JavaScript

(교재: groomEdu)

# 웹을 이루는 요소

정보의 구조 : HTML  
꾸며주는 스타일 : CSS

사용자와의 상호작용(논리) : JS

Cross browsing issue => JS로 짠 웹과 브라우저가 호환이 안되는 경우

자바스크립트 = ECMAScript (모든 브라우저와 호환 가능)

ES1 -> ES2 -> … -> ES6

# 크롬 개발자 도구

1. Network

웹을 실행시킬 때 필요한 파일들을 보여줌

2. Console

-> console.log(); 콘솔창에 출력

-> variable : [var ~ = 데이터;]

3. Elements: 웹페이지 코드

-> 코드가 바뀔 경우의 변화를 보기 위해 주로 사용

4. Sources : JS debug 용도

\* watch탭 : 코드를 watch탭에 넣으면 그 값이 어떻게 실행되는지 보여줌

\* Breakpoints : 코드 줄 번호 클릭 -> n번 줄까지만 실행됨.

위 두개의 탭을 주로 사용

# 자료형 - 객체

JS는 OOP이다.

객체란 key: value의 집합임

구조

Let a ={

Key1 : value1,

Key2 : value2,

Method : function() {}

}

\* 객체 내 데이터 접근

1. 객체명.key값;

2. 메서드에 접근 시 객체명.method();

\* 객체에 key값 추가

1. 객체명.neyKey = newProperty;

2. 객체명[newKey] = newProperty;

\* 객체 key 삭제

delete 객체명.key;

\* key값 모아보기

1. Object.keys(객체명); => [key1, key2, … ,key]

2. for(let key in 객체명) { console.log(key); }

# 제어문

1. 조건문

If (조건) {…} else {...}

1. 반복문

For(초기화식; 조건식; 증감식) { … }

* Break, continue

1. do

# Fucntion

-함수의 입력값 : parameter / 결괏값: return

- 인자는 복수개여도 가능 / return 없을 수도!

\* 익명함수

=> 함수명이 없는 함수 (입력하자마자 바로 실행되는 것)

Ex> function () {}

\* 생성자 함수로 객체 만들기

비슷한 형태의 객체를 반복적으로 만들어야 할 때 생성자 함수로 객체를 만드는게 더 용이하다.

1. Function A (name, professor, … ) {

This.name = name;

This.printInfo = function () {…};

}

2. let class1 = new A (‘류건’, ‘소프트웨어학부’) -> 만든 생성자 함수로 객체 만들기

\* 객체로서 함수의 특징

1. 변수에 대입이 가능하다 (함수 = 1급 객체 취급)

Ex> var add = function addNum(a,b) {return a+b;} ==> add(1,2) = 3

2. 다른 함수의 인자나 리턴 값으로 활용 가능하다

ex> Console.log(“두 수의 합은”, add(1,2), “입니다”);

3. Array, Object 등의 자료구조에 대입이 가능하다

Ex>

Var sub = function(a,b) {return a-b;}

Let calc = [add, sub] → 함수가 인자로서 들어갈 수 있음!!

Calc[0](2,3) → 5

# scope 개념

1. 전역 변수(global variable) : 전역scope에서 선언된 변수

코드 전체를 아우르는 객체 ex> console.log(); , window객체

1. 지역 변수(local variable) : 지역scope에서 선언된 변수
2. Block scope : 중괄호로 표시된 범위 {...}
3. 함수 scope : function 안에서만 유효한 범위

# this

\* binding : 호출한 대상에게 실제 함수를 연결짓는 것 (올바른 객체에 올바른 메서드 묶어주기

Ex>

1. 일반함수 호출 시 this의 binding (전역객체 = this)

함수는 전역객체의 메서드, 전역변수 = 전역객체의 property

Var name = ‘ryu’ ==> print(name) = print(window.name) 같은 의미!

Var sayHello = function () {

Var name = ‘kim’;

Console.log(this.name); -> ryu 호출!

}

2. 객체 메서드에서의 this (메서드를 호출한 객체 = this)

3. 생성자 함수 호출 시 this의 바인딩 (생성자 함수로 생성된 객체 = this)

Var a = function (name) { => var boy = new a(‘ryu’);

This.name = name;

}

\* new로 새로운 객체를 만들면 생성자 함수,

(new 없이) 그냥 호출되면 일반 함수.

4. 내부함수에서의 this는 무조건 전역객체에 binding !

# DOM

JS에서 DOM객체에 접근할 때

1. ID값 기준

Var a = document.getElementById(‘ID값’)

원하는 ID값에 해당하는 DOM을 브라우저에서 보여줌

\*HTML 전체를 원한다면 => get~().innerHTML

\* Text만을 원한다면 => get~().innerText

Ex> ***element.innerText = "<div style='color:red'>A</div>";***

***element.innerHTML = "<div style='color:red'>A</div>";***

첫 번째 예시 -> <div style='color:red'>A</div> 출력

두 번째 예시 -> A 출력

2. TagName 기준

Var a = document.getElementsByTagName(‘tagName’);

3. className 기준

Var a = document.getElementsByClassName(‘className’);

4. Name 기준

var a = document.getElementsByName(‘name’);

\* HTML tag의 property 조작하기

Ex> <a href=”~” ~> 를 조작하고 싶다면

1. ID 설정

<a id=”A” ~></a>

1. 속성값을 보고 싶을 때

Var x = document.getElementById(‘A’);

x.getAttribute(“href”) ==> a의 링크가 나옴 -> getter

1. 어떤 속성에 어떤 값을 주고 싶을 때

x.setAttribute(“href”, http://~~~) -> setter

\* Input tag 다루기

<input name=’userid’> ==> text창이 뜸!

얘 접근 -> var a = document.getElementNyName;

A는 Nodelist type임!

<input name>에 문자 입력 시 a.value값에 할당됌.

\* 다중 input tag에 접속

Document.getElementsByTagName(“input”);

-> console.log()로 확인하면 [input, input, … ]의 Nodelist type임

Ex> id 입력 후 제출 버튼을 누를 시 브라우저에서 id값을 출력해주는 기능

Onclick기능 필요!

1. Function 구현

<input name=”ID값”>

<input name=”passwd값”>

Id값 = document.getElementsByTagName(‘input’)[0].value

passwd값 = document.getElementsByTagName(‘input’)[1].value

1. Input tag에 onclick 기능 구현

# Date function

Const day = new Date(); // 생성자 호출

Day.getHours(); -> 현재 시간 호출

Day.getMinutes(); -> 현재 분 호출

Day.getSeconds(); -> 현재 시간 초 호출

# callback 함수

-> setTimeout(대상, ms); // 몇 ms 뒤에 대상을 실행시킬건지 설정

JS는 각 함수의 순차실행 (synchronous: 동기처리)를 엄수하지 않음

실행순서를 보장하지 않으므로 되는 놈부터 처리함 (Asynchronous)

A B C 순서로 명령 (단, B가 오래 걸림)

-> A C B 로 실행!

그럼에도 실행 순서가 보장되어야 한다면 => callback 함수 사용!

먼저 실행될 함수에 나중에 실행될 함수를 인자로 넣어주고, 이후에 먼저 실행시킨 함수 다음 나중에 실행될 함수를 run

A가 2초가 걸리고 B가 즉시 실행된다> A->B순서로 실행시키고 싶다면

Function A (callback) {

SetTimeout( function() {

A의 내용;

Callback();

}, 2000);

}

A(B); 실행!

# var, let, const

Let => 중복선언 X, 값 재할당 0, 블록 스코프 변수

Const => 중복선언 X, 블록스코프 변수, 값 재할당 X (상수의 개념)

# template literal : 문자열을 백틱(`)으로 표현

(ES6에 도입된 문자열 표기법)

=> concat해야하는 문자열 or 객체가 많을 경우 번거롭다

Ex> “안녕 나는” + name + “이야” => `안녕 나는 ${name} 이야`

1. 백틱 안 문자열에서 연산 가능!

${10+5} 을 출력 시 15로 계산이 이루어짐.

2. 제어문을 문자열 안에 바로 사용 가능!

`아 숙제 ${homework ? ‘다했다’ : ‘안했다’}`

# 화살표 함수 (Arrow Function) : 익명 함수

-> 구조 : function ( (매개변수) => {함수 본문, 리턴 값}

매개변수 1개 -> 소괄호 생략 가능

본문이 1줄 -> 중괄호 생략가능 / 중괄호 생략 시 리턴 값도 생략(암묵적)

Return 삽입 시 여러 개 !

Ex>

화살표 함수 큰 장점 : callback함수 가독성↑

SetTimeout( () => ~, 3000)

# Array(배열)

-> JS의 배열은 배열을 흉내낸 객체에 불과하다

Const var1 = [1,2,3,4];

Typeof var1 => Object;

Var1.length; => 배열의 길이 [ 문자열 길이도 length]

\* JS에서는 배열의 요소 개수가 배열의 길이 이지만 배열의 길이가 곧 요소의 개수를 나타내진 않는다.

Ex>

\* 배열의 요소 추가

1. 배열을 직접 건드림 : ArrayName.push(요소)

2. 배열을 직접 건드리지 않음 : ArrayName.concat(요소)

\* 배열의 맨 마지막 요소 빼내기

ArrayName.pop();

\* 배열 속 요소 순회

Ex> a = [1,2,3]

1. for 문

For(var i=0; i<a.length; i++) {

Console.log(`a[i]`)

}

2. foreach => 새 배열 return x

a.foreach ( function (i) {

Console.log(i);

}

3. map 함수 [foreach보다 많이 사용] => 새 배열 return (foreach와 다른 점)

배열.map(function() {

함수 본문

})

Const b = a.map()

\* Array filter

# 구조분해할당(destructure)으로 값 빼내기

배열의 index or 객체의 key 기준으로 원하는 값 빼내기

\* 객체의 destructure

Const a = {name: ‘건’, age: 23, job: ‘student’}

Const {name, age, job} = a;

=> const name = ‘건’, const age = 23, … 과 같다!

\* 변수명 변경 시 const {name: student, age : number, job} = a;

-> student = ‘건’ , number = 23

\* 중첩된 객체 속 item 가져오기

\* spread 와 destructure

객체나 배열의 내부나 함수를 호출해주는 인자에서 사용

1. 객체

a라는 객체를 b 객체 안에 포함시키고 싶으면

Const b = {

...a

Major : ‘컴공’

}

2. 배열

A, b라는 배열을 c에 넣고 싶으면

C = [...a, ...b]

C 안의 숫자들을 모조리 더하는 sum 함수

Sum (num1, num2, …) {  
 return num1 + …

}

Const result = sum(...a);

\* rest 문법

Destructuring으로 값을 받아오는 객체나 배열을 받아오는 변수 또는 함수 파라미터에서 많이 쓰임

# class

구조) ex>

Class Name {

Constructor(parameter) {

This.parameter = parameter;

}

SayHello() {

Console.log(‘안녕’);

}

}

Const geon = new Name(‘건’)

\* 상속

부모class -> 자식class 에 정보를 공유!

Ex> person의 class내용을 다 가지면서 student class 만들기

class Student extends Person {

Constructor(name, school, major) {

Super(name);

This.school = school;

This.major = major;

}

}

※ 자식class에서 this 키워드를 사용하고 싶다면 반드시 부모class의 생성자 (super)가 먼저 호출되어야 함

# 모듈

파일 간에 주고 받을 수 있는 코드 조각, 코드 단위

클래스, 함수, 배열, 객체 등 다양하게 사용 가능

내보내는 것 = 모듈 (export)

Ex>

export sayHello;

Const sayHello = 함수; or export sayHello = 함수;

\* export default : 딱 하나의 모듈만을 내보내겠다고 명시

\* export 모듈 as 새이름 : 내보내는 모듈 이름 변경

\* import {모듈1, 모듈2, … } from ‘내보낸 파일’ : 모듈 가져오기

-> 이름 변경 : import {모듈1 as 새이름, 모듈2 as 새이름};

# 예외처리(try catch finally)

\* 정의된 에러들

1. SyntaxError - 문법에러

2. TypeError - 사용불가변수 사용시

3. ReferenceError - 허용되지 않은 레퍼런스 범위

4. RangeError - 허용숫자범위초과

5. EvalError – eval() 의 잘못된 활용

6. URIError – encodeURI() or decodeURI() 의 잘못된 활용

사용자 정의 에러 등등

\* 구조

Try {

일단 실행 부분

} catch(e) {

에러생기면 실행할 block

} finally {

끝나면 이거 무조건 실행

}

\* catch(e)

e.name = 에러 이름만 보고 싶다.

e.message = 에러 내용

Console.log(e) : 어떤 에러인지 콘솔에 출력

\* throw : 에러를 일부로 발생시킴

-> throw new Error(‘메세지’);

-> throw “메세지”;

# promise Async, Await

\* Promise : 비동기처리에 사용되는 객체

A = new Promise(function () {…}); [비동기처리]

A = new Promise( (resolve, reject) => {…})

Ex>

function a(name) {

return new Promise( (resolve, reject) => {

setTimeout( () => {

console.log(`내이름은 ${name}`);

resolve(); -> 잘 실행된 경우 실행!

}, 2000);

}

}

a.(‘건’).then(~) : 잘 실행되었다면 then(~) 실행

a.(‘건’).catch(~) : 잘 실행이 안되었다면 catch(~) 실행

\* async, await : 동기함수처럼 비동기함수 읽고 쓰기 (외부정보를 갖고올 때 많이 쓴다)

Ex>

Async function 이름(…) {

ㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡ

(교재: 모던JavaScript)

# 주석

1. // 주석입니다. (한 줄을 다 차지)

2. /\* ~

여러 줄의 주석 사용 가능

\*/

**중첩 주석은 지원하지 않습니다.**

/\*...\*/안에 또 다른 /\*...\*/이 있을 수 없습니다.

주석을 중첩해 쓰면 에러가 발생합니다.

# 엄격 모드

새롭게 제정된 ES5에서는 새로운 기능이 추가되고 기존 기능 중 일부가 변경되었습니다. 기존 기능을 변경하였기 때문에 하위 호환성 문제가 생길 수 있겠죠? 그래서 변경사항 대부분은 ES5의 기본 모드에선 활성화되지 않도록 설계되었습니다. 대신 use strict라는 특별한 지시자를 사용해 엄격 모드(strict mode)를 활성화 했을 때만 이 변경사항이 활성화되게 해놓았습니다.

지시자 "use strict", 혹은 'use strict'는 단순한 문자열처럼 생겼습니다. 하지만 이 지시자가 스크립트 최상단에 오면 스크립트 전체가 “모던한” 방식으로 동작합니다.

Ex>

"use strict";

// 이 코드는 모던한 방식으로 실행됩니다. ...

"use strict"는 스크립트 최상단이 아닌 함수 본문 맨 앞에 올 수도 있다는 점을 알아두시기 바랍니다. 이렇게 하면 오직 해당 함수만 엄격 모드로 실행됩니다.

# 변수와 상수

\* 변수

변수(variable)는 데이터를 저장할 때 쓰이는 ‘이름이 붙은 저장소’ 입니다. 온라인 쇼핑몰 애플리케이션을 구축하는 경우 상품이나 방문객 등의 정보를 저장할 때 변수를 사용하죠.

자바스크립트에선 let 키워드를 사용해 변수를 생성합니다.

let user = 'John',

age = 25,

message = 'Hello';

이렇게 정의하는 스타일도 존재함.

같은 변수를 여러 번 선언하면 에러가 발생합니다.

let message = "This";  
  
// 'let'을 반복하면 에러가 발생합니다.  
let message = "That";

원하는 만큼 값을 변경할 수도 있습니다.

let message;  
  
message = 'Hello!';  
  
message = 'World!'; // 값이 변경되었습니다.  
  
alert(message);

\* 변수 명명 규칙

자바스크립트에선 변수 명명 시 두 가지 제약 사항이 있습니다.

변수명에는 오직 문자와 숫자, 그리고 기호 $와 \_만 들어갈 수 있습니다.

첫 글자는 숫자가 될 수 없습니다.

예약어 사용 불가

\* 상수

변화하지 않는 변수를 선언할 땐, let 대신 const를 사용합니다.

상수는 재할당할 수 없으므로 상수를 변경하려고 하면 에러가 발생합니다.

\* 대문자 상수

기억하기 힘든 값을 변수에 할당해 별칭으로 사용하는 것은 널리 사용되는 관습입니다.

이런 상수는 대문자와 밑줄로 구성된 이름으로 명명합니다.

# 자료형

자바스크립트의 변수는 자료형에 관계없이 모든 데이터일 수 있습니다. 따라서 변수는 어떤 순간에 문자열일 수 있고 다른 순간엔 숫자가 될 수도 있습니다.

// no error  
let message = "hello";  
message = 123456;

이처럼 자료의 타입은 있지만 변수에 저장되는 값의 타입은 언제든지 바꿀 수 있는 언어를 ‘동적 타입(dynamically typed)’ 언어라고 부릅니다.

1. 숫자형

숫자형엔 일반적인 숫자 외에 Infinity, -Infinity, NaN같은 '특수 숫자 값(special numeric value)'이 포함됩니다.

Infinity는 어떤 숫자보다 더 큰 특수 값, 무한대(∞)를 나타냅니다.

어느 숫자든 0으로 나누면 무한대를 얻을 수 있습니다.

NaN은 계산 중에 에러가 발생했다는 것을 나타내주는 값입니다. 부정확하거나 정의되지 않은 수학 연산을 사용하면 계산 중에 에러가 발생하는데, 이때 NaN이 반환됩니다.

내부 표현 방식 때문에 자바스크립트에선 (2^53-1)(9007199254740991) 보다 큰 값 혹은 -(2^53-1) 보다 작은 정수는 '숫자형’을 사용해 나타낼 수 없습니다.

사실 대부분의 상황에서 이런 제약사항은 문제가 되지 않습니다. 그렇지만 암호 관련 작업같이 아주 큰 숫자가 필요한 상황이거나 아주 높은 정밀도로 작업을 해야 할 때는 이런 큰 숫자가 필요합니다.

BigInt형은 표준으로 채택된 지 얼마 안 된 자료형으로, 길이에 상관없이 정수를 나타낼 수 있습니다.

BigInt형 값은 정수 리터럴 끝에 n을 붙이면 만들 수 있습니다.

// 끝에 'n'이 붙으면 BigInt형 자료입니다.  
const bigInt = 1234567890123456789012345678901234567890n;

2. 문자형

역 따옴표(백틱)로 변수나 표현식을 감싼 후 ${…}안에 넣어주면, 아래와 같이 원하는 변수나 표현식을 문자열 중간에 손쉽게 넣을 수 있습니다.

let name = "John";  
  
// 변수를 문자열 중간에 삽입  
alert( `Hello, ${name}!` ); // Hello, John!  
  
// 표현식을 문자열 중간에 삽입  
alert( `the result is ${1 + 2}` ); // the result is 3

3. 불린형

true, false

4. null형

null 값은 지금까지 소개한 자료형 중 어느 자료형에도 속하지 않는 값입니다.

null 값은 오로지 null 값만 포함하는 별도의 자료형을 만듭니다.

let age = null;

자바스크립트의 null은 자바스크립트 이외 언어의 null과 성격이 다릅니다. 다른 언어에선 null을 '존재하지 않는 객체에 대한 참조’나 '널 포인터(null pointer)'를 나타낼 때 사용합니다.

하지만 자바스크립트에선 null을 ‘존재하지 않는(nothing)’ 값, ‘비어 있는(empty)’ 값, ‘알 수 없는(unknown)’ 값을 나타내는 데 사용합니다.

let age = null;은 나이(age)를 알 수 없거나 그 값이 비어있음을 보여줍니다.

5. undefined형

undefined는 '값이 할당되지 않은 상태’를 나타낼 때 사용합니다.

변수는 선언했지만, 값을 할당하지 않았다면 해당 변수에 undefined가 자동으로 할당됩니다.

let age;

alert(age); // 'undefined'가 출력됩니다.

하지만 이렇게 undefined를 직접 할당하는 걸 권장하진 않습니다. 변수가 ‘비어있거나’ ‘알 수 없는’ 상태라는 걸 나타내려면 null을 사용하세요. undefined는 값이 할당되지 않은 변수의 초기값을 위해 예약어로 남겨둡시다.

* Wrapper 객체 : 원시자료형을 유용한 메서드로 감싸는 객체
* Number, boolean, string wrapper

6. 객체

객체(object)형은 특수한 자료형입니다.

객체형을 제외한 다른 자료형은 문자열이든 숫자든 한 가지만 표현할 수 있기 때문에 원시(primitive) 자료형이라 부릅니다. 반면 객체는 데이터 컬렉션이나 복잡한 개체(entity)를 표현할 수 있습니다.

7. 심볼(symbol)

심볼형은 객체의 고유한 식별자(unique identifier)를 만들 때 사용됩니다.

8.typeof 연산자

typeof 연산자는 인수의 자료형을 반환합니다. 자료형에 따라 처리 방식을 다르게 하고 싶거나 변수의 자료형을 빠르게 알아내고자 할 때 유용합니다.

typeof 연산자는 두 가지 형태의 문법을 지원합니다.

1. 연산자: typeof x
2. 함수: typeof(x)

괄호가 있든 없든 결과가 동일합니다.

typeof x를 호출하면 인수의 자료형을 나타내는 문자열을 반환합니다.

# alert, prompt, confirm을 이용한 상호작용

메시지가 있는 작은 창은 모달 창(modal window) 이라고 부릅니다. '모달’이란 단어엔 페이지의 나머지 부분과 상호 작용이 불가능하다는 의미가 내포되어 있습니다.

1. alert

alert 함수는 앞선 예제에서 살펴본 바 있습니다. 이 함수가 실행되면 사용자가 ‘확인(OK)’ 버튼을 누를 때까지 모달 창이 계속 떠있게 됩니다.

alert("Hello");

2. prompt (python의 input같은 느낌)

브라우저에서 제공하는 prompt 함수는 두 개의 인수를 받습니다.

result = prompt(title, [default]);

함수가 실행되면 텍스트 메시지와 입력 필드(input field), 확인(OK) 및 취소(Cancel) 버튼이 있는 모달 창을 띄워줍니다.

title: 사용자에게 보여줄 문자열

default : 입력 필드의 초깃값(선택값)

3. confirm

confirm 함수는 매개변수로 받은 question(질문)과 확인 및 취소 버튼이 있는 모달 창을 보여줍니다.

사용자가 확인 버튼을 누르면 true, 그 외의 경우는 false를 반환합니다.

let isBoss = confirm("당신이 주인인가요?");

alert( isBoss ); // 확인 버튼을 눌렀다면 true가 출력됩니다.

# 형 변환

함수와 연산자에 전달되는 값은 대부분 적절한 자료형으로 자동 변환됩니다. 이런 과정을 "형 변환(type conversion)"이라고 합니다.

alert가 전달받은 값의 자료형과 관계없이 이를 문자열로 자동 변환하여 보여주는 것이나, 수학 관련 연산자가 전달받은 값을 숫자로 변환하는 경우가 형 변환의 대표적인 예시입니다.

\* 문자형으로의 변환

String(value)

|  |  |
| --- | --- |
| **전달받은 값** | **형 변환 후** |
| 0, null, undefined, NaN, "" | false |
| 그 외의 값 | true |

\* 숫자형으로 변환

숫자형으로의 변환은 수학과 관련된 함수와 표현식에서 자동으로 일어납니다.

숫자형이 아닌 값에 나누기 /를 적용한 경우

alert( "6" / "2" ); // 3으로 연산

Number(value) 함수를 사용하면 주어진 값(value)을 숫자형으로 명시해서 변환할 수 있습니다.

|  |  |
| --- | --- |
| undefined | NaN |
| null | 0 |
| true and false | 1 과 0 |
| string | 문자열의 처음과 끝 공백이 제거됩니다. 공백 제거 후 남아있는 문자열이 없다면 0, 그렇지 않다면 문자열에서 숫자를 읽습니다. 변환에 실패하면 NaN이 됩니다. |

# 기본 연산자와 수학

-> 파이썬과 매우 흡사

\* 거듭제곱근 : 4 \*\* (1/2) -> 2

\* 문자열과 정수를 더했을 때 concatenate되는 경우

alert(2 + 2 + '1' ); // '221'이 아니라 '41'이 출력됩니다.

\* 덧셈 이외의 나머지 연산자

alert( 6 - '2' ); // 4, '2'를 숫자로 바꾼 후 연산이 진행됩니다.

alert( '6' / '2' ); // 3, 두 피연산자가 숫자로 바뀐 후 연산이 진행됩니다.

\* 단항 연산자 +와 숫자형으로의 변환

덧셈 연산자 +는 이항 연산자뿐만 아니라 단항 연산자로도 사용할 수 있습니다.

숫자에 단항 덧셈 연산자를 붙이면 이 연산자는 아무런 동작도 하지 않습니다. 그러나 피연산자가 숫자가 아닌 경우엔 숫자형으로의 변환이 일어납니다.

문자열이 숫자형으로 변할 땐 문자열 앞뒤의 공백이 삭제됩니다.

예시:

// 숫자에는 아무런 영향을 미치지 않습니다.  
let x = 1;  
alert( +x ); // 1  
  
let y = -2;  
alert( +y ); // -2  
  
// 숫자형이 아닌 피연산자는 숫자형으로 변화합니다.  
alert( +true ); // 1  
alert( +"" ); // 0

단항 덧셈 연산자는 짧은 문법으로도 Number(...)와 동일한 일을 할 수 있게 해줍니다.

let apples = "2";

let oranges = "3"; // 이항 덧셈 연산자가 적용되기 전에, 두 피연산자는 숫자형으로 변화합니다.

alert( +apples + +oranges ); // 5

\* 증가 감소 연산자

++와-- 연산자는 변수 앞이나 뒤에 올 수 있습니다.

counter++와 같이 피연산자 뒤에 올 때는, '후위형(postfix form)'이라고 부릅니다.

++counter와 같이 피연산자 앞에 올 때는, '전위형(prefix form)'이라고 부릅니다.

값을 증가시키고 난 후, 증가한 값을 바로 사용하려면 전위형 증가 연산자를 사용하면 됩니다.

값을 증가시키지만, 증가 전의 기존값을 사용하려면 후위형 증가 연산자를 사용하면 됩니다.

\* 비트 연산자

비트 연산자(bitwise operator)는 인수를 32비트 정수로 변환하여 이진 연산을 수행합니다.

비트 AND ( & )

비트 OR ( | )

비트 XOR ( ^ )

비트 NOT ( ~ )

왼쪽 시프트(LEFT SHIFT) ( << )

오른쪽 시프트(RIGHT SHIFT) ( >> )

부호 없는 오른쪽 시프트(ZERO-FILL RIGHT SHIFT) ( >>> )

# 비교 연산자

A>b a<b a==b a!=b

\* 불린형 반환

다른 연산자와 마찬가지로 비교 연산자 역시 값을 반환합니다. 반환 값은 불린형입니다.

true가 반환되면, ‘긍정’, ‘참’, '사실’을 의미합니다.

false가 반환되면, ‘부정’, ‘거짓’, '사실이 아님’을 의미합니다.

\* 문자열 비교

자바스크립트는 ‘유니코드’ 순으로 문자열을 비교합니다. '사전편집(lexicographical)'순이라고 불리기도 하는 이 기준을 적용하면 사전 뒤쪽의 문자열은 사전 앞쪽의 문자열보다 크다고 판단됩니다.

alert( 'Z' > 'A' ); // true

alert( 'Glow' > 'Glee' ); // true

alert( 'Bee' > 'Be' ); // true

\* 다른 형을 가진 값 간의 비교

비교하려는 값의 자료형이 다르면 자바스크립트는 이 값들을 숫자형으로 바꿉니다.

예시:

alert( '2' > 1 ); // true, 문자열 '2'가 숫자 2로 변환된 후 비교가 진행됩니다.  
alert( '01' == 1 ); // true, 문자열 '01'이 숫자 1로 변환된 후 비교가 진행됩니다.

\* 일치 연산자 ===

일치 연산자는 엄격한(strict) 동등 연산자입니다. 자료형의 동등 여부까지 검사하기 때문에 피연산자 a와 b의 형이 다를 경우 a === b는 즉시 false를 반환합니다.

Ex>

Let a=0

Let b=”0”

A === b // false

\* null이나 undefined와 비교하기

null이나 undefined를 다른 값과 비교할 땐 예상치 않은 일들이 발생합니다. 일단 몇 가지 규칙을 먼저 살펴본 후, 어떤 예상치 않은 일들이 일어나는지 구체적인 예시를 통해 살펴보도록 하겠습니다.

1. 일치 연산자 ===를 사용하여 null과 undefined를 비교

두 값의 자료형이 다르기 때문에 일치 비교 시 거짓이 반환됩니다.

alert( null === undefined ); // false

2. 동등 연산자 ==를 사용하여 null과 undefined를 비교

동등 연산자를 사용해 null과 undefined를 비교하면 특별한 규칙이 적용돼 true가 반환됩니다. 동등 연산자는 null과 undefined를 '각별한 커플’처럼 취급합니다. 두 값은 자기들끼리는 잘 어울리지만 다른 값들과는 잘 어울리지 못합니다.

alert( null == undefined ); // true

3. 산술 연산자나 기타 비교 연산자 <, >, <=, >=를 사용하여 null과 undefined를 비교

null과 undefined는 숫자형으로 변환됩니다. null은 0, undefined는 NaN으로 변합니다.

\* null vs 0

alert( null > 0 ); // (1) false  
alert( null == 0 ); // (2) false  
alert( null >= 0 ); // (3) true

위 비교 결과는 논리에 맞지 않아 보입니다. (3)에서 null은 0보다 크거나 같다고 했기 때문에, (1)이나 (2) 중 하나는 참이어야 하는데 둘 다 거짓을 반환하고 있네요.

이런 결과가 나타나는 이유는 동등 연산자 ==와 기타 비교 연산자 <, >, <=, >=의 동작 방식이 다르기 때문입니다. (1)에서 null > 0이 거짓을, (3)에서 null >= 0이 참을 반환하는 이유는 (기타 비교 연산자의 동작 원리에 따라) null이 숫자형으로 변환돼 0이 되기 때문입니다.

그런데 동등 연산자 ==는 피연산자가 undefined나 null일 때 형 변환을 하지 않습니다. undefined와 null을 비교하는 경우에만 true를 반환하고, 그 이외의 경우(null이나 undefined를 다른 값과 비교할 때)는 무조건 false를 반환합니다. 이런 이유 때문에 (2)는 거짓을 반환합니다.

# if와 '?'를 사용한 조건 처리

If(boolean) {

~

} else if {

~

} else {

~

}

\* 삼항 연산자

If (조건 ? true일때 리턴값 : false일때 리턴 값)

Ex>

let accessAllowed = (age > 18) ? true : false;

\* 다중 ‘?’

물음표 연산자?를 여러 개 연결하면 복수의 조건을 처리할 수 있습니다.

예시:

let age = prompt('나이를 입력해주세요.', 18);  
  
let message = (age < 3) ? '아기야 안녕?' :  
 (age < 18) ? '안녕!' :  
 (age < 100) ? '환영합니다!' :  
 '나이가 아주 많으시거나, 나이가 아닌 값을 입력 하셨군요!';  
  
alert( message );

-> age >= 3일 때 2번째 줄 실행

-> age >= 18일 때 3번째 줄 실행

-> age >= 100일 때 4번째 줄 실행

같은 예시 :

if (age < 3) {

message = '아기야 안녕?';

} else if (age < 18) {

message = '안녕!';

} else if (age < 100) {

message = '환영합니다!';

} else {

message = '나이가 아주 많으시거나, 나이가 아닌 값을 입력 하셨군요!';

}

새로 알게된 점

SetInterval 함수 -> 함수를 반복시킴.

SetTimeout 함수 -> 함수를 몇 초 뒤에 실행시킴.

# 논리 연산자

AND (&&)

OR (||)

NOT (!)

\* JS에서만 특별하게 기능하는 OR 연산자

1. OR "||" 연산자를 여러 개 체이닝(chaining) 하면 첫 번째 truthy를 반환합니다. 피연산자에 truthy가 하나도 없다면 마지막 피연산자를 반환합니다.

Let result = value1 || value2 || value3;

2. OR(||) 은 왼쪽부터 시작해서 오른쪽으로 평가를 진행하는데, truthy를 만나면 나머지 값들은 건드리지 않은 채 평가를 멈춥니다. 이런 프로세스를 '단락 평가’라고 합니다.

단락 평가의 동작 방식은 두 번째 피연산자가 변수 할당과 같은 부수적인 효과(side effect)를 가지는 표현식 일 때 명확히 볼 수 있습니다.

true || alert("not printed"); -> alert함수가 실행되지 않음

false || alert("printed"); -> alert함수가 실행됌.

단락 평가의 동작 방식은 두 번째 피연산자가 변수 할당과 같은 부수적인 효과(side effect)를 가지는 표현식 일 때 명확히 볼 수 있습니다.

\* AND 연산자

AND 연산자는 첫 번째 falsy를 반환합니다. 피연산자에 falsy가 없다면 마지막 값을 반환합니다.

Ex>

// 첫 번째 피연산자가 truthy이면,

// AND는 두 번째 피연산자를 반환합니다.

alert( 1 && 0 ); // 0 alert( 1 && 5 ); // 5

// 첫 번째 피연산자가 falsy이면,

// AND는 첫 번째 피연산자를 반환하고, 두 번째 피연산자는 무시합니다.

alert( null && 5 ); // null

alert( 0 && "아무거나 와도 상관없습니다." ); // 0

* **|| -> && -> !이 순으로 우선순위가 높다**

\* NOT 연산자

result = !value; 의 형태로 사용

-> NOT을 두 개 연달아 사용(!!)하면 값을 불린형으로 변환할 수 있다.

Result = !!true; // true

-> alert를 호출하면 undefined를 반환함!

\* Nullish 병합 연산자

nullish 병합 연산자(nullish coalescing operator) ??를 사용하면 짧은 문법으로 여러 피연산자 중 그 값이 ‘확정되어있는’ 변수를 찾을 수 있습니다.

a ?? b의 평가 결과는 다음과 같습니다.

a가 null도 아니고 undefined도 아니면 a

그 외의 경우는 b

nullish 병합 연산자 ??없이 x = a ?? b와 동일한 동작을 하는 코드를 작성하면 다음과 같습니다.

x = (a !== null && a !== undefined) ? a : b;

\* 연산자 우선순위

??의 연산자 우선순위는 5로 꽤 낮음

할당연산자 = 조건연산자 ? 과 같은 대부분의 연산자보다는 나중에 평가됌.

안정성 이슈로 인해 ??는 &&나 ||와 함께 사용하지 못함.

# for / while iteration

1. while

본문이 한 줄이면 중괄호를 안 써도 됨.

let i = 3;

while (i) alert(i--);

2. do-while 반복문

Do {

// 반복문 본문

} while(condition);

While과는 다르게 본문이 먼저 실행되고 나서 조건 여부를 따진다.

-> 본문을 최소한 한 번이라도 실행하고 싶을 때 사용!

3. for iteration

For(begin; condition; step) {

본문

}

Ex>

For(let i=0; i<3; i++) {

Alert(i);

}

\* for문의 element도 생략 가능

Let i=0;

For(; i<3; i++) {

I -> 0 1 2

}

\* Step도 생략 가능!

For(; i<3;)

= while(i<3)과 동일함

\* 전부 생략!

For(;;){

무한반복 됨

}

* **‘?’ 오른쪽엔 break나 continue가 올 수 없다**

\* break/ continue label

outer: for (let i = 0; i < 3; i++) {

for (let j = 0; j < 3; j++) {

let input = prompt(`(${i},${j})의 값`, '');

// 사용자가 아무것도 입력하지 않거나 Cancel 버튼을 누르면 두 반복문 모두를 빠져나옵니다.

if (!input) break outer; // (\*)

-> Label을 별도의 줄에 써도 가능함

Outer:

For (let i=0; i<3; ,,, ) {

…

}

# switch