2025.01.07.수

생성일	@2026년 1월 7일 오전 9:04
태그	

목차

▼ 1) JSON

- js 내장 함수
  - JSON.stringify(): 객체를 JSON 문자열로 변환
  - JSON.parse(): JSON 문자열을 객체로 변환
- 문자열, 숫자, 불리언, 객체, 배열, null 데이터 타입 지원
- undefined, 함수, Date, Symbol 데이터 타입 지원 안함.

▼ 2) Symbol

- 고유하고 변경 불가능한 원시 값
- 객체의 고유한 프로퍼티 키를 만들 때 사용
- 고유성, 불변성 특성을 지님
- 같은 Symbol 값을 반복 생성해도 서로 다르게 취급 (메모리 주소가 다름)
- 은닉 프로퍼티로 외부에서 쉽게 접근할 수 없음.
- 생성 방법

```
const sym1 = Symbol();
const sym2 = Symbol("description");
const sym3 = Symbol("description");
console.log(sym1 === sym2);
console.log(sym2 === sym3);
```

- 객체의 프로퍼티 키로 사용

```
const sym = Symbol('uniqueKey');
const obj = {
 [sym]: 'value'
};
console.log(obj[sym]); // "value" 출력

for (let key in obj) {
 console.log(key); // 출력되지 않음 (Symbol 프로퍼티는 열거되지 않음)
}
console.log(Object.keys(obj)); // 빈 배열 출력 (Symbol 키는 Object.keys로 접근 불가)
```

- 접근 방법 - 대괄호 표기법

```
const sym = Symbol('mySymbol');
const obj = {
 [sym]: 'value'
};
console.log(obj[sym]); // "value" 출력
```

- 접근 방법 - Object.getOwnPropertySymbols() 메서드 이용

```
const sym1 = Symbol('symbol1');
const sym2 = Symbol('symbol2');
const obj = {
 [sym1]: 'value1',
 [sym2]: 'value2'
};
const symbols = Object.getOwnPropertySymbols(obj);
console.log(symbols); // [Symbol(symbol1), Symbol(symbol2)]
console.log(obj[symbols[0]]); // "value1" 출력
console.log(obj[symbols[1]]); // "value2" 출력
```

- 접근 방법 - Reflect.ownKeys() 메서드 사용

```
const sym = Symbol(`mySymbol`);
const obj = {
 [sym]: `value`
 normalKey: `normalValue`
};
const keys = Reflect.ownKeys(obj);
console.log(keys);
console.log(obj[keys[0]]); // "normalValue" 출력
console.log(obj[keys[1]]); // "value" 출력
```

▼ 3) 이터러블과 제너레이터

- Set
  - add(value)
  - delete(value)
  - has(value)
  - clear()
  - size
- Map
  - set(key, value)
  - get(key)
  - delete(key)
  - has(key)
  - clear()
  - size
- 이터러블: 객체의 각 요소를 순차적으로 접근할 수 있는 객체
- 제너레이터: 이터레이터를 쉽게 생성하는 함수
  - function* 키워드 사용
  - yield 키워드를 통해 값을 하나씩 반환하고 멈춤/재개 가능
    - 값이 있다면 value, done: false, 값이 없다면 undefined, done:true

▼ 4) 옵셔널 체이닝 (Optional Chaining)

- 객체가 null 또는 undefined 인 경우데 오류를 발생시키지 않고 undefined를 반환
- 객체, 배열의 속성, 메서드에 접근할 때 오류를 방지한 안전한 접근 가능
- 객체 뒤에 ? 키워드 붙여 사용

```
const user = {
 name: `Alice`,
 address: {
 city: `Wonderland`
 }
};
console.log(user?.address?.city); // "Wonderland" 출력
console.log(user?.contact?.email); // undefined 출력, 오류 발생하지 않음
```

- null 병합 연산자
  - null, undefined 값을 받게 되었을 때 정해진 값을 반환
  - 삼항연산자와 비슷

```
const user = null;
const name = user?.name ?? `Anonymous`;
console.log(name); // "Anonymous"
```

▼ 5) 렉시컬과 클로저

- 렉시컬 스코프
  - 함수가 정의된 위치에 따라 상위 스코프가 결정되는 것
  - 지역 스코프의 변수가 전역 스코프 변수보다 우선 순위가 높다.

```
const a = 1;
const b = 1;
const c = 1;

function funcA(){
   const b = 2;
   const c = 2;
   console.log("2", a, b, c); // "2", 1, 2, 2
   funcB()
}

function funcB() {
   const c = 3;
   console.log("3", c, b, c); // "3", 1, 2, 3
}

console.log("1", a, b, c); // "1", 1, 1, 1
funcA();
```

- 클로저
  - 함수와 그 함수가 선언된 렉시컬 환경의 조합
  - 외부 함수가 종료되어도 내부 함수가 외부 함수의 변수에 접근할 수 있는 현상
  - 변수를 은닉하고 상태를 기억, 캡슐화 구현

```
function outerFunction() {
 let outerVariable = "I am outside!";
 function innerFunction() {
 console.log(outerVariable); // outerVariable에 접근 가능
 }
```

```
 return innerFunction;
}
const myClosure = outerFunction();
myClosure(); // "I am outside!" 출력
```

```
function createCounter() {
 let count = 0;
 return function() {
 count++;
 return count;
 }
}
const counter = createCounter();
console.log(counter()); // 1
console.log(counter()); // 2
console.log(counter()); // 3
```

▼ 6) this

- 함수가 실행되는 컨텍스트 참조, 호출 방식에 따라 달라짐
- 동작 방식
  - 전역 컨텍스트

```
console.log(this); // 전역 객체 출력, 브라우저 환경에선 windwos 객체 출력
```

  - 일반 함수 호출

```
function showThis() {
    console.log(this)
}
showThis(); // 전역 객체
```

  - 메서드 호출

```
const obj = {
    name: 'Alice'
    showThis: function() {
        console.log(this.name);
    }
};

obj.showThis(); // "Alice"
```

  - 생성자 함수 호출

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

const person = new Person('Alice');
console.log(person.name); // "Alice"
```

  - 클래스 선언

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
```

```
    }

    greet() {
        console.log("Hello, my name is $(this.name) and I am $(this.age) years old"
    }
}

const alice = new Person('Alice', 30);
alice.greet();

const bob = new Person('Bob', 25);
bob.greet();
```

- 동적 바인드 this
  - 호출되는 시점에 this가 결정되는 방식

```
function showThis() {
 console.log(this.name);
}
const obj1 = { name: 'Object 1' };
const obj2 = { name: 'Object 2' };
showThis.call(obj1); // Object 1
showThis.call(obj2); // Object 2
const boundShowThis = showThis.bind(obj1);
boundShowThis(); // "Object 1"
```

▼ 7) 프로토타입

- 자바스크립트의 상속 메커니즘
- 객체 지향 프로그래밍을 클래스가 아닌 프로토타입을 통해 구현
- 모든 객체는 자신을 생성한 생성자 함수의 프로토타입을 참조하는 속성을 가지고 있어서 프로토타입 객체로부터 메서드와 속성을 상속받을 수 있음 → 프로토타입 체인
- 생성자 함수와 클래스는 모두 .prototype 속성을 사용하여 공통된 메서드와 속성을 정의할 수 있음

▼ 8) 비동기 프로그래밍

- 자바스크립트는 기본적으로 싱글 스레드지만 비동기 처리 가능
- 비동기 처리를 통해 Non-blocking I/O 구현
- callback: 비동기로 실행할 함수
- setTimeout: 일정 시간 후에 지정된 함수 실행
- Promise: 비동기 작업의 완료/실패를 나타내는 객체
  - Pending: 대기
  - Fulfilled: 이행됨
  - Rejected: 거부됨
  - resolve: 완료시 콜백
  - reject: 실패시 콜백
  - then: 이행된 값을 인자로 받아 콜백
  - catch: 거부된 이유(에러)를 인자로 받아 콜백
  - finally: 무조건 실행되는 콜백

```
const myPromise = new Promise((resolve, reject) => {
  const success = true; // 비동기 작업의 성공 여부를 결정
  if (success) {
    resolve('The operation was successful!');
```

```
  } else {
    reject('The operation failed.');
  }
});

myPromise.then(result => {
  console.log(result); // "The operation was successful!" 출력
});

myPromise.catch(error => {
  console.error(error); // "The operation failed." 출력
});

myPromise.finally(() => {
  console.log('Promise has been settled (fulfilled or rejected).');
});
```

- Promise.all: 전달된 모든 Promise가 이행될 때 까지 기다리다가 결과를 배열로 반화, 하나라도 실패하면 실패 이유(에러)를 반환하고 나머지 이행 값 무시

```
function taskA() {
  return new Promise(resolve => {
    setTimeout(() => resolve('Task A Completed'), 1000);
  });
}

function taskB() {
  return new Promise(resolve => {
    setTimeout(() => resolve('Task B Completed'), 1000);
  });
}

function taskC() {
  return new Promise(resolve => {
    setTimeout(() => resolve('Task C Completed'), 1000);
  });
}

Promise.all([taskA(), taskB(), taskC()])
  .then(results => {
    console.log('All tasks completed:', results);
  }
  .catch(error => {
    console.log('One of the task failed:', error);
  })
```

- Promise.allSettled: 전달된 모든 Promise의 작업 생태 + 이행 결과 값을 성공 실패 상관없이 반환
  - .status: 상태
  - .value: 결과
  - .reason: 실패 이유

```
function taskA() {
 return new Promise(resolve => {
 setTimeout(() => resolve('Task A completed'), 1000);
 });
}
function taskB() {
 return new Promise((resolve, reject) => {
```

```
 setTimeout(() => reject('Task B failed'), 2000);
 });
}
function taskC() {
 return new Promise(resolve => {
 setTimeout(() => resolve('Task C completed'), 3000);
 });
}
Promise.allSettled([taskA(), taskB(), taskC()])
 .then(results => {
 console.log('All tasks completed:');
 results.forEach(result => console.log(result.status, result.value || result.reason));
```

- Promise.race: 가장 먼저 이행된 Promise 결과 반환 (성공 실패 상관 없음)
- async / await
  - async: Promise 반환
  - await: async 내부에서 사용되며 Promise가 끝날 때까지 함수 실행을 대기 시킴 → 비동기 작업을 동기 처럼 동작시킴

```
function fetchData() {
   return new Promise(resolve => {
      setTimeout(() => {
            resolve("Data fetched!");
         }, 2000);
      });
   }

   async function processData() {
      console.log("Fetching data...");
      const data = await fetchData();
      console.log(data)
   }

   processData();
```