JavaScript 언어 구성

js의 기본문법은 C와 유사하고,
객체지향 관련은 프로토타입 기반으로,
모든 타입(자료형)은 해당 타입 고유의 object 자료형을 상속하여 만들어진다.

[기본 문법]

- **C와 동일하나 포인터의 개념이 없음** (코딩할 때만 없다는 뜻)
- **타입은 자동으로 지정 (다이나믹 타이핑)**
    - 배열조차 크기나 타입 등을 지정하지 않아도 됨

js는 기본적으로 스크립트 언어(=인터프리터)이기 때문에 위에서 아래로 코드가 순차실행.

html에서 실행할 때도 마찬가지로, html에서 js의 실행위치에 따라
html태그가 먼저 로딩 될 지, js가 먼저 실행 될 지가 다름.

따라서 **보통의 html파일은 body태그 가장 밑에 js를 실행하도록 위치시키는게 좋음**

1. [변수의 할당]

var / let/ const [“변수” = “할당할 값”;]

- **var** : 전역변수 , 사용시 혼란을 주기 때문에 사용 X
- **let**, **const** : c언어와 동일한 동작. 사용 O

🚨변수 호이스팅

```
console.log(foo); // ① undefined
var foo = 123;
console.log(foo); // ② 123
{
  var foo = 456;
}
console.log(foo); // ③ 456
```

문제) console.log(foo)에서 변수 foo가 아직 선언되지 않았으므로
“ReferenceError: foo is not defined”를 기대했함

실행 결과) 실제로는 undefined가 출력됨

원인) **호이스팅 : var 선언문이나 function 선언문 등 모든 선언문이 해당 Scope의 선두로 옮겨진 것처럼 동작하는 형태 (즉, 자바스크립트는 모든 선언문(var, let, const, function, function*, class)이 선언되기 이전에 참조 가능하다.)**

var 키워드로 선언된 변수의 생명 주기

선언 단계(Declaration phase)

변수 객체(Variable Object)에 변수를 등록한다. 이 변수 객체는 스코프가 참조하는 대상이 된다.

초기화 단계(Initialization phase)

변수 객체(Variable Object)에 등록된 변수를 메모리에 할당한다. 이 단계에서 변수는 undefined로 초기화된다.

할당 단계(Assignment phase)

undefined로 초기화된 변수에 실제값을 할당한다.

var 키워드로 선언된 변수는 선언 단계와 초기화 단계가 한번에 이루어짐

①이 실행되기 이전에 `var foo = 123;`이 호이스팅되어 ①구문 앞에 `var foo;`가 옮겨진다.(실제로 변수 선언이 코드 레벨로 옮겨진 것은 아니고 변수 객체(Variable object)에 등록되고 undefined로 초기화된 것이다.)

이후 “var foo = 456;”에서 실값이 할당됨.

cf) 1-5 [함수] _ 함수 호이스팅

함수 선언문으로 정의된 함수는 자바스크립트 엔진이 스크립트가 로딩되는 시점에 바로 초기화하고
이를 함수 함수 객체에 저장. 즉
함수 선언, 초기화, 할당이 한번에 이루어짐

🚨레벨 스코프

자바스크립트는 블록 레벨 스코프가 아닌 **함수 레벨 스코프**.
하지만
let, const 키워드 사용 시 **블록 레벨 스코프** 사용 가능.

함수 레벨 스코프(Function-level scope)

함수 내에서 선언된 변수는 함수 내에서만 유효하며 함수 외부에서는 참조할 수 없다. 즉, 함수 내부에서 선언한 변수는 지역 변수이며 함수 외부에서 선언한 변수는 모두 전역 변수이다.

블록 레벨 스코프(Block-level scope)

코드 블록 내에서 선언된 변수는 코드 블록 내에서만 유효하며 코드 블록 외부에서는 참조할 수 없다.

2. [데이터 타입]

“내부적으로 전부 해당 타입을 만드는 object를 사용해 만든다”

number / string / boolean / null / undefined / object ...

primitive 타입(원시타입) : number, string, boolean, null, undefined

⇒ **변경 불가능한 값(상수, immutable) & 값에 의한 전달, 값에 의한 호출**

let str =’aaa’ 를 변경하려면 새 상수 ‘bbb’를 할당해야 함

reference 타입(*객체 타입*) : primitive 타입 외의 나머지(함수, 배열 등)

⇒ **변경 가능한 값(원소, mutable) & 참조에 의한 전달, 참조에 의한 호출**

let arr = [1, 2, 3, 4]의 3번째 값 변경 : arr[2] = 6;

모든 데이터 타입은 **“프로토타입”**을 갖고 있다.
프로토타입은 타 언어의 class라고 보면 된다.
string object(class)로 만들어진 'abcdef'는 string 타입의 프로토타입(멤버필드)을 사용할 수 있다.

즉 'abcdef' 같은 string 문자열에 우리가
.includes/charat/concat 등을 사용할 수 있는것은 다이나믹 타이핑이 이루어질 때 string object를 통해 'abcdef'가 string 데이터 타입으로 만들어지기 때문이다.

[추가 정보]

- **<number>**
  - 숫자 타입의 값은 배정밀도 64비트 부동소수점 형을 따름
    - 즉, 모든 수를 실수로 처리. 정수만을 표현하는 특별한 타입 X
- **<string>**
  - 문자열은 **‘ ’ 또는 “ “ 모두 사용 가능**. 일반적으로 작은 따옴표 사용
  - 원시 타입의 변경 불가능하다는 특징

```
var str = 'Hello';
str = 'world';
```

1. 첫번째 구문 : 메모리에 ‘Hello’가 생성되고 str이 ‘Hello’를 가리킴
2. 두번째 구문 : 메모리에 ‘world’를 생성하고 str이 기존에 가리키던 ‘Hello’가 아니라 ‘world’를 가리킴. 이때 문자열 ‘Hello’와 ‘world’는 모두 메모리에 있으며 변수 str이 가리키는 값이 변경된 것.

  - 즉. 가리키고 있던 메모리의 값은 변경 불가. 새로운 메모리를 가리키는 것.

```
var str = 'string';
// 문자열은 유사배열이다.
for (var i = 0; i < str.length; i++) {
  console.log(str[i]);
}

// 문자열을 변경할 수 없다.
str[0] = 'S';
console.log(str); // string
```

위 코드처럼 문자열을 변경할 수 없음

기존 문자열을 변경할 수 없고 새로운 문자열을 새롭게 할당하는 것만 가능

- **<undefined>**
  - 선언 이후 값을 할당하지 않은 변수
    - 자바스크립트 엔진에 의해 초기화
- **<null>**
  - 의도적으로 변수에 값이 없음을 명시할 때 사용
    - 변수가 기억하는 메모리 어드레스의 참조 정보를 제거하는 것을 의미하며 자바스크립트 엔진은 누구도 참조하지 않은 메모리 영역에 대해 가비지 콜렉션을 수행할 것임.

3. [연산자]

: 동일

```
45    let num = '10a' * 54;
46    const str = 'string11';
47    // str = '24'; // error
48    console.log({num, str});

문제    출력    디버그 콘솔    터미널

~/code/JS ❯ node script.js
{ num: NaN, str: 'string11' }
~/code/JS ❯
```

1_learn_assignment.js

[추가 정보]

- “문”의 끝에 **세미콜론** 붙이는 것을 권장
- **동등 비교(==) vs 일치 비교(===)**
  - 동등 비교 : x와 y의 값이 일치한지...
  - 일치 비교 : x와 y의 값과 타입이 일치한지...
  - 동등 비교연산자는 예측하기 힘든 결과를 도출함
    (아래의 동등 비교 예시를 이해할 필요는 없음)

```
'' == '0'           // false
0 == ''             // true
0 == '0'            // true

false == 'false'    // false
false == '0'        // true

false == undefined  // false
false == null       // false
null == undefined   // true
```

  - **따라서 일치 비교 연산자를 사용하는 것 권장**
- **삼항 조건 연산자** (C와 동일)

- 대부분의 if..else문은 삼항 조건 연산자로 바꿔서 사용할 수 있음
- 세가지 경우의 수를 갖는 삼항 조건 연산자

```
var num = 2;

// 0은 false로 취급된다.
var kind = num ? (num > 0 ? '양수' : '음수') : '영';
console.log(kind); // 양수
```

- **typeof 연산자**

```
typeof ''               // "string"
typeof 1                // "number"
typeof NaN              // "number"
typeof true             // "boolean"
typeof undefined        // "undefined"
typeof Symbol()         // "symbol"
typeof null             // "object"
typeof []               // "object"
typeof {}               // "object"
typeof new Date()       // "object"
typeof /test/gi         // "object"
typeof function () {}   // "function"
```

4. [타입]

: 상식적으로 자동 변환

(숫자+문자 = 문자) / (숫자 * 문자('10'같은) = 숫자) / 안맞으면 NaN

```
(function learn_type(run) {
    if (!run) return;

    let toBool = Boolean(1);
    console.log({toBool});

    let toStr = (12).toString()
    console.log({toStr});
    toStr = 12 + '';
    console.log({toStr});

    let toNum = Number('12');
    console.log({toNum});
    toNum = +'12';
    console.log({toNum});

    let nan = Number('str');
    console.log({nan});
})(1);
```

1_learn_type.js

```
{ num: NaN, str: 'string11' }
{ toBool: true }
{ toStr: '12' }
{ toStr: '12' }
{ toNum: 12 }
{ toNum: 12 }
{ nan: NaN }
```

typeof : 피연산자의 데이터 타입을 문자열로 반환

Object.prototype.toString

기본적으로 toString() 메서드는 Object에서 비롯된 모든 객체에 상속됨.

따라서 **메서드가 사용자 지정 개체에서 재정의되지 않으면 toString()은 “[object “type”]”을 반환.**

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

const dog1 = new Dog('Gabby');

//toString 메소드 반드시 재정의
Dog.prototype.toString = function dogToString() {
    return `${this.name}`;
};

console.log(dog1.toString());
// Expected output: "Gabby"
```

[추가 정보]

- +, - 단항 연산자에 “숫자 타입이 아닌 피연산자”가 사용되면 피연산자를 “숫자 타입”으로 변환하여 반환한다.
- 문자열 인터폴레이션(String Interpolation)

- 표현식의 평가 결과를 문자열 타입으로 암묵적 타입 변환

```
console.log(`1 + 1 = ${1 + 1}`); // "1 + 1 = 2"
```

- Truthy 값(참으로 인식할 값) vs Falsy 값(거짓으로 인식할 값)
  - Falsy값 종류 : false / undefined / null / 0, -0 / NaN / “(빈문자열)

5. [함수]

5-1) 선언문 vs 표현식

선언문 / 표현식 두 방법으로 정의가 가능

- 함수명 생략 불가능

선언문

function name(param) {

return ...

}

표현식 : 함수를 변수에 넣었음

let func = function name(param) {

return ...

}

*** 선언문과 표현식의 차이는 변수에 넣었는지 안 넣었는지 여부**

함수 표현식으로 정의한 함수는 함수명을 생략할 수 있음(익명 함수)

```
// 기명 함수 표현식(named function expression)
var foo = function multiply(a, b) {
    return a * b;
};

// 익명 함수 표현식(anonymous function expression)
var bar = function(a, b) {
    return a * b;
};
```

함수명으로 함수를 호출하면 안된다!!

함수가 할당된 변수를 사용해 함수를 호출하지 않고 기명 함수의 함수명을 사용해 호출하게 되면
에러가 발생. 이는 함수 표현식에서 사용한 함수명은 외부 코드에서 접근 불가능하기 때문.

(함수 선언문으로 정의한 함수를 함수명으로 호출할 수 있는 이유는 자바 스크립트 엔진에 의해 함수 표현식으로 형태가 변경되었기 때문)

Function 생성자 함수 (일반적 사용 X)

new Function(arg1, arg2, ... argN, functionBody)

(앞에서 배운 함수 선언문과 함수 표현식은 모두 내장 함수 Function 생성자 함수로
함수를 생성하는 것을 단순화시킨 축약법이라고 할 수 있다)

```
87   (function learn_function(run) {
88       if (!run) return;
89
90       let func = new Function('a', 'b', 'return a+b');
91       console.log({func, val: func(1, 3)}, Function('a', 'b', 'return a+b'));
92   })(1);
```

문제 출력 디버그 콘솔 터미널

```
{ func: [Function: anonymous], val: 4 } [Function: anonymous]
```

5-2) 함수 호이스팅

```
> //함수 호이스팅
  var res = square(5);

  function square(number) {
      return number * number;
  }
< undefined
```

호이스팅 : 선언문이 해당 Scoper의 선두로 옮겨진 것처럼 동작하는 특성

위 코드를 보면 함수 선언문으로 함수가 정의되기 이전에 함수 호출이 가능하다. 함수 선언문의 경우, 함수 선언의 위치와는 상관없이 코드 내 어느 곳에서든지 호출이 가능한데 이것을 함수 호이스팅(Function Hoisting)이라 한다.

🚨 함수 선언문으로 정의된 함수는 함수 선언, 초기화, 할당이 한번에 이루어짐.

(자바스크립트 엔진이 스크립트가 로딩되는 시점에 바로 초기화하고 이를 함수 객체에 저장).

🚨 함수 표현식의 경우 함수 호이스팅이 아니라 변수 호이스팅이 발생

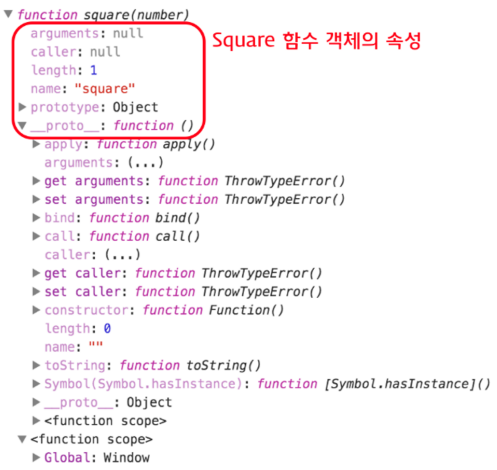
```
var res = square(5); // TypeError: square is not a function

var square = function(number) {
  return number * number;
}
```

변수 호이스팅은 변수 생성 및 초기화와 할당이 분리되어 진행되기 때문에
호이스팅된 변수는 undefined로 초기화 되고 실제값의 할당은 할당문에서 이루어짐.

5-3) 함수 객체의 프로퍼티

```javascript
function square(number) {
  return number * number;
}
console.dir(square);
```



arguments 프로퍼티

- 함수 호출 시 전달된 인수들(argument)의 정보를 담음
- 순회가능한 유사 배열 객체(length 프로퍼티를 가진 객체)
  - 유사 배열 객체는 배열이 아니므로 배열 메소드를 직접적으로는 사용 불가

```javascript
function multiply(x, y) {
    console.log(arguments);
    return x * y;
  }

  multiply();         // {}
  multiply(1);        // { '0': 1 }
  multiply(1, 2);     // { '0': 1, '1': 2 }
  multiply(1, 2, 3); // { '0': 1, '1': 2, '2': 3 }
```

- **arguments.length**

  → 함수 호출시 인자의 갯수 (가변 인자 함수 구현 유용)

caller 프로퍼티

- 자신을 호출한 함수를 의미

```javascript
function foo(func) {
    var res = func();
    return res;
}

  function bar() {
    return 'caller : ' + bar.caller;
}

console.log(foo(bar)); // caller : function foo(func) {...}
console.log(bar());    // null (browser에서의 실행 결과)
```

length 프로퍼티

- 함수 정의 시 작성된 매개변수 갯수를 의미

name 프로퍼티

- 함수명을 의미
- 기명함수의 경우 함수명, 익명함수의 경우 빈문자열

`__proto__` 접근자 프로퍼티

- 모든 객체에 [[Prototype]] 내부 슬롯이 있고 해당 슬롯은 프로토타입 객체를 가리킴
  - 프로토타입 객체 : 다른 객체에 공유 프로퍼티를 제공하는 객체
- [[Prototype]] 내부 슬롯이 가리키는 프로토타입 객체에 접근하기 위해 사용

prototype 프로퍼티

- 함수 객체만이 소유하는 프로퍼티
  - 즉, 일반 객체에는 prototype 프로퍼티가 없음
- 함수가 객체를 생성하는 생성자 함수로 사용될 때,
  생성자 함수가 생성한 인스턴스의 프로토타입 객체를 가리킴

함수의 여러 형태

-즉시 실행 함수

함수의 정의와 동시에 실행되는 함수

최초 한번만 호출되며 다시 호출할 수는 없음

⇒ 최초 한번만 실행이 필요한 초기화 처리 등에 사용

```
// 기명 즉시 실행 함수(named immediately-invoked function expression)
(function myFunction() {
    var a = 3;
    var b = 5;
    return a * b;
  }());

  // 익명 즉시 실행 함수(immediately-invoked function expression)
  (function () {
    var a = 3;
    var b = 5;
    return a * b;
  }());

  // SyntaxError: Unexpected token (
  // 함수선언문은 자바스크립트 엔진에 의해 함수 몸체를 닫는 중괄호 뒤에 ;가 자동 추가된다.
  function () {
    // ...
  }(); // => };();

  // 따라서 즉시 실행 함수는 소괄호로 감싸준다.
  (function () {
    // ...
  }());

  (function () {
    // ...
  })();
```

-내부 함수

-재귀 함수

-콜백 함수

함수를 명시적 호출하는 방식이 아니라, **특정 이벤트가 발생했을 때 시스템에 의해 호출되는 함수**

주로 비동기식 처리 모델에 사용

```
setTimeout(function () {
    console.log('1초 후 출력된다.');
  }, 1000);
```

대표적 예) 이벤트 핸들러 처리

```
<!DOCTYPE html>
<html>
<body>
  <button id="myButton">Click me</button>
  <script>
    var button = document.getElementById('myButton');
    button.addEventListener('click', function() {
      console.log('button clicked!');
    });
  </script>
</body>
</html>
```

- react에서...
  - onClick={handleClick()}
    - 컴포넌트가 렌더링될 때 함수를 바로 실행하며, 대부분의 경우 의도하지 않은 동작을 일으킴
  - onCick={ () ⇒ {handleClick() }
    - 사용자가 클릭할 때 handleClick 함수를 실행, 일반적으로 이벤트 핸들러로 사용되는 일반적 방법
    - 인자를 전달하고자 할 때도 화살표 함수 사용!!

[추가 정보]

- 다중 for문에서 가장 외부 for문을 빠져나가는 방법
  - 레이블 문을 사용

```
// outer라는 식별자가 붙은 레이블 for 문
outer: for (var i = 0; i < 3; i++) {
  for (var j = 0; j < 3; j++) {
    // i + j === 3이면 외부 for 문을 탈출한다.
    if (i + j === 3) break outer;
  }
}

console.log('Done!');
```

6. [배열]

- object 자료형과 동일하나 index(key)만 0부터 시작하는 자연수로 자동 지정(동적 배열)되는 자료형.

c와 달리 서로 다른 타입을 넣을 수 있다.
(한 타입만 사용하는게 배열의 성능면에서는 좋다)

```
> let arr1 = ['asdf', 14, {a: 12, b: 'asdf'}, function(){}]
< undefined
> const arr2 = [];
< undefined
```

const로 배열을 할당해도 원소의 값을 변경할 수 있음

- 배열에 주소가 들어가기 때문
- const로 선언한 배열은 arr2 = ['asdf'] 등 다른 object나 값을 할당하는 행위는 불가능

array object의 프로토타입 함수들은 필수로 숙지해야함.

concat / find / push / slice / splice / sort

map / forEach / filter / reduce

1-7. [스코프]

대부분의 C-family language는 블록 레벨 스코프(block-level scope)를 따름

```c
int main(void) {
  // block-level scope
  if (1) {
    int x = 5;
    printf("x = %d\n", x);
  }

  printf("x = %d\n", x); // use of undeclared identifier 'x'

  return 0;
}
```

코드 블록 ({...})내에서 유효한 스코프를 의미

자바스크립트는 함수 레벨 스코프(function-level scope)를 따름

함수 코드 블록 내에서 선언된 변수는 함수 코드 블록 내에서만 유효하고
함수 외부에서는 유효하지 않다(참조할 수 없다)는 것.

```javascript
var x = 0;
{
  var x = 1;
  console.log(x); // 1
}
console.log(x);   // 1

let y = 0;
{
  let y = 1;
  console.log(y); // 1
}
console.log(y);   // 0
```

🚨하지만 let 키워드 사용 : 블록 레벨 스코프 사용 가능

```javascript
var x = 'global';

function foo() {
  var x = 'local';
  console.log(x);
}

foo();          // local
console.log(x); // global
```

렉시컬 스코프 (static scoper)

함수를 어디서 호출하였는지가 아니라 **어디에 선언하였는지에 따라 결정되는 것**

함수의 선언 위치에 따라 상위 스코프를 결정 (정적 스코프)

- **Dynamic scope** : 런타임에 변수가 체크됨
- **Lexical scope** : 런타임을 참조하지 않고, 정적으로 블록별 스코프 체인을 따라 전역 스코프까지 올라가서 변수의 이름을 찾아냄.

컴파일 타임

개발언어로 작성된 소스코드가 compile 과정을 통해 컴퓨터가 이해할 수 있는 기계어로 변환되는 과정.

컴파일 동안 컴파일러는 코드의 syntax, semantic, type을 체크

<**컴파일타임** 에러>
-Syntax error
-Typpe check error

런타임

컴파일 과정을 마친 응용 프로그램이 사용자에 의해 실행되는 때를 의미

런타임 에러

컴파일이 완료되어 프로그램 실행 중에 컴파일 시에 감지하지 못한 에러들이 프로그램 실행 후에 발생되는 것.

<**런타임** 에러>
-0 나누기 오류
-null 참조 오류
-메모리 부족 오류
-index out of range