데이터 시각화(ECMA 6 기술)

Data visualization

ECMAScript

arrow 함수

- arrow 사전적 의미: “화살, 화살표(=>)”
- function(param) {코드} 형태를 축약
- Function 키워드를 사용하지 않고 대신 화살표 사용
- 무명 / 익명(anonymous) 함수 ∴ 호출하기 위해서는 함수 표현식과 같이 변수 할당

```
(param) => { 코드 };                param => { 코드 };

() => { 코드 };                     (param1, param2, , , , paramN) => { 코드 };

param => ( {key: value} );          (param1, param2, ...rest) => { 코드 };

(param1, param2 = 123, , , paramN) => { 코드 };

([one, two] = [1, 2]) => one + two;

({key: sum} = {key: 10 + 20}) => { 코드 };
```

ECMAScript

arrow 함수

- ES5와 차이

```
var es5 = function(one, two) {
    return one + two;
}

var sum = es5(1, 2);
console.log(sum);
```

```
let es6 = (one, two)  => {
    return one + two;
};
let result = es6(1, 2);
console.log(result);
```

- 함수와 기능은 같음, 단지 작성 형태가 다를뿐
- new 연산자, arguments를 사용할 수 없고 this, setTimeout, prototype 에 대한 사용방법이 조금 달라짐

ECMAScript

arrow 함수

- 함수 블록을 사용하지 않고도 한 줄에 작성할 수 있음

```
let total = (one, two) => one + two;
let result = total(1, 2);
console.log(result);
```

- 이 때, 화살표 앞에서 줄을 분리하면 SyntaxError 가 발생

```
let total = (one, two)
=> one + two;
let result = total(1, 2);
console.log(result);
```

```
let total = (one, two) =>
 one + two;
let result = total(1, 2);
console.log(result);
```

- 파라미터가 하나이면 소괄호를 제외해도 됨
- 파라미터가 없을 경우엔 소괄호만 작성

ECMAScript

Iteration

- 사전적 의미: Iteration = 되풀이, 반복
- 반복 처리를 나타내며 이를 위한 프로토콜(Protocol)을 갖고 있음
- 프로토콜? - 약속된 기준과 방법으로 데이터를 …
- 통신 프로토콜 – 약속된 기준과 방법으로 데이터를 송수신 하는 규약
- Iteration Protocol -> Iterable / Iterator 프로토콜로 구성

Iterable Protocol (이터러블 프로토콜)

- 오브젝트의 반복 처리 규약을 정의
- String, Array, Map, Set, TypedArray, Argument 오브젝트에 대해…
  (DOM / NodeList 도 기본적으로 이터러블 프로토콜을 가지고 있음)

- 이터러블 오브젝트에는 Symbol.iterator가 존재

- 만약, 사용자가 만든 오브젝트가 있다는 가정 Symbol.iterator를 개발자가 코드에 추가하면 이터러블 오브젝트가 됨

Iterable Protocol (이터러블 프로토콜)

- Symbol.iterator의 존재 여부를 확인하는 방법

```
let arrayObj = [];
let result = arrayObj[Symbol.iterator];
console.log(result);
```

```
let objectObj = {};
let result = objectObj[Symbol.iterator];
console.log(result);
```

- 배열(array) 는 존재하고 일반 오브젝트는 존재하지 않음

ECMAScript

Iterator Protocol (이터레이터 프로토콜)

- 오브젝트의 값을 차례대로 처리할 수 있는 방법을 제공
- 이 방법은 오브젝트에 있는 next() 메서드로 구현
  따라서, 오브젝트에 next() 메서드가 있으면 이터레이터 프로토콜이 적용된 것

```
let arrayObj = [1, 2];
let iteratorObj = arrayObj[Symbol.iterator]();
console.log("1: ", typeof iteratorObj);

console.log("2: ", iteratorObj.next());
console.log("3: ", iteratorObj.next());
console.log("4: ", iteratorObj.next());
```

- 이터러블 오브젝트 :
  반복 가능한 오브젝트를 나타냄

- 이터레이터 프로토콜:
  이터러블 오브젝트의 값을 작성한 순서대로 처리하는 규약

Spread 연산자

- 이터러블 오브젝트의 엘리먼트를 하나씩 분리하여 전개하는 연산자
- 전개한 결과를 변수에 할당하거나 호출하는 함수의 파라미터 값으로 사용 가능
- "..."과 같이 점(.) 세 개를 연속해서 작성하고 이어서 이터러블 오브젝트를 작성

```
let one = [11, 12];
let two = [21, 22];
let spreadObj = [51, ...one, 52, ...two];

console.log(spreadObj);
console.log(spreadObj.length);
```

```
//만약 그냥 이렇게 변수처럼 쓴다면??
let spreadObj = [51, one, 52, two];
```

ECMAScript

Spread 연산자

- 문자열일 경우…?

```
let spreadObj = […”music”];
console.log(spreadObj);
```

- Spread는 함수 파라미터로도 사용 가능

```
const values = [10, 20, 30];
get(…values);
function get(one, two, three){
   console.log( one + two + three);
}
```

- 호출 받는 함수 파라미터에도 사용가능 (rest 파라미터라고 부름)

```
let get = (one) => {
  console.log(one);
}
get(…[1, 2, 3];
```

```
let get = (…rest) => {
   console.log(rest);
   console.log(Array.isArray(rest));
}
get(…[1, 2, 3];
```

Spread 연산자

- 일반 변수 형태와 혼합했을 때?

```
let get = (one, ...rest) => {
    console.log(one);
    console.log(rest);
}
get(...[1,2,3]);
```

- spread와 rest 파라미터는 같다?
  > 아니다,
  사용하는 방법 (...)은 같지만
  spread 파라미터는 배열을 엘리먼트로 분리, 전개 하는 것이고
  rest 파라미터는 전개된 엘리먼트를 다시 배열에 설정

Array-like

- ES 6부터 도입된 것으로 Array는 아니지만 Array처럼 사용할 수 있는 Object 객체를 Array-like라고 함

```
let values = {0: "zero", 1: "one", 2:"two", length: 3};
for (var key in values){
   console.log(key, ':' , values[key]);
}

for(var k=0; k<values.length; k++){
   console.log(values[k]);
}
```

- 프로퍼티 키 값은 0부터 1씩 증가하면서 순차적으로 작성해야함

디스트럭처링

- 디스트럭처링 사적적 의미 “~의 구조를 파괴하다”
- 아래와 같은 형태가 디스트럭처링이라고 함 (그렇기 때문에 기능 의미와는 차이가 있음)

```
let one, two;
[one, two] = [1, 2];
```

- one 변수에 1이 할당되고, two 변수에 2가 할당됨
- ES6에서의 디스트럭처링 = 분할 할당

디스트럭처링 – Array 분할 할당

- 오른쪽 배열의 엘리먼트를 분할하여 왼쪽 변수에 엘리먼트 값을 할당

```
let one, two, three, foru, five;
const values = [1,2,3];

[one, two, three] = values;
console.log("A:" , one, two, three);

[one, two] = values;
console.log("B:", one, two);

[one, two, three, four] = values;
console.log("C:", one, two, three, four);

[one, two, [three, four]] = [1, 2, [73, 74]];
console.log("D:", one,two, three, four);
```

디스트럭처링 – Object 분할 할당

- 오른쪽 Object 오브젝트 프로퍼티 단위로 분할하고 프로퍼티 키와 이름이 같은 왼쪽 변수에 값을 할당

```
let {one, two} = {one: 1, nine:9};
console.log(one, two);

let three, four;
({three, four} = {three: 3, four:4});
console.log(three, four);
```

- 오른쪽이 Object이면 왼쪽도 Object여야함
- 오브젝트 할당에서 사전에 선언된 변수를 사용하려면, 소괄호( ) 안에 할당 코드를 작성

디스트럭처링 – 파라미터 분할 할당

- 호출하는 함수의 파라미터에 Object를 작성할 수 있음
- 이 때 호출받은 함수의 파라미터를 오브젝트 분할 할당 형태로 작성하면 함수 블록에서 직접 프로퍼티 이름 사용 가능

```
function total({one, plus: {two, five}}){
    console.log(one + two + five);
}

total({one: 1, plus: {two: 2, five:5}});


var obj = {
    one : 1,
    plus: {two: 2, five: 5}
};
total(obj);
```

오퍼레이션 – 프로퍼티 이름 조합

- 문자열과 변수를 조합하여 오브젝트의 프로퍼티 이름으로 사용 가능
- 이를 Computed property name 이라고 함
- 문자열 조합

```
let item = {
  [“one” + “two”]: 12
};
console.log(item.onetwo);
```

II

```
let item = {
  onetwo: 12
};
console.log(item.onetwo);
```

- 변수 값과 문자열 조합

```
let item = "tennis";
let sports = {
    [item] : 1,
    [item + "Game"] : "윔블던",
    [item + "Method"](){
        return this[item];
    }
};
console.log(sports.tennis);
console.log(sports.tennisGame);
console.log(sports.tennisMethod());
```

오퍼레이션 – Default Value

- 변수, 파라미터, 프로퍼티에 값이 할당되지 않을 때 사전에 정의한 값이 할당

```
let [one, two, five= 5] = [1, 2];
console.log(five);

[one, two, five =5] = [1, 2, 77];
console.log(five);

let {six, seven = 7} = {six : 6};
console.log(six, seven);
```

- 파라미터도 동일한 방법으로 사전 정의한 값을 할당할 수 있음

```
let plus = (one, two = 2) => one + two;
console.log(plus(1));
console.log(plus(1, undefined));
console.log(plus(1, 70));
```

오퍼레이션 – for -of

- 이터러블 오브젝트를 반복하여 처리
- for-in 문과 차이가 없지만 대상과 방법에 차이가 있음

```
for(var value of [10, 20, 30]){
   console.log(value);
}
```

- 문자열을 반복할 수 있음

```
For(var value of “ABC”){
  console.log(value);
}
```

오퍼레이션 – for -of

- HTML 태그에 반복적인 NodeList를 반복할 수 있다.

```
<!DOCTYPE html>
<html lang=ko>
   <head>
      <meta charset="utf-8">
      <title>타이틀</title>
      <script src="4_js.js" defer></script>
   </head>
   <body>
      <ul>
         <li>첫번째</li>
         <li>두번째</li>
         <li>세번째</li>
      </ul>
   </body>
</html>
```

```
let nodes =
document.querySelectorAll("li");
for (var node of nodes){
   console.log(node.textContent);
}
```

오퍼레이션 – for -of

- in 문의 경우 Object이며, 열거 가능한 프로퍼티가 대상
  (즉, enumerable 속성 값이 false이면 반복에서 제외)

- of 문은 이터러블 오브젝트로 prototype에 연결된 프로퍼티는 대상이 아님

```
let values = [10, 20, 30];
Array.prototype.music = function(){
    return "음악";
}
Object.prototype.sports = function(){
    return "스포츠";
}

for(var key in values){
    console.log(key, values[key]);
}
```

```
console.log("<<<for-of>>>");
for(var value of values){
    console.log(value);
}
```

오퍼레이션 – 거듭 제곱 연산자

- 곱하기 문자를 연속하여 2개를 작성한 형태 (ES7 스펙에 추가됨)

```
console.log(3**2);
console.log(3**3);
console.log(Math.pow(3,3));
```