모던 자바스크립트
Deep Dive

21장_빌트인 객체

객체의 종류

1. 표준 빌트인 객체(standard built-in objects/native objects/global objects)
- ECMA 스크립트 사양에 정의된 객체
- 자바스크립트 실행 환경(브라우저, Node.js)과 관계없이 언제나 사용 가능
- 전역 객체의 프로퍼티에서 제공하므로 별도의 선언 없이 전역 변수처럼 사용 가능

2. 호스트 객체(host object)
- EMCA 스크립트 사양에 정의되지 않은 객체
- 자바스크립트 실행 환경에서 추가로 제공하는 객체
- 브라우저 환경에서는 DOM, BOM, XMLHttpRequest, fetch, requestAnimationFrame, SVG, Web Storage, Web Component, Web Worker, 타이머 함수(setTimeout, setInterval)와 같은 클라이언트 사이트 API를 호스트 객체로 제공. Node.js 환경에서는 Node.js 고유의 API를 호스트 객체로 제공

3. 사용자 정의 객체(user defined object)

표준 빌트인 객체

- 자바스크립트는 40여개의 표준 빌트인 객체 제공
Object, String, Number, Boolean, Symbol, Date, Math, RegExp, Array, Map/Set, WeakMap/WeakSet, Function, Promise, Reflect, proxy, JSON, Error

- 인스턴스를 생성할 수 있는 생성자 함수 객체와 생성자 함수가 아닌 객체로 나뉨

- 생성자 함수 객체(내부 메서드 [[construct]]존재)인 표준 빌트인 객체는 프로포타입 메서드 및 정적 메서드 제공
String, Number, Boolean, Function, Array, Date

- 생성자 함수 객체가 아닌(내부 메서드 [[construct]]존재X) 표준 빌트인 객체는 정적 메서드만 제공
Math, JSON

```
const strObj = new String('Lee');

console.log(Object.getOwnPropertyDescriptors(String));
console.log(String.fromCharCode('100'));
console.log(Object.getOwnPropertyDescriptors(Object.getPrototypeOf(strObj)));

function sum(a, b) {}
sum.c = 1;
console.log(Object.getOwnPropertyDescriptors(sum));
```

표준 빌트인 객체

정적 메서드 존재 이유
- **클래스 또는 생성자 함수를 하나의 네임스페이스로 사용**
- 이름 충돌 가능성을 낮추고, 관련 함수들의 구조화 가능
- 이 같은 이유로 애플리케이션 전역에서 사용할 **유틸리티 함수를** (전역 함수로 정의하지 않고) **메서드로 구조화** 할 때 유용

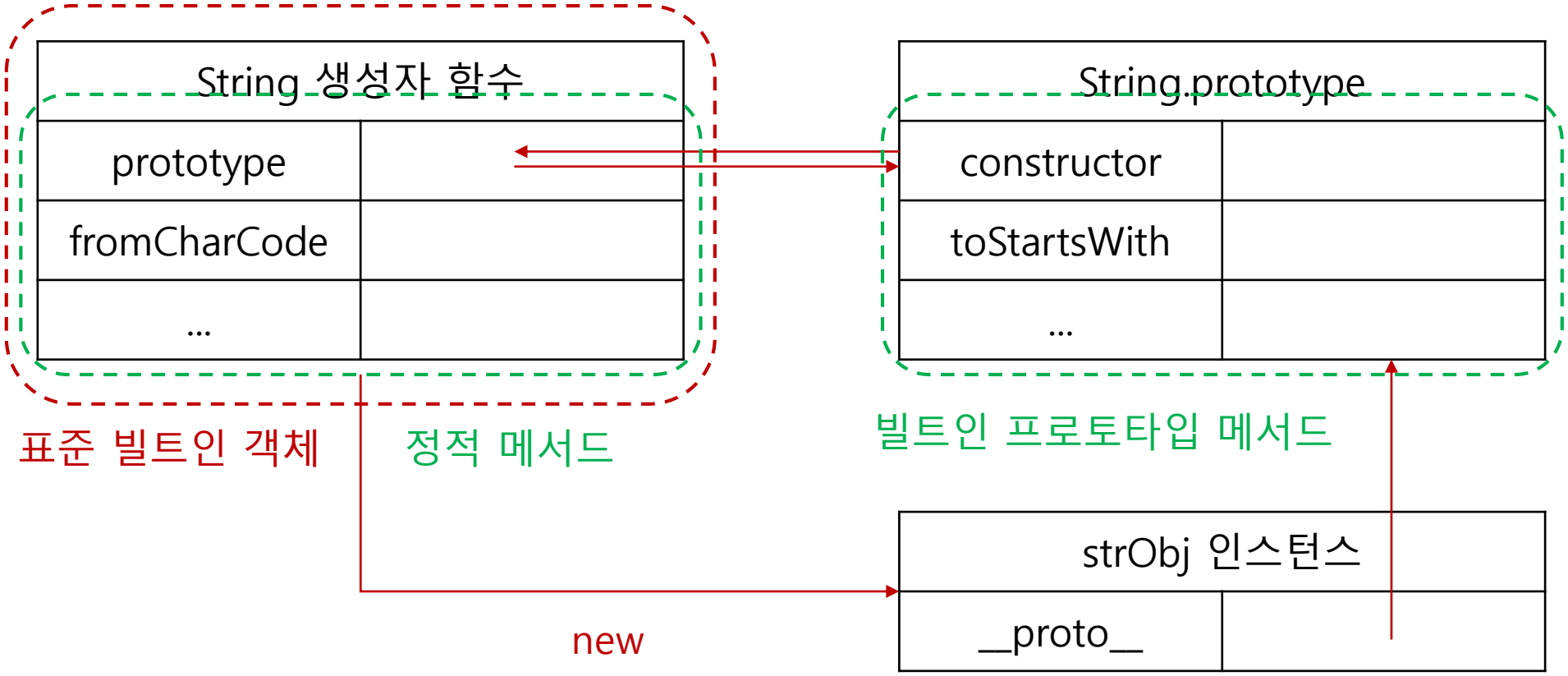
```
Math.max(1, 2, 3); // 1, 2, 3
Number.isNaN(NaN); // true
JSON.stringify({ a : 1 }); // "{"a":1}"
Object.is({},{}); // false;
Reflect.has({a : 1}, 'a'); //true
```

표준 빌트인 객체

```
1   const strObj  = new String('Lee'); // String {'Lee'}
2   console.log(typeof strObj); // object
3
4   const numObj = new Number(123); // Number {123}
5   console.log(typeof numObj); // object
6
7   const boolObj = new Boolean(true); // Boolean {true}
8   console.log(typeof boolObj); // object
9
10  const func = new Function('x', 'return x * x'); // f anonymous(x)
11  console.log(typeof func); // function. 함수 객체
12
13  const arr = new Array(1,2,3); // (3) [1,2,3]
14  console.log(typeof arr); // object
15
16  const regExp = new RegExp(/ab+c/i); // /ab+c/i
17  console.log(typeof regExp); // object
18
19  const date = new Date(); // 금일 날짜
20  console.log(typeof date); // object
```

표준 빌트인 객체

```
const strObj = new String('Lee');

console.log(Object.getOwnPropertyDescriptors(String));
console.log(Object.getOwnPropertyDescriptors(Object.getPrototypeOf(strObj)));
```



표준 빌트인 객체

모르겠음

```
const strObj = new String('Lee');
const str = 'Kim';
console.log(strObj); // [String:'Lee']
console.log(str); // 'Kim'
console.log(strObj + str); // LeeKim
```

원시값과 래퍼 객체(Warpper Object)

문자열, 숫자, 불리언 원시값이 있는데, String, Number, Boolean 등의 표준 빌트인 생성자 함수가 존재하는 이유는?

원시값을 객체처럼 사용하기 위해서
원시값을 유용하게 사용하기 위해서(개인적인 생각)

원시값은 객체가 아니므로 프로퍼티나 메서드를 가질 수 없음에도, 마치 객체처럼 동작이 가능

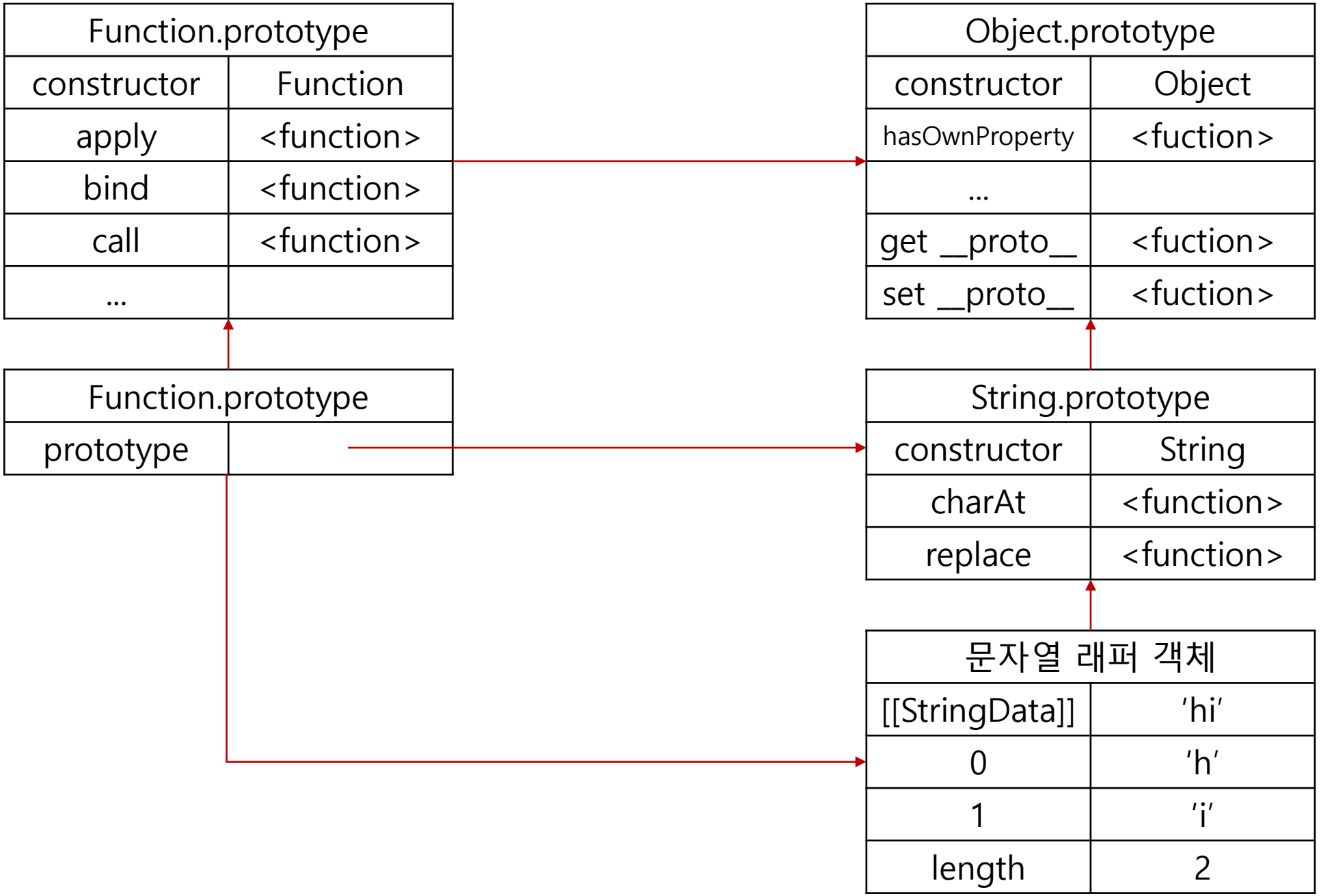
```
const str = 'hello';

// 원시 타입인 문자열이 프로퍼티와 메서드를 갖는 객체처럼 동작
console.log(str.length); // 5
console.log(str.toUpperCase()); // HELLO
```

원시값인 문자열, 숫자, 불리언에 대해 마치 객체처럼 접근하면, 자바스크립트 엔진은 일시적으로 원시값과 연관된 객체로 변환해준다. 이때 생성되는 객체를 **래퍼 객체**라고 한다.

원시값과 래퍼 객체(Warpper Object)

문자열에 대해 마침표 표기법(대괄호 표기법은 불가능)으로 접근하면, 그 순간 래퍼 객체인 String 생성자 함수의 인스턴스(래퍼 객체)가 생성되고, 문자열은 래퍼 객체의 [[StringData]] 내부 슬롯에 할당됨

이때 인스턴스는 빌트인 프로토타입 메서드 사용 가능


Function.prototype
constructor
Function
apply
<function>
bind
<function>
call
<function>
...
Object.prototype
constructor
Object
hasOwnProperty
<fuction>
...
get __proto__
<fuction>
set __proto__
<fuction>
Function.prototype
prototype
String.prototype
constructor
String
charAt
<function>
replace
<function>
문자열 래퍼 객체
[[StringData]]
'hi'
0
'h'
1
'i'
length
2

원시값과 래퍼 객체(Warpper Object)

래퍼 객체 코드의 구체적인 동작 과정

```
const str = 'hello';
// 1
// str은 암묵적으로 생성된 래퍼 객체를 가리키고 있음
// str의 Hello는 래퍼 객체의 [[StringData]] 내부 슬롯에 저장
// str은 name 프로퍼티 동적으로 추가
str.name = 'Lee';

// 2
// str은 다시 원래의 문자열 [[StringData]] 내부 슬롯의 원시값을 가리킴
// 1에서 생성된 래퍼 객체는 아무도 참조하지 않기 때문에 가비지 컬렉션의 대상

// 3
// str은 암묵적으로 새롭게 생성된 래퍼 객체(1의 래퍼 객체와는 다른)를 가리킴
// 래퍼 객체에는 name 프로퍼티가 존재하지 않음
console.log(str.name); // undefined

// 4
// str은 다시 원래의 문자열 [[StringData]] 내부 슬롯의 원시값을 가리킴
// 1에서 생성된 래퍼 객체는 아무도 참조하지 않기 때문에 가비지 컬렉션의 대상
console.log(typeof str, str);; // string, hello
```

추가

```
const str = '123';
console.log(Object.getOwnPropertyDescriptors(Object.getPrototypeOf(str)));
```

str은 객체가 아니므로 프로토타입이 없어야 함. 하지만 위 console 결과는 String 표준 빌트인 생성자 함수가 생성한 인스턴스의 프로토타입 메서드들을 전부 출력함. 책에서는 마침표 표기법으로 접근하는 경우만 언급하였으나, 이것 역시도 **래퍼 객체로 동작한 것으로 추측함**

```
const str = new String('123');
console.log(Object.getOwnPropertyDescriptors(str));
```

위 코드를 실행하면 str문자열의 내부 프로퍼티가 나옴. 이는 **래퍼 객체의 프로퍼티로 추측함**

원시값과 래퍼 객체(Warpper Object)

숫자 값도 마찬가지

```
1  // 표준 빌트인 객체로 인스턴스 생성
2  const numObj = new Number(1.5);
3
4  // toFixed는 빌트인 프로토타입 메서드
5  console.log(numObj.toFixed()); // 2
6
7  // isInteger는 정적 메서드
8  console.log(Number.isInteger(numObj)); // False
```

마침표 표기법으로 접근하는 경우 래퍼 객체가 생성되고, [[NumberData]] 내부 슬롯에 할당되는 과정을 거치고, 참조하지 않으면 가비지 컬렉션의 대상이 됨

끝맺음

ES6에서 도입된 Symbol도 래퍼 객체 생성

문자열, 숫자, 불리언, 심벌은 래퍼 객체에 의해 마치 객체처럼 사용할 수 있음.
이때 String, Number, Boolean, Symbol의 프로토타입 메서드 또는 프로퍼티 참조가 가능

그러므로 **String, Number, Boolean생성자 함수를 new 연산자와 함께 호출하여 문자열, 숫자, 불리언 인스턴스를 생성할 필요가 없으며 권장하지도 않음**. Symbol은 생성자 함수가 아니므로 논외로 함

이외에 null과 undefined는 래퍼 객체를 생성하지 않으므로, 객체처럼 사용하면 에러가 발생함