클래스

자바스크립트는 **프로토타입 기반(prototype-based)** 객체지향 언어다. 비록 다른 객체지향 언어들과의 차이점에 대한 논쟁이 있긴 하지만, 자바스크립트는 강력한 객체지향 프로그래밍 능력을 지니고 있다.

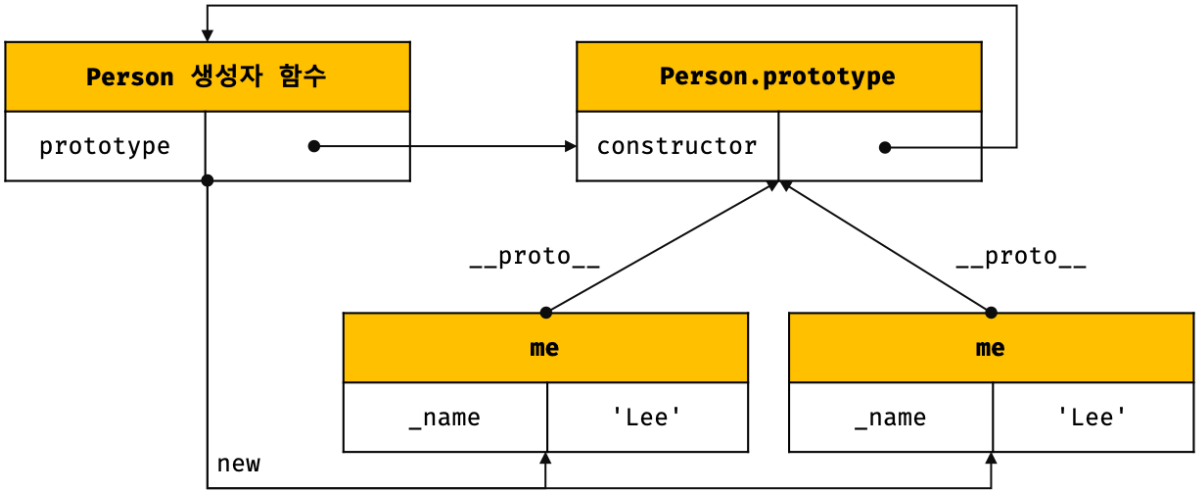
프로토타입 기반 프로그래밍은 클래스가 필요없는(class-free) 객체지향 프로그래밍 스타일로 프로토타입 체인과 클로저 등으로 객체 지향 언어의 상속, 캡슐화(정보 은닉) 등의 개념을 구현할 수 있다.

ES5에서는 생성자 함수와 프로토타입, 클로저를 사용하여 객체 지향 프로그래밍을 구현하였다.

```
// ES5
var Person = (function () {
  // Constructor
  function Person(name) {
    this._name = name;
  }

  // public method
  Person.prototype.sayHi = function () {
    console.log('Hi! ' + this._name);
  };

  // return constructor
  return Person;
}());

var me = new Person('Lee');
me.sayHi(); // Hi! Lee.
```

```
console.log(me instanceof Person); // true
```

위 예제를 프로토타입 관점에서 표현해 보면 아래와 같다.



하지만 클래스 기반 언어에 익숙한 프로그래머들은 프로토타입 기반 프로그래밍 방식이 혼란스러울 수 있으며 자바스크립트를 어렵게 느끼게하는 하나의 장벽처럼 인식되었다.

ES6의 클래스는 기존 프로토타입 기반 객체지향 프로그래밍보다 클래스 기반 언어에 익숙한 프로그래머가 보다 빠르게 학습할 수 있는 단순명료한 새로운 문법을 제시하고 있다. 그렇다고 ES6의 클래스가 기존의 프로토타입 기반 객체지향 모델을 폐지하고 새로운 객체지향 모델을 제공하는 것은 아니다. 사실 **클래스도 함수**이며 기존 프로토타입 기반 패턴의 문법적 설탕(Syntactic sugar)이라고 볼 수 있다. 다만, 클래스와 생성자 함수가 정확히 동일하게 동작하지는 않는다. 클래스가 보다 엄격하다. 따라서 클래스를 프로토타입 기반 패턴의 문법적 설탕이라고 인정하지 않는 견해도 일리가 있다.

클래스 정의 (Class Definition)

ES6 클래스는 class 키워드를 사용하여 정의한다. 앞에서 살펴본 Person 생성자 함수를 클래스로 정의해 보자.

클래스 이름은 생성자 함수와 마찬가지로 파스칼 케이스를 사용하는 것이 일반적이다. 파스칼 케이스를 사용하지 않아도 에러가 발생하지는 않는다.

```
// 클래스 선언문
class Person {
  // constructor(생성자)
  constructor(name) {
    this._name = name;
  }

  sayHi() {
    console.log(`Hi! ${this._name}`);
  }
}

// 인스턴스 생성
const me = new Person('Lee');
me.sayHi(); // Hi! Lee

console.log(me instanceof Person); // true
```

클래스는 클래스 선언문 이전에 참조할 수 없다.

```
console.log(Foo);
// ReferenceError: Cannot access 'Foo' before initializatio
n

class Foo {}
```

하지만 호이스팅이 발생하지 않는 것은 아니다. 클래스는 var 키워드로 선언한 변수처럼 호이스팅되지 않고 let, const 키워드로 선언한 변수처럼 호이스팅된다. 따라서 클래스 선언문 이전에 일시적 사각지대(Temporal Dead Zone; TDZ)에 빠지기 때문에 호이스팅이 발생하지 않는 것처럼 동작한다.

```
const Foo = '';
```

```
{
  // 호이스팅이 발생하지 않는다면 ''가 출력되어야 한다.
  console.log(Foo);
  // ReferenceError: Cannot access 'Foo' before initializat
ion
  class Foo {}
}
```

클래스 선언문도 변수 선언, 함수 정의와 마찬가지로 호이스팅이 발생한다. 호이스팅은 var, let, const, function, function*, class 키워드를 사용한 모든 선언문에 적용된다. 다시 말해, 선언문을 통해 모든 식별자(변수, 함수, 클래스 등)는 호이스팅된다. 모든 선언문은 런타임 이전에 먼저 실행되기 때문이다.

일반적이지는 않지만, 표현식으로도 클래스를 정의할 수 있다. 함수와 마찬가지로 클래스는 이름을 가질 수도 갖지 않을 수도 있다. 이때 클래스가 할당된 변수를 사용해 클래스를 생성하지 않고 기명 클래스의 클래스 이름을 사용해 클래스를 생성하면 에러가 발생한다. 이는 함수와 마찬가지로 클래스 표현식에서 사용한 클래스 이름은 외부 코드에서 접근 불가능하기 때문이다. 자세한 내용은 함수표현식(Function expression)을 참조하기 바란다.

```
// 클래스명 MyClass는 함수 표현식과 동일하게 클래스 몸체 내부에서만 유
효한 식별자이다.
const Foo = class MyClass {};

const foo = new Foo();
console.log(foo);  // MyClass {}

new MyClass(); // ReferenceError: MyClass is not defined
```

인스턴스 생성

마치 생성자 함수와 같이 new 연산자와 함께 클래스 이름을 호출하면 클래스의 인스턴스가 생성된다.

```
class Foo {}
```

```
const foo = new Foo();
```

위 코드에서 new 연산자와 함께 호출한 Foo는 클래스의 이름이 아니라 constructor(생성자)이다. 표현식이 아닌 선언식으로 정의한 클래스의 이름은 constructor와 동일하다. constructor에 대해서는 나중에 살펴보자.

```
// Foo는 사실 생성자 함수(constructor)이다.
console.log(Object.getPrototypeOf(foo).constructor === Fo
o); // tru
```

new 연산자를 사용하지 않고 constructor를 호출하면 타입 에러(TypeError)가 발생한다. constructor는 new 연산자 없이 호출할 수 없다.

```
class Foo {}

const foo = Foo(); // TypeError: Class constructor Foo cann
ot be invoked without 'new'
```

constructor

constructor는 인스턴스를 생성하고 클래스 필드를 초기화하기 위한 특수한 메소드이다.

클래스 필드(class field)클래스 내부의 캡슐화된 변수를 말한다. 데이터 멤버 또는 멤버 변수라고도 부른다. 클래스 필드는 인스턴스의 프로퍼티 또는 정적 프로퍼티가 될 수 있다. 쉽게 말해, 자바스크립트의 생성자 함수에서 this에 추가한 프로퍼티를 클래스 기반 객체지향 언어에서는 클래스 필드라고 부른다.

```
// 클래스 선언문
class Person {
  // constructor(생성자). 이름을 바꿀 수 없다.
  constructor(name) {
    // this는 클래스가 생성할 인스턴스를 가리킨다.
    // _name은 클래스 필드이다.
```

```
    this._name = name;
  }
}

// 인스턴스 생성
const me = new Person('Lee');
console.log(me); // Person {_name: "Lee"}
```

constructor는 클래스 내에 한 개만 존재할 수 있으며 만약 클래스가 2개 이상의 constructor를 포함하면 문법 에러(SyntaxError)가 발생한다. 인스턴스를 생성할 때 new 연산자와 함께 호출한 것이 바로 constructor이며 constructor의 파라미터에 전달한 값은 클래스 필드에 할당한다.

constructor는 생략할 수 있다. constructor를 생략하면 클래스에 `constructor() {}`를 포함한 것과 동일하게 동작한다. 즉, 빈 객체를 생성한다. 따라서 인스턴스에 프로퍼티를 추가하려면 인스턴스를 생성한 이후, 프로퍼티를 동적으로 추가해야 한다.

```
class Foo { }

const foo = new Foo();
console.log(foo); // Foo {}

// 프로퍼티 동적 할당 및 초기화
foo.num = 1;
console.log(foo); // Foo { num: 1 }
```

constructor는 인스턴스의 생성과 동시에 클래스 필드의 생성과 초기화를 실행한다. 따라서 클래스 필드를 초기화해야 한다면 constructor를 생략해서는 안된다.

```
class Foo {
  // constructor는 인스턴스의 생성과 동시에 클래스 필드의 생성과 초기
화를 실행한다.
  constructor(num) {
    this.num = num;
  }
}
```

```
const foo = new Foo(1);

console.log(foo); // Foo { num: 1 }
```

클래스 필드

클래스 몸체(class body)에는 메소드만 선언할 수 있다. 클래스 바디에 클래스 필드(멤버 변수)를 선언하면 문법 에러(SyntaxError)가 발생한다.

```
class Foo {
  name = ''; // SyntaxError

  constructor() {}
}
```

위 예제를 최신 브라우저(Chrome 72 이상) 또는 최신 Node.js(버전 12 이상)에서 실행하면 정상 동작한다. 이는 2019년 5월 현재 TC39 프로세스의 stage 3(candidate) 단계에 있는 클래스 몸체에서 직접 인스턴스 필드를 선언하고 private 인스턴스 필드를 선언할 수 있는 프로포절 Class field declarations proposal의 Field declarations을 최신 브라우저와 최신 Node.js가 구현했기 때문이다. 이에 대해서는 나중에 살펴보기로 하자.

클래스 필드의 선언과 초기화는 반드시 constructor 내부에서 실시한다.

```
class Foo {
  constructor(name = '') {
    this.name = name; // 클래스 필드의 선언과 초기화
  }
}
const foo = new Foo('Lee');
console.log(foo); // Foo { name: 'Lee' }
```

constructor 내부에서 선언한 클래스 필드는 클래스가 생성할 인스턴스를 가리키는 this에 바인딩한다. 이로써 클래스 필드는 클래스가 생성할 인스턴스의 프로퍼티가 되며, 클래스의 인스턴스를 통해 클래스 외부에서 언제나 참조할 수 있다. 즉, 언제나 `public` 이다.

ES6의 클래스는 다른 객체지향 언어처럼 private, public, protected 키워드와 같은 접근 제한자(access modifier)를 지원하지 않는다.

```
class Foo {
  constructor(name = '') {
    this.name = name; // public 클래스 필드
  }
}

const foo = new Foo('Lee');
console.log(foo.name); // 클래스 외부에서 참조할 수 있다.
```

Class field declarations proposal

TC39 프로세스의 stage 3(candidate) 단계에 있는 Class field declarations proposal과 Static class features에는 클래스에 관련한 몇가지 새로운 표준안이 제안되었다.

- Field declarations

- Private field

- Static public fields

```
class Foo {
  x = 1;            // Field declaration
  #p = 0;           // Private field
  static y = 20;    // Static public field
  static #sp = 30;  // Static private field
  // 2019/5 : Chrome 미구현
  // static #sm() {    // Static private method
  //   console.log('static private method');
  // }

  bar() {
    this.#p = 10; // private 필드 참조
    // this.p = 10; // 새로운 public p 필드를 동적 추가한다.
    return this.#p;
  }
}
```

```
const foo = new Foo();
console.log(foo); // Foo {#p: 10, x: 1}

console.log(foo.x); // 1
// console.log(foo.#p); // SyntaxError: Undefined private f
ield #p: must be declared in an enclosing class
console.log(Foo.y); // 20
// console.log(Foo.#sp); // SyntaxError: Undefined private
field #sp: must be declared in an enclosing class
console.log(foo.bar()); // 10
```

위 예제는 최신 브라우저(Chrome 72 이상) 또는 최신 Node.js(버전 12 이상)에서 정상 동작한다

Technical Committee 39(TC39) Ecma 인터내셔널에는 기술 위원회(Technial Committee)가 여럿 존재한다. 이 중 ECMA-262 명세(ECMAScript)의 관리를 담당하는 위원회가 바로 TC39이다. TC39는 Google, Apple, Microsoft, Mozilla 등 브라우저 벤더와 Facebook, Twitter와 같이 ECMA-262 명세(ECMAScript)를 제대로 준수해야 하는 기업으로 구성되어 있다.**TC39 프로세스**TC39 프로세스는 ECMA-262 명세(ECMAScript)에 새로운 표준 사양(제안. Proposal)을 추가하기 위해 공식적으로 명문화해 놓은 과정을 말한다. TC39 프로세스는 0 단계부터 4 단계까지 총 5개의 단계로 구성되어 있고 상위 단계로 승급하기 위한 명시적인 조건들이 존재한다. 승급 조건을 충족시킨 제안(Proposal)은 TC39의 동의를 통해 다음 단계(Stage)로 승급된다. TC39 프로세스는 아래의 단계를 거쳐 최종적으로 ECMA-262 명세(ECMAScript)의 새로운 표준 사양이 된다.stage 0: strawman ⇒ stage 1: proposal ⇒ stage 2: draft ⇒ stage 3: candidate ⇒ stage 4: finishedstage 3 상태의 제안은 심각한 문제가 없는 이상 변경되지 않고 대부분 stage 4로 승급된다. stage 4까지 승급한 제안은 큰 이변이 없는 한 새로운 ECMAScript 버전에 포함된다.현재 TC39 프로세스에 올라와 있는 제안을 확인하려면 ECMAScript proposals을 참고하기 바란다.

getter, setter

getter

getter는 클래스 필드에 접근할 때마다 클래스 필드의 값을 조작하는 행위가 필요할 때 사용한다.

getter는 메소드 이름 앞에 `get` 키워드를 사용해 정의한다. 이때 메소드 이름은 클래스 필드 이름처럼 사용된다. 다시 말해 getter는 호출하는 것이 아니라 프로퍼티처럼 참조하는 형식으로 사용하며 참조 시에 메소드가 호출된다. getter는 이름 그대로 무언가를 취득할 때 사용하므로 반드시 무언가를 반환해야 한다. 사용 방법은 아래와 같다.

```
class Foo {
  constructor(arr = []) {
    this._arr = arr;
  }

  // getter: get 키워드 뒤에 오는 메소드 이름 firstElem은 클래스
필드 이름처럼 사용된다.
  get firstElem() {
    // getter는 반드시 무언가를 반환해야 한다.
    return this._arr.length ? this._arr[0] : null;
  }
}

const foo = new Foo([1, 2]);
// 필드 firstElem에 접근하면 getter가 호출된다.
console.log(foo.firstElem); // 1
```

setter

setter는 클래스 필드에 값을 할당할 때마다 클래스 필드의 값을 조작하는 행위가 필요할 때 사용한다. setter는 메소드 이름 앞에 `set` 키워드를 사용해 정의한다. 이때 메소드 이름은 클래스 필드 이름처럼 사용된다. 다시 말해 setter는 호출하는 것이 아니라 프로퍼티처럼 값을 할당하는 형식으로 사용하며 할당 시에 메소드가 호출된다. 사용 방법은 아래와 같다.

```
class Foo {
  constructor(arr = []) {
    this._arr = arr;
  }

  // getter: get 키워드 뒤에 오는 메소드 이름 firstElem은 클래스
필드 이름처럼 사용된다.
  get firstElem() {
```

```
    // getter는 반드시 무언가를 반환하여야 한다.
    return this._arr.length ? this._arr[0] : null;
  }

  // setter: set 키워드 뒤에 오는 메소드 이름 firstElem은 클래스
필드 이름처럼 사용된다.
  set firstElem(elem) {
    // ...this._arr은 this._arr를 개별 요소로 분리한다
    this._arr = [elem, ...this._arr];
  }
}

const foo = new Foo([1, 2]);

// 클래스 필드 lastElem에 값을 할당하면 setter가 호출된다.
foo.firstElem = 100;

console.log(foo.firstElem); // 100
```

getter와 setter는 클래스에서 새롭게 도입된 기능이 아니다. getter와 setter는 접근자 프로퍼티(accessor property)이다.

```
// _arr은 데이터 프로퍼티이다.
console.log(Object.getOwnPropertyDescriptor(foo, '_arr'));
// {value: Array(2), writable: true, enumerable: true, conf
igurable: true}

// firstElem은 접근자 프로퍼티이다. 접근자 프로퍼티는 프로토타입의 프
로퍼티이다.
console.log(Object.getOwnPropertyDescriptor(Foo.prototype,
'firstElem'));
// {get: ƒ, set: ƒ, enumerable: false, configurable: true}
```

정적 메소드

클래스의 정적(static) 메소드를 정의할 때 **static** 키워드를 사용한다. 정적 메소드는 클래스의 인스턴스가 아닌 클래스 이름으로 호출한다. 따라서 클래스의 인스턴스를 생성하지 않아도 호출할 수 있다.

```
class Foo {
  constructor(prop) {
    this.prop = prop;
  }

  static staticMethod() {
    /*
    정적 메소드는 this를 사용할 수 없다.
    정적 메소드 내부에서 this는 클래스의 인스턴스가 아닌 클래스 자신을
가리킨다.
    */
    return 'staticMethod';
  }

  prototypeMethod() {
    return this.prop;
  }
}

// 정적 메소드는 클래스 이름으로 호출한다.
console.log(Foo.staticMethod());

const foo = new Foo(123);
// 정적 메소드는 인스턴스로 호출할 수 없다.
console.log(foo.staticMethod()); // Uncaught TypeError: fo
o.staticMethod is not a function
```

클래스의 정적 메소드는 인스턴스로 호출할 수 없다. 이것은 정적 메소드는 this를 사용할 수 없다는 것을 의미한다. 일반 메소드 내부에서 this는 클래스의 인스턴스를 가리키며, 메소드 내부에서 this를 사용한다는 것은 클래스의 인스턴스의 생성을 전제로 하는 것이다.

정적 메소드는 클래스 이름으로 호출하기 때문에 클래스의 인스턴스를 생성하지 않아도 사용할 수 있다. 단, 정적 메소드는 this를 사용할 수 없다. 달리 말하면 메소드 내부에서 this를

사용할 필요가 없는 메소드는 정적 메소드로 만들 수 있다. 정적 메소드는 Math 객체의 메소드처럼 애플리케이션 전역에서 사용할 유틸리티(utility) 함수를 생성할 때 주로 사용한다.

정적 메소드는 클래스의 인스턴스 생성없이 클래스의 이름으로 호출하며 클래스의 인스턴스로 호출할 수 없다고 하였다. 그 이유에 대해 좀 더 자세히 살펴보자.

prototype과 JavaScript OOP에 대한 사전 지식이 필요하다. 아직 학습 전이라면 이 부분은 지나쳐도 좋다.

위에서도 언급했지만 사실 클래스도 **함수**이고 기존 prototype 기반 패턴의 Syntactic sugar일 뿐이다.

```
class Foo {}

console.log(typeof Foo); // function
```

위 예제를 ES5로 표현해보면 아래와 같다. ES5로 표현한 아래의 코드는 ES6의 클래스로 표현한 코드와 정확히 동일하게 동작한다.

```
var Foo = (function () {
  // 생성자 함수
  function Foo(prop) {
    this.prop = prop;
  }

  Foo.staticMethod = function () {
    return 'staticMethod';
  };

  Foo.prototype.prototypeMethod = function () {
    return this.prop;
  };

  return Foo;
}());
```

```
var foo = new Foo(123);
console.log(foo.prototypeMethod()); // 123
console.log(Foo.staticMethod()); // staticMethod
console.log(foo.staticMethod()); // Uncaught TypeError: fo
o.staticMethod is not a function
```

함수 객체(자바스크립트의 함수는 객체이다. 객체로서의 함수를 강조하고자 함수 객체라 표현하였다.)는 prototype 프로퍼티를 갖는데 일반 객체의 과는 다른 것이며 일반 객체는 prototype 프로퍼티를 가지지 않는다.

함수 객체만이 가지고 있는 **prototype 프로퍼티는 함수 객체가 생성자로 사용될 때, 이 함수를 통해 생성된 객체의 부모 역할을 하는 프로토타입 객체**를 가리킨다. 위 코드에서 Foo는 생성자 함수로 사용되므로 생성자 함수 Foo의 prototype 프로퍼티가 가리키는 프로토타입 객체는 생성자 함수 Foo를 통해 생성되는 인스턴스 foo의 부모 역할을 한다.

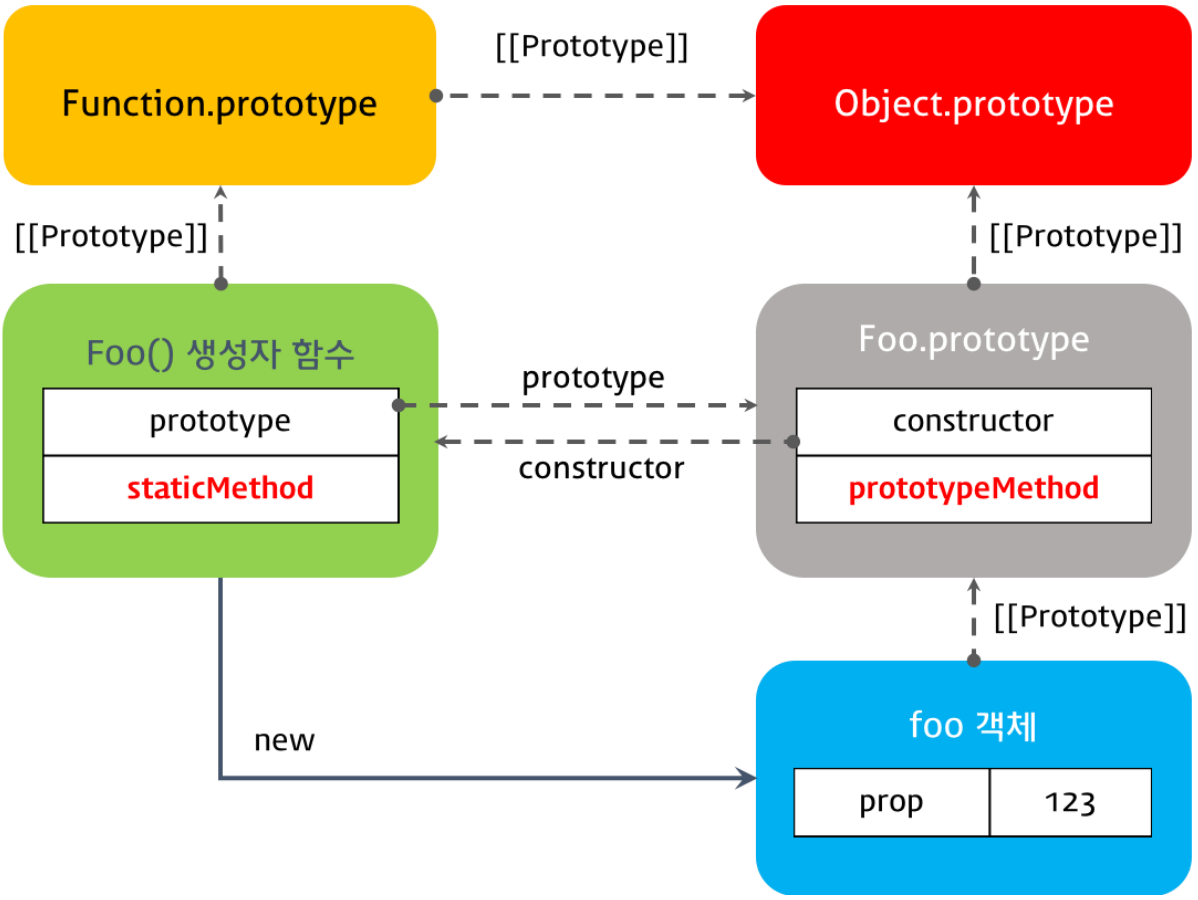
```
console.log(Foo.prototype === foo.__proto__); // true
```

그리고 생성자 함수 Foo의 prototype 프로퍼티가 가리키는 프로토타입 객체가 가지고 있는 constructor 프로퍼티는 생성자 함수 Foo를 가리킨다.

```
console.log(Foo.prototype.constructor === Foo); // true
```

정적 메소드인 staticMethod는 생성자 함수 Foo의 메소드(함수는 객체이므로 메소드를 가질 수 있다.)이고, 일반 메소드인 prototypeMethod는 프로토타입 객체 Foo.prototype의 메소드이다. 따라서 staticMethod는 foo에서 호출할 수 없다.

지금까지 설명한 내용을 프로토타입 체인 관점에서 표현하면 아래와 같다.



클래스 상속

클래스 상속(Class Inheritance)은 코드 재사용 관점에서 매우 유용하다. 새롭게 정의할 클래스가 기존에 있는 클래스와 매우 유사하다면, 상속을 통해 그대로 사용하되 다른 점만 구현하면 된다. 코드 재사용은 개발 비용을 현저히 줄일 수 있는 잠재력이 있으므로 매우 중요하다.

extends 키워드

extends 키워드는 부모 클래스(base class)를 상속받는 자식 클래스(sub class)를 정의할 때 사용한다. 부모 클래스 Circle을 상속받는 자식 클래스 Cylinder를 정의해 보자.

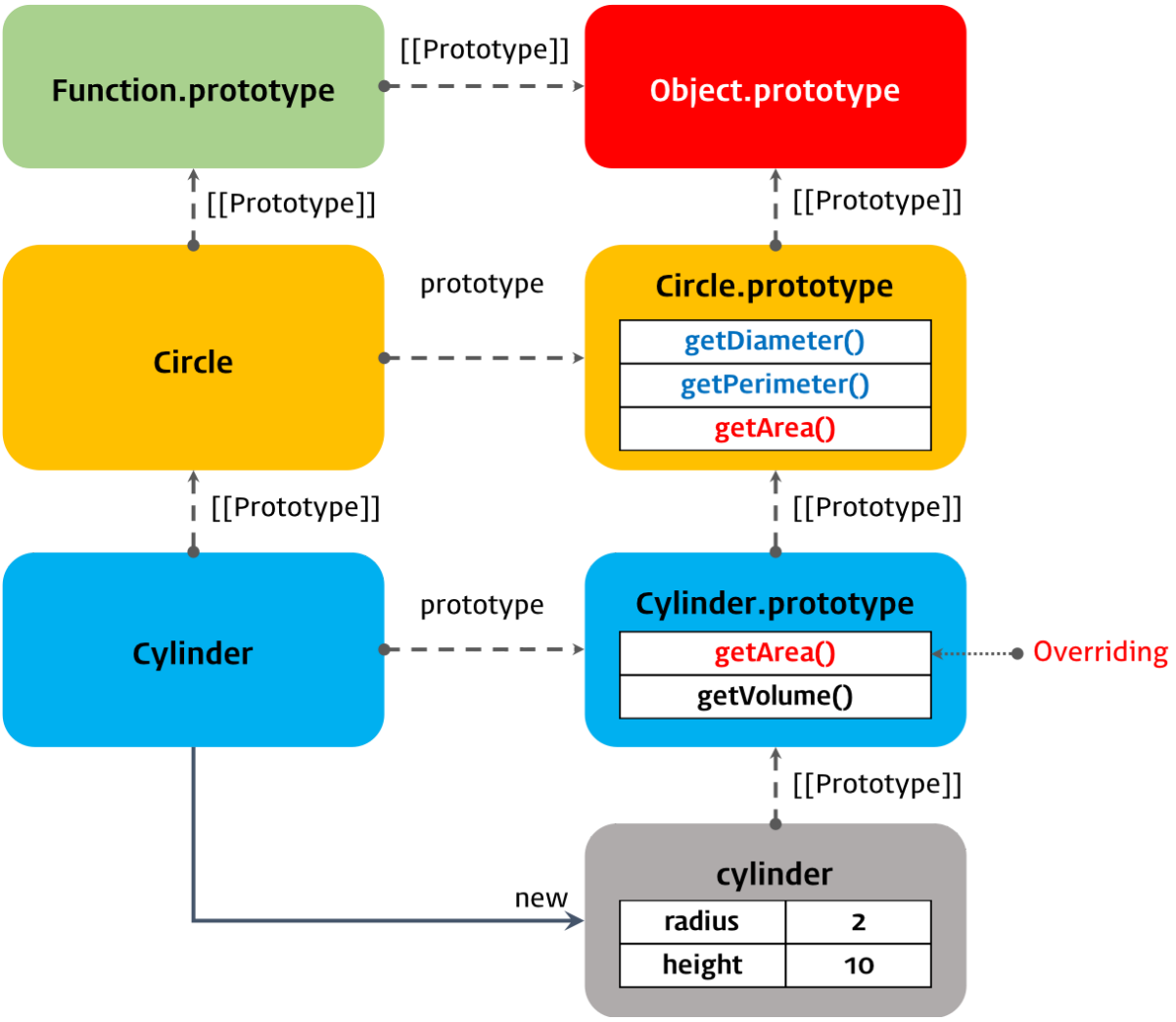
```
// 부모 클래스
class Circle {
  constructor(radius) {
    this.radius = radius; // 반지름
  }
```

```
  // 원의 지름
  getDiameter() {
    return 2 * this.radius;
  }

  // 원의 둘레
  getPerimeter() {
    return 2 * Math.PI * this.radius;
  }

  // 원의 넓이
  getArea() {
    return Math.PI * Math.pow(this.radius, 2);
  }
}

// 자식 클래스
class Cylinder extends Circle {
  constructor(radius, height) {
    super(radius);
    this.height = height;
  }

  // 원통의 넓이: 부모 클래스의 getArea 메소드를 오버라이딩하였다.
  getArea() {
    // (원통의 높이 * 원의 둘레) + (2 * 원의 넓이)
    return (this.height * super.getPerimeter()) + (2 * supe
r.getArea());
  }

  // 원통의 부피
  getVolume() {
    return super.getArea() * this.height;
  }
}

// 반지름이 2, 높이가 10인 원통
```

```
const cylinder = new Cylinder(2, 10);

// 원의 지름
console.log(cylinder.getDiameter());  // 4
// 원의 둘레
console.log(cylinder.getPerimeter()); // 12.566370614359172
// 원통의 넓이
console.log(cylinder.getArea());      // 150.79644737231007
// 원통의 부피
console.log(cylinder.getVolume());    // 125.66370614359172

// cylinder는 Cylinder 클래스의 인스턴스이다.
console.log(cylinder instanceof Cylinder); // true
// cylinder는 Circle 클래스의 인스턴스이다.
console.log(cylinder instanceof Circle);   // true
```

오버라이딩(Overriding)상위 클래스가 가지고 있는 메소드를 하위 클래스가 재정의하여 사용하는 방식이다.**오버로딩(Overloading)**매개변수의 타입 또는 갯수가 다른, 같은 이름의 메소드를 구현하고 매개변수에 의해 메소드를 구별하여 호출하는 방식이다. 자바스크립트는 오버로딩을 지원하지 않지만 arguments 객체를 사용하여 구현할 수는 있다.

위 코드를 프로토타입 관점으로 표현하면 아래와 같다. 인스턴스 cylinder는 프로토타입 체인에 의해 부모 클래스 Circle의 메소드를 사용할 수 있다.




```
console.log(cylinder.__proto__ === Cylinder.prototype); // true
console.log(Cylinder.prototype.__proto__ === Circle.prototype); // true
console.log(Circle.prototype.__proto__ === Object.prototype); // true
console.log(Object.prototype.__proto__ === null); // true
```

프로토타입 체인은 특정 객체의 프로퍼티나 메소드에 접근하려고 할 때 프로퍼티 또는 메소드가 없다면 [[Prototype]] 내부 슬롯이 가리키는 링크를 따라 자신의 부모 역할을 하는 프로토타입 객체의 프로퍼티나 메소드를 차례대로 검색한다. 그리고 검색에 성공하면 그 프로퍼티나 메소드를 사용한다.

super 키워드

super 키워드는 부모 클래스를 참조(Reference)할 때 또는 부모 클래스의 constructor를 호출할 때 사용한다.

위 “extends 키워드”의 예제를 보면 super가 메소드로 사용될 때, 그리고 객체로 사용될 때 다르게 동작하는 것을 알 수 있다. 다시 한번 예제를 살펴보자.

```
// 부모 클래스
class Circle {
...
}

class Cylinder extends Circle {
  constructor(radius, height) {
    // ① super 메소드는 부모 클래스의 constructor를 호출하면서 인수를 전달한다.
    super(radius);
    this.height = height;
  }

  // 원통의 넓이: 부모 클래스의 getArea 메소드를 오버라이딩하였다.
  getArea() {
    // (원통의 높이 * 원의 둘레) + (2 * 원의 넓이)
    // ② super 키워드는 부모 클래스(Base Class)에 대한 참조
    return (this.height * super.getPerimeter()) + (2 * super.getArea());
  }

  // 원통의 부피
  getVolume() {
    // ② super 키워드는 부모 클래스(Base Class)에 대한 참조
    return super.getArea() * this.height;
  }
}

// 반지름이 2, 높이가 10인 원통
const cylinder = new Cylinder(2, 10);
```

① super 메소드는 자식 class의 constructor 내부에서 부모 클래스의 constructor(super-constructor)를 호출한다. 즉, 부모 클래스의 인스턴스를 생성한다. **자식 클래스의 constructor에서 super()를 호출하지 않으면 this에 대한 참조 에러(ReferenceError)가 발생한다.**

```
class Parent {}

class Child extends Parent {
  // ReferenceError: Must call super constructor in derived class before accessing 'this' or returning from derived constructor
  constructor() {}
}

const child = new Child();
```

이것은 super 메소드를 호출하기 이전에는 this를 참조할 수 없음을 의미한다.

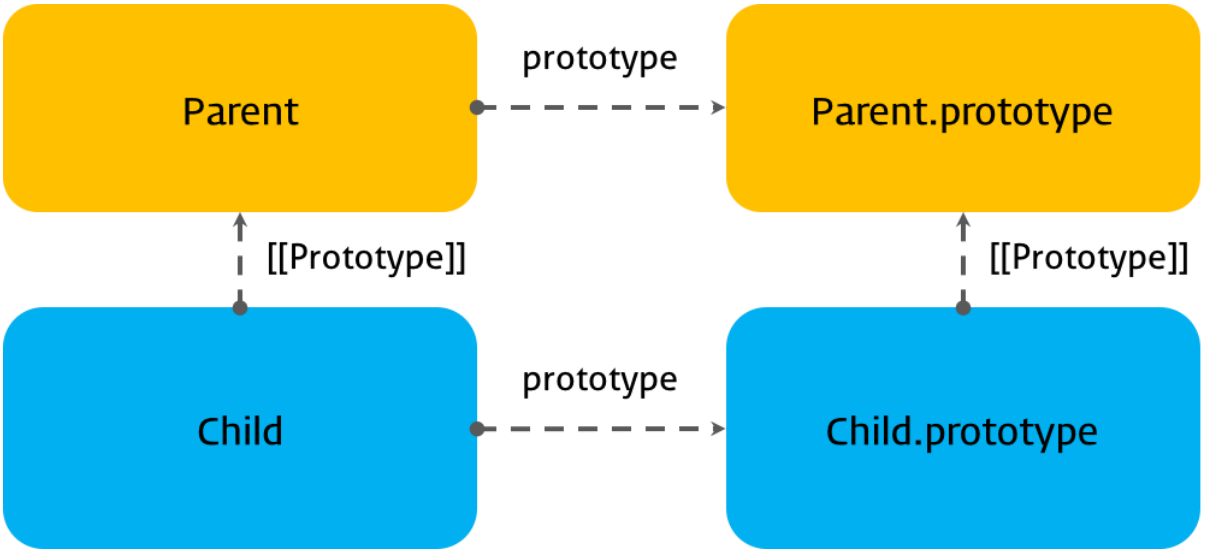
② super 키워드는 부모 클래스(Base Class)에 대한 참조이다. 부모 클래스의 필드 또는 메소드를 참조하기 위해 사용한다.

static 메소드와 prototype 메소드의 상속

프로토타입 관점에서 바라보면 자식 클래스의 [[Prototype]] 내부 슬롯이 가리키는 프로토타입 객체는 부모 클래스이다. 아래 예제를 살펴보자.

```
class Parent {}

class Child extends Parent {}

console.log(Child.__proto__ === Parent); // true
console.log(Child.prototype.__proto__ === Parent.prototype); // true
```

자식 클래스 Child의 프로토타입 객체는 부모 클래스 Parent이다. 이것을 그림으로 표현해 보면 아래와 같다.



자식 클래스의 프로토타입 객체는 부모 클래스이다

이것은 Prototype chain에 의해 부모 클래스의 정적 메소드도 상속됨을 의미한다.

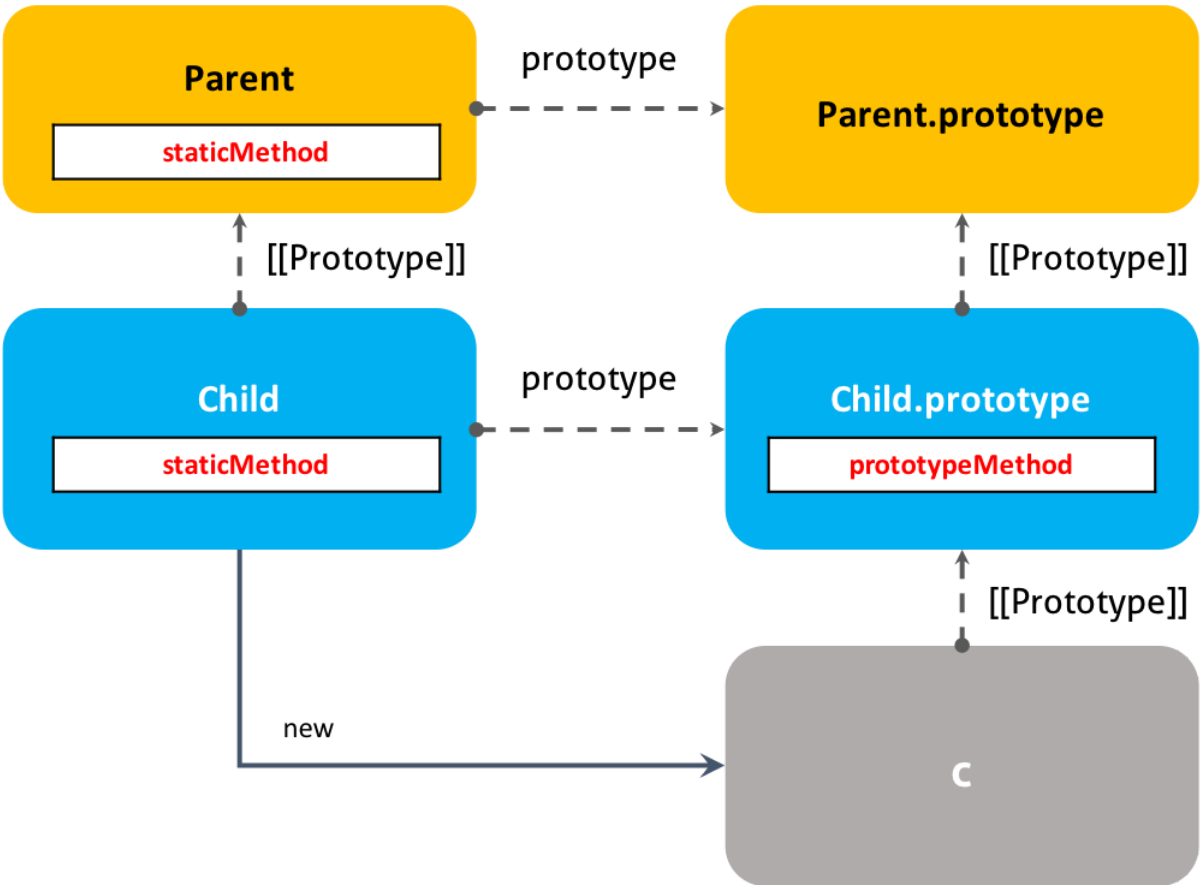
```
class Parent {
  static staticMethod() {
    return 'staticMethod';
  }
}

class Child extends Parent {}

console.log(Parent.staticMethod()); // 'staticMethod'
console.log(Child.staticMethod());  // 'staticMethod'
```

자식 클래스의 정적 메소드 내부에서도 super 키워드를 사용하여 부모 클래스의 정적 메소드를 호출할 수 있다. 이는 자식 클래스는 프로토타입 체인에 의해 부모 클래스의 정적 메소드를 참조할 수 있기 때문이다.

하지만 자식 클래스의 일반 메소드(프로토타입 메소드) 내부에서는 super 키워드를 사용하여 부모 클래스의 정적 메소드를 호출할 수 없다. 이는 자식 클래스의 인스턴스는 프로토타입 체인에 의해 부모 클래스의 정적 메소드를 참조할 수 없기 때문이다.

```
class Parent {
  static staticMethod() {
```

```
    return 'Hello';
  }
}

class Child extends Parent {
  static staticMethod() {
    return `${super.staticMethod()} wolrd`;
  }

  prototypeMethod() {
    return `${super.staticMethod()} wolrd`;
  }
}

console.log(Parent.staticMethod()); // 'Hello'
console.log(Child.staticMethod());  // 'Hello wolrd'
console.log(new Child().prototypeMethod());
// TypeError: (intermediate value).staticMethod is not a fu
nction
```


Parent
staticMethod
prototype
Parent.prototype
[[Prototype]]
[[Prototype]]
Child
staticMethod
prototype
Child.prototype
prototypeMethod
[[Prototype]]
new
c