String[] args) {

    int[] nums = {6, 4, 7, 2, 5, 0, 9, 1, 3, 8};

    /* 버블정렬 알고리즘은 배열의 크기가 n개일 때 배열의 첫 번째 요소부터
     * n-1까지 접근해 인접한 요소(다음에 오는 요소)와 크기를 비교하여
     * swap(자리바꿈)을 반복하는 알고리즘이다.
     **/
    for(int i = 0; i < nums.length - 1; i++) {
      for(int j = 0; j < nums.length -1 - i; j++) {

        /* 현재 배열의 요소와 바로 다음에 위치한 배열의 요소를 비교하여 현재의
         * 요소가 크다면 뒤 쪽으로 옮기고 다음 요소를 앞으로 옮기는 작업을 한다.
         **/
        if(nums[j] > nums[j + 1]) {

          /* 임시 저장소로 사용할 temp 변수를 선언하고 크기가 큰 현재
           * 위치의 요소를 temp 변수에 저장한 후 크기가 작은 다음 위치의
           * 요소를 현재 위치에 저장한다. 그리고 temp 변수에 저장된
           * 값을 다음 위치에 저장하여 현재 위치의 요소를 뒤 쪽으로 옮긴다.
           **/
          int temp = nums[j];
          nums[j] = nums[j + 1];
          nums[j + 1] = temp;
        }
      }
```

```
        // 향상된 for 문을 이용해 nums 배열의 정렬된 데이터를 출력한다.
        for(int k : nums) {
          System.out.print(k);
        }
        System.out.println();
      }
    }
}
```

[연습문제 4-1]

백준 온라인 저지(Baekjoon Online Judge) 2750번에 올라와 있는 수 정렬하기 문제를 풀어 보자.

문제 출처 :

https://www.acmicpc.net/problem/2750

2) 선택 정렬

선택 정렬(Selection Sort)은 데이터에서 최솟값(오름차순) 또는 최댓값(내림차순)의 위치를 찾아 선택하고 맨 앞에 있는 데이터와 자리바꿈(swap) 하면서 정렬하는 알고리즘 이다. 선택 정렬도 버블 정렬과 마찬가지로 시간 복잡도가 $O(n^2)$으로 속도는 느린 편이다.

▶ 선택 정렬

- com.javastudy.ds.sort

```
public class SelectionSort01 {

  /* 선택 정렬은 데이터에서 최솟값(오름차순) 또는 최댓값(내림차순)을 위치를 찾아
   * 선택하고 맨 앞에 있는 데이터와 자리바꿈(swap) 하면서 정렬하는 알고리즘 이다.
   **/
  public static void main(String[] args) {

    int[] nums = {6, 4, 7, 2, 5, 0, 9, 1, 3, 8};

    for(int i = 0; i < nums.length; i++) {

      // 오름차순 정렬로 현재 맨 앞의 요소를 가장 작은 값으로 초기 설정 한다.
      int minIndex = i;

      // 반복문에서 정렬이 안 된 배열 요소를 탐색해 가장 작은 값을 찾는다.
```

```
        for(int j = i + 1; j < nums.length; j++) {

          // minIndex의 값 보다 현재 위치의 값이 작으면 새로운 index를 설정
          if(nums[minIndex] > nums[j]) {
            minIndex = j;
          }
        }

        /* 탐색이 완료되면 가장 작은 값을 정렬이 안 된 요소들 중에서
         * 맨 앞의 요소와 자리바꿈(swap) 한다.
         **/
        int temp = nums[minIndex];
        nums[minIndex] = nums[i];
        nums[i] = temp;

        // 향상된 for 문을 이용해 nums 배열의 정렬된 데이터를 출력한다.
        for(int k : nums) {
          System.out.print(k);
        }
        System.out.println();
      }
    }
}
```

[연습문제 4-2]

백준 온라인 저지(Baekjoon Online Judge) 1427번에 올라와 있는 소트인사이드(자리수를 내림차순 정렬하기) 문제를 풀어 보자.

문제 출처 :

https://www.acmicpc.net/problem/1427

3) 삽입 정렬

삽입 정렬(Insertion Sort)은 이미 정렬된 데이터 범위에서 정렬되지 않은 데이터를 적절한 위치에 삽입 시켜서 정렬하는 알고리즘으로 두 번째 요소부터 n번째 요소까지 현재 요소의 앞쪽에 있는 요소들과 비교하여 현재 요소가 위치할 index에 요소를 삽입하는 방식이다. 삽입 정렬도 버블 정렬, 선택 정렬과 마찬가지로 시간 복잡도가 $O(n^2)$으로 속도는 느린 편이지만 구현은 쉬운 편이다.

▶ 삽입 정렬

- com.javastudy.ds.sort

```
public class InsertionSort01 {

  /* 삽입 정렬은 이미 정렬된 데이터 범위에서 정렬되지 않은 데이터를 적절한 위치에
   * 삽입 시켜서 정렬하는 알고리즘으로 두 번째 요소부터 n번째 요소까지 현재 요소의
   * 앞쪽에 있는 요소들과 비교해 현재 요소가 위치할 index에 요소를 삽입하는 방식이다.
   **/
  public static void main(String[] args) {

    int[] nums = {6, 4, 7, 2, 5, 0, 9, 1, 3, 8};

    // 최초 실행될 때 두 번째 요소가 현재 요소가 된다.
    for(int i = 1; i < nums.length; i++) {

      // 현재 선택된 요소의 값을 임시 변수에 저장한다.
      int temp = nums[i];

      // 현재 요소를 기준으로 이전 원소를 탐색하기 위한 index 변수
      int prev = i - 1;

      // 현재 선택된 요소가 이전 요소 보다 작은 경우만 index 0번까지 반복
      while(prev >= 0 && nums[prev] > temp) {
        // 현재 선택된 요소가 탐색 중인 요소 보다 작다면 탐색한 요소를 다음으로 보낸다.
        nums[prev + 1] = nums[prev];
        prev--;
      }

      // 탐색이 종료된 지점에 현재 선택된 요소의 값을 삽입한다.
      nums[prev + 1] = temp;

      // 향상된 for 문을 이용해 nums 배열의 정렬된 데이터를 출력한다.
      for(int k : nums) {
        System.out.print(k);
      }
      System.out.println();
    }
  }
}
```

[연습문제 4-3]

백준 온라인 저지(Baekjoon Online Judge) 11399번에 올라와 있는 ATM 문제를 풀어 보자.

문제 출처 :
https://www.acmicpc.net/problem/11399

4) 사용자 정의 객체 정렬하기

정수와 실수 그리고 문자열 등을 배열 또는 ArrayList 등에 담아서 그 안의 데이터를 정렬하려면 배열은 Arrays.sort() 메서드를 사용하면 되고 ArrayList는 Collections.sort() 메서드를 사용하면 된다. 그렇다면 사용자가 정의한 Car 클래스의 인스턴스를 5개 생성하고 ArrayList에 담아서 그 안의 인스턴스를 정렬하려면 어떻게 하면 될까? 결론부터 말하자면 정렬이 불가능하다 왜냐하면 Car 클래스의 인스턴스를 어떻게 정렬해야 하는지 기준이 없기 때문이다. 이 정렬 기준을 만들기 위해서는 Comparable 인터페이스를 구현해야 한다. Comparable 인터페이스에는 compareTo()라는 추상 메서드가 있는데 이 메서드에 정렬 기준을 만들어 오버라이드 하면 그 기준으로 오름차순 또는 내림차순 정렬이 가능하다.
다음의 사용자 정의 객체 정렬하기 예제는 Car 클래스에서 Comparable 인스턴스를 구현하면서 compareTo() 메서드 안에서 Car 클래스의 인스턴스를 자동차 가격인 price를 기준으로 오름차순 정렬할 수 있도록 오버라이딩 하는 예제 이다.

▶ 사용자 정의 객체 정렬하기

- com.javastudy.ds.sort

```
/* 이 Car 클래스는 Comparable 인터페이스의 compareTo() 추상 메서드를
 * 오버라이드 하면서 이 메서드 안에서 Car 클래스의 인스턴스를 정렬하는 기준을
 * 정의한 클래스 이다.
 **/
class Car implements Comparable<Car> {
  private String name;
  private int price;

  public Car(String name, int price) {
    this.name = name;
    this.price = price;
  }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public int getPrice() {
    return price;
  }
  public void setPrice(int price) {
    this.price = price;
```

```
    }

    public String toString() {
        return name + "(" + price + ")";
    }

    @Override
    public int compareTo(Car c) {
        // 자신의 가격이 파라미터로 받은 가격보다 크면 양수를 반환
        if(this.price > c.price)  return 1;

        // 자신의 가격이 파라미터로 받은 가격과 같다면 0을 반환
        else if(this.price == c.price) return 0;

        // 자신의 가격이 파라미터로 받은 가격보다 작다면 음수를 반환
        else return -1;
    }
}

public class UserObjectSort01 {

    public static void main(String[] args) {

        int[] nums = {4, 7, 3, 1, 9, 6, 5};

        // 숫자는 기본 오름차순 정렬
        Arrays.sort(nums);
        System.out.println(Arrays.toString(nums));

        ArrayList<String> sList = new ArrayList<>();
        sList.add("스프링 프레임워크");
        sList.add("스자바 스크립트");
        sList.add("Node.js");
        System.out.println(sList);

        // 문자열을 사전순 정렬(유니코드 순으로 정렬됨)
        Collections.sort(sList);
        System.out.println(sList);

        ArrayList<Car> cList = new ArrayList<>();
        cList.add(new Car("람보르기니", 600000000));
        cList.add(new Car("포르쉐", 35000000));
        cList.add(new Car("롤스로이스", 50000000));
        System.out.println(cList);
```

```
        /* 사용자 정의 객체는 Comparable 인터페이스의 추상 메서드인
         * compareTo() 메서드에서 정의한 정렬 기준으로 정렬 된다.
         **/
        Collections.sort(cList);
        System.out.println(cList);
    }
}
```

6. 함수형 프로그래밍

Java8부터 함수형 프로그래밍(Functional Programming)을 지원하기 위해서 람다식(Lambda Expression)과 스트림(Stream)이 새롭게 추가되었다. 이 람다식과 스트림을 활용하면 코드가 간결하고 가독성이 높은 함수형 프로그래밍 방식의 자바 코드를 작성할 수 있다.
람다식(Lambda Expression)은 간결하게 익명 함수를 표현하는 함수 표현식으로 줄여서 람다(Lambda) 또는 람다 함수(Lambda Function)라고도 부른다.
스트림은 "데이터의 흐름"을 의미하며 람다식을 사용할 수 있게 설계되어 함수형 프로그래밍을 지원하는 기술 중 하나이다. 스트림은 배열 또는 컬렉션과 같은 집합 데이터에 여러 기능의 메서드를 연결해 단계적으로 적용하고 최종적인 결과를 만들어 가는 API 이며 입출력을 위한 I/O 스트림과는 전혀 다른 기술이다.

1) 람다식

자바의 문법에서 모든 메소드는 클래스 내부에 만들어야 하므로 객체를 생성하고 그 객체 안에 있는 메소드를 호출하는 단계를 거쳐야 비로소 함수를 사용할 수 있다. 이것은 함수를 정의하고 그 함수를 사용해서 프로그램을 만들어가는 함수형 프로그래밍 기법과는 다른 프로그래밍 방식이다.
함수형 프로그래밍은 기본적으로 함수를 1급 객체로 취급해 함수가 독립적으로 사용되며 함수를 변수에 대입할 수 있고 다른 함수의 매개 변수로 사용할 수 있으며 함수의 반환 값으로도 사용할 수 있다. 하지만 자바는 객체 안에서만 함수가 사용되도록 문법적으로 정해놓았기 때문에 이를 해결하기 위해서 나온 방법이 람다식이다. 람다식은 익명 함수를 표현하는 간결한 함수 표현식으로 기존의 자바 객체지향에서 함수형 프로그래밍이 가능하도록 지원하기 위해서 도입 되었으며 함수형 인터페이스를 구현하는 익명 함수로 많이 사용된다.

▶ 인터페이스를 구현하는 3가지 방식

- com.javastudy.functional.lambda

```
interface Calculator {
  int add(int a, int b);
  //int minus(int a, int b);
}

class CalculatorImpl implements Calculator {
  @Override
  public int add(int a, int b) {
    return a + b;
  }
}

public class Lambda01 {

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

```
        /* Calculator 인터페이스를 구현하는 방법은 3가지 방법이 있다.
         * 그 첫 번째 방법은 인터페이스의 구현체 클래스를 정의하고 아래와
         * 같이 구현체 클래스의 인스턴스를 생성하여 사용하는 방법이 있다.
         **/
        Calculator cal1 = new CalculatorImpl();
        System.out.println(cal1.add(10, 20));

        /* 두 번째 방법은 인터페이스를 직접 구현하는 익명 클래스 방법이 있다.
         * 첫 번째와 두 번째 방법은 이전부터 인터페이스를 구현하는 방법이다.
         **/
        Calculator cal2 = new Calculator() {
          @Override
          public int add(int a, int b) {
            return a + b;
          }
        };
        System.out.println(cal2.add(20, 30));

        /* 세 번째 방법은 Java8에서 지원하는 람다식을 이용하는 방법이다.
         * 인터페이스 구현 클래스 없이 람다식을 사용해 인스턴스를 생성할 수 있으며
         * 이 람다식은 두 번째 방식인 익명 클래스 방식을 축약한 방식으로 볼 수 있다.
         *
         * 아래 코드에서 (int x, int y) 부분이 Calculator 인터페이스의 add 함수의
         * 파라미터 입력 부분에 해당하고 x + y;는 return x + y; 부분에 해당한다.
         * 람다식을 이용해 인터페이스의 구현 클래스 없이 인스턴스를 생성하려면 해당
         * 인터페이스에 정의된 추상 메서드가 오로지 한 개만 존재해야 한다.
         **/
        Calculator cal3 = (int a, int b) -> { return a + b; };
        System.out.println(cal3.add(50, 100));
    }
}
```

▶ 함수형 인터페이스와 람다식 표현 방법

- com.javastudy.functional.lambda

```
/* @FunctionalInterface 애노테이션은 람다식으로 구현할 함수형 인터페이스를
 * 정의하는 것으로 이 애노테이션이 지정된 인터페이스는 오로지 단일 추상 메서드만
 * 정의할 수 있다. 그러므로 람다식으로 구현할 인터페이스에는 두 개 이상의 추상 메서드를
 * 정의할 수 없도록 @FunctionalInterface 애노테이션을 지정하는 것이 좋다.
 *
 * 람다식은 함수형 프로그램을 지원하기 위해서 인터페이스의 추상 메서드를 구현할 때
 * 사용할 수 있는 이름 없는 함수이며 인터페이스 안에서 정의되어 있는 추상 메서드가
 * 여러 개라면 이름을 사용할 수 없는 람다식이 어떤 추상 메서드를 구현하고 있는지 알 수
 * 없기 때문에 함수형 인터페이스에 추상 메서드는 하나만 정의 할 수 있지만 Java8부터
```

```
 * 지원하는 default 메서드와 static 메서드는 여러 개를 정의할 수 있다.
 *
 * 자바에서 메서드를 호출하려면 클래스 이름이나 참조 변수를 통해서 호출 할 수 있다.
 * 그러므로 람다식을 사용하기 위해서 함수형 인터페이스를 만들고 그 안에 람다식으로
 * 구현할 추상 메서드를 정의하고 람다식으로 구현한 이름 없는 함수를 대입해 사용한다.
 **/
@FunctionalInterface
interface Cal01{
  public void add();

  // 만약 이 인터페이스에 추상 메서드가 두개 이상이라면 람다는 사용할 수 없다.
  // public int minus(int x, int y);

  // 함수형 인터페이스에도 default 메서드와 static 메서드는 여러 개 정의 할 수 있다.
  default void defaultPrint(String msg) {
    System.out.println(msg + " 출력...");
  }

  static void staticPrint(String msg) {
    System.out.println(msg + " 출력...");
  }
}

interface Cal02{
  public void add(int a);
}

interface Cal03 {
  public int add();
}

interface Cal04{
  public double add(int a, double b);
}


public class Lambda02 {

  public static void main(String[] args) {

    // 람다식은 다음과 같이 "(매개변수) -> { 실행문 }"과 같은 구조로 작성한다.
    Cal01 cal01 = () -> { System.out.println(10 + 10); };

    // 함수의 구현부가 한 줄이면 중괄호를 생략할 수 있다.
    //Cal01 cal01 = () ->  System.out.println(10 + 10);
```

```
        cal01.add();

        // 매개 변수가 한 개 일 때 람다식은 다음과 같다
        // Cal02 cal02 = (int a) -> { System.out.println(a + 20); };

        // 매개 변수가 한 개 일 때 아래와 같이 매개 변수의 자료형을 생략 할 수 있다.
        // Cal02 cal02 = (a) -> { System.out.println(a + 20); };

        /* 매개 변수가 한 개 일 때는 괄호를 생략할 수 있고 괄호를 생략하면 매개 변수의
         * 타입을 반드시 생략해야 한다. 함수 본문이 한 줄이면 중괄호 또한 생략할 수 있다.
         **/
        Cal02 cal02 = a -> { System.out.println(a + 20); };
        // Cal02 cal02 = a ->  System.out.println(a + 20);
        cal02.add(20);

        /* 람다식의 함수 본문에서 반환 값이 있으면 return 문을 사용할 수 있고
         * return 문이 사용되면 중괄호는 생략할 수 없다.
         **/
        Cal03 cal03 = () -> { return 30 + 30; };

        /* 앞에서와 같이 함수 본문이 한 줄이면 중괄호를 생략할 수 있으며 만약 함수
         * 본문이 return 문만 있는 경우 아래와 같이 return은 생략할 수 있으며
         * 이 때 중괄호도 함께 생략해야 된다.
         **/
        //Cal03 cal03 = () -> 30 + 30;
        System.out.println(cal03.add());

        /* 람다식 함수의 매개 변수도 여러 개를 사용할 수 있고 함수 본문이 return만
         * 있는 경우 return과 중괄호를 함께 생략할 수 있다.
         */
        //Cal04 cal04 = (int a, double b) -> { return a + b; };
        Cal04 cal04 = (int a, double b) -> a + b;
        System.out.println(cal04.add(40, 40));
    }
}
```

▶ 메서드 참조

- com.javastudy.functional.lambda

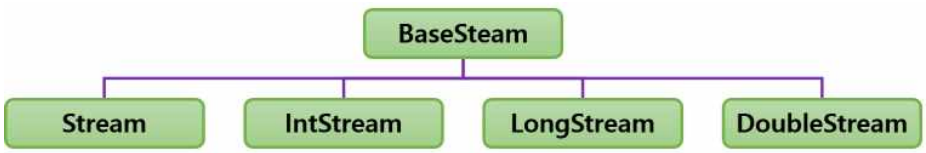
```
/* 람다식을 이용해 추상 메서드를 직접 구현하는 대신 이미 완성된 메서드를 참조해서
 * 인터페이스를 구현하는 방법이 있다. 메서드 참조에는 인스턴스 메서드 참조와
 * 스태틱 메서드를 참조하는 방식이 있다. 이외에도 추상 메서드의 첫 번째 매개 변수로
 * 넘어오는 인스턴스의 메서드를 참조하는 방법도 있다.
 **/
```

```
interface I01{
  void add();
}

class M01 {
  void plus() {
    System.out.println("plus");
  }
}

class M02 {
  static void plus() {
    System.out.println("static plus");
  }
}

public class Lambda03 {

  public static void main(String[] args) {

    // 익명 클래스 방식으로 인스턴스 메서드 사용
    I01 i01 = new I01() {
      @Override
      public void add() {
        /* M01 클래스의 인스턴스 메서드를 사용하려면
         * 먼저 인스턴스를 생성해야 한다.
         **/
        M01 m = new M01();
        m.plus();
      }
    };
    i01.add();

    // 람다식 방식으로 인스턴스 메서드 사용
    I01 i02 = () -> {
      M01 m = new M01();
      m.plus();
    };
    i02.add();

    /* 인스턴스 메서드 참조 방식으로 인스턴스 메서드 사용
     * M01 클래스의 인스턴스 메서드를 사용하려면 먼저 인스턴스를 생성해야 한다.
     * 인스턴스 메서드 참조는 더블콜론(::) 연산자(참조 연산자)를 이용해
     * 다음과 같이 인스턴스 메서드를 참조해 인터페이스를 구현할 수 있다.
     *
```

```
        * "인스턴스참조변수::인스턴스메서드"
        *
        * 아래에서 I01 i03 = m01::plus; 코드의 의미는 인터페이스 I01
        * 내부의 add() 메서드는 참조 변수 m01이 참조하는 인스턴스 내부의
        * plus() 메서드와 동일하다는 의미이다. 한 가지 주의 할 점은 이 두
        * 메서드의 매개 변수 타입과 반환 타입이 동일해야 한다는 것이다.
        **/
        M01 m01 = new M01();
        I01 i03 = m01::plus;
        i03.add();


        // 익명 클래스 방식으로 스태틱 메서드 사용
        I01 i04 = new I01() {
          @Override
          public void add() {
            // M02 클래스의 스태틱 메서드는 클래스 이름으로 접근
            M02.plus();
          }
        };
        i04.add();

        // 람다식 방식으로 스태틱 메서드 사용
        I01 i05 = () -> {
          M02.plus();
        };
        i05.add();

        /* 스태틱 메서드 참조 방식으로 스태틱 메서드 사용
        * M01 클래스의 스태틱 메서드는 클래스 이름으로 접근할 수 있다.
        * 스태틱 메서드 참조는 더블콜론(::) 연산자(참조 연산자)를 이용해
        * 다음과 같이 스택틱 메서드를 참조해 인터페이스를 구현할 수 있다.
        *
        * "클래스이름::스태틱메서드"
        *
        * 아래에서 I01 i06 = M02::plus; 코드의 의미는 인터페이스 I01
        * 내부의 add() 메서드는 클래스 M02 내부의 스태틱 메서드인 plus()
        * 메서드와 동일하다는 의미이다. 한 가지 주의 할 점은 이 두 메서드의
        * 매개 변수 타입과 반환 타입이 동일해야 한다는 것이다.
        **/
        I01 i06 = M02::plus;
        i06.add();
    }
}
```

2) 스트림

Java8에서 새롭게 등장한 스트림(Stream) API는 람다식을 사용할 수 있게 설계되어 함수형 프로그래밍을 지원하는 기술 중 하나이며 입출력을 위한 I/O 스트림과는 전혀 다른 기술이다.
스트림은 "데이터의 흐름"을 의미하는데 배열 또는 컬렉션에 대해서 여러 메서드를 단계적으로 적용하고 최종적인 결과를 만들어 가는 API가 바로 스트림 이다.
스트림은 배열 또는 컬렉션과 같은 집합 데이터에 스트림이 제공하는 메서드를 적용해 데이터를 가공하고 다음 단계로 연결해 또 다른 스트림의 메서드를 적용해 데이터를 가공하는 작업을 연속적으로 수행할 수 있다. 다시 말해 데이터가 모여 있는 원본 데이터를 바탕으로 스트림을 생성한 후 중간 단계에서 다양한 여러 작업을 연결해 연속적으로 데이터 흐름을 처리(파이프라인 연산)하고 최종적인 결과를 만들 수 있는 API이다.
스트림은 배열, 컬렉션 등의 데이터에 대해서 정렬을 하거나 필터를 적용해 필요한 데이터를 추출해서 최종적으로 숫자에 대해서 평균, 합계 등을 구하는 메서드 또는 최종 결과를 반환하거나 결과를 출력해 주는 메서드를 제공하고 있다. 그러므로 배열이나 컬렉션 등의 자료를 처리할 때 스트림을 활용하면 대상 자료가 무엇인지 상관없이 스트림의 메서드를 활용할 수 있기 때문에 일관성 있게 자료를 처리할 수 있는 장점이 있다.
스트림 작업은 대상 자료에 대해서 스트림을 생성한 후 중간 연산(intermediate operations)과 최종 연산(terminal operations) 단계로 구분해 데이터를 처리한다. 스트림에는 정렬, 필터, 데이터 변환 등의 중간 연산을 담당하는 메서드와 숫자 집계, 반복 출력, 조건에 맞는 결과 반환 등의 최종 연산을 담당하는 다양한 메서드를 지원한다.
스트림은 java.util.stream 패키지에 정의되어 있으며 아래와 같이 BaseStream 인터페이스를 기반으로 네 가지 종류를 제공하고 있다.



▶ 스트림 사용하기

- com.javastudy.functional.stream

```
public class Stream01 {

  public static void main(String[] args) {

    int[] arr = {1, 7, 4, 3, 8, 9, 2, 6, 5, 9, 4, 0, 1, 3, 6, 9, 7};

    /* 스트림 생성
     * 배열 또는 컬렉션 객체로부터 스트림을 생성할 수 있으며 이렇게 생성된
     * 스트림은 별도의 메모리에서 작업을 하므로 원본 데이터를 변경하지 않는다.
```

```
     **/
    IntStream stream1 = Arrays.stream(arr);

    /* 중간 연산
     * 스트림 연산은 중간 연산과 최종 연산이 있으며 중간 연산은 아래와 같이 여러
     * 연산을 연결하여 수행할 수 있다. 아래는 스트림에서 중복된 데이터를 제거하고
     * 이어서 데이터를 오름차순 정렬하는 중간 연산을 연속적으로 수행한 것이다.
     **/
    IntStream stream2 = stream1.distinct().sorted();

    /* 최종 연산
     * 중간 연산을 여러 번 수행하더라도 최종 연산을 수행하는 메서드가 호출되어야
     * 비로소 스트림의 연산이 적용된다. 예를 들어 데이터를 필터링하거나 정렬하는
     * 중간 연산 메서드가 호출된 후에 최종 연산을 수행하는 메서드가 호출되지 않으면
     * 필터링하거나 정렬한 결과를 가져올 수 없게된다. 또한 최종 연산은 각 요소를
     * 하나씩 순회하면서 연산을 수행하는데 이때 요소들이 "소모 된다"라고 한다.
     * 이렇게 소모된 요소는 재사용 할 수 없기 때문에 최종 연산을 수행한 스트림은
     * 다시 사용할 수 없다. 만약 다른 기능을 수행하려면 스트림을 다시 생성해야 한다.
     *
     * forEach 메서드는 현재 스트림에 있는 요소의 개수만큼 인수로 지정한
     * 람다식을 호출해 주는 최종 연산을 수행하는 메서드이다.  아래에서 이름 없는
     * 함수인 람다식은 스트림의 요소를 매개 변수로 받아서 하나씩 콘솔에 출력한다.
     **/
    stream2.forEach((i) -> System.out.printf("%s ", i < 9 ? i + ", " : i));
    //System.out.println("arr의 크기 : " + stream2.count());
  }
}
```

▶ 스트림 생성과 스트림 연산01

- com.javastudy.functional.stream

```
public class Stream02 {

  public static void main(String[] args) {

    /* 스트림 생성
     * 스트림을 생성하는 방법은 아주 다양하며 배열이나 컬렉션을 이용해 생성할 수도
     * 있고 스트림이 제공하는 여러 가지 메서드를 이용해 스트림을 생성할 수도 있다.
     **/

    /* 배열 스트림 생성
     * 배열 데이터를 Arrays.stream() 메서드를 이용해 스트림을 생성
     **/
    String[] strArr = {"Java", "HTML", "CSS", "JavaScript", "JavaFramework"};
```

```
Stream<String> strStream = Arrays.stream(strArr);

/* 컬렉션 스트림 생성
 * Collection 인터페이스에 추가된 default 메서드를 이용해 스트림을 생성
 **/
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
Stream<Integer> listStream = list.stream();

/* Stream.builder() 메서드로 스트림 생성
 * Stream 인터페이스의 스택틱 메서드인 builder() 메서드에 값을 지정해 스트림을
 * 생성할 수 있으며 마지막에 build() 메서드를 호출하면 Stream을 반환한다.
 **/
Stream<Integer> builderStream = Stream.<Integer>builder()
                                .add(8).add(3).add(2).add(9).add(5).
                                build();

/* Stream.iterate() 메서드로 스트림 생성
 * Stream 인터페이스의 스태틱 메서드인 iterate() 메서드의 인수로 초기 값과
 * 람다식을 사용해 스트림에 저정할 요소를 만들 수 있다. 아래에서 10은 초기 값이며
 * 초기 값에서 10을 더해서 반환하는 람다식을 지정해 입력 값에 10씩 증가하는 값을
 * 다음 요소의 입력 값이 되도록 하였으며 스트림의 사이즈는 5가 되도록 지정했다.
 **/
Stream<Integer> iterateStream = Stream.iterate(10, n -> n + 10).limit(5);

/* 중간 연산(intermediate operatons)
 * 중간 연산은 스트림을 생성한 후 다양한 여러 작업을 연결해 연속적으로 데이터 흐름을
 * 처리(파이프라인 연산)하는 작업을 수행한다. 중간 연산 단계에는 필터링(filtering),
 * 맵핑(mapping), 정렬(sorting), 반복 작업(iterating) 등을 수행한다.
 *
 * 최종 연산(terminal operations)
 * 최종 연산은 중간 연산을 수행한 결과를 집계(calculating), 반복 작업(iterating),
 * 최종 결과르 컬렉션 객체로 반환(collecting), 최종 결과에서 조건에 맞는 요소가
 * 있는지 체크한 결과를 반환(matching), 결과 값 환산(reduction) 등을 수행한다.
 **/
strStream.filter(s -> s.contains("Java")).forEach(System.out::println);

/* map() 메서드는 인수로 받은 람다 함수를 적용해 스트림으로 반환하는 메서드 이다.
 * 스트림의 각 요소에 20을 곱하고 내림차순 정렬해 스트림의 데이터를 콘솔에 출력한다.
 **/
listStream.map(i -> i * 20).sorted(Comparator.reverseOrder())
                           .forEach(i -> System.out.printf("%d, ", i));
System.out.println();

/* 최종 연산이 완료된 스트림은 요소가 모두 소모되어 다시 사용할 수 없다.
 * 아래 코드를 실행하면 IllegalStateException이 발생한다.
```

```
        **/
        //listStream.forEach(i -> System.out.println(i));

        // 스트림에서 홀수만 선택해 오름차순 정렬하고 결과를 리스트로 반환한다.
        List<Integer> resultList = builderStream.filter(i -> i % 2 != 0)
                                .sorted()
                                .collect(Collectors.toList());
        System.out.println(resultList);

        /* peek() 메서드는 반환 값이 없기 때문에 중간 결과를 확인하는 용도로 사용된다.
         * 스트림에서 데이터를 내림차순 정렬하고 각 요소에 10을 곱한 중간 결과를 콘솔에
         * 출력하고 확인하고 집계하기 위해 IntStream으로 변환 후 합계를 구해서 반환한다.
         **/
        int sum = iterateStream.sorted(Comparator.reverseOrder())
                    .map(n -> n * 10).peek(System.out::println)
                    .mapToInt(Integer::intValue).sum();
        System.out.println("sum : " + sum);
    }
}
```

▶ 스트림 생성과 스트림 연산02

- com.javastudy.functional.stream

```
public class Stream03 {

    public static void main(String[] args) {

        /* 기본형 스트림 생성
         * range(start, end) 메서드는 start 지점은 포함, end 지점이 포함되지 않는다.
         * rangeClosed(start, end) 메서드는 start 지점과 end 지점 모두가 포함된다.
         **/
        IntStream intStream = IntStream.range(1, 10);
        LongStream longStream = LongStream.rangeClosed(1, 1000);

        /* reduce() 메서드는 중간 연산 결과를 바탕으로 결과 값을 만들어 낼 때 사용할 수
         * 있는 최종 연산 메서드로 파라미터에 따라서 다음과 같이 3가지 종류가 있다.
         **/

        /* 인수가 하나인 reduce() 메서드 사용
         * 각 요소를 처리하는 계산 함수를 인수로 지정하면 각 요소를 순회하면서 결과를 만든다.

         **/
        OptionalInt sum1 = intStream.reduce(Integer::sum);
        System.out.println("sum1 : " + sum1.getAsInt());
```

```
    /* 인수가 두개인 reduce() 메서드 사용
     *  첫 번째 인수는 계산을 위한 초기 값, 두 번째 인수는 각 요소를 처리하는 계산 함수로
     * 각 요소를 순회하면서 중간 결과를 생성하고 마지막으로 최종 결과를 만들어 반환한다.
     **/
    long sum2 = longStream.reduce(0, (a, b) -> a + b);
    System.out.println("sum2 : " + sum2);

    // 기본형 스트림을 boxed() 메서드를 이용해 Wrapper 타입으로 박싱할 수 있다.
    Stream<Long> boxedStream = LongStream.rangeClosed(1, 1000).boxed();

    /* 인수가 세 개인 reduce() 메서드 사용
     * 병렬 처리 스트림을 연산할 때 인수가 세 개인 reduce() 메서드를 사용할 수 있다.
     * 첫 번째 인수는 초기 값, 두 번째 인수는 각 스레드가 처리하는 계산 함수, 세 번째
     * 인수는 각 스레드에서 연산한 결과를 조합해 최종 결과를 계산할 함수를 지정하면 된다.
     **/
    long sum3 = boxedStream.parallel()
        .reduce(0L,
                Long::sum,
                (a, b) -> {
                  //System.out.println("comine function called");
                  return a + b;
                });
    System.out.println("sum3 : " + sum3);

    /* Java8부터 Random 클래스는 세 가지의 타입(IntStream, LongStream,
     * DoubleStream)의 난수 스트림을 만들 수 있는 메서드를 제공한다.
     **/
    IntStream lottoStream = new Random().ints(6, 1, 46);
    lottoStream.sorted().forEach(System.out::println);

    // 스트림 인터페이스의 스태틱 메서드인 of() 메서드를 이용해 생성
    DoubleStream doubleStream =
        DoubleStream.of(9.3, 3.5, 0.3, 7.6, 4.9, 8.5, 2.1, 6.2, 1.8, 5.4);

    /* DoubleStream에 요소를 int 형으로 변환하고 Wrapper 타입으로 박싱한다.
     * 그리고 스트림을 정렬하고 map() 메서드에서 스트림의 요소를 다시 String으로
     * 변환한 후 reduce() 메서드에서 입력되는 값을 역순으로 변환해 String을 반환 한다.
     **/
    String result = doubleStream.mapToInt(n -> (int) n)
                    .boxed().sorted()
                    .map(n -> "" + n)
                    .reduce("", (a, b) -> b + a);
    System.out.println(result);
}
```

```
}
```

▶ 스트림과 람다를 사용하지 않는 경우

- com.javastudy.functional.stream

```
public class StreamAndLambdaNoUse01 {

  /* 다음과 같이 정수 20개를 저장한 배열이 있다.
   * 이 배열을 중복 데이터 없이 홀수로 역순 정렬된 배열이 되도록 만들어 보자.
   **/
  public static void main(String[] args) {

    int[] nums = {7, 2, 3, 6, 5, 4, 1, 9, 5, 3, 6, 5, 8, 1, 9, 1, 2, 8, 4, 7};

    // ArrayList에 배열에서 짝수를 추출해 저장
    ArrayList<Integer> nList = new ArrayList<>();
    for(int n : nums) {
      if(n % 2 != 0) {
        nList.add(n);
      }
    }

    // ArrayList에 있는 중복 데이터를 제거하기 위해서 HashSet으로 변경
    HashSet<Integer> set = new HashSet<>(nList);

    // HashSet을 ArrayList로 변경하고 역순으로 정렬
    ArrayList<Integer> uList = new ArrayList<>(set);
    Collections.sort(uList, Comparator.reverseOrder());

    // 다시 int형 배열로 만들기
    nums = new int[uList.size()];
    for(int i = 0; i < nums.length; i++) {
      nums[i] = uList.get(i);
    }

    // 중복 데이터 없이 홀수로 역순 정렬된 배열의 내용 확인
    System.out.println(Arrays.toString(nums));
  }
}
```

▶ 스트림과 람다를 사용하는 경우

- com.javastudy.functional.stream

```
public class StreamAndLambdaUse01 {
```

```
  /* 다음과 같이 정수 20개를 저장한 배열이 있다.
   * 이 배열을 중복 데이터 없이 홀수로 역순 정렬된 배열이 되도록 만들어 보자.
   **/
  public static void main(String[] args) {

    int[] nums = {7, 2, 3, 6, 5, 4, 1, 9, 5, 3, 6, 5, 8, 1, 9, 1, 2, 8, 4, 7};

    /* 스트림 API를 꼭 사용해야 되는 것은 아니지만 스트림 방식으로 코드를
     * 작성하면 일반적인 코드보다 간결하고 가독성이 뛰어난 코드를 작성할 수 있다.
     **/
    nums = Arrays.stream(nums)    // 배열에서 IntStream 생성
          .filter((n) -> n % 2 != 0)  // 짝수만 필터링
          .distinct()                 // 중복 제거
          .boxed()                    // 정렬을 위해 IntStream을 Stream<Integer>로 변환
          .sorted(Comparator.reverseOrder())  // 데이터 역순 정렬
          .mapToInt(Integer::intValue)        // 다시 IntStream으로 변환
          .toArray();                         // int형 배열로 변환

    // 중복 데이터 없이 홀수로 역순 정렬된 배열의 내용 확인
    System.out.println(Arrays.toString(nums));
  }
}
```