



Week1. Core JavaScript

목차

1. 자바스크립트를 왜 사용하는가?
 - a. HTML
 - b. CSS
 - c. JavaScript
 - d. JavaScript의 특징
 - i. 객체 지향 프로그래밍이란?
 - ii. 프로토타입 기반 객체 기반의 스크립트 언어
 - iii. ProtoType
 1. 프로토타입 체인
 2. 프로토타입 상속
 - a. 속성 상속
 - b. 메소드 상속
 3. 프로토타입 오버라이딩
 4. 프로토타입 새로딩
 - 자바스크립트에서의 Class
 - Class inheritance 와 Prototypal inheritance 는 같은가?
 - iv. 인터프리터 언어
2. 브라우저 렌더링 작동 원리와 DOM
 - a. 브라우저 엔진
3. 자바스크립트의 작동방식
 - a. 실행 컨텍스트
 - b. 동작방식 & 용어 설명
4. 비동기처리 (callback, promise, async/await)
 - a. 비동기 처리를 해야하는 이유
 - b. 비동기 처리 종류
 - c. 설명 강의

자바스크립트를 왜 사용하는가?

HTML



개발자 밈

HTML이란 Hyper Text Markup Language의 약어 입니다.

이름에서 알 수 있듯 HyperText의 문서를 만드는 언어인데요, 문서가 화면에 표시되는 형식을 나타내거나 데이터의 논리적인 구조를 명기하는 언어인 마크업 언어 입니다.

B2B 데이터 서비스의 글로벌 리더 기업 테크 타겟(Tech Target)

은 HTML이 프로그래밍 언어가 아니라는 것에 다음과 같은 논거를 가져왔습니다.

“**프로그래밍 언어**는 컴퓨터의 기능, 특히 CPU의 기능인 메모리에서 데이터를 읽고, 해당 데이터에 대해 조건부 논리를 수행해야 하며, 번개와 같은 속도로 반복적으로 논리를 실행하는데 프로그래밍 할 수 있는 방법을 제공해야 한다.”

‘HTML은 변수에 값을 할당하는 방법이 없다.’

HTML은 프로그래밍 언어일까 아닐까?

HTML(Hypertext Markup Language)은 프로그래밍 언어일까 아닐까? 웹 개발자와 그래픽 디자이너들에게 물으면 프로그래밍 언어라고 답할지도 모르겠지만, 정답은 프로그래밍어가 아니다. HTML은 변수, 조건문, 반복 루프가 없기 때문이다. HTML은 왜 프로그래밍 언어가 아닐까 B2B 데이터 서비스의 글로벌 리더 기업 테크 타겟(Tech Target)은 이 같은 질문에 상세한 논거를 들었다. 프로그래밍 언어는 컴퓨터의 기능, 특히 CPU의 기능인 스메모리에서 데

CW <https://www.cwn.kr/news/articleView.html?idxno=10998>



그러나 해당 부분에 대해 여러 논쟁이 있는데 아래 글을 참고하시면 됩니다.

HTML은 프로그래밍 언어인가? 라는 논쟁보다 중요한 것

더이상 HTML 논란은,, 네이버,,,

Y <https://yceffort.kr/2021/10/is-html-programming-language>

Is html programming language

#html #programming

<https://yceffort.kr/2021/10/is-html-programming-language>

CSS

Cascading Style Sheets의 약자입니다.

CSS는 사용자에게 문서를 표시하는 방법을 지정하는 언어입니다.

마크업 언어를 사용하여 구성된 텍스트 파일입니다.

규칙 기반 언어입니다.

JavaScript

HTML과 CSS와 같은 마크업 언어로 이루어진 페이지는 ‘정적’일 수 밖에 없습니다.

자바스크립트는 이러한 정적인 페이지를 동적으로 만들어주는 역할을 합니다.


웹에서의 동작을 정의하고 다양한 기능과 인터렉션을 추가하여 더 좋은 UX를 제공할 수 있습니다.

JavaScript의 특징

1. 프로토타입 기반 객체 기반의 스크립트 언어이다.
2. 인터프리터 언어이다.
3. 객체 지향형 프로그래밍과 함수형 프로그래밍을 모두 표현할 수 있다.

왜 자바스크립트를 사용할까?

자바스크립트(javascript)란 자바스크립트는 객체(object) 기반의 스크립트 언어이다. 스크립트는 웹페이지의 HTML 안에 작성할 수 있는데, 웹페이지를 불러올 때 스크립트가 자동으로 실행된다. 자바스크립트는 ‘웹페이지에 생동감을 불어넣기 위해’ 만들어진 프로그래밍 언어이다. HTML로는 웹의 내용을 작성하고, CSS로는 웹을 디자인하며, 자바스크립트로는 웹의 동작을 구현할 수 있다. 예를들어 웹페이지중 로그인 화면이 있다고 가정해보자. 이런 화면이

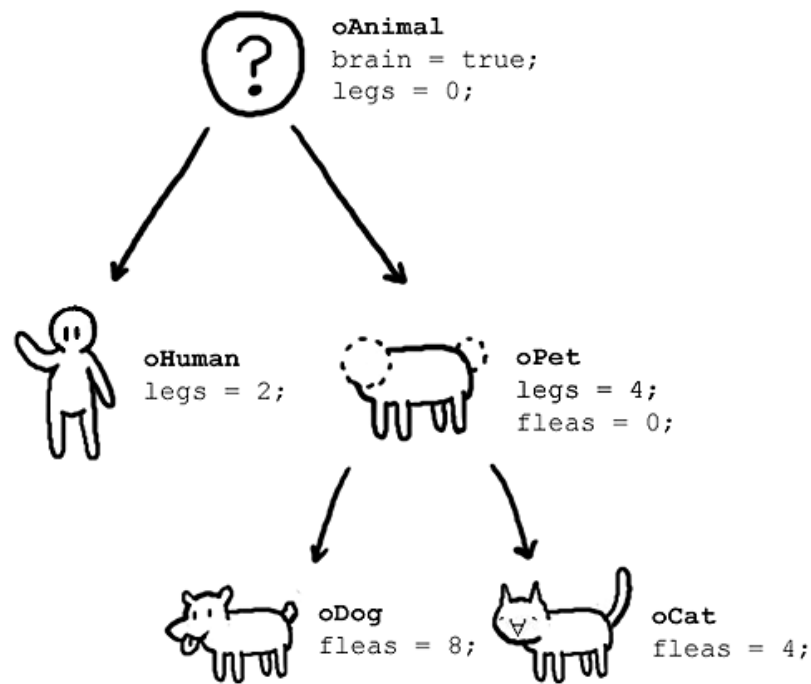
 <https://hgserver.tistory.com/68>



객체 지향 프로그래밍이란?

실세계에 존재하고 인지하고 있는 객체를 소프트웨어의 세계에서 표현하기 위해 핵심적인 개념 또는 기능만을 추출하는 추상화를 통해 모델링하려는 프로그래밍 패러다임.

필요한 데이터를 추상화시켜서 **상태와 행위를 가진 객체** 로 만들고, 각 객체들간의 상호작용을 통해 로직을 구성한다.



출처https://www.reddit.com/r/ProgrammerHumor/comments/418x95/theory_vs_reality/

객체지향 프로그래밍이란?

객체 지향 프로그래밍이란? 객체 지향 프로그래밍 (Object-Oriented Programming, OOP...

 <https://jongminfire.dev/객체지향-프로그래밍이란>



프로토타입 기반 객체 기반의 스크립트 언어

객체 지향 언어는 크게 클래스 기반 객체 지향 언어와 프로토타입 기반의 객체 지향 언어로 나뉩니다.

Class

객체의 형식이 정의된 클래스라는 개념을 가지고 해당 클래스를 사용해서 객체를 만들어 냅니다.

구체적으로 클래스란 같은 종류의 집단에 속하는 속성과 행위를 정의하는 것으로 객체지향 프로그램의

기본적인 사용자 정의 데이터형 이라고 할 수 있습니다.

클래스는 객체 생성에 사용되는 패턴이며 `new` 연산자를 통한 인스턴스화 과정이 필요합니다.


- **클래스**: 연관되어 있는 변수와 메서드의 집합
- **객체**: 클래스에 선언된 모양 그대로의 실체, 구현 대상
- **인스턴스**: 일반적으로 실행 중인 임의의 프로세스, 클래스의 현재 생성된 오브젝트

```
/* 클래스 */
public class Animal {
    ...
}
/* 객체와 인스턴스 */
public class Main {
    public static void main(String[] args) {
        Animal cat, dog; // '객체'

        // 인스턴스화
        cat = new Animal(); // cat은 Animal 클래스의 '인스턴스' (객체를 메모리에 할당)
        dog = new Animal(); // dog은 Animal 클래스의 '인스턴스' (객체를 메모리에 할당)
    }
}
```

[Java] 클래스, 객체, 인스턴스의 차이 - Heee's Development Blog

Step by step goes a long way.

 <https://gmlwjd9405.github.io/2018/09/17/class-object-instance.html>

Prototype

자바스크립트의 모든 객체는 자신의 부모 역할을 담당하는 객체와 연결되어 있는데 이 부모 객체를 ‘**프로토타입**’이라고 한다.

자바스크립트에는 클래스라는 개념이 없습니다. 그래서 기존의 만들어진 객체를 복사하여 새로운 객체를 생성하는 프로토타입 기반의 언어입니다.

그렇기 때문에 생성된 객체 역시 또 다른 객체의 원형이 될 수 있습니다.

자바스크립트 객체가 만들어지기 위해 그 객체의 부모가 되는 것.

그리고 Object 객체의 프로토타입을 기반으로 확장됩니다.

즉, 프로토타입 체인의 종점은 `Object.prototype` 이게 됩니다.

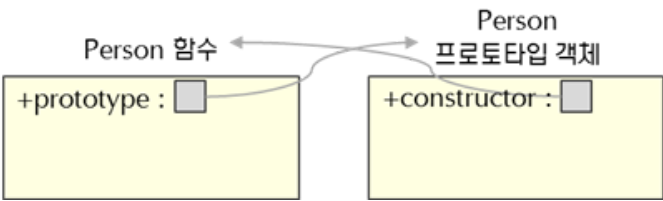
JavaScript에서는 함수를 정의하면 함수의 멤버로 `prototype` 속성이 있습니다.

```
interface ObjectConstructor {
    new(value?: any): Object;
    (): any;
    (value: any): any;

    /** A reference to the prototype for a class of objects. */
    readonly prototype: Object;

    /**
     * Returns the prototype of an object.
     * @param o The object that references the prototype.
     */
    getPrototypeOf(o: any): any;

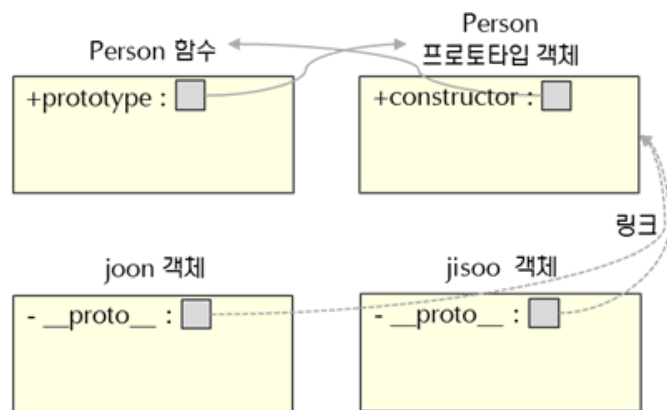
    .....
}
```



해당 속성은 다른 곳에 생성된 함수 이름의 프로토타입 객체를 참조합니다.

프로토타입 객체 멤버인 `constructor` 는 함수를 참조하는 내부 구조를 가집니다.

즉, `constructor` 는 자신을 생성한 생성자 함수를 가리키는 역할을 합니다.



그리고 new를 통해 생성되는 생성된 모든 객체는 프로토타입 객체의 원형이 되는 객체가 됩니다.

또한 객체 안에는 `__proto__` 라는 비표준 속성이 존재하는데, 이 속성은 객체 생성을 위해 사용된 원형인 프로토타입 객체를 숨은 링크로 참조합니다.

결국 프로토타입은 객체가 생성되고, 원형을 의미하는 프로토타입 객체를 참조하는 proto 속성을 활용한 링크와 같습니다.

```
const myObj = {
  a: 20
}
console.log(myObj.__proto__ === Object.prototype)
// --> true
```

프로토타입 상속

생성자 함수 사용

```
// 부모 생성자 함수
function Parent(name) {
  this.name = name;
}

// 부모 객체 메소드
Parent.prototype.sayName = function() {
  console.log(`My name is ${this.name}.`);
};

// 자식 생성자 함수
function Child(name, age) {
  Parent.call(this, name); // 부모 생성자 함수 호출
  this.age = age;
}

// 자식 객체의 프로토타입 설정
Child.prototype = Object.create(Parent.prototype);
// 해당 코드를 사용하지 않으면, constructor 프로퍼티는 부모 생성자 함수를 가리킴.
Child.prototype.constructor = Child;

// 자식 객체 생성
const child = new Child('Child', 10);

// 자식 객체에서 부모 객체의 메소드 호출
child.sayName(); // "My name is Child."

// 부모 객체의 프로퍼티 변경
Parent.prototype.name = 'New Parent';

// 자식 객체의 프로퍼티 값도 변경됨
console.log(child.name); // "New Parent"
```

▼ 다른 예시

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

Person.prototype.greet = function() {
  console.log(`Hello, my name is ${this.name} and I'm ${this.age} years old.`);
};
```

```
function Student(name, age, school) {
  Person.call(this, name, age);
  this.school = school;
}

Student.prototype = Object.create(Person.prototype);
Student.prototype.constructor = Student;

Student.prototype.study = function() {
  console.log(`I'm studying at ${this.school}.`);
};
```

```
const student1 = new Student('Alice', 20, 'Harvard');
student1.greet(); // Hello, my name is Alice and I'm 20 years old.
student1.study(); // I'm studying at Harvard.
```

장점으로는 생성자 함수를 사용하여 프로토타입 상속을 구현하면,
객체 리터럴 방식보다 더욱 강력하고 유연한 상속 방식을 구현할 수 있습니다.
또한, 생성자 함수 내부에 객체의 초기값을 설정하는 코드를 작성할 수 있으므로 객체를 생성할 때마다 값을 설정하는 번거로움을 줄일 수 있습니다.

생성자 함수를 사용하여 상속을 구현하면, 생성자 함수 내부에 정의된 속성과 메서드는 상속되지 않습니다.
이는 객체 리터럴 방식에서는 가능한 상속 방식이지만,
생성자 함수를 사용한 상속 방식에서는 상속을 위해 프로토타입 객체에 속성과 메서드를 추가해야 합니다.
이는 코드의 가독성을 떨어뜨릴 수 있습니다.

- `.call` 은 자바스크립트에서 함수를 호출할 때 사용되는 메서드 중 하나입니다. 이 메서드는 첫 번째 인자로 함수 내부에서 `this` 키워드가 가리킬 객체를 지정하고, 나머지 인자들을 해당 함수의 인자로 전달합니다.
- `.call` 을 이용한 상속은 객체의 메서드를 직접 상속 받는 것이 아니라 부모 객체를 인자로 전달하여 메소드를 호출합니다. 그렇기 때문에 직접 상속이 아니라, 인자로 전달하여 부모 객체의 기능을 사용할 수 있는 것입니다.
- 이 경우에 주석을 제거하지 않아도 'Hello, Alice'가 출력되는데, 해당 과정은 child에서 sayHello 메소드가 실행되는 것이 아닌, 프로토타입 체인을 따라 Parents 객체에서 정의된 sayHello를 실행하는 것입니다.

장점:

1. 프로토타입 체인을 이용한 상속 방법으로서, 상속을 쉽게 구현할 수 있습니다.
2. 상속 받은 메서드는 모든 객체에서 공유됩니다. 따라서, 메모리를 절약할 수 있습니다.
3. 기존에 생성된 객체를 기반으로 상속을 받을 수 있습니다.
4. 다중 상속이 가능합니다.

단점:

1. 코드의 가독성이 좋지 않습니다. 생성자 함수, 프로토타입 등의 구조가 복잡합니다.
2. 상속 구조가 복잡한 경우에는 유지보수가 어려울 수 있습니다.
3. 생성자 함수를 사용하여 객체를 만들 때, 인스턴스마다 프로토타입 체인을 복사합니다. 이로 인해, 메모리 사용량이 증가할 수 있습니다.
4. 상속 체인이 깊어질수록 성능이 저하될 가능성이 있습니다.

▼ 장점 2번과 3번이 충돌하지 않을까?

장점 2번에서는 상속 받은 메서드는 모든 객체에서 공유됩니다. 따라서, 메모리를 절약할 수 있습니다.
이는 단점 3번에서 인스턴스마다 프로토타입 체인을 복사한다는 내용과 조금 충돌합니다.

실제로는, 객체를 생성할 때 프로토타입 체인이 복사되므로 메모리를 사용하는 것은 사실입니다.
그러나, 이 복사된 체인에는 메서드와 같은 데이터가 포함되어 있지 않으며,
메서드는 프로토타입 체인의 상위에 위치하는 생성자 함수의 프로토타입 객체에 저장됩니다.
따라서, 메서드는 모든 인스턴스에서 공유되며 메모리를 절약할 수 있습니다.

그러나, 단점 3번에서 설명한 것처럼 객체의 인스턴스를 생성할 때마다 프로토타입 체인이 복사되므로,
생성된 인스턴스의 개수가 많아질수록 메모리 사용량도 증가할 수 있습니다.

객체 리터럴 방식 사용

```
// 부모 객체
const parent = {
  name: 'Parent',
  sayName() {
    console.log(`My name is ${this.name}.`);
  }
};

// 자식 객체
const child = Object.create(parent);

// 자식 객체에서 부모 객체의 메소드 호출
child.sayName(); // "My name is Parent."

// 부모 객체의 프로퍼티 변경
parent.name = 'New Parent';

// 자식 객체의 프로퍼티 값은 변경되지 않음
console.log(parent.name); // "New Parent"
console.log(child.name); // "Parent"
```

장점:

1. 구현이 간단하고 쉽습니다.
객체 리터럴 방식은 객체를 직접 생성하여 프로토타입 체인을 구성하기 때문에 구현이 간단하고 쉽습니다.

2. 코드의 가독성이 높습니다.
객체 리터럴 방식은 객체의 속성과 메서드를 직관적으로 표현할 수 있기 때문에 코드의 가독성이 높습니다.
3. 인스턴스의 생성이 빠릅니다.
객체 리터럴 방식은 인스턴스를 생성할 때마다 생성자 함수를 호출하는 과정이 없기 때문에 인스턴스의 생성이 빠릅니다.

▼ 장점 3번에 대한 설명

객체 리터럴 방식에서는 객체를 생성할 때, 생성자 함수를 호출하는 과정이 없기 때문에 인스턴스의 생성이 빠릅니다. 생성자 함수 방식은 객체를 생성할 때, 매번 생성자 함수를 호출하여 인스턴스를 생성하는데, 이는 객체 리터럴 방식보다 더 많은 작업을 필요로 합니다.

또한, 객체 리터럴 방식에서는 객체를 생성할 때, 생성자 함수를 호출하는 것이 아니므로, 생성자 함수에서 필요한 인자값을 전달할 필요가 없습니다. 생성자 함수 방식에서는 객체를 생성할 때, 생성자 함수에 필요한 인자값을 전달해야 하기 때문에, 이러한 작업도 객체 리터럴 방식보다 더 많은 작업을 필요로 합니다.

생성자 함수는 인스턴스를 생성하고, 생성된 인스턴스에 대한 초기화 작업을 수행합니다. 이를 위해 생성자 함수 내부에서는 인스턴스를 가리키는 `this` 키워드를 사용하고, 초기화 작업을 위한 프로퍼티나 메서드를 정의합니다. 이와 같은 동작들은 객체 리터럴 방식에서는 수행하지 않아도 되므로, 내부 동작이 더 복잡해집니다.

또한, 생성자 함수 방식에서는 인스턴스 생성을 위해 `new` 연산자를 사용해야 하며, 생성자 함수의 이름과 인자값 등을 정확히 입력해야 합니다. 이러한 작업은 객체 리터럴 방식보다 더 많은 작업을 필요로 합니다.

단점:

1. 상속 구현이 복잡합니다.
객체 리터럴 방식은 상속을 구현하기 위해서는 객체를 복제하거나, `Object.create()` 를 사용하는 등의 방법을 사용해야 합니다. 이는 코드의 복잡도를 증가시킬 수 있습니다.
2. 생성자 함수와 프로토타입 객체를 분리할 수 없습니다.
객체 리터럴 방식은 생성자 함수와 프로토타입 객체를 분리하여 구현할 수 없습니다. 이는 코드의 재사용성을 저하시킬 수 있습니다.
3. 객체 생성 후에는 속성과 메서드를 추가하기 어렵습니다.
객체 리터럴 방식으로 생성된 객체는 생성된 이후에는 속성과 메서드를 추가하기 어렵습니다. 이는 코드의 유지보수성을 저하시킬 수 있습니다.

▼ 단점 3번에 대한 설명

생성자 함수

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

Person.prototype.sayHello = function() {
  console.log("Hello, my name is " + this.name + " and I'm " + this.age + " years old.");
}

var john = new Person("John", 25);
john.sayHello(); // "Hello, my name is John and I'm 25 years old." 출력
```

객체 리터럴 방식

```
var person = {
  name: "John",
  age: 25
}

person.sayHello = function() {
  console.log("Hello, my name is " + this.name + " and I'm " + this.age + " years old.");
}

person.sayHello(); // "Hello, my name is John and I'm 25 years old." 출력
```

객체 리터럴 방식에서는 이름을 변경할 때 일일이 해당 객체를 가리키는 변수명을 모두 찾아서 일일이 변경해야 합니다.

예를 들어 `person`의 이름을 변경하기 위해서는

```
person.name = "siun"
```

이런 식으로 객체명과 속성의 이름을 모두 알아야 되기 때문에 어려움이 발생합니다.

이는 더불어 유지보수성 또한 저하시킬 수 있습니다.

두 방식의 차이

▼ 두 방식의 차이

생성자 함수 방식의 상속은 자식 생성자 함수에서 부모 생성자 함수를 호출하여 부모 객체를 초기화한 후, 부모 생성자 함수의 프로토타입 객체를 자식 생성자 함수의 프로토타입으로 설정합니다. 이로 인해 자식 객체의 인스턴스는 부모 객체의 메소드와 프로퍼티를 상속받을 수 있습니다.

객체 리터럴 방식의 상속은 `Object.create()` 메소드를 이용하여 부모 객체를 상속받는 새로운 객체를 생성하고, 이를 자식 객체의 프로토타입으로 설정합니다. 이로 인해 자식 객체는 부모 객체의 메소드와 프로퍼티를 상속받을 수 있습니다.

생성자 함수 방식의 상속은 부모 생성자 함수의 프로토타입 객체를 자식 생성자 함수의 프로토타입으로 설정하는 과정에서 부모 생성자 함수의 `prototype` 객체에 직접 접근해야 하므로, 부모 생성자 함수의 `prototype` 객체를 수정할 수 있습니다. 이를 이용하여 부모 생성자 함수의 `prototype` 객체에 새로운 메소드를 추가하면, 이 메소드는 자식 생성자 함수의 프로토타입에도 적용됩니다.

반면 객체 리터럴 방식의 상속은 `Object.create()` 메소드를 이용하여 새로운 객체를 생성하므로, 부모 객체의 프로퍼티와 메소드를 변경하더라도 자식 객체에 영향을 미치지 않습니다. 다만 자식 객체에서 부모 객체의 프로퍼티와 메소드를 재정의할 경우, 재정의한 내용이 자식 객체에 반영됩니다.

따라서 생성자 함수 방식의 상속은 부모 객체의 `prototype` 객체를 직접 수정할 수 있어서 부모 객체의 메소드와 프로퍼티를 동적으로 변경할 수 있는 장점이 있습니다. 반면 객체 리터럴 방식의 상속은 자식 객체에서 부모 객체의 메소드와 프로퍼티를 변경하는 것이 불가능하고, 상속 관계가 정적으로 결정된다는 한계가 있습니다.

▼ 차이가 발생하는 이유

생성자 함수를 통한 상속과 객체 리터럴 방식을 사용한 상속에서 이러한 차이가 있는 이유는 내부적으로 동작하는 원리 때문입니다.

생성자 함수를 사용한 상속에서 자식 객체는 부모 객체의 인스턴스로 생성됩니다. 따라서 자식 객체가 부모 객체를 상속받으면 부모 객체의 프로퍼티와 메소드를 자식 객체가 직접 참조하고 있습니다. 그렇기 때문에 부모 객체의 프로퍼티나 메소드가 변경되면 자식 객체에서도 변경된 값을 참조하게 됩니다.

반면, 객체 리터럴 방식을 사용한 상속에서는 `Object.create()` 메소드를 이용하여 새로운 객체를 생성합니다. 이때 새로 생성된 객체는 부모 객체의 프로토타입을 상속받아 프로토타입 체인을 구성합니다. 자식 객체가 부모 객체의 프로퍼티나 메소드를 사용할 때, 자식 객체의 프로퍼티나 메소드가 우선적으로 참조됩니다. 그렇기 때문에 부모 객체의 프로퍼티나 메소드가 변경되더라도 자식 객체에는 영향을 미치지 않습니다.

즉, 생성자 함수를 사용한 상속에서는 자식 객체가 부모 객체의 인스턴스로 생성되어 부모 객체의 프로퍼티와 메소드를 직접 참조하기 때문에 부모 객체의 변경 사항이 자식 객체에 반영됩니다. 반면 객체 리터럴 방식을 사용한 상속에서는 자식 객체가 부모 객체의 프로토타입을 상속받아 프로토타입 체인을 구성하기 때문에 자식 객체에서 우선적으로 자신의 프로퍼티나 메소드를 참조하게 되고, 부모 객체의 변경 사항이 자식 객체에 영향을 주지 않습니다.

프로토타입 체이닝

| __proto__를 따라 탐색하는 과정

프로토타입 체이닝은 객체가 어떤 메서드나 속성에 접근할 때 해당 객체의 프로토타입 체인을 따라가면서 해당 속성이나 메서드를 찾는 방식입니다. 즉, 현재 객체에 해당 속성이나 메서드가 없으면, 그 객체의 프로토타입에서 해당 속성이나 메서드를 찾고, 그 프로토타입 객체의 프로토타입에서 또 해당 속성이나 메서드를 찾는 식으로 계속해서 상위 객체로 올라가면서 검색합니다.

이렇게 객체의 프로토타입 체인을 따라가면서 속성이나 메서드를 찾는 것은 상속을 구현하기에 매우 효과적입니다. 또한, 객체의 프로토타입 체인을 동적으로 변경할 수 있기 때문에, 객체의 동작을 동적으로 변경할 수 있는 매우 강력한 기능을 제공합니다.

하지만, 프로토타입 체이닝은 검색 속도가 느리고, 잘못 사용하면 예상치 못한 결과를 초래할 수도 있기 때문에, 사용에 주의가 필요합니다.

프로토타입 오버라이딩

프로토타입 오버라이딩(Prototype overriding)은 자바스크립트에서 객체지향 프로그래밍을 구현할 때, 객체의 프로토타입에 이미 존재하는 메서드나 속성을 하위 객체에서 다시 정의하는 것을 말합니다.

프로토타입 체이닝에서는 객체가 어떤 메서드나 속성에 접근할 때 해당 객체의 프로토타입 체인을 따라가면서 해당 속성이나 메서드를 찾게 됩니다. 만약 하위 객체에서 프로토타입에 이미 존재하는 메서드나 속성을 다시 정의하면, 해당 객체에서는 오버라이딩된 메서드나 속성이 사용됩니다.

프로토타입 오버라이딩은 객체지향 프로그래밍에서 다형성을 구현하기 위해 사용됩니다. 즉, 같은 이름의 메서드가 하위 객체에서 다르게 동작하도록 정의할 수 있습니다. 하지만, 오버라이딩을 남발하면 코드를 이해하기 어려워지고, 예기치 못한 오류가 발생할 수 있기 때문에 적절하게 사용해야 합니다.

```
function Animal() {}

Animal.prototype.sayHello = function() {
  console.log("Hello, I'm an animal");
}

function Cat() {}

Cat.prototype = Object.create(Animal.prototype);

let cat = new Cat();
cat.sayHello(); // "Hello, I'm an animal"

Cat.prototype.sayHello = function() {
  console.log("Meow, I'm a cat");
}

cat.sayHello(); // "Meow, I'm a cat"
```

▼ 오버라이딩 가이드라인

- 기능 확장: 상위 객체의 기능을 유지하면서 하위 객체에서 기능을 확장하고자 할 때, 상위 객체의 메서드를 오버라이딩할 수 있습니다. 이 경우 기능이 겹치거나 중복되지 않도록 주의해야 합니다.
- 예외 처리: 하위 객체에서 상위 객체의 메서드와 다르게 동작하는 예외적인 경우에는 오버라이딩이 필요할 수 있습니다. 이 경우 예외적인 동작이 필요한 부분만 오버라이딩하고, 나머지 부분은 상위 객체의 메서드를 그대로 사용하는 것이 좋습니다.
- 인터페이스 일관성: 하위 객체에서 상위 객체와 같은 이름의 메서드를 사용하지만 다른 동작을 수행하는 경우, 이는 인터페이스 일관성을 해치기 때문에 좋지 않은 방법입니다. 이 경우 메서드의 이름을 변경하거나, 다른 메서드를 사용하는 것이 좋습니다.
- 코드 가독성: 오버라이딩을 남발하면 코드가 복잡해져 가독성이 떨어질 수 있습니다. 따라서 필요한 경우에만 오버라이딩을 사용하고, 코드를 단순하게 유지하는 것이 좋습니다.

프로토타입 새도잉

프로토타입 새도잉(Prototype Shadowing)은 부모 객체의 프로토타입에서 상속받은 메서드를 자식 객체에서 덮어쓰는 것이 아니라, 인스턴스 객체에 직접 메서드를 추가하여 부모 객체의 메서드를 가리는 것입니다.

자바스크립트에서 객체는 메서드나 프로퍼티를 찾을 때 먼저 자신의 프로토타입 체인을 검색합니다. 만약 해당 메서드나 프로퍼티가 자신의 프로토타입 체인 상에 없다면, 자바스크립트는 부모 객체의 프로토타입 체인을 검색합니다. 이렇게 부모 객체의 프로토타입 체인을 검색하다가 해당 메서드나 프로퍼티를 발견하면, 해당 메서드나 프로퍼티가 호출됩니다.

프로토타입 새도잉은 이러한 메서드나 프로퍼티 검색 과정에서, 자식 객체에서 직접 추가한 메서드나 프로퍼티가 부모 객체의 메서드나 프로퍼티를 가리는 현상입니다. 이 때, 프로토타입 체인은 건너뛰고 인스턴스 객체에서 직접 추가한 메서드나 프로퍼티가 호출됩니다.

▼ 프로토타입 오버라이딩과 프로토타입 새도잉의 차이

```
function Animal() {}

Animal.prototype.sayHello = function() {
  console.log("Hello, I'm an animal");
}

function Cat() {}

Cat.prototype = Object.create(Animal.prototype);

let cat = new Cat();
cat.sayHello(); // "Hello, I'm an animal"

cat.sayHello = function() {
  console.log("Meow, I'm a cat");
}

cat.sayHello(); // "Meow, I'm a cat"
```

위 코드를 보면 이전 코드와 달리 `cat` 인스턴스에 직접 `sayHello` 메서드를 추가하고, 이 메서드에서는 "Meow, I'm a cat"이 출력됩니다. 그러나, 이후 `cat.sayHello()` 를 호출하면 `cat` 인스턴스에 직접 추가한 `sayHello` 메서드가 호출되기 때문에, "Meow, I'm a cat"이 출력됩니다.

이는 구조적으로 차이를 가지는데,

프로토타입 오버라이딩은 부모 객체의 프로토타입에서 상속받은 메서드를 자식 객체에서 덮어쓰는 것입니다. 자식 객체에서 덮어쓴 메서드는 프로토타입 체인을 따라 호출될 때 가장 먼저 호출되며, 부모 객체의 메서드는 무시됩니다.

반면에, 프로토타입 새도잉은 부모 객체의 프로토타입에서 상속받은 메서드를 자식 객체에서 덮어쓰는 것이 아니라, 인스턴스 객체에 직접 메서드를 추가하여 부모 객체의 메서드를 가리는 것입니다. 이 때, 프로토타입 체인은 건너뛰고 인스턴스 객체에서 직접 추가한 메서드가 호출됩니다.

따라서, 프로토타입 오버라이딩과 프로토타입 새도잉은 구조적으로 완전히 다릅니다. 프로토타입 오버라이딩은 프로토타입 체인에 따라 동작하며, 부모 객체의 메서드를 덮어쓰는 것입니다. 프로토타입 새도잉은 인스턴스 객체에 직접 메서드를 추가하여 부모 객체의 메서드를 가리는 것입니다.

즉, 프로토타입 새도잉의 경우, 인스턴스 객체에서 메서드를 먼저 검색하고 쉼도잉한 메소드를 발견할 수 있기 때문에 프로토타입 체이닝이 발생하지 않습니다. 더불어, 오버라이딩은 부모 객체의 재정의의 할 때 프로토타입 체이닝을 이용하지만, 쉼도잉의 경우 재정의의 할 때 체이닝을 생략합니다.

추가적으로,

오버라이딩과 쉼도잉은 둘 다 부모 클래스의 메서드를 자식 클래스에서 재정의의 하는 것을 의미하지만, 동작 방식에서 차이가 있습니다.

오버라이딩은 자식 클래스에서 부모 클래스의 메서드를 덮어쓰는 것을 말합니다. 이때, 덮어쓴 메서드는 부모 클래스의 메서드를 완전히 대체하게 됩니다. 따라서 자식 클래스에서 오버라이딩한 메서드가 호출되면, 부모 클래스의 메서드는 더 이상 호출되지 않습니다.

반면에, 쉼도잉은 부모 클래스와 자식 클래스에서 같은 이름을 가진 메서드가 존재할 때, 자식 클래스에서 정의된 메서드가 호출되지만, 부모 클래스의 메서드는 여전히 존재하며 호출될 수 있습니다. 따라서 쉼도잉은 부모 클래스의 메서드와 자식 클래스의 메서드가 서로 다른 기능을 수행할 수 있으며, 부모 클래스의 메서드를 호출하려면 명시적으로 부모 클래스의 인스턴스를 생성하여 호출해야 합니다.

한줄요약: 부모 객체를 참조하나, 참조하지 않느냐

오버라이딩은 부모 객체를 참조하여 부모 객체의 메서드를 덮어쓰기 때문에 부모 객체와의 관계를 유지합니다. 이에 비해 체이닝은 부모 객체를 참조하지 않고, 부모 객체의 프로토타입 체인을 따라서 메서드를 검색합니다. 따라서 부모 객체와의 관계를 유지하지 않고, 자식 객체에서 메서드를 찾은 후에는 체인을 따라 검색을 멈춥니다. 이는 더 효율적인 검색과 메모리 사용을 가능하게 하지만, 상속의 개념을 미약하게 만들 수 있습니다.

자바스크립트의 Class

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  getName() {
    return this.name;
  }

  getAge() {
    return this.age;
  }
}

class Student extends Person {
  constructor(name, age, school) {
    super(name, age);
    this.school = school;
  }

  getSchool() {
    return this.school;
  }
}

const student1 = new Student('John', 20, 'ABC School');
console.log(student1.getName()); // "John"
console.log(student1.getAge()); // 20
console.log(student1.getSchool()); // "ABC School"
```

Class 문법은 ES6(ECMAScript 2015)에서 추가된 것으로, 기존의 프로토타입 기반 상속을 보다 객체지향적인 방식으로 구현할 수 있도록 도입되었습니다.

내부적으로는 프로토타입 기반의 상속을 사용합니다.

위의 코드를 프로토타입으로 구현하면 다음과 같습니다.

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

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

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

function Student(name, age, school) {
  Person.call(this, name, age);
  this.school = school;
}

Student.prototype = Object.create(Person.prototype);
Student.prototype.constructor = Student;

Student.prototype.getSchool = function() {
  return this.school;
}

const student1 = new Student('John', 20, 'ABC School');
console.log(student1.getName()); // "John"
console.log(student1.getAge()); // 20
console.log(student1.getSchool()); // "ABC School"
```


Class inheritance 와 Prototypal inheritance 는 같은가?

- Class 상속에서 인스턴스는 설계도(Class)로 하위 클래스 관계를 생성한다. 다시말해, 클래스를 인스턴스를 사용하는 것 처럼 사용할 수 없다. Class의 정의에 인스턴스의 함수를 불러올 수 없다. 인스턴스를 먼저 생성한 후에 해당 함수를 불러올 수 있다.
- Prototype 상속에서 인스턴스는 다른 인스턴스로부터 상속된다. 위임형 상속을 사용하는 것은 말 그대로 다른 오브젝트와 연결된 오브젝트라고 할 수 있다. 프로토타입 객체는 다른 객체의 기반이 된다. 연결상속을 사용하면 새인스턴스로 속성을 복사함으로써 상속을 구현하는 방법이다.

▼ 참고자료


자바스크립트는 상속을 어떻게 구현할까?

자바스크립트는 상속을 어떻게 구현할까? 결론부터 말하자면 자바스크립트는 프로토타입을 기반으로 상속을 구현한다. 나는 자바스크립트에 클래스가 도입된 ES2015 이후 자바스크립트를 처음 접했고 클래스를 사용해서 코드를 작성했기 때문에 프로토타입을 직접 사용해 본 적이 없

 <https://velog.io/@ssulv3030/자바스크립트는-상속을-어떻게-구현할까>


Syntactic sugar


In computer science, syntactic sugar is syntax within a programming language that is designed to make things easier to read or to express. It makes the language "sweeter" for human use: things can be expressed more clearly, more concisely, or in an alternative style that some may prefer. Syntactic sugar is usually a shorthand for a common operation that could also be expressed in an alternate, more verbose, form: The programmer has a choice of whether to use the shorter form or the longer form, but will usually use the shorter form since it is shorter and easier to type and read.

 https://en.wikipedia.org/wiki/Syntactic_sugar

PoimaWeb


웹 프로그래밍 튜토리얼


 <https://poimaweb.com/js-object-oriented-programming>



JavaScript : 프로토타입(prototype) 이해


JavaScript는 클래스라는 개념이 없습니다. 그래서 기존의 객체를 복사하여(cloning) 새로운 객체를 생성하는 프로토타입 기반의 언어입니다. 프로토타입 기반 언어는 객체 원형인 프로토타입을 이용하여 새로운 객체를 만들어냅니다. 이렇게 생성된 객체 역시 또 다른 객체의 원형이 될 수 있습니다. 프로토타입은 객체를 확장하고 객체 지향적인 프로그래밍을 할 수 있게 해줍니다. 프로토타입은 크게 두 가지로

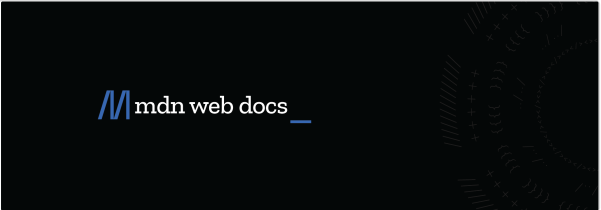
 <https://www.nextree.co.kr/p7323/>



상속과 프로토타입 - JavaScript | MDN


Java 나 C++ 같이 클래스 기반의 언어를 사용하던 프로그래머는 자바스크립트가 동적인 언어라는 점과 클래스가 없다는 것에서 혼란스러워 한 다. (ES2015부터 class 키워드를 지원하기 시작했으나, 문법적인 양념일 뿐이며 자바스크립트는 여전히 프로토타입 기반의 언어다.)

 https://developer.mozilla.org/ko/docs/Web/JavaScript/Inheritance_and_the_prototype_chain



Property Shadowing and __proto__ In Javascript Objects.

Before talking about Objects Shadowing and explaining how it works, we first have to define what shadowing in general is.

 <https://levelup.gitconnected.com/property-shadowing-and-proto-in-javascript-objects-d007fa062a63>

```
▶ hasOwnProperty: f hasOwnProperty()
▶ isPrototypeOf: f isPrototypeOf()
▶ propertyIsEnumerable: f propertyIsEnumerable()
▶ toLocaleString: f toLocaleString()
▶ toString: f toString()
▶ valueOf: f valueOf()
▶ __defineGetter__: f __defineGetter__()
▶ __defineSetter__: f __defineSetter__()
```

[10분 테코톡] 아놀드의 프로토타입 뽐내기

 우아한테크코스의 크루들이 진행하는 10분 테크토크입니다. 

'10분 테코톡'이란 우아한테크코스 과정을 진행하며 크루(수강생)들이 동료들과 학습한 내용을 공유하고 이야기하는 시간입니다. 서로가 성장

 <https://www.youtube.com/watch?v=TqFwNFTa3c4>

프로토타입 뽐내기

아놀드



인터프리터 언어

인터프리터 언어란?

코드를 실행하기 위해 컴파일 과정 없이 바로 실행하는 방식을 채택합니다.

즉, 프로그램을 작성하고 나서 바로 실행할 수 있는 언어를 말합니다.

인터프리터 언어는 일반적으로 코드의 한 줄씩 읽어서 실행하는 방식으로 작동합니다.

이때 코드의 한 줄이 실행되는 과정에서 오류가 발생하면, 해당 줄에서 실행이 중단되고 오류 메시지가 출력됩니다.

따라서 프로그래머는 코드를 작성하면서 오류를 발견하고 수정할 수 있습니다.

인터프리터 언어의 장점은 코드 수정 및 디버깅이 쉽다는 것입니다.

또한 코드 실행 전 컴파일 과정이 없으므로 개발 시간을 단축할 수 있습니다.

그러나 반면에 실행 속도가 상대적으로 느리다는 단점이 있습니다.

자바스크립트의 작동방식

실행 컨텍스트

실행 컨텍스트는 코드가 실행될 때 생성되는 환경 정보를 담고 있는 객체입니다.

변수, 함수 선언, 매개변수 등의 정보를 담고 있으며, 코드의 실행에 필요한 컨텍스트 정보를 제공합니다.

각 실행 컨텍스트는 콜 스택에 쌓여 실행되며, 가장 위에 있는 실행 컨텍스트가 현재 실행 중인 컨텍스트입니다.

실행 컨텍스트는 함수가 실행될 때 만들어집니다.

자바스크립트 엔진이 스크립트를 처음 만날 때 전역 컨텍스트를 생성하고, 콜 스택에 **push** 합니다.

그리고 계속 읽어나가면서 함수 호출을 발견할 때마다 실행 컨텍스트를 스택에 **push** 합니다.

실행 가능한 코드

전역 코드

전역 영역에 존재하는 코드

함수 코드

함수 내에 존재하는 코드

Eval 코드

Eval 함수로 실행되는 코드

▼ Eval 함수란?

문자열을 자바스크립트 코드로 해석하고 실행하는 기능을 합니다.

이 함수는 문자열을 실행 컨텍스트의 스코프 체인에서 실행하며 새로운 변수를 만들거나 기존의 변수를 변경할 수 있습니다.

그러나 Eval 함수는 보안상의 이유로 권장되지 않습니다. 외부에서 실행될 가능성이 있는 문자열을 인자로 받는 경우, 보안상의 위험이 될 수 있기 때문입니다. 따라서 eval() 함수를 대신할 수 있는 대안으로는 Function() 생성자 함수나 JSON.parse() 함수를 사용하는 것이 권장됩니다.

실행 컨텍스트 예시

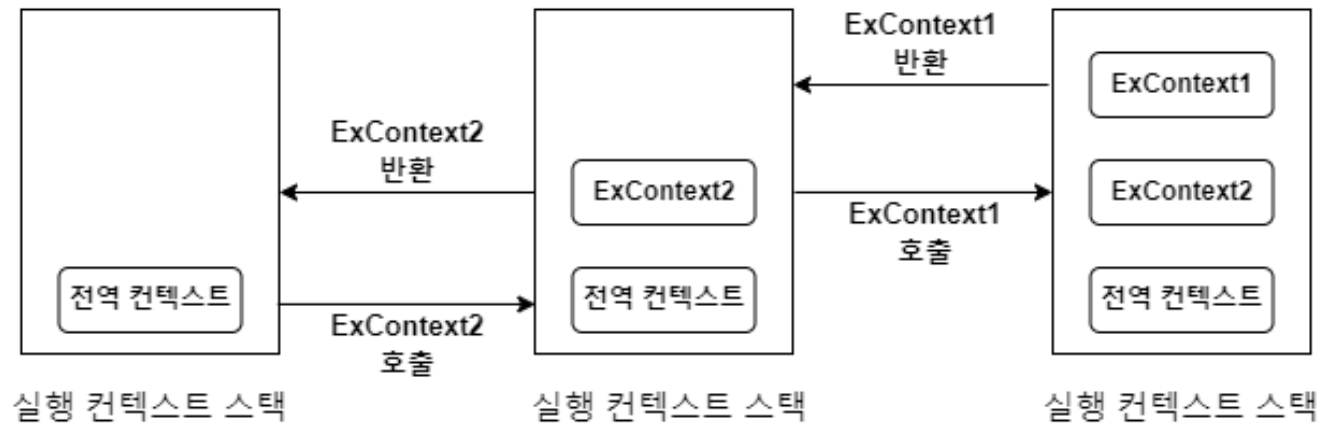
```
console.log("글로벌 컨텍스트 입니다");

function ExContext1() {
  console.log("1번 실행 컨텍스트 입니다");
};

function ExContext2() {
  ExContext1();
  console.log("2번 실행 컨텍스트 입니다");
};

ExContext2();

// 글로벌 컨텍스트 입니다.
// 1번 실행 컨텍스트 입니다
// 2번 실행 컨텍스트 입니다
```



출처 : <https://velog.io/@jminkyoun/TIL-34-실행-컨텍스트Execution-Context-란>

ES5 이전

실행 컨텍스트의 구성요소

1. Variable Object(변수 객체)

- 변수 객체는 현재 실행 컨텍스트 내에서 사용되는 지역 변수, 매개 변수, 함수 선언 등의 정보를 담고 있는 객체입니다.
- 변수 객체는 스코프 체인을 통해 접근할 수 있습니다.

2. Scope Chain(스코프 체인)

- 스코프 체인은 현재 실행 컨텍스트의 변수 객체와 상위 실행 컨텍스트의 변수 객체를 연결한 체인 구조입니다.
- 이를 통해 현재 실행 컨텍스트에서 변수나 함수를 찾지 못하면 상위 실행 컨텍스트의 변수 객체를 순차적으로 검색합니다.
- 스코프 체인은 실행 컨텍스트 객체의 내부 슬롯 중 하나인 `LexicalEnvironment` 프로퍼티에 저장된 환경 레코드(Environment Record)와 외부 렉시컬 환경(Outer Lexical Environment) 참조로 이루어집니다.
- 이 환경 레코드는 현재 실행 중인 함수의 변수와 매개변수 등을 저장하고 있으며, 외부 렉시컬 환경 참조는 현재 실행 중인 함수의 상위 스코프의 `LexicalEnvironment` 프로퍼티를 참조합니다.

```
let a = 10;

function outer() {
  let b = 20;

  function inner() {
    let c = 30;
    console.log(a + b + c);
  }

  inner();
}

outer();
```

이 코드에서 `outer` 함수는 전역 스코프에서 선언된 `a` 변수와 `b` 변수, 그리고 `inner` 함수를 포함합니다. `inner` 함수는 `outer` 함수 스코프에 접근할 수 있으므로, `inner` 함수에서 `a` 와 `b` 변수에 접근할 수 있습니다.

실행 컨텍스트가 `inner` 함수를 생성할 때, `inner` 함수의 스코프체인은 `inner` 함수의 렉시컬 환경과 `outer` 함수의 렉시컬 환경, 그리고 전역 렉시컬 환경으로 구성됩니다. 이렇게 구성된 스코프체인을 통해 `inner` 함수에서 `a` 와 `b` 변수에 접근할 수 있게 됩니다.

3. This Value(this)

- `this`는 현재 실행 컨텍스트 내에서의 `this` 값을 저장하는 프로퍼티입니다.
- `this` 값은 함수 호출 패턴에 따라 결정됩니다.

동작 방식

1. 전역 실행 컨텍스트 생성

- 전역 실행 컨텍스트는 자바스크립트 코드가 실행되기 전에 자동으로 생성됩니다.

2. 함수 실행 컨텍스트 생성

- 함수가 호출될 때마다 해당 함수의 실행 컨텍스트가 생성됩니다.

3. Variable Object 생성

- 변수 객체(Variable Object)가 생성되며, 해당 함수 내의 변수와 함수 선언문이 모두 변수 객체 내에 포함됩니다.

4. Scope Chain 생성

- 스코프 체인(Scope Chain)이 생성됩니다.
- 스코프 체인은 현재 실행 컨텍스트의 변수 객체와 상위 컨텍스트의 변수 객체들을 참조하는 연결 리스트입니다.

5. this 바인딩

- `this`가 바인딩됩니다.
- 함수 호출 시점에 결정되며, 함수 호출 방식에 따라 참조하는 객체가 다를 수 있습니다.

6. 코드 실행

- 변수와 함수 선언문이 변수 객체에 등록되고, 코드가 실행됩니다.
- 변수를 참조할 때는 스코프 체인을 검색하여 값을 찾습니다.
- 함수를 호출할 때는 새로운 실행 컨텍스트가 생성되고, 위의 단계들을 다시 수행합니다.

7. 실행 컨텍스트 제거

- 함수의 실행이 끝나면 해당 함수의 실행 컨텍스트가 제거됩니다.
- 제거되는 순서는 호출된 순서의 역순으로 처리됩니다.
- 전역 실행 컨텍스트는 프로그램이 종료될 때 제거됩니다.

ES6 이후

실행 컨텍스트의 구성요소

1. Lexical Environment (렉시컬 환경)

- 현재 실행 컨텍스트에서 사용할 수 있는 식별자들을 모아놓은 객체입니다.
- 실행 컨텍스트가 생성될 때 현재 스코프에 있는 변수, 함수, 매개변수 등의 정보를 담아 생성합니다.
- 쉽게 말해, 변수나 함수가 포함되어 있는 위치.

2. Environment Record (환경 레코드)

- 현재 실행 컨텍스트에서 사용할 수 있는 식별자들을 기록한 객체입니다.
- 실행 컨텍스트가 생성될 때 Lexical Environment 객체 안에 새로운 Environment Record 객체를 생성하여 할당합니다.
- 환경 레코드는 두 가지 유형이 존재하는데, 선언적 환경 레코드(Declarative Environment Record)와 객체 환경 레코드(Object Environment Record)가 있습니다.
- 선언적 환경 레코드는 주로 변수 선언과 함수 선언에 대한 정보를 담고 있으며, 식별자와 함께 값(undefined, 함수, 혹은 다른 값)을 바인딩합니다.
- 객체 환경 레코드는 객체에 대한 정보를 담고 있으며, 객체 프로퍼티에 접근할 때 사용됩니다. 객체 환경 레코드는 전역 객체와 같은 특정 객체에 대한 참조를 담고 있으며, 객체 프로퍼티의 이름과 값을 바인딩합니다.
- 렉시컬 환경의 정보를 담고 있는 구조

3. This Binding (this 바인딩 정보)

- 현재 실행 컨텍스트에서의 this 키워드의 값에 대한 정보를 담고 있습니다.
- 함수 호출 시 함수를 호출한 객체(메소드 호출) 또는 전역 객체(일반 함수 호출)를 가리킵니다.

4. Closure (클로저 정보)

- 함수가 생성될 때 해당 함수와 관련된 정보를 담은 객체입니다.
- 함수가 생성될 때 현재의 Lexical Environment 객체를 기억하여, 함수가 실행될 때 Lexical Environment를 유지합니다.
- 즉, 수가 정의될 당시의 렉시컬 환경(Lexical Environment)을 기억하고, 이를 실행할 때 다시 불러와 사용할 수 있게 하는 기능입니다.
- 함수 내부에 또 다른 함수가 정의되어 있을 때, 내부 함수에서 외부 함수의 변수를 참조하면, 이 내부 함수는 외부 함수의 변수에 대한 클로저를 가지게 됩니다. 이렇게 클로저를 통해 외부 함수의 변수를 계속해서 참조할 수 있습니다.

5. Generator/Async Function State (제너레이터 및 비동기 함수 상태)

- 제너레이터와 비동기 함수에서 yield나 await 키워드를 만났을 때 현재 상태를 저장하는 객체입니다.
- 함수가 다시 호출될 때 저장한 상태를 기억하여 이전 상태에서 실행을 이어나갈 수 있습니다.

동작 방식

1. Realm 생성

- 실행 컨텍스트를 생성할 때 사용하는 각각의 전역 객체(또는 global object)를 가리키는 객체를 생성합니다.

2. Lexical Environment 생성

- 실행 컨텍스트의 Lexical Environment 컴포넌트를 생성합니다.
- 이 때, 현재 실행 중인 코드의 렉시컬 환경과 관련된 정보들(예: 함수의 매개변수, 변수, 내부 함수 등)이 포함된 Environment Record와 상위 스코프를 가리키는 스코프 체인이 생성됩니다.
- Lexical Environment는 Variable Environment와 스코프 체인으로 구성되어 있습니다.

3. Variable Environment 생성

- 실행 컨텍스트의 Variable Environment 컴포넌트를 생성합니다.
- 이 때, 현재 실행 중인 코드의 렉시컬 환경과 관련된 정보들(예: 함수의 매개변수, 변수 등)이 포함된 Environment Record와 상위 스코프를 가리키는 스코프 체인이 생성됩니다.

4. this 바인딩 결정

- this 바인딩을 결정합니다.
- Outer Environment 참조 결정
 - 현재 실행 중인 코드의 외부 환경에 대한 참조를 결정합니다.
 - Generator 객체 생성 (제너레이터 함수 실행 시에만 해당)
 - 제너레이터 함수가 실행될 때, Generator 객체를 생성합니다.
 - Generator 객체에는 현재 실행 중인 함수의 실행 상태와 관련된 정보들이 포함됩니다.
 - 실행 컨텍스트 스택에 push
 - 현재 실행 중인 실행 컨텍스트를 실행 컨텍스트 스택에 push합니다.
 - 이 때, 실행 컨텍스트 스택에는 현재 실행 중인 실행 컨텍스트와 상위 실행 컨텍스트들이 존재합니다.
 - 코드 실행
 - 실행 컨텍스트에 속한 코드를 실행합니다.
 - 실행 컨텍스트 스택에서 pop
 - 현재 실행 중인 실행 컨텍스트를 실행 컨텍스트 스택에서 pop합니다.
 - 이 때, 실행 컨텍스트 스택에는 이전에 push한 상위 실행 컨텍스트들이 존재합니다.

변경 후 차이점

ES6 실행 컨텍스트는 ES5 이전 버전보다 성능 개선 및 메모리 최적화에 대한 목표가 있습니다. ES6에서는 이전 버전과 달리 변수와 함수 선언의 처리 방식에 차이가 있기 때문에 실행 컨텍스트 구성 요소와 동작 방식도 변경되었습니다.

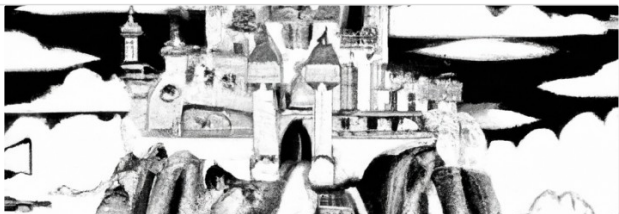
ES6에서는 Environment Record가 LexicalEnvironment의 일부로 통합되었고, 변수와 함수 선언의 처리에 대한 변경으로 인해 Scope Chain이 생성되는 방식이 다릅니다. 또한, this 바인딩 정보의 처리 방식에도 차이가 있습니다.

이러한 변경사항으로 ES6 실행 컨텍스트는 이전 버전보다 더 빠르고 효율적인 메모리 관리가 가능해졌습니다. 또한, let과 const 키워드의 등장으로 블록 스코프 변수 처리가 가능해져서 스코프 체인을 생성하는 방식도 달라졌습니다. 이로 인해 코드의 가독성과 유지보수성도 향상되었습니다.

What is a realm in JavaScript?

An easy to understand explanation of what realms are in JavaScript


<https://weizman.github.io/page-what-is-a-realm-in-js/>



실행 컨텍스트란 무엇인가요?

자바스크립트의 핵심 원리, 실행 컨텍스트에 대해 알아봅니다.

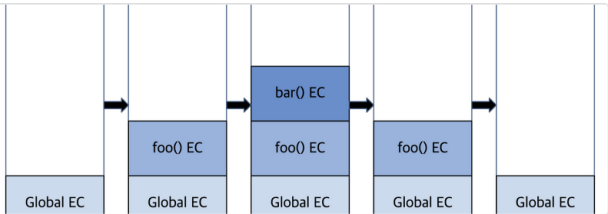
https://velog.io/@edie_ko/js-execution-context



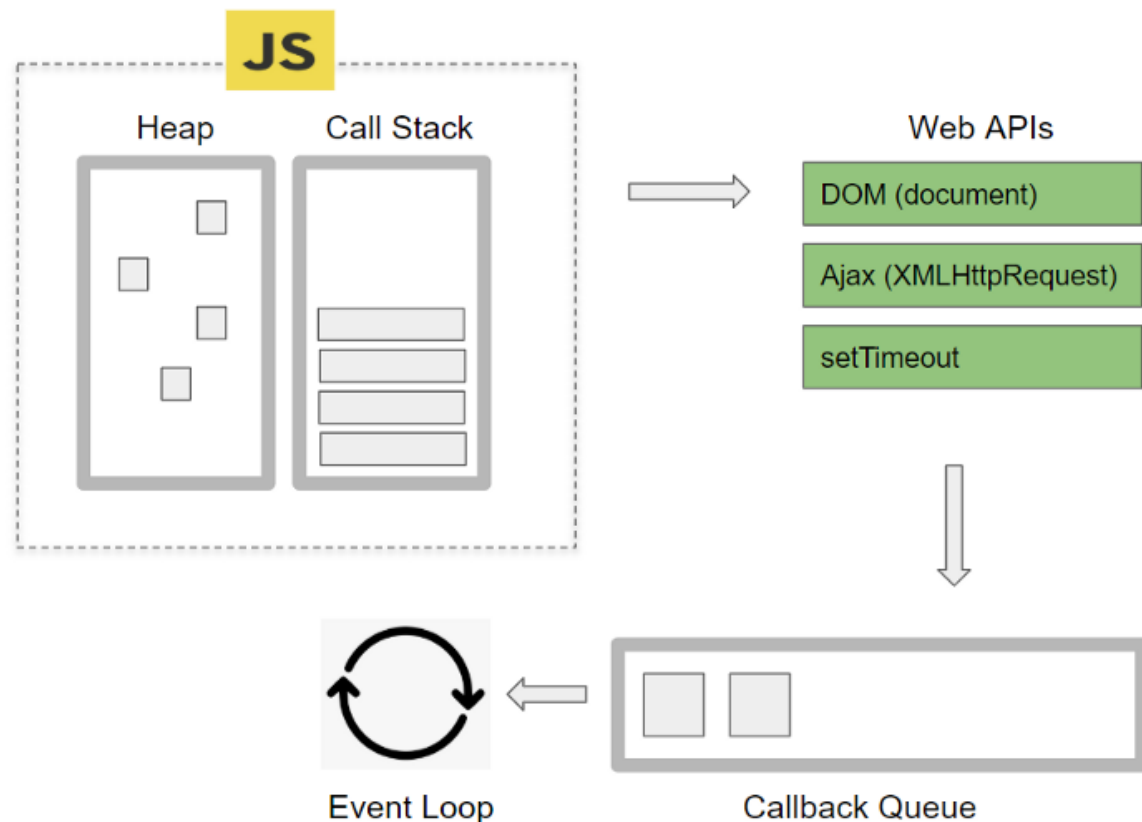
[Javascript] 실행 컨텍스트(Execution Context) 정리!!

실행 컨텍스트 정의 실행 가능한 코드를 형상화하고 구분하는 추상적인 개념 = 실행 가능한 코드가 실행되기 위해 필요한 환경 자바스크립트 엔진은 실행 가능한 코드를 실행하기 위해 필요한 정보를 형상화하고 구분하기 위해 실행 컨텍스트를 물리적 객체의 형태로 관리한다. 실행 가능한 코드 전역 코드 : 전역 영역에 존재하는 코드 함수 코드 : 함수 내에 존재하는 코드 Eval 코드 : eval 함수로 실행되는 코드 Eval 함수? 문자로 표현된

<https://blog.wanzargen.me/36>



동작 방식 & 용어 설명



Heap

메모리 할당이 발생하는 공간입니다.

Call Stack

실행된 코드의 환경을 저장합니다. 함수 호출 시 Call Stack에 저장되며, LIFO의 처리형태를 가집니다.

즉, 가장 위에 쌓인 함수를 먼저 처리합니다.

단 하나의 호출 스택을 이용하기 때문에 하나의 함수가 실행되는 동안 해당 함수의 작업이 끝날 때 까지 다른 작업을 할 수 없습니다.

자바스크립트는 indeed 싱글 스레드 언어입니다. 이는 자바스크립트 엔진이 하나의 메인 스레드(main thread)를 사용하여 코드를 실행하기 때문입니다. 또한, 자바스크립트의 콜 스택(call stack)도 하나이기 때문에, 코드는 한 번에 하나씩 실행됩니다.

Web APIs

Web API는 브라우저에서 제공하는 인터페이스 입니다. 브라우저가 제공하는 기능에 직접 접근합니다.

자바스크립트 코드는 브라우저가 제공하는 Web API를 통해 비동기적으로 실행될 수 있습니다.

예를 들어, setTimeout 함수를 호출하면, 해당 함수는 Web API에 의해 비동기적으로 실행되며,

지정된 시간이 경과하면 해당 함수가 콜백 큐(callback queue)에 추가됩니다.

이후, 이벤트 루프(event loop)가 실행되면서 콜백 큐의 함수들이 콜 스택으로 이동하여 실행됩니다.

이런 방식으로, 자바스크립트는 비동기적으로 실행되는 코드를 처리할 수 있습니다.

Web API의 존재 때문에, 자바스크립트는 싱글 스레드 언어이지만, 브라우저에서 실행되는 자바스크립트 코드는 WebAPI, 콜백 큐, 이벤트 루프 등의 메커니즘을 이용해서 비동기적으로 실행될 수 있습니다.

▼ Web API 카테고리

1. DOM APIs: Document Object Model APIs로, HTML, XML 등의 문서 구조를 다루는 데 사용됩니다.
DOM API를 이용하면, HTML 요소를 조작하거나, 이벤트를 처리하는 등의 다양한 작업을 수행할 수 있습니다.
2. Network APIs: HTTP 요청을 처리하는 데 사용되는 XMLHttpRequest(XHR) API나, 최근에는 fetch API 등을 이용하여 서버와 통신하는 작업을 할 수 있습니다.
3. Graphics APIs: 캔버스(Canvas) API나 WebGL API 등을 이용하여 그래픽을 생성하거나, 처리하는 작업을 할 수 있습니다.
4. Audio and Video APIs: 오디오(Audio) API나 비디오(Video) API 등을 이용하여 미디어를 처리하거나, 재생하는 작업을 할 수 있습니다.
5. Storage APIs: 로컬 스토리지(Local Storage) API나 IndexedDB API 등을 이용하여 브라우저 내부에 데이터를 저장하거나, 관리하는 작업을 할 수 있습니다.
6. Geolocation APIs: 브라우저가 제공하는 위치 정보를 이용하여, 현재 위치를 파악하는 작업을 할 수 있습니다.

Callback Queue

Callback Queue는 자바스크립트에서 비동기적으로 처리되어야 하는 콜백 함수들을 저장하는 자료구조입니다.

자바스크립트는 위에서 언급한 단일 콜 스택을 사용합니다. 이 Call Stack이 비어있을 때까지 코드를 순차적으로 실행합니다.

브라우저에서는 사용자 이벤트나 비동기 요청과 같이 실행에 시간이 걸리는 작업들이 많이 있습니다. 이러한 작업들이 즉시 실행되어야 한다면, 자바스크립트는 이를 처리하기 위해 대기열(queue)에 저장합니다. 그리고 콜 스택이 비어있을 때, 대기열에서 다음으로 실행될 콜백 함수를 꺼내와 콜 스택에 추가합니다.

콜백 큐에는 주로 비동기적으로 처리되어야 하는 작업들이 저장됩니다.

setTimeout, XMLHttpRequest, Promise와 같은 비동기 함수들은 실행 후 콜백 함수를 큐에 등록하며,

해당 함수가 실행 완료된 후 콜백 큐에서 순차적으로 실행됩니다.

Event Loop

이벤트 루프(Event Loop)는 비동기적으로 처리되는 작업들을 모니터링하고, 이를 적절하게 처리하는 메커니즘입니다. 이벤트 루프의 동작은 크게 콜 스택(Call Stack), 콜백 큐(Callback Queue), 그리고 Web API로 구성됩니다. 콜 스택은 현재 실행중인 함수들의 스택 구조를 갖고 있으며, 콜백 큐는 비동기적으로 처리되어야 할 작업들을 저장하는 큐입니다. Web API는 브라우저에서 제공하는 다양한 API들을 의미하며, 주로 비동기적으로 실행되는 작업들이 많이 사용됩니다. 이벤트 루프는 먼저 콜 스택을 모니터링하며, 콜 스택이 비어있을 때 콜백 큐에서 대기중인 작업들을 콜 스택으로 옮겨 실행합니다. 이를 통해, 자바스크립트 엔진은 비동기적으로 처리되어야 하는 작업들을 순차적으로 처리할 수 있습니다.

이벤트 루프의 동작은 브라우저의 Event Loop와 Node.js의 Event Loop가 약간 다르게 동작합니다. 브라우저의 Event Loop는 주로 UI 이벤트와 관련된 작업들을 처리하는데 중점을 둡니다. 반면 Node.js의 Event Loop는 파일 I/O나 네트워크 요청과 같은 I/O 작업들을 중심으로 동작합니다.

따라서, 이벤트 루프는 자바스크립트 엔진에서 비동기적으로 처리되는 작업들을 모니터링하고, 이를 적절히 처리하는 중요한 메커니즘입니다. 이를 이용해, 자바스크립트에서 비동기적인 프로그래밍을 가능하게 하며, 이를 통해 더욱 빠르고 유연한 애플리케이션을 개발할 수 있습니다.

비동기처리(callback, promise, async/await)

비동기 처리를 해야하는 이유

1. 사용자 인터페이스 반응성 향상: 사용자는 웹 페이지가 느리게 로드되거나 응답하지 않는 것을 싫어합니다. 비동기 처리를 통해 웹 페이지의 반응성을 향상시킬 수 있습니다. 예를 들어, 사용자가 데이터를 입력하거나 버튼을 클릭할 때, 비동기 처리를 사용하여 필요한 데이터를 백그라운드에서 로드하고 결과가 나올 때까지 사용자가 다른 작업을 수행할 수 있도록 할 수 있습니다.
2. 서버 요청 대기 시간 감소: 웹 애플리케이션에서 서버 요청을 동기적으로 처리하면, 모든 요청이 완료될 때까지 대기해야 합니다. 이는 대규모 데이터베이스 작업이나 느린 서버와 같은 상황에서 불필요한 지연을 초래할 수 있습니다. 비동기 처리를 사용하면 서버 응답을 기다리는 동안 다른 작업을 수행할 수 있습니다. 이를 통해 웹 애플리케이션의 성능을 향상시킬 수 있습니다.
3. 자원 효율성: 비동기 처리를 사용하면, 백그라운드에서 작업을 수행하므로 CPU와 메모리 등의 시스템 자원을 더욱 효율적으로 사용할 수 있습니다.
4. 복잡한 작업 처리: 웹 애플리케이션에서 복잡한 작업을 동기적으로 처리하면, 이 작업이 완료될 때까지 사용자가 다른 작업을 수행할 수 없습니다. 이러한 작업을 비동기 처리를 통해 백그라운드에서 수행하면, 사용자는 다른 작업을 수행하면서 작업이 완료될 때까지 기다리지 않아도 됩니다.

비동기 처리의 종류

Callback

장점

1. 콜백 함수는 비교적 간단하고 직관적입니다.
2. 콜백 함수를 사용하면 일반적으로 적은 코드 양으로 비동기 처리를 수행할 수 있습니다.
3. 콜백 함수는 JavaScript의 기본 문법으로 지원되므로 모든 브라우저와 환경에서 사용할 수 있습니다.

단점

1. 콜백 함수를 중첩하여 사용하면 콜백 지옥(callback hell) 문제가 발생할 수 있습니다.
2. 콜백 함수는 비동기 처리가 복잡한 경우에는 코드의 가독성을 해칠 수 있습니다.

Promise

장점

1. Promise는 비동기 처리에 대한 좀 더 명확한 추상화를 제공합니다.
2. Promise는 여러 처리를 위한 적절한 방법을 제공합니다. resolve나 reject를 통한 명확한 처리가 가능합니다.
3. Promise는 콜백 함수를 중첩하여 사용하는 콜백 지옥(callback hell) 문제를 해결합니다.

단점

1. Promise를 이해하는 데 일정한 학습이 필요합니다.
2. Promise는 비동기 처리를 위한 복잡한 구조를 가지므로 코드의 가독성을 해칠 수 있습니다.

async / await

장점

1. Async/await는 코드의 가독성과 유지 보수성을 향상시킵니다.
2. Async/await는 비동기 함수를 동기 함수처럼 작성할 수 있도록 해줍니다.
3. Async/await는 Promise를 기반으로 동작하며, 비동기 처리 코드를 간결하게 작성할 수 있습니다.

단점

1. Async/await는 Promise를 기반으로 동작하기 때문에 Promise의 단점과 유사합니다.
2. Async/await를 사용하면서 실수로 await 키워드를 빠뜨리는 등의 오류가 발생할 수 있습니다.

설명 강의

자바스크립트 11. 비동기 처리의 시작 콜백 이해하기, 콜백 지옥 체험 🙌 JavaScript Callback | 프론트엔드 개발자 입문편 (JavaScript ES6)

자바스크립트 비동기 처리를 위한 그 출발점, 콜백에 대해 정확하게 이해하고, 두가지 종류의 콜백과 콜백지옥은 무엇인지 한번 예제를 통해 알아보도록 해요. 다음에 이어지는 Promise 편에서 콜백지옥을 깔끔하게 바꿔보도록 할테니 꼭 실습해 보세요 🏠 (📄 자세한 내용)

📺 <https://www.youtube.com/watch?v=s1vpVCrT8f4>

자바스크립트 12. 프로미스 개념부터 활용까지 JavaScript Promise | 프론트엔드 개발자 입문편 (JavaScript ES6)

자바스크립트 비동기 처리를 위한 그 시작, 프로미스에 대해 정확하게 이해하고, 지난 시간에 만든 콜백지옥을 프로미스를 이용해서 간결하게 만들어 보면서 더 이해하는 시간을 가져볼게요 ❤️ 다음에 이어지는 async/await 편에서 만나요 🏠 (📄 자세한 내용)

📺 https://www.youtube.com/watch?v=JB_yU6Oe2eE

자바스크립트 13. 비동기의 꽃 JavaScript async 와 await 그리고 유용한 Promise APIs | 프론트엔드 개발자 입문편 (JavaScript ES6)

자바스크립트 비동기의 꽃 async와 await에 대해 정확하게 이해하고, 지난 시간에 배운 프로미스와 비교해서 재밌게 이해하는 시간을 가져볼게요 ❤️ (📄 자세한 내용)

📺 <https://www.youtube.com/watch?v=aoQSOZfz3vQ>