


* 예습
5/7(1h38m)
* 복습&예습
5/11 ~p.16, p.24~끝
* 복습
5/12(1h)

21. 빌트인 객체

빌트인 전역 프로퍼티 예시들

TABLE OF CONTENTS

1. 자바스크립트 객체의 분류
2. 표준 빌트인 객체 표준 빌트인 객체가 제공하는 메소드는?
3. 원시 값과 래퍼 객체 뜻/래퍼 객체 생성 과정 설명
4. 전역 객체 전역 객체의 종류
 - 4.1 빌트인 전역 프로퍼티
 - 4.1.1. Infinity
 - 4.1.2. NaN
 - 4.1.3. undefined
 - 4.2. 빌트인 전역 함수
 - 4.2.1. eval 사용 금지. 왜? 스코프를 교란시키고 ~
 - 4.2.2. isFinite
 - 4.2.3. isNaN
 - 4.2.4. parseFloat
 -  4.2.5. parseInt -> 제일 많이 사용할 것
 - 4.2.6. encodeURI / decodeURI
 - 4.2.7. encodeURIComponent / decodeURIComponent
 - 4.3. 암묵적 전역언제 발생하는지?

1. 자바스크립트 객체의 분류

자바스크립트 객체는 아래와 같이 크게 3개의 객체로 분류할 수 있다.

- 표준 빌트인 객체** ECMAScript
 표준 빌트인 객체(standard built-in objects / native objects / global objects)는 ECMAScript 사양에 정의된 객체를 말하며 애플리케이션 전역의 공통 기능을 제공한다. 표준 빌트인 객체는 ECMAScript 사양에 정의된 객체이므로 자바스크립트 실행 환경(브라우저 또는 Node.js 환경)과 관계없이 언제나 사용할 수 있다. 표준 빌트인 객체는 전역 객체의 프로퍼티로서 제공된다. 따라서 별도의 선언없이 전역 변수처럼 언제나 참조할 수 있다.
- 호스트 객체**
 호스트 객체(host objects)는 ECMAScript 사양에 정의되어 있지 않지만 자바스크립트 실행 환경(브라우저 환경 또는 Node.js 환경. “3.1 자바스크립트 실행 환경” 참고)에서 추가적으로 제공하는 객체를 말한다.
클라이언트에서 서버로 데이터 전송을 요청하는 것
 브라우저 환경에서는 DOM, BOM, Canvas, XMLHttpRequest, fetch, requestAnimationFrame, SVG, Web Storage, Web Component, Web worker와 같은 클라이언트 사이드 Web API를 호스트 객체로 제공하고 Node.js 환경에서는 Node.js 고유의 API를 호스트 객체로 제공한다.
- 사용자 정의 객체**
 사용자 정의 객체(user-defined objects)는 표준 빌트인 객체와 호스트 객체처럼 기본 제공되는 객체가 아닌 사용자가 직접 정의한 객체를 말한다.

2. 표준 빌트인 객체

자바스크립트는 Object, String, Number, Boolean, Symbol, Date, Math, RegExp, Array, Map/Set, WeakMap/WeakSet, Function, Promise, Reflect, Proxy, JSON, Error 등 40여개의 표준 빌트인 객체를 제공한다.

Math, Reflect, JSON을 제외한 표준 빌트인 객체는 모두 인스턴스를 생성할 수 있는 생성자 함수 객체이다. 생성자 함수 객체인 표준 빌트인 객체는 프로토타입 메소드와 정적 메소드를 제공하고 생성자 함수 객체가 아닌 표준 빌트인 객체는 정적 메소드만을 제공한다.

[주의] 생성자 함수 객체가 아니어서 프로토타입 메소드와 정적 메소드 중 프로토타입 메소드만 제공 가능할 것이라 생각했는데 그게 아니라는 것! 생성자 함수와 정적 메소드를 연결하지 말아야 할 듯.

예를 들어 표준 빌트인 객체인 String, Number, Boolean, Function, Array, Date는 생성자 함수로 호출하여 인스턴스를 생성할 수 있다.

JAVASCRIPT

```
// String 생성자 함수에 의한 String 객체 생성
const strObj = new String('Lee');
console.log(typeof strObj); // object [String]
console.log(strObj);        // String {"Lee"}

// Number 생성자 함수에 의한 Number 객체 생성
const numObj = new Number(123);
console.log(typeof numObj); // object
console.log(numObj);        // Number {123}

// Boolean 생성자 함수에 의한 Boolean 객체 생성
const boolObj = new Boolean(true);
console.log(typeof boolObj); // object
console.log(boolObj);        // Boolean {true}

// Function 생성자 함수에 의한 Function 객체(함수) 생성
const func = new Function('x', 'return x * x');
console.log(typeof func); // function
console.dir(func);        // f anonymous(x )

// Array 생성자 함수에 의한 Array 객체(배열) 생성
const arr = new Array(1, 2, 3);
console.log(typeof arr); // object
console.log(arr);        // (3) [1, 2, 3]

// RegExp 생성자 함수에 의한 RegExp 객체(정규 표현식) 생성
const regExp = new RegExp(/ab+c/i);
console.log(typeof regExp); // object
console.log(regExp);        // /ab+c/i

// Date 생성자 함수에 의한 Date 객체 생성
const date = new Date();
console.log(typeof date); // object
console.log(date);        // Tue Mar 19 2019 02:38:26 GMT+0900 (한국 표준시)
```

생성자 함수인 표준 빌트인 객체가 생성한 인스턴스의 프로토타입은 표준 빌트인 객체의 prototype 프로퍼티에 바인딩된 객체이다. 예를 들어 표준 빌트인 객체인 String을 생성자 함수로서 호출하여 생성한 String 인스턴스의 프로토타입은 String.prototype이다.

JAVASCRIPT

```
// String 생성자 함수에 의한 String 객체 생성
const strObj = new String('Lee');
console.log(typeof strObj); // object
console.log(strObj);        // String {"Lee"}
```

```
console.log(Object.getPrototypeOf(strObj) === String.prototype); // true
```

표준 빌트인 객체의 prototype 프로퍼티에 바인딩된 객체(예를 들어, String.prototype)는 다양한 기능의 빌트인 프로토타입 메소드를 제공한다. 그리고 표준 빌트인 객체는 인스턴스 없이도 호출 가능한 빌트인 정적 메소드를 제공한다.

예를 들어, 표준 빌트인 객체인 Number의 prototype 프로퍼티에 바인딩된 객체, Number.prototype은 다양한 기능의 빌트인 프로토타입 메소드를 제공한다. 이 프로토타입 메소드는 모든 Number 인스턴스가 상속을 통해 사용할 수 있다. 그리고 표준 빌트인 객체인 Number는 인스턴스 없이 정적으로 호출할 수 있는 정적 메소드를 제공한다.

JAVASCRIPT

```
// Number 생성자 함수에 의한 Number 객체 생성
const numObj = new Number(1.5);
console.log(typeof numObj); // object
console.log(numObj);        // Number {1.5}

// toFixed는 프로토타입 메소드이다. > Number.prototype의 프로토타입 메소드
// 소숫점자리를 반올림하여 문자열로 반환한다.
console.log(numObj.toFixed()); // 2
```

프로토타입 메소드여서 앞에
Number.prototype을 써야할 것 같은 착각이 들지만 아니라는 것.
또한 프로토타입 메소드이기 때문에 인스턴스로 호출!

표준 빌트인 객체인 Number는 인스턴스 없이 정적으로 호출할 수 있는 정적 메소드도 제공한다..

JAVASCRIPT

```
// isInteger는 정적 메소드이다.
// 정수(Integer)인지 검사하여 그 결과를 Boolean으로 반환한다.
console.log(Number.isInteger(0.5)); // false
위와 달리 이건 정적 메소드이기 때문에 인스턴스인 numObj가 아닌,
Number 생성자 함수로 호출
```

3. 원시 값과 래퍼 객체

문자열이나 숫자, 불리언 등의 원시 값이 있음에도 불구하고 문자열, 숫자, 불리언 객체를 생성하는 String, Number, Boolean 등의 표준 빌트인 생성자 함수가 존재하는 이유는 무엇일까?

아래 예제를 살펴보자. 원시 값은 객체가 아니므로 프로퍼티나 메소드를 가질 수 없음에도 불구하고 원시 값인 문자열이 마치 객체처럼 동작한다.

```
JAVASCRIPT
123.->          가          가          .(          )
(123).          .->
'str'.->

const str = 'hello';

// 원시 타입인 문자열이 프로퍼티와 메소드를 갖고 있다.
console.log(str.length); // 5
console.log(str.toUpperCase()); // HELLO
```

표준 빌트인 객체가 제공하는 프로토타입 메소드를 사용하려면 반드시 인스턴스를 생성하고 인스턴스로 프로토타입 메소드를 호출해야 한다. 그런데 위 예제를 살펴보면 원시 값으로 표준 빌트인 객체의 프로토타입 메소드를 호출하면 정상적으로 동작한다.

이는 원시값인 문자열, 숫자, 불리언 값의 경우, 마치 객체처럼 이들 원시 값에 대해 마침표 표기법(또는 대괄호 표기법)으로 접근하면 자바스크립트 엔진이 일시적으로 원시 값을 연관된 객체로 변환해 주기 때문이다. 즉, 원시 값을 객체처럼 사용하면 자바스크립트 엔진은 암묵적으로 연관된 객체를 생성하고 생성된 객체로 프로퍼티에 접근하거나 메소드를 호출하고 다시 원시 값으로 되돌린다.

이처럼 문자열, 숫자, 불리언 값에 대해 객체처럼 접근하면 생성되는 임시 객체를 레퍼 객체(wrapper object)라 한다.

예를 들어, 문자열에 대해 마침표 표기법으로 접근하면 그 순간 레퍼 객체인 String 생성자 함수의 인스턴스가 생성되고 문자열은 레퍼 객체의 [[StringData]] 내부 슬롯에 할당된다.

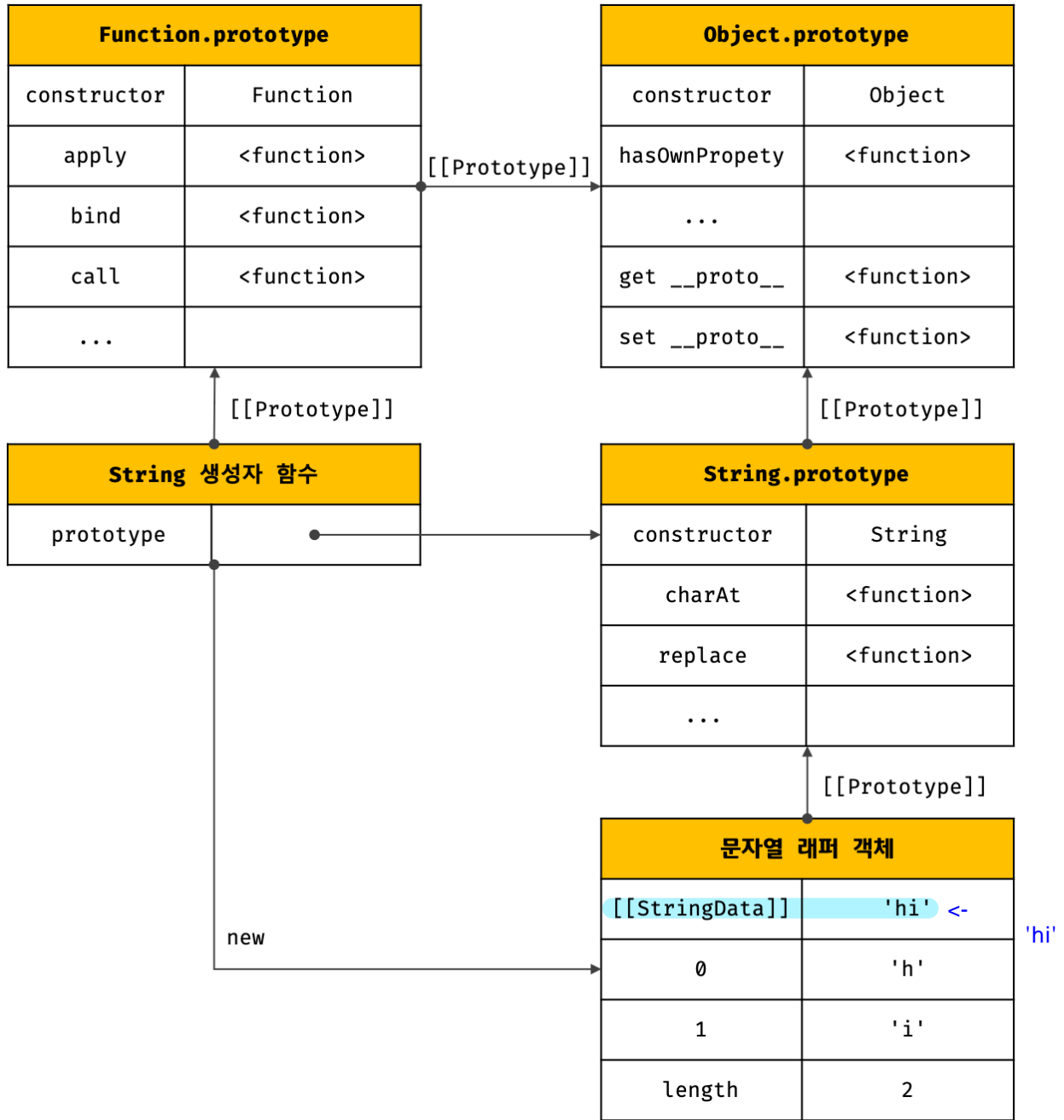
JAVASCRIPT

```
const str = 'hi';

// 원시 타입인 문자열이 레퍼 객체인 String 인스턴스로 변환된다.
console.log(str.length); // 2
console.log(str.toUpperCase()); // HI
    -> 만약에 toUpperCase를 못 쓴다면 for문으로 돌아 하나하나 바꿔줘야 함

// 레퍼 객체로 프로퍼티 접근이나 메소드 호출한 후, 다시 원시값으로 되돌린다.
console.log(typeof str); // string          object가          .
```

이때 문자열 래퍼 객체인 String 생성자 함수의 인스턴스는 String.prototype의 메소드를 상속받아 사용할 수 있다.



문자열 래퍼 객체의 프로토타입 체인

그 후, 래퍼 객체의 처리가 종료하면 래퍼 객체의 `[[StringData]]` 내부 슬롯에 할당된 원시값을 되돌리고 래퍼 객체는 가비지 컬렉션의 대상이 된다. 아래 예제를 살펴보자.



```
const str = 'hello'; // ①
```

```
// 래퍼 객체에 프로퍼티 추가
```

```
str.name = 'Lee'; // ② 무의미한 행동. 왜? 이미 원시값으로 되돌려 놓은 상태이기 때문에
```

```
// ③ str은 이전의 원시값으로 돌아간다.
```

```
// 이 시점에 str은 위 코드의 래퍼 객체가 아닌 새로운 래퍼 객체를 가리킨다.
console.log(str.name); // ④ undefined
// ⑤ str은 이전의 원시값으로 돌아간다.
```

① 식별자 str은 문자열을 값으로 가지고 있다. ② 식별자 str은 래퍼 객체를 가리킨다. ③ 식별자 str은 다시 원래의 문자열, 즉 래퍼 객체의 `[[StringData]]` 내부 슬롯에 할당된 원시값을 갖는다. ④ 식별자 str은 새로운(②에서 생성한 래퍼 객체와는 다른) 래퍼 객체를 가리킨다. ⑤ 식별자 str은 다시 원래의 문자열, 즉 래퍼 객체의 `[[StringData]]` 내부 슬롯에 할당된 원시값을 갖는다.

숫자도 마찬가지다. 숫자에 대해 마침표 표기법으로 접근하면 그 순간 래퍼 객체인 `Number` 생성자 함수의 인스턴스가 생성되고 숫자는 래퍼 객체의 `[[NumberData]]` 내부 슬롯에 할당된다. 이때 래퍼 객체인 `Number` 객체는 당연히 `Number.prototype`의 메소드를 상속받아 사용할 수 있다. 그 후, 래퍼 객체의 처리가 종료하면 래퍼 객체의 `[[NumberData]]` 내부 슬롯에 할당된 원시값을 되돌리고 래퍼 객체는 가비지 컬렉션의 대상이 된다.

JAVASCRIPT

```
const num = 1.5;

// 원시 타입인 숫자가 래퍼 객체인 String 객체로 변환된다.
console.log(num.toFixed()); // 2

// 래퍼 객체로 프로퍼티 접근이나 메소드 호출한 후, 다시 원시값으로 되돌린다.
console.log(typeof num, num); // number 1.5
```

불리언 값도 문자열이나 숫자와 마찬가지이지만 불리언 값으로 메소드를 호출할 일은 없으므로 그다지 유용하지는 않다.

ES6에서 새롭게 도입된 원시값인 심볼도 래퍼 객체를 생성한다. 심볼은 일반적인 원시값과는 달리 리터럴 표기법으로 생성할 수 없고 `Symbol` 함수를 통해 생성해야 하므로 다른 원시값과는 차이가 있다. 심볼에 대해서는 “28. 7번째 데이터 타입 `Symbol`”에서 자세히 살펴보도록 하자.

이처럼 문자열, 숫자, 불리언, 심볼은 암묵적으로 생성되는 래퍼 객체에 의해 마치 객체처럼 사용할 수 있으며 표준 빌트인 객체인 `String`, `Number`, `Boolean`, `Symbol`의 프로토타입 메소드 또는 프로퍼티를 참조할 수 있다. 따라서 `String`, `Number`, `Boolean` 생성자 함수를 `new` 연산자와 함께 호출하여 문자열, 숫자, 불리언 인스턴스를 생성할 필요가 없으며 권장하지도 않는다. `Symbol`은 생성자 함수가 아니므로 이 논의에서는 제외하도록 한다.

문자열, 숫자, 불리언, 심볼 이외의 원시값, 즉 `null`과 `undefined`는 레퍼 객체를 생성하지 않는다. 따라서 `null`과 `undefined` 값을 객체처럼 사용하면 에러가 발생한다.

4. 전역 객체

전역 객체(Global Object)는 코드가 실행되기 이전 단계에 자바스크립트 엔진에 의해 어떤 객체보다도 먼저 생성되는 특수한 객체이며 어떤 객체에도 속하지 않은 최상위 객체이다.

[주의] 프로토타입의 종점이라는 의미가 아님

전역 객체는 자바스크립트 환경에 따라 지칭하는 이름이 제각각이다. 브라우저 환경에서는 `window`(또는 `self`, `this`, `frames`)가 전역 객체를 가리키지만 Node.js 환경에서는 `global`이 전역 객체를 가리킨다.

globalThis

2019년 12월 현재, 전역 객체를 가리키는 식별자를 `globalThis`로 통일하는 제안이 stage 4에 올라와 있다. `globalThis`는 크롬 71, 파이어폭스 65, 사파리 12.1, Edge 79, Node.js 12.0.0 이상에 이미 구현되어 있다.

JAVASCRIPT

```
// 브라우저 환경
globalThis === this // true
globalThis === window // true
globalThis === self // true
globalThis === frames // true

// Node.js 환경(12.0.0 이상)
globalThis === this // true
globalThis === global // true
```

전역 객체는 표준 빌트인 객체(Object, String, Number, Function, Array...)들과 환경에 따른 호스트 객체(클라이언트 web API 또는 Node.js의 호스트 API) 그리고 `var` 키워드로 선언한 전역 변수와 전역 함수를 프로퍼티로 갖는다. [주의] 위에 열거한 것들이 전역 객체라는 것이 아니라, 전역 객체의 프로퍼티라는 것! 부분집합 개념!

즉, 전역 객체는 계층적 구조 상 어떤 객체에도 속하지 않은 모든 빌트인 객체(표준 빌트인 객체와 호스트 객체)의 최상위 객체이다. 전역 객체가 최상위 객체라는 것은 프로토타입 상속 관계 상에서 최상위 객체라는 의미가 아니고 객체의 계층적 구조 상 표준 빌트인 객체와 호스트 객체를 프로퍼티로 소유한다는 것을 말한다. &전역 객체 자신은 어떠한 객체의 프로퍼티도 아님.

전역 객체의 특징은 아래와 같다.

- 전역 객체는 개발자가 의도적으로 생성할 수 없다. 즉, 전역 객체를 생성할 수 있는 생성자 함수가 제공되지 않는다.
- 전역 객체의 프로퍼티를 참조할 때 window(또는 global)를 생략할 수 있다.

JAVASCRIPT

```
// 문자열 'F'를 16진수로 해석하여 10진수로 변환하여 반환한다.
console.log(window.parseInt('F', 16)); // 15
// window.parseInt는 parseInt으로 호출할 수 있다.
console.log(parseInt('F', 16)); // 15

console.log(window.parseInt === parseInt); // true
```

<전역 객체의 프로퍼티들>

- 전역 객체는 Object, String, Number, Boolean, Function, Array, RegExp, Date, Math, Promise와 같은 ^①모든 표준 빌트인 객체를 프로퍼티로 가지고 있다.
- 자바스크립트 실행 환경(브라우저 환경 또는 Node.js 환경. “3.1 자바스크립트 실행 환경” 참고)에 따라 추가적으로 프로퍼티와 메소드를 갖는다. 브라우저 환경에서는 DOM, BOM, Canvas, XMLHttpRequest, fetch, requestAnimationFrame, SVG, Web Storage, Web Component, Web worker와 같은 ^②호스트 객체 클라이언트 사이드 Web API를 호스트 객체로 제공하고 Node.js 환경에서는 Node.js 고유의 API를 호스트 객체로 제공한다.
- ^③var 키워드로 선언한 전역 변수와 ^④선언하지 않은 변수에 값을 할당한 암묵적 전역(“21.4.3. 암묵적 전역” 참고) 그리고 ^⑤전역 함수는 전역 객체의 프로퍼티가 된다.

JAVASCRIPT

```
// var 키워드로 선언한 전역 변수
var foo = 1;
console.log(window.foo); // 1
```

```
// 암묵적 전역. bar는 전역 변수가 아니라 전역 객체의 프로퍼티이다. -> (-> )
bar = 2; . ? window.
console.log(window.bar); // 2
```

```
// 전역 함수
function baz() { return 3; }
console.log(window.baz()); // 3
```

- let이나 const 키워드로 선언한 전역 변수는 전역 객체의 프로퍼티가 아니다. 즉, window.foo와 같이 접근할 수 없다. let이나 const 키워드로 선언한 전역 변수는 보이지 않는 개념적인 블록(전역 렉시컬 환경의 선언적 환경 레코드, “23. 실행 컨텍스트”에서 살펴볼 것이다.) 내에 존재하게 된다.

JAVASCRIPT

```
let foo = 123;
console.log(window.foo); // undefined
// window 객체에 존재하지 않는 foo 프로퍼티에 접근->undefined 반환
```

- 브라우저 환경의 모든 자바스크립트 코드는 하나의 전역 객체 window를 공유한다. 여러 개의 script 태그를 통해 자바스크립트 코드를 분리하여도 하나의 전역 객체 window를 공유하는 것은 변함이 없다. 이는 분리되어 있는 자바스크립트 코드가 하나의 전역을 공유한다는 의미이다.

전역 객체는 몇가지 프로퍼티와 메소드를 가지고 있다. 전역 객체의 프로퍼티와 메소드는 전역 객체를 가리키는 식별자, 즉 window나 global을 생략하여 참조/호출할 수 있으므로 전역 변수와 전역 함수처럼 사용할 수 있다. 이에 대해 살펴보자.

4.1 빌트인 전역 프로퍼티

빌트인 전역 프로퍼티(Built-in global property)는 전역 객체의 프로퍼티를 의미한다. 주로 애플리케이션 전역에서 사용하는 값을 제공한다.

4.1.1. Infinity

Infinity 프로퍼티는 양/음의 무한대를 나타내는 숫자값 Infinity를 갖는다.

JAVASCRIPT

```
// 전역 프로퍼티는 window를 생략하고 참조할 수 있다.
console.log(window.Infinity === Infinity); // true

// 양의 무한대
console.log(3/0); // Infinity
// 음의 무한대
console.log(-3/0); // -Infinity
```

```
// Infinity는 숫자 타입인 값이다.  
console.log(typeof Infinity); // number
```

4.1.2. NaN

NaN 프로퍼티는 숫자가 아님(Not-a-Number)을 나타내는 숫자값 NaN을 갖는다. NaN 프로퍼티는 Number.NaN 프로퍼티와 같다.

JAVASCRIPT

```
console.log(window.NaN); // NaN  
  
console.log(Number('xyz')); // NaN  
console.log(1 * 'string'); // NaN  
console.log(typeof NaN); // number
```

4.1.3. undefined

undefined 프로퍼티는 원시 타입 undefined를 값으로 갖는다.

JAVASCRIPT

```
console.log(window.undefined); // undefined  
  
var foo;  
console.log(foo); // undefined  
console.log(typeof undefined); // undefined
```

4.2. 빌트인 전역 함수

빌트인 전역 함수(Built-in global function)는 애플리케이션 전역에서 호출할 수 있는 빌트인 함수로서 전역 객체의 메소드이다.

4.2.1. eval

-> 따라서 무조건 따옴표로 감싸줘야 함.

문자열 형태로 매개변수에 전달된 코드를 런타임에 동적으로 평가하고 실행하여 결과값을 반환한다. 전달된 문자열 코드가 여러 개의 문으로 이루어져 있다면 모든 문을 실행 후 마지막 결과값을 반환한다.

JAVASCRIPT

```
/**
 * 주어진 코드를 런타임 평가하고 실행하여 결과값을 반환한다.
 * @param {string} code - 코드를 나타내는 문자열
 * @returns {*} 문자열 코드를 평가/실행한 결과값
 */
eval(code)
```

JAVASCRIPT

```
// 표현식인 문
console.log(eval('1 + 2;')); // 3
// 표현식이 아닌 문
console.log(eval('var x = 5;')); // undefined
// 변수 x가 선언되었다.
console.log(x); // 5

// 객체 리터럴은 반드시 괄호로 둘러싼다.
var o = eval('{ a: 1 }');
console.log(o); // {a: 1}

// 함수 리터럴은 반드시 괄호로 둘러싼다.
var f = eval('(function() { return 1; })');
console.log(f()); // 1
```

전달된 문자열 코드가 여러 개의 문으로 이루어져 있다면 모든 문을 실행 후 마지막 결과값을 반환한다.

JAVASCRIPT

```
console.log(eval('1 + 2; 3 + 4;')); // 7
```

`eval` 함수는 런타임에 자신이 호출된 기존의 스코프를 동적으로 수정한다. 아래 예제를 살펴보자.

JAVASCRIPT

```
var x = 1;

function foo() {
  // eval 함수는 런타임에 foo 함수의 스코프를 동적으로 수정한다.
  eval('var x = 2;');
  console.log(x); // 2
}

foo();

console.log(x); // 1
```

새로운 변수 뭐? `eval` 함수가 호출되는 시점이 언제임?

위 예제의 `eval` 함수는 새로운 변수를 선언하면서 `foo` 함수의 스코프에 선언된 변수를 동적으로 추가한다. `eval` 함수가 호출되는 시점에는 이미 `foo` 함수의 스코프가 존재한다. 따라서 `eval` 함수는 기존의 스코프를 동적으로 수정하는 것이다. 그리고 `eval` 함수에 전달된 코드는 이미 그 위치에 존재하던 코드처럼 동작한다. 즉, `eval` 함수가 호출된 `foo` 함수의 스코프에서 실행된다.

자바스크립트는 렉시컬 스코프("13.5. 렉시컬 스코프" 참고)를 따르므로 스코프는 함수 정의가 평가되는 시점에 결정된다. 다시 말해 스코프는 런타임에 결정되는 것이 아니다. 하지만 `eval` 함수는 런타임에 기존의 스코프를 동적으로 수정할 수 있다. 다시 말해 `eval` 함수는 렉시컬 스코프를 동적으로 수정할 수 있다. 하지만 성능적인 면에서 손해를 감수해야 한다.

엄격 모드(strict mode)에서 `eval` 함수는 기존의 스코프를 수정하지 않고 자신만의 독자적인 스코프를 생성한다.

JAVASCRIPT

```
var x = 1;

function foo() {
  'use strict';

  // 엄격 모드에서 eval 함수는 기존의 스코프를 수정하지 않고 자체적인 스코프를 생성한다.
  eval('var x = 2; console.log(x);'); // 2
  console.log(x); // 1
}
```

```
foo();

console.log(x); // 1
```

또한 eval 함수에 전달한 변수 선언문이 let, const 키워드를 사용했다면 엄격 모드가 적용된다.

JAVASCRIPT

```
var x = 1;

function foo() {
  // 'use strict';
  // eval 함수에 전달한 변수 선언문이 let, const 키워드를 사용했다면 엄격 모드가 적용된다.
  eval('const x = 2; console.log(x);'); // 2
  console.log(x); // 1
}

foo();

console.log(x); // 1
```

eval 함수를 통해 사용자로부터 입력 받은 콘텐츠(untrusted data)를 실행하는 것은 보안에 매우 취약하다. 또한 자바스크립트 엔진에 의해 최적화가 수행되지 않으므로 일반적인 코드 실행에 비해 처리 속도가 느리다. 따라서 eval 함수의 사용은 가급적 금지되어야 한다.

4.2.2. isFinite

즉, 전달되는 값이 숫자라는 것을 추측할 수 있음

매개 변수에 전달된 값이 정상적인 유한수인지 검사하여 그 결과를 불리언 타입으로 반환한다. 매개 변수에 전달된 값이 숫자가 아닌 경우, 숫자로 타입을 변환한 후 검사를 수행한다.

JAVASCRIPT

```
/**
 * 주어진 숫자가 유한수인지 확인하고 그 결과를 반환한다.
 * @param {number} testValue - 검사 대상 값
 * @returns {boolean} 유한수 여부 확인 결과값
```

```
*/
isFinite(testValue)
```

JAVASCRIPT

```
console.log(isFinite(Infinity)); // false
console.log(isFinite(NaN));      // false
console.log(isFinite('Hello'));  // false
console.log(isFinite('2005/12/12')); // false

console.log(isFinite(0));         // true
console.log(isFinite(2e64));      // true
console.log(isFinite('10'));     // true: '10' → 10
console.log(isFinite(null));     // true: null → 0
```

isFinite(null)은 true를 반환한다. 이것은 null을 숫자로 변환하여 검사를 수행하였기 때문이다. null을 숫자 타입으로 변환하면 0이 된다. (“9. 타입 변환과 단축 평가” 참고)

JAVASCRIPT

```
console.log(+null); // 0
```

4.2.3. isNaN

매개변수에 전달된 값이 NaN인지 검사하여 그 결과를 불리언 타입으로 반환한다. 매개변수에 전달된 값이 숫자가 아닌 경우, 숫자로 타입을 변환한 후 검사를 수행한다.

필요한 이유 : NaN은 NaN과 같지 않기 때문에(일치 비교 연산자)

// NaN은 자신과 일치하지 않는 유일한 값이다.

NaN === NaN; // → false

JAVASCRIPT

```
/**
 * 주어진 숫자가 NaN인지 확인하고 그 결과를 반환한다.
 * @param {number} testValue - 검사 대상 값
 * @returns {boolean} NaN 여부 확인 결과값
 */
isNaN(testValue)
```

JAVASCRIPT

```

// 숫자
console.log(isNaN(NaN)); // true
console.log(isNaN(10)); // false

// 문자열
console.log(isNaN('blabla')); // true: 'blabla' → NaN
console.log(isNaN('10')); // false: '10' → 10
console.log(isNaN('10.12')); // false: '10.12' → 10.12
console.log(isNaN('')); // false: '' → 0
console.log(isNaN(' ')); // false: ' ' → 0

// 불리언
console.log(isNaN(true)); // false: true → 1
console.log(isNaN(null)); // false: null → 0

// undefined
console.log(isNaN(undefined)); // true: undefined → NaN

// 객체
console.log(isNaN({})); // true: {} → NaN

// date
console.log(isNaN(new Date())); // false: new Date() → Number
console.log(isNaN(new Date().toString())); // true: String → NaN

```

4.2.4. parseFloat parse : '해석하다'라는 의미

매개변수에 전달된 문자열을 부동소수점 숫자(floating point number)로 변환하여 반환한다.

JAVASCRIPT

```

/**
 * 주어진 문자열을 부동소수점 숫자로 변환하여 반환한다.
 * @param {string} string - 변환 대상 값
 * @returns {number} 변환 결과값
 */
parseFloat(string)

```


JAVASCRIPT

```

console.log(parseFloat('3.14')); // 3.14
console.log(parseFloat('10.00')); // 10
// 공백으로 구분된 문자열은 첫번째 문자열만 변환한다.
console.log(parseFloat('34 45 66')); // 34
console.log(parseFloat('40 years')); // 40
// 첫번째 문자열을 숫자로 변환할 수 없다면 NaN을 반환한다.
console.log(parseFloat('He was 40')); // NaN
// 전후 공백은 무시된다.
console.log(parseFloat(' 60 ')); // 60

```



4.2.5. parseInt

매개변수에 전달된 문자열을 정수형 숫자(Integer)로 해석(parsing)하여 반환한다. 반환값은 언제나 10진수이다.

JAVASCRIPT

```

/**
 * 주어진 문자열을 정수형 숫자(Integer)로 해석(parsing)하여 반환한다.
 * 반환값은 언제나 10진수이다.
 * @param {string} string - 변환 대상 값
 * @param {number} [radix] - 진법을 나타내는 기수(2 ~ 36, 기본값 10)
 * @returns {number} 변환 결과값
 */
parseInt(string, radix);

```

JAVASCRIPT

```

// 주어진 문자열을 10진수 정수로 해석하여 반환한다.
console.log(parseInt('10')); // 10
console.log(parseInt('10.123')); // 10

```

주어진 변환 대상 값이 문자열이 아니면 문자열로 변환한 후 정수형 숫자로 해석하여 반환한다.

JAVASCRIPT

```
console.log(parseInt(10));    // 10
console.log(parseInt(10.123)); // 10
```

2번째 매개변수에는 진법을 나타내는 기수(2 ~ 36)를 지정할 수 있다. 기수를 지정하면 첫번째 매개변수에 전달된 문자열을 해당 기수의 숫자로 해석하여 반환한다. 이때 반환값은 언제나 10진수이다. 기수를 생략하면 첫번째 매개변수에 전달된 문자열을 10진수로 해석하여 반환한다.

JAVASCRIPT

```
// '10'을 10진수로 해석하고 10진수 정수로 그 결과를 반환한다
console.log(parseInt('10')); // 10
// '10'을 2진수로 해석하고 10진수 정수로 그 결과를 반환한다
console.log(parseInt('10', 2)); // 2
// '10'을 8진수로 해석하고 10진수 정수로 그 결과를 반환한다
console.log(parseInt('10', 8)); // 8
// '10'을 16진수로 해석하고 10진수 정수로 그 결과를 반환한다
console.log(parseInt('10', 16)); // 16
```

기수를 지정하여 10진수 숫자를 해당 기수의 문자열로 변환하여 반환하고 싶을 때는 `Number.prototype.toString` 메소드를 사용한다.

JAVASCRIPT

```
const x = 15;

// 15을 2진수로 변환하여 그 결과를 문자열로 반환한다.
console.log(x.toString(2)); // '1111'
// 15을 8진수로 변환하여 그 결과를 문자열로 반환한다.
console.log(x.toString(8)); // '17'
// 15을 16진수로 변환하여 그 결과를 문자열로 반환한다.
console.log(x.toString(16)); // 'f'

// 숫자값을 문자열로 변환한다.
console.log(x.toString()); // '15'
```

두번째 매개변수에 진법을 나타내는 기수를 지정하지 않더라도 첫번째 매개변수에 전달된 문자열이 “0x” 또는 “0X”로 시작하는 16진수 리터럴이라면 16진수로 해석하여 10진수 정수로 반환한다.

JAVASCRIPT

```
// 16진수 리터럴 '0xf'를 16진수로 해석하고 10진수 정수로 그 결과를 반환한다.
console.log(parseInt('0xf')); // 15
// 위 코드와 같다.
console.log(parseInt('f', 16)); // 15
```

하지만 2진수 리터럴과 8진수 리터럴은 제대로 해석하지 못한다.

JAVASCRIPT

```
// 2진수 리터럴(0b로 시작) => 0 이후 무시
console.log(parseInt('0b10')); // 0
// 8진수 리터럴(ES6에서 도입. 0o로 시작) => 0 이후 무시
console.log(parseInt('0o10')); // 0
```

ES5 이전까지는 비록 사용을 금지하고는 있었지만 “0”로 시작하는 숫자를 8진수로 해석하였다. ES6부터는 “0”로 시작하는 숫자를 8진수로 해석하지 않고 10진수로 해석한다. 따라서 문자열을 8진수로 해석하려면 지수를 반드시 지정하여야 한다.

JAVASCRIPT

```
// 문자열 '10'을 2진수로 해석한다.
console.log(parseInt('10', 2)); // 2
// 문자열 '10'을 8진수로 해석한다.
console.log(parseInt('10', 8)); // 8
```

첫번째 매개변수에 전달된 문자열의 첫번째 문자가 해당 지수의 숫자로 변환될 수 없다면 NaN을 반환한다.

JAVASCRIPT

```
// 'A'는 10진수로 해석할 수 없다.
console.log(parseInt('A0')); // NaN
// '2'는 2진수로 해석할 수 없다.
console.log(parseInt('20', 2)); // NaN
```

하지만 첫번째 매개변수에 전달된 문자열의 두번째 문자부터 해당 진수를 나타내는 숫자가 아닌 문자(예를 들어 2진수의 경우, 2)와 마주치면 이 문자와 계속되는 문자들은 전부 무시되며 해석된 정수값만을 반환한다.

JAVASCRIPT

```
// 10진수로 해석할 수 없는 'A'이후의 문자는 모두 무시된다.
console.log(parseInt('1A0')); // 1
// 2진수로 해석할 수 없는 '2'이후의 문자는 모두 무시된다.
console.log(parseInt('102', 2)); // 2
// 8진수로 해석할 수 없는 '8'이후의 문자는 모두 무시된다.
console.log(parseInt('58', 8)); // 5
// 16진수로 해석할 수 없는 'G'이후의 문자는 모두 무시된다.
console.log(parseInt('FG', 16)); // 15
```

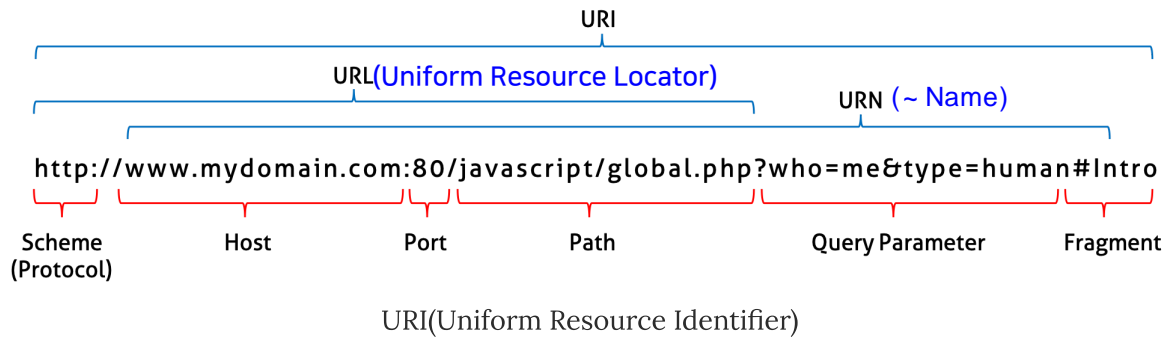
첫번째 매개변수에 전달된 문자열에 공백이 있다면 첫번째 문자열만 해석하여 반환하며 전후 공백은 무시된다. 만일 첫번째 문자열을 숫자로 해석할 수 없는 경우, NaN을 반환한다.

JAVASCRIPT

```
// 공백으로 구분된 문자열은 첫번째 문자열만 변환한다.
console.log(parseInt('34 45 66')); // 34
console.log(parseInt('40 years')); // 40
// 첫번째 문자열을 숫자로 변환할 수 없다면 NaN을 반환한다.
console.log(parseInt('He was 40')); // NaN
// 전후 공백은 무시된다.
console.log(parseInt(' 60 ')); // 60
```

4.2.6. encodeURIComponent / decodeURI

encodeURIComponent 함수는 매개변수로 전달된 URI(Uniform Resource Identifier)를 인코딩한다. URI는 인터넷에 있는 자원을 나타내는 유일한 주소를 말한다. URI의 하위개념으로 URL, URN이 있다.



JAVASCRIPT

```
/**
 * 완전한 URI를 전달받아 인코딩하여 이스케이프 처리한다.
 * @param {string} uri - 완전한 URI
 * @returns {string} 인코딩된 URI
 */
encodeURIComponent(uri)
```

JAVASCRIPT

```
/**
 * 인코딩된 URI을 전달받아 이스케이프 처리되기 이전으로 디코딩한다.
 * @param {string} encodedURI - 인코딩된 URI
 * @returns {string} 디코딩된 URI
 */
decodeURI(encodedURI)
```

인코딩이란 URI의 문자들을 이스케이프 처리하는 것을 의미한다.

JAVASCRIPT

```
// 완전한 URI
const uri = 'http://example.com?name=이웅모&job=programmer&teacher';

// encodeURIComponent 함수는 완전한 URI를 전달받아 인코딩하여 이스케이프 처리한다.
const enc = encodeURIComponent(uri);
console.log(enc);
// http://example.com?name=%EC%9D%B4%EC%9B%85%EB%AA%A8&job=programmer&teacher
```

이스케이프 처리는 네트워크를 통해 정보를 공유할 때 어떤 시스템에서도 읽을 수 있는 아스키 문자 셋 (ASCII Character-set)으로 변환하는 것이다. UTF-8 특수문자의 경우, 1문자당 1~3byte, UTF-8 한글 표현의 경우, 1문자당 3byte이다. 예를 들어 특수문자 공백(space)은 %20, 한글 '가'는 %EC%9E%90으로 인코딩된다.

URI 문법 형식 표준 RFC3986에 따르면 URL은 아스키 문자 셋으로만 구성되어야 하며 한글을 포함한 대부분의 외국어나 아스키 문자 셋에 정의되지 않은 특수문자의 경우, URL에 포함될 수 없다. 따라서 URL 내에서 의미를 갖고 있는 문자(% , ? , #)나 URL에 올 수 없는 문자(한글, 공백 등) 또는 시스템에 의해 해석될 수 있는 문자(< , >)를 이스케이프 처리하여 야기될 수 있는 문제를 예방하기 위해 이스케이프 처리가 필요하다. 단, 알파벳, 0~9의 숫자, - _ . ! ~ * ' () 문자는 이스케이프 처리에서 제외된다.

decodeURI 함수는 매개변수로 전달된 인코딩된 URI을 전달받아 이스케이프 처리되기 이전으로 디코딩한다.

JAVASCRIPT

```
const uri = 'http://example.com?name=이웅모&job=programmer&teacher';

// encodeURI 함수는 완전한 URI를 전달받아 인코딩하여 이스케이프 처리한다.
const enc = encodeURI(uri);
console.log(enc);
// http://example.com?name=%EC%9D%B4%EC%9B%85%EB%AA%A8&job=programmer&teacher

// decodeURI 함수는 인코딩된 완전한 URI를 전달받아 이스케이프 처리되기 이전으로 디코딩한다.
const dec = decodeURI(enc);
console.log(dec);
// http://example.com?name=이웅모&job=programmer&teacher
```

4.2.7. encodeURIComponent / decodeURIComponent

encodeURIComponent 함수는 매개변수로 전달된 URI(Uniform Resource Identifier) 구성 요소(component)를 인코딩한다. 여기서 인코딩이란 URI의 문자들을 이스케이프 처리하는 것을 의미한다. 단, 알파벳, 0~9의 숫자, - _ . ! ~ * ' () 문자는 이스케이프 처리에서 제외된다.

decodeURIComponent 함수는 매개변수로 전달된 URI 구성 요소를 디코딩한다.

JAVASCRIPT

```
/**
 * URI의 구성요소를 전달받아 인코딩하여 이스케이프 처리한다.
 * @param {string} uriComponent - URI의 구성요소
 * @returns {string} 인코딩된 URI의 구성요소
 */
encodeURIComponent(uriComponent)
```

JAVASCRIPT

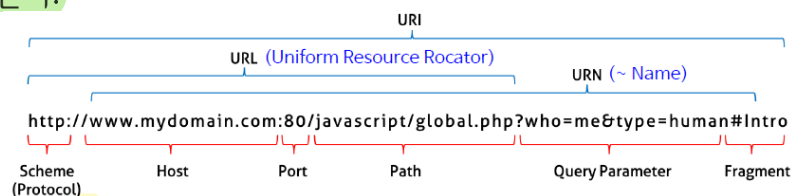
```
/**
 * 인코딩된 URI의 구성요소를 전달받아 이스케이프 처리되기 이전으로 디코딩한다.
 * @param {string} encodedURIComponent - 인코딩된 URI의 구성요소
 * @returns {string} 디코딩된 URI의 구성요소
 */
decodeURIComponent(encodedURIComponent)
```

`encodeURIComponent` 함수는 매개변수로 전달된 문자열을 URI의 구성요소인 쿼리 파라미터의 일부 간주한다. 따라서 쿼리 파라미터 구분자로 사용되는 `=`, `?`, `&`를 인코딩한다.

반면 `encodeURI` 함수는 매개변수로 전달된 문자열을 완전한 URI 전체라고 간주한다. 따라서 쿼리 파라미터 구분자로 사용되는 `=`, `?`, `&`를 인코딩하지 않는다.

JAVASCRIPT

```
// URI의 쿼리 파라미터
const uriComp = 'name=이웅모&job=programmer&teacher';
```



```
// encodeURIComponent 함수는 매개변수로 전달된 문자열을 URI의 구성요소인 쿼리 파라미터의 일부 간주한다.
```

```
// 따라서 쿼리 파라미터 구분자로 사용되는 =, ?, &를 인코딩한다.
```

```
let enc = encodeURIComponent(uriComp);
console.log(enc);
// name%3D%EC%9D%B4%EC%9B%85%EB%AA%A8%26job%3Dprogrammer%26teacher
```

```
let dec = decodeURIComponent(enc);
console.log(dec);
// 이웅모&job=programmer&teacher
```

```
// encodeURI 함수는 매개변수로 전달된 문자열을 완전한 URI로 간주한다.
```

```
// 따라서 쿼리 파라미터 구분자로 사용되는 =, ?, &를 인코딩하지 않는다.
enc = encodeURIComponent(uriComp);
console.log(enc);
// name=%EC%9D%B4%EC%9B%85%EB%AA%A8&job=programmer&teacher

dec = decodeURI(enc);
console.log(dec);
// name=이웅모&job=programmer&teacher
```

4.3. 암묵적 전역

먼저 아래 예제를 살펴보자.

JAVASCRIPT

```
var x = 10; // 전역 변수

function foo () {
  y = 20; // 선언하지 않은 식별자에 값을 할당
}
foo();

// 선언하지 않은 식별자 y를 전역에서 참조할 수 있다.
console.log(x + y); // 30
```

foo 함수 내의 y는 선언하지 않은 식별자이다. 따라서 `y = 20` 이 실행되면 참조 에러가 발생할 것처럼 보인다. 하지만 선언하지 않은 식별자 y는 마치 선언된 전역 변수처럼 동작한다. 이는 선언하지 않은 식별자에 값을 할당하면 전역 객체의 프로퍼티가 되기 때문이다.

foo 함수가 호출되면 자바스크립트 엔진은 변수 y에 값을 할당하기 위해 먼저 스코프 체인을 통해 선언된 변수인지 확인한다. 이때 foo 함수의 스코프와 전역 스코프 어디에서도 변수 y의 선언을 찾을 수 없으므로 참조 에러가 발생해야 한다. 하지만 자바스크립트 엔진은 `y = 20` 을 `window.y = 20` 으로 해석하여 전역 객체에 프로퍼티를 동적 생성한다. 결국 y는 전역 객체의 프로퍼티가 되어 마치 전역 변수처럼 동작한다. 이러한 현상을 **암묵적 전역(implicit global)**이라 한다.

하지만 y는 변수 선언없이 단지 전역 객체의 프로퍼티로 추가되었을 뿐이다. 따라서 y는 변수가 아니다. 따라서 변수가 아닌 y는 변수 호이스팅이 발생하지 않는다.

JAVASCRIPT

```
// 전역 변수 x는 호이스팅이 발생한다.
console.log(x); // undefined
// 전역 변수가 아니라 단지 전역 객체의 프로퍼티인 y는 호이스팅이 발생하지 않는다.
console.log(y); // ReferenceError: y is not defined

var x = 10; // 전역 변수

function foo () {
  y = 20; // 선언하지 않은 식별자에 값을 할당
}
foo();

// 선언하지 않은 식별자 y를 전역에서 참조할 수 있다.
console.log(x + y); // 30
```

또한 변수가 아니라 단지 프로퍼티인 y는 delete 연산자로 삭제할 수 있다. 전역 변수는 프로퍼티이지만 delete 연산자로 삭제할 수 없다.

JAVASCRIPT

```
var x = 10; // 전역 변수

function foo () {
  // 선언하지 않은 변수
  y = 20;
  console.log(x + y);
}

foo(); // 30

console.log(window.x); // 10 x: 전역 변수
console.log(window.y); // 20 y: 전역 객체의 프로퍼티

delete x; // 전역 변수는 삭제되지 않는다.
delete y; // 프로퍼티는 삭제된다.
```

```
console.log(window.x); // 10  
console.log(window.y); // undefined
```