JavaScript

- BITCAMP

한다은

(https://github.com/RayJUNRein/Programming.git)

◆ 자료형 – 숫자형 (number)

✓ parseInt 와 parseFloat

- 1) parseInt(str, radix) : 정수 반환 / radix : 원하는 진수 지정
- 2) parseFloat : 부동 소수점 숫자 반환

✓ 특수 숫자 값 (special numeric value)

- Infinity : 어떤 숫자보다 큰 특수 값, 무한대 / <-> -Infinity
- NaN : 계산 중에 에러가 발생했다는 것을 나타내주는 값

✓ Math 라이브러리

- Math.random() : 0 ~ 1 미만의 소수 반환
- Math.max(인수) : 인수 중 최대값 반환
- Math.min(인수) : 인수 중 최솟값 반환
- Math.floor() : 소수점 자릿수 버림

◆ 자료형 – 문자열 (string)

✓ 대, 소문자 변경

- 1) toLowerCase() : 대문자를 소문자로 변경
- 2) toUpperCase() : 소문자를 대문자로 변경

◆ 템플릿 리터럴 (Template Literals)

✓ **백틱 (backtick)**

- `문자열` 과 같이 사용
- 1) 별도 처리 없이 줄바꿈 가능 (일반적으론 '₩n', "₩n")
- 2) 결합 연산자 없이 문자열 출력 가능 = 문자열 보간 (String Interpolation)

ex) str = first + " * " + second + " = ???"

ex) str = `${first} * ${second} = ???`

◆ 비교 연산자

✓ == 동등 연산자 (Equality Operator) : 값만 비교

- ex) if (0 == false) -> true

✓ === 일치 연산자 (Strict Equality Operator) : 값 + 자료형 까지 비교

- ex) if (0 === false) -> false

◆ ... 전개 연산자 (Spread Operator)

- ES6부터 추가된 기능
- 배열의 원소들을 분해해서 개별요소로 만듦

✓ **배열 병합 (Array Concatenation)**

- ex) var arr = [0, ...arr1, 3, 4, ...arr2];

✓ **배열 복사 (Copying Arrays)**

- 참조 : 원본 내용도 변경 / 복사 : 원본 내용 변경 없음
- 얕은(Shallow) 복사 수행

: 원본 배열 내 객체 수정 시 새로운 배열 내 객체 값도 변경

◆ if 문

✓ 문자열이 조건인 경우

- true : 비어있지 않은 문자열 / false : 비어있는 문자열
- ex) if("test") -> true

✓ 객체가 조건인 경우

- 모두 true : 빈 객체라도 객체 자체가 존재한다고 생각해 무조건 true 반환

✓ 함수가 조건인 경우

- 모두 true : 무조건 true 반환

✓ true = 1, false = 0 인식 가능

- 자바 스크립트는 true/false 를 1/0 으로 표기 가능
- ex) if(!(i % 2)) = if(i % 2 == 0)
- ex) if(0 == false) -> true

◆ 함수 (function)

✓ 화살표 함수

- function Name(name) { } 은 const Name = (name) => { } 와 같음

✓ func.apply(context, args)

- func의 this를 context로 고정해주고, 유사 배열 객체인 args를 인수로 사용할 수 있게 해줌

◆ 배열 (Array)

✓ 배열의 함수

- arr.push(원소) : 배열 끝에 원소 추가하고, 배열의 길이 반환
- arr.pop() : 배열 끝 원소를 제거하고, 제거한 원소 반환
- arr.unshift(원소) : 배열 앞에 원소 추가하고, 배열의 길이 반환
- arr.shift() : 배열 앞 원소를 제거하고, 제거한 원소 반환
- arr.splice(index, deleteCount, elem1, ..., elemN) : index 부터 deleteCount 만큼 지우고, elemN 들로 교체, 지운 원소들 반환
- arr.slice(start, end) : start부터 (end 제외)end 인덱스까지 원소 복사한 새로운 배열 반환 (기존 배열 그대로)
- arr.concat(원소1, 원소2) : 기존 배열과 원소들에 속한 모든 요소들을 포함한 새로운 배열 반환
- arr.forEach(function(item, index, array) { }) : 배열 원소 각각에 대해 함수 실행
- arr.indexOf(원소) : 원하는 항목이 몇번째 원소인지 찾아줌 (없으면 -1 반환)
- arr.lastIndexOf(item, from) : 원하는 항목이 몇번째 원소인지 끝에서부터 찾아줌
- arr.find (function(item, index, array) { }) : 함수의 반환값을 true로 만드는 요소 찾으면 탐색 중단 + 해당 요소 반환 (없으면 undefined)
- arr.findIndex() : 함수의 반환값을 true로 만드는 요소 찾으면 탐색 중단 + 해당 요소의 인덱스 반환 (없으면 -1)

◆ 배열 (Array)

✓ 배열의 함수

- arr.filter(function(item, index, array) { }) : 함수의 반환 값을 true로 만드는 요소들을 찾음 + 결과 배열 반환
- arr.map(function(item, index, array) { }) : 배열 요소 각각에 대해 함수 실행 + 결과 배열 반환
- arr.sort(fn) : 배열의 요소를 문자형으로 변환한 이후 정렬하고 재정렬된 배열 반환

  * fn 정렬 함수 (ordering function) : 정렬 기준을 정의해주는 함수
- arr.reverse() : 배열의 요소를 역순으로 정렬하고 재정렬된 배열 반환
- arr.join(인수) : 배열 요소들 사이에 '인수'를 넣어 모두 하나의 문자열로 만듦
- arr.reduce(function(accumulator, item, index, array) { }, [initial]) : 배열 요소 각각에 대해 함수 실행 + 이전 함수 호출 결과를 인수로 사용 + 결과값 반환
- arr.every(fn) : 배열 요소 각각에 대해 함수 실행 + 모든 요소가 함수의 반환 값을 true로 만드는지 여부 반환
- arr.some(fn) : 배열 요소 각각에 대해 함수 실행 + 함수의 반환 값을 true로 만드는 요소가 하나라도 있는지 여부 반환

◆ 객체

✓ in 연산자

: "key" in object / 프로퍼티 존재 여부 확인

✓ hasOwnProperty

: obj.hasOwnProperty(key) / key에 대응하는 프로퍼티가 상속 프로퍼티가 아니고 obj에 직접 구현되어 있는 프로퍼티일 때 true 반환

◆ 반복 가능한 (이터러블, iterable) 객체

: 배열을 일반화한 객체 / 이 개념 사용하면 어떤 객체든 for … of 반복문 적용 가능 EX) 배열, 문자열

✓ Symbol.iterator (특수 내장 심볼)

- 1) for … of 시작되면 Symbol.iterator 호출 -> iterator (매서드 next 보유한 객체) 반환
- 2) for … of는 반환된 객체만을 대상으로 동작
- 3) 다음 값으로는 iterator 의 매서드 next() 호출하여 넘어감 -> next()는 {done: Boolean, value:any}형태 반환

◆ 맵 (Map)

: Key가 있는 데이터 저장 / Key에 다양한 자료형 허용 (객체와 차이)

✓ 매서드

- new Map() : 맵 생성
- map.set(key, value) : key를 이용해 value 저장
- map.get(key) : key에 해당하는 값 반환 / key 없으면 undefined
- map.has(key) : key가 존재하면 true, 없으면 false 반환
- map.delete(key) : key에 해당하는 값 삭제
- map.clear() : 맵 안의 모든 요소 제거
- map.size : 요소의 개수 반환

✓ 반복 작업

- for (let key of map.keys()) { } / for (let value of map.values()) { } / for (let entry of map.entries()) { }
- map.keys() : 각 요소의 key를 모은 반복 가능한(이터러블, iterable) 객체 반환
- map.values() : 각 요소의 값을 모은 iterable 객체 반환
- map.entries() : 요소의 [key, value] 를 한 쌍으로 하는 iterable 객체 반환
- map.forEach ((value, key, map) => { }) : 각 [key, value] 쌍을 대상으로 반복 작업

◆ 셋 (Set)

: 중복 허용 X

✓ 매서드

- new Set(iterable) : 셋 생성 / iterable 객체의 값을 복사해 넣어줌
- set.add(value) : 값 추가 + 셋 자신 반환
- set.has(value) : 셋 내에 값이 존재하면 true, 아니면 false 반

✓ 반복 작업

- for (let value of set) { } : 셋의 값을 대상으로 반복 작업 수행
- set.forEach((value, valueAgain, set) => { }) : 셋의 값을 대상으로 반복 작업 수행 / 같은 인자 2번 받는 이유는 맵과의 호환성 때문
- set.keys() : 셋 내의 모든 값 포함하는 iterable 객체 반환
- set.entries() : 셋 내의 각 값을 이용해 만든 [value, value] 배열 포함하는 iterable 객체 반환 / 맵과의 호환성 위함

◆ 클래스

✓ 생성자 constructor()

: 객체의 기본 상태를 설정해주는 생성자 매서드

✓ getter와 setter

- 1) getter : 프로퍼티를 읽으려고 할 때 실행
- 2) setter : 프로퍼티에 값을 할당하려 할 때 실행

✓ 정적(static) 매서드

: 매서드를 프로퍼티 형태로 직접 할당하는 것과 동일 (특정 클래스 인스턴스가 아닌 '전체' 에 필요한 기능을 만들 때 사용)

◆ JSON (JavaScript Object Notation)

: 값이나 객체를 나타내주는 범용 포맷, RFC 4627 표준에 정의되어 있음

- 데이터 교환 목적으로 사용하는 경우 많음 (특히 클라이언트 측 언어가 JS일 때)

✓ JSON.stringify

: 객체를 JSON으로 바꿔줌

✓ JSON.parse

: JSON을 객체로 바꿔줌

◆ 인터페이스 기능

✓ alert

- 사용자가 '확인(OK)' 버튼을 누를 때까지 메시지를 보여주는 창이 계속 떠있게 됨
- 모달 창 (modal window) : 페이지의 나머지 부분과 상호 작용이 불가능

◆ 호출 스케일링 (scheduling a call)

: 일정 시간이 지난 후 원하는 함수를 예약 실행(호출)할 수 있게 함

✓ setTimeout

: let timerId = setTimeout (func|code, [delay], [arg1], [arg2], …)

- func|code : 실행하고자 하는 코드 (함수 or 문자열)
- delay : 실행 전 대기 시간 (ms)
- arg1, arg2 : 함수에 전달할 인수들

◆ 제너레이터 (generator)

: 여러 개의 값을 필요에 따라 하나씩 반환(yeield) <-> 일반 함수는 0 or 1개의 값만 반환

- function* name() {
    yield 1;
    yield 2;
    return 3; }

✓ 동작 과정

- 1) 제너레이터 함수 호출하면 코드 실행 X / 실행 처리하는 제너레이터 객체 반환
- 2) next() 호출 -> 다음 yield <value>문 만날 때까지 실행 지속
- 3) 다음 yield <value>문 만나면 실행 멈추고 value 반환

◆ 프라미스 (promise)

- let promise = new Promise (function (resolve, reject) { })
- function (resolve, reject) {} : 실행 함수 / 프라미스가 만들어질 때 자동 실행

✓ resolve / reject

- 자바스크립트가 자체적으로 제공하는 콜백, 반드시 하나 호출해야 함
- resolve(value) : 성공한 경우 / 결과 나타내는 value와 함께 호출
- reject(error) : 에러 발생한 경우 / 에러 객체 나타내는 error와 함께 호출

✓ promise 객체 내부 프로퍼티

- state : 초기값 pending(보류) -> resolve 호출) fulfilled / reject 호출) rejected
- result : 초기값 undefined -> resolve(value) 호출) value / reject(error) 호출) error

◆ 소비 함수 등록에 사용하는 매서드

- 소비 함수 : 결과나 에러를 받음

✓ then

- promise.then (function (result) { }, function (error) { })
- function(result) {} : 프라미스가 이행되었을 때 실행되는 함수 / 실행 결과 받음
- function(error) {} : 프라미스가 거부되었을 때 실행되는 함수 / 에러 받음

✓ catch

- promise.catch (errorHandlingFunction) = .then (null, errorHandlingFunction)
- 에러 발생한 경우만 다룸

✓ finally

- promise.finally (function)
- 프라미스가 처리되면 항상 실행됨
- .then (function, function) 과의 차이 : 1) 인수받지 않음 (프라미스 성공, 실패 여부 모름)
  2) 자동으로 다음 핸들러에 결과, 에러 전달

◆ 프라미스 체이닝 (promise chaining)

- new Promise --- resolve -→ .then --- result -→ .then --- result -→ .then
- promise.then을 호출하면 프라미스 반환되므로 .then 호출 가능

◆ 에러 핸들링 (error handling)

✓ throw

- throw new Error( ) = reject (new Error ( ))
- 예외가 발생하면 암시적 try…catch에서 자동으로 에러를 잡고 reject처럼 다룸

◆ 프라미스 API

✓ Promise.all

- let promise = Promise.all ([…promises…])
- 요소 전체가 프라미스인 배열 받아 프라미스 모두 처리되면 새로운 프라미스 반환
- 새 프라미스의 result = 프라미스의 결과값 담은 배열 (순서 = 전달된 순서)

✓ Promise.race

- let promise = Promise.race (iterable)
- Promise.all 과 비슷하지만, 가장 먼저 처리되는 프라미스의 결과(or 에러)를 반환

◆ async와 await

✓ async 함수

- async function f( ) { return 1; }
- 항상 프라미스를 반환

✓ await

- let value = await promise
- async 함수 내에서만 동작 / await 키워드 만나면 프라미스가 처리될 때까지 기다렸다가 결과 반환