EliceAcademy

JavaScript : 4주차

자료구조와 에러처리

목차

자바스크립트에서 자료를 처리하는 여러 방식에 대해 이해할 수 있다.

자바스크립트 최신 문법에 대해 사용할 수 있다.

자바스크립트로 에러처리 할 수 있다.

01

ES6 최신 문법들

EliceAcademy

ES5 & ES6

버전	특징	출시 연도
ES1	...	1997
ES2	...	1998
ES3	정규 표현식, try - catch	1999
`ES5`	`HTML5와 함께 출현한 표준안`, `JSON`, `strict mode`, `접근자 프로퍼티`, `프로퍼티 어트리뷰트 제어`, `향상된 배열 조작 기능(forEach, map, filter, reduce, some, every)`	2009
`ES6(ECMAScript 2015)`	`let/const`, `클래스`, `화살표 함수`, `템플릿 리터럴`, `디스트럭처링 할당`, `스프레드 문법`, `rest 파라미터`, `심벌`, `프로미스`, `Map/Set`, `이터러블`, `for - of`, `제너레이터`, `Proxy`, `모듈 import/export`	2015
ES7(ECMAScript 2016)	지수(**)연산자, Array.prototype.includes, String.prototype.includes	2016
ES8(ECMAScript 2017)	async/await, Object 정적 메서드 (Object.values, Object.entriesm Object.getOwnPropertyDescriptors)	2017

Symbol

ES6에 도입된 7번째 데이터 타입
변경 불가능한 원시 타입의 값
다른 값과 중복되지 않은 유일무이한 값
⇒ 이름 충돌 위험이 없는 프로퍼티 키 값을 만들기 위해 주로 사용

다른 원시 타입과 다르게 리터럴 표기법이 없다
⇒ 생성된 심벌 값은 외부로 노출되지 않음
⇒ 무조건 Symbol() 를 호출하여 생성 생성자 함수가 아니라서 new 안 붙여야함
⇒ 다른 값과 절대 중복 되지 않는 유일무이한 값
⇒Symbol를 프로퍼키 키값을 사용, 접근할 때 대괄호 [] 사용

Symbol 장점

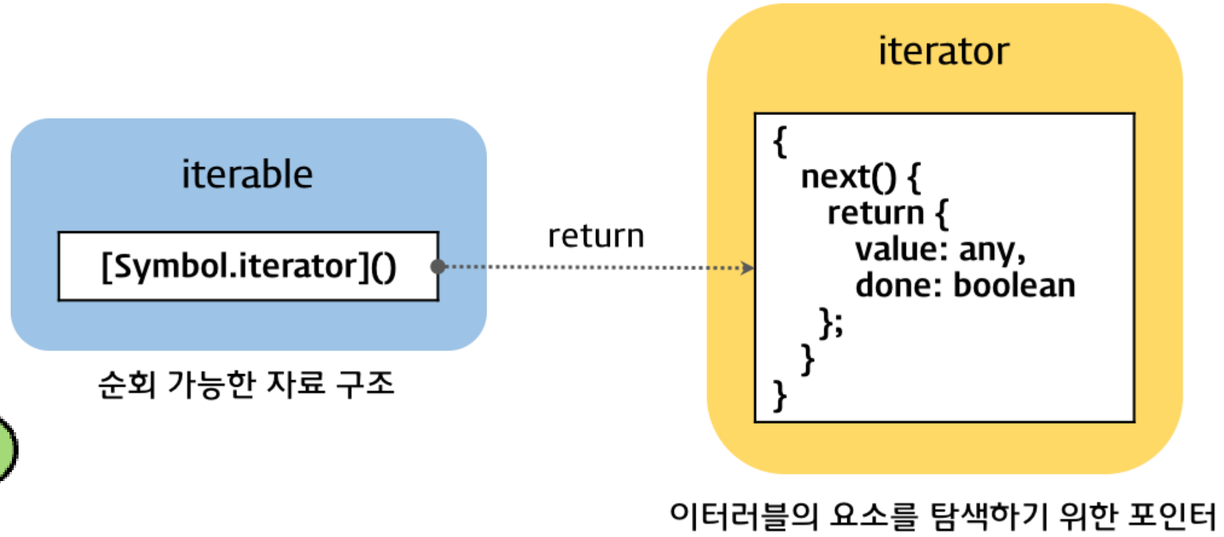
단순 문자열로 만든 키 값은 접근이 쉬움
특별한 상수 값이 대체불가 할 수 있게 유일무이한 값으로 사용 가능

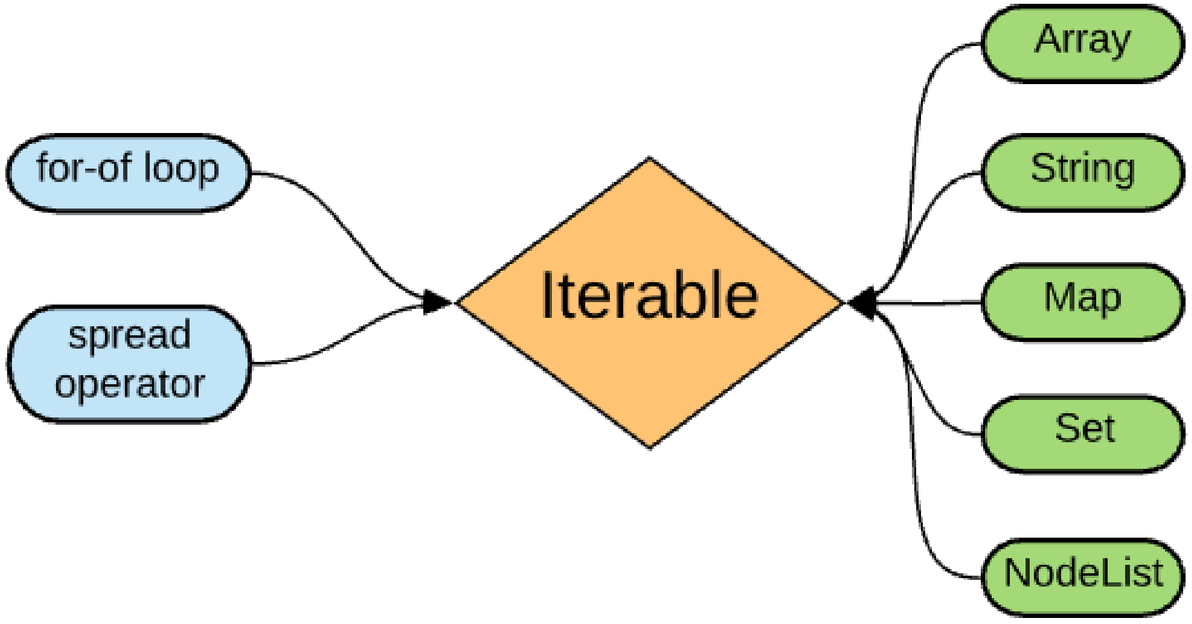
프로퍼티로 심벌 값을 사용하게 되면 다른 프로퍼티 키와 충돌 없음
기존 프로퍼티 혹은 미래 추가될 프로퍼티 키와 충돌 되지 않음 => 호환성 보장
심벌 값을 프로퍼티 키로 사용하여 생성한 프로퍼티는 외부로 노출이 안됨 => 은닉

⇒ Object.keys, Object.getOwnPropertyNames 메서드 : 심볼 속성 나오지 않음
⇒Object.getOwnPropetySymbols 메서드: 심볼 속성만 나옴
⇒ Reflect.ownKey() 메서드 : 심볼 포함 모든 객체 메서드 출력

Iteration 이터레이션: 반복, 순회

이터레이션 프로토콜 (**Iteration protocols**)
프로토콜 : 규격, 약속, 인터페이스





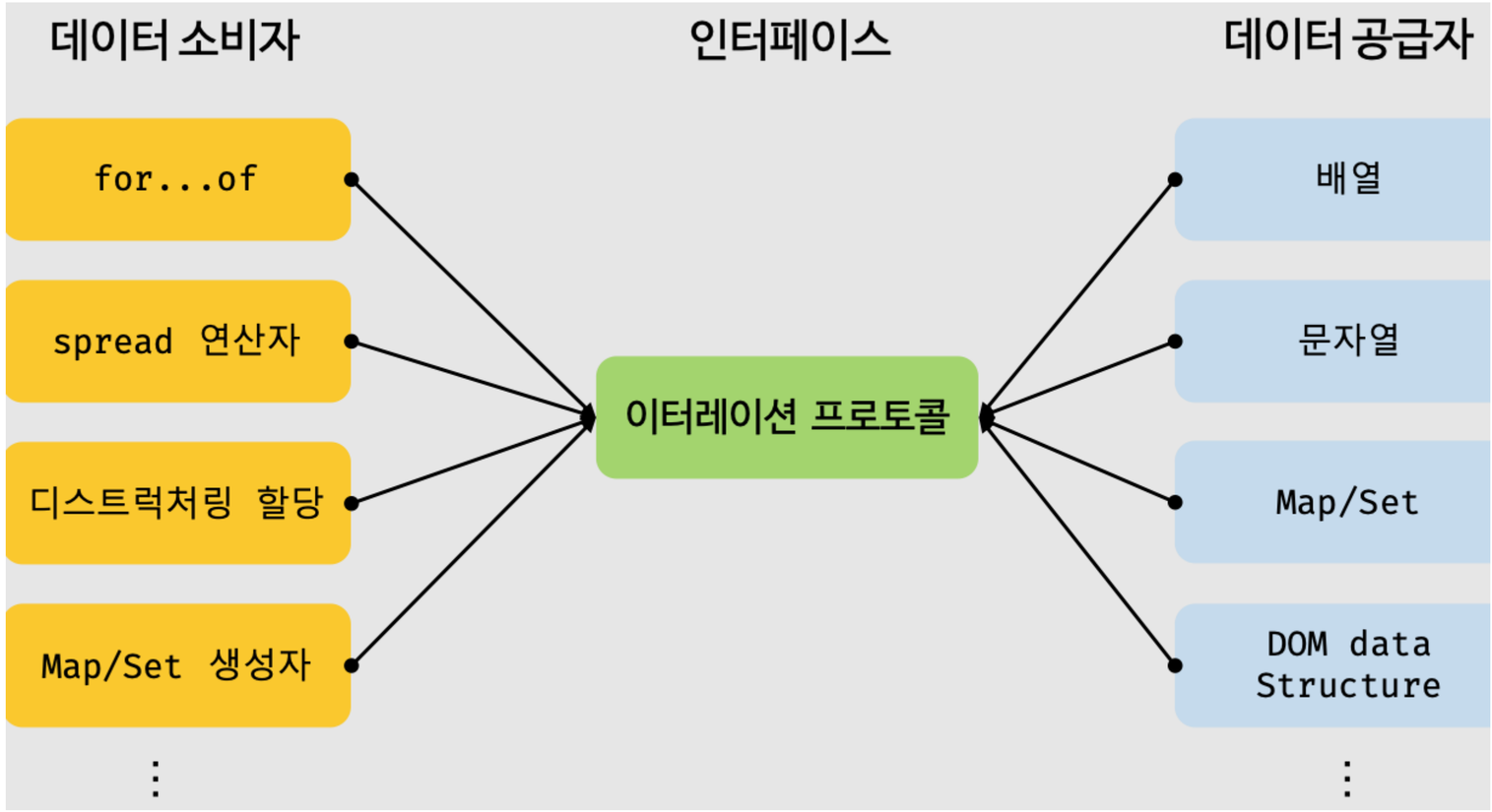
iterable은 반복 가능한 객체를 의미
iterable로 평가된 값은
for ... of , spread 연산자, 구조분해 할당 사용가능

1. 객체에 **[Symbol.iterator]** 메서드가 존재
2.[Symbol.iterator]은 Iterator 객체를 반환해야한다

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

이터레이션 프로토콜 필요성

다양한 데이터 타입들을
같은 방식으로 통합하여
접근하기 위해서



이터레이션 프로토콜 준수한 객체 -> 이터러블 객체

```
const evenMaker = {
    [Symbol.iterator]: () => {
        const max = 5;
        let num = 0;
        return {
            next() {
                return { value: num++ * 2, done: num > max };
            },
        };
    },
};

for (const num of evenMaker) {
    console.log(num);
}
```

[Symbol.iterator] : () 메서드를 구현
return 값으로 next 메서드를 갖는
iterator 객체를 반환
next 메서드는 done 과 value
프로퍼티를 가지는
iterator result 객체를 반환

next() => 다음 인자값 호출

제너레이터(generator)

코드 블록의 실행을 일시 중지 했다가 필요

한 시점에 재개 할 수 있는 특수한 함수

제너레이터 함수 정의

function* 키워드로 선언 , 하나이상의

yield 표현식 포함

제너레이터 함수는

화살표 함수로 정의 불가

생성자 함수로 호출 불가

```
// 함수 선언문으로 제너레이터 함수 정의
function* generatorFunc() {
    yield 1;
}

// 함수 표현식으로 제너레이터 함수 정의
const generatorFunc = function* () {
    yield 1;
};

// 메서드로 제너레이터 함수 정의
const obj = {
    *generatorFunc() {
        yield 1;
    },
};

// 클래스의 메서드로 제너레이터 함수 정의
class MyClass {
    *generatorFunc() {
        yield 1;
    }
}
```

일반 함수 vs 제너레이터 함수

일반 함수

1. 호출 시 제어권은 호출된 함수에게 있다 => 함수 코드 블록 일괄 실행
2. 함수 실행 시 단방향으로 함수 내부 값 전달 => 매개변수를 통해서
3. 코드 전체 실행 뒤 return 해준 값 한 개 반환 후 함수 코드블록 삭제

제너레이터 함수

1.함수 호출한 곳에 제너레이터 함수 일시 중시하거나 종료하는 제어권을 양도

2. 함수 실행 시 함수 호출한 공간과 양방향으로 함수 상태 주고 받을 수 있음

    함수 호출한 공간에 상태 전달하고, 전달 받을 수 있음

3. 함수 호출하면 제너레이터 객체를 반환

    함수 호출 시 함수 코드 실행이 아니라 이터레이터인 제너레이터 객체의 상태를 업데이트를 하면서 함수 실행

EliceAcademy

제너레이터 함수

next 메서드 호출하면
yield 표현식 까지 실행되고 일시중지

next 메서드는 iteratorYieldResult
객체 반환

value는 yield 키워드 다음에 있는 값
done은 함수가 종료되었는지 확인

next() 하게되면 중지된 시점부터
다시 실행

return() 하면 바로 함수 종료한다

```
function* generatorFunc() {
    yield 1;
    yield 2;
    yield 3;
}

// 제너레이터 호출 -> 제너레이터 객체 반환
const generator = generatorFunc();

console.log(generator); // Object [Generator] {}
console.log(Symbol.iterator in generator); // true
console.log('next' in generator); // true

console.log(generator.next()); // { value: 1, done: false }
console.log(generator.return()); // { value: undefined, done: true }
```

```
interface IteratorYieldResult<TYield> {
    done?: false;
    value: TYield;
}
```

spread 연산자 : 전개구문

하나로 뭉쳐 있는 여러 데이터들의 집합을 개별적인 값으로 분류

Iterable 객체 Array, String, Map, Set , DOM nodelist , arguments 와 객체

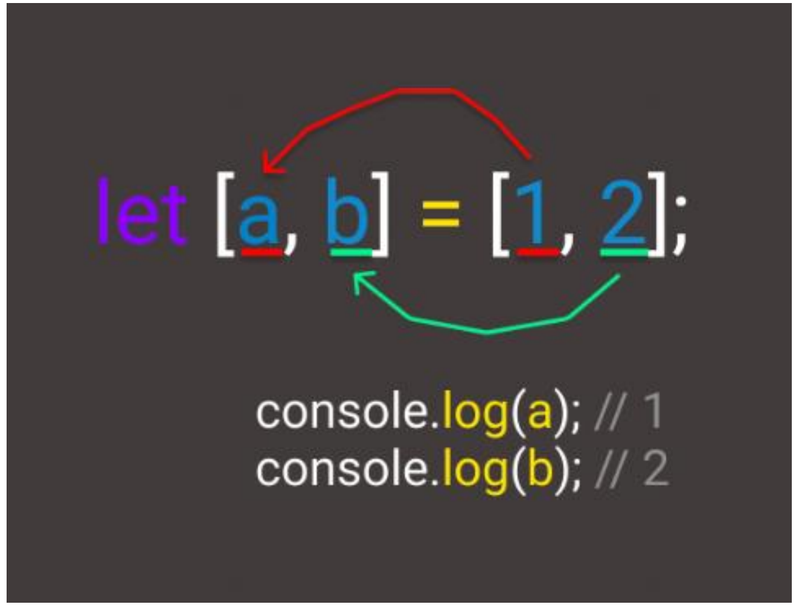
개별적인 값들의 목록을 반환이지 값이 아님 => 스프레드 문법의 결과를 변수에 할당 불가

,(쉼표) 로 구분한 값의 목록을 사용하는 문백 에서만 사용가능

1. 함수의 호출문의 인수 목록
2. 배열 리터럴의 요소 목록
3. 객체 리터럴의 프로퍼티 목록

구조분해 할당 (destructuring assignment)

이터러블 혹은 객체를 구조 분해하여 1개이상의 변수에 개별적으로 순서대로 할당
데이터 그룹화 쉽게 할 수 있다.



보통 배열과 객체에 많이 사용한다

배열 구조분해 할당은 변수이름[인덱스번호] 로 값 대신 원하는 식별자로 할당

객체에서 원하는 프로퍼티 값만 추출해서 변수 할당

배열과 객체의 구조분해 할당을 할 때 변수의 기본값 설정
rest 요소 사용 가능

참고
https://velog.io/@gil0127/%EA%B5%AC%EC%A1%B0-%EB%B6%84%ED%95%B4-%ED%95%A0%EB%8B%B9-%EA%B0%9D%EC%B2%B4-%EB%B0%B0%EC%97%B4

EliceAcademy

논리 연산자 단축 평가(short-circuit evaluation)

논리 연산의 결과를 결정하는 피연산자를 타입 변환하지 않고 그대로 반환
표현식을 평가하는 도중에 평가결과가 확정 → 나머지 평가 과정을 생략 특성이용

논리곱(&&) = 논리 연산의 결과를 결정하는 것은 두 번째 피연산자
논리합(||) = 논리 연산의 결과를 결정하는 것은 첫 번째 피연산자

```
// 논리합(||) 연산
"Cat" || "Dog"; // "Cat"
false || "Dog"; // "Dog"
"Cat" || false; // "Cat"
```

```
// 논리곱(&&) 연산
"Cat" && "Dog"; // "Dog"
false && "Dog"; // "false"
"Cat" && false; // "false"
```

단축 평가 표현식	평가 결과
true ll anything	true
false ll anything	anything
true && anything	anything
false && anything	false

옵셔널 체이닝 연산자 (ES11)

표기법 => ?.

객체를 가리키기를 기대하는 변수가 null 또는 undefined가 아닌지 확인하고 프로퍼티를 안전하게 참조할 때 유용

null 또는 undefined 만 false 로 인식

옵셔널 체이닝 도입 이전에는
논리곱(&&)을 사용한 단축 평가를 통해
→ 변수가 null 또는 undefined 인지 확인했음

```
// 논리곱 연산자 사용
let str = '';
let length = str && str.length;
console.log(length); // ''
// 옵셔널 체이닝 사용
str = '';
length = str?.length;
console.log(length);  // 0
```

널 병합 연산자 (nullish coalescing ES11)

표기법 => **??**

오직 null 과 undefined 값만 false 한 값으로 인식

기존 논리 연산자 단축 평가

0, -0, '' => false 한 값으로 인식

0 이나 빈 문자열을 유효한 값으로 인식하고 싶을 때 사용

0, -0, '' => true 한 값으로 인식

```
let num = 0;
console.log(num || '-1'); // -1
console.log(num ?? '-1'); // 0
```

02

Collection

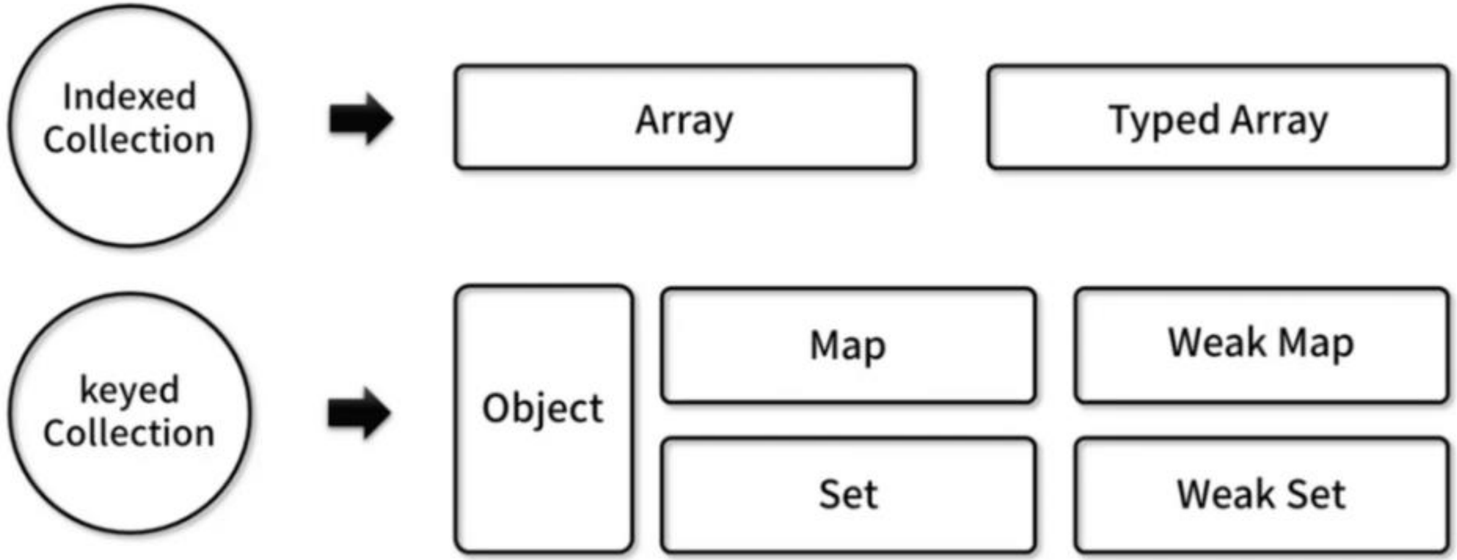
Collection

프로그래밍 언어가 많은 데이터를 담을 수 잇는 공간

JavaScript 에서 제공하는 Collection 종류

indexed Collection
인덱스로 값 접근
순서가 있는 자료구조

Keyed Collection
키로 값 접근
순서가 없는 자료구조



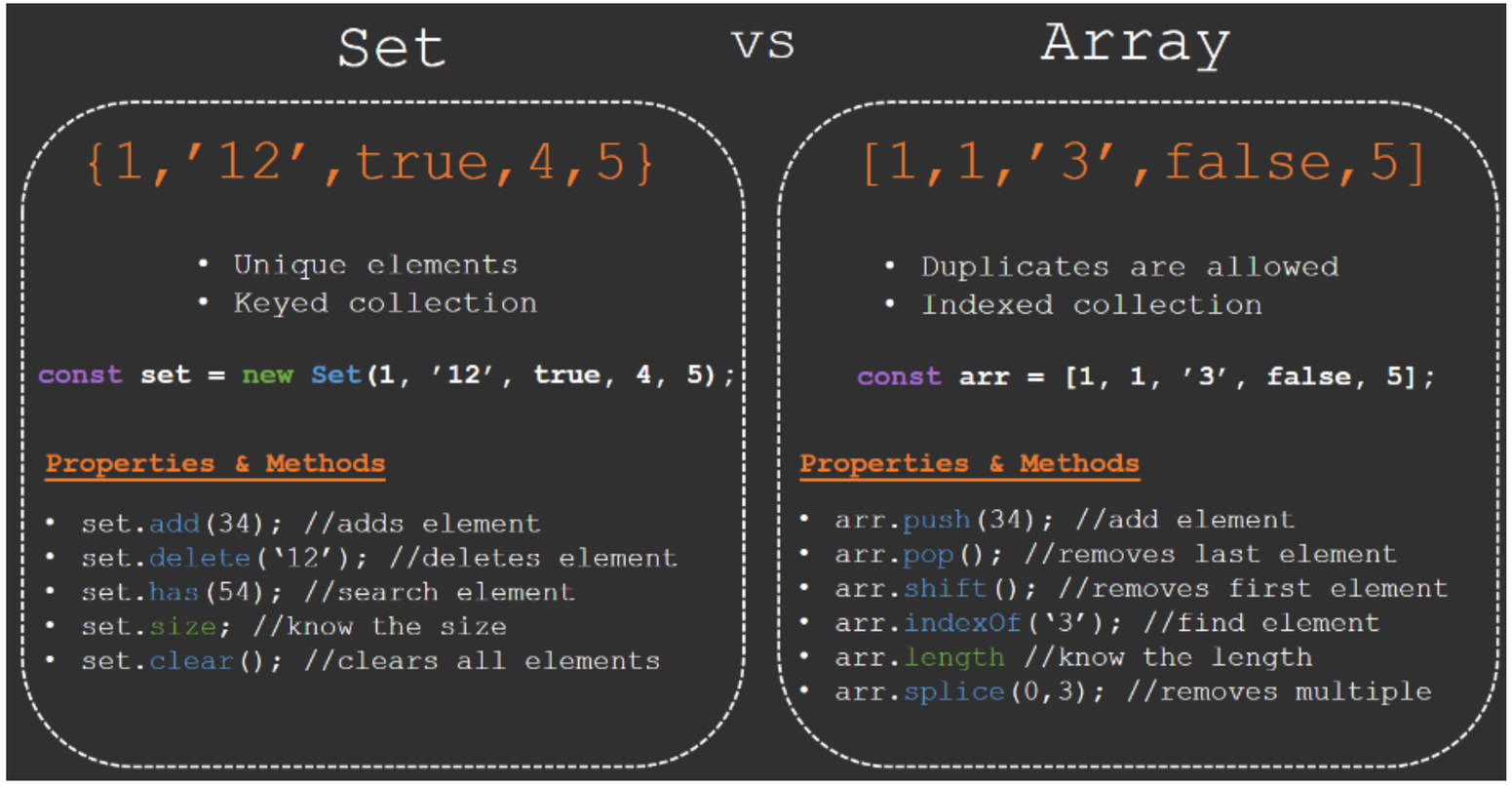
EliceAcademy

Set 과 Array 차이

중복 되지 않은 유일한 값들의 집합

set은 보통 수학적 집합을
구현하기 위한 자료구조

구분	배열	Set 객체
동일한 값의 중복을 허용	O	X
요소 순서에 의미	O	X
인덱스로 요소에 접근	O	X



참고
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set

EliceAcademy

set 과 array 함께 쓰기

Set to array

```
const arr = [...set];
```

Unique values from array

```
const arr = [1, 1, 3, 7, 8, 3];
const unique = new Set(arr);
```

Union of two Sets

A ∪ B


```
const union = new Set([...A, ...B]);
```

Difference of two Sets

A \ B


```
const difference = new Set([...A].filter
(elem => !B.has(elem)));
```

Intersection of two Sets

A ∩ B


```
const intersection = new Set([...A].filter
(elem => B.has(elem)));
```

Superset


```
const isSuperset = (set, subset) => {
  for (let elem of subset) {
    if (!set.has(elem)) {
      return false;
    }
  }
  return true;
}
```

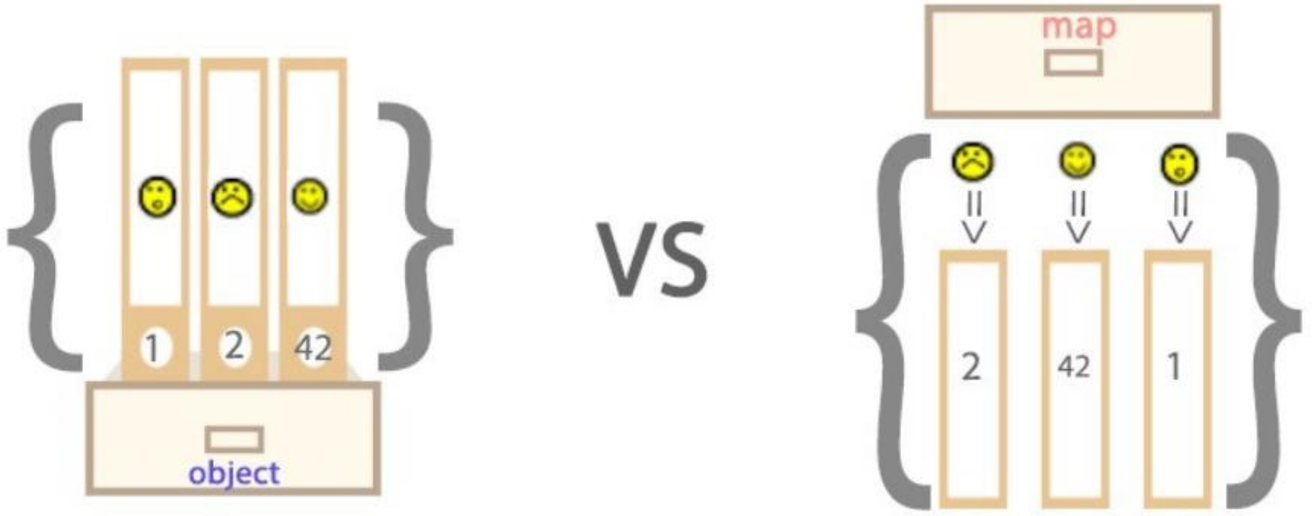
Object 와 Map 비교

키와 값의 쌍으로 이루어진 컬렉션 객체

Object 객체타입과 유사하지만
차이점이 있어서 구별에 주의!

실행 시까지 키를 알 수 없고
모든 키가 동일한 type
모든 값이 동일한 type 이면
object 대신 map 사용 추천

각 개별 요소가 각기 다르면 객체 사용



구분	객체	Map 객체
키로 사용할 수 있는 값	문자열 or 심벌 값	객체를 포함한 모든 값
이터러블	X	O
요소 개수 확인	Object.keys(obj).length	Map.prototype.size

EliceAcademy

03

에러

에러처리의 필요성

에러가 발생하지 않는 코드를 작성하는 것은 불가능
발생한 에러에 대해 대응하지 않고 방치하면 프로그램이 즉시 강제 종료

```
console.log('시작');

foo();
// ReferenceError: foo is not defined
// 강제 종료

console.log('끝'); // 실행 못함
```

try – catch

미리 에러가 날 수 있는 상황에 대처해서 강제 종료되는 것을 방지
언제나 에러나 예외적인 상황이 발생할 수 있다는 것을 전체하고 이에 대응하는 코드를 작성하는 것이 중요

```
try {
    foo();
} catch (err) {
    console.error(`에러 발생 : ${err}`);
}

console.log('끝');

// 시작
// 에러 발생 : ReferenceError: foo is not defined
// 끝
```

try - catch - finally 문

에러 처리를 구현하는 두가지 방법

1. 예외적인 상황이 발생하면 반환하는 값(null 또는 -1 같은)을
   조건문이나 단축 평가, 옵셔널 체이닝 연산자를 통해 처리
2. 에러 처리 코드를 미리 등록해 두고 에러가 발생하면 에러 처리 코드로 이동 처리

```
try {
    // 실행할 코드(에러 발생 여지가 있는 코드)
} catch (err) {
    // try 블록에서 에러가 발생하면 이 블록의 코드가 실행
    // 매개변수(이름 자유)에는 try 코드 블록에서 발생한 Error 객체가 전달
} finally {
    // 에러 발생과 상관없이 반드시 한 번 실행
}
```

Error 객체

Error 생성자 함수는 에러 객체를 생성
인수로 해당 에러에 대한 설명을 의미하는 에러 메시지(문자열) 전달
const error = new Error("Error !");
Error 객체는 3가지 프로퍼티를 갖는다->
Error.prototype 을 상속받음
name –> 에러 객체 이름
message
Error 생성자 함수 호출 시 인수로 전달한 에러 메시지 값
stack
에러를 발생시킨 콜 스택(= 실행 컨텍스트 스택)의 호출 정보를 나타내는 문자열

```
interface Error {
    name: string;
    message: string;
    stack?: string;
}

interface ErrorConstructor {
    new(message?: string): Error;
    (message?: string): Error;
    readonly prototype: Error;
}
```

Error 객체 종류

자바스크립트에선 Error 생성자 함수를 포함, 총 7가지 에러 객체를 생성할 수 있는 생성자 함수 제공

생성자 함수	인스턴스
Error	일반적 에러 객체
SyntaxError	문법에 맞지 않는 문을 해석할 때 발생하는 에러 객체
ReferenceError	참조할 수 없는 식별자를 참조할 경우 발생하는 에러 객체
TypeError	피연산자 or 인수의 데이터 타입이 유효하지 않을 때 발생하는 에러 객체
RangeError	숫자값의 허용 범위를 벗어났을 때 발생하는 에러 객체
URIError	encodeURI 또는 decodeURI 함수에 부적절한 인수를 전달했을 때 발생하는 에러 객체
EvalError	eval 함수에서 발생하는 에러 객체

https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Statements/try...catch

throw 문

에러를 발생시키는 문

Error 생성자 함수로 에러 객체를 생성한다고 에러가 발생하는 것은 아니다
즉, 에러 객체 생성과 에러 발생은 별개로 이뤄짐
에러를 발생시키려면 try 코드 블록에서 throw 문으로 에러 객체를 던지면
catch 문의 에러 변수가 생성되고 => 던져진 에러 객체가 할당 후 catch 문 실행

에러의 전파 (Propagating)

에러는 호출자(caller) 방향으로 전파된다
콜 스택 아래 방향으로 전파

throw 된 에러를 catch 하지 않으면 호출자로 에러 전파

이 에러를 적절히 캐치하여 대응하면 프로그램 강제 종료를 방지하고, 코드의 실행 흐름을 복구 가능

throw 된 에러를 어디에서도 캐치하지 않으면 프로그램은 강제 종료

EliceAcademy

에러의 전파 (Propagating)

```
const foo = () => {
    throw new Error('foo 함수에서 발생한 에러'); // 4
};

const bar = () => {
    foo(); // 3
};

const baz = () => {
    bar(); // 2
};

try {
    baz(); // 1
} catch (err) {
    console.error(err);
}
```