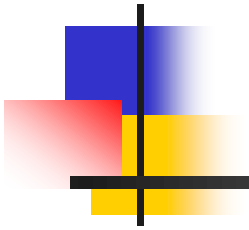


javascript 4강

37강.

양 명 숙
[now4ever7@gmail.com]

목차

- 익명함수
- 매개변수
- 가변인자 함수
- 콜백함수
- 함수를 리턴하는 함수
- 클로저 함수
- ES6

변수에 함수 저장하기

```
함수 표현식
var add = function(a, b) {
  return a + b;
};

var result = add(10, 20);
```

- 자바 스크립트의 기본 자료형
  - 숫자, 문자열, 논리형, 객체, 함수, undefined
  - 자바스크립트의 기본 자료형에는 배열이 없다, 배열도 객체
- 함수도 변수에 넣을 수 있는 데이터 값임

```
//var name='홍길동';  //변수에 문자열 데이터 넣기

function hello(name){
        document.write(name+"님 환영합니다.<br>");
}

hello('김길동');

var test=hello;  //변수에 함수 넣기
test('이길동');
```

  - 변수에 함수를 저장하면 변수를 함수처럼 사용할 수 있다.

- 함수를 선언하는 방법
1) 함수 표현식
2) 함수 선언문

익명 함수

- 함수 – 코드의 집합을 나타내는 자료형
  - 자바스크립트에서 함수는 하나의 자료형
- 익명 함수 (함수 표현식)
  - 함수이지만 이름이 없으므로 익명함수라고 부름
  - 이름이 없으므로 변수에 넣어서 사용

```
var 함수 = function() { ... }
```

- 선언적 함수 (함수 선언문)
  - 이름이 있는 함수

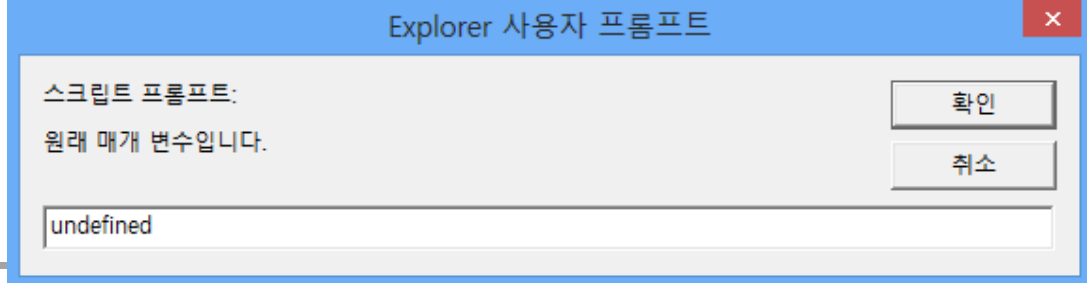
```
function 함수() { ... }
```

```
<script>
      // 변수를 생성합니다.
      var str = function () {
          var output = prompt('숫자를 입력해주세요.', '숫자');
          alert(output);
      };

      str();
  </script>
```

```
var sumNumbers = function(a, b) {
  return a + b;
};

var result=sumNumbers(10,20);
alert(result);  //30
```

```
function sumNumbers(a, b) {
  return a + b;
}
```

매개변수



- 자바스크립트는 함수를 생성할 때 지정한 매개변수보다 많거나 적은 매개변수를 사용하는 것을 허용함
- 예)

```
<script>
        // 함수를 호출합니다.
        alert('원래 매개 변수입니다.', '추가된 매개 변수입니다.'); //원래 함수에서 선언된 매개변수보다 많게 사용

        prompt('원래 매개 변수입니다.'); //선언된 매개변수보다 적게 사용
</script>
```

  - 원래 함수에서 선언된 매개변수보다 많게 사용 => 추가된 매개변수는 무시함
  - 원래 함수에서 선언된 매개변수보다 적게 사용 => 지정하지 않는 매개변수는 undefined로 입력됨
    - prompt() 함수의 두번째 매개변수를 입력하지 않으면 두번째 매개변수에 undefined 가 입력

가변인자 함수

- 가변인자 함수
  - 매개변수의 개수가 변할 수 있는 함수
  - 자바스크립트는 매개변수의 개수를 정의된 것과 다르게 사용해도 되지만, 가변인자 함수는 매개변수를 선언된 형태와 다르게 사용했을 때, 매개변수를 모두 활용하는 함수를 의미함
  - 예) Array() 함수

```
<script>
	// 배열을 생성합니다.
	var array1 = Array();
	var array2 = Array(10);
	var array3 = Array(273, 103, 57, 32);

	// 출력합니다.
	alert(array1 + '□n' + array2 + '□n' + array3);
</script>
```

Array() - 빈 배열을 만든다
Array(number) - 매개변수만큼의 크기를 가지는 배열을 만든다
Array(any, ..., any) - 매개변수를 배열로 만든다

가변인자 함수

- 예제) 매개변수로 입력된 숫자를 모두 더하는 sumAll() 함수 작성
  - 함수 내부에서 arguments 변수 찾기
  - 자바스크립트의 모든 함수는 내부에 변수 arguments 가 기본적으로 있다
  - arguments – 매개변수의 배열
  - arguments 객체의 자료형과 배열의 길이 출력하기

```
<script>
        // 함수를 생성합니다.
        function sumAll() {
            // 출력합니다.
            alert(typeof (arguments) + ' : ' + arguments.length);
        }

        // 함수를 호출합니다.
        sumAll(1, 2, 3, 4, 5, 6, 7, 8, 9);
    </script>
```

가변인자 함수

- 매개변수를 모두 더해 리턴하는 함수

```
<script>
	// 함수를 생성합니다.
	function sumAll() {
	    var output = 0;
	    for (var i = 0; i < arguments.length; i++) {
	        output += arguments[i];
	    }
	    return output;
	}

	// 함수를 호출 및 출력합니다.
	alert(sumAll(1, 2, 3, 4, 5, 6, 7, 8, 9));
</script>
```

콜백함수

- 자바스크립트에서는 함수도 하나의 자료형이므로 매개변수로 전달할 수 있음
- 콜백 함수
  - 매개변수로 전달하는 함수
  - 콜백함수는 주로 함수 내부의 처리 결과값을 함수 외부로 내보낼 때 사용함
    - 일종의 return 문과 비슷한 기능을 한다고 볼 수 있음
  - 로직구현 부분과 로직 처리부분을 나눠 코딩할 수 있게 됨
    - 로직 구현 부분은 동일하고 로직 처리 부분을 다양하게 처리해야 하는 경우 유용하게 사용할 수 있다
  - 예제) callTenTimes() – 함수를 매개변수로 받아 해당 함수를 10번 호출하는 함수

```
<script>
    // 함수 선언
    function callTenTimes(callback) {
        // 10회 반복
        for (var i = 0; i < 10; i++) {
            // 매개 변수로 전달된 함수를 호출
            callback();
        }
    }

    // 변수를 선언
    var callback = function () {
        alert('함수 호출');
    };

    // 함수를 호출
    callTenTimes(callback);
</script>
```

• 익명 콜백 함수
- 매개변수에 익명 함수를 곧바로 입력할 수 도 있음

```
<script>
    // 함수를 선언합니다.
    function callTenTimes(callback) {
        for (var i = 0; i < 10; i++) {
            callback();
        }
    }

    // 함수를 호출합니다.
    callTenTimes(function () {
        alert('함수 호출');
    });
</script>
```

setTimeout(function, millisecond)
-일정 시간 후 함수를 한 번 실행함

```
<script>
    // 3초 후에 함수를 실행합니다.
    setTimeout(function () {
        alert('3초가 지났습니다.');
    }, 3000);
</script>
```

```
<script>        // 콜백함수 사용
    function calculator(op, num1, num2, callback){
        var result="";
        switch(op) {
            case "+" :
                result = num1 + num2;
                break;
            case "-" :
                result = num1 - num2;
                break;
            case "*" :
                result = num1 * num2;
                break;
            case "/" :
                result = num1 / num2;
                break;
            default :
                result = "지원하지 않는 연산자입니다";
        }
        callback(result);
    }

    function print1(result){
        document.write("두 수의 합은 = "+ result+"입니다.", "<br>");
    }
    function print2(result){
        document.write("정답은 ="+ result+"입니다.<br>");
    }
    calculator("+", 10,20, print1);
    calculator("+", 10,20, print2);
</script>
```

```
두 수의 합은 = 30입니다.
정답은 =30입니다.
```

콜백함수

- 이벤트 리스너로 사용

```
$("#btnStart").click(function () {
        alert("클릭되었습니다.");
});
```

- 타이머 실행함수로 사용

```
setInterval(function () {
        alert("1 초마다 실행됨");
}, 1000);
```

함수를 리턴하는 함수

```
<script>
	// 함수를 생성합니다.
	function testFunction() {
	    return function () {
	        alert('Hello Function .. !');
	    };
	}

	// 함수를 호출합니다.
	//testFunction()(); //함수를 호출하면 함수가 리턴되므로 괄호를 한 번 더 사용해 해당 함수를 호출

	var result = testFunction();
	result();
</script>
```

- 함수를 리턴하는 함수를 사용하는 가장 큰 이유는 클로저 때문

클로저 함수

- 클로저란?
  - 함수 내부에 만든 지역변수가 사라지지 않고 계속해서 값을 유지하고 있는 상태를 말함

```
function 외부함수(){
        var 변수A;
        function 내부함수(){
                변수A 사용;
        }
}
```

  - 클로저는 일종의 현상이기 때문에 정해진 문법은 없으나 내부함수에서 내부함수를 포함하고 있는 함수(외부함수)의 변수A를 사용하는 구조인 경우로 표현할 수 있다
  - 내부함수를 클로저 함수라고 부름
  - 또한 변수A는 클로저 현상에 의해 외부함수() 호출이 끝나더라도 사라지지 않고 값을 유지하게 됨

예제

일반적인 경우 함수 내부에 위치하고 있는 지역변수는 함수가 종료됨과 동시에 메모리에서 사라짐

```
<script>
    // 다음 예제를 실행하면 1,2,3은 어떤 값이 출력될까?
    // 예제01: 일반 함수인 경우
    function addCount(){
        var count=0;
        count++;
        return count;
    }

    document.write("1. count = "+addCount(),"<br>");
    document.write("2. count = "+addCount(),"<br>");
    document.write("3. count = "+addCount(),"<br>");

    /*
     실행결과 :
     1.  count = 1
     2.  count = 1
     3.  count = 1
     */

</script>
```

클로저를 사용한 경우

```
<script>
    // 예제02: 클로저를 사용한 경우
     function createCounter(){
         var count=0;
         function addCount(){
             count++;
             return count;
         }
         return addCount;
     }

     var counter = createCounter();  (1)

     document.write("1. count = " + counter(),"<br>");  (2)
     document.write("2. count = " + counter(),"<br>");  (3)
     document.write("3. count = " + counter(),"<br>");  (4)

     /*
      실행결과 :
      1.    count = 1
      2.    count = 2
      3.    count = 3
      */

  </script>
```

지역변수가 사라지지 않고 계속해서 값을 유지하는 경우가 있다

(1) createCounter() 함수가 호출되면 지역변수 count가 0으로 초기화됨과 동시에 만들어짐. 그리고 내부에 addCount() 함수도 만들어지고, 마지막으로 addCount() 함수를 리턴하고 createCounter()함수는 종료됨
(2) 에서 counter()가 실행되면 addCount()함수가 실행되어 증가 연산자에 의해서 count값이 0에서 1로 증가하기 때문에 1이 출력됨
(3) ,(4) 둘 모두 counter()가 실행되면 count값이 증가하기 때문에 2와 3이 출력됨

변수가 메모리에서 제거되지 않고 계속해서 값을 유지하는 상태를 클로저라고 부르며 내부에 있는 함수를 클로저함수라고 함

createCounter()가 종료되면 일반함수처럼 addCount()함수와 count 는 사라질까?
-createCounter()가 종료되더라도 사라지지 않고 계속해서 값을 유지하고 있게 됨
-이유는 addCount() 함수 내부에 count변수를 사용하고 있는 상태에서 외부로 리턴되어 클로저 현상이 발생하기 때문

클로저가 적용된 또 다른 경우

```
<script>
      // 예제 03: 버튼을 클릭하면 클릭할 때마다 1씩 증가시켜 주세요.
      window.onload=function() {
         document.getElementById("btnStart").onclick = function () {
            start();
            document.write("시작합니다.");
         };

         function start() {
            var count = 0;
            setInterval(function () {
               count++;
               document.write(count);
            }, 1000);
         }
      };
   </script>
</head>

<body>
   <button id="btnStart" >시작</button>
</body>
```

시작합니다.1234567891011

버튼을 클릭하면 start()가 실행되면서 지역변수인 count변수가 만들어지고 setInterval()이 실행된 후 함수가 종료되며 지역변수도 같이 사라져야 하는데, setInterval()의 익명함수에서 count를 사용하고 있기 때문에 값이 계속해서 증가함
이때 이 익명함수를 클로저 함수라고 부름

이처럼 클로저는 변수가 사라지지 않고 계속해서 값을 유지하는 상태를 모두 클로저라고 부름

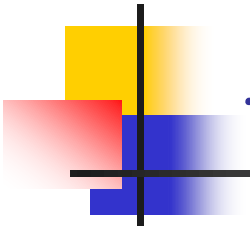
클로저

```
<script>
      // 함수를 선언합니다.
      function test(name) {
          var output = 'Hello ' + name + ' .. !';
      }

      // 출력합니다.
      alert(output);
  </script>
```

- 함수 안에 있는 변수는 지역변수이므로 함수 외부에서 사용 불가
- 클로저를 이용하면 이 규칙을 위반할 수 있음

```
<script>
      // 함수를 선언합니다.
      function test(name) {
          var output = 'Hello ' + name + ' .. !';
          return function () {
              alert(output);
          };
      }

      // 출력합니다.
      test('JavaScript')();  //output은 함수 test()를 호출할 때 생성됨
  </script>
```

변수 output은 지역변수이므로 함수가 종료될 때 사라져야 함
그러나 해당 변수가 이후에도 활용될 가능성이 있으므로 자바스크립트는 변수를 제거하지 않고 남겨둠

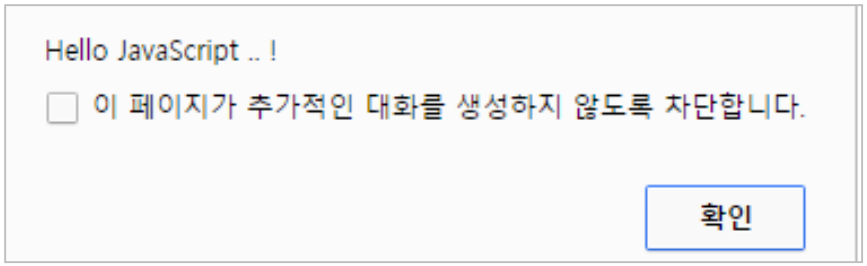


클로저

- 클로저의 정의는 다양함
  - 지역변수를 남겨두는 현상을 클로저라고 부르기도 하고
  - 함수 test() 내부의 변수들이 살아있는 것이므로 test() 함수로 생성된 공간을 클로저라고 부르기도 함
  - 리턴된 함수 자체를 클로저라고 부르기도 하며
  - 살아남은 지역변수 output을 클로저라고 부르기도 함
- 지역변수 output을 남겨둔다고 해서 외부에서 마음대로 사용할 수 있는 것은 아님
  - 반드시 리턴된 클로저 함수를 사용해야 지역변수 output을 사용할 수 있음
  - 클로저 함수로 인해 남은 지역변수는 각각의 클로저 함수의 고유한 변수임

```
<script>
    // 함수를 선언합니다.
    function test(name) {
        var output = 'Hello ' + name + ' .. !';
        return function () {
            alert(output);
        };
    }

    // 변수를 선언합니다.
    var test_1 = test('Web');
    var test_2 = test('JavaScript');

    // 함수를 호출합니다.
    test_1();
    test_2();
</script>
```

함수 test_1(), test_2()를 호출하면 각 함수가 고유한 지역변수 output 이 있다는 것을 알 수 있음



클로져

- 클로져(Closure)
  - 클로져는 함수의 실행이 끝난 뒤에도 함수에 선언된 변수의 값을 접근할 수 있는 자바스크립트의 성질
  - 자바스크립트를 다른 언어와 비교했을 때 차별화되는 유일한 특징

```
function addCounter() {
  var counter = 0;

  return function() {
    return counter++;
  };
}
```

  - 위 코드는 addCounter()라는 함수를 하나 생성하고 counter 변수를 하나 선언한 코드
  - counter라는 변수는 현재 함수 안에 선언되어 있기 때문에 함수 안에서만 유효한 유효 범위를 갖게 됨

클로져

```
function addCounter() {
  var counter = 0;
}

addCounter();
console.log(counter); // Uncaught
ReferenceError: counter is not defined
```

- 함수 밖에서 counter 변수를 참조하려고 하면 오류가 발생
- 그 이유는 함수 밖에서 counter라는 변수가 선언된 적이 없기 때문
- 다시 클로져 코드로 돌아가보자

```
function addCounter() {
  var counter = 0;

  return function() {
    return counter++;
  };
}
```

클로져

- counter 변수 다음으로 주목할 부분은 함수를 반환하는 부분
- 함수를 반환할 수 있는 이유는 '함수를 변수나 인자로 넘길 수 있는 자바스크립트의 성질(일급 객체)' 때문
- addCounter() 함수를 실행
  - addCounter();
  - console.log(addCounter());
- 출력된 결과

```
f (){
    return counter++;
 }
```

- addCounter() 함수의 역할은 addCounter() 함수를 실행했을 때 함수를 반환하는 것
- 반환된 함수를 살펴보면 counter++라는 코드가 보임
- 그 변수를 아래와 같이 접근하면 오류

클로져

```
function addCounter() {
  var counter = 0;

  return function() {
    return counter++;
  };
}

addCounter();
console.log(counter); // Uncaught ReferenceError: counter is not defined
```

- addCounter()함수의 실행이 끝난 시점에서는 counter라는 변수는 더 이상 접근할 수 없는 상태가 됨
- 함수 안에 선언한 변수는 함수 안에서만 유효 범위를 갖기 때문

클로져

```
function addCounter() {
  var counter = 0;

  return function() {
    return counter++;
  };
}

var add = addCounter();
add(); // 0
add(); // 1
add(); // 2
```

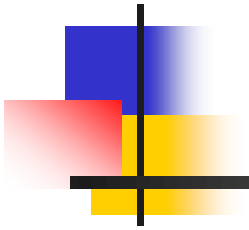
- 위와 같이 코드를 실행했을 때 동작하는 이유는?
- addCounter()라는 함수가 반환한 함수를 add라는 변수에 담아놨기 때문에 add 변수 자체가 함수처럼 동작하는 것
- "add 변수가 addCounter()가 반환한 함수를 참조하고 있다"
- 함수의 실행이 끝나고 나서도 함수 안의 변수를 참조할 수 있는게 클로져

클로져

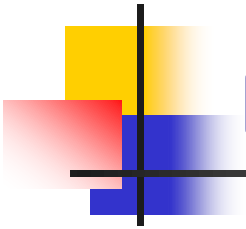
- 이러한 패턴을 응용하면 자바스크립트에 없는 private 변수나 함수형 프로그래밍을 할 수 있다
- 함수형 프로그래밍
  - 함수형 프로그래밍이란 특정 기능을 구현하기 위해서 함수의 내부 로직은 변경하지 않은 상태로 여러 개의 함수를 조합하여 결과 값을 도출하는 프로그래밍 패턴
  - 커링(currying)이 함수형 프로그래밍의 대표적인 예

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

function curry(fn, a) {
  return function(b) {
    return fn(a, b);
  };
}

var add3 = curry(add, 3);
add3(4); // 7
```

  - 위와 같이 클로져를 활용하면 함수를 조합하여 기능을 구현해나갈 수 있다

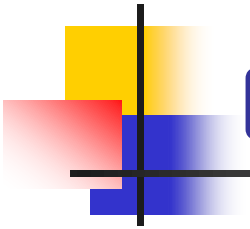


ES6



ES6

- ECMAScript란?
  - ECMAScript (또는 ES) : 자바스크립트 표준 문법
- ES6 : ECMAScript 표준의 6번째 에디션, ECMAScript2015
- 넷스케이프에서 1995년 개발한 자바스크립트는 웹 브라우저에서 동적인 기능을 제공하기 위한 언어이다.
  - 현재는 대부분의 브라우저에서 이 언어를 제공하고 있다. 그런데 표준 규격없이 여러 브라우저에서 독자적인 특성이 추가되면서 호환성 문제가 발생하기 시작했다.
  - 이에 ECMA 국제 기구에서 "ECMAScript Standard"라는 표준을 만들었다.
  - 현재의 자바스크립트는 ECMAScript와 BOM(Browser Object Model)와 DOM(Document Object Model)을 포괄하는 개념이다.



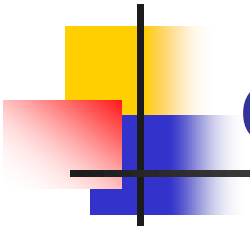
ES6

- ECMAScript가 필요한 이유
  - 라이브러리들이 최신 언어 명세들을 이용해서 작성된다.
  - 표준을 따라간다.
  - 편안하다.

const, let

- [1] const, let
  - ES6 이전에 변수를 선언하는 방법은 var
  - 기존의 var 은 function-scope 기반이어서 호이스팅 문제가 있었다. 그래서 ES6에는 let 과 const가 생겼다.
- let, const는 기본적으로 var과 다르게 재선언이 불가능.
- const는 let과 다르게 재할당도 불가능

```
/*다시 선언할 수 있는 var 변수*/
var num1 = 1;
var num1 = 2;

/* 다시 선언할 수 없는 let 변수 */
let num2 = 1;
let num2 = 2; // Uncaught SyntaxError: Identifier 'a' has already been declared
num2 = 3;

/* 재선언, 재할당 모두 불가능한 const 변수 */
const num3 = 1;  //상수
const num3 = 2; // Uncaught SyntaxError: Identifier 'a' has already been declared
num3 = 3 // Uncaught TypeError:Assignment to constant variable.
```



const & let

- const와 let 예약어는 ES6에서 사용하는 변수 선언 방식
- let
  - let 예약어는 한번 선언하면 다시 선언할 수 없다.

```
// 똑같은 변수를 재선언할 때 오류
let a = 10;
let a = 20; // Uncaught SyntaxError: Identifier 'a' has already been declared
```

- const
  - const 예약어는 한번 할당한 값을 변경할 수 없다.

```
// 값을 다시 할당했을 때 오류
const a = 10;
a = 20; // Uncaught TypeError: Assignment to constant variable.
```

  - 단, 객체 {}또는 배열 []로 선언했을 때는 객체의 속성(property)과 배열의 요소(element)를 변경할 수 있다.

const & let

```
// 객체로 선언하고 속성 값을 변경
const b = {
    num: 10,
    text: 'hi'
};
console.log(b.num); // 10

b.num = 20;
console.log(b.num); // 20

// 배열로 선언하고 배열 요소를 추가
const c = [];
console.log(c); // []

c.push(10);
console.log(c); // [10]
```

- 블록 유효범위
  - ES5의 var를 이용한 변수 선언 방식과 let & const를 이용한 변수 선언 방식의 가장 큰 차이점은 블록 유효범위임

const & let

- var의 유효 범위
  - var의 유효 범위는 함수의 블록 단위로 제한됨
  - 함수 스코프(function scope)라고 표현

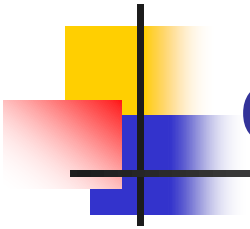
```
var a = 100;
function print() {
    var a = 10;
    console.log(a); //10
}
print(); // 10
```

  - print 함수 앞에 선언한 a와 print 함수 안에 선언한 a는 각자 다른 유효 범위를 갖는다
  - var a = 100; 는 자바스크립트 전역 객체인 window에 추가가 되고 var a = 10;는 print() 함수 안에서만 유효한 범위를 갖는다.
- for 반복문에서의 var 유효 범위
  - 헷갈릴 수 있는 부분이 "var의 유효 범위가 {}에 제한되나?" 임

const & let

```
var a = 10;
for (var a = 0; a < 5; a++) {
    console.log(a); // 0 1 2 3 4
}
console.log(a); // 5
```

- var a = 10; 로 변수 a를 선언한 상태에서 for 반복문에 동일한 변수 이름 a를 사용
- 이렇게 되면 **{} 으로 변수의 유효 범위가 제한되지 않기 때문에** for 반복문이 끝나고 나서 console.log(a); 를 출력하면 for 반복문의 마지막 결과 값이 출력됨
- 자바나 다른 언어의 개발자들에게는 이 부분이 가장 헷갈릴 것
- 이러한 문제점을 해결하고 다른 언어와 통일감을 주기 위해 ES6에서는 **const와 let의 변수 유효 범위를 블록{}으로 제한**하였다.



const & let

- const와 let의 블록 유효범위
  - 반복문 코드에 var 대신 let을 적용해보자

```
var a = 10;
for (let a = 0; a < 5; a++) {
    console.log(a); // 0 1 2 3 4
}
console.log(a); // 10
```

  - 반복문의 조건 변수 a를 let으로 선언하면 변수의 유효 범위가 for 반복문의 {} 블록 안으로 제한됨

화살표 함수 (arrow function)

- [2] 화살표 함수 (arrow function)
  - ES6에 추가된 문법

```
/* 기존 함수 선언식 */
function func(){
   const data = 'hello wolrd';
   return data;
}

/* 함수 표현식 */
const func = function(){
   const data = 'hello wolrd';
   return data;
}
```

  - ES6에서는 => 를 이용해서 좀더 간결하게 작성 가능

화살표 함수 (arrow function)

```
const func1 = () =>{
   const data = 'hello wolrd';
   return data;
}

/* 매개변수가 1개 라면 괄호 생략 가능*/
const func2 = num =>{
   return num;
}

/*바디가 1개라면 중괄호와 return 생략 가능 */
const func3 = num => num + 1;
```

```
/* 함수 표현식 */
const func = function(){
   const data = 'hello wolrd';
   return data;
}
```

화살표 함수(Arrow Function)

- 화살표 함수(Arrow Function)는 ES6의 새로운 함수 정의 방식
- 기존의 함수 정의 방식
  - 기존 자바스크립트의 함수 정의 방식

```
var a = function() {
   // ...
};
```

- 화살표 함수를 이용한 함수 정의
  - 화살표 함수를 이용한 함수 정의 방식

```
var a = () => {
   // ...
};
```

- 화살표 함수의 다양한 문법
  - 화살표 함수를 정의하는 방식은 간단한 자바스크립트 표현식부터 {} 를 이용한 선언 방식까지 여러 방법이 있다.

화살표 함수(Arrow Function)

- 1. 단순한 자바스크립트 표현식
  - 단순한 자바스크립트 표현식의 경우 아래와 같이 간소화 문법을 사용할 수 있다.

```
() => 10 + 20; // {} 필요 없음
```

- 2. 함수 선언 방식
  - 복잡한 자바스크립트 선언문이 들어갈 경우에는 {}를 사용하여 아래와 같이 구현합니다.

```
() => {
    print();
    log();
    return 10 + 20;
};
```

화살표 함수(Arrow Function)

- 3. 전달 인자(parameter)가 하나인 경우
  - 인자를 1개만 선언하는 경우 인자를 받을 때 사용하는 소괄호() 를 생략할 수 있습니다.

```
const a = (num) => { return num * 10 };
const b = num => num * 10;
a(10); // 100
b(10); // 100
```

배열관련 메소드

- [3] 배열관련 메소드
- <1> map
  - 리액트를 하면서 가장 많이 사용하는 메소드 중 하나
  - 배열에 있는 값을 일괄적으로 변경할 때 사용
  - 3개의 인자값을 가지는데,
  - 첫번째가 currenValue, 배열의 요소값을 의미하고, index는 그 요소의 위치, array는 배열 자체를 의미

```
const list = [1, 2, 3, 4, 5].map((currenValue, index, array) => {
    return currenValue + 1;
});
```

결과
[2, 3, 4, 5, 6]

  - 중괄호, return 생략한 코드

```
const list = [1, 2, 3, 4, 5].map((currenValue, index, array) =>
currenValue + 1 );
```

배열관련 메소드

- <2> filter
  - 기존 ES5에서 배열에서 값을 빼려면 indexOf 와 split 등 복잡한 과정을 거쳤지만, filter를 사용한다면 쉽게 구현할 수 있다.
  - 조건이 참이라면 요소를 반환하고 거짓이라면 반환하지 않는다.

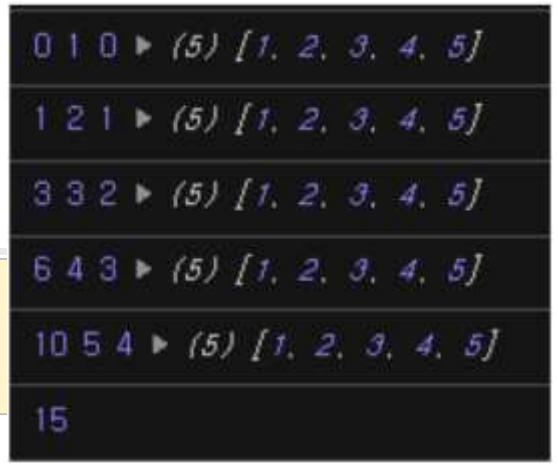
```
const list = [1, 2, 3, 4, 5].filter((currenValue, index, array) =>
currenValue % 2 === 0 )

// list : [2, 4]
```

- <3> reduce
  - 값을 누적시킬때 유용
  - reduce를 잘 이용한다면 위에 map, filter를 모두 대체할 수 있다.
  - 함수 다음에 들어가는 **초기값**이 중요한데 처음 루프가 돌때 아래 코드에 들어간 0 이 previousValue로 들어가게된다.

```
const total = [1, 2, 3, 4, 5].reduce((previousValue, currentValue,
currentIndex, array) => {
   return previousValue += currentValue
}, 0);

// total : 15
```

배열관련 메소드

```
reduce((previousValue, currentValue, currentIndex, array) => {
 return previousValue + currentValue
}, initialValue);
```



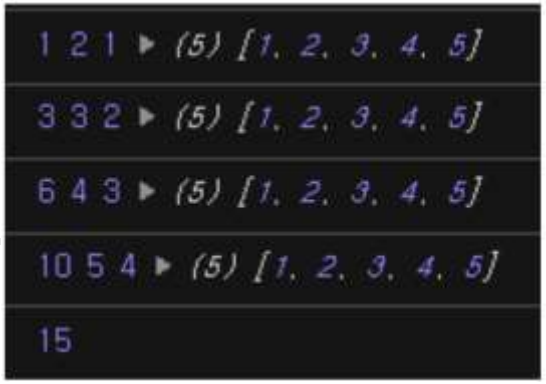
- previousValue 는 정확하게는 **누적값**

```
const numbers = [1, 2, 3, 4, 5]
const summary = numbers.reduce((accumulation, currentValue, currentIndex, array) => {
        console.log(accumulation, currentValue, currentIndex, array);
        return accumulation + currentValue;
}, 0);

console.log(summary);
```

```
reduce((previousValue, currentValue) => previousValue + currentValue,
initialValue);
```

- 1. initialValue : 초깃값. 생략하면 배열의 첫번째 숫자가 들어감
  - 여기에서는 초깃값으로 0을 넣었다
- 2. 이 초깃값이 accumulate로 누적이 되는 것
- 3. 그러면 currentValue에 numbers의 0번째부터 온다
- 4. 그럼 currentIndex가 그 0번째
- 5. array는 바로 numbers 라는 배열을 가리킴 => [1, 2, 3, 4, 5]
- 6. 그럼 이 return 값이 1
- 7. 이게 accumulation이 됨
- 8. 이 과정이 반복

배열관련 메소드



- 초깃값 생략

```
const summary = numbers.reduce((accumulation, currentValue, currentIndex, array) => {
    console.log(accumulation, currentValue, currentIndex, array);
    return accumulation + currentValue;
  });

console.log(summary);
```

- 값은 같아지지만, 진행과정이 조금 차이가 난다
- 초깃값이 0 이 아닌 배열의 첫 번째 수 부터

Rest 파라미터와 Spread 연산자

- [4] Rest 파라미터와 Spread 연산자
- <1> rest
  - 함수 파라미터에서 아규먼트를 배열로 받아 올 때 사용하는 문법.

```
const func = (...rest) => {
   console.log(rest); // [1, 2, 3]
}

func(1, 2, 3)
```

- <2> spread
  - 배열을 분해하거나 객체를 분해하여 각각의 요소로 만든다.

```
let list = [1, 2, 3];

console.log(list); // [1, 2, 3]
console.log(...list); // 1, 2, 3
```

  - 이를 이용해서 객체에 새로운 값을 좀 더 간편하게 추가할 수 있다.

```
let me = { name : 'hong', age : 21 } ;
me = {...me, married : false} // { name : 'hong', age : 21, married : false }
```

스프레드 오퍼레이터(Spread Operator)

- 스프레드 오퍼레이터 – 한글로 번역해보면 펼침 연산자
  - 스프레드 오퍼레이터는 **특정 객체 또는 배열의 값을 다른 객체, 배열로 복제하거나 옮길 때 사용**
  - 연산자의 모양은 …
- 스프레드 오퍼레이터 사용법

```
// obj 객체를 newObj 객체에 복제
var obj = {
  a: 10,
  b: 20
};
var newObj = {...obj};
console.log(newObj); // {a: 10, b: 20}

// arr 배열을 newArr 배열에 복제
var arr = [1,2,3];
var newArr = [...arr];
console.log(newArr); // [1, 2, 3]
```

  - 위 코드들은 모두 스프레드 오퍼레이터를 이용하여 특정 객체, 배열의 값을 각각 새로운 객체와 배열에 복제하는 코드
  - 스프레드 오퍼레이터를 사용하는 이유는 무엇일까?

스프레드 오퍼레이터(Spread Operator)

- 기존 자바스크립트의 객체 복제 방식
  - 스프레드 오퍼레이터를 사용하지 않고 기존 자바스크립트 문법으로만 구현

```
// 객체의 값을 복사하는 경우
var obj = {
  a: 10,
  b: 20
};
var newObj = {
  a: obj.a,
  b: obj.b
};
console.log(newObj); // {a: 10, b: 20}
```

```
// 배열의 값을 복사하는 경우
var arr = [1,2,3];
var newArr = [arr[0], arr[1], arr[2]];
console.log(newArr); // [1, 2, 3]
```

- 객체를 복사하는 경우, 새로운 객체인 newObj에 새로운 속성들을 선언하고 각 속성에 obj의 속성들을 일일이 접근해서 대입해줘야 함.
- 배열 newArr의 경우에는 기존 배열 arr의 인덱스에 일일이 접근하여 새로운 요소를 만들어줘야 함.
- 스프레드 오퍼레이터를 사용하게 되면 타이핑해야 하는 코드의 양이 확연히 줄어듬.

디스트럭처링 (destructuring)

- [5] 디스트럭처링 (destructuring)
  - 배열과 객체의 값을 쉽게 추출 할 수 있다.

```
const [a, b, c] = [1, 2, 3];
console.log(a, b, c) // 1, 2, 3

const {a, b, c} = {a : 1, b : 2, c :3};
console.log(a, b, c) // 1, 2, 3

const {c, a, b, d=5} = {a : 1, b : 2, c :3};
console.log(a, b, c, d) // 1, 2, 3, 5
```

  - 객체 디스트럭처링이 간편한 이유는 key를 기반으로 추출하기 때문에 순서가 달라도 된다는 점
  - 객체는 프로퍼티의 이름으로 분해를 하므로 순서가 달라도 일치하는 프로퍼티 명이 있다면 분할 대입이 되고, 존재 하지 않는 프로퍼티는 무시됨. 또한 분할 대입시 존재하지 않아도 디폴트값 선언으로 프로퍼티를 지정 할 수가 있다.

구조 분해 문법(Destructuring)

- 디스트럭처링이라고 하는 이 ES6 문법은 한글로 번역하면 구조 분해 문법
- '구조'라는 단어를 먼저 파악
- 기존 자바스크립트에서의 '구조'
  - 기존 자바스크립트에서 객체와 배열의 구조

```
var arr = [1, 2, 3, 4];
var obj = {
  a: 10,
  b: 20,
  c: 30
};
```

- 전형적인 객체, 배열 선언 방식
- 왼쪽에 변수 이름을 넣고 오른쪽에 데이터 타입을 선언
- '구조'라는 단어는 이러한 선언 형식을 의미

구조 분해 문법(Destructuring)

- '구조 분해'란
  - 이러한 변수 선언 형식이 아래와 같이 자유로워지는 것을 의미

```
var obj = {
  a: 10,  b: 20,  c: 30
};
```

```
var { a, b, c } = obj;
console.log(a); // 10
console.log(b); // 20
console.log(c); // 30
```

구조 분해 문법(Destructuring)

- 특정 객체의 값을 꺼내오는 방법
- 기존 자바스크립트에서 특정 객체의 값을 꺼내올 때

```
var person = {
  name: 'hong',
  address: 'seoul',
  hobby: 'sports',
  age: '20'
};

var name = person.name;
var address = person.address;
var hobby = person.hobby;
var age = person.age;
```

  - 객체의 특정 속성 값을 꺼내올 때마다 일일이 변수를 하나 생성하고 담아줘야 한다는 점
  - 꺼내야 할 속성이 많으면 많을수록 새로운 변수를 생성하고 대입하는 식의 반복 작업을 계속해줘야 함

구조 분해 문법(Destructuring)

- 구조 분해 문법을 적용하면 훨씬 더 간편하게 꺼내올 수 있다.

```
var person = {
  name: 'hong',
  address: 'seoul',
  hobby: 'sports',
  age: '20'
};

var { name, address, hobby, age } = person;
console.log(name); // hong
console.log(address); // seoul
console.log(hobby); // sports
console.log(age); // 20
```

- 구조 분해 문법을 사용하면 코드 라인 숫자를 줄일 수 있고, 전체적으로 코드가 더 간결해지는 것을 알 수 있다

향상된 객체 리터럴(Enhanced Object Literal)

- 향상된 객체 리터럴이란
  - 기존 자바스크립트에서 사용하던 객체 정의 방식을 개선한 문법
  - 자주 사용하던 문법들을 좀 더 간결하게 사용할 수 있도록 객체 정의 형식을 바꿨다.
- 기존 객체 정의 방식
  - 기존 자바스크립트의 객체 정의 방식

```
var person = {
  // 속성: 값
  address: 'seoul',
  coding: function() {
    console.log('Hello js');
  }
};
```

- 축약 문법 1 – 속성과 값이 같으면 1개만 기입
  - 객체를 정의할 때 속성(property)와 값(value)이 같으면 축약이 가능

향상된 객체 리터럴(Enhanced Object Literal)

```
var address = 'seoul';

var person = {
  //address : address,
  address
};

console.log(person); // {address: "seoul"}
```

- 위와 같은 축약 문법을 뷰 컴포넌트 등록 방식과 뷰 라우터 등록 방식에 대입해보면 다음과 같다.

```
// #1 - 컴포넌트 등록 방식에서의 축약 문법
const myComponent = {
  template: '<p>My Component</p>'
};

new Vue({
  components: {
    // myComponent: myComponent
    myComponent
  }
});
```

향상된 객체 리터럴(Enhanced Object Literal)

```
// #2 - 라우터 등록 방식에서의 축약 문법
const router = new VueRouter({
  // ...
});

new Vue({
  // router: router,
  router
});
```

향상된 객체 리터럴(Enhanced Object Literal)

- 축약 문법 2 – 속성에 함수를 정의할 때 function 예약어 생략
- 기존에 객체를 정의할 때 객체의 속성에 함수를 연결하여 사용하는 경우가 많았다.

```
const person = {
  // 속성: 함수
  coding: function() {
    console.log('Hello js');
  }
};
person.coding(); // Hello js
```

- ES6에서는 아래와 같이 축약하여 코딩하는 것을 추천

```
const person = {
  coding() {
    console.log('Hello js');
  }
};
person.coding(); // Hello js
```

- function 예약어를 생략해도 동일하게 동작

템플릿 리터럴(Template Literal)

- 템플릿 리터럴 – 자바스크립트에서 문자열을 입력하는 방식
  - 기존에는 var str = 'Hello ES6'와 같은 방식으로 사용하였으나 ES6에서는 백틱(back-tick)이라는 기호(`)를 사용하여 정의
- const str = `Hello ES6`;
  - 백틱을 이용하게 되면 **여러 줄에 걸쳐 문자열을 정의할 수도 있고, 자바스크립트의 변수를 문자열 안에 바로 연결할 수** 있다.
- 여러 줄에 걸쳐 문자열 선언하기
  - 기존 자바스크립트의 문자열 선언 방식인 작은 따옴표(')의 문제점은 아래와 같다.

```
var str = 'Template literals are string literals allowing embedded expressions. □n' +
'You can use multi-line strings and string interpolation features with them. □n' +
'They were called "template strings" in prior editions of the ES2015 specification.';
```

  - 작은 따옴표를 이용하여 긴 문자열을 선언하게 되면 자동으로 개행이 되지 않기 때문에 라인 브레이커(line breaker)인 ₩n를 개행할 곳 중간 중간에 추가해야 했다.
  - 문장이 길면 길수록 +와 ₩n를 계속 추가해줘야 함
- 백틱을 이용해서 문자열을 선언하게 되면 개행할 필요가 없다.

템플릿 리터럴(Template Literal)

```
const str = `Template literals are string literals allowing embedded expressions.
You can use multi-line strings and string interpolation features with them.
They were called "template strings" in prior editions of the ES2015 specification.`;
```

- 뷰에서는 CDN 방식으로 뷰를 적용할 때 컴포넌트의 template 속성에 적용

```
Vue.component('my-cmp', {
  template: `
    <div>
      <h1>My Component</h1>
      <p>back-tick is useful</p>
    </div>
  `
});
```

템플릿 리터럴(Template Literal)

- 문자열 중간에 변수 바로 대입하기
  - 기존 문자열 정의 방식에서 번거로웠던 부분은 자바스크립트 변수 값을 문자열과 함께 사용하는 부분.

```
var language = 'Javascript';
var expression = 'I love ' + language + '!';
console.log(expression); // I love Javascript!
```

- 문자열에 특정 변수의 값을 함께 사용하려면 +를 이용하여 문자열 중간에 해당 변수를 연결해줘야 했다.
- ES6에서는 **템플릿 리터럴을 이용하면 간편하게 문자열과 변수를 함께 사용할** 수 있다.

```
var language = 'Javascript';
var expression = `I love ${language}!`;
console.log(expression); // I love Javascript!
//${}를 이용하면 위와 같이 변수의 값을 대입할 수 있을 뿐만 아니라 간단한 연산도 할 수 있다.

var language = 'Javascript';
var expression = `I love ${language.split('').reverse().join('')}!`;
console.log(expression); // I love tpircsavaJ!
```

  - language의 문자열 순서를 역으로 바꾸는 코드

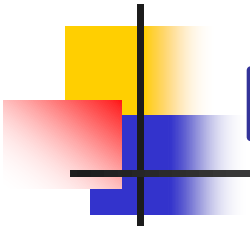
Import & Export

- 임포트(Import)와 익스포트(Export)
  - 자바스크립트의 코드를 모듈화 할 수 있는 기능
  - 모듈화란 다른 파일에 있는 자바스크립트의 기능을 특정 파일에서 사용할 수 있는 것
- 모듈화의 필요성
  - 기본적으로 자바스크립트의 유효 범위는 전역으로 시작
  - 아래와 같이 HTML 페이지를 로딩하면 원하는 결과가 나오지 않는다

```
<!-- index.html -->
<body>
  <script src="a.js"></script>
  <script src="b.js"></script>
  <script>
    getTotal(); // 200
  </script>
</body>
```

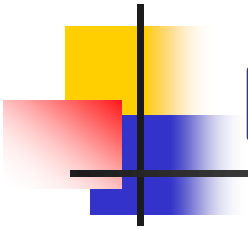
```
// a.js
var total = 100;
function getTotal() {
  return total;
}
```

```
// b.js
var total = 200;
```



Import & Export

- 다른 프로그래밍 언어에서는 파일 마다 변수의 유효 범위가 다른 경우가 많지만, 자바스크립트는 기본적으로 변수의 유효 범위가 전역으로 잡히기 때문에 네임스페이스 모듈화 패턴이나 Require.js와 같은 모듈화 라이브러리를 이용하여 모듈화 기능을 구현해왔다.
- 이제는 프로그래밍 패턴이나 별도의 모듈화 라이브러리를 사용하지 않고도 자바스크립트 언어 자체에서 모듈화를 지원함.



Import & Export

- import & export 기본 문법
  - 모듈화 기능을 사용하기 위한 기본적인 import, export 문법
- export 문법
  - export 변수, 함수
    - 다른 파일에서 가져다 쓸 변수나 함수의 앞에 export 라는 키워드를 붙임
    - 익스포트된 파일은 임포트로 불러와 사용할 수 있다
- import 문법
  - import { 불러올 변수 또는 함수 이름 } from '파일 경로';
    - 익스포트된 변수나 함수를 {}에 선언한 뒤 해당 파일이 있는 경로를 적어줌

Import & Export

- import & export 기본 예제

```
// math.js
export var pi = 3.14;

// app.js
import { pi } from './math.js';

console.log(pi); // 3.14
```

- math.js라는 파일에서 pi를 익스포트하고 app.js 파일에서 임포트하여 콘솔에 출력하는 코드
- 변수가 아니라 함수를 내보내고 싶다면

Import & Export

```
// math.js
export var pi = 3.14;
export function sum(a, b) {
  return a + b;
}
// app.js
import { sum } from './math.js';

sum(10, 20); // 30
```

- math.js에 두 숫자의 합을 구하는 sum() 함수를 익스포트 한 뒤 app.js에서 임포트하여 사용한 코드

- 브라우저 지원 범위
  - ES6의 기본적인 문법들이 최신 브라우저에서 지원되는데 반해 import, export는 아직 보조 도구가 있어야만 사용할 수 있다.
- 가급적 실무 코드에서 사용할 때는 웹팩(Webpack)과 같은 모듈 번들러를 이용하여 구현하는 것이 좋다

[1] Import
< Import 문법 >
import name from "module-name"; // export default로 export한 멤버를 name에 받음.
import * as name from "module-name"; // export되는 모든 멤버를 name에 받음.
import { member } from "module-name"; // export된 멤버 member를 member로 받음.
import { member as alias } from "module-name"; // export된 멤버 member를 alias로 받음.
import { member1 , member2 } from "module-name";
import { member1 , member2 as alias2 , [...] } from "module-name";
import defaultMember, { member [, [...]] } from "module-name";
import defaultMember, * as alias from "module-name";
import defaultMember from "module-name";
import "module-name"; // import만 하면 되는 경우 ex) import "main.css";

[2] Export

< Export 특징 >
default export는 스크립트당 하나만 존재 가능
export default는 var,let,const 사용 불가하다.
default export의 경우, 모듈 당 딱 하나의 default export가 있으며
default export는 함수 또는 클래스, 오브젝트, 혹은 다른 것들이 될 수 있다.
이값은 가장 간단하게 import 할 수 있도록 하기 때문에 내보낼 값 중 메인에 해당하는 값으로 고려해야한다.

< Export 문법 >

```
export { name1, name2, …, nameN };
export { variable1 as name1, variable2 as name2, …, nameN };
export let name1, name2, …, nameN; // 또는 var
export let name1 = …, name2 = …, …, nameN; // 또는 var, const

export default expression;
export default function (…) { … } // 또는 class, function*
export default function name1(…) { … } // 또는 class, function*
export { name1 as default, … };

export * from …;
export { name1, name2, …, nameN } from …;
export { import1 as name1, import2 as name2, …, nameN } from …;
```

Async & Await

- 어싱크 어웨이트
  - 자바스크립트 비동기 처리 패턴의 최신 문법
  - Promise와 Callback에서 주는 단점들을 해결하고 자바스크립트의 비동기적 사고 방식에서 벗어나 동기적(절차적)으로 코드를 작성할 수 있게 도와줌
- 기본 문법
  - async 함수의 기본 문법

```
async function fetchData() {
  await getUserList();
}
```

- async 함수는 함수의 앞에 async를 붙여주고 함수의 내부 로직 중 비동기 처리 로직 앞에 await를 붙여주면 됨
- Promise 객체를 반환하는 API 호출 함수 앞에 await를 붙인다

Async & Await

- 기본 예제

```
async function fetchData() {
  var list = await getUserList();
  console.log(list);
}

function getUserList() {
  return new Promise(function(resolve, reject) {
    var userList = ['user1', 'user2', 'user3'];
    resolve(userList);
  });
}
```

```
const user=async () => {
  try {
    const response = await
axios.get('/user?ID=kim')
    console.log(response)
  } catch (error) {
    console.error(error)
  }
}
```

- fetchData() 함수에서 getUserList() 함수를 호출하고 나면 Promise 객체가 반환됨
- 그리고 그 Promise는 실행이 완료된 상태(resolve)이며 실행의 결과로 userList 배열을 반환
- 따라서 fetchData()를 호출하면 userList의 배열이 출력됨
  - fetchData(); // ['user1', 'user2', 'user3']

기본 매개 변수

- 기본 매개 변수 (Default Parameters)

```
var link = function (height, color, url) {
    var height = height || 50
    var color = color || 'red'
    var url = url || 'http://abc.com'
    ...
}
```

- 함수에 넘겨주는 인자값에 대한 default 처리를 위해 위와 같이 처리 했었다면
- ES6에서는 아래와 같이 간단히 처리할 수 있다.

```
var link = function(height = 50, color = 'red', url = 'http://abc.com') {
  ...
}
```

- 주의해야 할 점이 있다. 인자값으로 0 또는 false가 입력될 때 두 예시의 결과는 다르다.
  - ES5에서는 || 처리 시 0 또는 false 값이 입력 되어도 거짓이 되므로 기본값으로 대체된다.
  - 하지만 ES6의 기본 매개 변수를 사용하면 undefined 를 제외한 입력된 모든 값(0, false, null 등)을 인정한다.

Rest Parameter

Rest Parameter - 정해지지 않은 수의 인자를 배열로 나타낼 수 있게 한다.
Rest Parameter 사용방법 - parameter앞에 ...을 붙인다.

```
function foo(...rest) {
  console.log(Array.isArray(rest)); // true
  console.log(rest); // [ 1, 2, 3, 4, 5 ]
}
foo(1, 2, 3, 4, 5);
```

function foo(param1, param2, ...rest){~~} 처럼 앞에 parameter는 일반적인 parameter로 받을 수 있고 그 뒤부터는 Rest parameter로 받을 수 있다.

단, Rest parameter 항상 제일 마지막 parameter로 있어야 한다.
(예) function foo(...rest, param1, param2){~}는 사용 불가능하다.

Rest Parameter와 argument 차이점
[1] argument : 유사 배열
(유사 배열 객체(array-like object)는 간단하게 순회가능한(iterable) 특징이 있고 length 값을 알 수 있는 특징이 있는 것이다.
즉, 배열처럼 사용할 수 있는 객체를 말한다. 즉, arguments는 유사배열객체이기 때문에 Array 오브젝트의 메서드를 사용할 수 없다.)

Rest Parameter

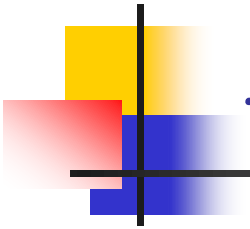
[2] rest parameter : 배열

< Rest Parameter 예시 1 >

```
function foo (a, b, ...c) {
  console.log(c);                    // ["c", "d", "e", "f"]
  console.log(Array.isArray(c));     // true
}

foo('a', 'b', 'c', 'd', 'e', 'f');
```

< Rest Parameter 예시 2 >

```
// "arguments" is different
function foo2 (a, b, ...c) {
  console.log(arguments);
  console.log(Array.isArray(arguments)); // false
}

foo2(1, 2, 3, 4, 5);
```



클래스

- ES6의 클래스 문법
  - 1) ES 6 이전

```
function Dog(name) {
  this.name = name;
}

Dog.prototype.say = function() {
  console.log(this.name + ': 멍멍');
}
var dog = new Dog('검둥이');
dog.say(); // 검둥이: 멍멍
```

클래스

- 2) ES6

```
class Dog {
  constructor(name) {
     this.name = name;
  }
  say() {
     console.log(this.name + ': 멍멍');
  }
}

const dog = new Dog('흰둥이');
dog.say(); // 흰둥이: 멍멍
```

```
//기존 방식 - 클래스 문법
function Dog(name){
	this.name=name;
}

Dog.prototype.say=function(){
	console.log(this.name +" : " + "멍멍!");
}

var dog = new Dog('검둥이');
dog.say();  //검둥이 : 멍멍!

//ES6 방식
class Dog2{
	constructor(name){
		this.name=name;
	}

	say(){
		console.log(this.name +" => " + "멍멍멍");

	}
}

const dog2 = new Dog2('흰둥이');
dog2.say();  //흰둥이 => 멍멍멍
```

상속

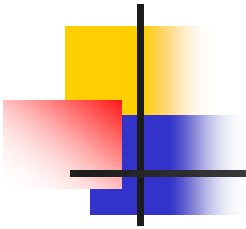
```
class Car {
 constructor(modelName, modelYear, type, price) {
   this.modelName = modelName;
   this.modelYear = modelYear;
   this.type = type;
   this.price = price;
 }

 getModelName() {
   return this.modelName;
 }

 getModelYear() {
   return this.modelYear;
 }
```

다음 코드에서 기본 자동차 클래스에 충전시간(chargeTime)만 추가한 새로운 자동차 타입인 전기 자동차(ElectricCar)를 정의했습니다. 이때, extends를 사용해서 Car 클래스의 특성을 모두 상속받을 수 있습니다.

```
class ElectricCar extends Car {
 constructor(modelName, modelYear, price, chargeTime) {
   super(modelName, modelYear, "e", price);
   this.chargeTime = chargeTime;
 }

 setChargeTime(time) {
   this.chargeTime = time;
 }

 getChargeTime() {
   return this.chargeTime;
 }
}
```

비교연산자

- 비교 연산자
  - a == b // a와 b는 동등하다.
  - a === b // a와 b는 일치한다.
  - a !== b // a와 b는 일치하지 않는다.
  - a != b // a와 b는 동등하지 않다.
  - a < b // a보다 b보다 작다.
  - a <= b // a는 b보다 작거나 같다.
- 애매한 비교를 하게 되는 동등 연산자 (== , !=)
  - 애매한 결과를 초래할 수 있기에 사용을 추천하지 않는다.
- 동등과 일치는 거의 비슷하게 동작하지만, 일치하지 않는 결과를 같다라고 표현하는 것은 모순일 수 있다.
- 숫자 3과 문자열 3을 같다고 인식하는데 타입이 다름으로 3==="3" 은 false를 출력하고, 3 == "3"은 true를 출력한다는 점을 유념해야 한다.



비교연산자

- 조건 연산자
  - const a = (b>0) ? "양수" : "음수";
    - (b>0)의 값이 true라면 두 번째 피연산자인 "양수"가 출력되고, false라면 세 번째 피연산자인 "음수"가 출력된다.