JavaScript

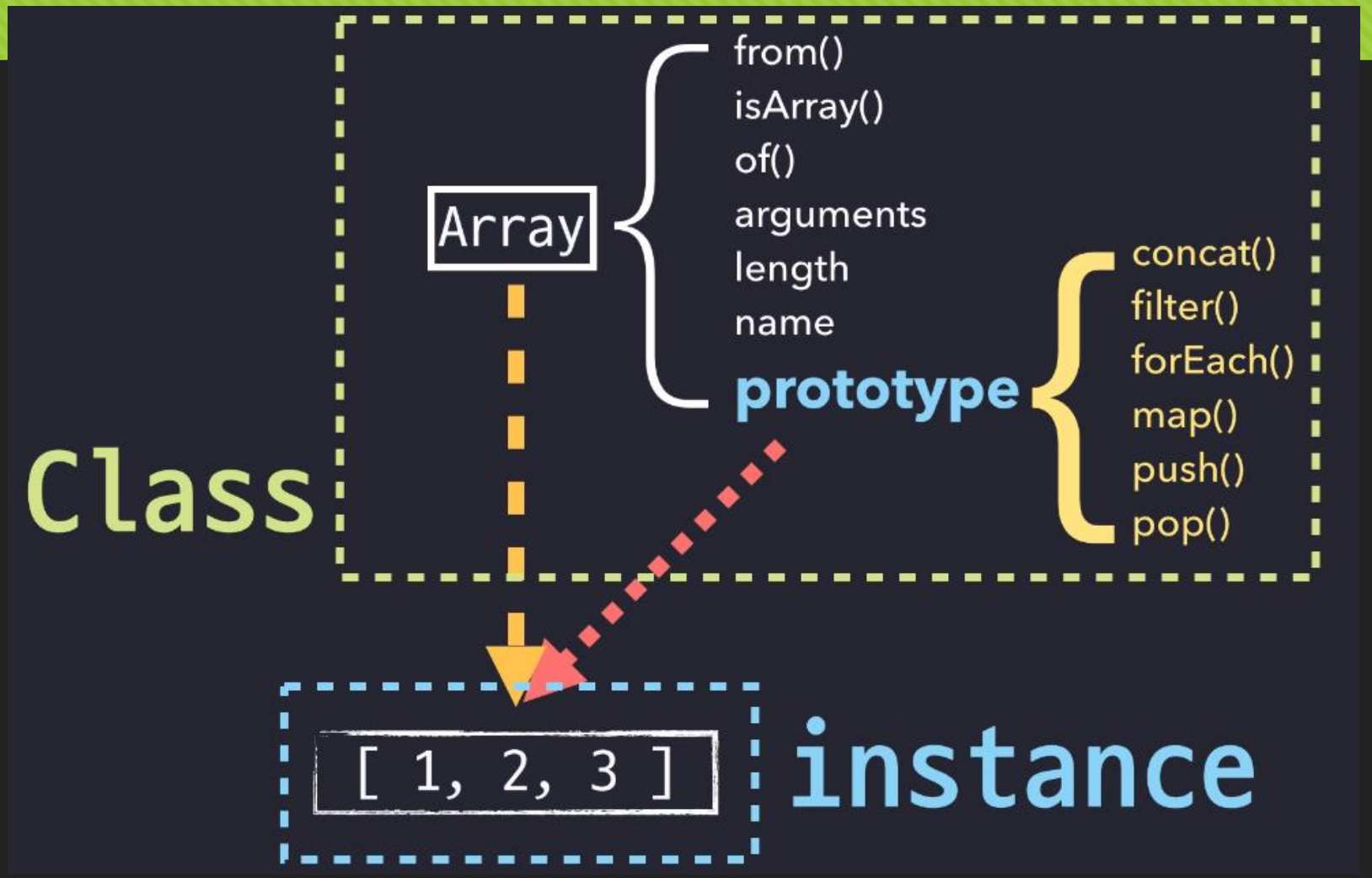
Class – 김근형 강사

생성자 함수 방식

- 생성자 함수 방식 선언
  - 자바 스크립트에서는 Class라는 개념이 존재하지 않는다.
  - 단 자바 스크립트 내에서는 생성자 함수라는 방식을 통해 객체를 생성했다.

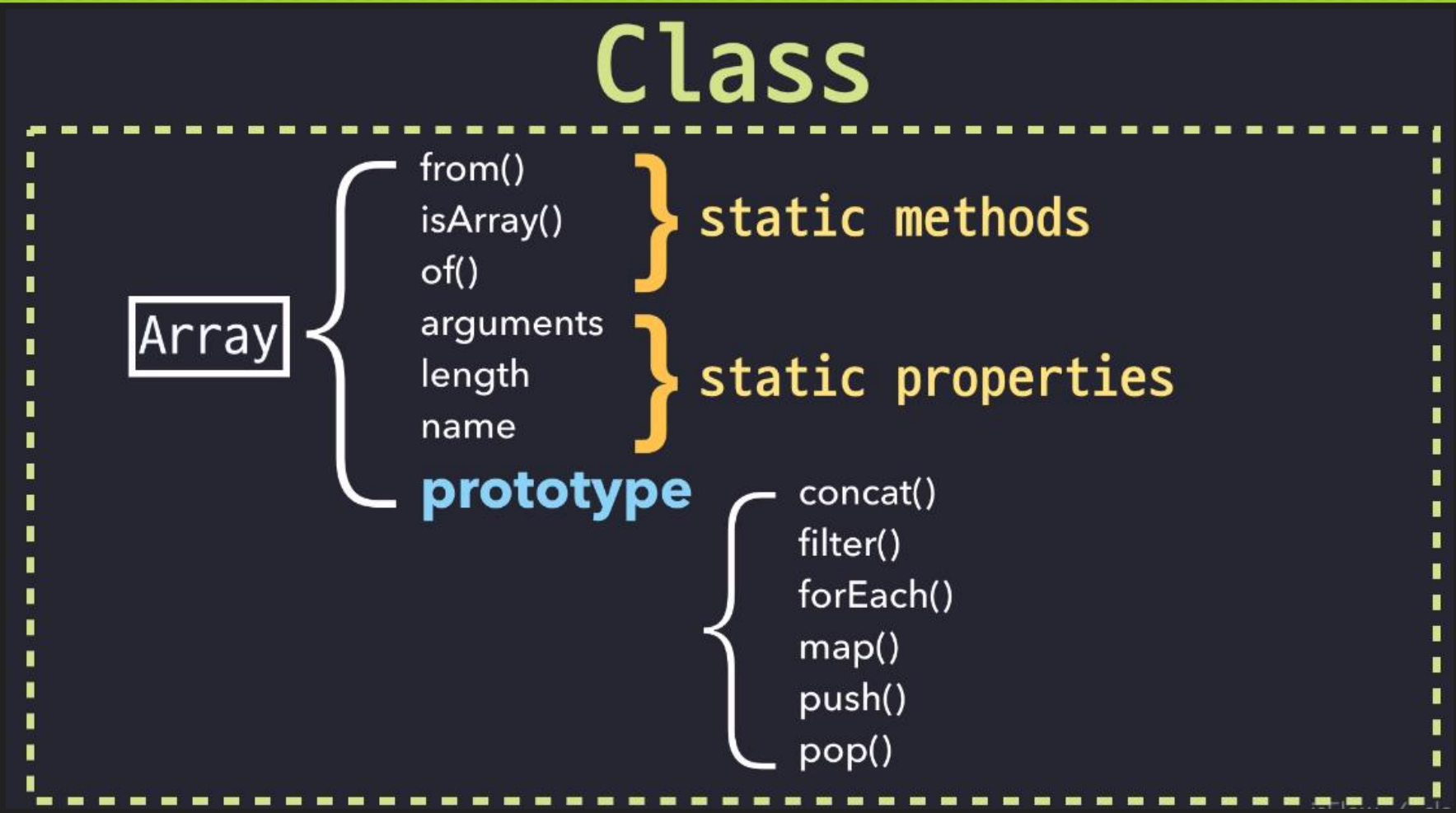
생성자 함수 방식

- 생성자 함수 방식 선언
  - 자바 스크립트에서는 Class라는 개념이 존재하지 않는다.
  - 단 자바 스크립트 내에서는 생성자 함수라는 방식을 통해 객체를 생성했다.



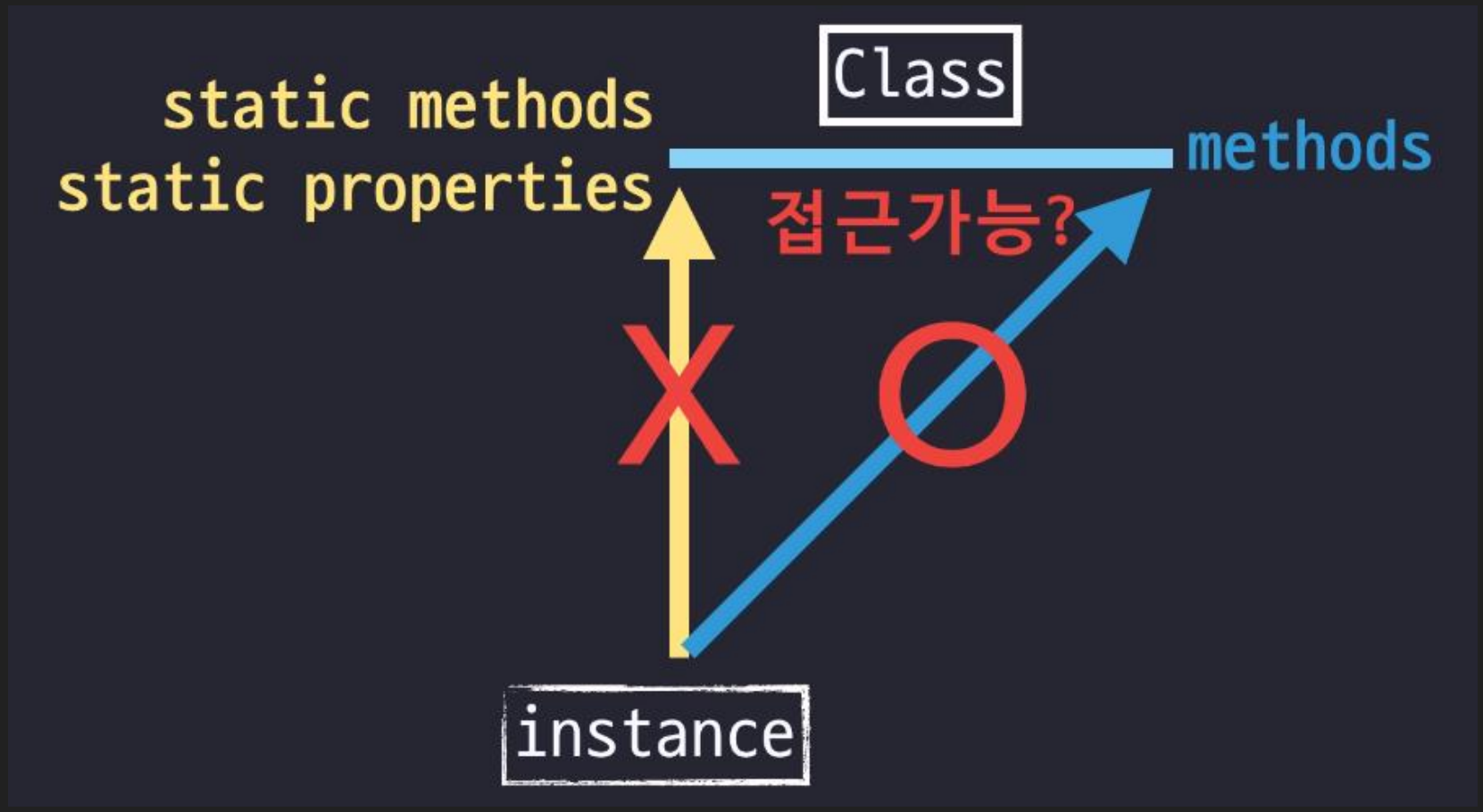
생성자 함수 방식

- 생성자 함수 방식 선언
  - 생성자 함수는 정적 메서드와 정적 프로퍼티를 갖고 있으며 프로토타입 메서드를 가진다.



생성자 함수 방식

- 생성자 함수 방식 선언
  - 스태틱 메서드의 접근은 불가능하지만 프로토타입 메서드는 접근이 가능하다.



생성자 함수 방식

```
function Person(name, age) {
  this._name = name;
  this._age = age;
}
```

static method

```
Person.getInformations = function(instance) {
  return {
    name: instance._name,
    age: instance._age
  };
}
```

(prototype) method

```
Person.prototype.getName = function() {
  return this._name;
}
```

(prototype) method

```
Person.prototype.getAge = function() {
  return this._age;
}
```

생성자 함수 방식

```
var gomu = new Person('고무', 30);
```

OK	`console.log(gomu.getName());`
OK	`console.log(gomu.getAge());`
ERROR	`console.log(gomu.getInformations(gomu));`
OK	`console.log(Person.getInformations(gomu));`

생성자 함수 방식

- 생성자 함수 방식 선언의 단점
  - 생성자 함수 방식을 통해 자바 스크립트 내에서 클래스처럼 만들 수 있지만 가독성이 좋지 않았다.
  - 사용하기가 불편하고 코드가 길어지면서 다른 언어의 객체 생성 방식과 많은 거리를 두게 되었다.
  - ECMA2015 이후에 이런 방식의 변경이 강하게 요구되었고 기존의 생성자 함수 방식의 단점을 고치고 다른 언어들이 사용하는 Class 방식에 대해 연구되기 시작했다.

Class

- Class
  - ECMA2015 부터 Class라는 개념이 등장했다.
  - Class는 기존의 생성자 함수를 통한 객체 생성 방식을 대체하기 위해 만들어졌다.

```
class PersonClass {
    // PersonType 생성자와 동일합니다.
    constructor(name) {
        this.name = name;
    }
    // PersonType.prototype.sayName과 동일합니다.
    sayName() {
        console.log(this.name);
    }
}
```

Class

- Class 특징
  - 생성자 함수 선언 방식에서의 복잡함과 가독성을 Class 선언 방식에서 간소화 시킴으로서 많은 이점을 가져왔다.+
  - 클래스에서 제공되는 여러가지 기능들을 통해 이전에 생성자 함수에서 하지못했던 기능들을 추가적으로 스는 것이 가능하다.
  - 함수선언 방식에서는 호이스팅이 일어나지만 클래스 선언 방식은 호이스팅이 일어나지 않는다. 즉, 클래스를 사용하기 위해서는 클래스를 먼저 선언 해야 하며, 그렇지 않으면 에러가 발생한다.
  - class와 function의 클래스 차이점은 function은 글로벌 오브젝트로 설정되지만 class는 글로벌 오브젝트로 설정되지 않는다. 따라서 window.class 방식으로 접근이 불가능하다.(undefined 반환)
  - Class 오브젝트의 프로퍼티는 for문으로 열거할 수 없다.

Class

- Class 정의
  - 클래스를 정의하는 방법은 총 두가지 방법이 존재한다
  - Class 선언 방식

```
class ClassName(){ … }
class ClassName extends SuperClassName(){ … }
```

  - Class 표현식

```
let className = class { }
let className = class InnerName { }
let className = class extends SuperName { }
let className = class InnerName extends SuperName { }
```

Class

- Class 선언 방식 / Class 표현식 예제

```
class Member {
  getName() {
    return "이름";
  }
};
let obj = new Member();

console.log(obj.getName());
```

```
let Member = class {
  getName() {
    return "이름";
  }
};
let obj = new Member();
console.log(obj.getName());
```

Class

- Class 선언 방식 / Class 표현식
  - 위에서 설명하는 class 는 function 오브젝트, String 오브젝트와 같이 하나의 오브젝트 타입이다.
  - Class 선언은 함수 선언과 달리 호이스팅이 되지 않는다.
  - Class 표현식 형태는 익명 클래스 형태와 가명 클래스 형태의 두 형태로 만들 수 있다.
  - Class 표현식 형태로 선언 시 Class 재사용이 불가능하다.

Class

- 인스턴스 생성
  - 인스턴스 생성은 생성자 함수를 통해 인스턴스 생성할 때와 동일하다.

let/const/var instanceName = new ClassName([parameter, ...])

```
let obj = new Class1();
console.log(obj.getName());
```

```
class1
```

Property

- property
  - Class 내에서는 Property를 설정하는 것이 가능하다.
  - Property를 설정하는 방법은 (1) Class 내에 직접 Property를 선언하거나 (2) 메서드 내에서 this를 통해 프로퍼티를 바인딩 하는 방법이 있다.
    - 1번 방식은 Chrome 72 이상이거나 최신 Node.js에서만 동작한다.
  - 이런식의 클래스 내에서 선언된 프로퍼티를 클래스 필드라 부른다.
  - 바인딩 시에 외부에 데이터를 가져올 경우 메서드를 통해 데이터를 받아 외부 데이터를 넣을 수 있다.

```
class Class1{
    // 프로퍼티 직접 선언
    age = 23;
    // 메서드를 통해 선언
    setName(name){
        this.name = name;
    }

    getName(){
        return this.name;
    }

    getAge(){
        return this.age;
    }
}

let obj = new Class1();
obj.setName("홍길동");
console.log(obj.getName()); // 홍길동
console.log(obj.getAge());  // 23
```

Method

- 메서드(Method)
  - 생성자 함수에서 메서드를 만드는 방법과 달리 Class 에서 메서드를 선언하는 방식은 다르다
  - 단 기존의 생성자 함수의 메서드 기능과 Class 함수에서의 메서드 기능은 전부 동일하다.
  - class에서 메서드 작성시에는 function과 : 이 들어가지 않고 이름만 작성한다.
  - 단 { } 끝에 세미콜론(;)은 작성해도 되고 하지 않아도 상관은 없다.

Method

- 생성자 함수에서의 Method와 Class에서의 Method 차이

```
class Class1{
    method1(){
        return "class method1";
    }
    method2(){
        return "class method2";
    }
}

function Class2(){}

Class2.prototype.method1 = function(){
        return "function method1";
    }
Class2.prototype.method2 = function(){
        return "function method2";
    }

let obj1 = new Class1();
let obj2 = new Class2();
console.log(obj1.method1());
console.log(obj1.method2());
console.log(obj2.method1());
console.log(obj2.method2());
```

Method

- Class 메서드와 this의 활용
  - 생성자 함수 안에서 함수를 선언하고 this를 그 안에 선언했을 시 글로벌에 선언된 프로퍼티를 가르쳤었다.
  - 하지만 Class 내부에서 함수를 선언하고 그 안에서 this를 선언할 경우 해당 this는 글로벌에 선언된 프로퍼티를 가르치지 않으며 에러가 난다.
  - 생성자 함수 선언에서는 접근 가능한 글로벌 함수로의 this 접근이 Class에서 적용되지 않으며 Class 내에서 에러가 나지 않게 하기 위해서는 값을 받은 인스턴스에 따로 해당 이름의 프로퍼티를 선언해 주어야만 정상적으로 동작한다.

Method

- Class 메서드와 this의 활용

```
var name = "임꺽정";

var Class1 = function(){};

Class1.prototype.getMethod2 = function(){
    let f = function(){
        return this.name;
    }
    return f;
}

var obj = new Class1();
var func = obj.getMethod2();
console.log(func()); // 임꺽정
```

```
var name = "임꺽정";

class Class1{
    setMethod(name){
        this.name = name;
    }
    getMethod1(){
        return this.name;
    }
    getMethod2(){
        let f = function(){
            return this.name;
        }
        return f;
    }
}

let obj = new Class1();
obj.setMethod('홍길동');
console.log(obj.getMethod1());
let func = obj.getMethod2()
// console.log(func());  // 에러
```

Method

- 동적 Method 추가
  - Method를 추가하는 방법은 이전의 생성자 함수 선언과 같이 prototype을 이용해 선언할 수 있다.
  - Class에서 선언된 Method 들은 전부 prototype 에 저장되기 때문이다.
  - 하지만 이러한 추가의 경우 생성된 인스턴스에서 추가한 메서드를 공유할 수 있도록 엔진이 처리하기 때문에 부하가 상당히 걸릴 수 있다.

```
class Foo{
    getName(){
        return '홍길동'
    }
}

Foo.prototype.getAge = function(){
    return 23;
}

var foo = new Foo();
console.dir(foo);
console.log(foo.getName()); // 홍길동
console.log(foo.getAge()); // 23
```

```
▼Foo i
  ▼__proto__:
    ▶getAge: ƒ ()
    ▶constructor: class Foo
    ▶getName: ƒ getName()
    ▶__proto__: Object
홍길동
23
```

Constructor

- 생성자(Constructor)
  - constructor는 클래스 인스턴스를 생성하고 생성한 인스턴스를 초기화하는 역할을 한다.

    new Member() -> Member.prototype.constructor

  - 클래스에서 constructor를 생성하지 않으면 prototype에 존재하는 constructor가 실행된다.(디폴트 constructor)

Constructor

- 생성자(Constructor)

```
class Member {
  constructor(name){
    this.name = name;
  }
  getName() {
    return this.name;
  }
}
let memberObj = new Member("스포츠");
console.log(memberObj.getName());
```

```
class Member {
  constructor(name, age){
    this.name = name;
    this.age = age;
  }
  getName() {
    return this.name;
  }
  getAge() {
    return this.age;
  }
}
let memberObj = new Member("홍길동", 23);
console.log(memberObj.getName());
console.log(memberObj.getAge());
```

Constructor

- constructor 반환 값 변경
  - constructor에 일반적으로 return문을 작성하지 않으며, 생성한 인스턴스를 반환한다.
  - 한편 return 문으로 인스턴스 이외의 값을 반환할 수 있다.
  - 하지만 constructor에서 숫자나 문자를 반환하면 이를 무시하고 인스턴스를 반환한다.

```
class Member {
  constructor(){
    return 1;
  }
  getName(){
    return "이름";
  }
};
let memberObj = new Member();
console.log(memberObj.getName());
```

이름

```
class Member {
  constructor(){
    return {name: "홍길동"};
  }
  getName(){
    return "이름";
  }
}
let memberObj = new Member();

console.log(memberObj.name);
console.log(memberObj.getName);
```

홍길동

undefined

Getter Setter

- Getter Setter
  - Class 내에서 Getter와 Setter 메서드 작성이 가능하다.
  - get 과 set을 메서드 앞에 씀으로써 getter와 setter 메서드를 만들 수 있다.
  - getter 메서드는 매개변수가 없어야 하고 setter 메서드는 매개변수가 하나만 존재해야 한다.

```
class Member {
  get name() {
    return "이름";
  }
};
let memberObj = new Member();
console.log(memberObj.name);
```

이름

```
class Member {
  set name(name) {
    this._name = name;
  }
  get name() {
    return this._name;
  }
};
let memberObj = new Member();
memberObj.name = "홍길동";

console.log(memberObj.name);
```

홍길동

Static

- Static 메소드
  - Static 메소드는 정적 메소드로써 객체에서는 사용되지 않고 클래스 자체에서 호출되는 메서드를 의미한다.
  - static 메서드 내에서는 this를 사용할 수 없다.
  - static 메소드는 클래스의 인스턴스 필요없이 호출 가능하다.
  - 또한 클래스의 인스턴스에서 static 메소드를 호출 할 수 없다.

```
lass Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }

    static distance(a, b) {
        const dx = a.x - b.x;
        const dy = a.y - b.y;
        return Math.sqrt(dx*dx + dy*dy
    }


onst p1 = new Point(5, 5);
onst p2 = new Point(10, 10);
onsole.log(Point.distance(p1, p2));

ar p3 = new Point(1, 1);
/ p3.distance(p1, p2);  // error
```

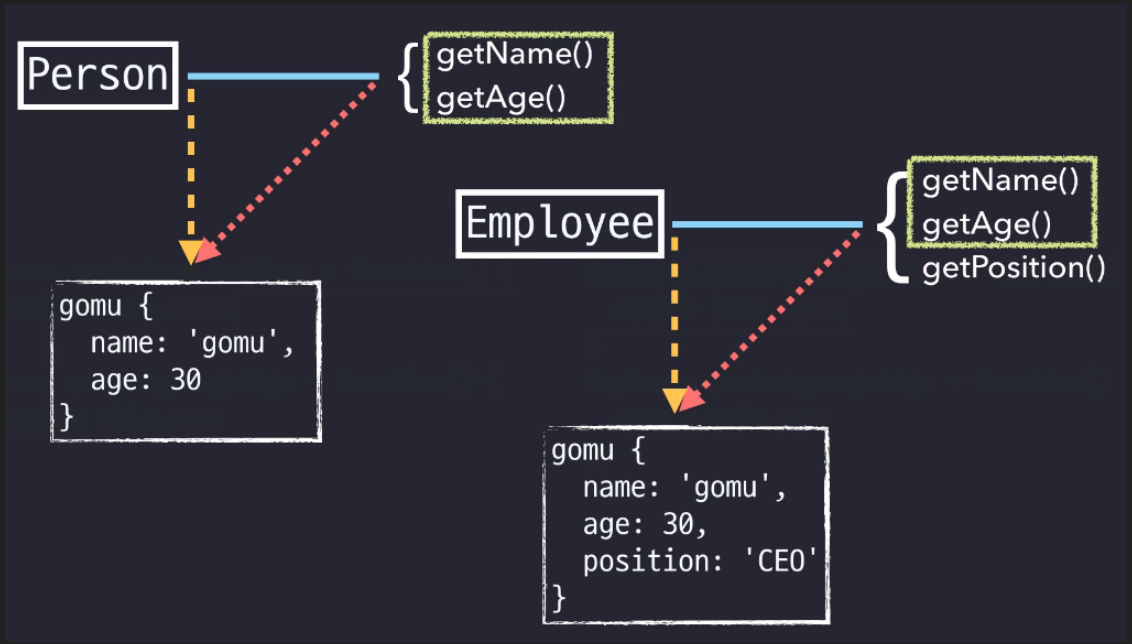
Static

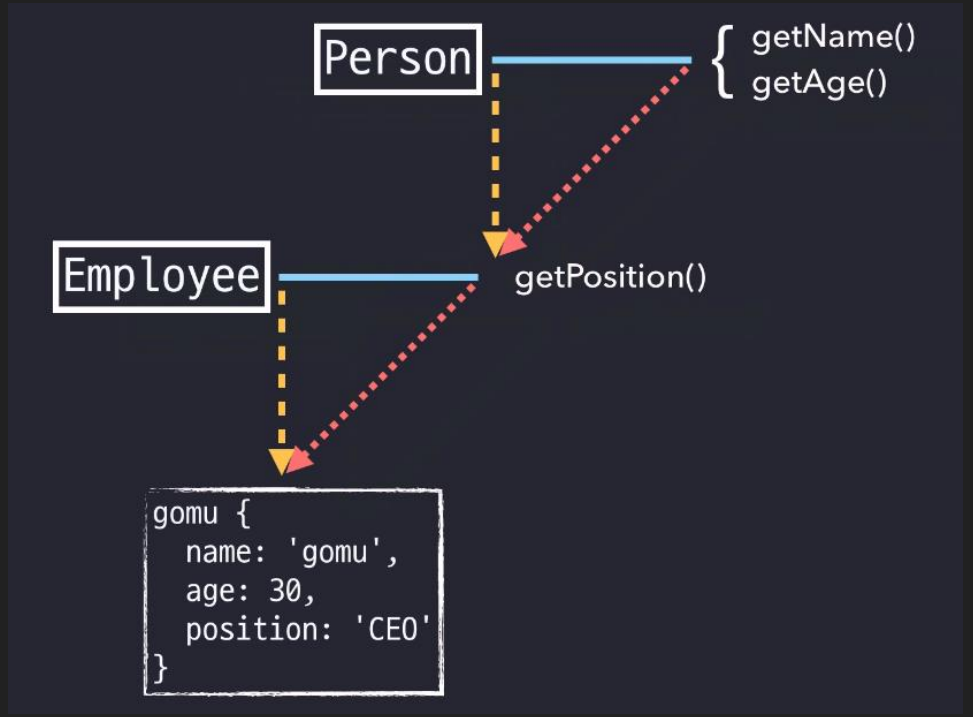
- Static 메서드
  - 클래스 내의 staic 메서드는 this.constructor를 통해 접근이 가능하다.

```
class Member{
    getStatic(){
        this.constructor.memberName();
        console.log("static 메서드 호출");
    }

    static memberName(){
        console.log('내 이름은 홍길동');
    }
}

var member = new Member();
member.getStatic();
```

상속(Inheritance)

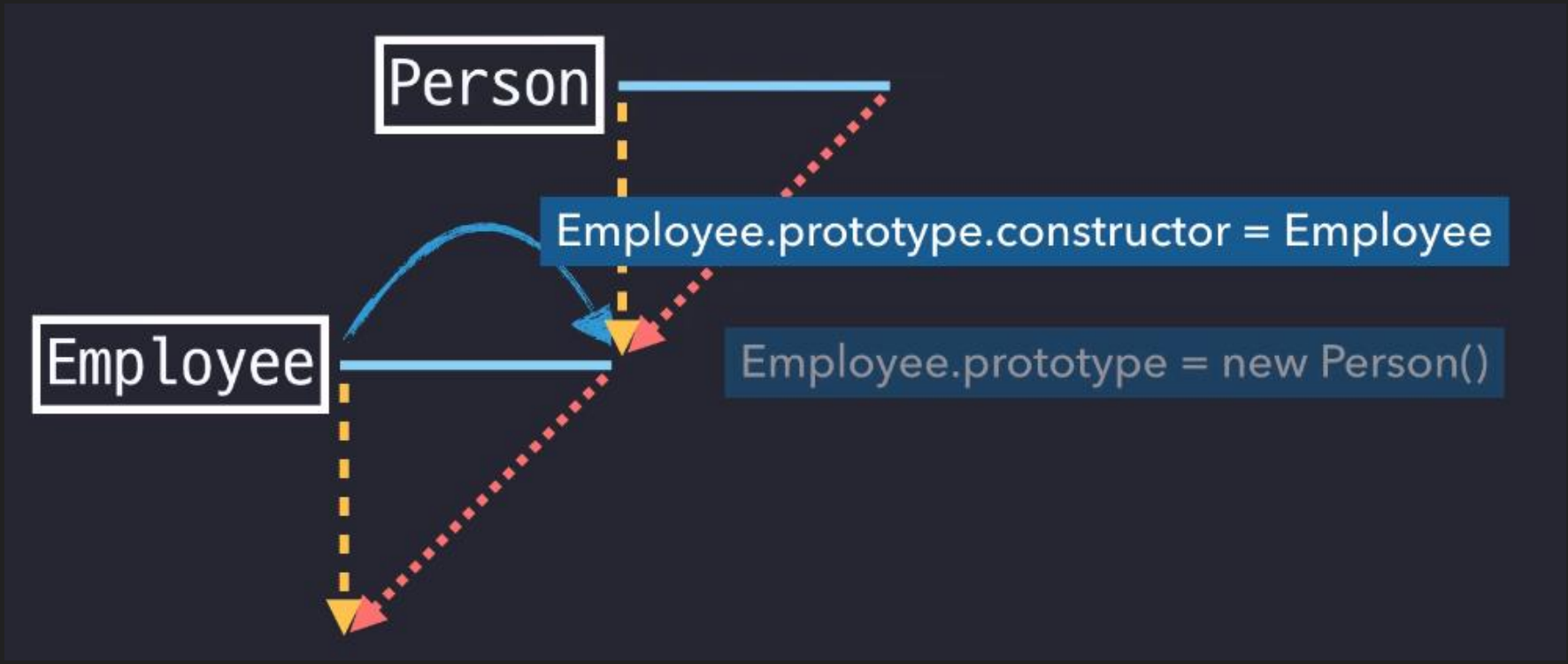
- 생성자 함수에서 상속하는 방식





상속(Inheritance)

- 생성자 함수에서 상속하는 방식



상속(Inheritance)

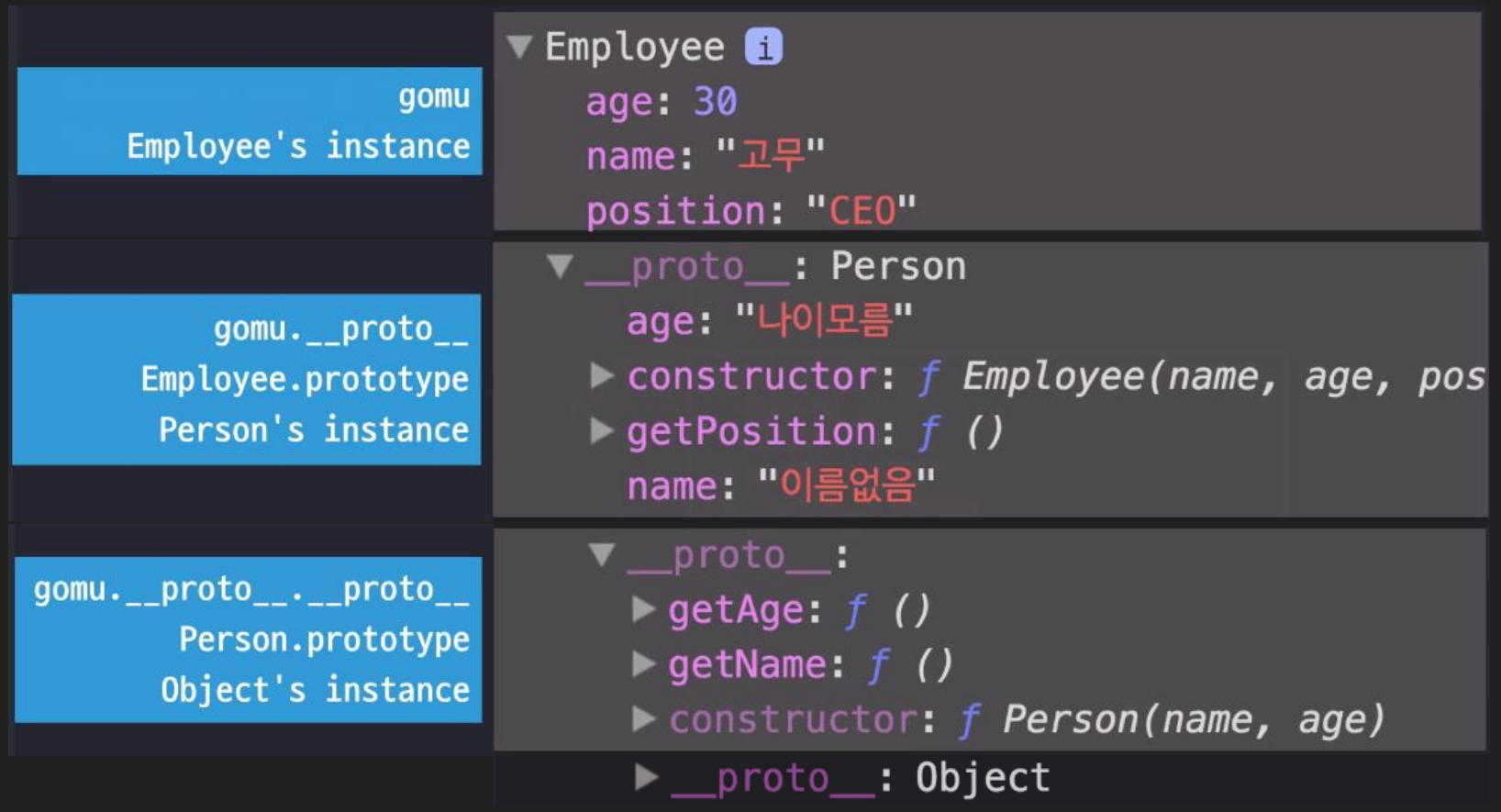
- 생성자 함수에서 상속하는 방식

```
// 6-2-2
function Person(name, age) {
  this.name = name || '이름없음';
  this.age = age || '나이모름';
}
Person.prototype.getName = function() {
  return this.name;
}
Person.prototype.getAge = function() {
  return this.age;
}

function Employee(name, age, position) {
  this.name = name || '이름없음';
  this.age = age || '나이모름';
  this.position = position || '직책모름';
}
Employee.prototype = new Person();
Employee.prototype.constructor = Employee;
Employee.prototype.getPosition = function() {
  return this.position;
}
var gomu = new Employee('고무', 30, 'CEO');
console.dir(gomu);
```

```
▼ Employee i
    age: 30
    name: "고무"
    position: "CEO"
  ▼ __proto__: Person
      age: "나이모름"
    ▶ constructor: ƒ Employee(name, age, pos
    ▶ getPosition: ƒ ()
      name: "이름없음"
    ▼ __proto__:
      ▶ getAge: ƒ ()
      ▶ getName: ƒ ()
      ▶ constructor: ƒ Person(name, age)
      ▶ __proto__: Object
```

상속(Inheritance)

- 생성자 함수에서 상속하는 방식



상속(Inheritance)

- 생성자 함수에서 상속하는 방식
  - 생성자 함수를 통해 상속을 구현하려고 할 경우 상당히 복잡한 메커니즘을 거친다.
  - prototype과 constructor의 참조를 통해 상속을 구현하는 방식이 대표적
  - 그래서 일반 생성자 함수에서 나타내는 상속은 가독성이 떨어지고 많은 오류를 낳았다.
  - class 에서는 이러한 상속을 간단하게 할 수 있도록 기능이 추가되었다.

상속(Inheritance)

- Extends 함수
  - 클래스의 상속은 Extends라는 함수를 Class선언부에 사용함으로써 상속을 표현할 수 있다.

```
class subclass extends superclass {  }
```

```
class Sports {
  constructor(member){
    this.member = member;
  }
  getMember(){
    return this.member;
  }
};
class Soccer extends Sports {
  setMember(member){
    this.member = member;
  }
};
```

```
let obj = new Soccer(11);
console.log(obj.getMember()); // 11
obj.setMember(40);
console.log(obj.getMember()); // 40
```

상속(Inheritance)

- 프로토타입 기반 클래스 상속

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

Animal.prototype.speak1 = function () {
    console.log(this.name + ' makes a noise.');
}

class Dog extends Animal {
    speak2() {
        console.log(this.name + ' barks.');
    }
}

let d = new Dog('바둑이');
d.speak1();
d.speak2();
```

```
바둑이 makes a noise.
바둑이 barks.
```

상속(Inheritance)

- 일반 객체 클래스 상속
  - 일반 객체는 extends 키워드를 통해 상속할 수 없다. 상속하고 싶다면 Object.setPrototypeOf 메소드를 사용하여 상속해야 한다.

```
var Animal = {
    speak() {
        console.log(this.name + ' makes a noise.');
    }
};

class Dog {
    constructor(name) {
        this.name = name;
    }

    bark() {
        console.log(this.name + ' barks.');
    }
}

Object.setPrototypeOf(Dog.prototype, Animal);
var d = new Dog('Mitzie');
d.speak();
d.bark();
```

```
Mitzie makes a noise.
Mitzie barks.
```

상속(Inheritance)

- Super
  - super는 상속받은 super클래스의 필드나 메서드를 참조할 때 쓰는 키워드이다.
  - 실제 메서드 오버라이딩이 발생하거나 강제로 super클래스의 메서드를 참조할 경우 쓴다.

상속(Inheritance)

- 프로토타입 기반 생성자 함수에서 사용 방법
  - 프로토타입 기반의 클래스에서 부모 클래스의 메소드를 호출하기 위해서 call과 같은 함수를 사용해야 한다.

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

Cat.prototype.speak = function () {
    console.log(this.name + ' makes a noise.');
};

function Lion(name) {
    // `super()` 호출
    Cat.call(this, name);
}

// `Cat` 클래스 상속
Lion.prototype = new Cat();
Lion.prototype.constructor = Lion;

// `speak()` 메서드 오버라이드
Lion.prototype.speak = function () {
    Cat.prototype.speak.call(this);
    console.log(this.name + ' roars.');
};

var lion = new Lion("BIG");
lion.speak();
```

```
BIG makes a noise.
BIG roars.
```

상속(Inheritance)

- ES6 클래스 기반의 부모 클래스 메소드 호출
  - 클래스는 super 메소드를 사용하여 부모 클래스의 메소드를 호출 할 수 있다.

```
class Cat {
    constructor(name) {
        this.name = name;
    }

    speak() {
        console.log(this.name + ' makes a noise.');
    }
}

class Lion extends Cat {
    speak() {
        super.speak();
        console.log(this.name + ' roars.');
    }
}

var lion = new Lion("BIG");
lion.speak();
```

```
BIG makes a noise.
BIG roars.
```

상속(Inheritance)

- 메서드 오버라이딩
  - 자식 객체가 부모 객체의 기능을 그대로 덮어 씌우는 것을 뜻함
  - 오버라이드 메서드는 부모 메서드의 이름과 매개변수를 자색 클래스 내에서 동일하게 맞춰 선언하는 것을 의미한다.
  - 자식 클래스로 객체를 선언하고 메서드를 사용하게 되면 부모 클래스의 메서드가 아닌 자식 클래스의 메서드가 우선 실행된다.

상속(Inheritance)

- 메서드 오버라이딩

```
class Sports {
  setGround(ground){
    this.ground = ground;
  }
};
class Soccer extends Sports {
  setGround(ground){
    //super.setGround();
    this.ground = ground+1;
  }
};
let obj = new Soccer(11);
obj.setGround("상암구장");
console.log(obj.ground);
```

상암구장1

상속(Inheritance)

- constructor 오버라이드
  - 생성자를 오버라이딩 하기 위해서는 반드시 super를 통해 상위 클래스의 생성자를 호출해야만 한다.

```
class Sports {
  constructor(member){
    this.member = member;
    console.log(this.member);
  }
};
class Soccer extends Sports {
  constructor(member){
    super(member);
    this.member = 456;
    console.log(this.member);
  }
};
let obj = new Soccer(123);
```

```
123
456
```

상속(Inheritance)

- constructor 오버라이드
  - 서브 클래스와 슈퍼 클래스 양쪽에 constructor를 작성하지 않아도 인스턴스가 생성된다. 이때 default constructor를 사용한다.
  - 서브 클래스에 constructor를 작성하지 않고 슈퍼 클래스에 constructor를 작성하면 서브클래스의 default constructor 클래스가 호출되고 슈퍼 클래스의 constructor가 호출된다.
  - 서브 클래스의 constructor를 작성하고 슈퍼 클래스에 constructor를 작성하지 않으면 서브클래스의 constructor가 호출되지만 constructor에서 에러가 난다.
  - 서브 클래스의 슈퍼클래스 양쪽에 constructor를 작성하면 서브클래스의 constructor가 호출되지만 constructor에서 에러가 발생한다.

상속(Inheritance)

- 일반 객체 에서 super의 사용
  - 일반 객체 내에서도 super의 사용이 가능하다.

```
let Sports = {
  getTitle(){
    console.log("Sports");
  }
};
let Soccer = {
  getTitle(){
    super.getTitle();
    console.log("Soccer");
  }
};
Object.setPrototypeOf(Soccer, Sports);
Soccer.getTitle();
```

```
Sports
Soccer
```

상속(Inheritance)

- 빌트인 프로젝트 상속
  - 빌트인 오브젝트를 상속받게 되면 빌트인 오브젝트의 메서드를 마치 서브 클래스에서 선언한 것처럼 사용할 수 있게 된다.

```
class ExtendArray extends Array {
  constructor(){
    super();
  }
  getTotal(){
    let total = 0;
    for (var value of this){
      total += value;
    };
    return total;
  }
};

let obj = new ExtendArray();
obj.push(10, 20);

console.log(obj.getTotal());
```

```
30
```

ETC

- Class 에서 Computed Name Property 사용하기

```
let type = "Type";
class Sports {
  static ["get" + type](kind){
    return kind ? "스포츠" : "음악";
  }
}
console.log(Sports["get" + type](1));
```

스포츠

ETC

- this
  - static 메서드에서 this는 클래스 오브젝트를 참조한다.

```
class Sports {
  static setGround(ground){
      this.ground = ground;
  }
  static getGround(){
      return this.ground;
  }

  getWord() {
      return 'word';
  }
};

Sports.prototype.setMember = function(member) {
    Sports.setGround(member);
    this.member = member;
}
```

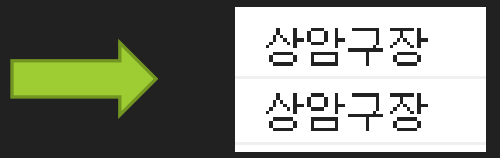
```
Sports.setGround("상암구장");
console.log(Sports.getGround());

let ss = new Sports();
ss.setMember("인천구청");
console.log(ss.getWord());
console.log(Sports.getGround());
```

```
상암구장
word
인천구청
```

ETC

- this
  - constructor 안에서 this.constructor.name() 형태로 정적 메서드를 호출할 수 있다.

```
class Sports{
  constructor(){
    console.log(Sports.getGround());
    console.log(this.constructor.getGround());
  }
  static getGround(){
    return "상암구장";
  }
};
let obj = new Sports();
```



ETC

- class 제너레이터
  - 클래스 안에 제너레이터 함수를 작성할 수 있다.
  - 클래스 안에 작성한 제너레이터 함수는 prototype에 연결된다.
  - 그래서 정적 메서드로 호출할 수 없고 인스턴스를 생성하여 호출해야 한다.

```
class Member{
  *gen() {
    yield 10;
    yield 20;
  }
};
let obj = new Member();
let genObj = obj.gen();

console.log(genObj.next());
console.log(genObj.next());
```

```
▼Object i
    done: false
    value: 10
  ▶__proto__: Object
▼Object i
    done: false
    value: 20
  ▶__proto__: Object
```

ETC

- new.target
  - new.target은 메타(meta) 프로퍼티로 생성자 함수와 클래스에서 constructor를 참조한다.
  - new 연산자로 인스턴스를 생성하지 않으면 new.target 값은 undefined 가 된다.

```
let sports = function(){
  console.log(new.target);
}
sports();
new sports();
```

```
undefined
f (){
  console.log(new.target);
}
```

ETC

- new.target – name property
  - 클래스, 함수, 오브젝트에 name 프로퍼티가 존재하며 이름이 설정된다.

```
class Sports {
  constructor(){
    console.log("Sports:", new.target.name);
  }
};
class Soccer extends Sports {
  constructor(){
    super();
      console.log("Soccer:", new.target.name);
  }
};
let sportsObj = new Sports();
let soccerObj = new Soccer();
```

```
Sports: Sports
Sports: Soccer
Soccer: Soccer
```

ETC

- Image 오브젝트 상속
  - DOM(Document Object Model)에서 제공하는 Image 인터페이스, Audio 인터페이스 등을 상속받을 수 있다.
  - Image같은 인터페이스는 웹 페이지에 png 파일과 같은 이미지 파일을 표현하기 위한 속성을 제공하지만 자체를 그대로 사용할 수 없고 상속을 통해서 쓸 수가 있다.

ETC

```
class ExtendsImage extends Image{
  constructor() {
    super();
  }
  setProperty(image){
    this.src = image.src;
    this.alt = image.alt;
    this.title = image.title;
  }
};
let imageObj = new ExtendsImage();

let properties = {
  src: "file/rainbow.png",
  alt: "나무와 집이 있고 그 위에 무지개가 있는 모습",
  title: "무지개"
};

imageObj.setProperty(properties);
document.querySelector("body").appendChild(imageObj);
```