21장. 빌트인 객체 p.320

21.1 자바스크립트 객체의 분류

- 표준 빌트인 객체 (standard built-in obtects/native objects/global objects)
  - ECMAScript 사양에 정의된 객체
  - 애플리케이션 전역의 공통 기능을 제공
  - ECMAScript 사양에 정의된 객체이므로 자바스크립트 실행 환경 (브라우저 또는 Node.js 환경) 과 관계 없이 언제나 사용 가능
  - 전역 객체의 프로퍼티로서 제공됨 -> 따라서 별도의 선언 없이 전역 변수처럼 언제나 참조 가능
- 호스트 객체 (host objects)
  - ECMAScript 사양에 정의되어 있지 않지만 자바스크립트 실행 환경 (브라우저 또는 Node.js 환경) 에서 추가로 제공하는 객체
  - 브라우저 환경에서는 클라이언트 사이드 Web API를 호스트 객체로 제공
  - Node.js 환경에서는 Node.js 고유의 API를 호스트 객체로 제공
- 사용자 정의 객체 (user-defined objects)
  - 표준 빌트인 객체와 호스트 객체처럼 기본 제공되는 객체가 아닌 사용자가 직접 정의한 객체

21.2 표준 빌트인 객체

- 자바스크립트는 40여 개의 표준 빌트인 객체를 제공

  > 예) Object, String, Number, Boolean, Symbol, Date, Math, RegExp, Array, Map/Set, WeakMap/WeakSet, Function, Promise, Reflect, Proxy, JSON, Error 등

- 생성자 함수 객체인 표준 빌트인 객체
  - 프로토타입 메서드와 정적 메서드를 제공
  - Math, Reflect, JSON을 제외한 표준 빌트인 객체
- 생성자 함수 객체가 아닌 표준 빌트인 객체
  - 정적 메서드만 제공
  - Math, Reflect, JSON
- 예제) 몇 가지 표준 빌트인 객체를 생성자 함수로 호출하여 인스턴스를 생성한 코드

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

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

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

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

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

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

// Date 생성자 함수에 의한 Date 객체 생성
const date = new Date(); // Fri May 08 2020 10:43:25 GMT+0900 (대한민국 표준시)
console.log(typeof date); // object
```

- 생성자 함수인 표준 빌트인 객체가 생성한 인스턴스의 프로토타입: 표준 빌트인 객체의 prototype 프로퍼티에 바인딩된 객체
- 예를 들어, 표준 빌트인 객체인 String을 생성자 함수로서 호출하여 생성한 String 인스턴스의 프로토타입은 String.prototype이다. 아래 코드를 통해 확인 가능

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

// String 생성자 함수를 통해 생성한 strObj 객체의 프로토타입은 String.prototype이다.
console.log(Object.getPrototypeOf(strObj) === String.prototype); // true
```

- 표준 빌트인 객체의 prototype 프로퍼티에 바인딩된 객체: 다양한 기능의 빌트인 프로토타입 메서드를 제공
- 표준 빌트인 객체: 인스턴스 없이도 호출 가능한 빌트인 정적 메서드를 제공
- 아래 예제) 표준 빌트인 객체인 Number의 prototype 프로퍼티에 바인딩된 객체 Number.prototype은 다양한 빌트인 프로토타입 메서드를 제공한다는 걸 보여준다.
  - 이 프로토타입 메서드는 모든 Number 인스턴스가 상속을 통해 사용할 수 있다.
  - 표준 빌트인 객체인 Number는 인스턴스 없이 정적으로 호출할 수 있는 정적 메서드를 제공한다.

```
// Number 생성자 함수에의한 Number 객체 생성
const numObj = new Number(1.5); // Number {1.5}

// toFixed는 Number.prototyped 프로토타입 메서드다.
// Number.prototype.toFixed는 소수점 자리를 반올림하여 문자열로 반환한다.
console.log(num0bj.toFixed()); // 2

// islnteger는 Number의 정적 메서드다.
// Number.islnteger는 인수가 정수(integer)인지 검사하여 그 결과를 Boolean으로 반환한다.
console.log(Number.isInteger(0.5)); // false

console.log(Number.toFixed(1.5)); // TypeError: Number.toFixed is not a function
```

21.3 원시값과 래퍼 객체

> Q. 문자열이나 숫자, 불리언 등의 원시값이 있는데도 문자열, 숫자, 불리언 객체를 생성하는 String, Number, Boolean 등의 표준 빌트인 생성자 함수가 존재하는 이유는 무엇일까?
>
> > 원시값도 객체처럼 사용할 수는 있지만 프로토타입 메서드나 프로퍼티를 참조할 수 없기 때문에 프로토타입 메서드나 프로퍼티를 참조할 수 있는 표준 빌트인 생성자 함수를 사용한다.

- 다음 예제를 보자.

```
const str = 'hello ' ;
console.log(str.length); // 5
console.log(str.toUpperCase()); // HELLO
```

  - 원시값은 객체가 아니므로 프로퍼티나 메서드를 가질 수 없는데도 원시값인 문자열이 마치 객체처럼 동작한다.
    - 이렇게 동작이 가능한 이유: 원시값인 문자열, 숫자, 불리언 값의 경우 이들 원시값에 대해 마치 객체처럼 접근하면 **자바스크립트 엔진이 일시적으로 원시값을 연관된 객체로 변환해 주기 때문이다.**
    - 객체로 변환한 이후에는: 생성된 객체로 프로퍼티에 접근하거나 메서드를 호출하고 다시 원시값으로 되돌린다.
- 래퍼 객체: 위 예제처럼 문자열, 숫자, 불리언 값에 대해 객체처럼 접근하면 생성되는 임시 객체
- 아래 예제를 보면, 문자열에 대해 객체처럼 접근하면 그 순간 래퍼 객체인 String 생성자 함수의 인스턴스가 생성되고 문자열은 래퍼 객체의 [[StringData]] 내부 슬롯에 할당된다.

```
const str = 'hi' ;

// 원시 타입인 문자열이 래퍼 객체인 String 인스턴스로 변환된다.
console.log(str.length); // 2
console.log(str.toUpperCase()); // HI

// 래퍼 객체로 프로퍼티에 접근하거나 메서드를 호출한 후, 다시 원시값으로 되돌린다.
console.log(typeof str); // string
```

- 이때 문자열 래퍼 객체인 String 생성자 함수의 인스턴스는 String.prototype의 메서드를 상속받아 사용할 수 있다.
- 그 후 래퍼 객체의 처리가 종료되면 래퍼 객체의 [[StringData]] 내부 슬롯에 할당된 원시값으로 원래의 상태, 즉 식별자가 원시값을 갖도록 되돌리고 래퍼 객체는 가비지 컬렉션의 대상이 된다. 아래 예제로 확인 가능

```
// ① 식별자 str은 문자열을 값으로 가지고 있음
const str = 'hello';

// ② 식별자 str: 암묵적으로 생성된 래퍼 객체를 가리킴
// 식별자 str의 값 'hello': 래퍼 객체의 [[StringData]] 내부 슬롯에 할당됨

// 래퍼 객체에 name 프로퍼티가 동적 추가됨
str.name = 'Lee';

// ③ 식별자 str은 다시 원래의 문자열, 즉 래퍼 객체의 [[StringData]] 내부 슬롯에 할당된 원시값을
// 이때 ②에서 생성된 래퍼 객체: 아무도 참조하지 않는 상태이므로 가비지 컬렉션의 대상이 됨
// ④ 식별자 str은 새로게 암묵적으로 생성된(②에서 생성된 래퍼 객체와는 다른) 래퍼 객체를 가리킴
// 하지만 새롭게 생성된 래퍼 객체에는 name 프로퍼티가 존재하지 않음
console.log(str.name); // undefined

// ⑤ 식별자 str은 다시 원래의 문자열, 즉 래퍼 객체의 [[StringData]] 내부 슬롯에 할당된 원시값을
// 이때 ④에서 생성된 래퍼 객체: 아무도 참조하지 않는 상태이므로 가비지 컬렉션의 대상이 됨
console.log(typeof str, str); // string hello
```

- 숫자 값과 불리언 값도 마찬가지이다. 하지만 불리언 값으로 메서드를 호출하는 경우는 없으므로 그다지 유용하지는 않다.
- ES6에서 새롭게 도입된 원시값인 심벌도 래퍼 객체를 생성한다.
- 심벌은 일반적인 원시값과는 달리 리터럴 표기법으로 생성할 수 없고 Symbol 함수를 통해 생성해야 하므로 다른 원시값과는 차이가 있다.
  - 심벌에 대해서는 33장 “7번째 데이터 타입 Symbol”에서 살펴보기
- 이처럼 문자열, 숫자, 불리언, 심벌은 암묵적으로 생성되는 래퍼 객체에 의해 마치 객체처럼 사용할 수 있으며, 표준 빌트인 객체인 String, Number, Boolean, Symbol의 프로토타입 메서드 또는 프로퍼티를 참조할 수 있다.
- 따라서 String, Number, Boolean 생성자 함수를 new 연산자와 함께 호출하여 문자열, 숫자, 불리언 인스턴스를 생성할 필요가 없으며 권장하지도 않는다.

> Symbol은 생성자 함수가 아니므로 이 논의에서는 제외하도록 한다.

- 문자열, 숫자, 불리언, 심벌 이외의 원시값, 즉 (null과 undefined는 래퍼 객체를 생성하지 않는다.)
  - 따라서 null과 undefined 값을 객체처럼 사용하면 에러가 발생한다.

21.4 전역 객체

- 전역 객체
  - 코드가 실행되기 이전 단계에 자바스크립트 엔진에 의해 어떤 객체보다도 먼저 생성되는 특수한 객체
  - 어떤 객체에도 속하지 않은 최상위 객체
  - 자바스크립트 환경에 따라 지칭하는 이름이 제각각임
    - 브라우저 환경: window (또는 self, this, frams)
    - Node.js 환경: global
- ES11에서 도입된 globalThis
  - 브라우저 환경과 Node.js 환경에서 전역 객체를 가리키던 다양한 식별자를 통일한 식별자
  - globalThis는 표준 사양이므로 ECMAScript 표준 사양을 준수하는 모든 환경에서 사용할수 있음

```
// 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 계층적 구조상 어떤 객체에도 속하지 않은 모든 빌트인 객체 (표준 빌트인 객체와 호스트 객체) 의 최상위 객체 이므로

> 표준 빌트인 객체 (Object, String, Number, Function, Array 등) 와 환경에 따른 호스트 객체 (클라이언트 Web API 또는 Node.js의 호스트 API) , 그리고 var 키워드로 선언한 전역 변수와 전역 함수를 프로퍼티로 갖는다.

- 전역 객체가 최상위 객체라는 것은
  - 프로토타입 상속 관계상에서 최상위 객체라는 의미가 아니다.
  - 전역 객체 자신은 어떤 객체의 프로퍼티도 아니며 객체의 계층적 구조상 표준 빌트인 객체와 호스트 객체를 프로퍼티로 소유한다는 것을 말한다.
- 전역 객체의 특징
  - 특징: 전역 객체를 생성할수 있는 생성자 함수가 제공되지 않기 때문에 개발자가 의도적으로 생성할 수 없다.
  - 특징: 전역 객체의 프로퍼티를 참조할때 window (또는 global) 를 생략할수 있다.

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

window.parseInt === parseInt; // true
```

- 특징: 전역 객체는 모든 표준 빌트인 객체를 프로퍼티로 가지고 있다.
- 특징: 자바스크립트 실행 환경(브라우저 환경 또는 Node.js 환경)에 따라 추가적으로 프로퍼티와 메서드를 갖는다.
  - 브라우저 환경에서는 클라이언트 사이드 Web API를 호스트 객체로 제공
  - Node.js 환경에서는 Node.js 고유의 API를 호스트 객체로 제공
- 특징: var 키워드로 선언한 전역 변수와 선언하지 않은 변수에 값을 할당한 암묵적 전역, 그리고 전역 함수는 전역 객체의 프로퍼티가 된다.

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

// 선언하지 않은 변수에 값을 할당한 암묵적 전역, 즉 bar는 전역 변수가 아니라 전역 객체의 프로퍼티다.
bar = 2; // window.bar = 2
console.log(window.bar); // 2

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

- 특징: let 이나 const 키워드로 선언한 전역 변수는 전역 객체의 프로퍼티가 아니다. 그러므로 window.foo와 같이 접근할 수 없다.

  > let 이나 const 키워드로 선언한 전역 변수는 보이지 않는 개념적인 블록(전역 렉시컬 환경의 선언적 환경 레코드) 내에 존재하게 된다.

```
let foo = 123;
console.log(window.foo); // undefined
```

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

- 전역 객체는 몇 가지 프로퍼티와 메서드를 가지고 있다.
  - 이 프로퍼티와 메서드는 전역 객체를 가리키는 식별자, 즉 window나 global을 생략하여 참조/호출할 수 있으므로 전역 변수와 전역 함수처럼 사용할 수 있다.

21.4.1 빌트인 전역 프로퍼티

- 빌트인 전역 프로퍼티
  - 전역 객체의 프로퍼티를 의미
  - 주로 애플리케이션 전역에서 사용하는 값을 제공
- 빌트인 전역 프로퍼티의 종류
  - Infinity 프로퍼티
    - 무한대를 나타내는 숫자값 Infinity를 갖는다.

```
// 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 숫자값이다.
console.log(typeof Infinity) ; // number
```

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

```
console.log(window.NaN); // NaN

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

  - undefined
    - undefined 프로퍼티는 원시 타입 undefined를 값으로 갖는다

```
console.log(window.undefined); // undefined

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

21.4.2 빌트인 전역 함수

- 빌트인 전역 함수(built-in global function)
  - 애플리케이션 전역에서 호출할 수 있는 빌트인 함수로서 전역 객체의 메서드다.
- 빌트인 전역 함수: eval

> eval 함수의 사용은 금지해야 한다.

- eval 함수는 자바스크립트 코드를 나타내는 문자열을 인수로 전달받는다.
- 전달받은 문자열 코드가 표현식이라면 eval 함수는 문자열 코드를 런타임에 평가하여 값을 생성
- 전달받은 인수가 표현식이 아닌 문이라면 eval 함수는 문자열 코드를 런타임에 실행한다.
- 문자열 코드가 여러 개의 문으로 이루어져 있다면 모든 문을 실행한다.

```
/**
* 주어진 문자열 코드를 런타임에 평가또는 실행한다.
* @param {string} code - 코드를 나타내는문자열
* @returns {*} 문자열 코드를 평가/실행한 결과값
*/

eval(code)

// 표현식인 문
eval('1 + 2;'); // -> 3
// 표현식이 아닌 문
eval('var x = 5;'); // -> undefined

// eval 함수에 의해 런타임에 변수 선언문이 실행되어 x 변수가 선언되었다.
console.log(x); // 5

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

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

- 인수로 전달받은 문자열 코드가 여러 개의 문으로 이루어져 있다면 모든 문을 실행한 다음, 마지막 결과값을 반환한다.

```
eval('1 + 2; 3 + 4;'); // -> 7
```

- eval 함수는 자신이 호출된 위치에 해당하는 기존의 스코프를 런타임에 동적으로 수정한다.

```
const 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
```

> 위 예제의 eval 함수는 새로운 x 변수를 선언하면서 foo 함수의 스코프에 선언된 x 변수를 동적으로 추가한다.

> **함수가 호출되면 런타임 이전에 먼저 함수 몸체 내부의 모든 선언문을 먼저 실행하고 그 결과를 스코프에 등록한다.**
> 따라서 위 예제의 eval 함수가 호출되는 시점에는 이미 foo 함수의 스코프가 존재한다.
> **하지만 eval 함수는 기존의 스코프를 런타임에 동적으로 수정한다.**
> 그리고 eval 함수에 전달된 코드는 이미 그 위치에 존재하던 코드처럼 동작한다.
> 즉, eval 함수가 호출된 foo 함수의 스코프에서 실행된다.

- 단, strict mode(엄격 모드)에서 eval 함수는 기존의 스코프를 수정하지 않고 eval 함수 자신의 자체적인 스코프를 생성한다.

```
const x = 1;
function foo () {
  'use strict';

  // strict mode에서 eval 함수는 기존의 스코프를 수정하지 않고 eval 함수 자신의 자체적인 스코프를
  eval('var x = 2; console.log(x);'); // 2
  console.log(x); // 1
}
foo();
console.log(x); // 1
```

- 또한 인수로 전달받은 문자열 코드가 let, const 키워드를 사용한 변수 선언문이라면 암묵적으로 strict mode가 적용된다.

```
const x = 1;

function foo() {
  eval('var x = 2; console.log(x);'); // 2
  // let, const 키워드를 사용한 변수 선언문은 strict mode가 적용된다.
  eval('const x = 3; console.log(x);'); // 3
  console.log(x); // 2
}
foo();
console.log(x); // 1
```

- eval 함수의 사용은 금지해야 한다.
  - eval 함수를 통해 사용자로부터 입력받은 콘텐츠(untrusted data)를 실행하는 것은 보안에 매우 취약하다.
  - 또한 eval 함수를 통해 실행되는 코드는 자바스크립트 엔진에 의해 최적화가 수행되지 않으므로 일반적인 코드 실행에 비해 처리 속도가 느리다.

- 빌트인 전역 함수: isFinite
  - 전달받은 인수가 정상적인 유한수인지 검사하여 유한수이면 true를 반환, 무한수이면 false를 반환
  - 인수의 타입이 숫자가 아닌 경우, 숫자로 타입을 변환한 후 검사를 수행, 이때 인수가 NaN으로 평가되는 값이라면 false를 반환

```
/**
* 전달받은 인수가 유한수인지 확인하고 그 결과를 반환한다.
* @param {number} testValue - 검사 대상 값
* @returns {boolean} 유한수 여부 확인 결과
*/
isFinite(testValue)


// 인수가 유한수이면 true를 반환한다.
isFinite(0);            // - true
isFinite(2e64);         // - true
isFinite('10');         // - true: '10' -> 10
isFinite(null);         // - true: null -> 0

// 인수가 무한수 또는 NaN으로 평가되는 값이라면 false를 반환한다.
isFinite(Infinity);     // -> false
isFinite(-Infinity);    // -> false

// 인수가 NaN으로 평가되는 값이라면 false를 반환한다.
isFinite(NaN);          // -> false
isFinite('Hello');      // -> false
isFinite('2005/12/12'); // -> false
```

- isFinite(null)은 true를 반환한다.
  - 이것은 null을 숫자로 변환하여 검사를 수행했기 때문이다. null을 숫자 타입으로 변환하면 0이 된다.

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

- 빌트인 전역 함수: isNaN
  - 전달받은 인수가 NaN인지 검사하여 그 결과를 불리언 타입으로 반환한다.
  - 전달받은 인수의 타입이 숫자가 아닌 경우 숫자로 타입을 변환한 후 검사를 수행한다.

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


// 숫자
isNaN(NaN);         // -> true
isNaN(10);          // -> false

// 문자열
isNaN('blabla');    // -> true: 'blabla' => NaN
isNaN('10');        // -> false: '10' => 10
isNaN('10.12');     // -> false: '10.12' => 10.12
isNaN('');          // -> false: '' => 0
isNaN(' ');         // -> false: ' ' => 0

// 불리언
isNaN(true);        // -> false: true -> 1
isNaN(null);        // -> false: null -> 0

// undefined
isNaN(undefined);   // -> true: undefined => NaN

// 객체
isNaN({});          // -> true: {} => NaN

// date
isNaN(new Date()); // -> false: new Date() => Number
isNaN(new Date().toString()); // -> true: String => NaN
```

- 빌트인 전역 함수: parseFloat
  - 전달받은 문자열 인수를 부동소수점 숫자(floating point number), 즉 실수로 해석(parsing)하여 반환한다.

```
/**
* 전달받은 문자열 인수를 실수로 해석하여 반환한다.
* @param {string} string - 변환 대상 값
* @returns {number} 변환 결과
*/
parseFloat(string)


// 문자열을 실수로 해석하여 반환한다.
parseFloat('3.14');         // -> 3.14
parseFloat('10.00');        // -> 10

// 공백으로 구분된 문자열은 첫 번째 문자열만 변환한다.
parseFloat('34 45 66');     // -> 34
parseFloat('40 years');     // -> 40

// 첫 번째 문자열을 숫자로 변환할 수 없다면 NaN을 반환한다.
parseFloat('He was 40');    // -> NaN

// 앞뒤 공백은 무시된다.
parseFloat(' 60 '); // -> 60
```

- 빌트인 전역 함수: parseInt
  - 전달받은 문자열 인수를 정수(integer)로 해석(parsing)하여 반환한다.

```
/**
* 전달받은 문자열 인수를 정수로 해석하여 반환한다.
* @param {string} string - 변환 대상 값
* @param {number} [radix] - 진법을 나타내는 기수(2 ~ 36, 기본값 10)
* @returns {number} 변환 결과
*/
parseInt(string, radix);


// 문자열을 정수로 해석하여 반환한다.
parseInt('10');     // -> 10
parseInt('10.123'); // -> 10
```

  - 전달받은 인수가 문자열이 아니면 문자열로 변환한 다음, 정수로 해석하여 반환한다.

```
parseInt(10);       // -> 10
parseInt(10.123);   // -> 10
```

  - 두 번째 인수로 진법을 나타내는 기수(2 ~ 36)를 전달할수 있다.
    - 기수를 지정하면 첫 번째 인수로 전달된 문자열을 해당 기수의 숫자로 해석하여 반환한다.
    - 이때 반환값은 언제나 10진수다.
    - 기수를 생략하면 첫 번째 인수로 전달된 문자열을 10진수로 해석하여 반환한다.

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

- 참고로 기수를 지정하여 10진수 숫자를 해당 기수의 문자열로 변환하여 반환하고 싶을 때는 Number.prototype.toString 메서드를 사용한다.

```
const x = 15;

// 10진수 15를 2진수로 변환하여 그 결과를 문자열로 반환한다.
x.toString(2);                  // -> '1111'
// 문자열 '1111'을 2진수로 해석하고 그 결과를 10진수 정수로 반환한다.
parseInt(x.toString(2), 2);     // -> 15

// 10진수 15를 8진수로 변환하여 그 결과를 문자열로 반환한다.
x.toString(8);                  // -> '17'
// 문자열 '17'을 8진수로 해석하고 그 결과를 10진수 정수로 반환한다.
parseInt(x.toString(8), 8);     // -> 15

// 10진수 15를 16진수로 변환하여 그 결과를 문자열로 반환한다.
x.toString(16);                 // -> 'f'
// 문자열 'f'를 16진수로 해석하고 그 결과를 10진수 정수로 반환한다.
parseInt(x.toString(16), 16);   // -> 15

// 숫자값을 문자열로 변환한다.
x.toString();                   // -> '15'
// 문자열 '15'믈 10진수로 해석하고 그 결과를 10진수 정수로 반환한다.
parseInt(x.toString());         // -> 15
```

- 두 번째 인수로 진법을 나타내는 기수를 지정하지 않더라도 첫 번째 인수로 전달된 문자열이 "0x" 또는 "0X"로 시작하는 16진수 리터럴이라면 16진수로 해석하여 10진수 정수로 반환한다.

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

- 하지만 2진수 리터럴과 8진수 리터럴은 제대로 해석하지 못한다.

```
// 2진수 리터럴(0b로 시작)은 제대로 해석하지 못한다. 0 이후가 무시된다.
parseInt('0b10'); // -> 0
// 8진수 리터럴(ES6에서 도입. 0o로 시작)은 제대로 해석하지 못한다. 0 이후가 무시된다.
parseInt('0o10'); // -> 0
```

- ES5 이전까지는 비록 사용을 금지하고는 있었지만 "0"으로 시작하는 숫자를 8진수로 해석했다.
- ES6부터는 "0"으로 시작하는 숫자를 8진수로 해석하지 않고 10진수로 해석한다.
- 따라서 문자열을 8진수로 해석하려면 지수를 반드시 지정해야 한다.

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

- 첫 번째 인수로 전달한 믄자열의 첫 번째 문자가 해당 지수의 숫자로 변환될 수 없다면 NaN을 반환한다.

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

- 하지만 첫 번째 인수로 전달한 문자열의 두 번째 문자부터
- 해당 진수를 나타내는 숫자가 아닌 문자(예를 들어 2 진수의 경우 2)와 마주치면
- 이 문자와 계속되는 문자들은 전부 무시되며 해석된 정수값만 반환한다.

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

- 첫 번째 인수로 전달한 문자열에 공백이 있다면 첫 번째 문자열만 해석히여 반환하며 앞뒤 공백은 무시된다.
  - 만일 첫 번째 문자열을 숫자로 해석할 수 없는 경우 NaN을 반환한다.

```
// 공백으로 구분된 문자열은 첫 번째 문자열만 변환한다.
parseInt('34 45 66');  // -> 34
parseInt('40 years');  // -> 40
// 첫 번째 문자열을 숫자로 변환할 수 없다면 WaM을 반환한다.
parseInt('He was 40'); // -> NaN
// 앞뒤 공백은 무시된다.
parseInt(' 60 ');      // -> 60
```

- 빌트인 전역 함수: encodeURI / decodeURI
  - encodeURI 함수
    - 완전한 URI(Uniform Resource Identifier)를 문자열로 전달받아 이스케이프 처리를 위해 인코딩한다.
  - 단어 설명
    - URI

        - 인터넷에 있는 자원을 나타내는 유일한 주소를 말한다. URI의 하위개념으로 URL, URN이 있다.
    - https://www.mydomain.com:80/docs/search?category=javascript&lang=ko#intro
        - URI : https://www.mydomain.com:80/docs/search?category=javascript&lang=ko#intro
        - URL : https://www.mydomain.com:80/docs/search
        - URN : www.mydomain.com:80/docs/search?category=javascript&lang=ko#intro
        - Scheme(Protoco) : https
        - Host(Domain) : www.mydomain.com
        - Port : 80
        - Path : docs/search
        - Query(Query String) : category=javascript&lang=ko
            - Fragment : intro
    - 인코딩
        - URI의 문자들을 이스케이프 처리하는 것을 의미한다.
    - 이스케이프 처리
        - 네트워크를 통해 정보를 공유할 때 어떤 시스템에서도 읽을 수 있는 아스키 문자 셋으로 변환하는 것이다.
        - 단, 알파벳, 0~9의 숫자, - _ . ! ~ * ' ( ) 문자는 이스케이프 처리에서 제외된다.
        - UTF-8 특수 문자의 경우 1문자당 1~3바이트 UTF-8 한글 표현의 경우 1문자당 3바이트다.
            - 예를 들어. 특수 문자인 공백 문자는 %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 특수 문자의 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 수있는 문자(<, >)를 이스케이프 처리하여 야기될 수 있는 문제를 예방하기 위해 이스케이프 처리가 필요하다.

- encodeURI 함수 예제 코드

```
/**
* 완전한 URI를 문자열로 전달받아 이스케이프 처리를 위해 인코딩한다.
* @param {string} uri - 완전한 URI
* @returns {string} 인코딩된 URI
*/
encodeURI(uri)


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

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

- decodeURI 함수
  - 인코딩된 IJRI를 인수로 전달받아 이스케이프 처리 이전으로 디코딩한다.

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


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

// encodeURI 함수는 완전한 URI를 전달받아 이스케이프 처리를 위해 인코딩한다.
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 URI을 전달빋아 이스케이프 처리 이전으로 디코딩한다.
const dec = decodeURI(enc);
console.log(dec);
// http://example.com?name=이웅모&job=programmer&teacher
```

- 빌트인 전역 함수: encodeURIComponent / decodeURIComponent
  - encodeURIComponent 함수
    - URI 구성 요소(component)를 인수로 전달받아 인코딩한다.
  - decodeURIComponent 함수
    - 매개변수로 전달된 URI 구성 요소를 디코딩한다.

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

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

- encodeURIComponent 함수는 인수로 전달된 문자열을 URI의 구성요소인 쿼리 스트링의 일부로 간주한다. 따라서 쿼리 스트링 구분자로 사용되는 =, ?, &까지 인코딩한다.
- 반면 encodeURI 함수는 매개변수로 전달된 문자열을 완전한 URI 전체라고 간주한다. 따라서 쿼리 스트링 구분자로 사용되는 =, ?, &은 인코딩하지 않는다.

```
// URI 의 쿼리 스트링
const uriComp = 'name=이웅모&job=programmer&teacher';

// encodeURIComponent 함수는 인수로 전달받은 문자열을 URI의 구성요소인 쿼리 스트링의 일부로
// 따라서 쿼리 스트링 구분자로 사용되는 =, ?, &까지 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(dnc);
// 이웅모&job=programmer&teacher

// encodeURI 함수는 인수로 전달받은 문자열을 완전한 URI 로 간주한다.
// 따라서 쿼리 스트링 구분자로 사용되는 =, ?, &클 인코딩하지 않는다.
enc = encodeURI(uriComp);
console.log(enc);
// name=%EC%9D%B4%EC%9B%85%EB%AA%A8&job=programmer&teacher

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

21.4.3 암묵적 전역

먼저 다음 예제를 살펴보자.

```
var x = 10; // 전역 변수

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

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

> foo 함수 내의 y는 선언하지 않은 식별자다.
> 따라서 y = 20이 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 y 변수에 값을 할당하기 위해 먼저 스코프 체인을 통해 선언된 변수인지 확인한다.
> 이때 foo 함수의 스코프와 전역 스코프 어디에서도 y 변수의 선언을 찾을 수 없으므로 참조에러가 발생한다.
> 하지만 자바스크립트 엔진은 y = 20을 window.y = 20으로 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 선언 없이 단지 전역 객체의 프로퍼티로 추가되었을 뿐이다.

> 따라서 y는 변수가 아니므로 변수 호이스팅이 발생하지 않는다.
> ```js
> // 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 () {
  // 선언하지 않은 식별자에 값을 할당
  y = 20; // window.y = 20;
}
foo();

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

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

```
var x = 10; // 전역 변수

function foo () {
  // 선언하지 않은 식별자에 값을 할당
  y = 20; // window.y = 20;
  console.log(x + y);
}
foo(); // 30

console.log(window.x); // 10
console.log(window.y); // 20

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

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