배열2

2020-04-10

이승진

**학습목표**

함수형 프로그래밍 개념 파악

배열의 메소드

**목차**

[1. 함수형 프로그래밍 2](#_Toc37371826)

[1) 자바스크립트 화살표 함수 2](#_Toc37371827)

[2) this 문제 2](#_Toc37371828)

[3) Java lambda expression 4](#_Toc37371829)

[4) 함수형 프로그래밍 (functional programming paradigm) 4](#_Toc37371830)

[5) 함수형 프로그래밍 방식의 특징 5](#_Toc37371831)

[6) 함수형 프로그래밍과 multi thread 6](#_Toc37371832)

[7) 함수형 프로그래밍 언어가 적절한 분야 6](#_Toc37371833)

[2. 배열의 메소드 8](#_Toc37371834)

[1) map 8](#_Toc37371835)

[2) filter 11](#_Toc37371836)

[3) reduce - 합계 계산 12](#_Toc37371837)

[4) reduce - 최대값 계산 14](#_Toc37371838)

[5) forEach 15](#_Toc37371839)

[6) 배열 복제 15](#_Toc37371840)

[7) every 16](#_Toc37371841)

[8) find 16](#_Toc37371842)

[9) findIndex 17](#_Toc37371843)

[10) sort 17](#_Toc37371844)

[3. 기타 19](#_Toc37371845)

[1) let, var 19](#_Toc37371846)

[2) 전역 변수 20](#_Toc37371847)

[4. 과제 21](#_Toc37371848)

# 함수형 프로그래밍

## 자바스크립트 화살표 함수

자바스크립트의 화살표 함수는 그냥 함수이다.

|  |
| --- |
| let a2 = a1.map( (value) => value \* 2 ); |

위의 코드에서 노란색으로 칠한 부분이 화살표 함수이다.

|  |
| --- |
| (value) => value \* 2 |

위의 코드를 아래의 코드로 바꿔도 된다. 즉 위와 아래의 코드가 거의 동일하다.

|  |
| --- |
| function noname(value) {  return value \* 2;  } |

화살표 함수 문법은, 자바스크립트 함수를 간결하게 구현하기 위한 문법이다.

## this 문제

function 키워드로 구현한 함수와 화살표 함수의 차이점은,

function 키워드 함수에는 this 문제가 발생할 수 있지만,

화살표 함수에서는 this 문제가 발생하지 않는다는 점이다.

화살표 함수가 콜백함수로 전달되어서 호출될 때,

화살표 함수 본문의 this는, 화살표 함수가 구현된 소스코드 문맥에서의 this 이다.

function 키워드 함수가 콜백함수로 전달되어 호출될 때,

function 키워드 함수 본문의 this는, 이 함수가 호출되는 문맥에서의 this 이다.

### callback1.js

|  |
| --- |
| let obj = {    count: 0,    startTimer: function() {      console.log(this.count);      let callback = function() { console.log(this.count++); };      setInterval(callback, 1000);    }  }  obj.startTimer(); |

출력

|  |
| --- |
| 0  NaN  NaN  NaN  NaN  ... 생략 ... |

console.log(this.count);

startTimer 메소드 내부에서 this는 obj 객체를 참조한다.

그래서 위 코드는 0을 출력한다.

let callback = function() { console.log(this.count++); };

setInterval(callback, 1000);

callback 함수는 setInterval 함수 내부에서 반복 호출된다.

function 키워드 함수의 본문에서 this는, 호출되는 문맥에서 this 이다.

callback 함수가 setInterval 함수 내부에서 호출되므로,

console.log(this.count++);

위 코드에서 this는 setInteval 함수 내부 문맥에서 this 이다.

이 this에는 count 멤버 변수가 없으므로, this.count 값은 undefined 이고,

this.count++ 값은 NaN 이다.

this 문제 발생!

### callback2.js

|  |
| --- |
| let obj = {    count: 0,    startTimer: function() {      console.log(this.count);      let callback = () => { console.log(this.count++); };      setInterval(callback, 1000)    }  }  obj.startTimer(); |

출력

|  |
| --- |
| 0  0  1  2  3  4  ... 생략 ... |

let callback = () => { console.log(this.count++); };

setInterval(callback, 1000);

화살표 함수 본문의 this는, 화살표 함수가 구현된 소스코드 문맥에서의 this 이다.

이 화살표 함수는 obj 객체의 메소드 내부에서 구현되었다.

따라서 이 화살표 함수 본문의 this는 obj 객체를 참조한다.

obj 객체에는 count 멤버 변수가 있고, 그 값이 0 이다.

따라서 화살표 함수로 구현된 콜백 함수가 setInterval 내부에서 반복 호출될 때,

0, 1, 2, 3, 값을 출력한다.

this 문제가 발생하지 않는다.

## Java lambda expression

Java 언어의 lambda expression 문법은 자바스크립트 화살표 함수 문법과 유사하다.

그렇지만, 자바스크립트 화살표 함수 문법은 함수를 만들기 위한 문법인데,

java lambda expression 문법은 함수가 아니고 객체를 만들기 위한 문법이다.

annonymous inner class 문법을 조금 더 간결하게 구현하기 위한 것이 java lambda expression 이다.

즉 java lambda expression은, 아래의 일을 실행하는 소스 코드를 간결하게 구현한 것이다.

- interface를 implements 하여 annoymous inner class를 구현한다

- 그 클래스에 메소드가 한 개 재정의된다. (interface의 메소드 재정의)

- 그 클래스의 객체를 한 개 생성한다

|  |
| --- |
| int[] a2 = Arrays.stream(a1).map( value -> value \* 2 ).toArray(); |

자바 lambda expression 문법에 대한 자세한 내용은, Java2 강의노트를 참고하라.

## 함수형 프로그래밍 (functional programming paradigm)

### 예제

|  |
| --- |
| package net.skhu.stream1;  import java.util.Arrays;  public class Example4 {  static double average1(String s) {  String[] a = s.split("[, ]+");  int sum = 0, count = 0;  for (int i = 0; i < a.length; ++i) {  int value = Integer.valueOf(a[i]);  if (value > 0) {  sum += value;  ++count;  }  }  return (double)sum / count;  }  static Double average2(String s) {  return Arrays.stream(s.split("[, ]+"))  .mapToInt(Integer::parseInt)  .filter(e -> e > 0)  .average()  .getAsDouble();  }  public static void main(String[] args) {  String s = "7, 13, -8, 6,9, 15, 0, -21,2,5";  System.out.println(average1(s));  System.out.println(average2(s));  }  } |

average1 메소드는 함수형 프로그래밍(functional programming) 방식으로 구현되었고

average2 메소드는 절차적 프로그래밍(procedural programming) 방식으로 구현되었다.

두 메소드가 하는 일은 동일하다.

(1) 파라미터로 주어진 문자열을 쪼개서 문자열 배열을 만든다.

(2) 문자열 배열의 원소를 정수로 변환한다

(3) 0 보다 큰 값들의 평균을 계산한다

Arrays.stream(s.split("[, ]+"))

s.split 메소드가 리턴하는 String[] 배열을, 스트림(stream)으로 변환한다

스트림이란 것은, 자동차 조립 공장에서 컨베이어 벨트가 돌아가는 것을 연상하면 된다.

스트림 => 컨베이어 벨트

스트림 위의 데이터 => 컨베이어 벨트 위의 조립 부품

스트림 처리 함수 => 컨베이어 벨트에서 작업하는 작업자.

스트림 위에서 데이터가 이동한다.

스트림 처리 함수는,

스트림에서 입력 받은 데이터에 대해서,

자신이 담당한 작업을 하고,

작업 결과 데이터를 스트림에 출력한다.

스트림에 출력된 데이터는, 다음 스트림 처리 함수가 받게 된다.

.mapToInt(Integer::parseInt)

입력 스트림의 값들을 하나씩 int 형으로 변환하여 출력 스트림에 출력한다.

.filter(e -> e > 0)

입력 스트림의 값들 중에서 0 보다 큰 값들만 출력 스트림에 출력한다.

.average()

입력 스트림의 값들의 평균을 계산하여 리턴한다.

.getAsDouble()

바로 앞 메소드(average)가 리턴 한 값을, doulbe 형으로 리턴한다.

## 함수형 프로그래밍 방식의 특징

### 실행 절차

구체적인 실행 절차를 프로그래머가 결정하지 않는다.

구체적인 실행 절차는

컴파일 될 때, 컴파일러에 의해서 결정될 수도 있고

실행될 때, 인터프리터/가상기계에 의해서 결정될 수도 있다.

프로그래머가 실행 절차까지 결정하지 않아도 되기 때문에,

구현하기 쉽다.

예:

|  |
| --- |
| Arrays.stream(s.split("[, ]+"))  .mapToInt(Integer::parseInt)  .filter(e -> e > 0)  .average()  .getAsDouble(); |

위 소스코드만 봐서는 구체적인 실행 절차를 알 수 없다.

전부 int 형으로 변환된 후에 filter 처리될 수도 있고,

아니면 하나씩 int 형으로 변환하자마자 filter 처리될 수도 있다.

스트림 처리 함수가 하나씩 순서대로 실행될 수 있고,

스트림 처리 함수가 멀티 스레드로 동시에 실행될 수도 있다.

## 함수형 프로그래밍과 multi thread

절차적 프로그래밍 방식에서는 multi thread 처리 절차를 프로그래머가 자세하게 구현해야 한다.

함수형 프로그래밍 방식에서는 multi thread 처리 절차를 프로그래머가 구현할 필요 없다.

예:

|  |
| --- |
| static Double average3(String s) {  return Arrays.stream(s.split("[, ]+"))  .parallel()  .mapToInt(Integer::parseInt)  .filter(e -> e > 0)  .average()  .getAsDouble();  } |

스트림 처리 중간에 .parallel() 메소드를 호출하면,

그 이후 stream 메소드들은 multi thread 에 의해서 동시에 진행된다.

절차적으로 구현된 average1 코드를 multi thread로 변경하려면, 사실상 전부 새로 구현해야 한다.

함수형 프로그래밍 방식에서는 프로그래머가 multi thread 실행 절차까지 결정하지 않아도 되기 때문에,

구현하기 쉽다.

## 함수형 프로그래밍 언어가 적절한 분야

### 함수형 프로그래밍 언어가 적절한 분야 - 빅데이터 처리

데이터 처리 절차가 복잡한 분야

대규모 병렬 처리가 필요한 분야

빅데이터 처리 엔진으로 유명한 apache spark 오픈소스 프로젝트가 scala 언어로 구현되었다.

빅데이터 처리 => 데이터 처리 절차가 복잡 + 대규모 병렬 처리가 필요

scala 언어는

java 언어와 동일한 바이트코드로 변환되어, java virtual machine 에서 실행되는

함수형 프로그래밍 언어.

### 함수형 프로그래밍 언어가 적절한 분야 - 비동기 I/O

비동기 I/O를 구현할 때, 함수형 프로그래밍 방식이 유리하다.

비동기 I/O 방식

장점 = 효율적이다

단점 = 작업 절차를 구현할 때, 어려운 프로그래밍 기법이 필요하다

함수형 프로그래밍 방식

장점 = 작업 절차를 자세하게 구현하지 않아도 되기 때문에 구현하기 쉽다.

단점 = 익숙하지 않은 개념이라서, 새로 공부해야 한다.

비동기 I/O + 함수형 프로그래밍 방식

장점 = 효율적이고, 구현하기 쉽다.

단점 = 새로 공부해야 할 것들이 많다.

### 비동기 I/O + 함수형 프로그래밍 방식의 웹 개발 기술

함수형 프로그래밍 언어를 사용한다면

Scala 언어 + Play 프레임웍 2.0

Java 언어의 lambda expression 문법을 사용하여 함수형 프로그래밍을 한다면

Spring Framework 에서도 비동기 작업 구현이 가능하다. 검색 키워드: Spring Async

Android 에서도 비동기 작업 구현이 가능하다.

Android 표준 라이브러리의 AsyncTask 클래스

rxJava 오픈소스 라이브러리

### 함수형 프로그래밍 공부를 시작하려면

(1) Java lambda expression 문법 공부

(2) Java Stream API 공부

위 두 기술은, 취업 기술 면접의 예상 문제이기도 함.

# 배열의 메소드

## map

### map1.js

|  |
| --- |
| let a1 = [1, 2, 3, 4, 5];  let a2 = a1.map((value) => value \* 2);  console.log(a2.join()); |

출력

|  |
| --- |
| 2, 4, 6, 8, 10 |

### map 메소드

a1.map((number) => number \* 2);

a1 배열의 원소 각각에 대해서, (value) => value \* 2 콜백함수를 호출한다.

이 콜백함수가 리턴하는 값들을 순서대로 배열로 모아서 리턴한다.

즉 리턴되는 배열은 [2, 4, 6, 8, 10] 이다.

a1.map((number, index) => number \* index);

a1 배열의 원소 각각에 대해서, (value, index) => value \* index 콜백함수를 호출한다.

value 파라미터는 배열의 원소 값이다.

index 파라미터는 그 원소의 배열에서 인덱스 값이다.

이 콜백함수가 리턴하는 값들을 순서대로 배열로 모아서 리턴한다.

즉 리턴되는 배열은 [0, 2, 6, 12, 20] 이다.

### join 메소드

a2.join()

a2 배열의 원소들을 결합한 문자열 한 개를 생성하여 리턴한다.

예를 들어 a2 배열의 값이 [2, 4, 6, 8, 10] 이라면,

a2.join() 메소드는, "2,4,6,8,10" 문자열을 리턴한다.

### java 예제 #1 - java8 stream API

|  |
| --- |
| package net.skhu.stream1;  import java.util.Arrays;  public class Example1 {  public static void main(String[] args) {  int[] a1 = {1, 2, 3, 4, 5};  int[] a2 = Arrays.stream(a1).map(value -> value \* 2).toArray();  System.out.println(Arrays.toString(a2));  }  } |

### java 예제 #2

|  |
| --- |
| package net.skhu.stream1;  import java.util.Arrays;  import java.util.function.IntUnaryOperator;  class MyOperator1 implements IntUnaryOperator {  @Override  public int applyAsInt(int value) {  return value \* 2;  }  }  public class Example1a {  public static void main(String[] args) {  IntUnaryOperator myOperator = new MyOperator1();  int[] a1 = {1, 2, 3, 4, 5};  int[] a2 = Arrays.stream(a1).map(myOperator).toArray();  System.out.println(Arrays.toString(a2));  }  } |

lambda exression을 사용하지 않고

앞의 java 예제 #1과 동일한 기능을 구현한 소스코드이다.

위 두 java 예제는 소스코드의 형태만 다를 뿐, 동일한 형태의 바이트코드로 컴파일된다.

anonymous inner class 문법을 활용하면, 조금 더 간결하게 구현할 수 있다.

### java 예제 #3

|  |
| --- |
| package net.skhu.stream1;  import java.util.Arrays;  import java.util.function.IntUnaryOperator;  public class Example1b {  public static void main(String[] args) {  IntUnaryOperator myOperator = new IntUnaryOperator() {  @Override  public int applyAsInt(int value) {  return value \* 2;  }  };  int[] a1 = {1, 2, 3, 4, 5};  int[] a2 = Arrays.stream(a1).map(myOperator).toArray();  System.out.println(Arrays.toString(a2));  }  } |

불필요한 지역변수 myOperator를 없애자

### java 예제 #4

|  |
| --- |
| package net.skhu.stream1;  import java.util.Arrays;  import java.util.function.IntUnaryOperator;  public class Example1c {  public static void main(String[] args) {  int[] a1 = {1, 2, 3, 4, 5};  int[] a2 = Arrays.stream(a1).map(new IntUnaryOperator() {  @Override  public int applyAsInt(int value) {  return value \* 2;  }  }).toArray();  System.out.println(Arrays.toString(a2));  }  } |

예제 #4 에서, map 메소드의 파라미터는, annonymous inner class 문법으로 구현한 객체이다.

이 부분에 lambda expression 문법을 적용하면, 소스코드를 좀 더 간결하게 만들 수 있는데,

그렇게 구현한 것이 java 예제 #1 이다.

java 예제 #1

|  |
| --- |
| value -> value \* 2 |

java 예제 #4

|  |
| --- |
| new IntUnaryOperator() {  @Override  public int applyAsInt(int value) {  return value \* 2;  }  } |

위 자바 코드들은 소스코드 형태만 다를 뿐, 동일한 형태의 바이트코드로 컴파일된다.

## filter

### filter1.js

|  |
| --- |
| let a1 = "1,2,3,4,5,6,7,8,9,10".split(",");  let a2 = a1.filter((e) => e % 2 === 0);  console.log(a2); |

출력

|  |
| --- |
| [ '2', '4', '6', '8', '10' ] |

### split 메소드

"1,2,3,4,5,6,7,8,9,10".split(",")

문자열을 쪼개서 배열을 만들어 리턴한다.

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10] 배열이 리턴된다.

### filter 메서드

a1.filter((e) => e % 2 === 0)

a1 배열의 원소 각각에 대해서, (e) => e % 2 === 0 콜백함수를 호출한다.

이 콜백함수가 true를 리턴하는 항목만 순서대로 모아서 리턴한다.

즉 리턴되는 배열은 [2, 4, 6, 7, 10] 이다.

### java8 stream API

|  |
| --- |
| package net.skhu.stream1;  import java.util.Arrays;  public class Example2 {  public static void main(String[] args) {  int[] a1 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  int[] a2 = Arrays.stream(a1).filter(value -> value % 2 == 0).toArray();  System.out.println(Arrays.toString(a2));  }  } |

## reduce - 합계 계산

### reduce1.js

|  |
| --- |
| function callback(a, b) {    console.log(a + ", " + b);    return a + b;  }  let a1 = [10, 20, 30, 40, 50];  let sum = a1.reduce(callback);  console.log(sum); |

배열의 합계를 계산한다.

출력

|  |
| --- |
| 10, 20  30, 30  60, 40  100, 50  150 |

### reduce

a1.reduce(callback)

a1 배열의 원소 각각에 대해서 callback 콜백함수를 호출한다.

호출 순서는 다음과 같다.

1회 호출 callback(10, 20)

배열의 첫째, 둘째 원소를 파라미터로 콜백 함수가 호출된다.

리턴 값은 30 이다.

2회 호출 callback(30, 30)

이전 호출의 리턴 값과, 배열의 셋째 원소를 파라미터로 콜백 함수가 호출된다.

리턴 값은 60 이다.

3회 호출 callback(60, 40)

이전 호출의 리턴 값과, 배열의 네째 원소를 파라미터로 콜백 함수가 호출된다.

리턴 값은 100 이다.

4회 호출 callback(100, 50)

이전 호출의 리턴 값과, 배열의 다섯째 원소를 파라미터로 콜백 함수가 호출된다.

리턴 값은 150 이다.

reduce 함수의 최종 리턴 값은, 마지막 콜백 함수 호출의 리턴 값이다.

즉 150 이 리턴된다.

### reduce2.js

|  |
| --- |
| let a1 = [10, 20, 30, 40, 50];  let sum = a1.reduce((a, b) => a + b);  console.log(sum); |

배열의 합계를 계산한다.

출력

|  |
| --- |
| 150 |

### java

|  |
| --- |
| package net.skhu.stream1;  import java.util.Arrays;  import java.util.OptionalInt;  public class Example2 {  public static void main(String[] args) {  int[] a1 = {10, 20, 30, 40, 50};  OptionalInt sum = Arrays.stream(a1).reduce((a, b) -> a + b);  System.out.println(sum.getAsInt());  }  } |

## reduce - 최대값 계산

### reduce3.js

|  |
| --- |
| let a1 = [7, 3, 8, 6, 9, 5, 1, 4];  let max = a1.reduce((a, b) => a > b ? a : b);  console.log(max); |

배열의 원소의 최대값을 구한다.

출력

|  |
| --- |
| 9 |

### reduce

a1.reduce((a, b) => a > b ? a : b)

a1 배열의 원소 각각에 대해서 callback 콜백함수를 호출한다.

호출 순서는 다음과 같다.

1회 호출 (7, 3) => a > b ? a : b

배열의 첫째, 둘째 원소를 파라미터로 콜백 함수가 호출된다.

리턴 값, 즉 a > b ? a : b 표현식의 값은 7 이다.

2회 호출 (7, 8) => a > b ? a : b

이전 호출의 리턴 값과, 배열의 셋째 원소를 파라미터로 콜백 함수가 호출된다.

리턴 값, 즉 a > b ? a : b 표현식의 값은 8 이다.

3회 호출 (8, 6) => a > b ? a : b

이전 호출의 리턴 값과, 배열의 네째 원소를 파라미터로 콜백 함수가 호출된다.

리턴 값, 즉 a > b ? a : b 표현식의 값은 8 이다.

4회 호출 (8, 9) => a > b ? a : b

이전 호출의 리턴 값과, 배열의 다섯째 원소를 파라미터로 콜백 함수가 호출된다.

리턴 값, 즉 a > b ? a : b 표현식의 값은 9 이다.

5회 호출 (9, 5) => a > b ? a : b

이전 호출의 리턴 값과, 배열의 여섯째 원소를 파라미터로 콜백 함수가 호출된다.

리턴 값, 즉 a > b ? a : b 표현식의 값은 9 이다.

6회 호출 (9, 1) => a > b ? a : b

이전 호출의 리턴 값과, 배열의 일곱째 원소를 파라미터로 콜백 함수가 호출된다.

리턴 값, 즉 a > b ? a : b 표현식의 값은 9 이다.

7회 호출 (9, 4) => a > b ? a : b

이전 호출의 리턴 값과, 배열의 여덟째 원소를 파라미터로 콜백 함수가 호출된다.

리턴 값, 즉 a > b ? a : b 표현식의 값은 9 이다.

reduce 함수의 최종 리턴 값은, 마지막 콜백 함수 호출의 리턴 값이다.

즉 9 가 리턴된다.

### java

|  |
| --- |
| package net.skhu.stream1;  import java.util.Arrays;  import java.util.OptionalInt;  public class Example3 {  public static void main(String[] args) {  int[] a1 = {7, 3, 8, 6, 9, 5, 1, 4};  OptionalInt sum = Arrays.stream(a1).reduce((a, b) -> a > b ? a : b);  System.out.println(sum.getAsInt());  }  } |

## forEach

### forEach.js

|  |
| --- |
| let a = [1, 2, 3, 4, 5];  a.forEach(n => console.log(n)); |

출력

|  |
| --- |
| 1  2  3  4  5 |

forEach 메소드의 파라미터는 콜백 함수이다.

forEach 메소드는 배열의 각 원소에 대해서 이 콜백 함수를 호출한다.

## 배열 복제

### slice.js

|  |
| --- |
| let a1 = [1, 2, 3, 4, 5];  let a2 = a1.slice(0);  console.log(a2);  console.log(a1 == a2); |

출력

|  |
| --- |
| [ 1, 2, 3, 4, 5 ]  false |

a1.slice(0)

a1 배열을 복제한 새 배열이 만들어져서 리턴된다.

## every

### every.js

|  |
| --- |
| let a1 = [1, 2, 3, 4];  let a2 = [0, 2, 4, 6];  let isEven = (i) => i % 2 == 0;  console.log(a1.every(isEven));  console.log(a2.every(isEven)); |

출력

|  |
| --- |
| false  true |

every 메소드의 파라미터는 콜백 함수이다.

every 메소드는 배열의 각 원소에 대해서 이 콜백 함수를 호출한다.

콜백 함수 호출이 모두 true를 리턴하면, every 메소드도 true를 리턴한다.

콜백 함수 호출이 하나다로 false를 리턴하면, every 메소드는 false를 리턴한다.

## find

### find.js

|  |
| --- |
| let persons = [    { name: "홍길동", age: 16 },    { name: "임꺽정", age: 18 },    { name: "전우치", age: 19 }  ];  let person = persons.find(p => p.age == 18);  console.log(person); |

출력

|  |
| --- |
| { name: '임꺽정', age: 18 } |

find 메소드의 파라미터는 콜백 함수이다.

find 메소드는 배열의 각 원소에 대해서 이 콜백 함수를 호출한다.

find 메소드는, 이 콜백 함수가 true를 리턴하는 첫번째 원소를 리턴한다.

## findIndex

### findIndex.js

|  |
| --- |
| let persons = [    { name: "홍길동", age: 16 },    { name: "임꺽정", age: 18 },    { name: "전우치", age: 19 }  ];  let index = persons.findIndex(p => p.age == 18);  console.log(index); |

출력

|  |
| --- |
| 1 |

findIndex 메소드의 파라미터는 콜백 함수이다.

findIndex 메소드는 배열의 각 원소에 대해서 이 콜백 함수를 호출한다.

findIndex 메소드는, 이 콜백 함수가 true를 리턴하는 첫번째 원소의 index를 리턴한다.

## sort

### sort1.js

|  |
| --- |
| let persons = [    { name: "홍길동", age: 16 },    { name: "임꺽정", age: 18 },    { name: "전우치", age: 19 }  ];  persons.sort((p1, p2) => p1.age - p2.age);  console.log(persons); |

출력

|  |
| --- |
| [  { name: '홍길동', age: 16 },  { name: '임꺽정', age: 18 },  { name: '전우치', age: 19 }  ] |

sort 메소드의 파라미터는 콜백 함수이다.

이 콜백 함수의 리턴 값을 기준으로 배열을 정렬한다.

이 콜백 함수는 두 원소를 비교하여,

첫째 파라미터가 더 크면 양수를,

둘째 파라미터가 더 크면 음수를,

두 파라미터의 값이 같으면 0을 리턴해야 한다.

### sort2.js

|  |
| --- |
| let persons = [    { name: "홍길동", age: 16 },    { name: "임꺽정", age: 18 },    { name: "전우치", age: 19 }  ];  persons.sort((p1, p2) => p1.name.localeCompare(p2.name));  console.log(persons); |

출력

|  |
| --- |
| [  { name: '임꺽정', age: 18 },  { name: '전우치', age: 19 },  { name: '홍길동', age: 16 }  ] |

# 기타

## let, var

변수를 선언할 때, var 키워드나 let 키워드를 사용한다.

|  |
| --- |
| var i1 = 0;  var s1 = "hello";  let i2 = 3;  let s2 = "world"; |

var과 let의 차이는 다음과 같다.

var 키워드에 의해서 생성되는 지역 변수는,

그 지역 변수를 포함하는 가장 가까운 함수의 본문 블럭에 소속된다.

앞에서 생성한 변수와 동일한 이름의 변수를 다시 선언해도 에러가 발생하지 않는다.

let 키워드에 의해서 생성되는 지역 변수는,

그 지역 변수를 포함하는 가장 가까운 { } 블럭에 소속된다.

앞에서 생성한 변수와 동일한 이름의 변수를 다시 선언하면 ERROR

|  |
| --- |
| var a = 3;  var a = 4; // OK  let b = 3;  let b = 4; // ERROR |

|  |
| --- |
| function test() { // 함수 본문 블럭 시작  var sum = 0;  for (var i = 1; i <= 10; ++i) {  var s = "hello";  sum = sum + i;  }  console.log(sum); // 55 출력됨  console.log(i); // 11 출력됨  console.log(s); // "hello" 출력됨  } // 함수 본문 블럭 끝 |

위 소스코드에서 sum, i, s 변수는 모두 test 함수 본문 블럭에 소속된다.

s 변수는 for 문 내부 블럭에서 생성하였지만, 함수 본문 블럭에 소속됨에 주의하자.

|  |
| --- |
| function test() {  let sum = 0;  for (let i = 1; i <= 10; ++i) { // 블럭 시작  let s = "hello";  sum = sum + i;  } // 블럭 끝  console.log(sum); // 55 출력됨  console.log(i); // 에러  console.log(s); // 에러  } |

위 소스코드에서, sum 변수는 test 함수 본문 블럭에 소속되지만,

i, s 변수는 for 문 내부 블럭에 소속된다.

따라서 for 문 외부에서 i, s 변수를 사용할 수 없다.

Java 언어의 지역 변수와 같은 문법으로 지역 변수를 사용하려면, let을 사용해야 한다.

|  |
| --- |
| function test() {  for (var i = 1; i <= 10; ++i)  var sum = (sum + i) || i;  console.log(sum); // 55 출력됨  } |

var 키워드의 특징을 이용하면, 위와 같은 구현이 가능하다.

## 전역 변수

함수나 메소드 내부가 아니고, 전역 스코프(scope)에서 변수를 생성하면,

var, let 키워드를 사용하더라도, 그 변수는 전역 변수가 된다.

# 과제

### personhw1.js

|  |
| --- |
| let persons = [    { name: "홍길동", age: 16 },    { name: "임꺽정", age: 18 },    { name: "전우치", age: 19 }  ]; |

이 배열을 age 내림차순으로 정렬하여 출력하는 코드를 구현하라.

### personhw2.js

위 persons 배열에서 age 최대값 사람을 찾아서 출력하는 코드를 구현하라.

reduce 메소드를 사용하여 구현하라.