

ch06. 자바스크립트(1)

학습 목차

1. 변수
2. 자료형
3. 함수정의
4. 매개변수 정의 방법 개선
5. 다양한 함수 형태
6. 클로저
7. 객체
8. 프로퍼티 존재 확인
9. Module
10. String
11. Array
12. spread 문법(ES6)
13. destructuring 할당(ES6)

학습 목표

- 자바스크립트 함수 정의 방법을 알고 활용할 수 있습니다.
- 콜백 함수 개념을 이해하고 활용할 수 있습니다
- 클로저 함수 개념을 이해하고 활용할 수 있습니다.
- 객체 생성 방법을 이해하고 활용할 수 있습니다
- 화살표 함수를 이해하고 활용할 수 있습니다.
- Module 개념을 이해하고 활용할 수 있습니다.
- 내장 객체를 활용할 수 있습니다

1. 변수

- 변수 선언 방법
 - ECMAScript 5(ES5) 버전 – 권장하지 않음
 - 키워드 `var` 사용 : 호이스팅, 블록 단위 변수 미 지원, 중복 선언 허용 등의 문제점

```
<script>
  document.write(`value = ${value} <br>`);
  var value=10;
  document.write(`value = ${value}`);
</script>
```

```
value = undefined
value = 10
```

호이스팅 : 코드에 선언된 변수 및 함수를 코드 상단으로 끌어 올리는 것(선언 부분만)

- ECMAScript 6(ES6) 버전 – 권장
 - 키워드 `let`, `const` 사용 – 블록 변수 지원, 중복 선언 불가
 - 사용 예
 - `let` name;
 - `const` DISC = 35; //상수 선언, 값 변경 불가, 반드시 초기화

1. 변수

- let, var 차이점

```
<html>
<head>
  <meta charset="UTF-8">
  <title>유효 범위</title>
</head>
<body>
  <script>
    var x='global';
    function sub(){
      var x='local';
      console.log(x); //local
    }

    sub();
    console.log(x); //global
  </script>
</body>
</html>
```

```
<html>
<head>
  <meta charset="UTF-8">
  <title>유효 범위</title>
</head>
<body>
  <script>
    var first='global';
    {
      var second = 'local';
      document.write(first);
      document.write(second);
    }
    document.write(first);
    document.write(second);
  </script>
</body>
</html>
```

var 대신
let을
사용하면?

```
<html>
<head>
  <meta charset="UTF-8">
  <title>유효 범위</title>
</head>
<body>
  <script>
    var js1='자바 스크립트';
    var js1='javascript'; //변수 중복 선언, 오류 없음
    alert(js1);
    // let js2='scope';
    // let js2='local'; //변수 중복 선언, 오류 발생
    // alert(js2);
    // const CNT=1; //변수 CNT는 값 변경 불가
    // CNT=2; //값 변경 오류
  </script>
</body>
</html>
```

2. 자료형

- 자료형
 - number, string, boolean, undefined, null, 객체 타입, Symbol(ES6 에서 추가)
- 템플릿 문자열 - ES6에서 추가
 - 문자열 안에 값을 채우는 방법
 - 문자열을 백틱(`)으로 묶고 값이 들어가는 부분은 \${}안에 표시

```
<head>
  <meta charset="utf-8" />
  <title> 템플릿 문자열 </title>
</head>
<body>
  <script>
    let str='string';
    console.log(` ${str} (type) : ${typeof str}`); //템플릿 문자열
    str=str + 50;
    console.log(` ${str} (type) : ${typeof str}`);
    str=20;
    console.log(` ${str} (type) : ${typeof str}`);
    str='30'/2; //숫자로 자동 인식
    console.log(` ${str} (type) : ${typeof str}`);
  </script>
</body>
```

2. 자료형

• Symbol

- 심볼 값은 **유일한 값**
- **충돌 위험**이 없는 프로퍼티 키를 만들기 위해 사용 ✓
- 심볼 값 생성 - Symbol 함수 사용

1:Symbol()
2:symbol
3:false
4:exSymbol
5:Symbol(exSymbol)
6:literal object
7:30

//1. 심볼값 생성

```
const exSymbol = Symbol(); //new 연산자를 사용하면 오류
//console.log(`1: ${exSymbol}`); //error, 심볼은 자동 형 변환 되지 않음
console.log(`1:${exSymbol.toString()}`);
console.log(`2:${typeof exSymbol}`);
```

//2. 설명(디버깅용으로 사용, 심볼 값 생성에 영향을 주지 않음)이 있는 심볼값 생성

```
const exSymbol1 = Symbol('exSymbol');
const exSymbol2 = Symbol('exSymbol');
console.log(`3:${exSymbol1 === exSymbol2}`);
console.log(`4:${exSymbol1.description}`);
console.log(`5:${exSymbol1.toString()}`);
```

Symbol()
 ↳ 코 안에 문자열을 넣을 수 있음
심볼값 생성이 관여하지 않음.

// 3. 심볼과 프로퍼티 키 - 심볼을 프로퍼티 키로 사용하고 접근 시 대괄호([])사용

```
const keySymbol = Symbol('conflict');
const obj={
  [keySymbol] : 'literal object',
  keySymbol : 30
};
console.log(`6:${obj[keySymbol]}`);
console.log(`7:${obj.keySymbol}`);
```

2. 자료형

- 전역 Symbol

- Symbol.for()

- 인자로 전달받은 문자열에 해당하는 키로 저장된 심볼 반환, 없으면 생성
 - Symbol() 로 생성된 심볼 값은 키가 없음

- Symbol.keyFor()

- 심볼의 키 값 추출

```
// 3. Symbol.for()
```

```
//foo라는 키로 저장된 Symbol이 없으면 새로운 Symbol 생성
```

```
const s1 = Symbol.for('foo');
```

```
//foo라는 키로 저장된 Symbol이 있으면 해당 Symbol을 반환
```

```
const s2 = Symbol.for('foo'); → s1에서 foo라는 키로 심볼을 만들었기 때문에 foo라는 키의 Symbol은 s2이 반환.
```

```
console.log(`1:${s1} === s2`); // true
```

$s1 == s2$

1:true
2:myKey
3:undefined

```
//4. Symbol.keyFor()
```

```
const shareSymbol = Symbol.for('myKey');
```

```
const key1 = Symbol.keyFor(shareSymbol); //심볼의 키값 추출
```

```
console.log(`2:${key1}`); // myKey
```

```
const unsharedSymbol = Symbol('myKey'); //Symbol 함수로 생성된 심볼값은 키가 없음
```

```
const key2 = Symbol.keyFor(unsharedSymbol);
```

```
console.log(`3:${key2}`); // undefined
```

Symbol.for() 는 키가 있음

3. 함수 정의

- 자바 스크립트 함수는 **일급 객체**
 - 함수를 값으로 사용할 수 있음
- 함수 정의 방법
 - 함수 선언문
 - 함수 표현식
 - Function 생성자 함수
 - 화살표 함수(ES6) (람다식)
- 특징
 - 무명의 리터럴로 표현 가능
 - 변수나 자료 구조(객체, 배열...)에 저장 가능
 - 함수의 파라미터로 전달 가능
 - 반환 값(return value)으로 사용 가능

식별자 함수 이름 매개 변수 목록

```
const f= function add ( a, b ) // 식별자(변수)에 함수 리터럴 할당
{
  var sum;
  sum = a + b;
  return sum; //값 반환 시 return 키워드 사용
};
```

함수 몸체

f(23, 40); //식별자로 함수 호출
//인수와 매개변수 개수의 일치 여부를 확인하지 않음

인공객체의 특징

3. 함수 정의 - 함수 표현식

- 함수 리터럴로 생성한 함수를 함수 객체 변수에 할당
- 기명 또는 익명 함수 형식으로 사용

<pre><script> // 기명 함수 표현식 let fn = function multiply(a, b) { return a * b; }; // 익명 함수 표현식 let fa = function(a, b) { return a * b; }; let fc=fa; //동일한 익명 함수의 참조를 갖는다 console.log(fa(10, 5)); //console.log(multiply(10, 5)); // multiply is not defined, error console.log(fn(12, 5)); //식별자이름으로 호출 console.log(fc(12, 8)); </script></pre>	<pre>//함수 호출 전 함수를 선언하는 규칙을 준수하려면 //함수표현식을 사용하는 것이 좋음 console.log(fa(10, 5)); //오류 발생 - Cannot access 'fa' before initialization, 변수 호이스팅 let fa = function(a, b) { return a * b; };</pre>
---	---

3. 함수 정의 - 화살표 함수(ES6)

- function 키워드 대신 화살표 (=>)를 사용하여 함수 정의
- 함수 표현식으로 정의
- 콜백 함수로 정의할 때 유용

arrow function
40
3
undefined
6
▶ {id: '창의코딩', content: '모두의 웹'}

<script>

const arrow1 = () => console.log('arrow function'); // 매개변수가 없는 경우 ()생략 불가
arrow1();

// 매개변수가 하나인 경우() 생략 가능, 하나의 값 반환 시 return & {} 생략 가능
const arrow2 = x => x + x; //(x)=>{ return x+x; }
console.log(arrow2(20));

// 매개변수가 여러 개인 경우
const arrow3 = (a, b) => a + b;
console.log(arrow3(1, 2));

const arrow4 = (a, b) => { a + b }; // "{}"를 사용했는데 return이 없을 때
console.log(arrow4(10, 20)); // undefined

const arrow5 = (a, b) => { // 여러 줄 작성
let c = 3;
return a + b + c;
}
console.log(arrow5(1, 2));

const arrow6 = (id, content) => ({ id, content }); //객체 리터럴 반환 시 소괄호()로 감싸야 한다
//const arrow6 = (id, content) => { return { id, content }; };
console.log(arrow6('창의코딩', '모두의 웹'));

</script>

화살표 함수의 경우는 알아야 하나.. 파라미터가 아예 없으면, 생략할 수 있지..

3. 함수 정의

- 화살표 함수와 일반 함수의 차이점
 - 화살표 함수는 인스턴스를 생성할 수 없다
 - 중복된 매개변수 이름을 선언할 수 없다
 - 일반 함수는 에러가 발생하지 않음

```
<script>
function normal(a, a) { return a+a; }
console.log(normal(2, 3)); //6

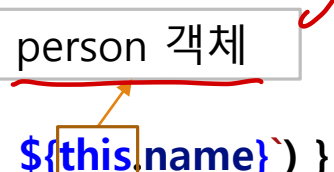
const arrow = (a, a) => { return a+a; }
//Uncaught SyntaxError: Duplicate parameter name not allowed in this context
console.log(arrow(3,4));
</script>
```

- 함수 자체의 this 바인딩을 갖지 않는다 – 객체 메소드를 화살표 함수로 정의하지 않는다

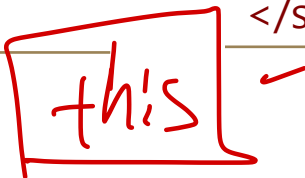
일반함수

```
<script>
const person={
  name : 'JavaScript',
  sayHi() { console.log(`Hi ${this.name}`) }
};
person.sayHi(); //Hi JavaScript
</script>
```

person 객체



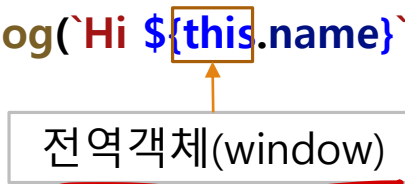
this



화살표 함수

```
<script>
const person={
  name : 'Hello',
  sayHi : () => console.log(`Hi ${this.name}`)
};
person.sayHi(); //Hi
</script>
```

전역객체(window)



3. 함수 정의

this

```
<script>
```

```
// this -> 객체 자신의프로퍼티나 메소드를 참조하기 위한 자기 참조 변수
```

```
// this 바인딩은 함수 호출방식에 따라 동적으로 결정
```

```
//1. 일반 함수(중첩 함수, 콜백 함수포함) 호출: 전역 객체(window)
```

```
function sub(){
```

```
  console.log(`sub()_this : ${this}`);
```

```
  function inSub(){ console.log(`inSub()_this : ${this}`); }
```

```
  inSub();
```

```
}
```

```
sub();
```

```
//2. 메소드 내부 호출: 메소드를 호출한 객체
```

```
const person={
```

```
  name : 'Hallym',
```

```
  getName(){
```

```
    return this.name; //this -> person 객체
```

```
  }
```

```
}
```

```
console.log(person.getName())
```

```
</script>
```

sub()_this	: [object Window]
inSub()_this	: [object Window]
Hallym	

4. 매개변수 정의 방법 개선 - ES6

- 매개변수 기본 값 *제외*

```
<head>
  <title>매개변수와 반환값이 있는 함수</title>
  <script>
    function defaultPara(a, b=20, c=30){
      return a+b+c;
    }
  </script>
</head>
<body>
  <script>
    document.write(`defaultPara(30) : ${defaultPara(30)} <br>`);
    document.write(`defaultPara(30, 20) : ${defaultPara(30, 20)} <br>`);
    document.write(`defaultPara(30, 20, 10) : ${defaultPara(30, 20, 10)} <br>`);
  </script>
</body>
```

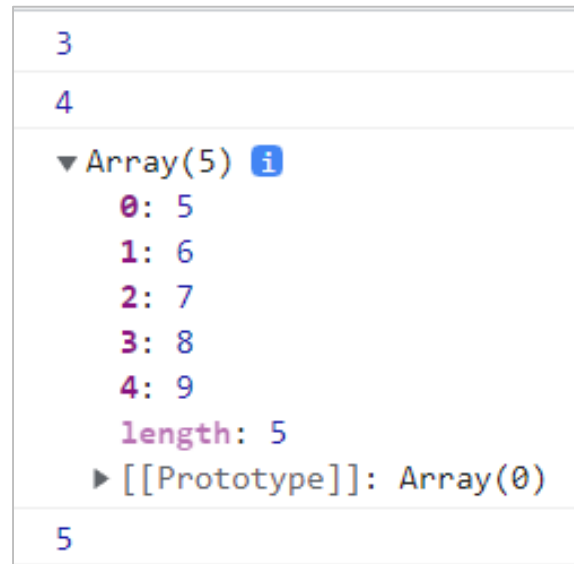
```
defaultPara(30) : 80
defaultPara(30, 20) : 80
defaultPara(30, 20, 10) : 60
```

4. 매개변수 정의 방법 개선 - ES6

- Rest 파라미터(나머지 매개변수) *나머지*
 - 매개변수 이름 앞에 세개의 점 ...을 붙여서 정의한 매개변수
 - 함수에 전달된 인수 목록을 배열로 받음
 - Rest 파라미터는 하나만 선언
 - 일반 매개변수와 함께 사용 가능 - 받은 값을 순차적으로 할당, Rest 파라미터는 반드시 마지막

```
<script>
  const bar = (p1, p2, ...rest) => {
    console.log(p1);
    console.log(p2);
    console.log(rest);
    console.log(rest.length);
  }

  bar(3, 4, 5, 6, 7, 8, 9);
</script>
```



5. 다양한 함수 형태

- 즉시 실행 함수 *즉시*
 - 함수 정의와 동시에 즉시 호출되는 함수
 - 단 한번만 호출, 다시 호출할 수 없다
 - 반드시 그룹 연산자(`(..)`)로 감싸야 함
 - 인수와 반환값도 가질 수 있음

```
<script>
    (function(){
        let sum=10+20;
        alert('즉시 실행 함수 addNumber() : ${sum}');
    }());
    document.write('즉시 실행 함수 종료<br>');
</script>
```

```
<script>
    let value = (function (a, b) {
        return a + b;
    })(34, 12));
    console.log(value);
</script>
```


5. 다양한 함수 형태

- 중첩 함수(내부 함수)
 - 함수 내부에 정의된 함수
 - 외부 함수
 - 중첩 함수를 포함하는 함수
- ✓ ◦ 중첩 함수는 외부 함수 내부에서만 호출

Window. onload가
실행될 때

```
<script>
  //외부 함수
  function outer(){
    let value=1;

    //중첩 함수 ✓
    function inner(){
      let data=2;

      //외부 함수의 변수 참조
      console.log(value + data);
    }
    inner();
  }
  outer();
</script>
```

5. 다양한 함수 형태

- 콜백 함수

- 함수의 매개변수를 통해 다른 함수의 매개변수로 전달하는 함수
- 사용 예
 - 비동기 데이터 처리, 이벤트 핸들러

- 고차 함수

- 매개변수를 통하여 함수 외부에서 콜백 함수를 전달 받은 함수

```
<script>  
function repeat(n, f) { //고차 함수  
  for(let i=0; i<n; i++){  
    f(i); //매개변수로 받은 caAll 함수 호출  
  }  
}
```

```
const caAll = function(i){ console.log(i); };
```

콜백 함수

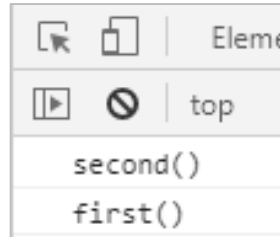
```
[ repeat(5, caAll); //콜백함수:caAll, 함수 자체를 전달해야 하므로 caAll()와 같이 작성하면 함수 호출이 됨 주의  
  repeat(6, (i) => { if(i%2) console.log(i); }); //화살표 함수를 콜백 함수로 사용한 예  
  repeat(10, function(i){ if(i%2==1) console.log(i); }); //익명 함수를 콜백 함수로 사용한 예  
</script>
```

5. 다양한 함수 형태

- 콜백 함수를 사용한 비동기 데이터 처리

//비동기 처리가 이루어지지 않은 경우

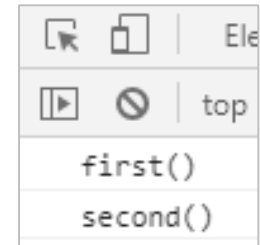
```
<script>
function first() {
  setTimeout(() => {
    console.log('first()');
  }, 1000);
}
function second() {
  console.log('second()');
}
first();
second();
</script>
```



→ = 동기처리됨.

//비동기 데이터 처리가 이루어진 경우

```
<script>
function first(callback) {
  setTimeout(function () {
    console.log('first()');
    callback();
  }, 1000);
}
function second() {
  console.log('second()');
}
first(() => second());
</script>
```



6. 클로저

- 클로저란?
 - 외부 함수보다 중첩 함수가 더 오래 유지되는 경우 중첩 함수는 생명주기가 종료한 외부 함수의 식별자를 참조할 수 있으며 이러한 중첩 함수를 클로저라 함
 - 외부 함수의 식별자를 참조하지 않으면 클로저가 아님
 - 상태(state)를 안전하게 변경하고 유지하기 위해 사용 – 전역변수사용 억제

```
<script>
  const increase = () =>{
    let num=0;
    return ++num;
  }
  console.log(increase()); //1
  console.log(increase()); //1
  console.log(increase()); //1
</script>
```

```
<script>
  const outFunc = function () {
    let num = 0;
    //클로저
    const increase = function () {
      return ++num;
    };
    return increase;
  };

  const inFunc = outFunc();
  console.log(inFunc()); //1
  console.log(inFunc()); //2
  console.log(inFunc()); //3
</script>
```

7. 객체 - 분류와 생성 방법

- 객체 분류

- 표준 빌트인 객체 – 별도의 선언없이 사용
 - Object, String, Number, Boolean, Symbol, Date, Math, Array
- 호스트 객체
 - 자바스크립트 실행 환경(브라우저)에서 추가로 제공하는 객체
 - DOM, BOM...
- 사용자 정의 객체
 - 사용자 직접 정의한 객체

객체 리터럴, 클래스, 증보

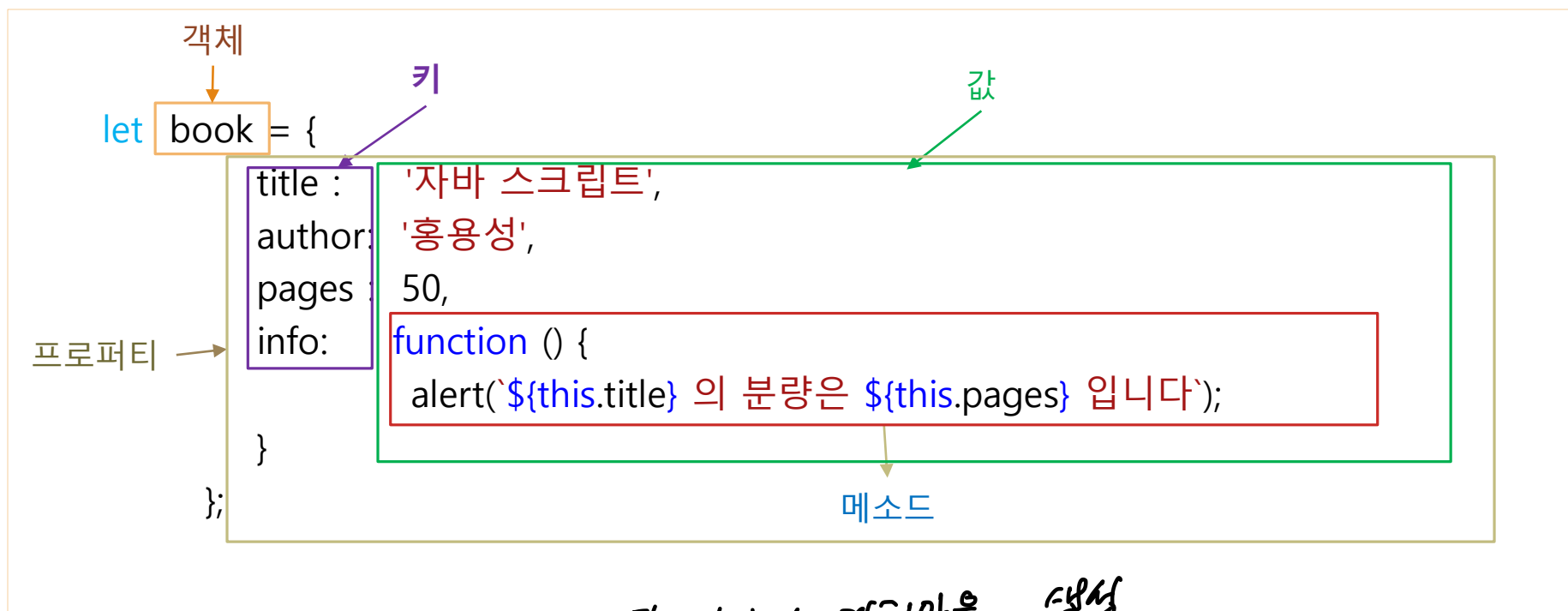
- 객체 생성 방법

- 객체 리터럴
- Object 생성자 함수
- 생성자 함수
- Object.create 메소드
- 클래스(ES6)

7. 객체 - 객체 리터럴

• 객체 리터럴

- 가장 일반적인 자바스크립트의 객체 생성 방식, 단 하나의 객체만 생성
- 중괄호({})를 사용하여 객체를 생성, {} 내에 1개 이상의 프로퍼티를 기술하며 ,(쉼표)로 구분
- { } 내에 아무것도 기술하지 않으면 빈 객체 생성



객체 리터럴 - 단 하나의 객체만을 생성
- JS에서 가장 일반적으로 쓰이는 객체 생성 방법.

7. 객체 - 객체 리터럴

- 객체 구성

- 키(key)과 값(value)으로 구성된 프로퍼티(Property)들의 집합

- 프로퍼티 키

- 빈 문자열을 포함하는 모든 문자열 또는 symbol 값
- 문자열이나 symbol 값 이외의 값을 지정하면 묵시적으로 문자열로 타입 변환
- 식별자 작성 규칙을 따르지 않을 경우 반드시 따옴표 사용

- 프로퍼티 값

- 자바스크립트에서 사용할 수 있는 모든 값과 함수
- 프로퍼티 값이 함수일 경우, 일반 함수와 구분하기 위해 메소드라 칭함

⇒ 키: 값 ✓

```
let student = {  
  's-name': 'JavaScript',  
  snumber: 202105, //유효한 식별자이므로 따옴표 생략  
  1: 10 //프로퍼티 키 1은 문자열로 변환  
};
```

7. 객체 - 객체 리터럴

- 프로퍼티 접근

- 마침표(.)와 [] 사용 ✓
- 프로퍼티 키가 유효한 식별자이면 프로퍼티 값은 마침표 표기법, 대괄호 표기법 모두 사용 } 강 대괄호 쓰심
- 프로퍼티 키가 유효한 식별자가 아니거나 예약어인 경우 대괄호 표기법 사용 } ㅋㅋ.
- 객체에 존재하지 않는 프로퍼티 접근하면 undefined 반환
- 대괄호([]) 표기법을 사용하는 경우, 대괄호 내에 들어가는 프로퍼티 키는 반드시 문자열. ✓

```
<script>
  let book = {
    'book-title': '자바 스크립트',
    author: '홍용성',
  }
```

```
console.log(book.age); //undefined
console.log(book.book-title); //title is not defined 대괄호 표기법 사용해야 함
console.log(book['book-title']);
console.log(book.author);
console.log(book[author]); //, author is not defined
console.log(book['author']);
</script>
```


7. 객체 - 객체 리터럴

- 프로퍼티 갱신 & 동적 생성 & 삭제

(person) 이라는 객체 리터럴
<script>
 let person = {
 name: 'hallym',
 address: '춘천시'
 };

function display(){ document.write('객체 메소드', '
'); }

let obj1={}; //빈 객체 리터럴 생성 ✓
obj1.name='literal'; obj1.age=30; obj1.ds = display; *객체 메소드임* //프로퍼티 동적 생성

person.name='software'; //프로퍼티 갱신
person.age = 30; //프로퍼티 동적 생성
console.log('name : \${person.name}');
console.log(person);
delete person.address; //프로퍼티 삭제, 피 연산자는 프로퍼티 키이어야 한다
console.log(person);
</script>

*person 객체 리터럴에는
age 라는 프로퍼티가
있었음
→ 그래서, 동적 생성됨*

name : software
▶ {name: "software", address: "춘천시", age: 30}
▶ {name: "software", age: 30}

7. 객체 - 객체 리터럴

- ES6에서 추가된 객체 리터럴 확장 기능
 - 프로퍼티 축약 표현

객체 리터럴 확장기능: 프로퍼티 축약표현,
메소드 축약표현

```
<script>
//ES5
let x=1, y=2;
let obj5 = {
  x:x,
  y:y
};
console.log(obj5);
```

```
▶ {x: 1, y: 2}
▶ {xx: 1, yy: 2}
```

```
//ES6
let xx=1, yy=2;
const obj6={xx, yy}; //프로퍼티 값으로 변수 사용시 키와 변수 이름이 동일하면 키 생략 가능
                        //프로퍼티 키는 변수이름으로 자동 생성
console.log(obj6);
</script>
```

구태에 쓸 일은 없습니다

7. 객체 - 객체 리터럴

- ES6에서 추가된 객체 리터럴 확장 기능
 - 메소드 축약 표현

▶ {name: 'Hallym', sayHi: f}
Hi! undefined
▶ {name: 'Hallym', sayHi: f}
✖ ▶ Uncaught TypeError: obj6.sayHi is not a constructor at object_ex01.html:29

```
<script>
//ES5
let obj5 = {
  name : 'Hallym',
  sayHi : function(){
    console.log(`Hi! ${this.name}`);
  }
};
console.log(obj5);
new obj5.sayHi(); //인스턴스 생성 가능

//ES6
let obj6 = {
  name : 'Hallym',
  sayHi(){ //메소드 축약 표현
    console.log(`Hi! ${this.name}`);
  }
};
console.log(obj6);
new obj6.sayHi(); //인스턴스 생성 불가
</script>
```

7. 객체 - 클래스(ES6)

• 클래스 정의

- 인스턴스 프로퍼티는 반드시 constructor 내부에 정의 되어야 함
 - constructor는 최대 한 개만 존재
 - 생략 가능 – 빈 constructor가 암묵적으로 정의, 빈 객체 생성
- 객체 생성 시 반드시 new 사용, 생략하면 오류
- 클래스에는 메소드만 선언
- 클래스에 선언되는 메소드는 prototype

• 클래스 특징

- 클래스는 함수
- 무명의 리터럴로 생성 가능
- 변수나 자료구조에 저장 가능
- 함수 매개변수로 전달 가능
- 함수 반환값으로 사용 가능

생성자, constructor.

```
<script>
class Student{
  //생성자 :인스턴스 생성 & 초기화
  constructor(name){
    this.name = name;
  }

  sayName(){ //프로토타입 메소드
    console.log(`My name is ${this.name}`);
  }

  static sayHello(){ //정적 메소드
    console.log('Hello!!!');
  }
}

const std=new Student('hallym');
console.log(typeof Student); //function
console.log(std.name);
std.sayName();
Student.sayHello();
</script>
```

7. 객체 - 클래스(ES6)

• 접근자 프로퍼티 → *getter(), setter()*

- 데이터 프로퍼티 값을 읽거나 저장할 때 사용하는 접근자 함수로 구성
- 메소드 이름 앞에 *get*, *set*을 사용해 정의

```
<script>
class Student{
  constructor(name){
    this.name = name;
  }

  get Name(){ //getter 함수
    return this.name;
  }

  set Name(name){ //setter 함수, 하나의 매개변수만 선언 ✓
    this.name=name;
  }
}
const std=new Student('hallym'); //인스턴스 생성
console.log(std.Name); //getter 함수 호출
std.Name = 'software'; //setter 함수 호출
console.log(std.Name);
</script>
```

```
<script>
//객체 리터럴 접근자 프로퍼티 사용 예
const student = {
  firstname: 'hallym',
  lastname:'software',
  get fullName(){
    return `${this.firstname} ${this.lastname}`;
  },

  set fullName(name){
    [this.firstname, this.lastname] = name.split(' ');
  }
}
console.log(student.fullName);
student.fullName='big data';
console.log(student.fullName);
</script>
```

7. 객체 - 클래스(ES6) ⇒ 커리, 시터

- 접근자 프로퍼티

- 데이터 프로퍼티와 접근자 프로퍼티 이름이 동일한 경우 오류 발생
- 동일한 이름을 사용하지 않도록 주의

```
// get name(){  
    return this.name;  
}  
  
set name(name){  
    this.name=name;  
}
```

```
✖ ▶ Uncaught RangeError: Maximum call stack size exceeded  
    at set name [as name] (js_class.html:23:18)  
    at set name [as name] (js_class.html:23:18)  
    at set name [as name] (js_class.html:23:18)  
    at set name [as name] (js_class.html:23:18)  
    at set name [as name] (js_class.html:23:18)  
    at set name [as name] (js_class.html:23:18)  
    at set name [as name] (js_class.html:23:18)  
    at set name [as name] (js_class.html:23:18)  
    at set name [as name] (js_class.html:23:18)  
    at set name [as name] (js_class.html:23:18)  
    at set name [as name] (js_class.html:23:18)
```

* 이거 제한이 무한순환중.
조심하셈.

8. 프로퍼티 존재 확인

- in 연산자
 - 객체 내에 특정 프로퍼티 존재여부를 확인
 - 존재하면 true, 존재하지 않으면 false
 - 상속받은 모든 프로토타입 프로퍼티도 확인
 - 사용법
- key in object => key:프로퍼티 키를 나타내는 문자열, object : 객체

```
<script>
  const Person = {
    name: 'Hallym',
    address: '춘천시'
  };

  console.log('name' in Person);
  console.log(Reflect.has(Person, 'name')); //ES6
  console.log('age' in Person);
  console.log(Person.hasOwnProperty('age'));
  console.log('toString' in Person);
  console.log(Person.hasOwnProperty('toString')); //객체 고유의 프로퍼티이면 true, 그 외는 false
</script>
```

true
true
false
false
true
false

8. 프로퍼티 존재 확인 X

```
<script>
  const person = {
    name : 'hallym',
    age : 40,
    print(){
      console.log(`name : ${this.name}, age:${this.age}`);
    }
  }

  for(let key in person){ //객체에 주로 사용
    console.log(`key : ${key}, value : ${person[key]}`);
  }

  let array = ['one', 'two'];
  array.name = 'my array';
  for (let index in array) {
    console.log(`${index} : ${array[index]}\\n`); //배열 요소 이외의 것도 함께 출력
  }

  for(let index of array){ //배열에 사용
    console.log(`${index}\\n`); //배열 요소만 출력
  }
</script>
```

- for(변수 선언문 in 객체)
 - 객체의 모든 프로퍼티를 순회하면서 열거

key : name, value : hallym
key : age, value : 40
key : print, value : print(){ console.log(`name : \${this.name}, age:\${this.age}`); }
0 : one
1 : two
name : my array
one
two

9. Module



- 여러 기능들에 관한 코드가 모여있는 하나의 파일
- 장점
 - 유지보수 용이
 - namespace
 - 재사용
- export
 - 모듈 내부의 식별자 공개 – 변수, 클래스, 함수
- import
 - 다른 모듈에서 공개한 식별자 로드
 - 다른 모듈이 export한 식별자 이름으로 import
- ES6 모듈(ESM) 파일 임을 명확히 하기 위해 확장자는 '*.mjs' 사용을 권장

9. Module - 사용 예



```
//파일명:moEx.js  
//선언문 앞에 export 키워드 사용  
export const pi=Math.PI;
```

```
export class Person{  
  constructor(name){  
    this.name = name  
  }  
  write(){ console.log(this.name); }  
}
```

```
export function Add(num1, num2) {  
  return num1 + num2;  
}
```



```
//export할 대상을 하나의 객체로 구성하여 한번에 export  
const pi=Math.PI;
```

```
class Person{  
  constructor(name){  
    this.name = name  
  }  
}
```

```
function Add(num1, num2) {  
  return num1 + num2;  
}
```

```
export {pi, Person, Add};
```

9. Module - 사용 예



```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
  <!--애플리케이션의 진입점에 해당하는 파일은 반드시 script 태그로 로드해야 함-->
  <script type="module" src="molm.js"></script>
</head>
<body>
  <h1>자바 스크립트 모듈 사용하기</h1>
</body>
</html>
```

```
import {pi, Person, Add} from './moEx.js'
```

```
//모듈이 export한 식별자 이름을 변경하여 import - as
// import {pi as PI, Person as P, Add as A} from './moEx.js'
```

```
const person = new Person('JavaScript');
//const person = new P('JavaScript');
console.log(pi);
person.write();
console.log(Add(23,45));
```

10. String

- String 생성자 함수

- new 연산자와 함께 호출 -> String 객체 생성
- new 연산자 생략 -> 기본 타입의 문자열 반환

여기서는 String 타입이랑
일반 문자열이랑 좀 다른듯.

- String 객체

- 유사 배열 객체 -> 인덱스 사용 문자 접근
- 초기값이 주어지면 내부 문자열의 수정을 할 수 없으며 메소드는 새로운 문자열을 생성하여 반환

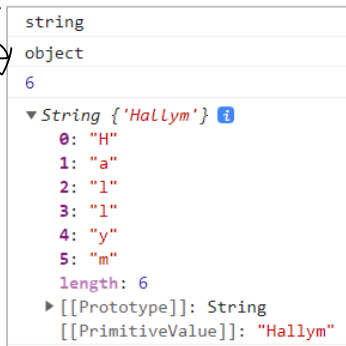
- 프로퍼티

- length : 문자열의 길이

<script>

```
let st1=String(123); //문자열 반환, 명시적 타입 변환
let st2=new String(123); //String 객체 생성, 문자열이 아닌 값을 전달하면 문자열로 강제 변환
let st3= new String('Hallym');
console.log(typeof st1);
console.log(typeof st2);
console.log(st3.length);
console.log(st3);
console.log(st3[3]);
```

</script>



10. String

- 메소드

메소드 이름	설명
charAt(index)	index에 해당하는 위치의 문자 반환
charCodeAt(index)	index에 해당하는 위치의 문자 코드 반환
concat(string, ..., string)	문자열을 연결하여 새로운 문자열 반환.
indexOf(searchString, index)	문자열을 검색하여 처음 발견된 곳의 index를 반환. 발견하지 못한 경우 -1 반환, 두번째 인수는 검색을 시작 할 위치
lastIndexOf(searchString, index)	문자열을 검색하여 마지막으로 발견된 곳의 index 반환. 발견하지 못한 경우 -1을 반환, 두번째 인수는 검색을 시작 할 위치
replace(searchString, replacement)	첫번째 인수와 일치하는 문자열을 검색하여 두번째 인수로 전달한 문자열로 대체. 원본 문자열은 변경되지 않고 결과가 반영된 새로운 문자열 반환
slice(start, end)	substring()와 동일, 단 음수 인수 전달 가능, 음수이면 문자열 뒤에서 시작
split(separator, limit)	첫번째 인수로 전달한 문자열 검색하여 구분한 후 분리된 문자열로 이루어진 배열 반환. 원본 문자열은 변경되지 않음.
substring(start, end)	첫번째 인수로 전달한 start 인덱스에 해당하는 문자부터 두번째 인자에 전달된 end 인덱스에 해당하는 문자의 바로 이전 문자까지 모두 반환
toLowerCase()	대상 문자열의 모든 문자를 소문자로 변경하여 반환
toUpperCase()	대상 문자열의 모든 문자를 대문자로 변경하여 반환
trim()	문자열 양쪽 끝에 있는 공백 문자를 제거한 문자열 반환

10. String

- 메소드

<script>

```
let exam_str1 = 'JavaScript 입문을 환영합니다';
let exam_str2 = ' jQureyMobile 입문 ';
console.log(`exam_str2.trim(): ${exam_str2.trim()}`);
console.log(`exam_str1.charAt(12) : ${exam_str1.charAt(12)}`);
console.log(`exam_str1.charCodeAt(6) : ${exam_str1.charCodeAt(6)}`);
```

```
console.log(`exam_str1.concat('내장 객체', ' 문서객체') : ${exam_str1.concat('내장 객체 ', '문서객체')}`);
console.log(`exam_str1.indexOf('입문', 3) : ${exam_str1.indexOf('입문', 3)}`);
console.log(`exam_str1.lastIndexOf('입문', exam_str1.length-1) : ${exam_str1.lastIndexOf('입문', exam_str1.length-1)}`);
console.log(`'script'.toUpperCase() : ${'script'.toUpperCase()}`);
console.log(`'JAVA'.toLowerCase() : ${'JAVA'.toLowerCase()}`);
```

```
//검색된 문자열이 여러 개 존재하면 첫번째로 검색된 문자열만 대체
console.log(`exam_str1.replace('Java', '자바') : ${exam_str1.replace('Java', '자바')}`);
console.log(`exam_str1.slice(5, 10) : ${exam_str1.slice(5, 10)}`);
console.log(`exam_str1.slice(-5) : ${exam_str1.slice(-5)}`); //뒤에서 5자리를 잘라내어 반환
```

```
console.log(`exam_str1.split(' ', 2) : ${exam_str1.split(' ', 2)}`);
console.log(`exam_str1.split(' ') : ${exam_str1.split(' ')}`);
console.log(`exam_str1.substring(3, 5) : ${exam_str1.substring(3, 5)}`);
console.log(`exam_str1.substring(0, str.indexOf(' ')) : ${exam_str1.substring(0, exam_str1.indexOf(' '))}`);
```

</script>

```
exam_str2.trim(): jQureyMobile 입문
exam_str1.charAt(12) : 문
exam_str1.charCodeAt(6) : 114
exam_str1.concat('내장 객체', ' 문서객체') : JavaScript 입문을 환영합니다내장 객체 문서객체
exam_str1.indexOf('입문', 3) : 11
exam_str1.lastIndexOf('입문', exam_str1.length-1) : 11
'script'.toUpperCase() : SCRIPT
'JAVA'.toLowerCase() : java
exam_str1.replace('Java', '자바') : 자바Script 입문을 환영합니다
exam_str1.slice(5, 10) : cript
exam_str1.slice(-5) : 환영합니다
exam_str1.split(' ', 2) : JavaScript,입문을
exam_str1.split(' ') : JavaScript,입문을,환영합니다
exam_str1.substring(3, 5) : aS
exam_str1.substring(0, str.indexOf(' ')) : JavaScript
```

10. String

- 메소드(ES6)

메소드 이름	설명
includes(searchString, index)	인수로 전달한 문자열이 포함되어 있으면 불리언 값으로 반환, 두번째 인수는 검색을 시작 할 위치
repeat(count)	인수로 전달한 숫자 만큼 반복해 연결한 새로운 문자열 반환
startsWith(searchString, index)	인수로 전달한 문자열로 시작하는지 확인하여 불리언 값으로 반환, 두번째 인수는 검색을 시작 할 위치
endsWith(searchString, index)	인수로 전달한 문자열로 끝나는지 확인하여 불리언 값으로 반환, 두번째 인수는 검색할 문자열의 길이

<script>

```
const str='Hello javaScript';
console.log(str.startsWith('He')); //true
console.log(str.startsWith('ja')); //false
console.log(str.startsWith('ja',6)); //true
console.log(str.endsWith('pt')); //true
console.log(str.endsWith('va')); //false
console.log(str.endsWith('va',10)); //true, 문자열 처음부터 10자리까지 'va'로 끝나는지 확인
console.log(`script`.repeat(3):${'script'.repeat(3)});
console.log(`str.includes(' 환영'):${str.includes(' 환영')});
```

</script>

```
true
false
true
true
false
true
'script'.repeat(3):scriptscriptscript
str.includes(' 환영'):false
```

11. Array

- 배열 생성
 - 배열 리터럴
 - Array 생성자 함수
 - Array.of 메소드(ES6)
 - ↳ • 전달된 인수를 요소로 갖는 배열 생성
 - Array.from 메소드(ES6)
 - ↳ • 인수로 받은 유사 배열 객체와 이터러블 객체를 배열로 변환하여 반환
- 배열 요소 참조
 - []표기법과 인덱스 사용
 - 존재하지 않는 요소 참조 시 undefined반환
- 프로퍼티
 - length : 배열 요소의 개수 ✓

11. Array

배열 생성

```
<script>
  let arr1 = []; //빈 배열 생성
  let arr2 = [1, 2, 3];
  let arr3 = new Array(); //빈 배열 생성
  let arr4 = new Array(5); //크기가 5인 빈 배열 생성
  let arr5 = new Array(1, 2, 3, 4, 5); //인수를 요소로 갖는 배열 생성
  console.log(arr1);
  console.log(arr2);
  console.log(arr3);
  console.log(arr4);
  console.log(arr5);

  let dim1 = Array.of(1, 2, 3);
  let dim2 = Array.from({ 0: 'a', 1: 'b', 2: 'c', length: 3 });
  let dim3 = Array.from({ length: 2 }); //undefined 요소로 채움

  //두 번째 인수로 전달한 콜백 함수의 반환값으로 구성된 배열 반환
  let dim4 = Array.from({ length: 3 }, (v, i) => i+1);
  console.log(dim1);
  console.log(dim2);
  console.log(dim3);
  console.log(dim4);
</script>
```

```
▼ [] ⓘ
  length: 0
  ► [[Prototype]]: Array(0)

▼ (3) [1, 2, 3] ⓘ
  0: 1
  1: 2
  2: 3
  length: 3
  ► [[Prototype]]: Array(0)

▼ [] ⓘ
  length: 0
  ► [[Prototype]]: Array(0)

▼ (5) [empty × 5] ⓘ
  length: 5
  ► [[Prototype]]: Array(0)

▼ (5) [1, 2, 3, 4, 5] ⓘ
  0: 1
  1: 2
  2: 3
  3: 4
  4: 5
  length: 5
  ► [[Prototype]]: Array(0)
```

```
▼ (3) [1, 2, 3] ⓘ
  0: 1
  1: 2
  2: 3
  length: 3
  ► [[Prototype]]: Array(0)

▼ (3) ['a', 'b', 'c'] ⓘ
  0: "a"
  1: "b"
  2: "c"
  length: 3
  ► [[Prototype]]: Array(0)

▼ (2) [undefined, undefined] ⓘ
  0: undefined
  1: undefined
  length: 2
  ► [[Prototype]]: Array(0)

▼ (3) [1, 2, 3] ⓘ
  0: 1
  1: 2
  2: 3
  length: 3
  ► [[Prototype]]: Array(0)
```

11. Array *[i] : length*

- 유사 배열 객체

- 키가 인덱스 값으로 되어있고 길이를 나타내는 length 속성을 갖는 객체

- 이터러블 객체

- Symbol.iterator 메소드를 가지고 있는 객체
- 빌트인 이터러블 : Array, String, DOM.....
- for ... of문으로 순회, 스프레드 문법과 배열 디스트럭처링 할당 가능
- for(변수 선언문 of 이터러블) ✓

```
<script>
  const arrayLike={ 0:'a', 1:'b', 2:'c', length : 3 } //유사배열 객체
  const iterable = [3,4,5,6,7]; //이터러블 객체

  console.log(`arrayLike : ${Symbol.iterator in arrayLike}`); //false
  console.log(`iterable : ${Symbol.iterator in iterable}`); //true

  console.log('유사배열 객체(for~of)')
  //오류 발생 : Uncaught TypeError: arrayLike is not iterable
  //유사 배열 객체는 for ... of 사용 불가
  // for(let data of arrayLike){
  //   console.log(data);
  // }

  console.log('이터러블 객체(for~of)')
  for(let data of iterable){
    console.log(data); //3, 4, 5, 6, 7
  }
</script>
```

11. Array

- 원소 추가 & 삭제

<script>

```
let dim1 = new Array(); //빈 배열 생성, 또는 let dim1=[];  
let dim2 = new Array(5); //크기가 5인 빈 배열 생성
```

//배열 요소 추가- 서로 다른 타입을 원소로 가질 수 있다

//동적으로 배열 요소 추가 가능

//단, 값이 할당되지 않은 인덱스 위치의 요소는 생성되지 않으며 배열 길이는 마지막 인덱스를 기준으로 계산.

```
dim1[1] = 'css';
```

```
dim1[2] = 30; dim1[4] = true;
```

```
console.log(`dim1.length = ${dim1.length}`);
```

```
for (let i=0; i< dim1.length;i++) {
```

```
    console.log(dim1[i]); //생성되지 않은 요소는 undefined
```

```
}
```

//배열 요소 삭제- 값만 삭제 됨(권장하지 않음)

```
delete dim1[2];
```

```
console.log(`요소 삭제 후 결과 : ${dim1}`);
```

//해당 요소를 완전히 삭제(권장)

```
dim1.splice(2, 1);
```

```
console.log(`요소 삭제 후 결과 : ${dim1}`);
```

</script>

dim1.length = 5
undefined
css
30
undefined
true
요소 삭제 후 결과 : ,css,,,true
요소 삭제 후 결과 : ,css,,true

11. Array

• 메소드

메소드 이름	설명
<i>shift()</i>	배열의 첫 원소를 제거하여 반환
<i>unshift()</i>	배열의 첫 원소에 새로운 원소를 추가하여 반환
concat(items)	인수로 전달된 요소를 원본 배열의 마지막 요소로 추가한 새로운 배열 반환
join(separator)	배열의 모든 원소를 separator로 연결한 문자열 반환, 구분자가 생략되면 ,(coma)가 기본 구분자
<i>pop()</i>	배열의 마지막 원소를 제거하여 반환
<i>push(item)</i>	배열의 마지막 부분에 item 요소 추가하고 변경된 length값 반환
reverse()	배열 원소 순서를 반대로 정렬하여 반환
slice(start, end)	start 인덱스부터 end에 해당하는 인덱스 전까지 복사하여 반환, start가 음수이면 배열 끝에서의 인덱스
<i>splice(start, deleteCount, item)</i>	start : 시작 위치, start만 지정하면 모든 요소 제거 deleteCount : 시작위치부터 제거할 요소 개수-생략 가능, item : 삭제한 위치에 추가 될 요소 - 생략 가능
indexOf(search, index)	인수로 전달된 search를 검색하여 인덱스 반환, 두번째 인수는 검색을 시작할 인덱스 중복 요소가 있으면 첫번째 인덱스, 없으면 -1반환
includes(search, index)	인수로 전달된 search 검색, 두번째 인수는 검색을 시작할 인덱스 요소가 있으면 true, 없으면 false 반환(indexOf() 대신 사용 권장)
<i>fill(items, start, end)</i>	인수로 전달된 items을 배열 요소로 채움, start 인덱스부터 end 인덱스 전까지 items로 채움

이탤릭체로 표기된 메소드는 원본 배열을 변경함

11. Array

- 메소드

1)월요일,화요일,수요일,목요일,금요일,토요일,일요일
2)Script
3)34,12,true,100
4)12,true,100
5)node.js,Spring,12,true,100
6)12,true
7>true,100
8)t-p-i-r-c-S-a-v-a-j- -o-l-l-e-H

```
<script>
let dim1 = new Array(34, 12, true, 'Script');
let dim2 = new Array('월요일', '화요일', '수요일');
let dim3 = new Array('목요일', '금요일', '토요일', '일요일');

console.log(`1)${dim2.concat(dim3)}`); //두 개의 배열을 결합한 결과 반환
console.log(`2)${dim1.pop()}`); //배열 마지막 원소 반환

dim1.push(100); //배열 마지막에 원소 추가
// dim1[dim1.length]=100; //push()와 동일, length를 사용하는 것이 성능이 좋음
console.log(`3)${dim1}`);

dim1.shift(); //배열의 첫 원소 제거한 후 반환
console.log(`4)${dim1}`);

dim1.unshift('node.js', 'Spring'); //배열 첫 원소에 새로운 원소 추가하여 반환
console.log(`5)${dim1}`);

let dim_slice = dim1.slice(2, 4); //부분 배열 반환
console.log(`6)${dim_slice.toString()}`);

dim_slice = dim1.slice(-2); //배열의 끝에서 요소 반환
console.log(`7)${dim_slice.toString()}`);

const str='Hello javaScript';
const str_reverse = str.split('').reverse().join('-'); //메소드 체이닝
console.log(`8)${str_reverse.toString()}`);
</script>
```

11. Array

- 메소드

```
<script>
const item1 = [1, 2, 3, 4];
const res1 = item1.splice(1, 2); // item1[1]부터 2개의 요소를 제거하고 제거된 요소를 배열로 반환.
console.log(item1); // [ 1, 4 ], 원본 배열이 변경된다
console.log(res1); // [ 2, 3 ], 제거한 요소를 배열로 반환

const item2 = [1, 2, 3, 4];
// item2[1]부터 2개의 요소를 제거하고 그 자리에 새로운 요소를 추가한다. 제거된 요소가 반환.
const res2 = item2.splice(1, 2, 20, 30);
console.log(item2); // [ 1, 20, 30, 4 ], 원본 배열이 변경된다
console.log(res2); // [ 2, 3 ], 제거한 요소를 배열로 반환

const arr = [1, 2, 2, 3, 5];
// 배열 arr 처음부터 요소 2를 검색하여 인덱스 반환, 중복되는 값은 가장 먼저 검색한 인덱스 반환
console.log(arr.indexOf(2)); // -> 1
// 두번째 인수는 검색을 시작할 인덱스이다
console.log(arr.indexOf(2, 2)); // 2
console.log(arr.includes(2)); //true
console.log(arr.includes(10)); //false

arr.fill(10); //전체 배열 요소를 10으로 채움
console.log(arr);
arr.fill(6, 1); //인덱스 1부터 끝까지 6으로 채움
console.log(arr);
arr.fill(4, 2, 4); //인덱스 2부터 3까지 4로 채움
console.log(arr);
</script>
```

▶ (2) [1, 4]
▶ (2) [2, 3]
▶ (4) [1, 20, 30, 4]
▶ (2) [2, 3]
1
2
true
false
▶ (5) [10, 10, 10, 10, 10]
▶ (5) [10, 6, 6, 6, 6]
▶ (5) [10, 6, 4, 4, 6]

11. Array

고차함수 `sort()`

- 고차 함수

- `sort()`

- 정렬된 배열 반환(오름차순) - 원본 배열 변경
 - 숫자 자료 정렬 - 정렬 순서를 정의하는 비교 함수를 인수로 전달

```
<script>
const fruits = ['Banana', 'Orange', 'Apple'];
fruits.sort();
console.log(fruits.toString());

const points = [40, 100, 1, 5, 2, 25, 10];
points.sort();
console.log(points); // [ 1, 10, 100, 2, 25, 40, 5 ], 유니코드 값으로 비교

// 숫자 배열 오름차순 정렬 - 비교 함수의 반환 값이 0보다 작은 경우, a를 우선하여 정렬.
points.sort((a, b) => a - b);
console.log(points); // [ 1, 2, 5, 10, 25, 40, 100 ]

// 숫자 배열 내림차순 정렬 - 비교 함수의 반환 값이 0보다 큰 경우, b를 우선하여 정렬.
points.sort((a, b) => b - a);
console.log(points); // [ 100, 40, 25, 10, 5, 2, 1 ]
</script>
```

11. Array

고차함수 for Each()

- 고차 함수

- forEach()

- 배열을 순회하며 배열의 각 요소에 대하여 인자로 주어진 콜백 함수 실행, 반환값은 없음
- 콜백 함수의 인수를 통해 배열 요소의 값, 인덱스, 메소드를 호출한 배열을 전달
- forEach 메소드는 원본 배열(this)을 변경하지 않으나 콜백 함수는 원본 배열(this)을 변경할 수는 있다.
- break, continue 사용 불가
- 가독성이 좋다 - 권장

```
<script>
```

```
const dim=[1,2,3];
```

```
const pows=[];
```

```
dim.forEach(item=>pows.push(item**2));  
console.log(pows);
```

//콜백 함수 인수 : 요소값, 인덱스, forEach 메소드를 호출한 배열

```
pows.forEach((item,index, arr) => arr[index]=item**2); //콜백 함수에서 원본 배열 변경
```

```
console.log(pows);
```

```
</script>
```

dim 배열을 순회하면서
각 인덱스에 연산으로
결과를 적용

▶ (3) [1, 4, 9]

▶ (3) [1, 16, 81]

제곱 (square)

콜백이 원본 pows 배열은
변경시킴.

11. Array

고차함수 `map()`, `filter()`

고차 함수

`map()` - 배열 순회, 원본 배열 변경 되지 않음 ✓

- 배열 요소에 대하여 콜백 함수의 반환값으로 새로운 배열을 생성하여 반환.
- 콜백 함수의 인수를 통해 배열 요소의 값, 인덱스, 메소드를 호출한 배열 전달

`filter()` - 배열 순회, 원본 배열 변경 되지 않음 ✓

- 배열 요소에 대하여 콜백 함수의 실행 결과가 true인 요소 값만을 추출한 새로운 배열 반환 ✓
- 콜백 함수의 인수를 통해 배열 요소의 값, 인덱스, 메소드를 호출한 배열 전달

<script>

```
const number = [1, 4, 9];  
// const roots = number.map(item => Math.sqrt(item));  
const roots = number.map((item, index, arr) => {  
  console.log(`요소값 : ${item}, 인덱스 : ${index}, this : ${arr[index]}`);  
  return Math.sqrt(item);  
});
```

```
console.log(roots);  
console.log(number);
```

```
const odd = number.filter(item => item%2);  
console.log(odd);
```

```
const dim=[1,2,1,3,5,2,7,4,7,4];
```

```
const result =dim.filter((v, i, arr) => arr.indexOf(v) === i);  
console.log(result); //중복요소 제거
```

</script>

요소값 : 1, 인덱스 : 0, this : 1
요소값 : 4, 인덱스 : 1, this : 4
요소값 : 9, 인덱스 : 2, this : 9
▶ (3) [1, 2, 3]
▶ (3) [1, 4, 9]
▶ (2) [1, 9]
▶ (6) [1, 2, 3, 5, 7, 4]

그게 === i니까
중복되지 않는것들 +
중복되더라도 첫번째만은
이 === i 이기 때문에
filter 특성으로
true인 애들만
arr에 들어감.
(=> 중복제거가능)

11. Array

- 고차 함수

- find() - 배열 순회

- 배열 요소에 대하여 콜백 함수를 실행하여 결과가 참인 첫번째 요소 반환
 - 참인 요소가 없으면 undefined 반환
 - 콜백 함수의 인수를 통해 배열 요소의 값, 인덱스, 메소드를 호출한 배열 전달

- findIndex() - 배열 순회

- 배열 요소에 대하여 콜백 함수를 실행하여 그 결과가 참인 첫번째 요소의 인덱스 반환
 - 참인 요소가 존재하지 않는다면 -1 반환
 - 콜백 함수의 인수를 통해 배열 요소의 값, 인덱스, 메소드를 호출한 배열 전달

```
<script>
```

```
const users = [  
  { id: 1, name: 'Lee' },  
  { id: 2, name: 'Kim' },  
  { id: 2, name: 'Choi' },  
  { id: 3, name: 'Park' }  
];
```

```
// 콜백 함수를 실행하여 그 결과가 참인 첫번째 요소를 반환  
let result = users.find(item => item.id === 2);  
console.log(result); // { id: 2, name: 'Kim' }  
result = users.find(item => item.id === 10);  
console.log(result); // undefined
```

```
// filter()는 새로운 배열 반환 ✓  
result = users.filter(item => item.id === 2);  
console.log(result); // [{ id: 2, name: 'Kim' }, { id: 2, name: 'Choi' }]
```

```
let index = users.findIndex(item => item.id === 2);  
console.log(index);  
index = users.findIndex(item => item.id === 10);  
console.log(index);
```

```
</script>
```

▶ {id: 2, name: 'Kim'}
undefined
▼ (2) [{...}, {...}] i
▶ 0: {id: 2, name: 'Kim'}
▶ 1: {id: 2, name: 'Choi'}
length: 2
▶ [[Prototype]]: Array(0)
1
-1

11. Array

- 고차 함수

- reduce()

- 배열을 순회해 각 요소에 대해 **이전의 콜백 함수 실행 반환 값**을 전달해 콜백 함수를 실행하고 그 결과를 반환
 - 원본 배열 변경 없음

배열명.reduce(function(acc, cur, index, array) { } [, initialValue])

- 초기값 또는 콜백함수 이전 반환값(acc)
- reduce()를 호출한 배열 요소 값 (cur)과 인덱스(index)
- reduce()를 호출한 배열 (arr)
- initialValue(생략가능) : callback의 최초 호출에서 **첫 번째 인수에 제공하는 값**. 초기값을 제공하지 않으면 **배열의 첫 번째 요소를 사용**

11. Array

- 고차함수 – `reduce()` 사용 예

```
const numbers = [1, 2, 3, 4];
```

```
const result = numbers.reduce((number1, number2) => number1 + number2);
```

```
/*  
1,2 => 3 배열값 1번째, 2번째 부터 시작  
3,3 => 6  
6,4 => 10  
*/
```

```
console.log(result); // 10;
```

```
const numbers = [1, 2, 3, 4];
```

```
const result = numbers.reduce((number1, number2) => number1 + number2, 10);
```

```
/* 초기값을 10으로 초기화하여  
10,1 => 11 initialValue값, 배열 값 1번째 부터 시작  
11,2 => 13  
13,3 => 16  
16,4 => 20  
*/  
console.log(result); // 20;
```

```
const initialValue = 0;
```

```
const list = [ { x : 1 }, { x : 2 }, { x : 3 } ];
```

```
let sum = list.reduce(
```

```
function (accumulator, currentValue) { return accumulator + currentValue.x; }, initialValue  
);
```

```
console.log(sum) // logs 6
```

12. spread 문법(ES6)

- 스프레드 문법(spread syntax) ...
 - 하나로 뭉쳐있는 여러 값들의 집합을 개별적인 값들의 목록으로 생성
 - 단, Array, String, Map, Set, DOM 컬렉션, arguments 에 사용/ - 이터러블 객체에 한정
 - 사용
 - 함수 호출문 인수 목록 ✓ 함수 파라미터
 - 배열 리터럴 요소 목록
 - 객체 리터럴의 프로퍼티 목록

```
<script>  
  //1. 함수의 인수로 사용 ✓  
  function foo(x, y, z) {  
    console.log(x); // 1  
    console.log(y); // 2  
    console.log(z); // 3  
  }
```

여러 값들의 집합을
↓
개별적인 값들의 목록으로 생성.

```
  const arr = [1, 2, 3];  
  //...[1, 2, 3]는 [1, 2, 3]을 개별 요소로 분리한다(→ 1, 2, 3)  
  // spread 문법에 의해 분리된 배열의 요소는 개별적인 인자로서 각각의 매개변수에 전달  
  foo(...arr);  
</script>
```

12. spread 문법(ES6)

<script>

//2. 배열에서 사용 ✓

```
const arr = [1, 2, 3];  
console.log([...arr, 4, 5, 6]); //== arr.concat([4, 5, 6])
```

```
const arr1 = [1, 2, 3];  
const arr2 = [4, 5, 6];  
arr1.push(...arr2); // == arr1.push(4, 5, 6);  
console.log(arr1); // [ 1, 2, 3, 4, 5, 6 ]
```

```
const arr3 = [1, 2, 3, 6];  
const arr4 = [4, 5];  
arr3.splice(3, 0, ...arr4); // == arr3.splice(3, 0, 4, 5);  
console.log(arr3); // [ 1, 2, 3, 4, 5, 6 ]
```

```
const arr5 = [1, 2, 3];  
const copy = [...arr5];  
console.log(copy); // [ 1, 2, 3 ]
```

```
copy.push(4); // copy를 변경한다.  
console.log(copy); // [ 1, 2, 3, 4 ]  
console.log(arr5); // [ 1, 2, 3 ], // arr5는 변경되지 않는다.
```

</script>

스프레드와
객체 리터럴
어디서 사용

<script>

//3. 객체 리터럴에 사용 ✓

```
const n = { x: 1, y: 2, ...{ a: 3, b: 4 } };  
console.log(n); // { x: 1, y: 2, a: 3, b: 4 }  
// 객체의 병합  
const merged = { ...{ x: 1, y: 2 }, ...{ y: 10, z: 3 } };  
console.log(merged); // { x: 1, y: 10, z: 3 }
```

```
// 특정 프로퍼티 변경  
const changed = { ...{ x: 1, y: 2 }, y: 100 };  
// changed = { ...{ x: 1, y: 2 }, ...{ y: 100 } }  
console.log(changed); // { x: 1, y: 100 }
```

```
// 프로퍼티 추가  
const added = { ...{ x: 1, y: 2 }, z: 0 };  
// added = { ...{ x: 1, y: 2 }, ...{ z: 0 } }  
console.log(added); // { x: 1, y: 2, z: 0 }
```

</script>

13. destructuring 할당(ES6)

- 구조화된 배열 또는 객체를 destructuring 하여 개별적인 변수에 할당하는 것
- 배열 또는 객체 리터럴에서 필요한 값만을 추출하여 변수에 할당하거나 반환할 때 유용

<script>

```
const arr = [1, 2, 3];
```

//1. 배열의 인덱스를 기준으로 배열로부터 요소를 추출하여 변수에 할당, 디스트럭처링을 사용할 때는 반드시 초기화
// const [one, two, three]; -> SyntaxError: Missing initializer in destructuring declaration

```
const [one, two, three] = arr;  
console.log(one, two, three); // 1 2 3
```

```
let x, y, z;  
[x, y] = [1, 2]; console.log(x, y); // 1 2  
[x, y] = [1]; console.log(x, y); // 1 undefined  
[x, y] = [1, 2, 3]; console.log(x, y); // 1 2
```

```
const obj = { firstName: 'Ungmo', lastName: 'Lee' };  
//2. 프로퍼티 키를 기준으로 디스트럭처링 할당  
// 변수 lastName, firstName가 선언되고 obj가 Destructuring되어 할당  
const { lastName, firstName } = obj;  
console.log(firstName, lastName); // Ungmo Lee
```

</script>

13. destructuring 할당(ES6) *중요*

```
<script>
```

```
//3. 객체 프로퍼티 키로 필요한 값만 추출하여 변수에 할당
```

```
const str="hallym";
```

```
const {length} = str;
```

```
const per ={id:2022, content:'HTML'};
```

```
const {id} = per;
```

```
console.log(id);
```

```
//4. 함수 매개변수 활용
```

```
function print({value, completed}){
```

```
  console.log(`value : ${value}, 상태 : ${completed? '완료':'실행중'}`);
```

```
}
```

```
  print({id:1, value:'HTML', completed:true});
```

```
</script>
```

Q & A

- 자바스크립트에 대한 학습이 모두 끝났습니다.
- 모든 내용을 이해 하셨나요?
- 아직 이해가 안되는 내용이 있다면 다시 한번 복습하시기 바랍니다.
- 질문은 한림 SmartLEAD 쪽지 또는 e-mail 또는 전화상담을 이용하시기 바랍니다.
- 과제가 제출되었습니다.
- 다음 시간에는 자바스크립트 DOM 에 대하여 알아보도록 하겠습니다
- 수고하셨습니다.^ ^