

25장. 클래스

2022.03.26

1. 클래스는 프로토타입의 문법적 설탕인가?

프로토타입 기반 프로그래밍 방식은 자바스크립트를 어렵게 느끼는 하나의 장벽

1. 클래스는 프로토타입의 문법적 설탕인가?

프로토타입 기반 프로그래밍 방식은 자바스크립트를 어렵게 느끼는 하나의 장벽

-> ES6에서 클래스를 도입

1. 클래스는 프로토타입의 문법적 설탕인가?

프로토타입 기반 프로그래밍 방식은 자바스크립트를 어렵게 느끼는 하나의 장벽

-> ES6에서 클래스를 도입

-> 클래스는 사실 함수이며 프로토타입 기반 패턴을 클래스 기반 패턴처럼 사용할 수 있게 함

1. 클래스는 프로토타입의 문법적 설탕인가?

클래스와 생성자 함수의 차이점

1. 클래스는 프로토타입의 문법적 설탕인가?

클래스와 생성자 함수의 차이점

1. 클래스는 new 연산자 없이 호출하면 에러가 발생한다.

1. 클래스는 프로토타입의 문법적 설탕인가?

클래스와 생성자 함수의 차이점

1. 클래스는 new 연산자 없이 호출하면 에러가 발생한다.
2. 클래스는 상속을 지원하는 extends와 super 키워드를 제공한다.

1. 클래스는 프로토타입의 문법적 설탕인가?

클래스와 생성자 함수의 차이점

1. 클래스는 new 연산자 없이 호출하면 에러가 발생한다.
2. 클래스는 상속을 지원하는 extends와 super 키워드를 제공한다.
3. 클래스는 호이스팅이 발생하지 않는 것처럼 동작한다.

1. 클래스는 프로토타입의 문법적 설탕인가?

클래스와 생성자 함수의 차이점

1. 클래스는 new 연산자 없이 호출하면 에러가 발생한다.
2. 클래스는 상속을 지원하는 extends와 super 키워드를 제공한다.
3. 클래스는 호이스팅이 발생하지 않는 것처럼 동작한다.
4. 클래스 내의 모든 코드에는 암묵적으로 strict mode가 지정되어 실행되며 strict mode를 해제할 수 없다.

1. 클래스는 프로토타입의 문법적 설탕인가?

클래스와 생성자 함수의 차이점

1. 클래스는 new 연산자 없이 호출하면 에러가 발생한다.
2. 클래스는 상속을 지원하는 extends와 super 키워드를 제공한다.
3. 클래스는 호이스팅이 발생하지 않는 것처럼 동작한다.
4. 클래스 내의 모든 코드에는 암묵적으로 strict mode가 지정되어 실행되며 strict mode를 해제할 수 없다.
5. 클래스의 constructor, 프로토타입 메서드, 정적 메서드는 열거되지 않는다.

1. 클래스는 프로토타입의 문법적 설탕인가?

클래스와 생성자 함수의 차이점

1. 클래스는 new 연산자 없이 호출하면 에러가 발생한다.
2. 클래스는 상속을 지원하는 extends와 super 키워드를 제공한다.
3. 클래스는 호이스팅이 발생하지 않는 것처럼 동작한다.
4. 클래스 내의 모든 코드에는 암묵적으로 strict mode가 지정되어 실행되며 strict mode를 해제할 수 없다.
5. 클래스의 constructor, 프로토타입 메서드, 정적 메서드는 열거되지 않는다.

-> 클래스는 새로운 객체 생성 매커니즘으로 보는 것이 조금 더 합당하다.

2. 클래스 정의



2. 클래스 정의



```
class Person {}
```



```
const Person = class {}
```

2. 클래스 정의



```
class Person {}
```



```
const Person = class {}
```



```
const Person = class MyClass {}
```

2. 클래스 정의

```
class Person {}
```

```
const Person = class {}
```

```
const Person = class MyClass {}
```

1. 무명 리터럴로 생성할 수 있다. 즉, 런타임에 생성이 가능하다.
2. 변수나 자료구조(객체, 배열)에 저장할 수 있다.
3. 함수의 매개변수에게 전달할 수 있다.
4. 함수의 반환값으로 사용할 수 있다.

2. 클래스 정의



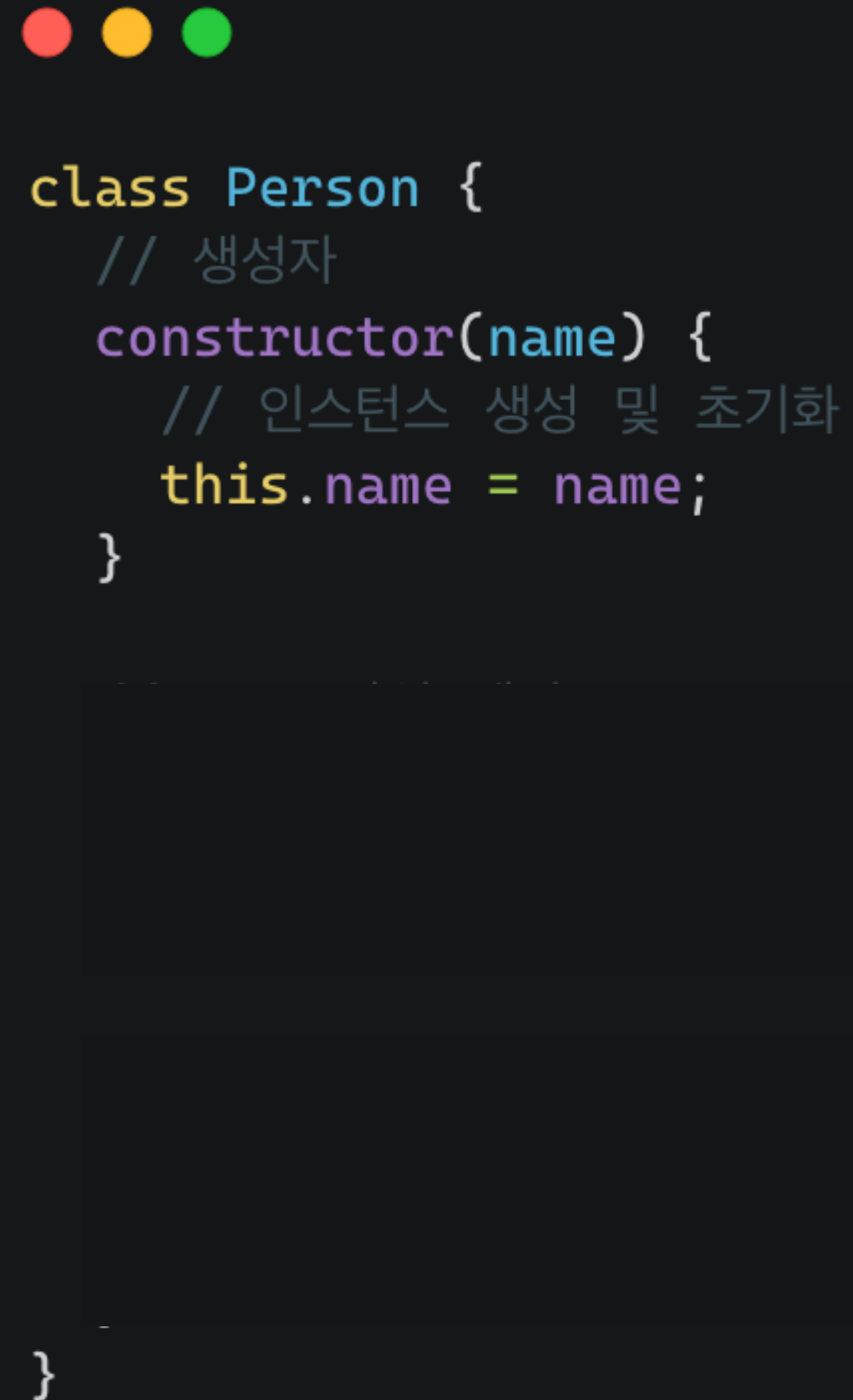
```
class Person {
```

```
}
```

클래스 몸체에 정의할 수 있는 메서드

- constructor
- 프로토타입 메서드
- 정적 메서드

2. 클래스 정의



```
class Person {  
    // 생성자  
    constructor(name) {  
        // 인스턴스 생성 및 초기화  
        this.name = name;  
    }  
  
    // ...  
  
}
```

클래스 몸체에 정의할 수 있는 메서드

- constructor
- 프로토타입 메서드
- 정적 메서드

2. 클래스 정의



```
class Person {  
  // 생성자  
  constructor(name) {  
    // 인스턴스 생성 및 초기화  
    this.name = name;  
  }  
  
  // 프로토타입 메서드  
  sayHi() {  
    console.log(`Hi! My name is ${this.name}`);  
  }  
}
```

클래스 몸체에 정의할 수 있는 메서드

- constructor
- 프로토타입 메서드
- 정적 메서드

2. 클래스 정의

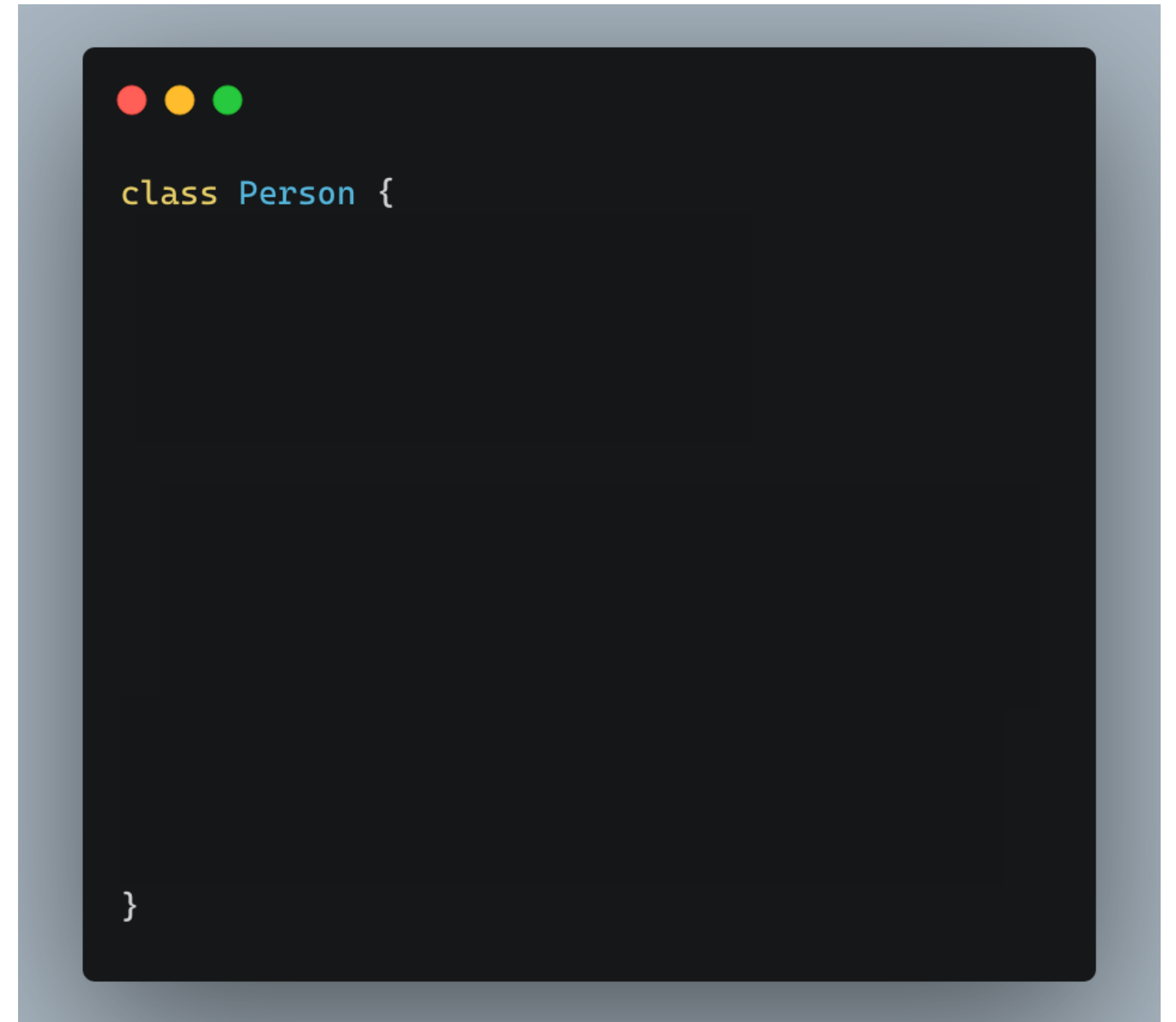
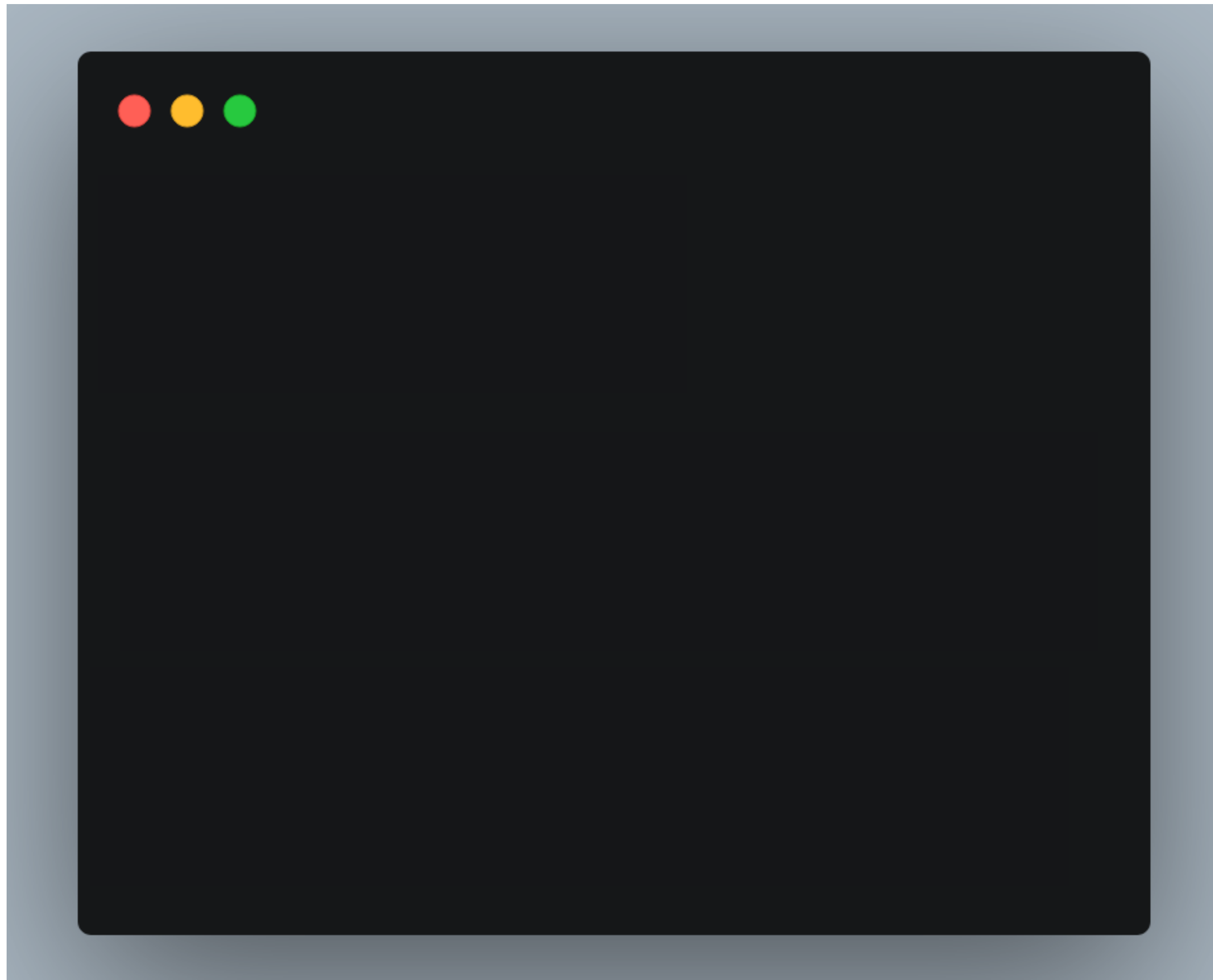


```
class Person {  
  // 생성자  
  constructor(name) {  
    // 인스턴스 생성 및 초기화  
    this.name = name;  
  }  
  
  // 프로토타입 메서드  
  sayHi() {  
    console.log(`Hi! My name is ${this.name}`);  
  }  
  
  // 정적 메서드  
  static sayHello() {  
    console.log("Hello");  
  }  
}
```

클래스 몸체에 정의할 수 있는 메서드

- constructor
- 프로토타입 메서드
- 정적 메서드

2. 클래스 정의



2. 클래스 정의



// 생성자 함수

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



```
class Person {  
  // 생성자  
  constructor(name) {  
    // 인스턴스 생성 및 초기화  
    this.name = name;  
  }  
}
```

```
}
```

2. 클래스 정의

```

// 생성자 함수
function Person(name) {
  this.name = name;
}

// 프로토타입 메서드
Person.prototype.sayHi = function() {
  console.log(`Hi! My name is ${this.name}`);
}
```

```

class Person {
  // 생성자
  constructor(name) {
    // 인스턴스 생성 및 초기화
    this.name = name;
  }

  // 프로토타입 메서드
  sayHi() {
    console.log(`Hi! My name is ${this.name}`);
  }
}
```

2. 클래스 정의

```

// 생성자 함수
function Person(name) {
  this.name = name;
}

// 프로토타입 메서드
Person.prototype.sayHi = function() {
  console.log(`Hi! My name is ${this.name}`);
}

// 정적 메서드
Person.sayHello = function() {
  console.log("Hello");
}

```

```

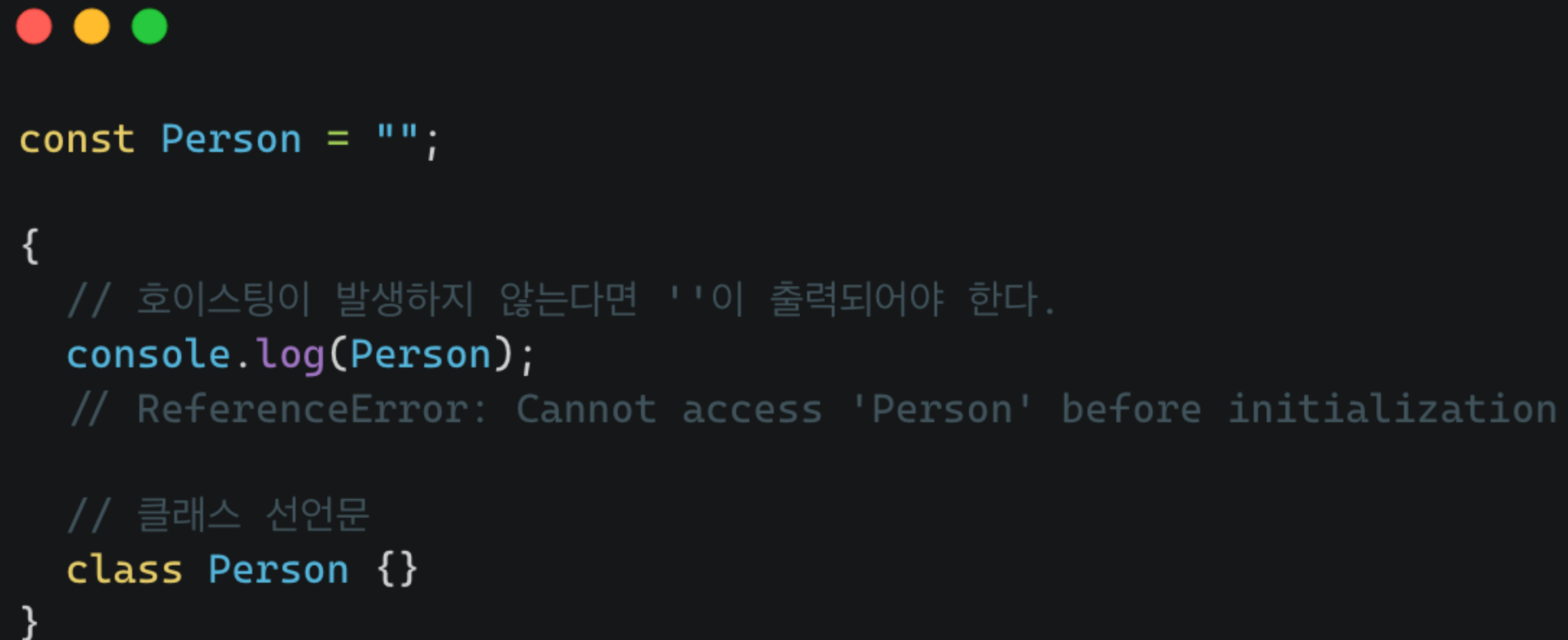
class Person {
  // 생성자
  constructor(name) {
    // 인스턴스 생성 및 초기화
    this.name = name;
  }

  // 프로토타입 메서드
  sayHi() {
    console.log(`Hi! My name is ${this.name}`);
  }

  // 정적 메서드
  static sayHello() {
    console.log("Hello");
  }
}

```

3. 클래스 호이스팅



```
const Person = "";  
  
{  
  // 호이스팅이 발생하지 않는다면 ''이 출력되어야 한다.  
  console.log(Person);  
  // ReferenceError: Cannot access 'Person' before initialization  
  
  // 클래스 선언문  
  class Person {}  
}
```

let, const 키워드로 선언한 변수처럼 호이스팅된다.

4. 인스턴스 생성



```
class Person {}
```

```
const me = Person();
```

```
// TypeError: Class constructor Foo cannot be invoked without 'new'
```

new 키워드 없이 클래스를 호출하면 에러가 발생한다.

4. 인스턴스 생성



```
const Person = class MyClass {}  
  
const me = new Person();  
  
console.log(MyClass);  
// ReferenceError: MyClass is not defined  
  
const you = new MyClass();  
// ReferenceError: MyClass is not defined
```

기명 클래스 표현식의 클래스 이름을 사용하면 에러가 발생한다.

4. 인스턴스 생성



```
const Person = class MyClass {}  
  
const me = new Person();  
  
console.log(MyClass);  
// ReferenceError: MyClass is not defined  
  
const you = new MyClass();  
// ReferenceError: MyClass is not defined
```

기명 클래스 표현식의 클래스 이름을 사용하면 에러가 발생한다.
기명 클래스 표현식의 클래스 이름은 클래스 몸체 내부에서만 유효하다.

5. 메서드 constructor

constructor는 인스턴스를 생성하고 초기화하기 위한 메서드다. 이름은 변경할 수 없다.

5. 메서드

constructor

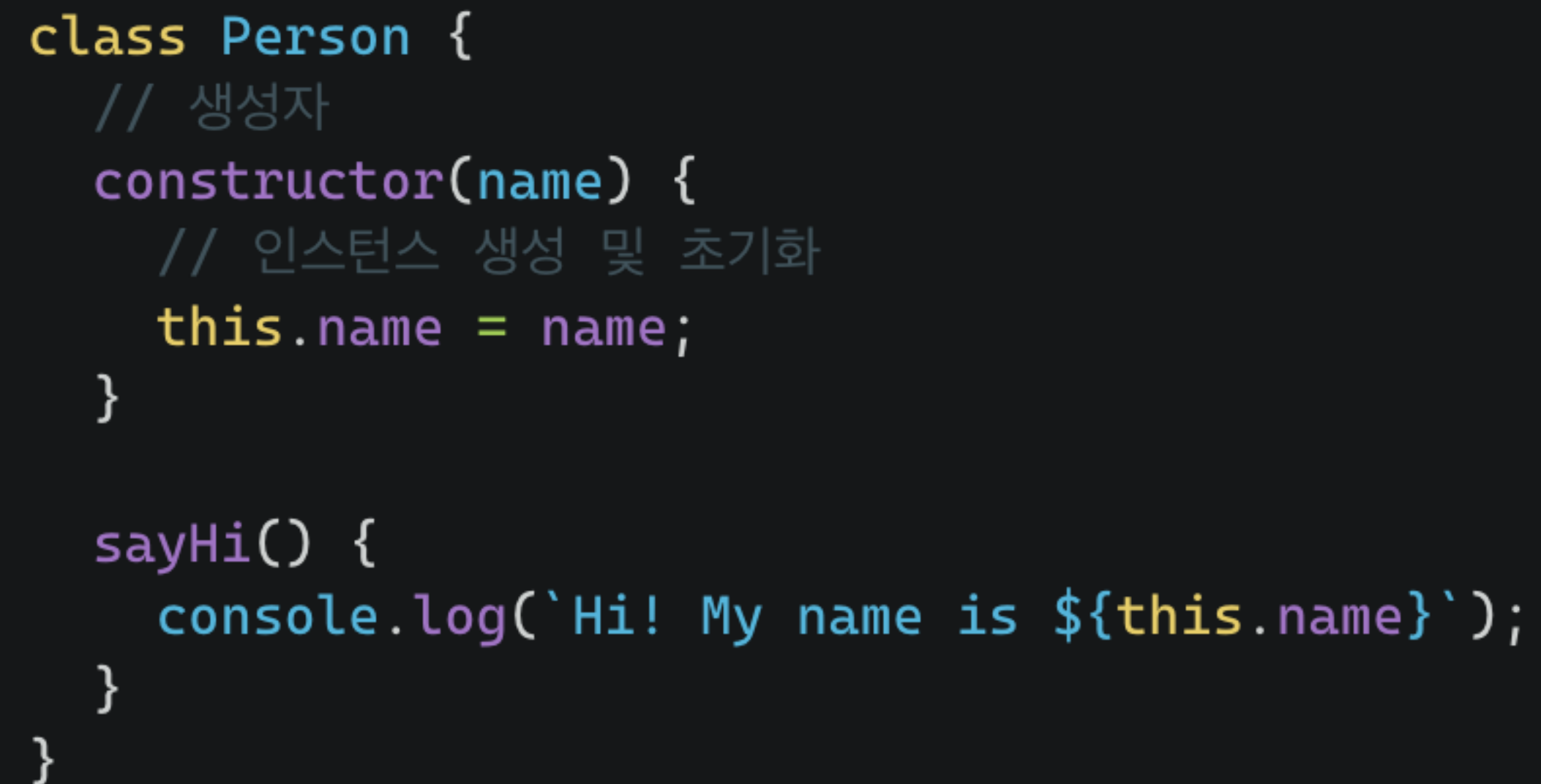
constructor는 인스턴스를 생성하고 초기화하기 위한 메서드다. 이름은 변경할 수 없다.

클래스의 constructor 메서드와 프로토타입의 constructor 프로퍼티는 직접적인 관련이 없다.

클래스의 constructor 메서드 -> 인스턴스를 생성하고 초기화하는 메서드

프로토타입의 constructor 프로퍼티 -> 해당 객체를 생성한 생성자 함수

5. 메서드 constructor



```
class Person {  
  // 생성자  
  constructor(name) {  
    // 인스턴스 생성 및 초기화  
    this.name = name;  
  }  
  
  sayHi() {  
    console.log(`Hi! My name is ${this.name}`);  
  }  
}
```

클래스가 생성한 인스턴스에서는 constructor 메서드가 보이지 않는다.

5. 메서드 constructor

```
>> console.dir(me)
▼ Object { name: "Lee" }                                     debugger eval code:1:9
  로그  name: "Lee"
        ▼ <prototype>: Object { ... }
          ▼ constructor: class Person {
            constructor(name) {
              length: 1
              name: "Person"
              ▶ prototype: Object { ... }
              ▶ <prototype>: function ()
              ▶ sayHi: function sayHi()
              ▶ <prototype>: Object { ... }
```

me를 생성한 Person class를 가리킨다.

5. 메서드 constructor

```
>> console.dir(me)
▼ Object { name: "Lee" }
  name: "Lee"
  <prototype>: Object { ... }
    ▼ constructor: class Person {
      constructor(name) {
        length: 1
        name: "Person"
        ▶ prototype: Object { ... }
        ▶ <prototype>: function ()
        ▶ sayHi: function sayHi()
        ▶ <prototype>: Object { ... }
    }
  ▶ sayHi: function sayHi()
  ▶ <prototype>: Object { ... }
```

debugger eval code:1:9

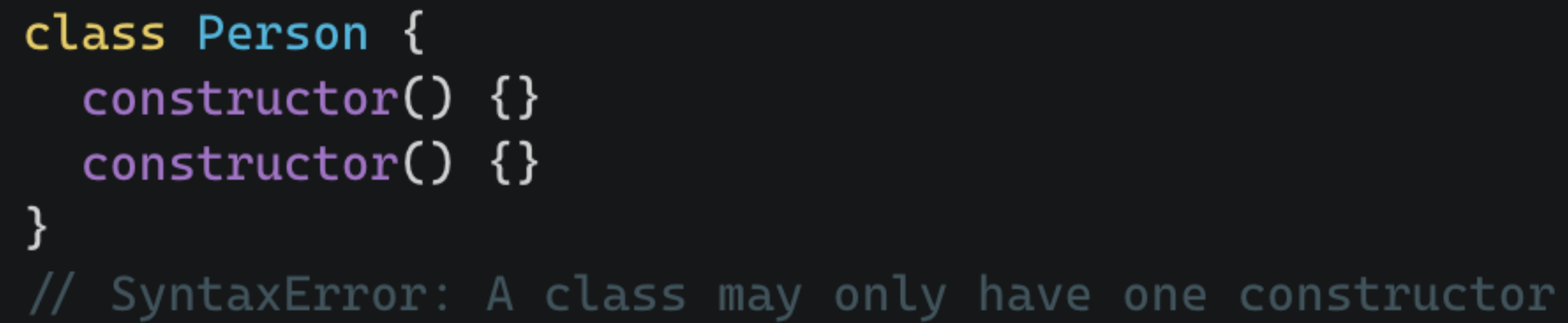
로그

me를 생성한 Person class를 가리킨다.

클래스의 constructor 메서드는 클래스가 평가되면 constructor 안에 기술된 동작을 수행하고 함수 객체의 일부가 된다.

5. 메서드

constructor - 생성자 함수와의 차이점



```
class Person {  
  constructor() {}  
  constructor() {}  
}  
// SyntaxError: A class may only have one constructor
```

클래스 내에 최대 1개만 존재할 수 있다.

5. 메서드

constructor - 생성자 함수와의 차이점



```
class Person {  
    // constructor는 생략하면 아래와 같이 빈 constructor가 암묵적으로 정의된다.  
    constructor() {}  
}
```

생략 가능하다.

5. 메서드 constructor



```
class Person {  
  constructor(name, address) {  
    // 인수로 인스턴스 초기화  
    this.name = name;  
    this.address = address;  
  }  
}  
  
const me = new Person("Lee", "Seoul");  
console.log(me); // {name:"Lee", address: "Seoul"}
```

인스턴스 생성과 동시에 프로퍼티 추가를 통해 초기화를 실행할 땐 constructor를 생략하면 안된다.

5. 메서드 constructor

```
class Person {  
  constructor(name, address) {  
    // 인수로 인스턴스 초기화  
    this.name = name;  
    this.address = address;  
  
    return {};  
  }  
}  
  
const me = new Person("Lee", "Seoul");  
console.log(me); // {}
```

constructor가 반환값을 가지면 생성된 인스턴스가 아닌 return에 명시된 객체가 반환된다.

5. 메서드 constructor

```
class Person {  
  constructor(name, address) {  
    // 인수로 인스턴스 초기화  
    this.name = name;  
    this.address = address;  
  
    return 100;  
  }  
}  
  
const me = new Person("Lee", "Seoul");  
console.log(me); // {name:"Lee", address: "Seoul"}
```

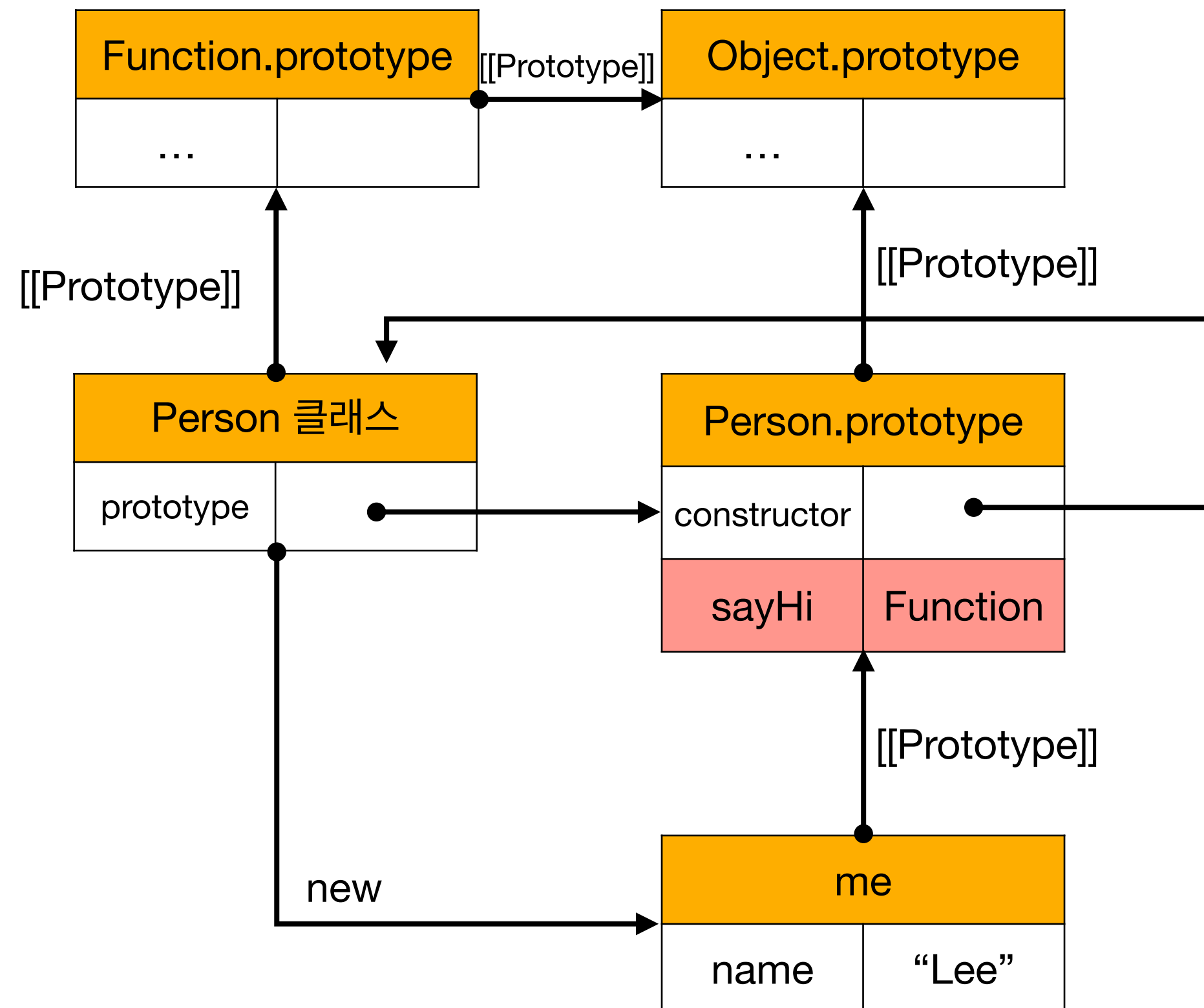
원시값을 반환하면 원시값은 무시된다.

5. 메서드

프로토타입 메서드



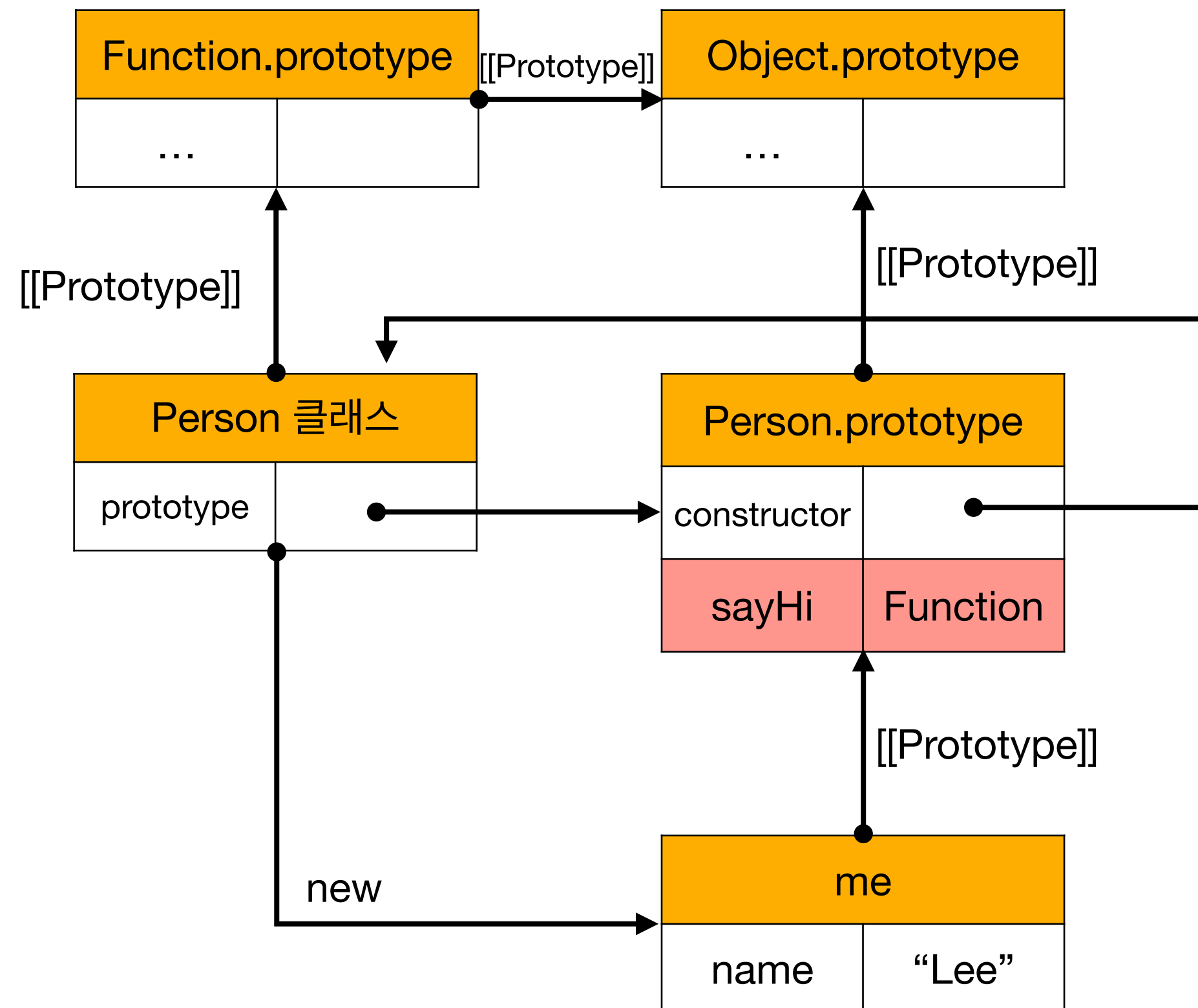
```
class Person {  
  // 생성자  
  constructor(name) {  
    // 인스턴스 생성 및 초기화  
    this.name = name;  
  }  
  
  // 프로토타입 메서드  
  sayHi() {  
    console.log(`Hi! My name is ${this.name}`);  
  }  
}
```



5. 메서드

프로토타입 메서드

```
class Person {  
  // 생성자  
  constructor(name) {  
    // 인스턴스 생성 및 초기화  
    this.name = name;  
  }  
  
  // 프로토타입 메서드  
  sayHi() {  
    console.log(`Hi! My name is ${this.name}`);  
  }  
}
```



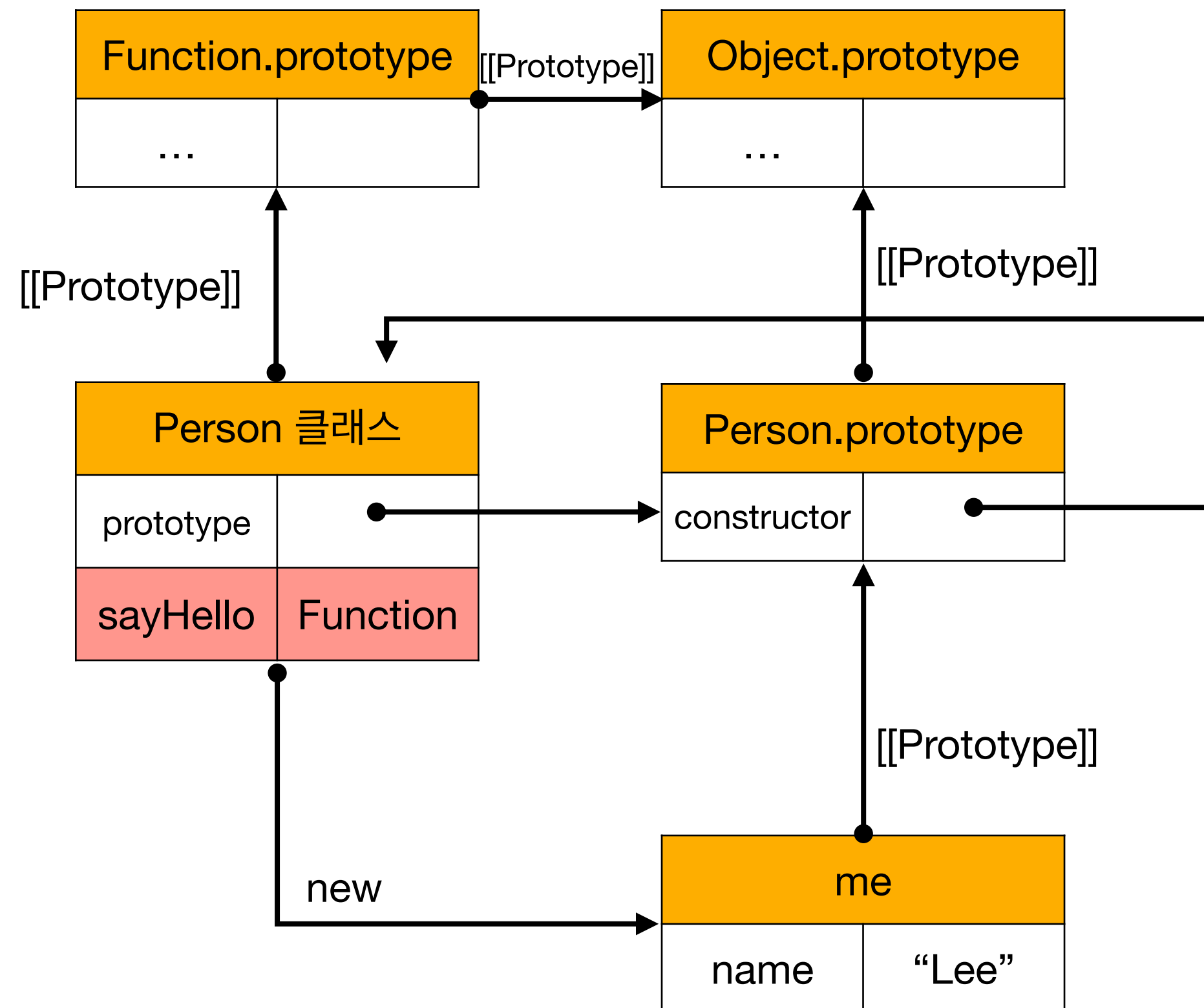
클래스는 프로토타입 기반의 객체 생성 매커니즘이다.

5. 메서드

정적 메서드



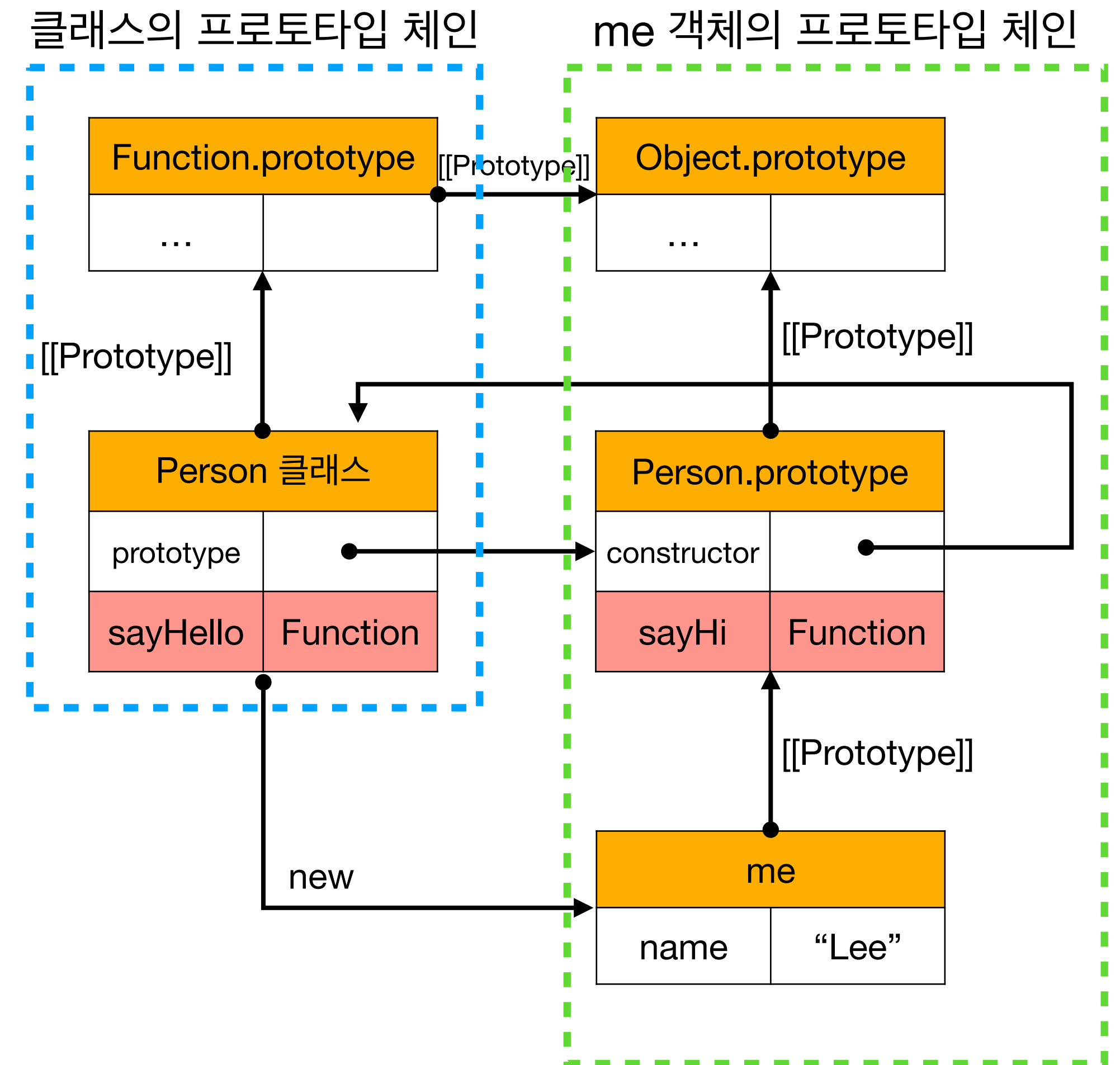
```
class Person {  
  // 생성자  
  constructor(name) {  
    // 인스턴스 생성 및 초기화  
    this.name = name;  
  }  
  
  // 정적 메서드  
  static sayHello() {  
    console.log("Hello");  
  }  
}
```



5. 메서드

정적 메서드 vs 프로토타입 메서드

	정적 메서드	프로토타입 메서드
속해있는 프로토타입 체인	서로 다름	
호출자	클래스	인스턴스
인스턴스 프로퍼티 참조 가능 여부	X	O



5. 메서드

클래스에서 정의한 메서드의 특징

5. 메서드

클래스에서 정의한 메서드의 특징

1. function 키워드를 생략한 메서드 축약 표현을 사용한다.

5. 메서드

클래스에서 정의한 메서드의 특징

1. function 키워드를 생략한 메서드 축약 표현을 사용한다.
2. 객체 리터럴과는 다르게 클래스에 메서드를 정의할 때는逗마가 필요 없다.

5. 메서드

클래스에서 정의한 메서드의 특징

1. function 키워드를 생략한 메서드 축약 표현을 사용한다.
2. 객체 리터럴과는 다르게 클래스에 메서드를 정의할 때는逗마가 필요 없다.
3. 암묵적으로 strict mode로 실행된다.

5. 메서드

클래스에서 정의한 메서드의 특징

1. function 키워드를 생략한 메서드 축약 표현을 사용한다.
2. 객체 리터럴과는 다르게 클래스에 메서드를 정의할 때는逗마가 필요 없다.
3. 암묵적으로 strict mode로 실행된다.
4. for ... in 문이나 Object.keys 메서드 등으로 열거할 수 없다.

5. 메서드

클래스에서 정의한 메서드의 특징

1. function 키워드를 생략한 메서드 축약 표현을 사용한다.
2. 객체 리터럴과는 다르게 클래스에 메서드를 정의할 때는逗마가 필요 없다.
3. 암묵적으로 strict mode로 실행된다.
4. for ... in 문이나 Object.keys 메서드 등으로 열거할 수 없다.
5. 내부 메서드 [[Construct]]를 갖지 않는 non-constructor다. 따라서 new 연산자와 함께 호출할 수 없다.

6. 클래스의 인스턴스 생성 과정



```
class Person {  
    constructor(name) {  
  
    }  
}
```


6. 클래스의 인스턴스 생성 과정

1. 인스턴스 생성과 this 바인딩

인스턴스가 될 빈 객체가 생성되고,
이 객체가 this에 바인딩된다.

```
class Person {  
  constructor(name) {  
    // 1. 암묵적으로 인스턴스가 생성되고 this에 바인딩된다.  
    console.log(this) // Person {}  
    console.log(Object.getPrototypeOf(this) === Person.prototype) // true  
  
  }  
}
```

6. 클래스의 인스턴스 생성 과정

1. 인스턴스 생성과 this 바인딩

인스턴스가 될 빈 객체가 생성되고,
이 객체가 this에 바인딩된다.

2. 인스턴스 초기화

this에 바인딩되어있는 인스턴스에
프로퍼티를 추가하고, 값을 초기화한다.

```
class Person {  
  constructor(name) {  
    // 1. 암묵적으로 인스턴스가 생성되고 this에 바인딩된다.  
    console.log(this) // Person {}  
    console.log(Object.getPrototypeOf(this) === Person.prototype) // true  
  
    // 2. this.에 바인딩되어있는 인스턴스를 초기화한다.  
    this.name = name;  
  
  }  
}
```

6. 클래스의 인스턴스 생성 과정

1. 인스턴스 생성과 this 바인딩

인스턴스가 될 빈 객체가 생성되고,
이 객체가 this에 바인딩된다.

2. 인스턴스 초기화

this에 바인딩되어있는 인스턴스에
프로퍼티를 추가하고, 값을 초기화한다.

3. 인스턴스 반환

```
class Person {  
  constructor(name) {  
    // 1. 암묵적으로 인스턴스가 생성되고 this에 바인딩된다.  
    console.log(this) // Person {}  
    console.log(Object.getPrototypeOf(this) === Person.prototype) // true  
  
    // 2. this.에 바인딩되어있는 인스턴스를 초기화한다.  
    this.name = name;  
  
    // 3. 생성된 인스턴스가 바인딩된 this가 암묵적으로 반환된다.  
  }  
}
```

7. 프로퍼티

인스턴스 프로퍼티

```
class Person {  
  constructor(name) {  
    // 인스턴스 프로퍼티  
    this.name = name; // name 프로퍼티는 public하다.  
  }  
}  
  
const me = new Person("Lee");  
  
// name은 public하다.  
console.log(me.name); // Lee
```

constructor 내부에서 this에 추가한 프로퍼티는 언제나 인스턴스의 프로퍼티가 된다.

7. 프로퍼티

접근자 프로퍼티

```
const person = {
  firstName: "Ungmo",
  lastName: "Lee",

  get fullName() {
    return `${this.firstName} ${this.lastName}`;
  }

  set fullName(name) {
    [this.firstName, this.lastName] = name.split(" ");
  }
};

console.log(`${person.firstName} ${person.lastName}`);

// setter로 프로퍼티 값 설정
person.fullName = "Heegun Lee";
console.log(person); // {firstName: "Heegun", lastName: "Lee"}

// getter로 프로퍼티 값 참조
console.log(person.fullName);
```

```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  get fullName() {
    return `${this.firstName} ${this.lastName}`;
  }

  set fullName(name) {
    [this.firstName, this.lastName] = name.split(" ");
  }
};

const person = new Person("Ungmo", "Lee");

console.log(`${person.firstName} ${person.lastName}`);

// setter로 프로퍼티 값 설정
person.fullName = "Heegun Lee";
console.log(person); // {firstName: "Heegun", lastName: "Lee"}

// getter로 프로퍼티 값 참조
console.log(person.fullName);
```

7. 프로퍼티

접근자 프로퍼티

```
const person = {
  firstName: "Ungmo",
  lastName: "Lee",

  get fullName() {
    return `${this.firstName} ${this.lastName}`;
  }

  set fullName(name) {
    [this.firstName, this.lastName] = name.split(" ");
  }
};

console.log(`${person.firstName} ${person.lastName}`);

// setter로 프로퍼티 값 설정
person.fullName = "Heegun Lee";
console.log(person); // {firstName: "Heegun", lastName: "Lee"}

// getter로 프로퍼티 값 참조
console.log(person.fullName);
```

```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  get fullName() {
    return `${this.firstName} ${this.lastName}`;
  }

  set fullName(name) {
    [this.firstName, this.lastName] = name.split(" ");
  }
};

const person = new Person("Ungmo", "Lee");

console.log(`${person.firstName} ${person.lastName}`);

// setter로 프로퍼티 값 설정
person.fullName = "Heegun Lee";
console.log(person); // {firstName: "Heegun", lastName: "Lee"}

// getter로 프로퍼티 값 참조
console.log(person.fullName);
```


7. 프로퍼티

접근자 프로퍼티

```
const person = {
  firstName: "Ungmo",
  lastName: "Lee",

  get fullName() {
    return `${this.firstName} ${this.lastName}`;
  }

  set fullName(name) {
    [this.firstName, this.lastName] = name.split(" ");
  }
};

console.log(`${person.firstName} ${person.lastName}`);

// setter로 프로퍼티 값 설정
person.fullName = "Heegun Lee";
console.log(person); // {firstName: "Heegun", lastName: "Lee"}

// getter로 프로퍼티 값 참조
console.log(person.fullName);
```

```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  get fullName() {
    return `${this.firstName} ${this.lastName}`;
  }

  set fullName(name) {
    [this.firstName, this.lastName] = name.split(" ");
  }
};

const person = new Person("Ungmo", "Lee");

console.log(`${person.firstName} ${person.lastName}`);

// setter로 프로퍼티 값 설정
person.fullName = "Heegun Lee";
console.log(person); // {firstName: "Heegun", lastName: "Lee"}

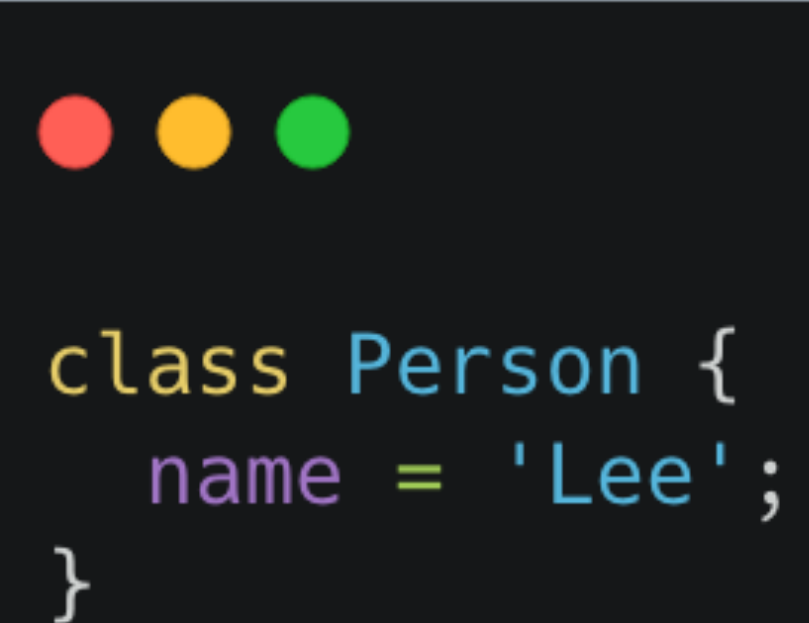
// getter로 프로퍼티 값 참조
console.log(person.fullName);
```

7. 프로퍼티

클래스 필드 정의



```
class Person {  
    constructor() {  
        this.name = 'Lee';  
    }  
}
```



```
class Person {  
    name = 'Lee';  
}
```


7. 프로퍼티

클래스 필드 정의 - 주의사항

1. this는 constructor 안에서만 사용해야한다.

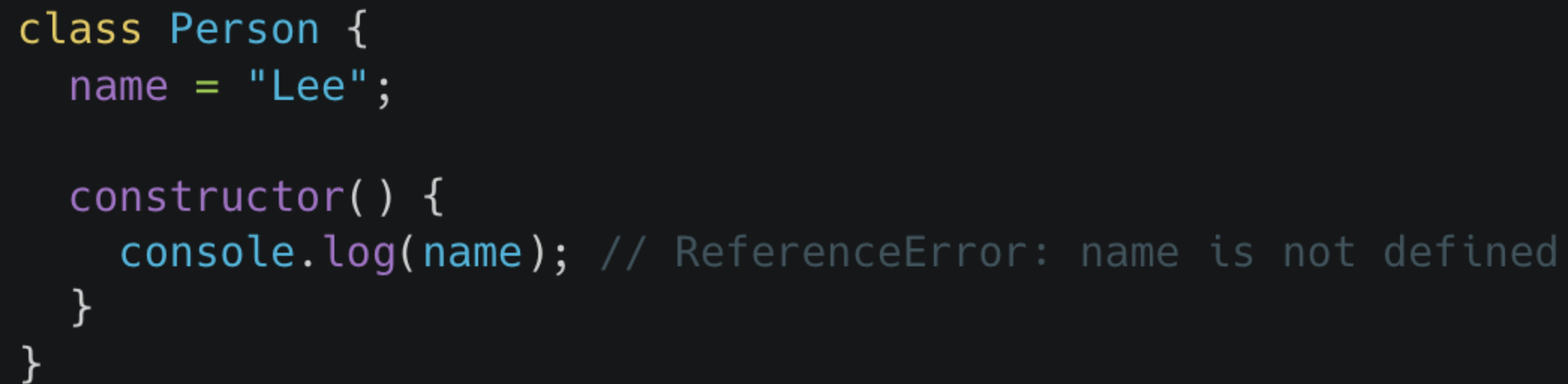


```
class Person {  
  this.name = ''; // SyntaxError: Unexpect token '.'  
}
```

7. 프로퍼티

클래스 필드 정의 - 주의사항

2. 참조할 땐 this를 사용해야한다.



```
class Person {  
  name = "Lee";  
  
  constructor() {  
    console.log(name); // ReferenceError: name is not defined  
  }  
}
```

7. 프로퍼티

클래스 필드 정의 - 주의사항

3. 외부값으로 초기화할 필요가 있다면 constructor에서 초기화를 해야한다.



```
class Person {  
  name;  
  
  constructor(name) {  
    this.name = name;  
  }  
}  
  
const me = new Person("Lee");  
console.log(me); // Person {name: "Lee"}
```

7. 프로퍼티

클래스 필드 정의 - 주의사항

4. 클래스 필드를 통해 메서드를 정의할 수 있지만 권장하지 않는다.

```
class Person {  
  name = 'Lee';  
  
  // 클래스 필드를 통해 메서드 정의  
  // 인스턴스 메서드가 된다.  
  getName = function() {  
    return this.name;  
  }  
}
```

```
class Person {  
  name = 'Lee';  
  
  // 프로토타입 메서드가 된다.  
  getName() {  
    return this.name;  
  }  
}
```

인스턴스 메서드는 인스턴스가 생성될 때마다 중복 생성되기 때문이다.

7. 프로퍼티

private 필드

```
class Person {  
  name;  
  #age = 100;  
  
  constructor(name, age) {  
    this.name = name;  
    this.#age = age;  
  }  
}  
  
const me = new Person("Lee", 33);  
  
console.log(me.name); // Lee  
  
// SyntaxError: Private field `#name` must be declared in an enclosing class  
console.log(me.#age);
```

7. 프로퍼티

private 필드 - 주의점




```
class Person {  
  name;  
  
  constructor(name, age) {  
    this.name = name;  
    this.#age = age;  
    // SyntaxError: Private field `#name` must be declared in an enclosing class  
  }  
}
```

private 필드를 constructor에서 직접 정의하면 에러가 발생한다.

7. 프로퍼티

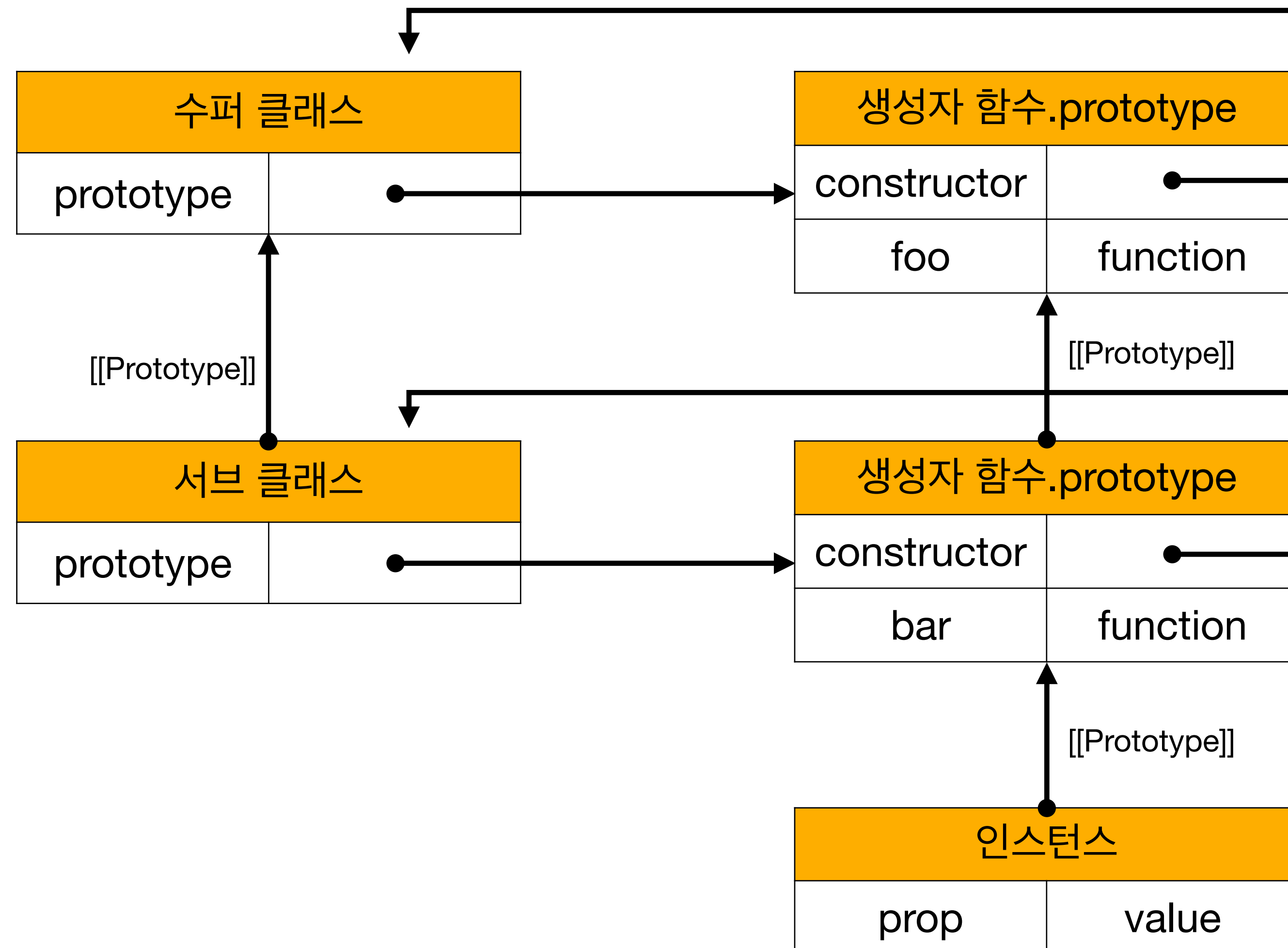
static 필드



```
class MyMath {  
    static PI = 22 / 7;  
    static #num = 10;  
    static increment() {  
        return ++MyMath.#num;  
    }  
}  
  
console.log(MyMath.PI);  
console.log(MyMath.increment());
```

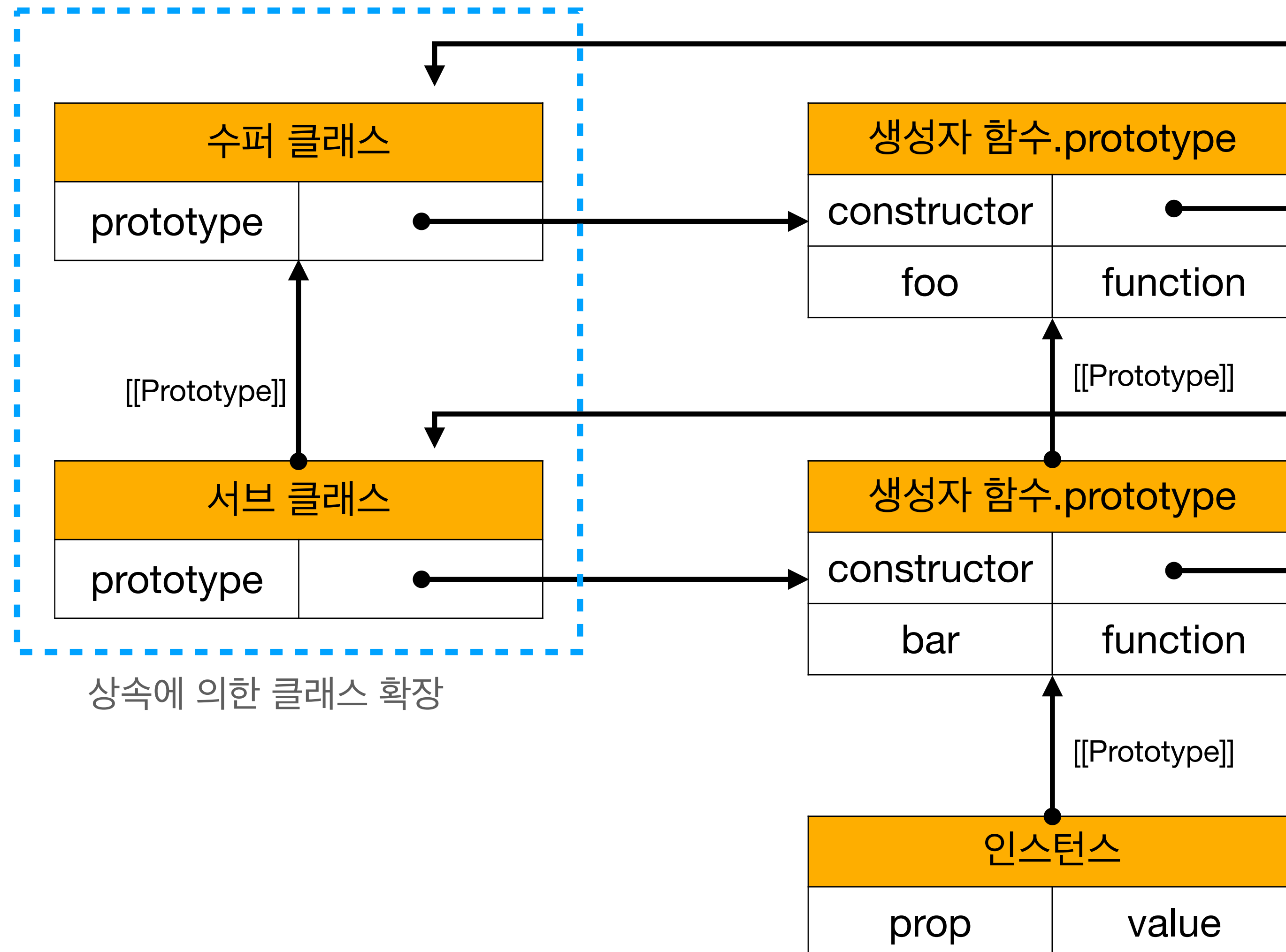
8. 상속에 의한 클래스 확장

클래스 상속과 생성자 함수 상속



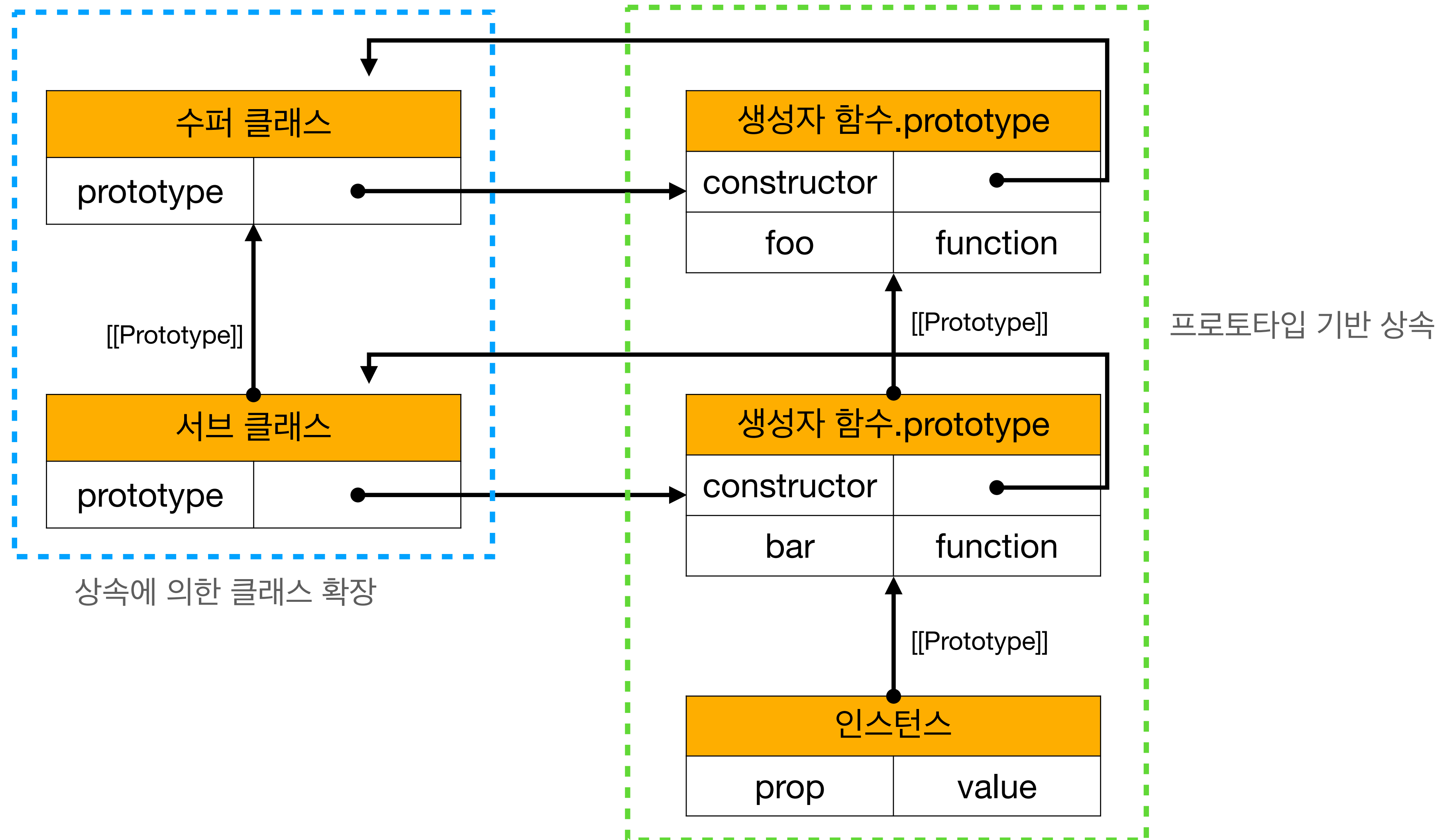
8. 상속에 의한 클래스 확장

클래스 상속과 생성자 함수 상속



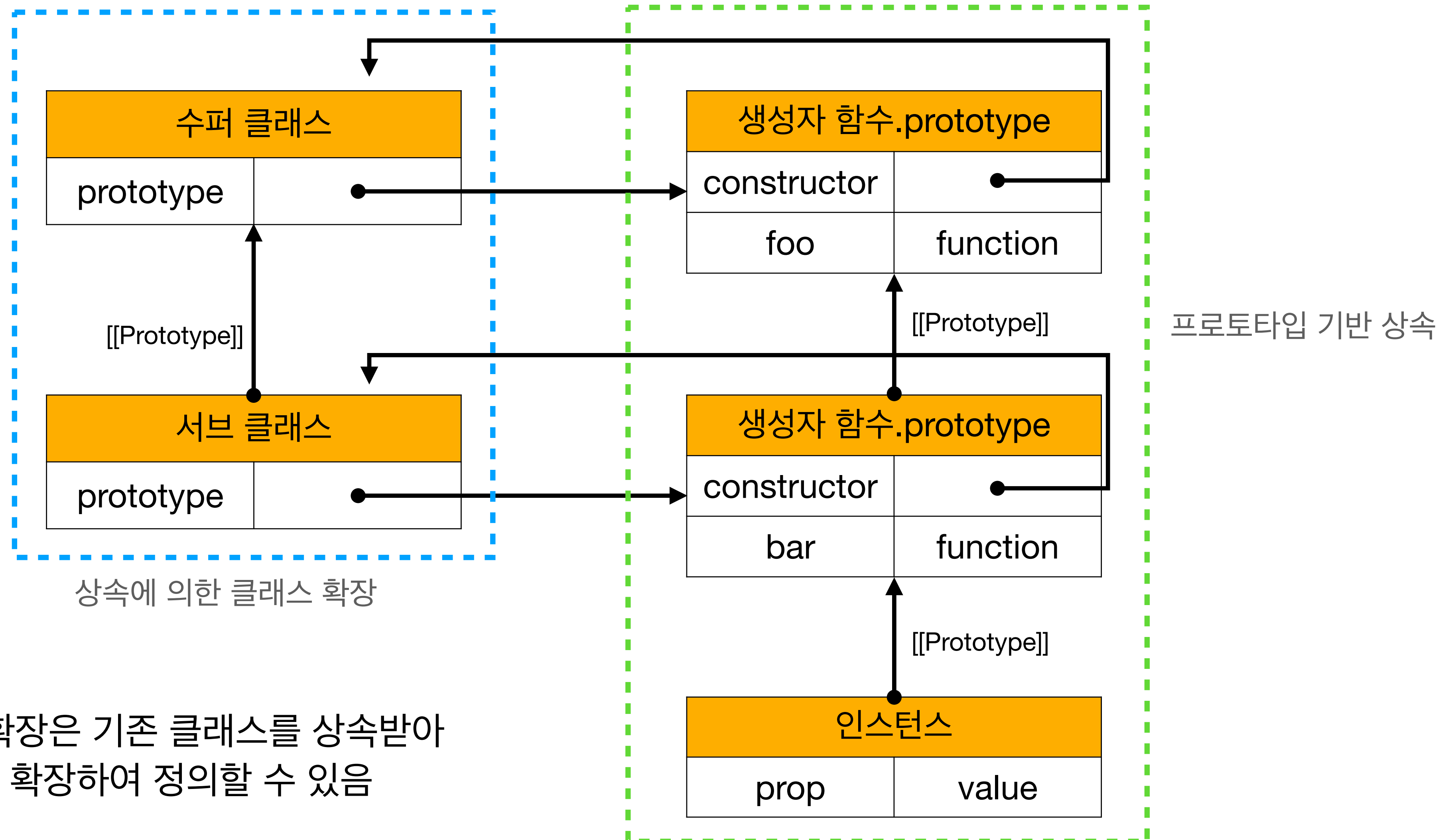
8. 상속에 의한 클래스 확장

클래스 상속과 생성자 함수 상속



8. 상속에 의한 클래스 확장

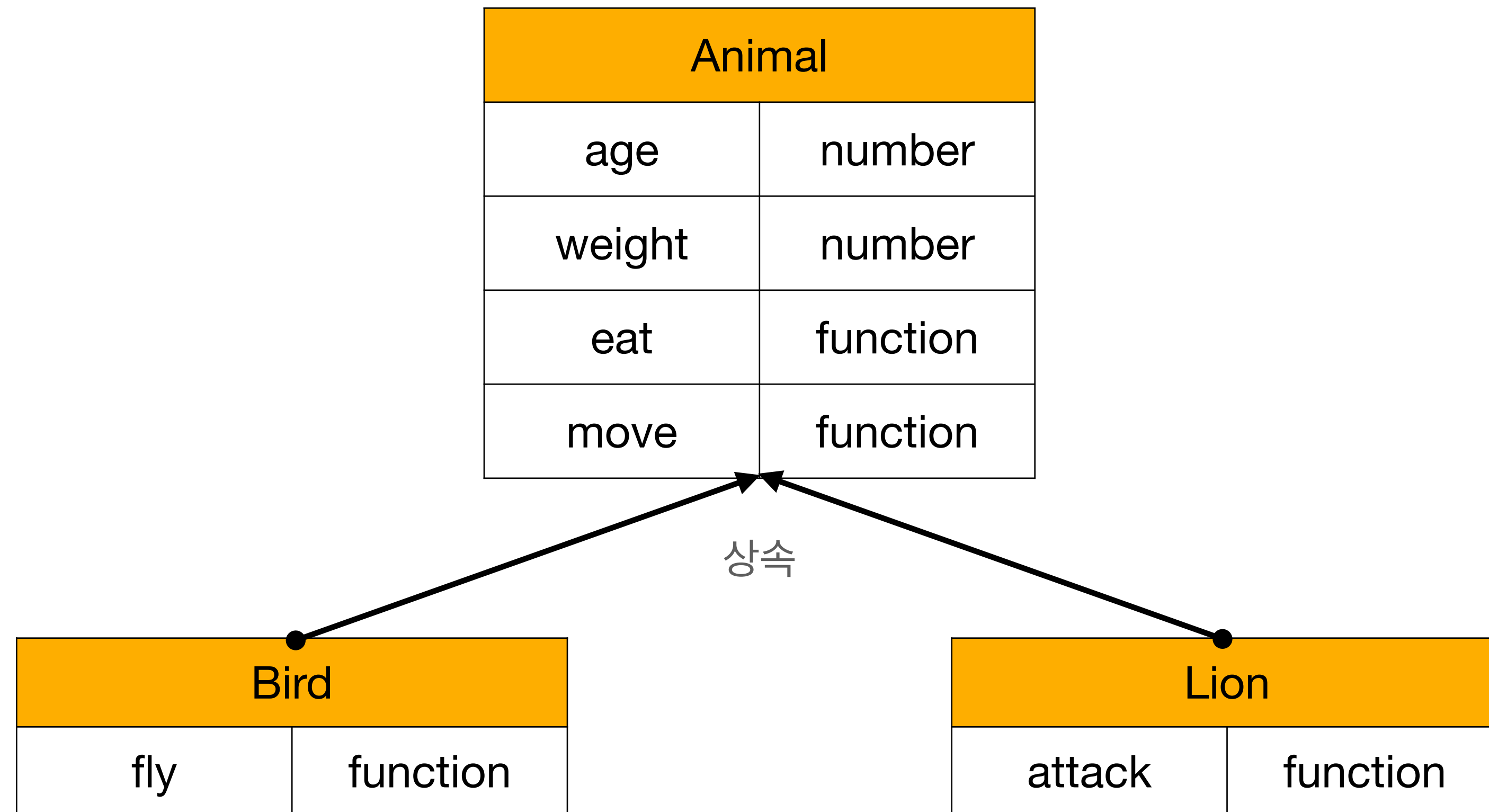
클래스 상속과 생성자 함수 상속



상속에 의한 클래스 확장은 기존 클래스를 상속받아 새로운 클래스를 확장하여 정의할 수 있음

8. 상속에 의한 클래스 확장

클래스 상속과 생성자 함수 상속



Animal을 확장하여 Bird, Lion을 새롭게 정의할 수 있다.

8. 상속에 의한 클래스 확장

클래스 상속과 생성자 함수 상속

```
class Animal {
  constructor(age, weight) {
    this.age = age;
    this.weight = weight;
  }

  eat() { return 'eat'; }

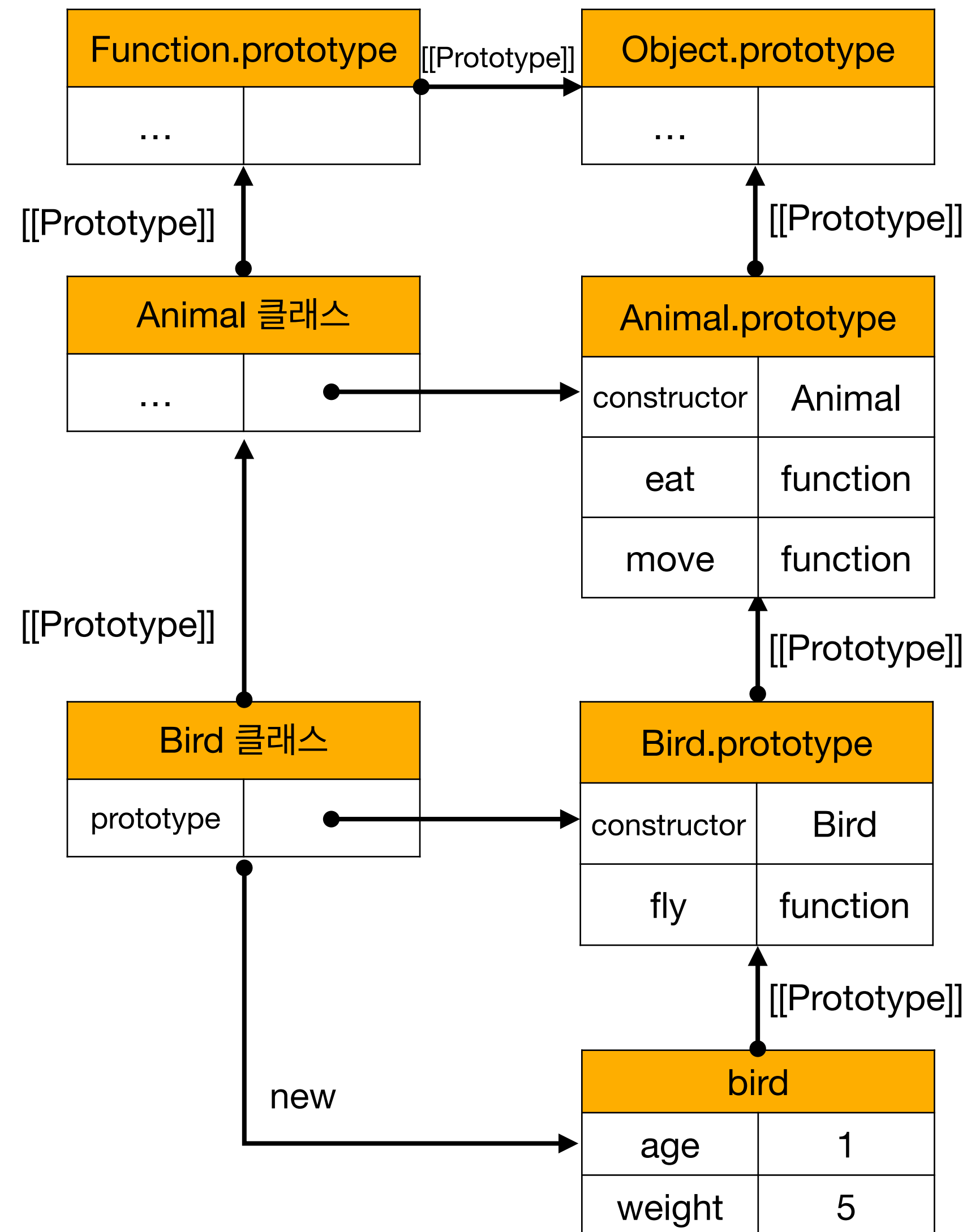
  move() { return 'move'; }
}

class Bird extends Animal {
  fly() { return 'fly'; }
}

const bird = new Bird(1, 5);

console.log(bird) // Bird {age: 1, weight: 5}
console.log(bird instanceof Bird) // true
console.log(bird instanceof Animal) // true

console.log(bird.eat()) // eat
console.log(bird.move()) // move
console.log(bird.fly()) // fly
```

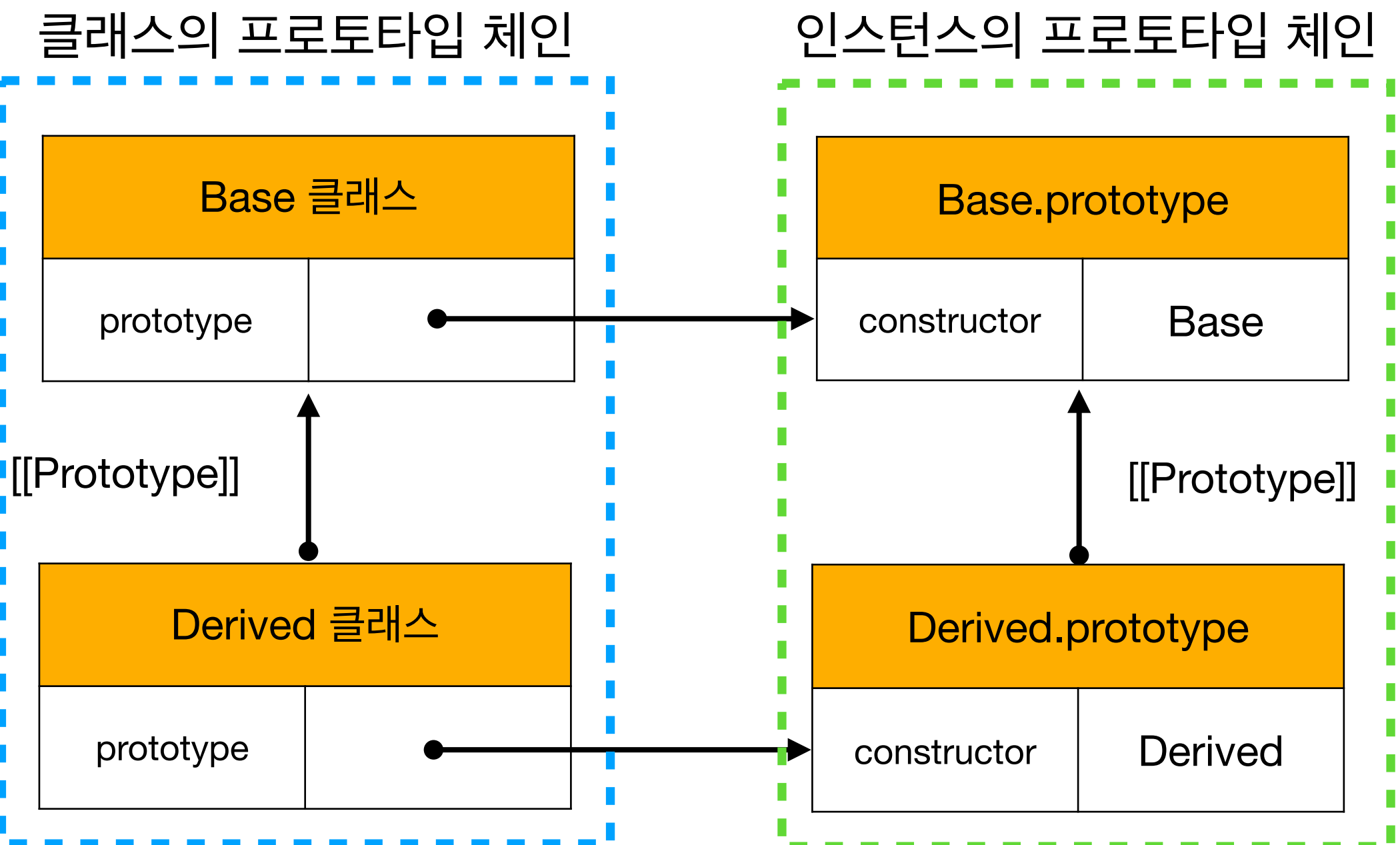


8. 상속에 의한 클래스 확장

extends 키워드

```
// 슈퍼(베이스/부모) 클래스
class Base {}

// 서브(파생/자식) 클래스
class Derived extends Base {}
```

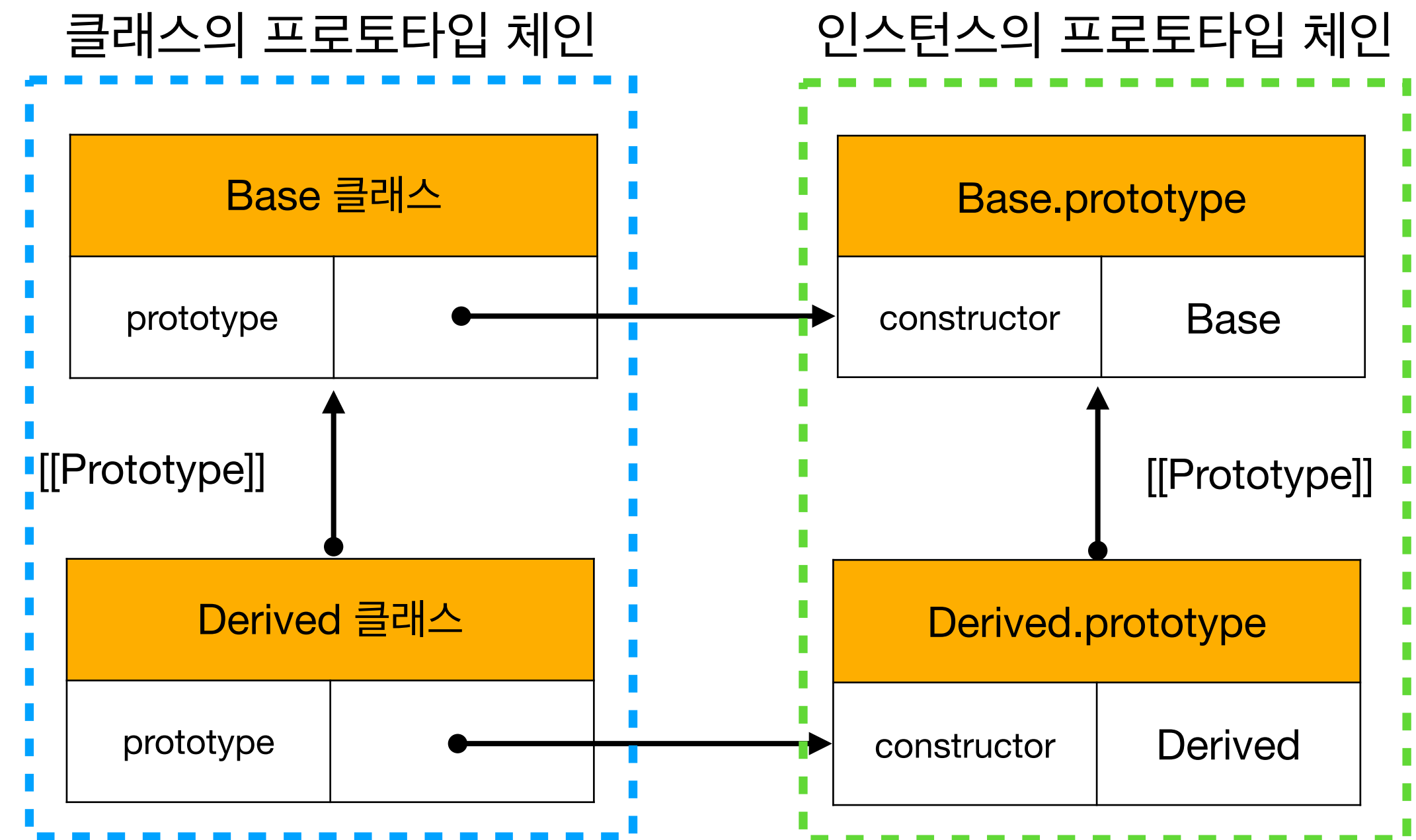


8. 상속에 의한 클래스 확장

extends 키워드

```
// 수퍼(베이스/부모) 클래스
class Base {}

// 서브(파생/자식) 클래스
class Derived extends Base {}
```



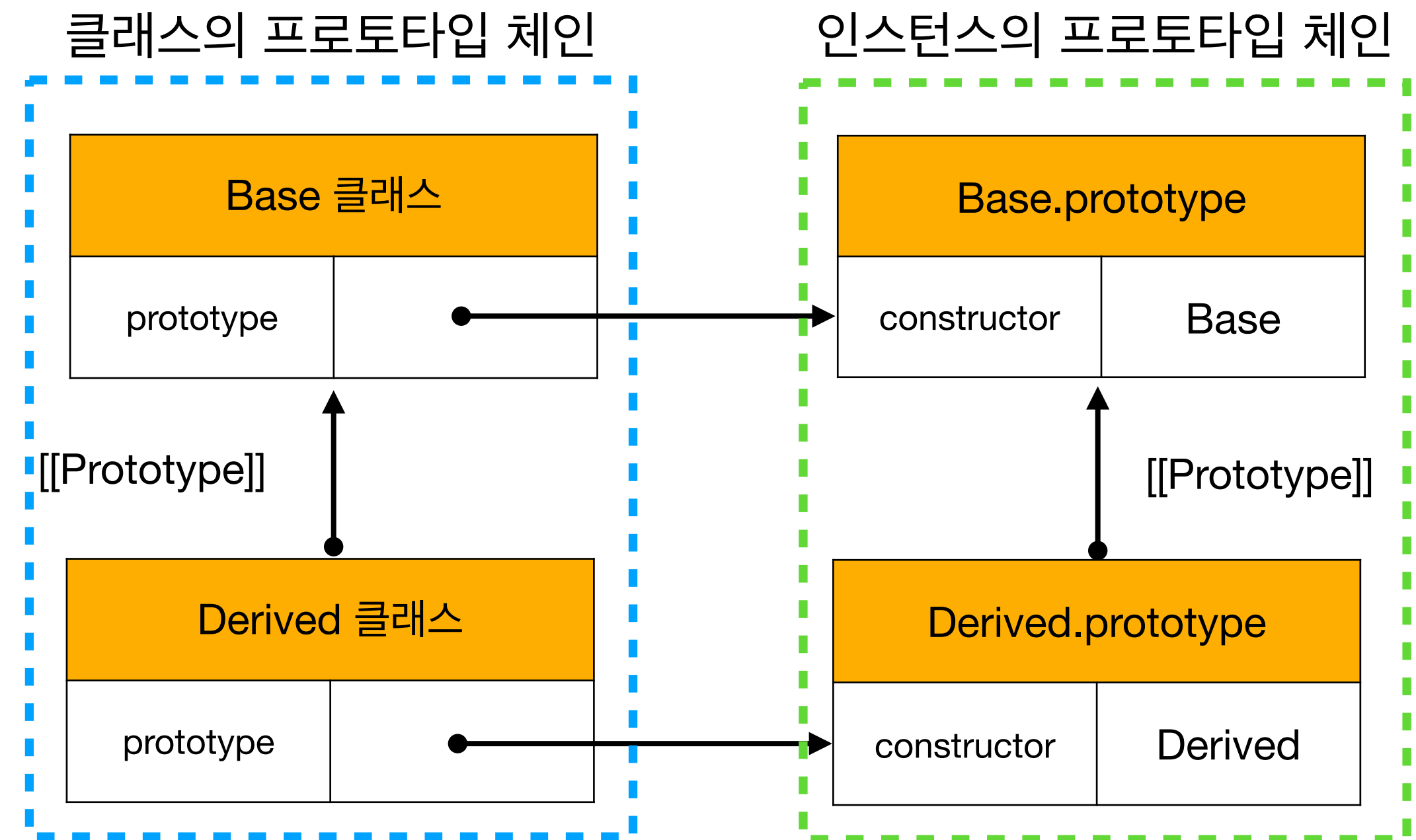
extends 키워드의 역할은 수퍼클래스와 서브클래스 간의 상속 관계를 설정하는 것이다.

8. 상속에 의한 클래스 확장

extends 키워드

```
// 수퍼(베이스/부모) 클래스
class Base {}

// 서브(파생/자식) 클래스
class Derived extends Base {}
```



extends 키워드의 역할은 수퍼클래스와 서브클래스 간의 상속 관계를 설정하는 것이다.

이를 통해 프로토타입 메서드, 정적 메서드 모두 상속이 가능하다.

8. 상속에 의한 클래스 확장

동적 상속

extends 키워드는 클래스뿐만 아니라 생성자 함수를 받아 클래스를 확장할 수도 있다.

```
function Base(a) {  
  this.a = a;  
}  
  
// 생성자 함수를 상속받는 서브클래스  
class Derived extends Base {}  
  
const derived = new Derived(1);  
  
console.log(derived); // Derived {a:1}
```

8. 상속에 의한 클래스 확장

동적 상속

`extends` 키워드 다음에는 클래스 뿐만 아니라 `[[Construct]]` 내부 메서드를 갖는, 함수 객체로 평가될 수 있는 모든 표현식을 사용할 수 있다.

8. 상속에 의한 클래스 확장

동적 상속

extends 키워드 다음에는 클래스 뿐만 아니라 [[Construct]] 내부 메서드를 갖는, 함수 객체로 평가될 수 있는 모든 표현식을 사용할 수 있다.

```
function Base1() {}

class Base2 {}

let condition = true;

class Derived extends (condition ? Base1 : Base2) {}

const derived = new Derived();

console.log(derived); // Derived {}

console.log(derived instanceof Base1); // true
console.log(derived instanceof Base2); // false
```

8. 상속에 의한 클래스 확장

서브 클래스의 constructor

```
class MyClass {}
```

```
// 위 코드를 아래와 같이 처리한다.
```

```
class MyClass {  
    constructor() {}  
}
```

8. 상속에 의한 클래스 확장

서브 클래스의 constructor



```
class Base {  
    constructor() {}  
}
```

```
class Derived extends Base {}
```

// 서브클래스는 아래와 같이 처리된다.

```
class Derived extends Base {  
    constructor(..args) { super(...args); }  
}
```

8. 상속에 의한 클래스 확장

super 키워드

8. 상속에 의한 클래스 확장

super 키워드

1. super를 호출하면 슈퍼클래스의 constructor를 호출한다.

8. 상속에 의한 클래스 확장

super 키워드

1. super를 호출하면 수퍼클래스의 constructor를 호출한다.
2. super를 참조하면 수퍼클래스의 메서드를 호출할 수 있다.

8. 상속에 의한 클래스 확장

super 키워드 - 호출할 때 주의점

```
// 슈퍼 클래스
class Base {}

// 서브 클래스
class Derived extends Base {
  constructor() {
    // ReferenceError: Must call super constructor in derived class before accessing 'this'
    // or returning from derived constructor
    console.log('constructor call');
  }
}

const derived = new Derived();
```

8. 상속에 의한 클래스 확장

super 키워드 - 호출할 때 주의점

1. 서브클래스에서 constructor를 생략하지 않는 경우, 반드시 super를 호출해야한다.

```
// 슈퍼 클래스
class Base {}

// 서브 클래스
class Derived extends Base {
  constructor() {
    // ReferenceError: Must call super constructor in derived class before accessing 'this'
    // or returning from derived constructor
    console.log('constructor call');
  }
}

const derived = new Derived();
```

8. 상속에 의한 클래스 확장

super 키워드 - 호출할 때 주의점

```
class Base {}

class Derived extends Base {
  constructor() {
    // ReferenceError: Must call super constructor in derived class before accessing 'this'
    // or returning from derived constructor
    this.a = 1;
    super();
  }
}

const derived = new Derived();
```

8. 상속에 의한 클래스 확장

super 키워드 - 호출할 때 주의점

2. 서브클래스의 constructor에서 super를 호출하기 전에는 this를 참조할 수 없다.

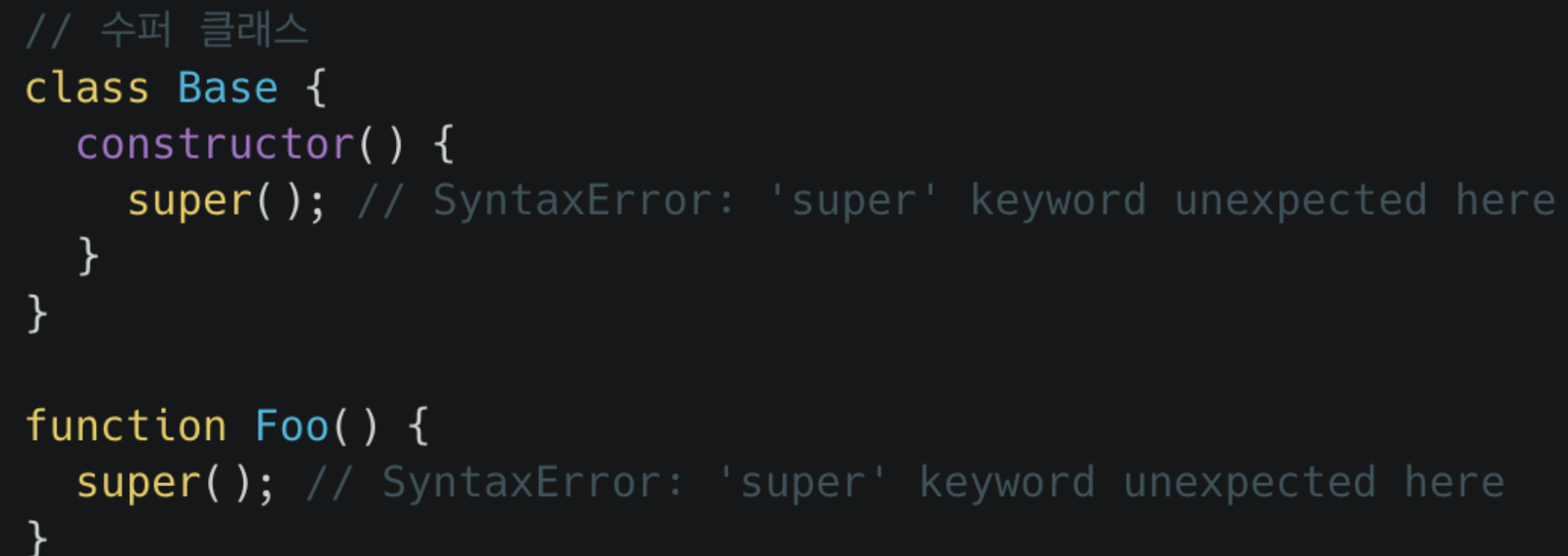
```
// 슈퍼 클래스
class Base {}

// 서브 클래스
class Derived extends Base {
  constructor() {
    // ReferenceError: Must call super constructor in derived class before accessing 'this'
    // or returning from derived constructor
    this.a = 1;
    super();
  }
}

const derived = new Derived();
```

8. 상속에 의한 클래스 확장

super 키워드 - 호출할 때 주의점



```
// 슈퍼 클래스
class Base {
  constructor() {
    super(); // SyntaxError: 'super' keyword unexpected here
  }
}

function Foo() {
  super(); // SyntaxError: 'super' keyword unexpected here
}
```

8. 상속에 의한 클래스 확장

super 키워드 - 호출할 때 주의점

3. super는 반드시 서브클래스의 constructor에서만 호출한다.

```
// 슈퍼 클래스
class Base {
  constructor() {
    super(); // SyntaxError: 'super' keyword unexpected here
  }
}

function Foo() {
  super(); // SyntaxError: 'super' keyword unexpected here
}
```

8. 상속에 의한 클래스 확장

super 키워드 - 참조할 때 주의점

```
// 슈퍼 클래스
class Base {
  constructor(name) {
    this.name = name;
  }

  sayHi() {
    return `Hi! ${this.name}`;
  }
}

class Derived extends Base {
  sayHi() {
    // super.sayHi는 슈퍼클래스의 프로토타입 메서드를 가리킨다.
    return `${super.sayHi()}. How are you doing?`;
  }
}

const derived = new Derived("Lee");
console.log(derived.sayHi()); // Hi! Lee. How are you doing?
```

8. 상속에 의한 클래스 확장

super 키워드 - 참조할 때 주의점

1. 서브클래스의 프로토타입 메서드 내에서 super.sayHi는 슈퍼클래스의 프로토타입 메서드 sayHi를 가리킨다.

```
// 슈퍼 클래스
class Base {
  constructor(name) {
    this.name = name;
  }

  sayHi() {
    return `Hi! ${this.name}`;
  }
}

class Derived extends Base {
  sayHi() {
    // super.sayHi는 슈퍼클래스의 프로토타입 메서드를 가리킨다.
    return `${super.sayHi()}. How are you doing?`;
  }
}

const derived = new Derived("Lee");
console.log(derived.sayHi()); // Hi! Lee. How are you doing?
```


8. 상속에 의한 클래스 확장

super 키워드 - 참조할 때 주의점

1. 서브클래스의 프로토타입 메서드 내에서 super.sayHi는 슈퍼클래스의 프로토타입 메서드 sayHi를 가리킨다.

```
// 슈퍼 클래스
class Base {
  constructor(name) {
    this.name = name;
  }

  sayHi() {
    return `Hi! ${this.name}`;
  }
}

class Derived extends Base {
  sayHi() {
    // super.sayHi는 슈퍼클래스의 프로토타입 메서드를 가리킨다.
    return `${super.sayHi()}. How are you doing?`;
  }
}

const derived = new Derived("Lee");
console.log(derived.sayHi()); // Hi! Lee. How are you doing?
```

8. 상속에 의한 클래스 확장

super 키워드 - 참조할 때 주의점

1. 서브클래스의 프로토타입 메서드 내에서 super.sayHi는 슈퍼클래스의 프로토타입 메서드 sayHi를 가리킨다.

```
// 슈퍼 클래스
class Base {
  constructor(name) {
    this.name = name;
  }

  sayHi() {
    return `Hi! ${this.name}`;
  }
}

class Derived extends Base {
  sayHi() {
    // super.sayHi는 슈퍼클래스의 프로토타입 메서드를 가리킨다.
    return `${super.sayHi()}. How are you doing?`;
  }
}

const derived = new Derived("Lee");
console.log(derived.sayHi()); // Hi! Lee. How are you doing?
```

8. 상속에 의한 클래스 확장

super 키워드 - 참조할 때 주의점

```
// 슈퍼 클래스
class Base {
  static sayHi() {
    return `Hi!`;
  }
}

class Derived extends Base {
  static sayHi() {
    return `${super.sayHi()}. How are you doing?`;
  }
}

console.log(Derived.sayHi()); // Hi! How are you doing?
```

8. 상속에 의한 클래스 확장

super 키워드 - 참조할 때 주의점

1. 서브클래스의 정적 메서드 내에서 super.sayHi는 수퍼클래스의 정적 메서드 sayHi를 가리킨다.

```
// 수퍼 클래스
class Base {
  static sayHi() {
    return `Hi!`;
  }
}

class Derived extends Base {
  static sayHi() {
    return `${super.sayHi()}. How are you doing?`;
  }
}

console.log(Derived.sayHi()); // Hi! How are you doing?
```

8. 상속에 의한 클래스 확장

super 키워드 - 참조할 때 주의점

1. 서브클래스의 정적 메서드 내에서 super.sayHi는 슈퍼클래스의 정적 메서드 sayHi를 가리킨다.

```
// 슈퍼 클래스
class Base {
  static sayHi() {
    return `Hi!`;
  }
}

class Derived extends Base {
  static sayHi() {
    return `${super.sayHi()}. How are you doing?`;
  }
}

console.log(Derived.sayHi()); // Hi! How are you doing?
```

8. 상속에 의한 클래스 확장

super 키워드 - 참조할 때 주의점

1. 서브클래스의 정적 메서드 내에서 super.sayHi는 수퍼클래스의 정적 메서드 sayHi를 가리킨다.



```
// 수퍼 클래스
class Base {
  static sayHi() {
    return `Hi!`;
  }
}

class Derived extends Base {
  static sayHi() {
    return `${super.sayHi()}. How are you doing?`;
  }
}

console.log(Derived.sayHi()); // Hi! How are you doing?
```

8. 상속에 의한 클래스 확장

표준 빌트인 생성자 함수 확장

```
class MyArray extends Array {  
  uniq() {  
    return this.filter((v, i, self) => self.indexOf(v) === i);  
  }  
  
  average() {  
    return this.reduce((pre, cur) => pre + cur, 0) / this.length;  
  }  
}  
  
const myArray = new MyArray(1, 1, 2, 3);  
console.log(myArray);  
  
console.log(myArray.uniq());  
console.log(myArray.average());
```

8. 상속에 의한 클래스 확장

표준 빌트인 생성자 함수 확장

```
class MyArray extends Array {  
  uniq() {  
    return this.filter((v, i, self) => self.indexOf(v) === i);  
  }  
  
  average() {  
    return this.reduce((pre, cur) => pre + cur, 0) / this.length;  
  }  
}  
  
const myArray = new MyArray(1, 1, 2, 3);  
console.log(myArray);  
  
console.log(myArray.uniq());  
console.log(myArray.average());
```