프로토타입

Java, C++과 같은 Class 기반 객체지향 프로그래밍 언어와 달리 **JavaScript 는 프로토타입 기반 객체지향 프로그래밍 언어**입니다. 프로토타입 기반 언어이기 때문에 해당 개념을 잘 이해하고 있어야 JavaScript의 동작 원래를 이해할 수 있습니다

Class 기반 객체지향 프로그래밍 언어는 객체 생성 이전에 클래스를 정의하고 이를 통해 객체(인스턴스)를 생성합니다. 하지만 **프로토타입 기반 객체지향 프로그래밍 언어는 클래스 없이도 객체를 생성할 수 있습니다.**

예를 들어 Java 에선 아래와 같이 객체를 생성합니다.

```
public class Student { // 클래스를 정의하고
    private int age;
    private String name;

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

Student student = new Student("sally", 24); // 이를 통해 객체 생
```

하지만 JavaScript는 다르죠. 아래와 같이 바로 객체를 생성할 수 있습니다.

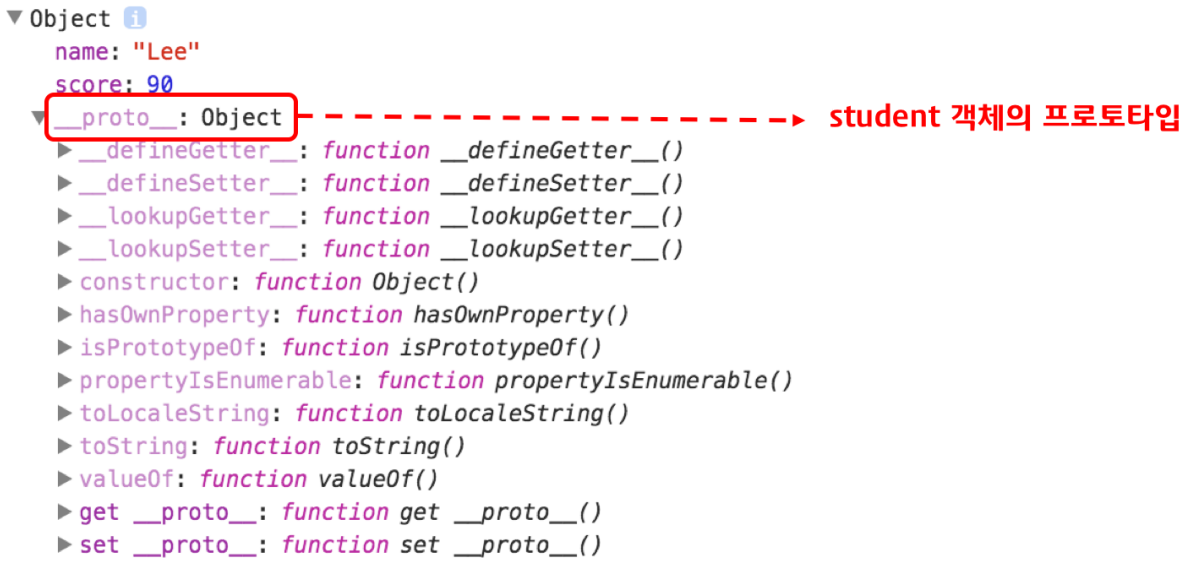
```
let student = { name: 'Lee', score: 90 }
```

JavaScript의 모든 객체는 자신의 부모 역할을 담당하는 객체와 연결되어 있습니다. 그리고 이건 마치 객체 지향의 상속 개념과 같이 부모 객체의 속성이나 메서드를 상속받아 사용할 수 있게 합니다. 이때, 이러한 부모 객체를 **프로토타입 객체** 또는 줄여서 **프로토타입** 이라고 합니다.

프로토타입 객체는 생성자 함수에 의해 생성된 각각의 객체에 **공유 프로퍼티**를 제고하기 위해 사용합니다.

```
let student = { name: 'Lee', score: 90 }

// student 객체에 hasOwnProperty라는 함수는 없지만 실행은 됩니다.
console.log(student.hasOwnProperty('name')) // true
console.dir(student); // student를 찍어보면 아래와 같이 나옵니다.
```



ECMAScript 공식 스펙에서는 **JavaScript의 모든 객체는 [[Prototype]] 이라는 인터널 슬롯을 가집니다. [[Prototype]]의 값은 null 또는 객체이며 상속을 구현하는데 사용됩니다. [[Prototype]] 객체의 데이터 프로퍼티는 get 액세스를 통해 자식 객체의 프로퍼티처럼 사용할 수 있지만 set 의 접근은 허용되지 않습니다.** 라고 되어 있습니다.

- Internal Slot 은 JS engine 의 내부 로직으로 이중 대괄호 [[...]] 로 감싼 이름들입니다. Javascript는 이들에 대한 직접적인 접근이나 호출 방법을 제공하지 않지만 [[Prototype]] 과 같은 Internal slot은 __proto__ 를 통해 간접적으로 접근할 수 있는 방법을 제공합니다.

[[Prototype]] 의 값은 프로토타입 객체이며 위 이미지의 **__proto__** 프로퍼티로 접근할 수 있습니다. __proto__ 프로퍼티에 접근하면 내부적으로 Object.getPrototypeOf 가 호출되어 프로토타입 객체를 반환합니다.

위에서 student 객체는 __proto__ 프로퍼티로 자신의 부모 객체(프로토타입 객체)인 Object.prototype 을 가리키고 있습니다.

```
let student = {
  name: 'Lee',
  score: 90
}

console.log(student.__proto__ === Object.prototype); // true
```

- **[[Prototype]] === __proto__ = 자신의 부모 역할을 하는 객체**
- **prototype 속성 = 함수에서만 만들어지는거고, 이 생성자 함수로 인해 만들어진 객체의 부모 객체**

[[Prototype]] vs prototype 프로퍼티

모든 객체는 자신의 프로토타입 객체를 가리키는 [[Prototype]] 인터널 슬롯을 갖으며 상속을 위해 사용됩니다. 그리고 JavaScript 에선 함수 또한 객체이므로 [[Prototype]] 을 갖습니다. 그런데 **함수 객체는 일반 객체와 달리 prototype 프로퍼티도 소유**하게 됩니다.

> 여기서 주의해야 할 것은 prototype 프로퍼티는 [[Prototype]] 과는 다르다는 것입니다. 둘 모두 프로토타입 객체를 가리키지만 관점의 차이가 있습니다.

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

let foo = new Person('Lee');

let bar = { name: 'Lee' }

console.dir(Person) // prototype 프로퍼티가 있습니다.
console.dir(foo) // prototype 프로퍼티가 없습니다.
```

실제 실행해 보면 결과값을 확인하실 수 있습니다.

- **[[Prototype]]**
  - 함수를 포함한 모든 객체가 가지고 있는 Internal Slot 입니다.
  - 객체의 입장에서 자신의 부모 역할을 하는 프로토타입 객체를 가리키는 함수 객체의 경우 Function.prototype 을 가집니다. 그 이유에 대해선 조금 있다 얘기하도록 하겠습니다.

```
console.log(Person.__proto__ === Function.prototype); // t
```

- **prototype 프로퍼티**
  - 함수 객체만 가지고 있는 프로퍼티입니다.
  - **함수 객체가 생성자로 사용될 때 이 함수를 통해 생성될 객체의 부모 역할을 하는 객체(프로토타입 객체) 를 가리킵니다.**

```
console.log(Person.prototype === foo.__proto__); // true
```

constructor 프로퍼티

프로토타입 객체는 constructor 프로퍼티를 갖습니다. 이 constructor 프로퍼티는 객체의 입장에서 **자신을 생성한 객체**를 가리킵니다.

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

let foo = new Person('Lee');
```

위를 예로 들어 Person() 생성자 함수에 의해 생성된 객체를 foo라고 하겠습니다. 이 foo 객체를 생성한 객체는 Person() 이라는 생성자 함수입니다. 이때 foo 객체 입장에서 자신을 생성한 객체는 Person() 이라는 생성자 함수이며, foo 객체의 프로토타입 객체는 Person.prototype 입니다. 따라서 프로토타입 객체인 **Person.prototype 의 constructor 프로퍼티는 Person() 생성자 함수를 가리킵니다.**

```
// Person.prototype 이라는 객체는 Person이 생성했기 때문에
// Person.prototype 의 생성자는 Person입니다.
console.log(Person.prototype.constructor === Person);

// foo 객체를 생성한 객체는 Person 이라는 생성자 함수입니다.
console.log(foo.constructor === Person);

// Person 생성자 함수를 생성한 객체는 Function 이라는 생성자 함수입니다
console.log(Person.constructor === Function);
```

Prototype chain

JavaScript 는 특정 객체의 프로퍼티나 메서드에 접근하려고 할 때 해당 객체에 접근하려는 프로퍼티 또는 메서드가 없다면 [[Prototype]] 이 가리키는 링크를 따라 자신의 부모 역할

을 하는 프로토타입 객체의 프로퍼티나 메서드를 차례대로 검색합니다. 그리고 이것을 **프로토타입 체인**이라고 합니다.

```
let student = {
  name: 'Lee',
  score: 90
}

// 위의 예제에서 봤듯이 student에는 hasOwnPerperty라는 함수가 없기 때문
// Object.prototype.hasOwnProperty() 를 실행합니다.
// hasOwnProperty는 프로토타입 객체의 기본 메서드입니다.
console.log(student.hasOwnProperty('name')); // true
```

객체 리터럴 방식으로 생성된 객체의 프로토타입 체인

객체의 생성 방법에는 3가지가 있습니다.

- 객체 리터럴 : 이제까지 예시처럼 일반적인 객체 생성 방법을 뜻합니다.
- 생성자 함수
- Object() 생성자 함수

객체 리터럴 방식으로 생성된 객체는 결국 내장 함수인 Object() 생성자 함수로 객체를 생성하는 것을 단순화시킨 것입니다. JavaScript 엔진은 객체 리터럴로 객체를 생성하는 코드를 만나면 내부적으로 Object() 생성자 함수를 사용해 객체를 생성합니다.

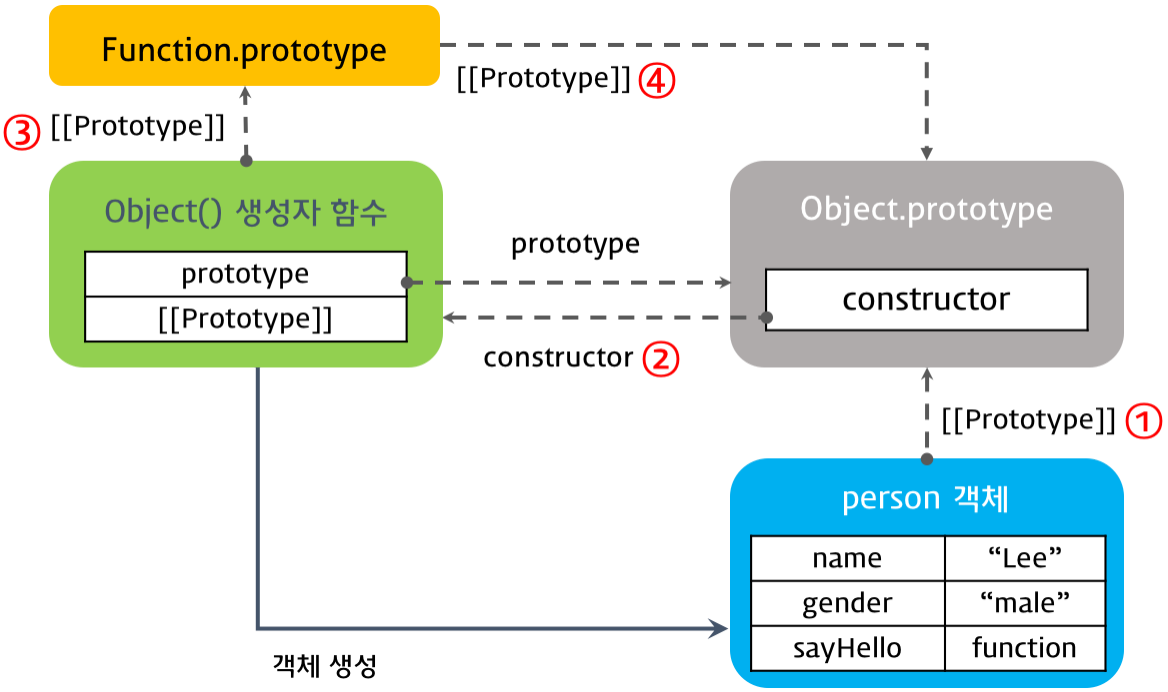
물론 Object() 생성자 함수는 함수입니다. 따라서 함수 객체인 Object() 생성자 함수는 일반 객체와 달리 prototype 프로퍼티가 있습니다.

- **prototype 프로퍼티는 함수 객체가 생성자로 사용될 때 이 함수를 통해 생성된 객체의 부모 역할을 하는 객체, 즉 프로토타입 객체를 가리킵니다.**

- **[[Prototype]] 또한 객체의 입장에서 자신의 부모 역할을 하는 객체, 즉 프로토타입 객체를 가리킵니다.**

```
var person = {
  name: 'Lee',
  gender: 'male',
  sayHello: function(){
    console.log('Hi! my name is ' + this.name);
  }
};

console.log(person.__proto__ === Object.prototype);   // ① t
console.log(Object.prototype.constructor === Object); // ② t
console.log(Object.__proto__ === Function.prototype); // ③ t
console.log(Function.prototype.__proto__ === Object.prototype
```



결론적으로 객체 리터럴을 사용해 객체를 생성한 경우, 그 객체의 프로토타입 객체는 Object.prototype 입니다.

생성자 함수로 생성된 객체의 프로토타입 체인

생성자 함수로 객체를 생성하기 위해서는 우선 생성자 함수를 정의해야 합니다.

JavaScript에서 함수를 정의하는 방식은 3가지가 있습니다.

- 함수 선언식
- 함수 표현식
- Function() 생성자 함수

함수 표현식으로 함수를 정의할 때 함수 리터럴 방식을 사용합니다.

```
let square = function(number) {
  return number * number;
};
```

함수 선언식의 경우 JavaScript 엔진이 내부적으로 기명 함수 표현식으로 변환됩니다.

```
let square = function square(number) {
    return number * number;
};
```

결국 함수 선언식이든, 표현식이든 모두 함수 리터럴 방식을 사용하고, 함수 리터럴 방식은 Function() 생성자 함수로 함수를 생성하는 것을 단순화 시킨 것입니다.

즉, 3가지 함수 정의 방식 모두 Function 생성자 함수를 통해 함수 객체를 생성합니다. 따라서 **어떤 방식으로 함수 객체를 생성해도 모든 함수 객체의 prototype 객체는**

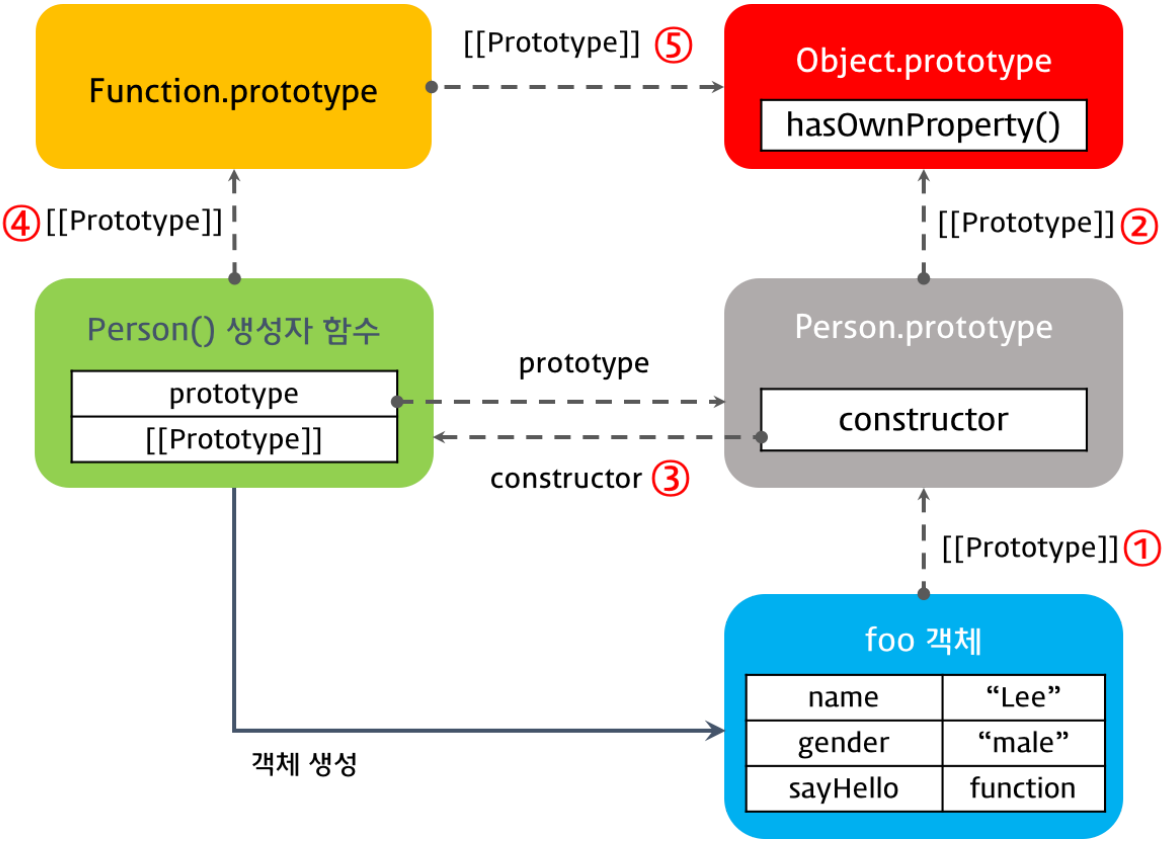
Function.prototype 입니다. 생성자 함수도 함수 객체이므로 생성자 함수의 prototype 객체는 Function.prototype 입니다.

이제 객체의 관점에서 prototype 객체를 살펴보겠습니다.

```
function Person(name, gender) {
  this.name = name;
  this.gender = gender;
  this.sayHello = function(){
    console.log('Hi! my name is ' + this.name);
  };
}

var foo = new Person('Lee', 'male');

console.log(foo.__proto__ === Person.prototype);
console.log(Person.prototype.__proto__ === Object.prototype);
console.log(Person.prototype.constructor === Person);
console.log(Person.__proto__ === Function.prototype);
console.log(Function.prototype.__proto__ === Object.prototype
```



foo 객체의 프로토타입 객체 Person.prototype 객체와 Person() 생성자 함수의 프로토타입 객체인 Function.prototype 의 프로토타입 객체는 Object.prototype 객체입니다. 이는 객체 리터럴 방식이나 생성자 함수 방식이나 결국은 모든 객체의 부모 객체인 Object.prototype 객체에서 프로토타입 체인이 끝나기 때문입니다. 이때, Object.prototype 객체를 **프로토타입 체인의 종점** 이라고 합니다.

프로토타입 객체의 확장

프로토타입 객체도 결국 객체이므로 일반 객체와 같이 프로퍼티를 추가/삭제할 수 있습니다. 그리고 이렇게 추가/삭제된 프로퍼티는 즉시 프로토타입 체인에 반영됩니다.

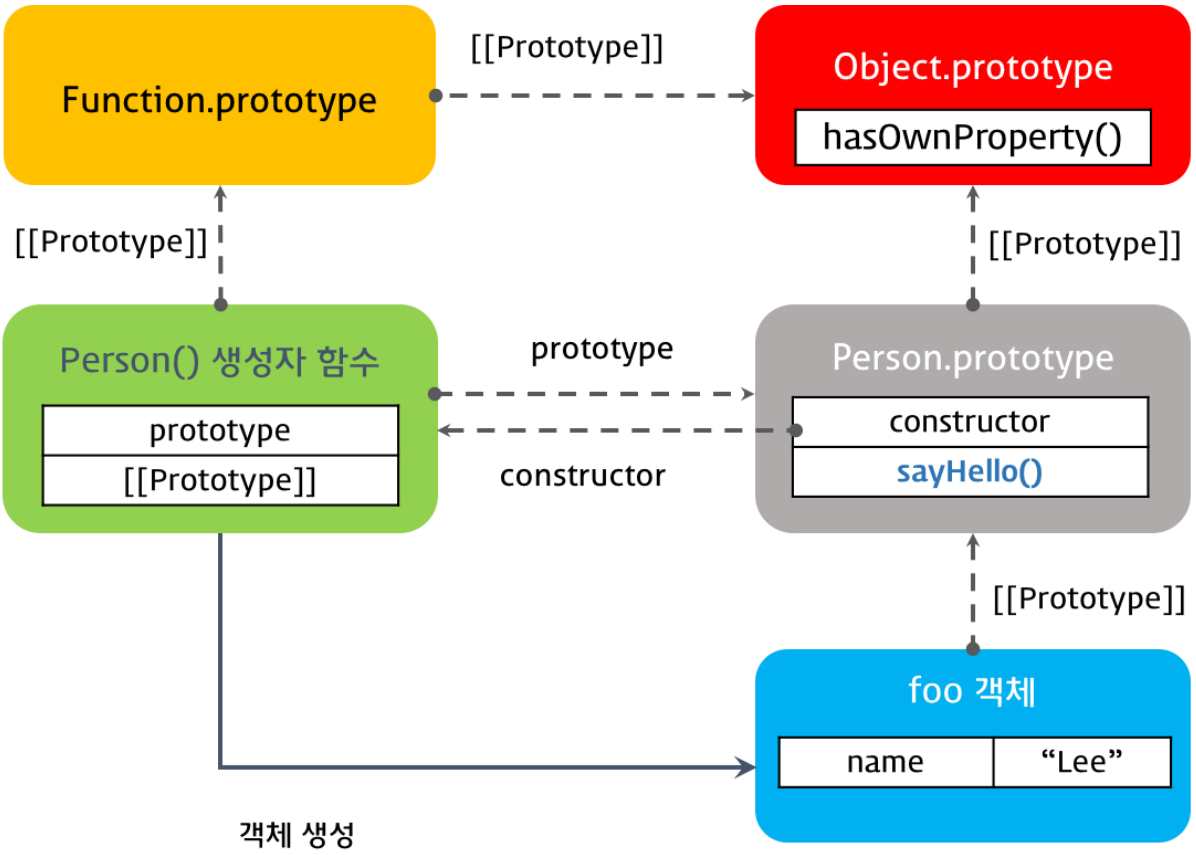
```
function Person(name) {
    this.name = name;
}
```

```
let foo = new Person('Lee');

Person.prototype.sayHello = function() {
    console.log('Hi, my name is ' + this.name);
};

foo.sayHello();
```

생성자 함수 Person은 프로토타입 객체 Person.prototype과 prototype 프로퍼티에 의해 바인딩 되어 있습니다. Person.prototype 객체는 일반 객체와 같이 프로퍼티를 추가/삭제가 가능합니다. 위의 예에서는 Person.prototype 객체에 sayHello 라는 메서드를 추가했고, sayHello는 프로토타입 체인에 반영이 됩니다. 따라서 생성자 함수 Person 에 의해 생성된 모든 객체는 프로토타입 체인에 의해 부모 객체인 Person.prototype 의 메서드를 사용할 수 있게 됩니다.



원시 타입의 확장

JavaScript에서 원시 타입(숫자, 문자열, Boolean, null, Undefined)을 제외한 모든 것은 객체입니다. 그런데 아래 예제를 보면 원시 타입인 문자열이 객체와 유사하게 동작합니다.

```
let str = 'test';
console.log(typeof str) // string
console.log(str.constructor === String) // true
console.dir(str) // test

let strObj = new String('test')
console.log(typeof strObj) // object
console.log(strObj.constructor === String) // true
console.dir(strobj)
// {0: "t", 1: "e", 2: "s", 3: "t", length: 4, __proto__: Str

console.log(str.toUpperCase()) // TEST
console.log(strObj.toUpperCase()) // TEST
```

원시 타입 문자열과 String() 생성자 함수로 생성한 문자열 객체의 타입은 분명히 다릅니다. 원시 타입은 객체가 아니므로 프로퍼티나 메서드를 가질 수 없습니다. 하지만 **원시타입으로 프로퍼티나 메서드를 호출할 때 원시 타입과 연관된 객체로 일시적으로 변환되어 프로토타입 객체를 공유**하게 됩니다. 아래 예시를 보면 확실히 이해가 되실겁니다.

원시 타입은 객체가 아니므로 프로퍼티나 메서드를 직접 추가할 수 없습니다.

```
let str = 'test';

// 에러가 발생하지 않습니다.
str.myMethod = function() {
    console.log('str.myMethod')
}
```

```
str.myMethod(); // Uncaught TypeError: str.myMethod is not a
```

하지만 String 객체의 프로토타입 객체 String.prototype에 메서드를 추가하면 원시 타입, 객체 모두 메서드를 사용할 수 있습니다.

```
let str = 'test';

String.prototype.myMethod = function () {
  return 'myMethod';
};

console.log(str.myMethod());      // myMethod
console.log('string'.myMethod()); // myMethod
console.dir(String.prototype);
```

분명 원시 타입을 가진 str 변수에는 프로퍼티나 메서드를 가질 수 없지만 String의 프로토타입 객체에 myMethod를 추가했더니 str 변수에서도 해당 메서드를 호출할 수 있게 됩니다.

String 타입 뿐만이 아니라 String.prototype, Number.prototype, Array.prototype 등 여러 원시 타입에 대한 프로토타입 객체가 있어 프로퍼티나 메서드를 할당할 수 있게 됩니다. 이러한 프로토타입 객체 또한 Object.prototype 을 프로토타입 체인에 의해 자신의 프로토타입 객체로 연결합니다.

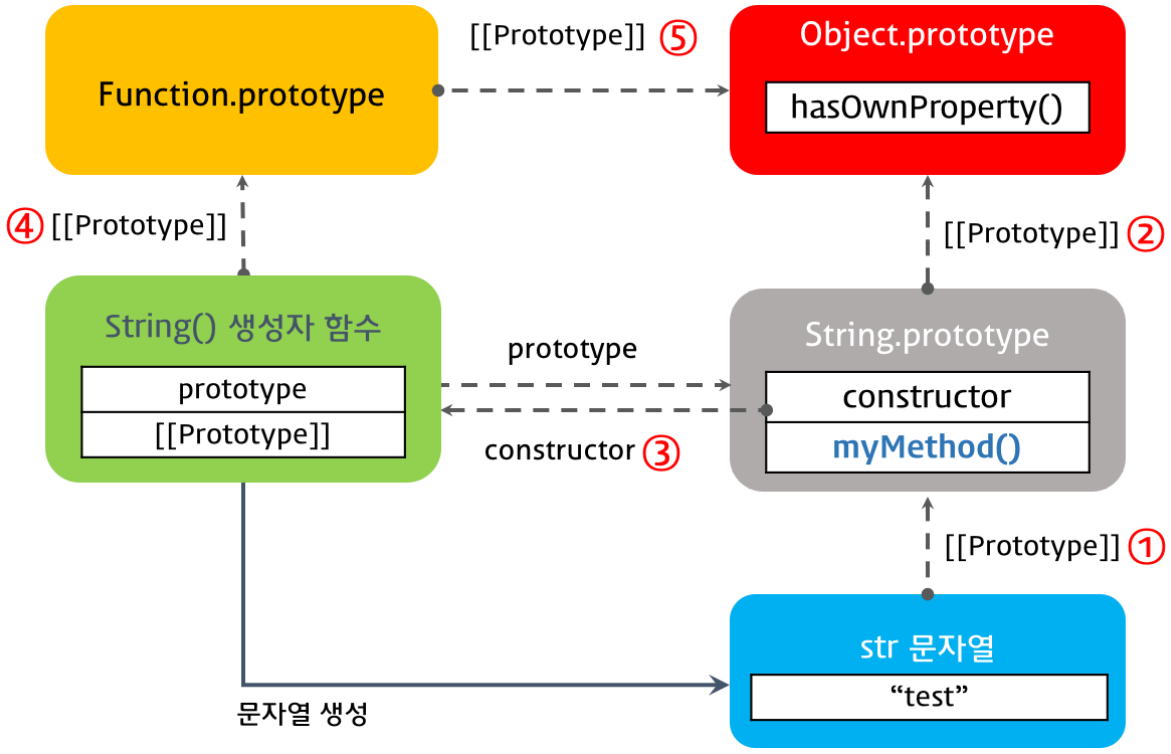
결국 요약하면 JavaScript 는 표준 내장 객체의 프로토타입 객체에 개발자가 정의한 메서드의 추가를 허용한다는 것입니다.

```
var str = 'test';

String.prototype.myMethod = function() {
  return 'myMethod';
}
```

```
console.log(str.myMethod());
console.dir(String.prototype);

console.log(str.__proto__ === String.prototype);
console.log(String.prototype.__proto__  === Object.prototype)
console.log(String.prototype.constructor === String);
console.log(String.__proto__ === Function.prototype);
console.log(Function.prototype.__proto__  === Object.prototyp
```



프로토타입 객체의 변경

객체를 생성할 때 프로토타입은 결정됩니다. 그리고 결정된 프로토타입 객체는 다른 임의의 객체로 변경할 수 있습니다. 이러한 특징을 활용해 객체의 상속 또한 구현할 수 있습니다.

이때, 주의할 것은 아래와 같습니다.

- 프로토타입 객체를 변경하면 프로토타입 객체 변경 시점 이전에 생성된 객체는 기존 프로토타입 객체를 [[Prototype]] 에 바인딩 합니다.
- 프로토타입 객체를 변경하면 프로토타입 객체 변경 시점 이후에 생성된 객체는 변경된 프로토타입 객체를 [[Prototype]] 에 바인딩 됩니다.