ECMAScript 6

inky4832@daum.net

1장. ECMAScript6 개요

ES6 개요

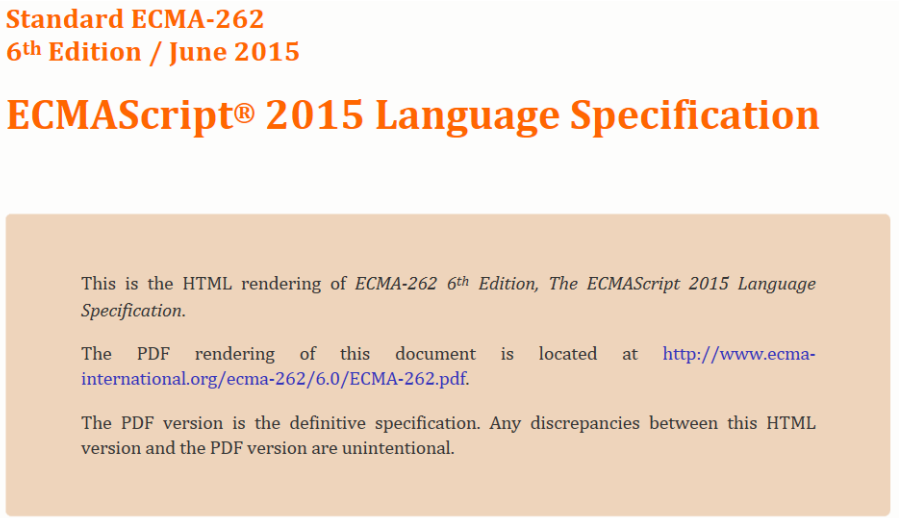
1) ECMAScript 란?

ECMAScript (ES)는 ECMAScript International에서 표준화 한 스크립팅 언어 사양.
응용 프로그램에서 클라이언트측 스크립팅을 하기 위해 사용됨.
JavaScript. Jscript, ActionScript 등은 모두 ECMAScript 스펙 적용을 받는다.
(http://www.ecma-international.org/ecma-262/7.0/index.html)

- ECMAScript2015(ECMAScript 6)
- ECMAScript2016(ECMAScript 7)

2) ECMAScript 6 버전에서 사용 가능한 기능

Support for constants (상수지원)
Block Scope (블록 범위)
Arrow Functions (화살표 기능)
Extended Parameter Handling (확장 매개변수 처리)
Template Literals (템플릿 리터럴)
Extended Literals (확장된 리터럴)
Enhanced Object Properties (향상된 개체 속성)
De-structuring Assignment (비 구조화 과제)
Modules (모듈)
Classes (classes) *객체지향 개념 대폭 지원
Iterators (이터레이터)
Generators (제너레이터)
Collections (컬렉션)
New built in methods for various classes (새로운 메서드)
Promises (프라미스)


Standard ECMA-262
6th Edition / June 2015

ECMAScript® 2015 Language Specification

This is the HTML rendering of *ECMA-262 6th Edition, The ECMAScript 2015 Language Specification.*

The PDF rendering of this document is located at http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf.

The PDF version is the definitive specification. Any discrepancies between this HTML version and the PDF version are unintentional.

2. let, const 및 블록 스코프

1) var 키워드

- 변수 선언시 사용
- 변수 종류 2가지 (전역 변수, 로컬 변수)

전역변수와 로컬변수로 구분하는 이유는 스코프(scope)가 달라지기 때문.
일반적으로 javascript는 블록 스코프가 아닌 **함수 스코프(function scope)**를 따른다.

ES6는 변수를 설정하기 위한 새로운 2가지 키워드를 제공 (let 과 const)

2) let 키워드 (*)

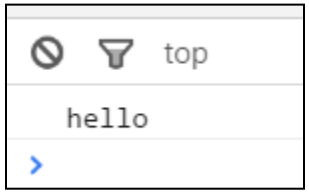
- var 키워드의 문제점 해결 목적으로 등장.
- let 키워드를 사용하면 함수 스코프가 아닌 **블록 스코프를 따른다.**

따라서 블록밖에 동일한 이름의 변수가 있어도 scope가 다르기 때문에 변수에 각각 값을 설정할 수 있다.

- **let 변수는 호이스팅(hoisting)되지 않는다.**

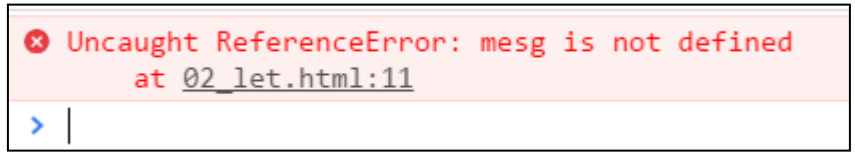
2. let, const 및 블록 스코프

```
<script type="text/javascript">

        if(true){
            var mesg = "hello";
        }
        console.log(mesg);
</script>
```




```
for(var i = 0 ; i < 10 ; i++){
    console.log("aaa");
}
console.log(i);
```


```
<script type="text/javascript">

        if(true){
            let mesg = "hello";
        }
        console.log(mesg);
</script>
```




```
for(let i = 0 ; i < 10 ; i++){
    console.log("aaa");
}
console.log(i);
```

3) const 키워드 (**)

- 상수 작성시 사용.
- const 변수는 선언만 할 수 없으며 반드시 초기화 필요.
- const 변수는 값을 변경할 수 없는 것만 제외하고 let 변수와 기능이 동일.

```
const mesg = "hello";
try{
    mesg = "world";
}catch(e){
    console.log("상수값 변경 불가");
}
```

```
Navigated to http://localhost:8
html
상수값 변경 불가
```

1) 함수 (function)

-함수는 읽기 쉽고 유지 보수가 가능하며 재사용 가능한 코드의 구성요소.

-함수 생성방법 3가지

가. 함수 선언식 (선언적 방법으로서 이름이 있는 함수)
나. 함수 표현식 (함수 리터럴, 이름이 없는 익명 함수)
다. Function 생성자 이용

2) default 파라미터

ES6에서 함수는 전달된 값이 없거나 정의되지 않은 경우 매개 변수를 기본값으로 초기화 할 수 있다.

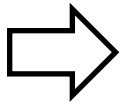
```
<script type="text/javascript">

        function aaa(a = 1, b = '홍길동') {
                console.log(a, b);
        }
        aaa();
        aaa(100);
        aaa(200, "유관순");
        aaa(b = 300, a = "이순신");
</script>
```

```
1 "홍길동"
100 "홍길동"
200 "유관순"
300 "이순신"
>
```

3) rest 파라미터 (**)

- 자바의 가변 인자와 동일한 기능을 한다.
- ‘spread 연산자’ 라고 하는 ‘…변수명’ 이용.
- 내부적으로 배열(Array)로 처리한다. 따라서 Array객체에서 제공하는 메서드를 사용할 수 있다.

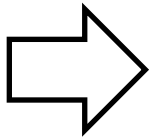
```
<script type="text/javascript">

        function aaa(...xxx){
            console.log(xxx);
            for(let i in xxx){
                console.log(i);
            }
        }
        aaa(1);
        aaa(1,2,3);
</script>
```



```
r/02_rest_parameter_spread.html
▼ [1]
    0: 1
    length: 1
  ▶ __proto__: Array(0)
0
▼ [1, 2, 3]
    0: 1
    1: 2
    2: 3
    length: 3
  ▶ __proto__: Array(0)
0
1
2
```

```
function bbb(n,...xxx){
  console.log(xxx);
  for(let i in xxx){
      console.log(i);
  }
}
  bbb(1);
  bbb(1,2,3);
```

나머지 매개 변수는 함수의 매개 변수 목록에서 마지막에 있어야 된다.



```
▶ []
▶ [2, 3]
0
1
```

spread 연산자

- '…변수' 형식
- 사용 용도 2가지

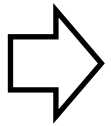
가. 함수의 파라미터

나. 값 (배열 및 객체)

.

```
function aa(x,y,z){
   console.log(x+y+z);
}

var x = [10,20,30];

aa(x); //undefined
aa(...x);  //60
aa(...[10,20,30]); //60
```

```
<script type="text/javascript">

        var aa = [1,2,3, ...[10,9,8]];
        console.log(aa);
        for(let i=0 ; i < aa.length; i++){
            console.log(aa[i]);
        }
</script>
```



```
▶ [1, 2, 3, 10, 9, 8]
1
2
3
10
9
8
```

매우 중요

For object literals (new in ECMAScript 2018):

```
var obj1 = { foo: 'bar', x: 42 };
var obj2 = { foo: 'baz', y: 13 };

var clonedObj = { ...obj1 };
// Object { foo: "bar", x: 42 }

var mergedObj = { ...obj1, ...obj2 };
// Object { foo: "baz", x: 42, y: 13 }
```

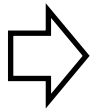
4) arrow 함수

- arrow 함수는 일반적인 함수 표현식을 function키워드 없이 => 이용하여 표현한 방법.

```
([param1, parma2,…param n] )=>statement;
```

1. 파라미터 없고 리턴 타입 없는 함수 형태

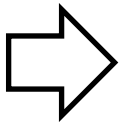
```
//1. 함수 표현식 이용
 var a = function(){
     console.log("a");
  }
a();
```



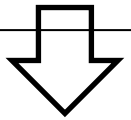
```
//람다 표현식
var a2 = ()=>{
    console.log("a2");
};
a2();
```

2. 파라미터 있고 리턴 타입 없는 함수 형태

```
//1. 함수 표현식 이용
 var a = function(x){
     console.log("a" + x);
  }
a(10);
```



```
 //람다 표현식
 var a2 = (x)=>{
     console.log("a2"+x);
 };
 a2(10);
```



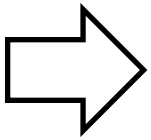
```
// 파라미터가 한개이면 () 생략 가능.
var a3 = x =>{
    console.log("a3"+x);
};
a3(10);
```

```
//2. 함수 표현식 이용
 var k = function(x,y){
     console.log("k" + x+"\t"+y);
  }
k(10,20);
```

```
// 파라미터가 여러개인 경우에는 () 생략불가
var k2 = (x,y)=>{
    console.log("k2" + x+"\t"+y);
}
k2(10,20);
```

3. 파라미터 있고 리턴 타입 있는 함수 형태

```
//1. 함수 표현식 이용
 var a = function(x){
      return x +10;
  }
console.log(a(10));
```



```
//람다 표현식
var a2 = (x)=>{
    return x +10;
};
console.log(a2(10));
```

```
var a3 = x=>{
    return x +10;
};
console.log(a3(10));
```

```
// return 및 {} 생략
var a4 = x=> x + 10;
console.log(a4(10));
```

* arrow 함수 사용시 주의할 점

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

-arrow 함수는 this를 바인딩 하지 않는다. 따라서 객체의 메서드를 생성할 때는 사용하지 않도록 한다.

```
this.n=20;
// 익명 함수
 var a = {
     n:10,
     b: function(){
         console.log(this.n); //10
     }
 }
a.b();

 // arrow 함수
 var a2 = {
         n:10,
         b: ()=>{
             console.log(this.n); // 20
         }
     }
 a2.b();
```

5) generator 함수

- generator 함수는 다음과 같은 표현식을 사용하여 선언한 함수를 의미한다.

문법:

function* 함수명(){}

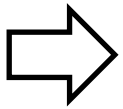
- generator 함수를 호출하면 generator 객체를 생성하여 반환한다.
일반적으로 함수를 호출하면 {}이 실행되지만, generator 함수는 {}을 실행하지 않고 generator 객체를 생성하여 반환한다.

```
<script type="text/javascript">

        function* a(){
            console.log("1");
        }
        var x = a();  // x가 generator 객체이다.
        console.log(typeof a);  // function
        console.log(typeof x);  // object

</script>
```

함수 블록이 실행 안됨.

- generator 함수내의 코드를 실행하기 위하여 generator 객체의 next() 메서드를 호출해야 된다.

```
<script type="text/javascript">

        function* a(){
            console.log("1");
            console.log("2");
            console.log("3");
        }
        var x = a();
        x.next();

</script>
```



```
Navigated to http://localhost:8
erator2.html
1
2
3
```

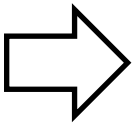
- generator 함수 안에 yield 키워드를 작성하면 함수 블록의 코드 모두를 실행하지 않고 yield 단위로 나누어 실행 가능하다.
따라서 yield가 여러 개가 작성되어 있으면, yield 수만큼 next() 메서드를 호출해야 generator 함수 전체를 실행하게 된다.

```
<script type="text/javascript">

        function* a(){
            console.log("1");
            yield console.log("2");
            console.log("3");
        }
        var x = a();
        x.next();
        x.next();

</script>
```

```
Navigated to http://localhos
erator3_yield.html
1
2
3
```

- yield 키워드 다음에 설정한 표현식을 next() 메서드를 호출할 때 리턴 가능.
리턴 형태는 { value:값 , done:불린값 }
yield가 수행되었으면 false 값 반환, 수행되지 않으면 true 값 반환됨.

```
<script type="text/javascript">

        function* a(k,k2){
            console.log("1");
            yield k+k2;
            yield '홍길동';
            console.log("end");
        }
        var x = a(10,20);
        console.log(x.next());
        console.log(x.next());

</script>
```



```
Navigated to http://localhost:8090/01ECMAS
erator3_yield2.html
1
▶ Object {value: 30, done: false}
▶ Object {value: "홍길동", done: false}
```

4. 디스트럭처링 [destructuring]

1) destructuring – 구조분해할당 (***)

- destructuring은 객체의 구조를 해체하는 것을 의미한다.
배열이나 객체의 데이터를 해체하여 다른 변수로 추출할 수 있다.
이러한 이유 때문에 '구조 분해 할당'이라고도 한다.

가. 배열 디스트럭처링

```
// 1. 배열
let one,two;
[one,two] = [100,200];
console.log(one,two);
```

```
[a, b, ...rest] = [10, 20, 30, 40, 50];

console.log(rest);
// expected output: [30,40,50]
```

```
var [,,n] = [10,20,30]
console.log(n); //30
```

```
// default value
let a2,b2,c2;
[a2,b2,c2=999] = [100,200];
console.log(a2,b2,c2);  // 100 200 999
```

```
let a,b,c;
[a,b,c] = [100,200];
console.log(a,b,c);  // 100 200 undefined
```

4. 디스트럭처링 (destructuring)

나. 객체 디스트럭처링 (*****)

= 기준으로 양쪽 모두 객체형식으로 지정해야 된다. Key값을 기준으로 값을 할당한다.
미리 선언된 변수를 사용하기 위해서는 반드시 () 안에 코드를 작성해야 된다.

```
//2. 객체
let a,b;
({a,b} = {a:100,b:200});
console.log(a,b);

let {a2,b2} = {a2:100,b2:200};
console.log(a2,b2);
```

```
let {a3,b3} = {a3:'100',xxx:'200'};
console.log(a3,b3);   // 100  undefined
```

```
//default value
let {a4,b4=999} = {a4:'100',xxx:'200'};
console.log(a4,b4);   // 100  999
```

다. 파라미터 디스트럭처링 (*****)

호출 받는 함수의 파라미터를 객체 디스트럭처링 형태로 작성하면,
함수에서 직접 key값을 이용하여 value 값을 사용할 수 있다.

```
//3. parameter destructuring
function a({x,y}){
    console.log( x, y);
}
a({x:100,y:200});


function a2([x,y]){
    console.log( x, y);
}
a2(['A','B']);
```

```
//default value
function a3([x,y,z='hello']){
    console.log( x, y, z);
}
a3(['A','B']);
```

```
//1. 일반적인 함수 형태
function a(x,y){
  return x+y;
}
var result = a(10,20);
console.log(result);

//2. 람다함수와 디스트럭처링 적용 형태
var result2 = ([x,y])=> x+y;
console.log(result2([10,20]));
```

5. Object 관련

ES6에서 객체(object)관련하여 추가된 기능들을 정리함.

가. 객체의 key값 이름을 문자열과 변수 조합으로 설정 가능.
[] 이용하여 설정한다.

```
//1. 객체의 key값을  문자열과 문자열 혼합하여 사용 가능.
let mesg = { ['one'+'two']:100};
console.log( mesg,  mesg.onetwo);
```

```
▼Object {onetwo: 100}  100
   onetwo: 100
  ▶__proto__: Object
```

```
//2. 객체의 key값을  문자열과 변수값 혼합하여 사용 가능.
let xyz = "sport";
let mesg2 = { [xyz] :'축구', [xyz+"01"]:'야구',[xyz+"02"]:'농구'};
console.log(mesg2);
```

```
▼Object {sport: "축구", sport01: "야구", sport02: "농구"}
   sport: "축구"
   sport01: "야구"
   sport02: "농구"
  ▶__proto__: Object
```

나.객체속성 축약표시 가능

```
var name='홍';
var age = 20;
var person ={name,age);
console.log(person);
```

다. for~of 반복문

반복할 수 있는 대상은 반드시 iterable 객체만 가능하고 실제값 반환.

```
// 반복 처리 ( for~of)
let a = ['a','b','c'];
for(let x of a){
    console.log(x);
}

let b = "hello";
for(let x2 of b){
    console.log(x2);
}
```

```
Navigated to http://localhf.html
a
b
c
h
e
2 l
o
```

```
let a = ['a','b','c'];
for(let x in a){
    console.log(x);
}

let b = "hello";
for(let x2 in b){
    console.log(x2);
}
```

```
Navigated to http://localhosf.html
0
1
2
0
1
2
3
4
```

라. 메서드 선언 방식 변경 1

ES5에서 사용했던 메서드명:function(){} 형식에서 :function이 제거됨.

```
// ES5에서 사용했던 객체 표현식
var person = {
    name: "홍길동",
    age: 20 ,
    setName: function(n){ this.name = n;},
    setAge: function(n){ this.age = n;},
    getName: function(){ return this.name;},
    getAge: function(){ return this.age;}
};
 person.setName("이순신");
 console.log(person.getName()+"\t"+person.getAge());
```

```
//ES6에서 변경된 메서드 선언방법
var person2 = {
    name:"홍길동",
    age: 20 ,
    setName(n){ this.name = n;},
    setAge(n){ this.age = n;},
    getName(){ return this.name;},
    getAge(){ return this.age;}
};
person2.setName("이순신");
console.log(person2.getName()+"\t"+person2.getAge());
```

마. 메서드 선언 방식 변경 2

- ES6에서는 get 과 set 키워드를 사용하여 메서드 사용시 가독성 향상 가능.
- 사용방법은 . (dot)로 접근하고 일반적인 메서드 호출과 다르게 ( ) 사용 안함.

```
//ES6에서 변경된 메서드 선언방법
var person = {
    name:"홍길동",
    age: 20 ,
    set setName(n){ this.name = n;},
    set setAge(n){ this.age = n;},
    get getName(){ return this.name;},
    get getAge(){ return this.age;}
};
person.setName="이순신";
person.setAge = 40;
console.log(person.getName+"\t"+person.getAge);
```

6. Number / String 객체 추가 메서드

가. Number.isNaN()

- 값이 NaN(Not a Number) 인지 판별.
- NaN 이면 true 리턴하고 아니면 false 리턴.

```
<script type="text/javascript">

        console.log("1:" , Number.isNaN(NaN));
        console.log("2:" , Number.isNaN("NaN"));
        console.log("3:" , Number.isNaN("ABC"));
        console.log("4:" , Number.isNaN(undefined));
        console.log("5:" , Number.isNaN({}));
        console.log("6:" , Number.isNaN(null));
        console.log("7:" , Number.isNaN(''));
        console.log("8:" , Number.isNaN(true));
        console.log("9:" , Number.isNaN(0.123));
        console.log("10:" , Number.isNaN(0/0));

</script>
```

```
Navigated to http://loc
NaN.html
1: true
2: false
3: false
4: false
5: false
6: false
7: false
8: false
9: false
10: true
```

6. Number / String 객체 추가 메서드

나. Number.isInteger()

- 값이 정수인지 판별. 타입까지 판별.
  1.0은 정수이고 1.02는 소수.
- 정수이면 true 리턴하고 아니면 false 리턴.

```
<script type="text/javascript">

        console.log("1:" ,  Number.isInteger(0));
        console.log("2:" ,  Number.isInteger(1.0));
        console.log("3:" ,  Number.isInteger(-123));
        console.log("4:" ,  Number.isInteger("123"));
        console.log("5:" ,  Number.isInteger(1.02));
        console.log("6:" ,  Number.isInteger(NaN));
        console.log("7:" ,  Number.isInteger(true));

</script>
```

```
Navigated to http:/
Integer.html
1: true
2: true
3: true
4: false
5: false
6: false
7: false
```

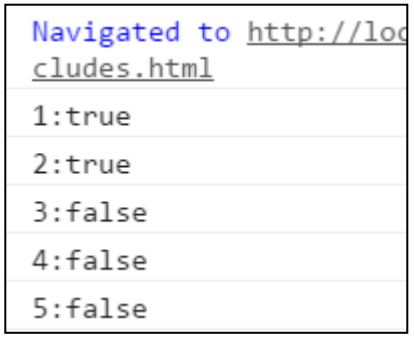
다. Number. parseInt()

```
console.log("1:" ,  Number.parseInt("123") + 1);
```

다. includes() 메서드

- 대상 문자열에 지정된 문자열 존재 여부 판별
- 첫번째 인자에는 찾을 문자열 지정, 두번째 인자에는 시작 인덱스값(옵션)

```
<script type="text/javascript">

            let mesg = "123hello안녕983가나다라";

            console.log("1:" + mesg.includes("12"));
            console.log("2:" + mesg.includes("안녕"));
            console.log("3:" + mesg.includes("안녕하"));
            console.log("4:" + mesg.includes("나라"));
            console.log("5:" + mesg.includes("12" , 5));
</script>
```



6. Number / String 객체 추가 메서드

라. startsWith() 메서드

- 대상 문자열이 지정된 문자열로 시작 여부 판별, 두 번째 인자는 시작 index.

```
<script type="text/javascript">

            let mesg = "123가나다라";

            console.log("1:" + mesg.startsWith("12"));
            console.log("2:" + mesg.startsWith("가나"));
            console.log("3:" + mesg.startsWith("가나",3));
</script>
```

```
1:true
2:false
3:true
```

마. endsWith() 메서드

- 대상 문자열이 지정된 문자열로 끝나는지 여부 판별, 두 번째 인자는 길이.

```
<script type="text/javascript">

            let mesg = "123가나다";

            console.log("1:" + mesg.endsWith("가나다"));
            console.log("2:" + mesg.endsWith("가나"));
            console.log("3:" + mesg.endsWith("가나",5));   // 5 글자만 사용
</script>
```

```
All | Errors  Warnings
1:true
2:false
3:true
```

바. repeat () 메서드

- 대상 문자열을 파라미터에 지정한 수만큼 복제하여 반환.

```
<script type="text/javascript">

            let mesg = "hello";

            console.log('1:' , mesg.repeat());
            console.log('2:' , mesg.repeat(1));
            console.log('3:' , mesg.repeat(2));
            console.log('4:' , mesg.repeat(3));

</script>
```

```
Navigated to http://localhos
epeat.html
1:
2: hello
3: hellohello
4: hellohellohello
```

7. template 리터럴

template 리터럴 (***)

- 문자열 처리를 보다 더 효율적으로 처리하기 위한 방법.
- `` (back-tick) 이라는 역따옴표를 사용하여 표현한다.
- `` 안에 ${exp} 형식의 특별한 표현식을 삽입할 수 있다.

```
let mesg =`hello`;
console.log("1:" , mesg );
```

```
        let mesg2 =`hello
world`;
        console.log("2:" , mesg2 );
```

```
let a2 = 10;
let b2 = 20;
console.log("4:" , `합계:${a2+b2}` );
```

```
let a = '홍길동';
let b = `이순신`;
let result = `이름은 ${a}이고 별명은 ${b}이다`;
console.log("3:" , result);
```

```
1: hello
2: hello
world
3: 이름은 홍길동이고 별명은 이순신이다
4: 합계:30
```

- Array-like 객체

ES6에서 사용되는 객체로서,
{key:value} 형태의 객체 특징 + 배열의 특징 ➔ array-like 객체

문법:

let arrLike = { 0:값, 1:값,…. , length:개수 };

```
var x = {0:"홍길동",1:"이순신",length:3};
console.log(x, Array.isArray(x));
console.log(x[0],x[1],x[2] , x.length);
```

```
Navigated to http://localhost:8090/01ECMAScript/08%2
▶ Object {0: "홍길동", 1: "이순신", length: 3} false
홍길동 이순신 undefined 3
```

- ES6에서 추가된 Array객체 메서드 9가지

가. Array.from() 메서드

새로운 Array객체를 생성.

문법:

Array.from(값, [function, 객체]);

값: array-like 객체 또는 iterable 객체
function: 배열 요소마다 호출되는 함수
객체: function에서 this 키워드 사용시 참조하는 인스턴스.

```
//1. 첫번째 인자는 array-like 객체 또는 iterable 객체 지정
let arr = Array.from('hello');
console.log(arr, arr[0]);
```

```
▶ ["h", "e", "l", "l", "o"] "h"
```

8. Array 관련

```
//2. 두번째 인자는 배열생성하면서 수행되는 함수 지정.
let arr2 = Array.from('world',function(v){
    console.log(">>", v);
    return v;
});
console.log(arr2);
```

```
>> w
>> o
>> r
>> l
>> d
▶ ["w", "o", "r", "l", "d"]
```

```
//3. 세번째 인자는 함수내에서 this 사용시 참조할 객체 지정.
let arr3 = Array.from('happy',function(v){
    console.log("** ", v);
    return v + this.mesg;    //this는 3번째 인자
},{mesg:'값'});
console.log(arr3);
```

```
** h
** a
2 ** p
** y
▶ ["h값", "a값", "p값", "p값", "y값"]
```

```
// array-like 객체 지정
let arr = Array.from({0:100,1:200,length:2});
console.log(arr , arr[0] , arr[1] , arr.length , Array.isArray(arr) );
```

```
▶ [100, 200] 100 200 2 true
```

나. Array.of() 메서드

새로운 Array객체를 생성.

문법:

Array.of(값, [값2,값3,...]);

Array.of() 메서드가 호출되면 우선 Array 객체가 생성되고, 이어서 파라미터에 설정값들을 생성한 배열에 추가한다.

```
let arr = Array.of( 10,20,30 );
console.log(arr, arr[0], arr[1], arr[2], arr.length );
```

▶ [10, 20, 30] 10 20 30 3

```
let arr2 = Array.of( "홍길동","이순신" );
console.log(arr2, arr2[0], arr2[1], arr2.length );
```

▶ ["홍길동", "이순신"] "홍길동" "이순신" 2

다. copyWithin() 메서드

Index 범위의 값을 복사하여 **같은 배열**의 지정한 위치에 설정.

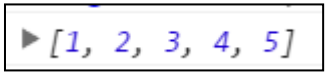
문법:

arr.copyWithin(a, [b, c]);

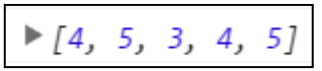
a: 복사된 값을 설정하기 위한 시작 index.
b: 값을 복사하기 위한 시작 index
c: 값을 복사하기 위한 끝 index

결국 b에서 부터 c-1까지 복사해서 a 위치부터 설정한다.

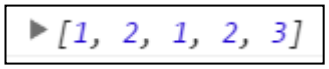
```
var arr =  [1,2,3,4,5];
let copyArr = arr.copyWithin( 0 );
console.log(copyArr);
```



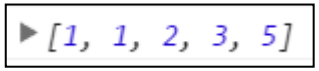
```
var arr =  [1,2,3,4,5];
let copyArr2 = arr.copyWithin( 0 , 3 );
console.log(copyArr2);
```




```
var arr =  [1,2,3,4,5];
let copyArr3 = arr.copyWithin( 2 );
console.log(copyArr3);
```




```
var arr =  [1,2,3,4,5];
let copyArr4 = arr.copyWithin( 1 ,0, 3);
console.log(copyArr4);
```



라. fill() 메서드

Index 범위의 값을 지정한 값으로 변경한다.

문법:

arr.fill(a, [b, c]);

a: 설정할 값.
b: 값을 설정하기 위한 시작 index
c: 값을 설정하기 위한 끝 index

결국 b에서 부터 c-1까지 a 값으로 설정한다.

```
var arr =  [1,2,3,4,5];
let copyArr = arr.fill( 9 );
console.log(copyArr);


var arr2 =  [1,2,3,4,5];
let copyArr2 = arr2.fill( 9, 3 );
console.log(copyArr2);

var arr3 =  [1,2,3,4,5];
let copyArr3 = arr3.fill( 9, 2, 4 );
console.log(copyArr3);
```

```
▶ [9, 9, 9, 9, 9]
▶ [1, 2, 3, 9, 9]
▶ [1, 2, 9, 9, 5]
```

마. entries() 메서드

배열을 {key:value} 형태로 반환. key는 배열 index값이고 value는 배열요소 값.
for~of 반복문 사용하여 iterator로 처리 가능.

문법:

arr.entries();

```
var arr =  [10,20,30,40,50];
let x = arr.entries();

for(var [key,value] of x){
    console.log(key, value);
}
```

```
Navigated to http://
0 10
1 20
2 30
3 40
4 50
```

바. keys() 메서드

배열에서 key값만 반환

문법:

arr.keys();

```
var arr =  [10,20,30,40,50];
let x = arr.keys();

for(var key of x){
    console.log(key);
}
```

```
Navigated to ht
0
1
2
3
4
```

사. find() 메서드

callback 함수에서 true를 반환하면 처리중인 배열요소 값을 반환한다.

문법:

arr.find(function [,obj]);

function: 배열 요소가 반복 될 때마다 호출된다. true 리턴시 find() 종료하고 처리중인 배열 요소를 반환한다.
function(ele,idx,all){} 처럼 3가지 파라미터 변수 설정 가능하다.
- ele : 처리중인 배열 요소
- idx : 처리중인 배열 index
- all: 전체 배열

obj : function에서 this로 접근할 객체 지정.

```
var arr =  [10,20,30,40,50];
let xxx = arr.find(function(ele,idx, all){
    console.log(ele,idx,all);
    return ele == 30;
});
console.log(xxx);
```

```
Navigated to http://localhost
10 0 ▶ [10, 20, 30, 40, 50]
20 1 ▶ [10, 20, 30, 40, 50]
30 2 ▶ [10, 20, 30, 40, 50]
30
```

아. findIndex() 메서드

callback 함수에서 true를 반환하면 처리중인 배열요소의] index 값을 반환한다.

문법:

arr.findIndex(function [,obj]);

function: 배열 요소가 반복 될 때마다 호출된다. true 리턴시 findIndex() 종료하고 처리중인 배열 요소의 index값을 반환한다.
function(ele,idx,all){} 처럼 3가지 파라미터 변수 설정 가능하다.
- ele : 처리중인 배열 요소
- idx : 처리중인 배열 index
- all: 전체 배열

obj : function에서 this로 접근할 객체 지정.

```
var arr =  [10,20,30,40,50];
let xxx = arr.findIndex(function(ele,idx, all){
    console.log(ele,idx,all);
    return ele == 30;
});
console.log(xxx);
```

```
10 0 ▶ Array(5)
20 1 ▶ Array(5)
30 2 ▶ Array(5)
2
```

* 자바스크립트 class 작성 방법 2가지

가. 클래스 선언 방식 (***)

```
//1. 클래스 선언식
 class Person{
      setName(name){
          this.name = name;
      }
      setAge(age){
          this.age = age;
      }
      getName(){
          return this.name;
      }
      getAge(){
          return this.age;
      }
 }
```

```
let p = new Person();
p.setName("홍길동");
p.setAge(20);
console.log(p.name , p.age);
console.log(p.getName(),p.getAge());
```

```
Navigated to http://1
홍길동 20
홍길동 20
```

가. 클래스 선언 방식 / setter 메서드 , getter 메서드

```
//1. 클래스 선언식
class Person{
    set setName(name){
        this.name = name;
    }
    set setAge(age){
        this.age = age;
    }

    get getName(){
        return this.name;
    }
    get getAge(){
        return this.age;
    }
}
```

```
let p = new Person();
//p.setName("홍길동");
//p.setAge(20);
p.setName="홍길동";
p.setAge=20;
console.log(p.name , p.age);
console.log(p.getName,p.getAge);
```

```
홍길동 20
홍길동 20
```

나. 클래스 표현 방식

```
// 2. 클래스 표현식
var Person = class {
    set setName(name){
        this.name = name;
    }
    set  setAge(age){
        this.age = age;
    }

    get getName(){
        return this.name;
    }
    get getAge(){
        return this.age;
    }
}
```

```
let p = new Person();
//p.setName("홍길동");
//p.setAge(20);
p.setName="홍길동";
p.setAge=20;
console.log(p.name , p.age);
console.log(p.getName,p.getAge);
```

* 생성자 (constructor)

- 클래스 인스턴스를 생성하고 생성한 인스턴스를 초기화 역할.
- 오버로딩 생성자 안됨. 반드시 constructor 는 단 하나만 지정 가능.

만약 명시적으로 constructor를 지정하지 않으면 prototype의 생성자가 호출됨.
이 생성자를 'default 생성자 ' 라고 한다.

```
class Person{

    // 생성자 ( 클래스당 하나씩 )
    constructor(name,age){
        console.log("constructor(name,age)");
        this.name = name;
        this.age = age;
    }
    setName(name){
        this.name = name;
    }
    setAge(age){
        this.age = age;
    }

    getName(){
        return this.name;
    }
    getAge(){
        return this.age;
    }
}
```

```
let p = new Person();
console.log(p.name , p.age);
console.log(p.getName(),p.getAge());

let p2 = new Person("홍길동",20);
console.log(p2.name , p2.age);
console.log(p2.getName(),p2.getAge());
```

```
Navigated to http://loca
constructor(name,age)
undefined undefined
undefined undefined
constructor(name,age)
홍길동 20
홍길동 20
```

* 상속 (inheritance)

- ES6에서는 자바와 같이 extends 키워드로 상속 표현.
- 부모 클래스의 멤버를 자식 클래스가 상속 받아서 사용 가능

constructor와 상속

1. 자식 클래스와 부모 클래스 양쪽 constructor를 작성하지 않아도 인스턴스 생성 된다. (default constructor 사용)
2. 부모 클래스에만 constructor를 작성하면, 자식 클래스의 'default 생성자'가 호출되고 부모 클래스의 constructor가 호출된다.
3. 자식 클래스에만 constructor를 작성하면 자식 클래스의 constructor가 호출되고 반드시 부모 constructor를 명시적으로 호출해야 된다.
4. 자식과 부모 클래스 양쪽 constructor를 작성하면 자식 constructor가 호출되지만 반드시 부모 constructor를 명시적으로 호출해야 된다.

* 메서드 오버라이딩 (method overriding)

```
class Employee{
    constructor(name,salary){
        this.name = name;
        this.salary = salary;
    }
    getEmployee(){
        return this.name+"\t"+this.salary;
    }
}

class Manager extends Employee{
     constructor(name,salary,depart){
        super(name,salary); // 반드시 명시적으로 호출
        this.depart = depart;
    }
     getEmployee(){
        return super.getEmployee()+"\t"+ this.depart;
    }
}
```

```
let man = new Manager("홍길동",2000,"관리");
console.log(man.getEmployee());
```

```
Navigated to http://lo
홍길동    2000    관리
```

* static method

```
class Employee{
    getEmployee(){
        return "hello";
    }
    static getXXX(){
        return "world";
    }
}
```

일반 메서드는 객체 생성후에
인스턴스 변수로 참조

static 메서드는 클래스명.메서드
형식으로 참조

```
console.log(Employee.getXXX());

let emp = new Employee();
console.log(Employee.getXXX(), emp.getEmployee());
```

```
world
world hello
```

* Promise 객체

- javascript에서 특정 작업을 비동기 처리 해주는 객체.
- Promise 객체에서 비동기 처리를 내부적으로 해주기 때문에 이에 맞추어 코드를작성.

```
function xxx(){

  return new Promise(function(resolve,reject){
      console.log("1");
      resolve();
      //reject();
      console.log("2");
  });
}
```

```
1
2
end
success
```

```
1
2
end
fail
```

```
xxx().then(function(){
   console.log("success");
},function(){
    console.log("fail");
});
console.log("end");
```

- 실행 동작은 다음과 같다.

가. 성공 -> resolve() 호출 -> then 메서드의 첫번째 함수 수행
나. 실패 -> reject() 호출 -> then 메서드의 두번째 함수 수행

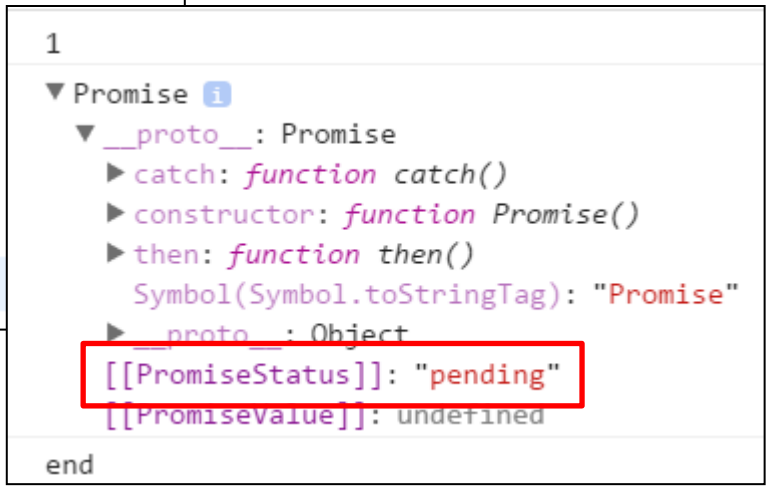
주의할 점은 코드 끝까지 수행하고 나서 resolve()또는 reject()가 나중에 호출된다. 즉 비동기적으로 수행된다.

* Promise 상태 2가지

- 비동기로 처리가 되기 때문에 특정 상태를 저장해야 된다.
- [[PromiseStatus]] 속성에 저장됨.

가. Pending 상태

```
function xxx(){
  return new Promise(function(resolve,reject){
      console.log("1");
  });
}

var result = xxx();
console.log(result);
console.log("end");
```



나. resolve 상태 또는 reject 상태 (settled 상태)

```
function xxx(param){
  return new Promise(function(resolve,reject){
       console.log("1");
       if(param == "ok"){
           resolve();
       }else{
           reject();
       }
       console.log("2");
  });
}
```

```
var result = xxx("ok");
console.log(result);
console.log("end");
```

```
1
2
▼Promise
  ▶__proto__: Promise
    [[PromiseStatus]]: "resolved"
    [[PromiseValue]]: undefined
end
```

```
var result = xxx("ok2");
console.log(result);
console.log("end");
```

```
1
2
▼Promise {[[PromiseStatus]]: "rejected", [[PromiseValue]]: undefined}
  ▶__proto__: Promise
    [[PromiseStatus]]: "rejected"
    [[PromiseValue]]: undefined
end
▶Uncaught (in promise) undefined
```

* 파라미터 사용
 - resolve(값) 또는 resovle(값) 사용 가능. 단 하나의 파라미터 값만 사용 가능
따라서 여러 개의 값을 전달하기 위해서는 배열 이용.
 - [[PromiseValue]]에 저장됨

```
function xxx(param){

  return new Promise(function(resolve,reject){
      console.log("1");
      if(param == "ok"){
          resolve(100);
      }else{
          reject(200);
      }
      console.log("2");
  });
}
var result = xxx("ok");
console.log(result);
result.then(function(x){
   console.log("success"+ x);
},function(y){
    console.log("fail" + y);
});
console.log("end");
```

```
1
2
▼ Promise {[[PromiseStatus]]: "resolved", [[PromiseValue]]: 100}
  ▶ __proto__: Promise
    [[PromiseStatus]]: "resolved"
    [[PromiseValue]]: 100
end
success100
```

```
function xxx(param){

  return new Promise(function(resolve,reject){
      console.log("1");
      if(param == "ok"){
          resolve([100,1000]);
      }else{
          reject({key:200,key2:2000});
      }
      console.log("2");
  });
}
var result = xxx("ok");
console.log(result);
result.then(function(x){
   console.log("success", x[0] , x[1] );
},function(y){
    console.log("fail" , y.key , y.key2);
});
console.log("end");
```

```
1
2
▼ Promise {[[PromiseStatus]]: "resolved", [[
  ▶ __proto__: Promise
    [[PromiseStatus]]: "resolved"
  ▼ [[PromiseValue]]: Array(2)
      0: 100
      1: 1000
      length: 2
    ▶ __proto__: Array(0)
end
success 100 1000
```

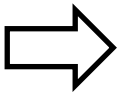
* then() 메서드의 체인

```
function xxx(){

  return new Promise(function(resolve,reject){
      console.log("1");
      resolve();
      //reject();
      console.log("2");
  });
}

xxx().then(function(){
   console.log("success1");
},function(){
    console.log("fail1");
}).then(function(){
    console.log("success2");
},function(){
    console.log("fail2")
});
console.log("end");
```

```
1
2
end
success1
success2
```

```
function xxx(){

  return new Promise(function(resolve,reject){
      console.log("1");
      //resolve();
      reject();
      console.log("2");
  });
}

xxx().then(function(){
   console.log("success1");
},function(){
    console.log("fail1");
}).then(function(){
    console.log("success2");
},function(){
    console.log("fail2")
});
console.log("end");
```

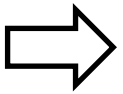
*의도와 다르게 두번 째 then()에서 success2 수행됨.



```
Navigated to http://localh
1
2
end
fail1
success2
```

```
function xxx(){

  return new Promise(function(resolve,reject){
      console.log("1");
      //resolve();
      reject();
      console.log("2");
  });
}

xxx().then(function(){
   console.log("success1");
},function(){
    console.log("fail1");
    reject();
}).then(function(){
    console.log("success2");
},function(){
    console.log("fail2")
});
console.log("end");
```

* reject() 명시적 호출로 then()에서 fail2 수행됨.



```
Navigated to http://lo
1
2
end
fail1
fail2
```

```
function xxx(){

  return new Promise(function(resolve,reject){
      console.log("1");
      resolve();
      console.log("2");
  });
}

xxx().then(function(){
   console.log("success1");
   return "홍길동";
}).then(function(p){
    console.log("success2" + p);
    return "안녕하세요"+p;
}).then(function(p2){
    console.log("success3" + p2);
});;
console.log("end");
```

```
Navigated to http://localhost:809
1
2
end
success1
success2홍길동
success3안녕하세요홍길동
```

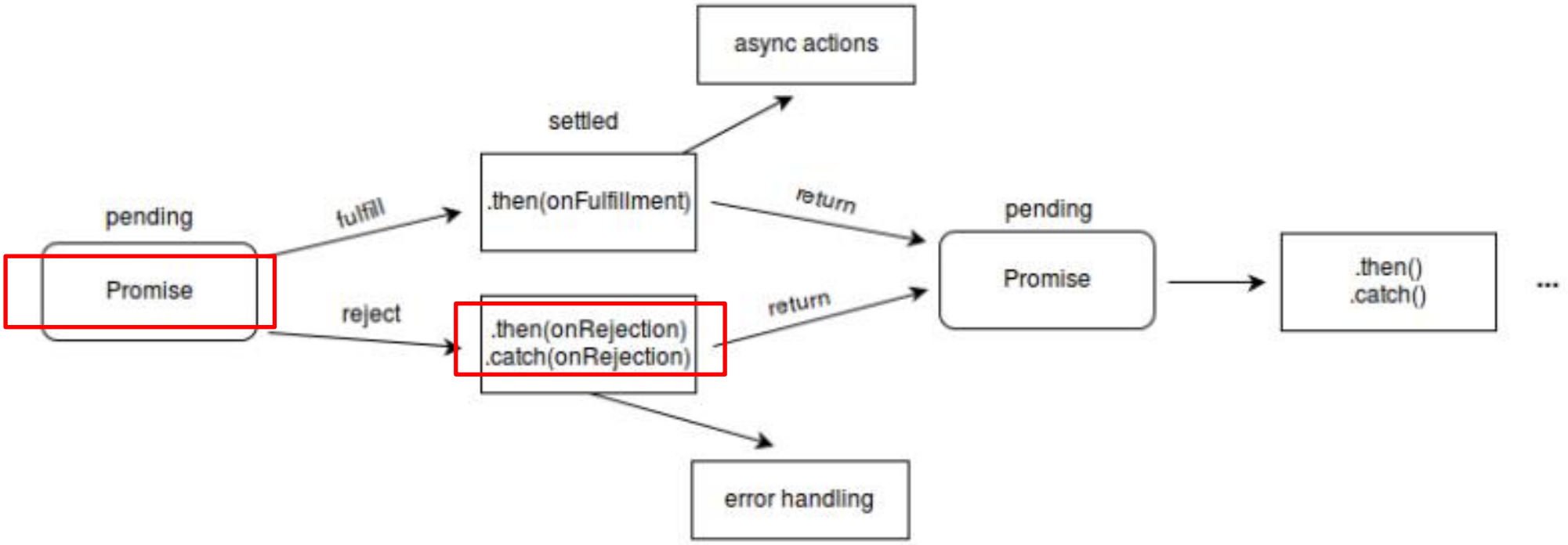
* catch를 이용한 reject() 처리

- then() 메서드의 두 번째 함수에서 처리했던 실패 코드를 catch() 에서 사용.

```
function xxx(){

  return new Promise(function(resolve,reject){
      console.log("1");
      reject();
      console.log("2");
  });
}

xxx().then(function(){
   console.log("success");
 }).catch(function(){
     console.log("fail");
 });
console.log("end");
```

```
1
2
end
fail
```

* Promise 상태도


async actions
settled
pending
fulfil
.then(onFulfillment)
return
pending
Promise
.then()
.catch()
...
Promise
reject
.then(onRejection)
.catch(onRejection)
return
error handling

* 자바스크립트 코드를 모듈화하기 위한 방법.

가. export 사용하여 외부에서 사용 가능하도록 설정한다.

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 …;
```

나. import 사용하여 외부 모듈을 사용할 수 있다.

Syntax

```
import defaultExport from "module-name";
import * as name from "module-name";
import { export } from "module-name";
import { export as alias } from "module-name";
import { export1 , export2 } from "module-name";
import { export1 , export2 as alias2 , [...] } from "module-name";
import defaultExport, { export [ , [...] ] } from "module-name";
import defaultExport, * as name from "module-name";
import "module-name";
var promise = import(module-name);
```

11. module

b.js

```
1
2  export class Person{
3      constructor(n){
4          this.name = n;
5      }
6  }
7
8  export function aaa(){
9      console.log("hello");
10 }
11
```

c.js

```
1
2  export default function bbb(){
3      console.log("world");
4  }
5
```

a.js

```
1
2  import {Person,aaa} from './b.js';
3  import bbb from './c.js';
4
5    var p = new Person("aa");
6    aaa();
7    bbb();
8   console.log(">>>",p.name);
```

a.js 01_module.html

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <meta charset="UTF-8">
5  <title>Insert title here</title>
6  <script type="module" src="a.js"></script>
7  </head>
8  <body>
9
10 </body>
11 </html>
```

수고하셨습니다.