
<JS의 동작 원리 이해와 브라우저에서의 실행>

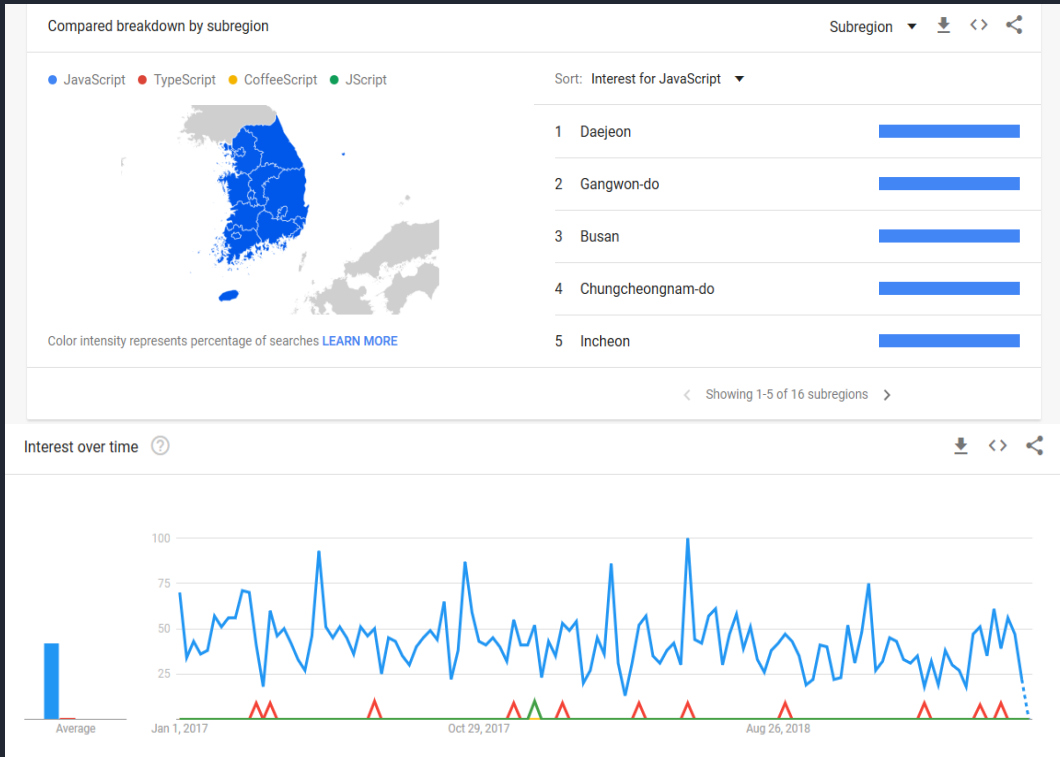
JS엔진 동작원리와 브라우저별 사용엔진과 최적화 방법 및 JS의 실행과정.

작성일: 2019. 05. 16

목차

1. 개요-웹 프론트 엔드 언어, JavaScript
 1. 유일한 웹 프론트 엔드 언어, JavaScript 원리 이해의 중요성
 2. JavaScript 의 목적과 역사
2. 본론- JavaScript 의 특징 및 세부적인 동작 원리와 각 브라우저별 최적화 방법
 1. JavaScript 의 특징
 2. JavaScript 의 동작 원리
 3. 웹 브라우저별 사용 JavaScript 엔진과 각각의 최적화 방식의 차이
 4. 브라우저상에서의 JavaScript 실행 과정
3. 평가
 1. 단일 call stack 언어인 JavaScript
 2. 인터프리터 언어인 JavaScript
 3. JavaScript 의 readability
 4. JavaScript 의 writability
 5. JavaScript 의 reliability
4. 결론-요약 및 재언
 1. 요약
 2. 재언- JavaScript 동작 원리 이해의 중요성.

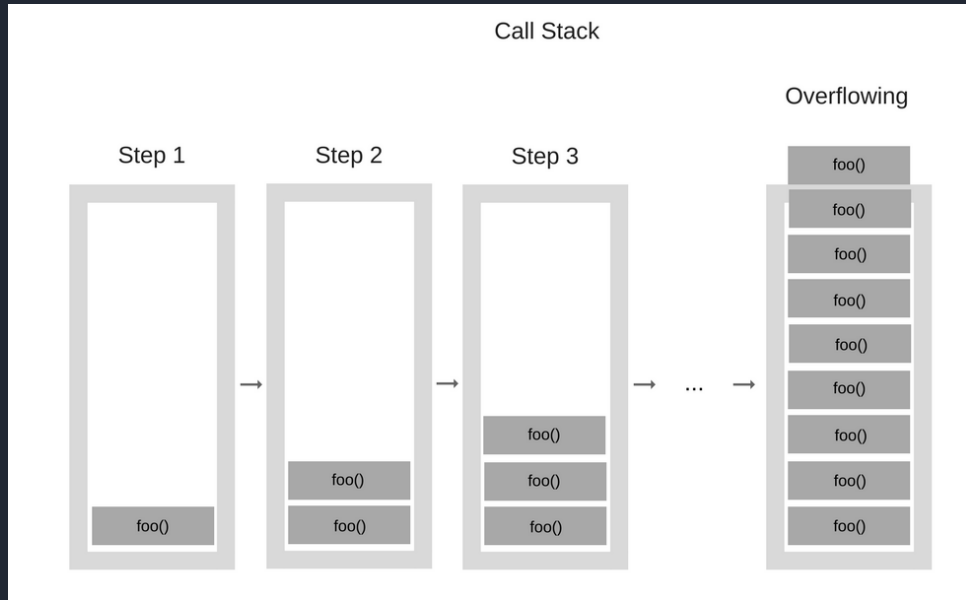
개요- 유일한 웹 프론트 엔드 언어, JavaScript 원리 이해의 중요성



웹사이트	인기도 (1달 간 방문객 수) ^[1]	프론트엔드 (클라이언트 측)
구글 ^[2]	1,600,000,000	JavaScript
페이스북	1,100,000,000	JavaScript
유튜브	1,100,000,000	JavaScript
야후!	750,000,000	JavaScript
아마존	500,000,000	JavaScript
위키피디아	475,000,000	JavaScript
트위터	290,000,000	JavaScript
빙	285,000,000	JavaScript
이베이	285,000,000	JavaScript
MSN	280,000,000	JavaScript
마이크로소프트	270,000,000	JavaScript
Linkedin	260,000,000	JavaScript
Pinterest	250,000,000	JavaScript
워드프레스	240,000,000	JavaScript

- JavaScript 는 동적으로 콘텐츠를 바꾸고, 멀티미디어를 다루는 등 웹 페이지를 꾸며주도록 하는 프로그래밍 언어인데, HTML, CSS 등과 더불어 사용되는 현재 유일하게 쓰이는 웹 프론트 엔드 언어임.

- JavaScript 는 ECMA에서 표준 사양을 만들고 있으므로 웹 표준 기술로서 입지가 탄탄함. 위의 두 이유 때문에 앞으로도 동적인 웹 개발을 하기 위해 JavaScript 를 잘 알고, 활용할 수 있어야함.



- 재귀함수를 이용해 문제를 해결 했을 때, 상황에 따라 이를 반복문으로 해결 할 수 있다면 반복문으로 해결 하는 것이 더 좋은 경우가 있다.
- 구조적으로 for문을 이용하는 것이 Overhead가 더 적고, 비교적 stack 메모리 상의 문제가 생길 확률도 더 적기 때문.
- 이렇듯 언어를 사용할 때는 언어가 가진 특성과 작동하는 원리를 잘 이해하고 사용해야 해야함. 하지만 많은 개발자들이 JavaScript를 사용할 때 내부작동 방식이나 언어특성, 실행되는 환경에 대한 낮은 이해를 가지고 개발하는 경우가 많음.
- 이번 발표에서는 JavaScript 작동에 대한 이해를 높이기 위해 JavaScript 특성에 대한 설명과, 실행되는 엔진 내부의 동작을 알아보고 브라우저별로 사용하고 있는 엔진의 코드 interpret 방식과 브라우저 내에서의 실행 순서를 알아볼 것이다.



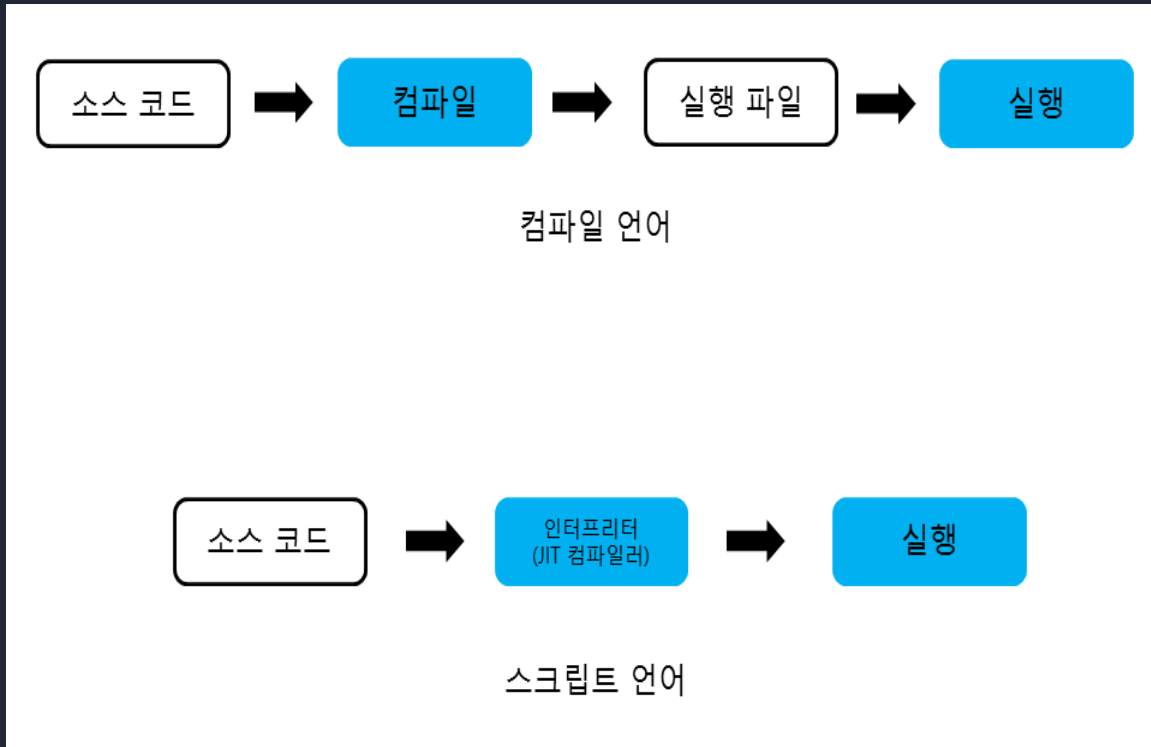
- 자바스크립트는 1995년 넷스케이프 커뮤니케이션즈에서 HTML페이지에 경량의 프로그램 언어를 통하여 인터렉션을 구현하기 위해 브렌던 아이크가 개발.
- 자바스크립트(영어: JavaScript)는 객체 기반의 스크립트 프로그래밍 언어.
- 웹 브라우저 내에서 주로 사용되며 웹페이지 내의 요소들을 동적으로 조작하기 위해 사용.
- JavaScript 는 Java와 전혀 관련이 없음.
 - 브렌던 아이크(Brendan Eich)가 처음에는 모카(Mocha)라는 이름으로, 나중에는 라이브스크립트(LiveScript)라는 이름으로 개발하였으며, 최종적으로 자바스크립트가 되었다.



JavaScript

- 변수스코프와 클로저등의 규칙은 Lisp dialect Scheme 에서 가져옴.
- 프로토타입 상속은 스몰토크에서 파생된 Self 프로그래밍 언어에서 영향을 받음.
- 자바스크립트가 나온 이후 , MS사는 IE 3.0에서 동작하는 'JScript'라는 똑같은 언어를 만들어냄.
- 넷스케이프사는 자바스크립트를 표준화 하기 위해, 표준화 기구인 ECMA International에 요청을 했고,ECMA-262라 불리는 명세서가 나오면서 ECMAScript라는 표준언어가 생김
- 1997년 JavaScript를 이용해 DOM등을 다루어 콘텐츠를 변화시키는 동적 HTML을 구현할 수 있게 됨.
- 1999년, XMLHttpRequest API가 발표되면서 클라이언트측 스크립트를 http또는 https로 요청 후 텍스트로 받음.

JS의 특징 및 세부적인 동작 원리와 각 브라우저별 최적화 방법



- JavaScript는 웹 요소를 제어하기 위한 스크립트 언어로 개발됨.
- 스크립트언어는 컴파일러 언어와 달리 실행시에 인터프리터를 이용해 한 줄 한 줄 번역하는 방식으로 구동.
- 인터프리터 언어는 비교적 느리지만 바로 실행할 수 있고 동작을 확인해가면서 프로그램을 개발할 수 있다는 장점이 있음.
- 다만 V8, SpiderMonkey 엔진 등등 에서 JIT 컴파일러 기능을 지원해주어 속도면에서의 단점이 극복 됨.


```
function Person(gender) {  
  this.gender = gender;  
  alert('Person instantiated');  
}  
Person.prototype.sayHello = function()  
{  
  alert ('hello');  
};  
var person1 = new Person('Male');  
var person2 = new Person('Female');  
// call the Person sayHello method.  
person1.sayHello(); // hello
```

▲위의 예에서는 Person 함수 객체에 sayHello()라는 메서드를 정의하고 사용하고 있다.

- JavaScript 는 처리와 관련된 데이터와 절차를 하나의 객체로 묶어 관리하는 OOP 언어의 특성을 가지고 있다.
- 다만 C++와 Java 와 달리 Class를 이용해 객체를 생성하지 않고 프로토타입을 상속하는 프로토타입 기반 객체 지향 언어.
- JS에선 함수가 일급 객체로 사용된다. 또한 함수에서 함수를 인수로 넘길 수 있다. 이러한 특성때문에 함수형 프로그래밍언어의 특성을 가짐.
- 최근 ES6에 여러 객체지향 문법이 추가되면서 자바나 C++같은 다른 객체지향 언어들과 비슷한 방식으로 객체지향 프로그래밍을 할 수 있게 되기도 함.

```
var A=10;  
let B=1.1;  
var C='10';
```

VS

```
int A=10;  
float B=1.1;  
String C='10';
```

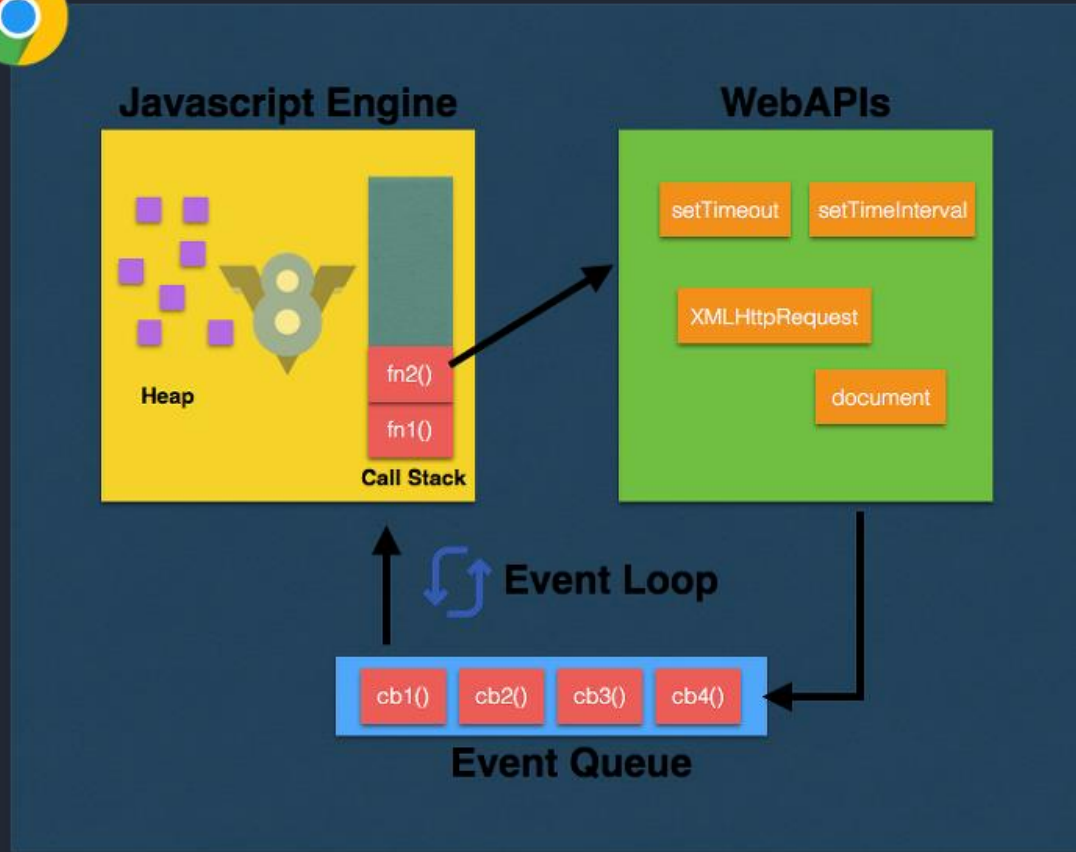
- JavaScript 는 Type binding을 동적으로 진행함. 따라서 프로그램을 실행하는 도중에 변수에 저장되는 데이터 타입이 동적으로 바뀔 수 있음.
- 이러한 특성 때문에 JavaScript는 var , let 등을 이용해 Implicit declaration 을 해주어 타입을 undefined로 두고 Runtime 시에 값을 변수에 할당할 때 타입을 바인딩.
- 또한 JavaScript 는 동적 언어이기 때문에 객체를 생성 한 후에도 프로퍼티와 메서드를 동적으로 추가/삭제 할 수 있음.
- 동적 특성 때문에 유연성이 증대되지만 runtime 시에 overhead가 비교적 큼.
- 동적 타입 바인딩의 단점을 극복하기 위해 Typescript가 탄생함.

```
window.onload = function(){  
  Foo1();  
  Foo2();  
};  
//페이지가 로드되면 두 함수가  
자동으로 실행될 수 있도록  
window 객체의 전역 콜백 함수인  
onload를 이용.
```

```
function updateTXT(text){  
  let R=document.getElementById("TXT");  
  R.innerHTML=text;  
};  
//DOM을 이용해 도큐먼트 내의 태그에 접근해 값을 바  
꿈.
```

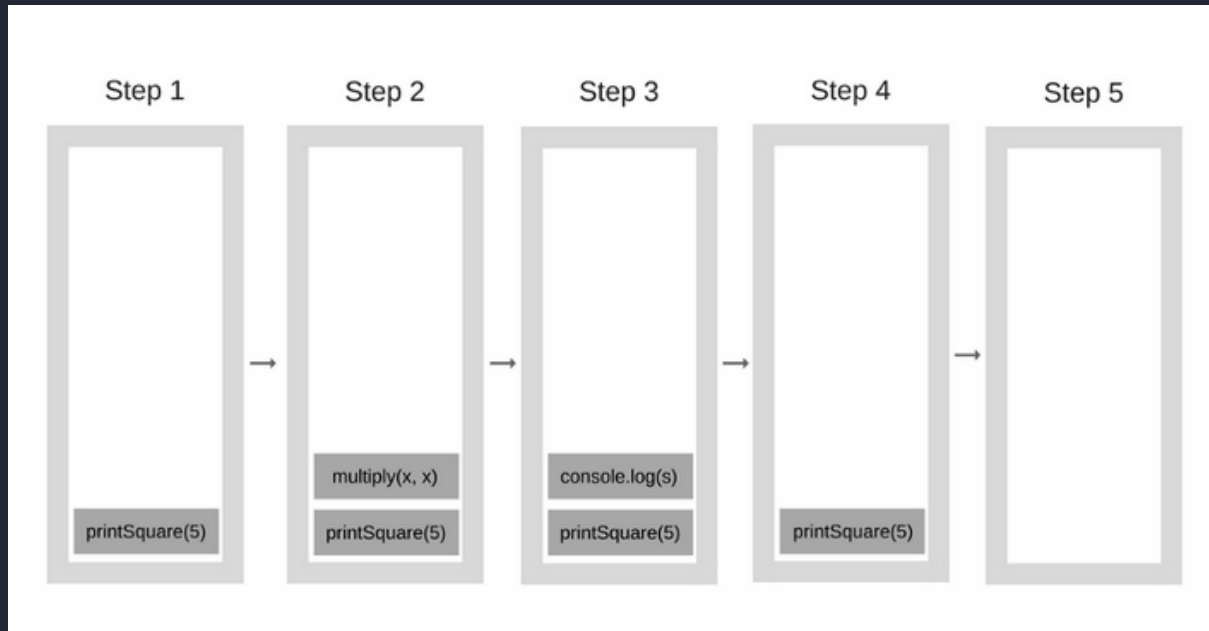
- 클라이언트 측 JavaScript는 ECMA가 규정한 코어 언어와 웹 브라우저의 API로 구성 되어있음.이를 이용해 웹 요소들을 조작할 수 있음.
- 주요 API
 1. window 인터페이스: 자바스크립트로 브라우저 또는 창을 조작하는 기능을 제공.
 2. DOM: 자바스크립트로 HTML 문서의 요소를 제어하는 기능을 제공.
 3. XMLHttpRequest: 서버와 비동기로 통신하는 기능을 제공.

본론-JS의 동작 원리- 브라우저 환경에서의 동작



- JS엔진 내부요소와 브라우저 환경의 기본적인 구조도.

- 브라우저 환경에서의 JS 엔진은 크게 메모리 힙 영역과 콜 스택으로 이루어져 있음.
- Memory Heap : 메모리 allocation이 일어나는 곳.
- Call Stack : 코드 실행에 따라 호출 스택이 쌓이는 곳으로 Js에서는 단 하나의 Call Stack 만을 이용.
- WebAPIs : 브라우저에서 제공하는 API로서 브라우저 내부 객체 조작 인터페이스를 제공하거나, 정보를 제공해주어 web개발을 용이하게 함.
- Event loop : 현재 실행중인 태스크가 있는지, 태스크 큐에 태스크가 있는지 반복적으로 확인해서 현재 실행중인 태스크가 없을 때 큐의 첫번째 태스크를 꺼내와 실행.
- Event Que : 콜백함수들이 대기하는 큐



```
function multiply(x, y) {  
  return x * y;  
}
```

```
function printSquare(x) {  
  var s = multiply(x, x);  
  console.log(s);  
}
```

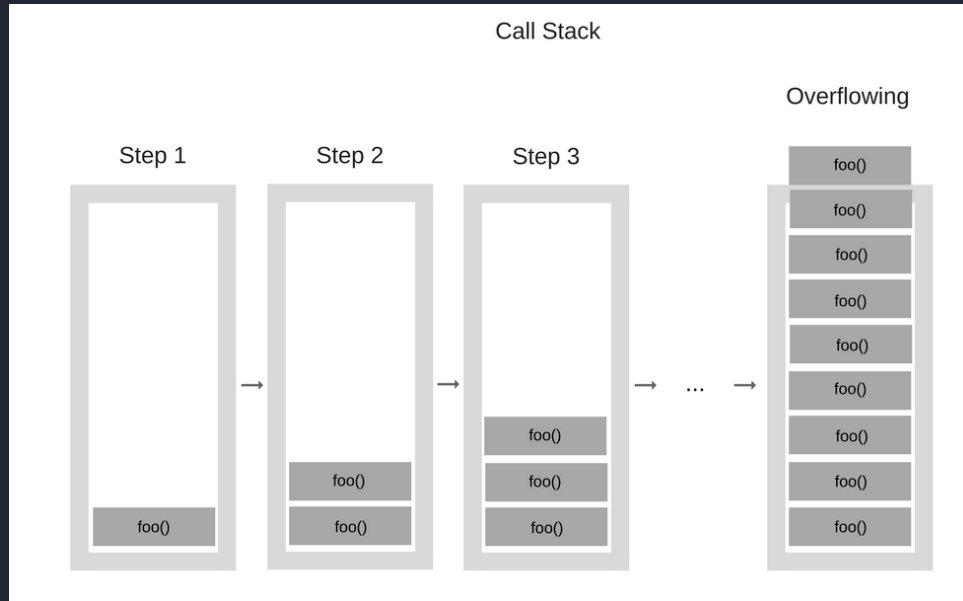
```
printSquare(5);
```

- JS는 단일 call stack의 단일 쓰레드 기반 언어로써 하나의 콜 스택을 이용해서 함수 호출에 따라 처리 해 줌.

• Run-to-Completion

1. 단일 call stack을 이용하기 때문에 하나의 함수가 실행되면 다음 함수가 실행될 때 까지는 어떤 작업도 중간에 끼어들지 못하는 방식.
 2. Deadlock이 발생하지 않는 장점이 있음.
 3. 그러나 하나의 프로세스가 실행되는 동안 Block 되기 때문에 비동기 처리 등을 해주어야 함.
- `printSquare(5)`가 호출 된 후, 내부의 `multiply(x,x)` 함수, `Console.log(s)`가 순차적으로 처리된 후 `printSquare(5)`가 마지막으로 종료 된다.

본론-JS의 동작 원리-앞서 말한 재귀 함수의 문제점.

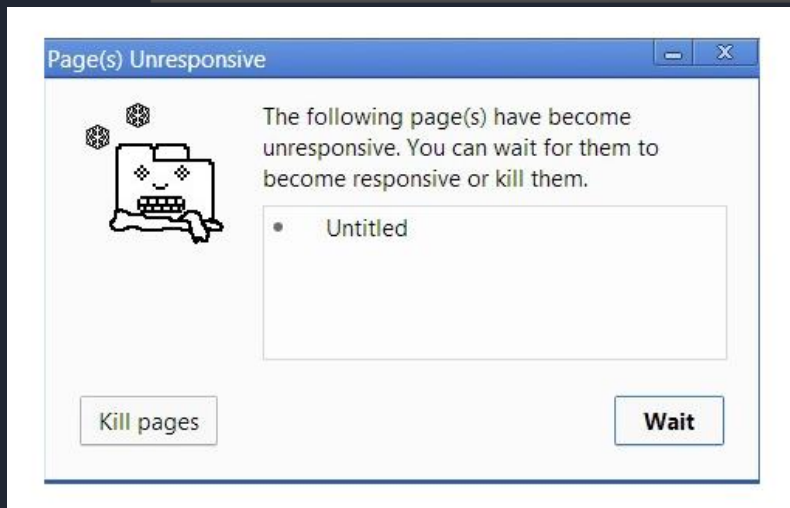


```
function foo() {  
  foo();  
}
```

```
foo();
```


- 코드가 실행 되는 구조를 잘 이해하고 사용하지 않으면 이러한 문제가 발생하기 쉽다.
- **Stack overflow** : 콜스택이 최대 크기에 다 달았을 때 나타나는 문제이다. 특히 재귀함수를 면밀히 테스트 하지 않은 경우 생긴다.
- 이러한 경우 상황에 따라 함수 내부에서 반복문을 이용해서 처리 해주는 것이 더 바람직함.
- 혹은 트램폴리닝 등의 결과대신 반환된 함수를 드라이버 함수가 실행시켜주는 방식으로 처리하면 해결 가능하다.

본론-JS의 동작 원리- 싱글 쓰레드, 콜 스택이 하나인 언어



```
function foo1(){  
  
    while(1){  
        console.log(".");  
    }  
}  
function foo2() {  
    alert("안와요!");  
}  
  
foo1();  
foo2();
```

- 싱글 쓰레드 기반 언어이기 때문에 콜스택 내에 수행시간이 너무 길거나 끝나지 않는 함수가 있을 경우 다음 함수가 실행되지 않음.
- JS 엔진은 하나의 함수를 실행 하고 있을 때 다른 함수를 실행할 수 없는 상태인 Block 상태가 됨. 이 경우 브라우저는 아무 일도 할 수 없고 이 상태가 지속될 경우 에러를 일으킴.
- 수행시간이 긴 함수를 수행하려면 비동기 콜백 등을 이용해서 해결해야 함.
- JS는 단일 스택을 이용하는 단일 스레드 기반의 언어지만 이러한 문제 때문에 실제 자바스크립트가 구동되는 브라우저 환경에서는 주로 여러 스레드가 사용된다. 이런 상황에서 단일 호출 스택을 사용하는 자바스크립트와 연동을 위해 event loop과 Callback Queue 사용.

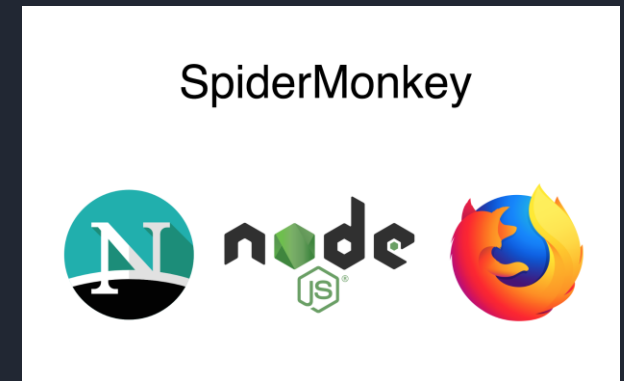
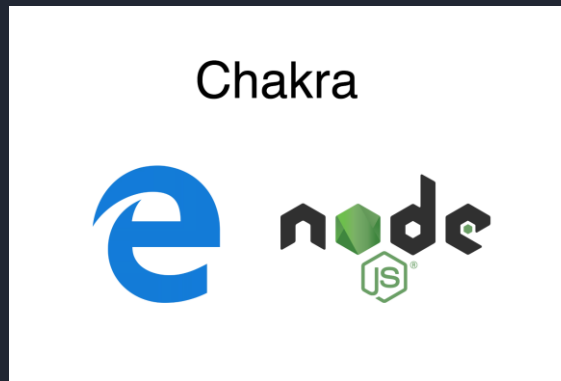
 당신의 브라우저를 느려지게 만드는 웹페이지가 있습니다. 무엇을 하시겠습니까?

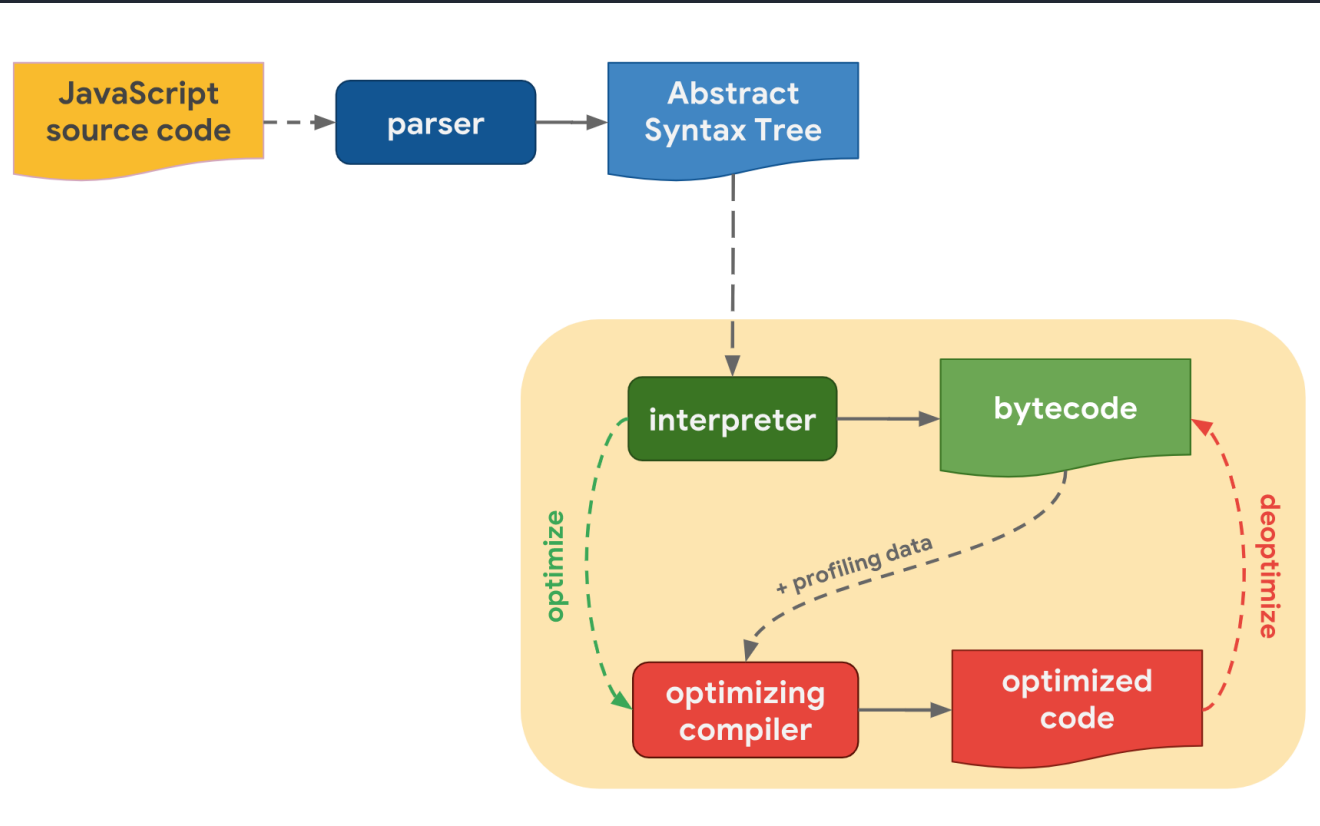
```
function first() {  
  console.log('first');  
}  
function second() {  
  console.log('second');  
}  
function third() {  
  console.log('third');  
}  
first();  
setTimeout(second(), 1000); // Invoke `second`  
after 1000ms  
third();
```

- setTimeout 뿐만 아니라 브라우저의 다른 비동기 함수들 이나 Node.js의 IO 관련 함수들 등 모든 비동기 방식의 API들은 이벤트 루프를 통해 콜백 함수를 실행함.
- 샘플코드는 다음과 같은 순서로 실행
 1. first();실행 first출력
 2. setTimeout(second,1000);를 callstack에 올린 후 API에 1000ms후에 처리를 요청.
 3. API에서 setTimeout()처리 후 1000ms후 callback queue에 전달.
 4. Third();실행 third 출력.
 5. Event loop가 지속적으로 call Stack 이 비어 있는지 확인하다가 없을 때 second를 call Stack 에 전달한다.
 6. second(); 실행, second 출력

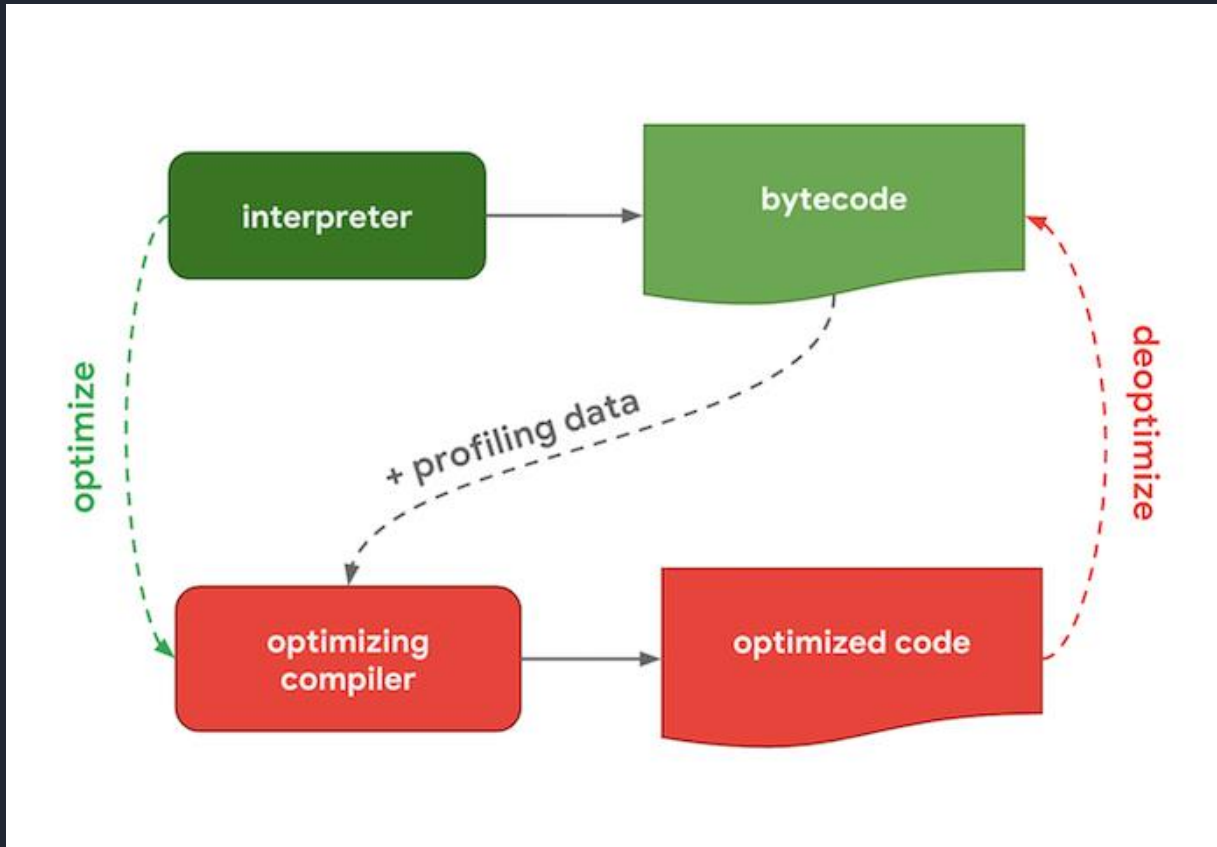


- V8:오픈소스, 구글에서 개발해 Chrome 에서 사용, C++로 작성됨
- **SpyderMonkey** : 최초의 자바스크립트 엔진. 예전에는 넷스케이프 네비게이터에 사용됐고 지금은 파이어폭스에 사용됨
- **JavaScriptCore**: 오픈소스, 니트로라는 이름으로도 알려져 있으며 애플이 사파리를 위해 개발함
- **Chakra(Jscript8)**: 인터넷익스플로러
- **Chakra(JavaScript)**: 마이크로소프트엣지

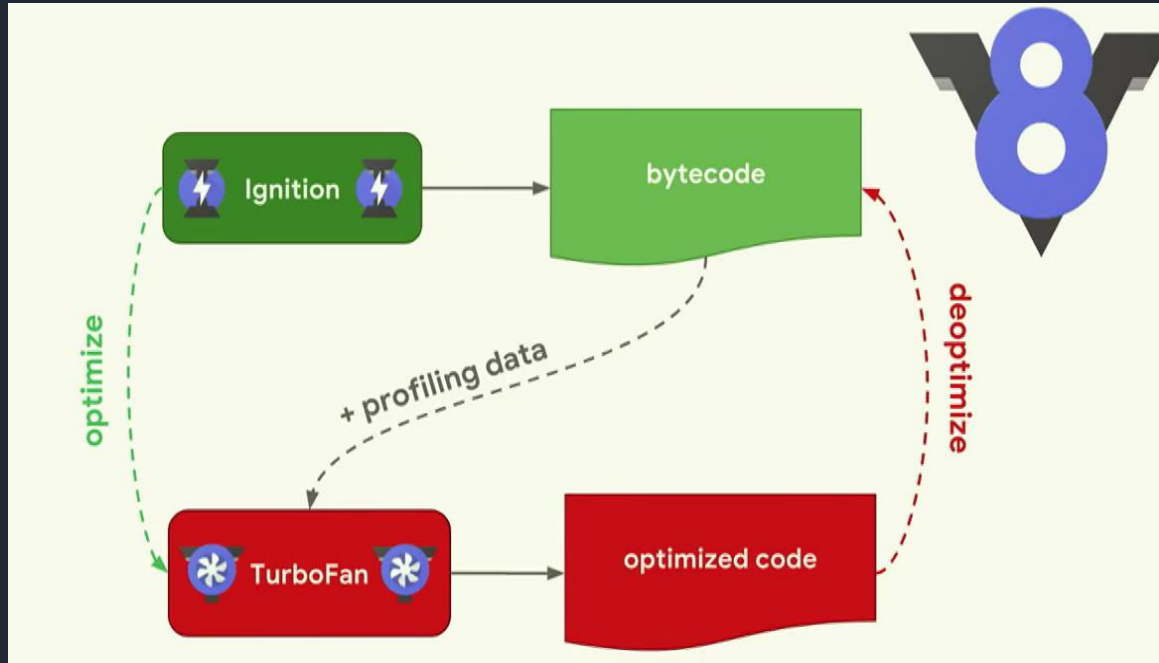




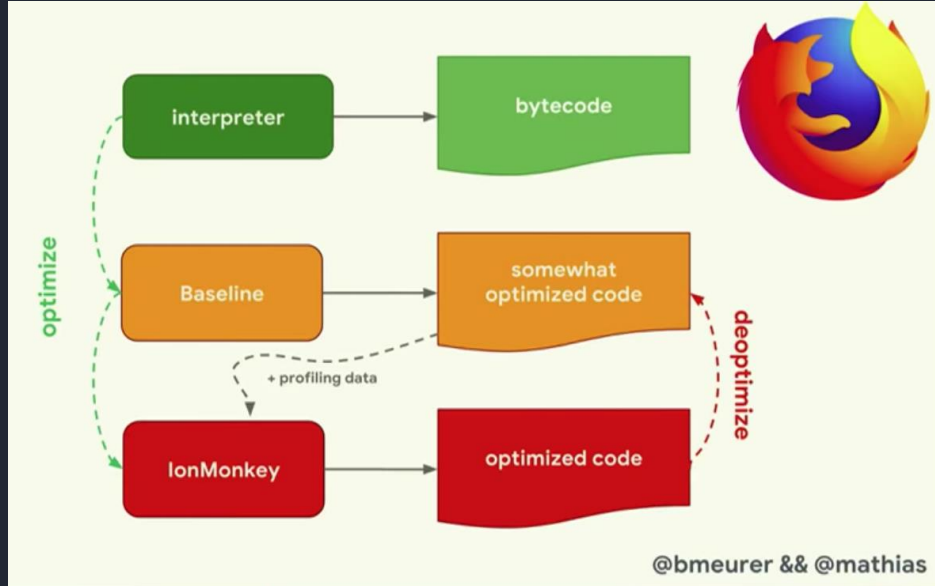
1. JavaScript 엔진은 소스를 파싱해서 AST를 생성함.
2. 인터프리터가 AST를 바탕으로 바이트 코드를 생성함.
3. 코드를 더 빠르게 실행하기 위해 바이트 프로파일링 된 데이터와 함께 최적화 컴파일러로 보내짐.
4. 프로파일링 데이터를 기반으로 최적화 컴파일러에서 매우 최적화된 기계어를 생성한다.
5. 만약 정확하지 않은 결과가 나왔다면 deoptimize 해서 다시 바이트코드로 되돌린다.



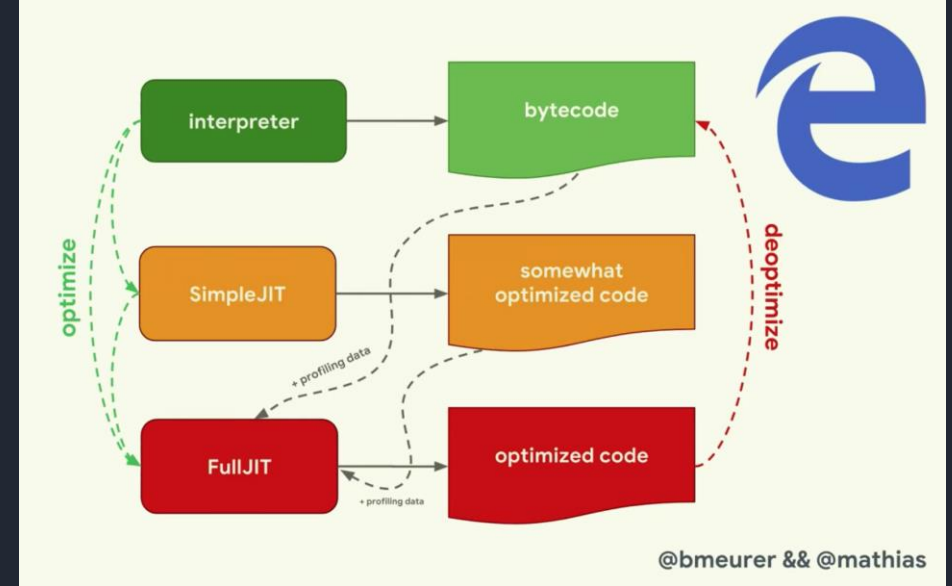
1. Interpreter: 최적화 되지 않은 바이트 코드를 생성한다.
2. Optimizing Compiler: 기존 바이트 코드를 기반으로 최적화 된 기계어 코드를 생성한다.
 1. 만약 인터프리트 모드라면 하나씩 읽어 실행한다.
 2. JIT 모드라면 바이트코드를 기반으로 컴파일 한다.



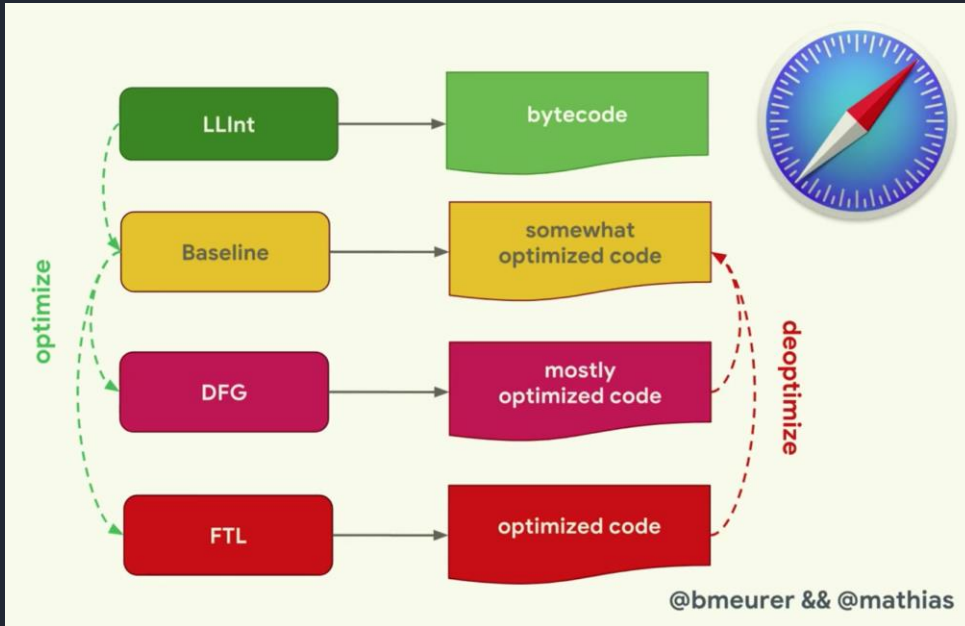
- 크롬은 V8엔진을 사용함.
- 5.8버전까진 풀 코드 젠과 크랭크 샤프트를 이용하며 **bytecode**를 생성하지 않았으나 메모리 overhead를 줄이기위해 **ignition bytecode interpreter**를 만들었다.
- V8 5.9버전부터는 인터프리터를 **Ignition**이라고 부르는데, 이곳에서 **bytecode** 생성 및 실행을 담당 하게됨.
- **Ignition**에서 바이트코드가 실행 될 때 프로파일링 데이터를 수집하여 실행 속도를 빠르게 함.
- 이때 자주실행 되는 함수가 있으면 이를 **Turbofan**으로 보내 최적화된 기계어 코드를 만들어 낸다.
- V8은 일괄적으로 최적화를 진행하는 것이 아닌 반복적으로 실행 되는 코드에 대해서만 최적화 해주는 **Adaptive Compilation**을 진행 한다.



- Firefox는 Spider Monkey를 엔진으로 사용.
- 인터프리터가 Baseline을 이용해 코드를 일차적으로 최적화 함.
- IonMonkey에서는 코드를 실행하면서 생긴 프로파일링 데이터와 Baseline를 이용해 고도로 최적화된 코드를 만들어 냄.
- 만약 최적화가 실패하면 IonMonkey는 이를 Baseline 단계 코드로 되돌림.



- Edge는 Chakra core를 엔진으로 사용한다.
- Chakra core또한 두개의 최적화 컴파일러를 가진다.
- 인터프리터가 Simple JIT로 bytecode를 보내 약간 최적화된 코드를 만듦.
- Full JIT에선 이를 프로파일링 데이터와 SimpleJIT의 코드를 이용해 더욱 최적화된 코드를 생성



- 사파리는 JavaScript Core를 엔진으로 이용.
- JavaScript Core는 최적화를 세번에 걸쳐 진행
- LLInt 로 interpret 한 후 Baseline 컴파일러를 이용해 heuristics 한 방법으로 최적화를 진행
- 일차적으로 최적화한 결과를 DFG와 FTL을 이용해 최적화를 한다.

사용자 인터페이스 : 웹 페이지를 보는 창을 제외한 브라우저 인터페이스.

브라우저 엔진 : 사용자 인터페이스와 렌더링 엔진 사이의 상호 작용을 담당.

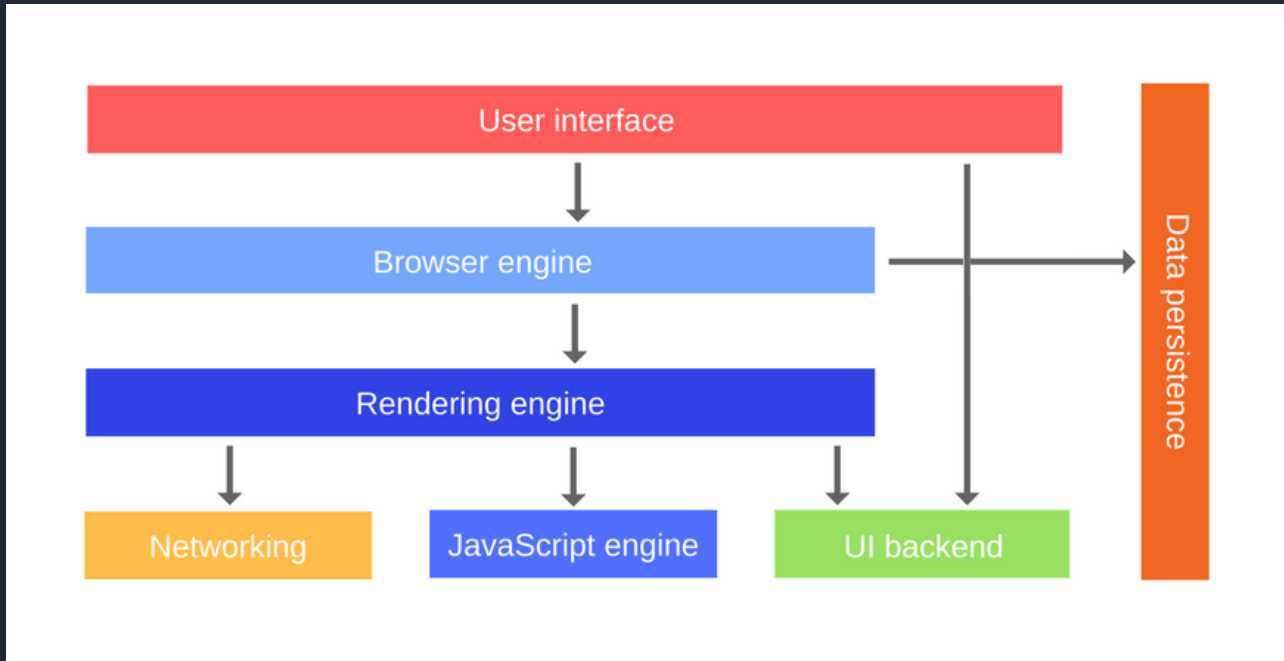
렌더링 엔진 : HTML과 CSS를 파싱하고 화면에 파싱된 내용을 표시.

네트워킹 : 이것은 다양한 플랫폼에서 여러 구현 방식을 사용해 만들어진 XHR 요청과 같은 네트워크 호출.

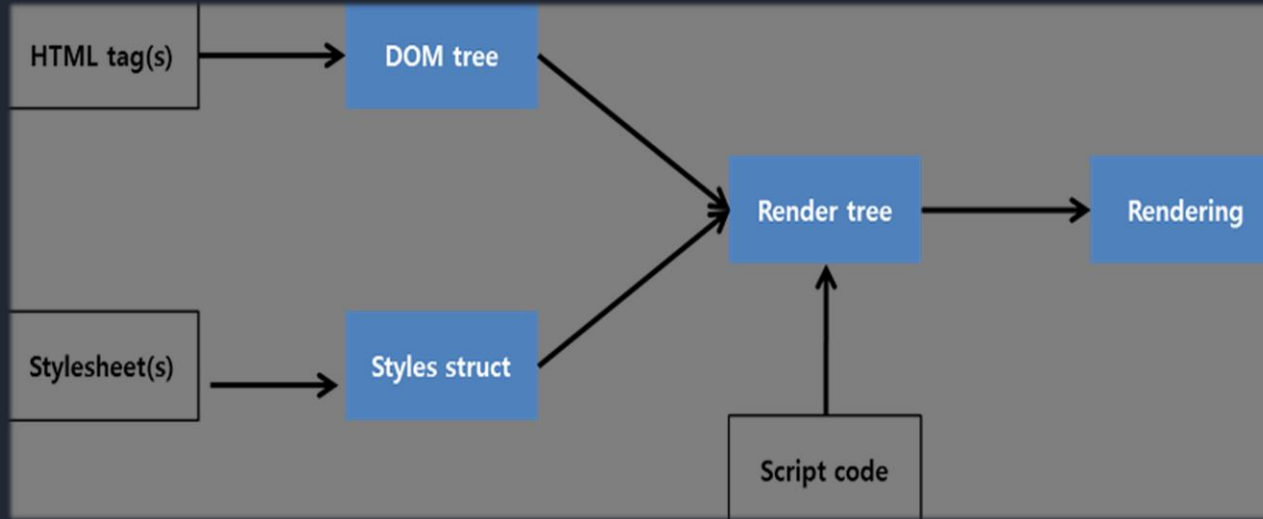
UI 백 엔드 : 체크박스 및 윈도우와 같은 핵심 위젯을 그리기 위해 사용.

자바스크립트 엔진 : V8, Spidermonkey 등 자바스크립트가 실행되는 엔진.

데이터 지속성(Data persistence) 앱 데이터 로컬 저장 .



본론- 브라우저상에서의 자바스크립트 실행 과정



```
<script src="../../test.js" ></script>
```

```
<script type="text/JavaScript" ></script>
```

1. 웹 브라우저로 웹페이지를 열면 Window객체 생성 window객체는 웹페이지의 전역객체로 웹페이지와 탭마다 생성.
2. Document객체가 window 객체의 프로퍼티로 생성되며 DOM Tree 구축.
3. HTML 구문을 작성 순서에 따라 분석하며 Document 객체 요소와 텍스트 노드를 추가해감.
4. HTML 문서안에 Script요소가 있으면 Script 요소 안의 코드 또는 외부 파일에 저장된 코드의 구문을 분석.
5. 오류가 발생하지 않았을 경우 Script를 동기적으로 실행 됨. (Script요소의 구문을 분석해서 실행할 때는 HTML 문서의 분석이
마침)


```
window.onload = function(){  
  let img1=document.getElementById("img1");  
  
  img1.width="300";  
  img1.height="300";  
  
};  
// 모든 리소스를 단계7 에서 읽어 오기 전에 4  
// 번 단계에서 script요소를 실행하게 되면 객체  
// 조작이 정상적으로 되지 않는다. 따라서 리소스를  
// 정상 적으로 읽어온 후 Window 객체를 상대로  
// load 이벤트가 생겼을 때 객체를 조작해줌.
```

6. 웹 브라우저는 DOM tree 구축 완료 후 Document 객체에 이를 알리기 위해 DOMContentLoaded 이벤트를 발생시킨다.

7. Img 등의 요소가 이미지 파일 등의 외부리소스를 읽어 들임.

8. 모든 리소스를 읽어들인 후 document.readyState값이 "complete"로 바뀌며 Window 객체를 상대로 load 이벤트를 발생 시킴.

9. 다양한 이벤트를 수신하며 이벤트가 발생하면 이벤트 처리가 비동기로 호출됨.

평가

- **JavaScript**는 단일 쓰레드이기 때문에 다음과 같은 특성이 있었음.
 1. 하나의 함수가 실행되면 다음 함수가 실행될 때 까지는 어떤 작업도 중간에 끼어들지 못 함.
 2. Deadlock이 발생하지 않는 장점이 있음.
 3. 하나의 프로세스가 실행되는 동안 Block 되기 때문에 비동기 처리 등을 해주어야 함.
- 비동기 처리를 위해 web Worker 등의 API를 사용할 필요가 있음.

Worker는 Worker() 생성자를 통해 생성되며 지정된 Javascript 파일에 포함된 코드를 Worker 쓰레드에서 실행합니다. (Worker는 현재 Window와 분리된 DuplicatedWorkerGlobalScope라는 별도의 Global context에서 동작합니다.) Worker 쓰레드에서 어떠한 코드도 실행할 수 있지만, 몇가지 예외가 있습니다. 예를들어 Worker 내에서는 DOM을 직접 다룰 수 없습니다. 또한 Window의 기본 메서드와 속성을 사용할 수 없습니다.-MDN

- JavaScript는 스크립트 언어로 다음과 같은 특징이 있었음.
 - 인터프리터를 이용해 실행시에 한 줄 한 줄 번역하는 방식으로 구동.
 - 인터프리터 언어는 바로 실행할 수 있고 동작을 확인해가면서 프로그램을 개발할 수 있다는 장점이 있음.
- 그러나 실행시간에 기계어로 번역하기 때문에 순수 컴파일 방식에 비해 매우 느림.
 - 이 때문에 이를 해결하기 위해 JIT 컴파일러와, 이와 관련된 최적화 방식들이 나옴.

여러 JIT 최적화 방법이 있는 이유?

- 인터프리터는 바이트코드를 빠르게 생성할 수 있지만 효율적인 코드가 아님.
- 엔진마다 구체적인 부분에서 차이가 있긴 하지만 모두 parser, interpreter, optimize compiler가 포함된 같은 구조를 이용.
- 최적화 컴파일러는 느리지만 최적화된 기계어 코드를 생성
- 결국 두 요소의 trade-off를 잘 맞추는 것이 중요.

Table 1.1 Language evaluation criteria and the characteristics that affect them

Characteristic	CRITERIA		
	READABILITY	WRITABILITY	RELIABILITY
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

- JavaScript의 Readability 는 좋지 않기로 유명함. Readability는 유지 보수에도 큰 영향을 미침.
- JavaScript는 전반적으로 뛰어난 writability 를 가지지만 그 trade-off로 Readability가 많이 떨어지는 편

```
var A;  
var B;  
var C;
```

```
A=0;  
B=0.1;  
C="type??";
```

- JavaScript의 특성 중 Implicit 한 type은 writability를 향상 시켜주지만 선언만을 보고 어떤 값이 들어갈지 짐작 할 수 없게 만들어 Readability에 악영향을 미침.

```
function funcDeclarations() {  
  return 'A function declaration';  
}  
funcDeclarations();  
//함수 선언식-호이스팅에 영향을 받음.
```

```
var funcExpression = function () {  
  return 'A function expression';  
}  
funcExpression();  
//함수 표현식- 호이스팅에 영향을 받지 않음.
```

- 함수를 선언하는 방법이 여러가지임. 함수 표현식, 함수 선언식 등. -feather multiplicity

```
logMessage();//정상 작동 hoisting 적용  
sumNumbers();//에러!-함수로 인식하지 못 함.
```

```
function logMessage() {  
  return 'worked';  
}
```

```
var sumNumbers = function () {  
  return 10 + 20;  
};
```

//같은 함수임에도 함수 선언 방식에 따라 호이스팅 여부가 결정됨.

- 변수와 함수의 hoisting 때문에 Scope 영역이 블록의 최 상위부터 시작돼 Readability를 저해함.

```
function printX(){  
  console.log(x);//undefined  
  var x=100;  
  console.log(x);//100  
}
```

```
printX();
```

- 선언문은 항상 자바스크립트 엔진 구동시 가장 최우선으로 해석하기 때문에 변수가 존재 하긴 하지만 type이 정해지지 않은 undefined 가 됨.
- 할당 구문은 런타임 과정에서 이루어지기 때문에 호이스팅 되지 않음.

```
0 == '' //true
0 == '0' //true
1 == true //true
false == '0' //true
null == undefined //true
false == null //false
false == undefined //false
```

```
0 === '' //false
0 === false //false
1 === true //false
NaN === NaN //false
null === undefined //false
```

- “==”, “!=” 등의 연산자가 두 비교 객체의 type이 다를 경우 강제로 타입을 변환시켜 orthogonality를 저해 시킴.

- 암묵적 타입 변환 없이 강력하게 두 값을 타입까지 비교하고 싶을 때는 “===”, “!==” 등을 이용

문법	자료형	형변환 결과
<code>var result1 = "10"+20;</code>	문자+숫자	문자로 형변환
<code>var result2 = 10+"20";</code>	숫자+문자	문자로 형변환
<code>var result3 = "10"+true;</code>	문자+boolean	boolean으로 형변환

```
3 * "3" // 9
1 + "2" + 1 // 121
true + true // 2
10 - true // 9
1 + bar // "1 promise is a boy :)"
4 * [] // 0
4 * [2] // 8
4 + [2] // "42"
4 + [1, 2] // "41, 2"
4 * [1, 2] // NaN
```

- JavaScript의 Writability는 매우 우수한 편이다.
- 사용자에게 세부적인 사항에 대한 기술을 시키지 않도록 하는 여러 특성들과 묵시적 변환들이 있음.
- 변수를 선언할 때 var, let 등 추상적인 데이터 타입을 지원함
- Case Sensitive 특성을 가지고 있다.

```
function Apple (type) {  
  
  this.type = type;  
  
  this.color = "red";  
  
  this.getInfo = function() {  
    return this.color + ' ' + this.type + ' apple';  
  };  
  
}
```

- OOP를 지원하여서 프로세스의 뛰어난 추상화를 가능하게 함.

```
function Apple (type) {  
  this.type = type;  
  this.color = "red";  
}  
  
Apple.prototype.getInfo = function() {  
  return this.color + ' ' + this.type + ' apple';  
};
```

- 동적 언어 특성 및 prototype 를 이용하는 OOP의 특성을 가져 객체 생성 후에도 멤버 메소드의 추가가 가능.

```
function makeFunc() {  
    var name = "Mozilla";  
  
    function displayName() {  
        alert(name);  
    }  
  
    return displayName;  
}  
var myFunc = makeFunc();  
//myFunc변수에 displayName을 리턴함  
//유효범위의 어휘적 환경을 유지  
myFunc();  
//리턴된 displayName 함수를 실행(name 변수에 접근)
```

- 함수형 언어의 특성을 지원함.
- 자바 스크립트의 모든 함수는 클로저를 정의함.
- 클로저는 내부함수가 외부함수의 맥락에 접근할 수 있는 것을 말함.
- 함수안의 함수가 return 되더라도 return 하는 함수가 클로저를 형성하기 때문에 값을 유지한다.
- 클로저를 활용하면 변수를 은닉하여 지속성을 보장하는 등의 기능 구현 가능
- 옆의 예제는 displayName()함수가 실행되기 전에 외부함수인 makeFunc()로부터 리턴되어 myFunc 변수에 저장됨.

```
var a = 13; // Number 선언  
var b = "thirteen"; // String 선언
```

VS

```
int a = 13; // int 선언  
String b = "thirteen"; // String 선언
```

- Java와 달리 loosely typed language 임

```
try {  
  foo.bar();  
} catch (e) {  
  if (e instanceof EvalError) {  
    alert(e.name + ": " + e.message);  
  } else if (e instanceof RangeError) {  
    alert(e.name + ": " + e.message);  
  }  
  // ... etc  
}
```

- 예외처리를 지원한다.

요약 및 재언

1. JavaScript의 역사와 JavaScript의 사용 동향을 보며 웹 프론트 엔드를 개발 하기 위해 필수적으로 알아야 함을 알아봄.
2. 재귀문을 예로 들며 프론트 엔드 개발을 하더라도 JavaScript의 작동 원리를 잘 이해 해야 함을 알아봄.
3. JavaScript에 대한 전반적인 이해를 위해 JavaScript의 특성을 알아 봄.
 - Interpreter 언어임.
 - OOP의 특성과 함수형 언어의 특성을 동시에 가지고 있다.
 - 동적 언어로서 let, var 과 같은 implicit 한 type를 사용한다.
 - 브라우저에서 지원하는 API등을 이용해 웹요소를 조작 할 수 있다.
4. JavaScript가 어떻게 동작하는지 알기 위해 JavaScript엔진의 내부 요소를 공부 해봄.
 - JavaScript엔진은 크게 heap 영역과 call stack으로 나뉘고 이외의 call back queue, event loop, API 등의 요소가 있었음.
 - JavaScript는 Call stack 이 하나기 때문에 단일 쓰레드로 작업을 처리함.
 - 단일 쓰레드라는 특성 때문에 비동기 처리를 해주지 않으면 하나의 함수를 처리하는데 너무 오랜 시간이 걸려 브라우저에서 에러를 일으킴.
 - 이런 문제를 해결하기 위해 event loop, web worker 등이 있었음.

5. JavaScript가 해석 되고, 실행되는 과정을 이해 하기 위해 브라우저별 JavaScript 엔진을 알아보고 각각의 코드 최적화 방식을 공부함.

- V8(chrome), SpyderMonkey (firefox), JavaScriptCore(사파리), Chakra(window edge).
- V8 : ignition bytecode interpreter 를 이용해 바이트 코드로 번역을 한 후 자주 실행 되는 코드에 대해 Turbofan으로 최적화.
- SpydeMonkey : 인터프리터가 Baseline을 이용해 바이트코드를 최적화된 기계어로 바꾼 후 IonMonkey가 이차로 최적화 .
- JavaScriptCore : JavaScript Core는 최적화를 세번에 걸쳐 진행 LLInt 로 interpret 한 후 Baseline 으로 일차적으로 최적화한 결과를 DFG와 FTL을 이용해 최적화.
- Chakra : 인터프리터가 일차적으로 번역한 바이트코드를 기반으로 Simple JIT가 최적화 하고, Full JIT에서 프로파일링 데이터와 simple JIT의 최적화 결과를 이용해 다시 최적화.

6. 웹페이지 렌더링 요소와, 웹페이지 렌더링 시 JavaScript의 실행에 대해 알아봄.

7. JavaScript의 특성에 대한 평가와 JavaScript의 언어에 대한 평가.

- 단일 쓰레드 언어인 JS
- 인터프리터 언어인 JS
- **Writability**
- **Readability**
- **Reliability**

- JavaScript는 웹 프론트 엔드 개발을 하기 위해 꼭 알아야 하는 언어임.
- 코드를 작성할 때는 해당 언어의 특성과 작동 원리를 잘 알고 있어야 문제 발생률을 줄이고, 효율성을 증대 시킬 수 있음. 특히 JavaScript는 단일 스택 언어로, 이러한 특성을 잘 이해해야 비동기 방식의 처리와 call back 함수 등을 효율적으로 활용할 수 있음.
- JavaScript 는 인터프리터 언어로써 구동이 컴파일러 언어보다 느리다. 이를 해결하기 위해 여러 최적화 방식과 JIT 방식이 있었음.
- 우리가 알아봤듯 JavaScript가 작동하는 원리는 간단하지 않다. 또한 JavaScript의 판독성은 좋지 못함.
 - 하지만 잘 사용할 경우 뛰어난 프로세스의 추상화를 지원해 탁월한 Writability를 제공함.
 - 더더욱 언어의 특성과 원리를 이해 하고 사용해야 하는 이유가 됨.
 - 본 내용 이후의 스터디 방향은 이러한 JavaScript의 작동 원리를 고려해 최적화된 코드를 작성하는 방법과 가독성이 좋은 코드를 작성하는 방법.

[1]

- <https://trends.google.co.kr/trends/explore?cat=31&geo=KR&q=javascript,jscript,typeScript>
- https://ko.wikipedia.org/wiki/%EC%9C%A0%EB%AA%85%ED%95%9C_%EC%9B%B9%EC%82%AC%EC%9D%B4%ED%8A%B8%EC%97%90_%EC%82%AC%EC%9A%A9%EB%90%98%EB%8A%94_%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%98%EB%B0%8D_%EC%96%B8%EC%96%B4%EB%93%A4

[2],[3]

- "The Past, Present, and Future of JavaScript" Axel Rauschmayer

[4],[5],[6],[7]

- 모던 자바스크립트 입문, 이소히로시, 한빛 미디어
- <https://stackoverflow.com/questions/107464/is-javascript-object-oriented>
- <https://codeburst.io/javascript-object-oriented-programming-using-es6-3cd2ac7fbbd8>

[8], [9], [10], [11], [12]

- <https://meetup.toast.com/posts/89>
- <https://engineering.huiseoul.com/%EC%9E%90%EB%B0%94%EC%8A%A4%ED%81%AC%EB%A6%BD%ED%8A%B8%EB%8A%94-%EC%96%B4%EB%96%BB%EA%B2%8C-%EC%9E%91%EB%8F%99%ED%95%98%EB%8A%94%EA%B0%80-%EC%97%94%EC%A7%84-%EB%9F%B0%ED%83%80%EC%9E%84-%EC%BD%9C%EC%8A%A4%ED%83%9D-%EA%B0%9C%EA%B4%80-ea47917c8442>

[13],[14],[15],[16],[17],[18]

- <https://velog.io/@godori/JavaScript-%EC%97%94%EC%A7%84-%ED%86%BA%EC%95%84%EB%B3%B4%EA%B8%B0-mdjowmjlcb#%EC%B2%98%EB%A6%AC-%EB%B0%A9%EB%B2%95%EC%9D%98-%EC%B0%A8%EC%9D%B4%EC%A0%90%EA%B3%BC-%EA%B3%B5%ED%86%B5%EC%A0%90>
- <https://shlrur.github.io/javascripts/javascript-engine-fundamentals-optimizing-prototypes/>

[19], [20], [21]

- 모던 자바스크립트 입문, 이소히로시, 한빛 미디어

[22]

- 프로그래밍 언어론, 로버트 W. 세베스타

[23], [24]

- <https://itmining.tistory.com/72>

[25]

- 모던 자바스크립트 입문, 이소히로시, 한빛 미디어

[26]

- <https://developer.mozilla.org/ko/docs/Web/JavaScript/Guide/Closures>

[27]

- https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Global_Objects/Error