

05

Javascript

PNU Mini Bootcamp – Front End

자바스크립트 소개

자바스크립트 역사

- 1995년 Brendan Eich(브렌던 아이크 – 자바스크립트 창시자)에 의해 처음 개발
- 초기에는 LiveScript라고 불림
- 1996년 네스케이프 커뮤니케이션즈에서 LiveScript를 자바스크립트(Javascript)로 이름을 변경하면서 네스케이프 브라우저의 표준 스크립트 언어로 채택
- 자바보다 조금 늦게 나온 자바스크립트를 만들 때 자바의 문법은 약간 도용해서 만들었고 그리고 자바의 인기를 이용하기 위한 홍보전략 차원에서 이름에 “자바”가 추가되어 자바스크립트라고 명명

자바스크립트 소개

자바스크립트는 대단한 애플리케이션을 만들 목적이 아니었다.

- 브라우저에서 실행되는 HTML 문서의 화면 구성은 HTML/CSS 조합으로 하고 그곳에 자바스크립트를 추가해서 언어적인 기능이 가능하게 하겠다는 의도이며 이런 목적으로 넷스케이프 브라우저에 자바스크립트가 추가된 것
- 브라우저에서 간단한 언어적인 처리만을 목적으로 해서 자바스크립트가 이용
- 이런 탄생 배경이 있다보니 자바스크립트는 다른 소프트웨어 언어와 비교해 보면 문법이 약간 유연

프런트 웹 애플리케이션이 인기를 끌면서 덩달아서 인기를 얻게 된 언어

- 브라우저에서 실행되는 웹이 애플리케이션 급으로 개발되기를 원하다 보니 그 애플리케이션을 개발할 수 있는 소프트웨어 언어가 있어야 하고 그러다 보니 자바스크립트로 작성하는 코드의 규모 및 개발하고자 하는 내용이 다양화 되었으며 이로인해 자바스크립트의 인기가 올라가게 된 것

자바스크립트 소개

Node.js 때문에 더욱 인기있는 언어가 되었다.

- Node.js 가 나오기 시작하면서 더 이상 자바스크립트가 브라우저만을 위한 소프트웨어가 아닌 브라우저 밖에서 실행되는 애플리케이션을 개발하기 위한 언어로 사용이 가능해 졌습니다.
- 그로인해 백엔드 웹 애플리케이션이 자바스크립트로 개발되기 시작

자바스크립트는 ECMA 단체에서 표준을 정의하고 있다.

- 넷스케이프 브라우저 이외 다양한 브라우저에서 채택
- 브라우저별로 자바스크립트를 독자적으로 발전을 시키다 보니 자바스크립트 호환성 문제가 발생
- ECMA 인터내셔널(<https://ecma-international.org/>) 이라는 비영리 단체에서 자바스크립트 표준을 책정

자바스크립트 소개

자바스크립트 적용 분야

Front-End Web Application

- HTML
- CSS
- JAVASCRIPT

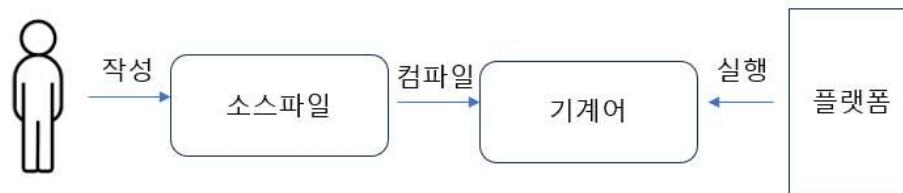
Back-End Web Application

- Java
- .NET
- PHP
- Python
- JAVASCRIPT(Node.js + Express)

자바스크립트 소개

자바스크립트는 스크립트 언어이다.

- 소프트웨어 언어는 컴파일 언어와 스크립트 언어로 구분
- 컴파일 언어는 소스 파일을 플랫폼에서 해석 가능한 형태로 변형시켜 실행
- 자바, C 등



- 스크립트 언어는 소스 파일을 직접 해석해 실행
- 자바스크립트



자바스크립트 소개

자바스크립트 실행

- 브라우저에 내장되어 있는 자바스크립트 엔진에 의해 실행
- Node.js 에 의해 실행
- 자바스크립트 문법 및 언의의 API 는 동일
- 브라우저 내장 객체 vs Node.js 내장 객체

자바스크립트 소개

자바스크립트 대체 언어

- 자바스크립트의 탄생 목적이 브라우저에서 실행되는 HTML 문서내에서 간단한 연산등에 사용하는 언어로 개발
- 2010년대로 넘어오면서 프런트엔드 웹 애플리케이션이 개발되기 시작하고, 백엔드 웹 애플리케이션이 자바스크립트로 개발되기 시작하면서 인기를 끌기 시작한 소스코드언어 언어
- 자바스크립트로 애플리케이션을 개발하기 시작하면서 자바스크립트를 대체할 수 있는 언어 등장
- Typescript, CoffeeScript, Kotlin, Swift, Dart 등
- 타입스크립트는 확장가자 ts
- 타입스크립트로 만든 소스는 브라우저의 자바스크립트 엔진이나 노드에 의해 실행 안됨
- 타입스크립트로 작성한 소스가 실행되기 위해서는 자바스크립트로 변형작업 필요
- 이를 transpiler 혹은 compiler 라고 부른다.

자바스크립트 소개

ECMA2015

- 자바스크립트에 대한 표준은 ECMA 단체에서 정의
- 자바스크립트를 또 다른 용어로 ECMAScript 라고도 합니다.
- ECMAScript 의 표준이 1 버전부터 5버전까지 나왔는데 이 5버전이 나온 시점이 90년대 후반
- ECMA 6 버전이며 줄여서 ES6
- ES6를 발표하면서 자바스크립트도 정식의 소프트웨어 언어처럼 많은 애플리케이션을 개발하기 위한 기법을 제공
- 그 이후 계속 버전을 변경하면서 ECMA 7, 8 버전이 발표되었는데 이름을 새로운 버전이 발표된 년도를 추가 해서 ECMA2015, ECMA2016, ECMA2017, ECMA2018, ECMA2019 이런식으로 부르기 시작
- ECMA6 와 ECMA2015는 동일한 버전을 이야기하며 줄여서 ES6 혹은 ES2015 라고 부릅니다.
- ES2015가 나오면서 거의 매년 새로운 버전을 발표하고 있는데 개발자들이 흔히 각각의 세부 버전을 명시하지 않고 ES2015 이후에 나온 버전을 통칭해서 ES2015 라고 부르고 있습니다.

자바스크립트 기초 작성법

자바스크립트 코드 위치 - HTML 문서 내에 작성

- 브라우저에서 자바스크립트를 실행시키려면 HTML 파일 필수
- HTML 을 파싱하면서 HTML 의 자바스크립트 코드를 브라우저의 자바스크립트 엔진이 실행
- 자바스크립트를 HTML 에 추가하려면 <script> 태그를 이용
- <script> 태그 사이에 추가된 자바스크립트는 브라우저의 자바스크립트 엔진에 의해 실행
- <script> 태그는 HTML 문서 어디에다 작성해도 상관없지만 일반적으로 <head> 태그 사이에 추가하거나 <body> 태그 사이에 추가
- <script type="application/javascript">

```
<body>
  <h1 id="title">Hello JavaScript</h1>
  <button onclick="myButtonClick()">click</button>

  <script>
    const myButtonClick = () => {
      let title = document.getElementById('title')
      if(title.style.backgroundColor === 'yellow'){
        title.style.backgroundColor = 'pink'
      }else{
        title.style.backgroundColor = 'yellow'
      }
    }
  </script>
</body>
```

자바스크립트 코드 추가

자바스크립트 기초 작성법

자바스크립트 코드 위치 - HTML 문서 외부에 작성

- 확장자가 JS 인 파일
- 자바스크립트 파일을 HTML에서 <script> 태그로 이용
- <script src="test.js"></script>

자바스크립트 기초 작성법

자바스크립트 코드 위치 - 브라우저의 HTML 실행 순서

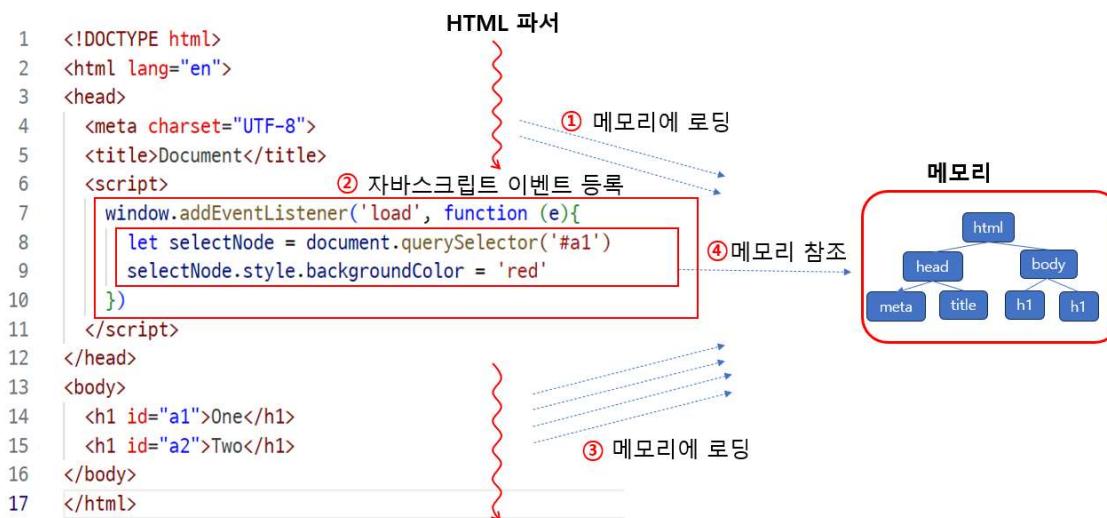
- 자바스크립트 코드에서 DOM 노드를 이용한다면 이용하는 DOM 노드가 메모리에 로딩된 이후에 해당 자바스크립트 코드가 실행되게 해주어야 합니다.



자바스크립트 기초 작성법

자바스크립트 코드 위치 - 브라우저의 HTML 실행 순서

- 자바스크립트의 코드 위치를 이용하고자 하는 DOM 노드의 태그 아랫부분에 작성
- 해도 되지만 또 다른 방법
- 문서의 모든 태그를 메모리에 로딩하게 되면 로딩 완료 이벤트를 발생시키며, 이 이벤트를 이용해 로딩 완료 후 DOM 노드를 이용되게 처리

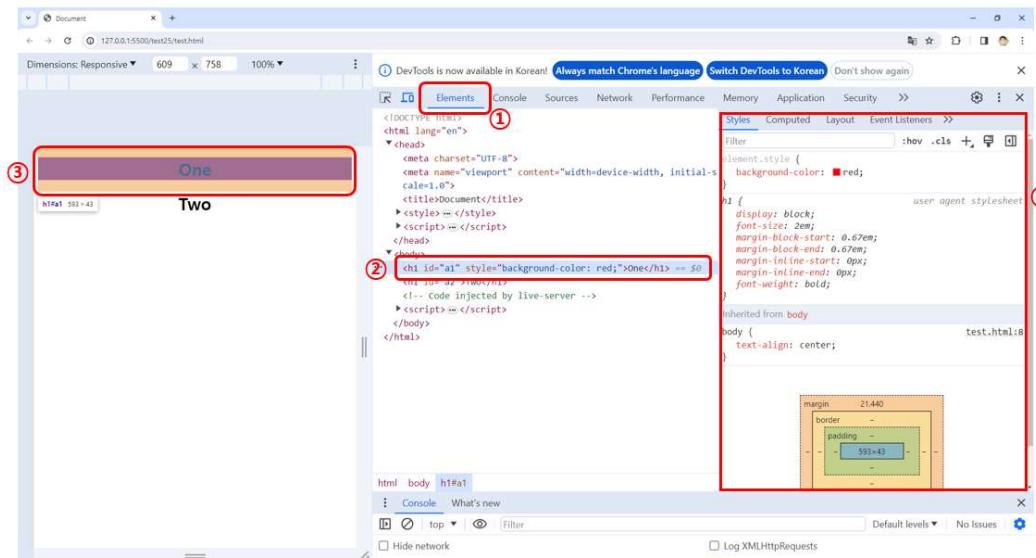


자바스크립트 기초 작성법

크롬 브라우저 개발자 도구 이용한 디버깅

Element

- 개발자 도구의 'Element' 탭에서 브라우저에 출력되는 HTML 문서의 태그들과 태그에 적용된 CSS 스타일을 살펴 볼 수 있습니다.

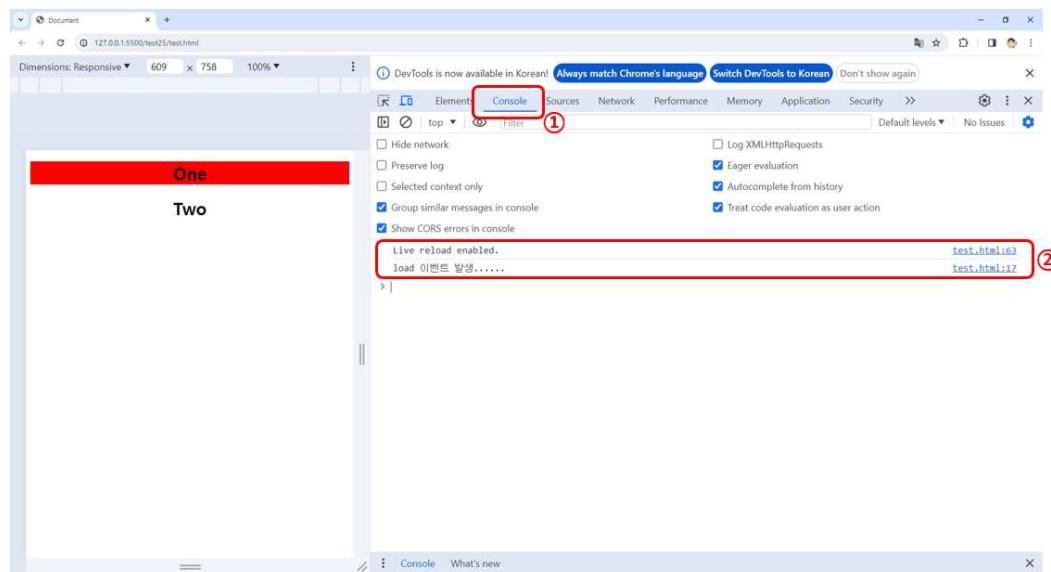


자바스크립트 기초 작성법

크롬 브라우저 개발자 도구 이용한 디버깅

Console

- 자바스크립트 코드에서 console.log() 로 출력시킨 내용
- 자바스크립트 코드가 실행되다가 발생한 에러 로그

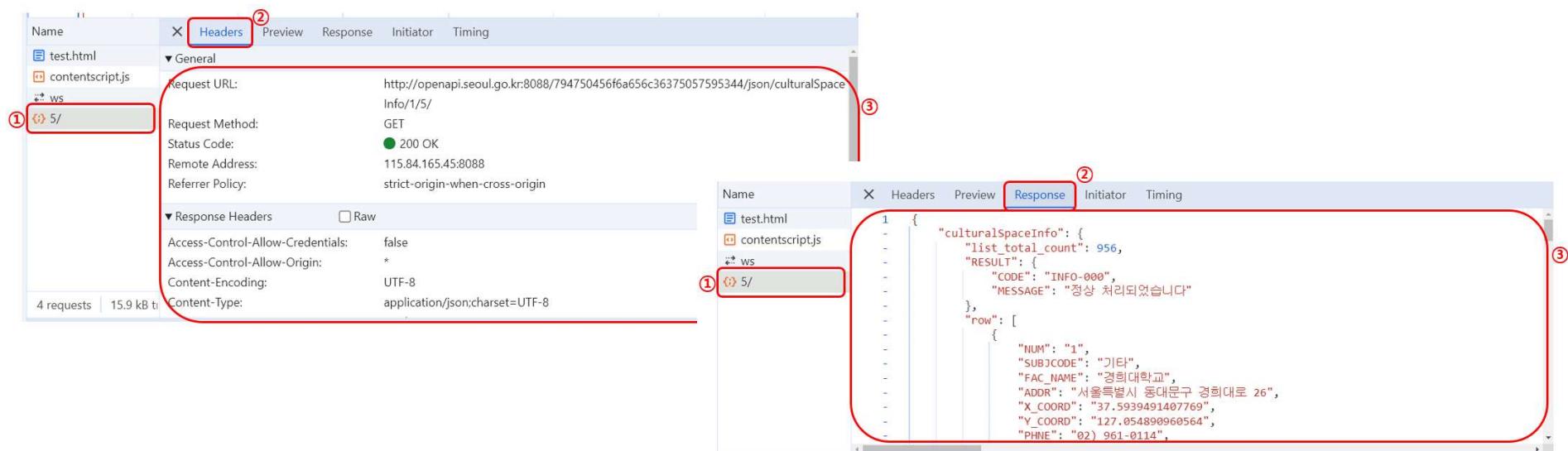


자바스크립트 기초 작성법

크롬 브라우저 개발자 도구 이용한 디버깅

Network

- Network 탭을 클릭하면 HTML 문서를 브라우저에 출력하기 위해서 발생한 네트워킹 이력을 볼 수 있습니다.

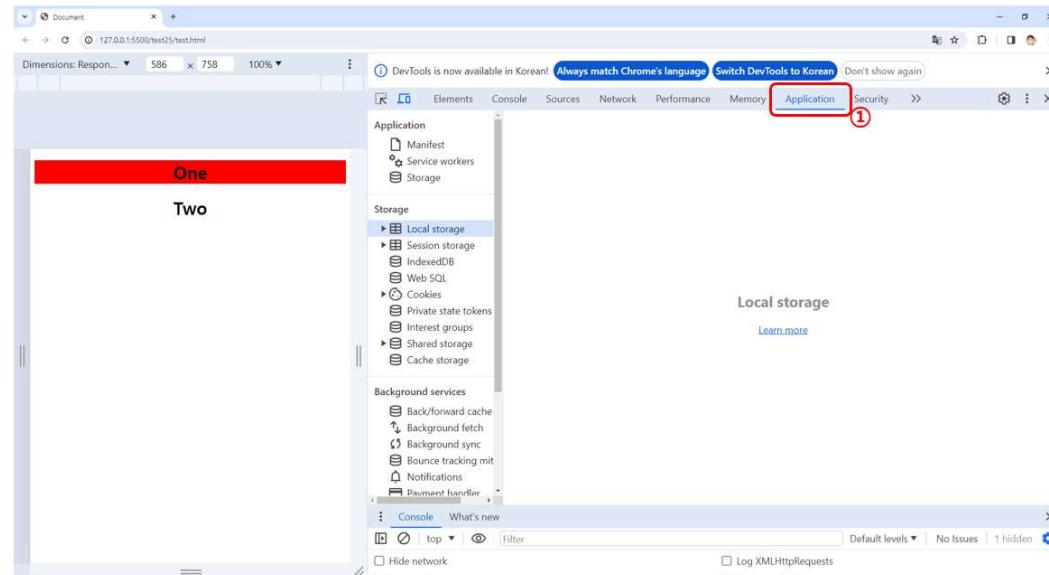


자바스크립트 기초 작성법

크롬 브라우저 개발자 도구 이용한 디버깅

Application

- HTML 문서를 실행시키면서 애플리케이션 적인 다양한 정보를 제공



자바스크립트 기초 작성법

주석

- HTML 주석은 <!-- -->
- CSS 주석은 /* */
- 자바스크립트 주석은 한 주석은 // 이며 여러 줄 주석은 /* */

자바스크립트 기초 작성법

자바스크립트는 대소문자를 구분한다.

```
<script>
    let data = "hello";
    if (data === "Hello") {
        console.log("data 는 Hello 입니다.");
    } else {
        console.log("data 는 Hello 가 아닙니다.");
    }
</script>
```

라인을 구분하기 위한 세미콜론을 강제하지 않는다.

```
let data = 10
let result = data * 10
console.log(result)
```

자바스크립트 기초 작성법

문자열을 묶을 때는 싱글쿼트 더블쿼트 모두 가능하다.

- 자바스크립트는 문자와 문자열을 구분하지 않습니다
- 문자열을 표현할 때 싱글쿼트를 사용해도 되고 더블쿼트를 사용해도 됩니다.

```
let name1 = '홍길동'  
let name2 = "홍길동"
```

자바스크립트 기초 작성법

window 함수 – alert prompt confirm

- 브라우저에서 실행되는 자바스크립트 코드에서 기본이 되는 객체는 window
- window 는 브라우저 자체를 지칭하는 객체
- window 에서 제공하는 기초 함수 중 다이얼로그를 브라우저 창에 띄워 유저에게 어떤 메시지를 출력하거나 유저 입력을 받을 수 있다.

| window 의 다이얼로그 함수 | | | |
|------------------------|------------|-------|-------------------------|
| 함수명 | 용도 | 버튼 | 결과 값 |
| <code>alert()</code> | 간단한 메시지 출력 | 확인 | <code>undefined</code> |
| <code>confirm()</code> | 확인/취소 선택 | 확인/취소 | <code>true/false</code> |
| <code>prompt()</code> | 사용자 글 입력 | 확인/취소 | 입력값/ <code>null</code> |

자바스크립트 기초 작성법

document.write()

- document 객체는 브라우저에서 실행되는 HTML 문서 자체를 지칭
- document.write() 함수는 HTML 문서에 어떤 문자열을 출력하기 위해서 사용

```
var name = prompt('이름을 입력해 주세요')
document.write('안녕하세요 ' + name + '님, 환영합니다.')
```

console.log()

- 결과 혹은 문자열을 출력하기 위한 구문인데 출력하는 위치가 자바스크립트 코드를 실행시켜주는 플랫폼의 콘솔

```
console.log('첫번째 출력입니다. - 안녕하세요')
console.log('두번째 출력입니다.', "홍길동", "임꺽정")
console.log('세번째 출력입니다.', 10, true)
```

자바스크립트 기초 작성법

strict mode

- 코드 첫 줄 혹은 함수 내용 첫 줄에 'use strict' 선언

```
"use strict";  
//자바스크립트 구문 작성..
```

- 엄격모드가 선언되지 않은 경우를 용어상으로 느슨한 모드(sloppy mode)
- 자바스크립트는 다른 소프트웨어 언어와 비교해 느슨한 문법을 지원

```
let data1 = 10  
data2 = 20  
console.log(data1, data2)//10, 20
```

자바스크립트 기초 작성법

strict mode

- 자바스크립트의 느슨한 모드를 엄격하게 문법을 적용시켜 실행시켜 주기를 원할 수도 있으며 이를 위해서 엄격모드를 지원

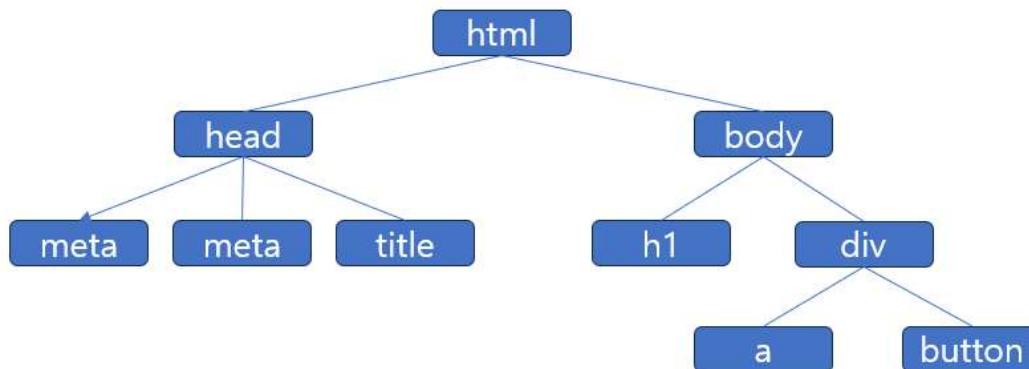
```
"use strict";
```

```
let data1 = 10  
data2 = 20  
console.log(data1, data2) //Uncaught ReferenceError: data2 is not defined
```

자바스크립트 기초 작성법

DOM Node

- HTML 문서내의 태그를 자바스크립트에서는 흔히 DOM 노드(Node) 라고 부릅니다.
- HTML 문서 내에 있는 하나의 태그로 감싸져 있는 부분을 노드
- 자바스크립트에서 HTML 문서를 흔히 DOM 이라고 지칭
- Document Object Model 의 약어



자바스크립트 기초 작성법

Node Selector

- `querySelector()` : 조건에 맞는 첫 노드를 반환
- `querySelectorAll()` : 조건에 맞는 모든 노드를 배열로 반환
- `document.querySelector()` 를 사용하면 전체 HTML 문서내에서 원하는 노드를 획득
- `element.querySelector()` 를 이용하면 특정 노드 하위에 있는 노드들 중 조건에 맞는 노드를 획득
- CSS Selector 이용
- 단어만 사용하면 태그명, 단어앞에 #을 추가하면 id 값, 단어 앞에 .(dot)을 추가하면 css 클래스명을 지칭

```
let parentNode = document.querySelector('#a2')
let selectNode = parentNode.querySelectorAll('div')
```

Basic Syntax – 변수

변수 선언

- 자바스크립트에서 변수를 선언하기 위해서는 let, var, const이라는 예약어를 이용



네이밍 규칙

- 식별자는 문자, 숫자, 언더스코어(_), 달러기호(\$)의 조합으로 작성한다.
- 식별자의 첫글자는 문자, 언더스코어(_), 달러기호(\$)로 시작할 수 있다.
- 식별자에는 공백이 추가될 수 없다.
- 자바스크립트의 예약어는 식별자로 사용할 수 없다.

```
let 1a = 10//error
let if = 20//error
let add user = 30//error
```

Basic Syntax – 변수

네이밍 규칙 – 코드 컨벤션

- 두 단어가 연결되어 식별자로 사용하는 경우 camel case 로 작성한다.
- 식별자는 의미있는 단어를 이용한다.
- 변수와 함수는 소문자로 시작하고 클래스명은 대문자로 시작한다.
- 상수 변수의 경우 전체 식별자를 대문자로 작성한다.

Basic Syntax – 변수

변수 - var

- var 예약어로 변수를 선언하는 방법은 자바스크립트 초기부터 제공되던 오래된 방법

```
var productName = "에어조던"  
  _____  
  예약어   식별자   연산자   데이터
```

Basic Syntax – 변수

상수 변수 선언

- const로 선언된 변수는 선언과 동시에 초기값을 대입해 주어야 합니다.
- 이후 값 변경이 불가능합니다.

```
1 //변수 선언
2 let data1 = 10
3 const data2 = 10
4
5 //변수 값 변경
6 data1 = 20
7 data2 = 20 //error 발생, Uncaught TypeError: Assignment to constant variable.
```

Basic Syntax – 변수

변수와 호이스팅

- 호이스팅(Hoisting)은 ‘무언가를 끌어올린다’라는 의미의 단어
- 아래에 선언된 변수를 위에서 사용하게 해주는 기법



- 선언된 변수가 호이스팅이 되면 어디선가 값 대입이 되기 전까지는 undefined 상태

```
console.log(`step1, data = ${data}`)//step1, data = undefined  
data = 20  
console.log(`step2, data = ${data}`)//step2, data = 20  
var data = 10;  
console.log(`step3, data = ${data}`)//step3, data = 10
```

Basic Syntax – 변수

변수와 호이스팅

- 호이스팅은 var로 선언된 변수에 대해서만 지원
- let, const로 선언된 변수는 호이스팅이 지원되지 않습니다.

```
console.log(data1)//error.. Cannot access 'data1'  
before initialization
```

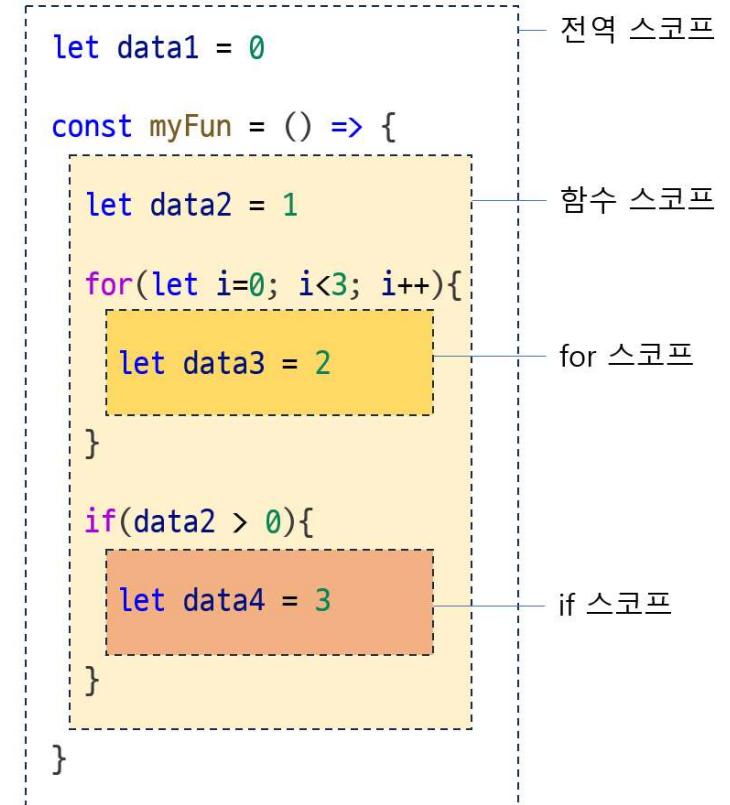
```
console.log(data2)//error.. Cannot access 'data2'  
before initialization
```

```
let data1 = 10  
const data2 = 20
```

Basic Syntax – 변수

스코프

- 코드가 실행되는 영역을 스코프라고 하는데 같은 영역에서 실행되는 코드를 {}로 묶어서 개발
- 하나의 {} 내에 선언된 코드들은 동일 스코프에서 실행된다는 표현을 합니다.
- 대표적인 것이 함수, for, if



Basic Syntax – 변수

중복선언

- 동일 스코프내에서 중복인지? 다른 스코프에서 중복 선언인지에 따라 다릅니다.
- 다른 스코프에 동일 이름으로 변수가 중복되었다고 하더라도 두 변수는 개별 변수가 되며 상호 영향을 미치지 않게 됩니다.

```
let data = '홍길동'

const myFun = () => {
  let data = '김길동'
  console.log(`in myFun, data = ${data}`)
}
myFun()
console.log(`out myFun, data = ${data}`)

//in myFun, data = 김길동
//out myFun, data = 홍길동
```

Basic Syntax – 변수

중복선언

- 동일 스코프내에서 이미 선언된 변수명과 동일한 변수를 다시 중복으로 선언하는 것은 var로 선언된 변수는 가능
- let, const로 선언된 변수는 불가능

```
var data1 = 10  
let data2 = 10  
const data3 = 10
```

```
var data1 = '홍길동'  
let data2 = '홍길동'//error  
let data3 = '홍길동'//error
```

Basic Syntax – 변수

변수 - 3가지 방법 정리 비교

| | 초기화 | 중복 선언 | 스코프 | 값 변경 |
|--------------|---------------|-------|--------|------|
| var | 선언과 초기화 분리 가능 | 가능 | 함수 스코프 | 가능 |
| let | 선언과 초기화 분리 가능 | 불가 | 블록 스코프 | 가능 |
| const | 선언과 동시에 초기화 | 불가 | 블록 스코프 | 불가 |

Basic Syntax – 데이터 타입

데이터 타입 – 타입 유추

- 모든 소프트웨어 언어에서는 데이터를 타입으로 구분

`10 + 10` → ?

`"hello" + "world"` → ?

- 자바스크립트에서는 개발자 코드에 타입을 명시하지는 않습니다.
- 자바 코드(Strong typed Language)

```
int no = 10;  
String name = "홍길동";
```

- 자바스크립트 코드(Weakly typed Language)

```
let no = 10  
let name = "홍길동"
```

Basic Syntax – 데이터 타입

데이터 타입 – 타입 유추

- let이라는 예약어는 변수를 선언하기 위한 예약어지, 타입이 아닙니다.
- 데이터 타입을 위한 예약어도 없습니다.

```
let data = 10  
data = "홍길동"  
data = true
```

- 데이터 타입이 없는 것이 아니라 대입되는 값을 보고 타입이 유추

```
let data = 10 _____ 숫자타입으로 유추  
data = "홍길동" _____ 문자타입으로 유추  
data = true _____ 논리타입으로 유추
```

Basic Syntax – 데이터 타입

데이터 타입 – 자바스크립트의 데이터 타입들

- 자바스크립트에서 데이터 타입은 숫자타입, 문자타입, 논리타입 그리고 객체타입

숫자 타입

- 타입 유추 기법을 이용해 개발자 코드에 타입이 명시되지 않음으로 정수, 실수를 통칭해서 숫자타입으로 판단

```
let data1 = 10
let data2 = 10.0
console.log(`data1 type is ${typeof data1}`)//data1 type is number
console.log(`data2 type is ${typeof data2}`)//data2 type is number
```

Basic Syntax – 데이터 타입

문자타입

- 문자 타입은 문자열 혹은 문자 하나를 표현하기 위한 타입
- 문자 타입의 데이터는 싱글쿼터(') 혹은 더블쿼터(")로 묶여야 합니다.

```
let data1 = 'a'  
let data2 = 'hello'  
let data3 = "hello"  
console.log(`data1 type is ${typeof data1}`)//data1 type is string  
console.log(`data2 type is ${typeof data2}`)//data2 type is string  
console.log(`data3 type is ${typeof data3}`)//data2 type is string
```

- 숫자 데이터도 싱글쿼터 혹은 더블쿼터로 묶으면 문자타입으로 인지

```
let data1 = 10  
let data2 = 20  
let data3 = '10'  
let data4 = '20'  
console.log(data1 + data2)//30  
console.log(data3 + data4)//1020
```

Basic Syntax – 데이터 타입

문자타입 – Template String

- Template string 은 Template literal 이라고도 하며 문자열 데이터를 만드는 방법을 의미
- 백틱(backtick) 으로 문자열 표현
- 동적인 결과가 추가된 문자열을 만들기 위해서 이용

```
console.log(`안녕하세요 ${name}님, 함수 호출 결과는 ${calSum(10)}이고 연산 결과  
는 ${10 + 20}입니다.`)
```

Basic Syntax – 데이터 타입

논리타입

- 논리타입은 참/거짓 데이터를 가지는 타입입니다. 데이터로 true/ false 를 의미

```
let data1 = true
let data2 = 10 < 5
console.log(`data1 type is ${typeof data1}`)//data1 type is boolean
console.log(`data2 : ${data2}`)//data2 : false
```

Basic Syntax – 데이터 타입

null

- null 은 객체에 대입되는 값
- 객체가 선언되기는 했는데 아직 값이 없는 상태를 표현하는 값

```
let data1 = null
console.log(data1, `data1 type is ${typeof data1}`)//null 'data1 type is object'
```

undefined

- undefined 는 데이터 타입
- undefined 는 변수가 선언되기는 했지만 값이 대입되지 않아서 데이터 타입을 유추할 수 없는 상태의 타입을 의미

```
let data1 = undefined
console.log(data1, `data1 type is ${typeof data1}`)//undefined 'data1 type is
undefined'
```

Basic Syntax – 데이터 타입

컬렉션 타입 – 배열

- 배열을 한마디로 정의하자면 여러 개의 데이터를 표현하는 타입
- 배열 변수에 들어가는 데이터를 명시할때는 대괄호([])를 이용
- [] 안에 콤마(,)로 데이터를 구분해서 여러건의 데이터를 나열

```
let users = ['홍길동', '김길동', '이길동']
```

- 데이터는 순서에 의해 이용

```
console.log(users[0], users[1], users[2])//홍길동 김길동 이길동
```

Basic Syntax – 데이터 타입

컬렉션 타입 – 배열

- 배열의 데이터 개수를 판단할 때는 배열의 length 변수

```
console.log(users.length)//3
```

- 데이터를 변경하고 싶다면 인덱스로 그 데이터의 위치를 지정하고 그곳에 변경하고자 하는 데이터를 대입

```
let users = ['홍길동', '김길동', '이길동']
console.log(users)//['홍길동', '김길동', '이길동']
//데이터 변경
users[1] = '박길동'
console.log(users)//['홍길동', '박길동', '이길동']
```

Basic Syntax – 데이터 타입

컬렉션 타입 – 배열

- 기존 배열에 신규 데이터를 추가하고 싶다면 push()라는 함수를 이용
- 배열에 담긴 데이터를 제거하고 싶다면 pop() 함수를 이용

```
let users = ['홍길동', '김길동', '이길동']
console.log(users.length, users)//3 ['홍길동', '김길동', '이길동']
users.push('박길동')
users.push('정길동')
console.log(users.length, users)//5 ['홍길동', '김길동', '이길동', '박길동', '정길동']
users.pop()
console.log(users.length, users)//4 ['홍길동', '김길동', '이길동', '박길동']
```

Basic Syntax – 연산자

산술 연산자

| 연산자 | 설명 |
|-----|---------------------------|
| + | 이항 연산자, 더하기 연산 |
| - | 이항 연산자, 빼기 연산 |
| * | 이항 연산자, 곱하기 연산 |
| / | 이항 연산자, 나누기 연산 |
| % | 이항 연산자, 두 수를 나누어 나머지 값 반환 |
| ++ | 단항 연산자, 데이터에 1 값을 더한다. |
| -- | 단항 연산자, 데이터에 1 값을 뺀다. |
| ** | 이항 연산자, 거듭제곱 |

Basic Syntax – 연산자

할당 연산자

| 연산자 | 설명 |
|-----|------------|
| = | 할당 연산자 |
| += | 더하기 할당 연산자 |
| -+ | 빼기 할당 연산자 |
| *= | 곱하기 할당 연산자 |
| /= | 나누기 할당 연산자 |
| %= | 나머지 할당 연산자 |

Basic Syntax – 연산자

비교 연산자

| 연산자 | 설명 |
|--------------------|------------------------------|
| <code>==</code> | 값이 같으면 true |
| <code>!=</code> | 값이 다르면 true |
| <code>====</code> | 값과 타입이 같으면 true |
| <code>!==</code> | 값 혹은 타입이 다르면 true |
| <code>></code> | 왼쪽의 데이터가 오른쪽 보다 크면 true |
| <code>>=</code> | 왼쪽의 데이터가 오른쪽 보다 크거나 같으면 true |
| <code><</code> | 왼쪽의 데이터가 오른쪽 보다 작으면 true |
| <code><=</code> | 왼쪽의 데이터가 오른쪽 보다 작거나 같으면 true |

Basic Syntax – 연산자

논리 연산자

| 연산자 | 설명 |
|-----|---|
| && | 논리 AND 연산자, 앞 뒤 데이터가 모두 true 인 경우 true |
| | 논리 OR 연산자, 앞 뒤 데이터 중 하나 이상의 데이터가 true 이면 true |
| ! | 논리 NOT 연산자, 단항 연산자, true 를 false 로, false 를 true 로 변경 |

Basic Syntax – 제어문

if

- 조건문은 if 문과 switch문 그리고 3항 연산자가 있습니다.
- if 문은 어떤 조건에 만족하는 경우에 특정 코드를 실행
- 경우에는 else 예약어를 이용해 조건에 만족하지 않은 경우에 실행할 코드를 명시
- 조건을 여러 번 주어야 하는데 이때는 else() 를 사용

```
if( 조건 ) {  
    조건에 만족하는 경우 실행되야 하는 구문  
}
```

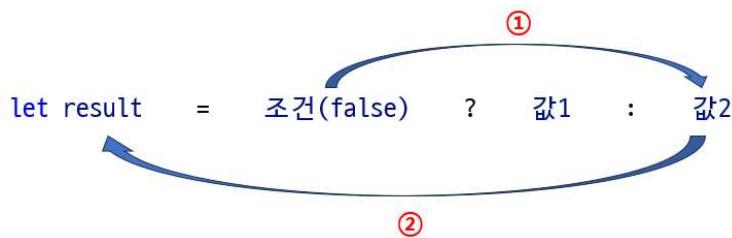
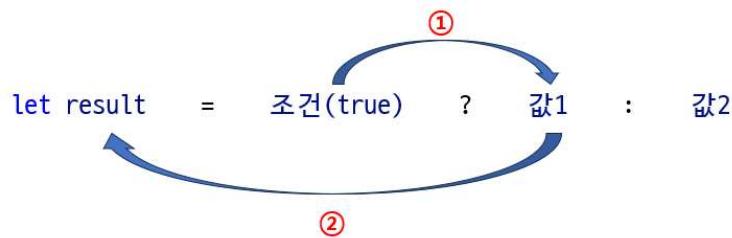
```
if( 조건 ){  
    조건에 만족하는 경우 실행되야 하는 구문  
}else {  
    조건에 만족하지 않는 경우 실행되야 하는 구문  
}
```

```
if( 조건 ) {  
  
}else if( 조건 ) {  
  
}else if( 조건 ) {  
  
}else {  
  
}
```

Basic Syntax – 제어문

3항 연산자

- 3항 연산자도 결과가 나오지만 조건을 명시할 수 있고 그 조건에 따라 다르게 결과가 나오게 하는 연산자



Basic Syntax – 제어문

switch

- switch – case 문은 어떤 데이터의 값이 여러 개가 나올 수 있는데 그 값이 어떤 것인지에 따라 실행되는 구문을 다르게 조건을 주고자 할 때 사용
- 데이터의 값에 대한 조건은 case 예약어로 명시
- default 는 생략이 가능하며 작성한다면 case 가 나열되고 맨 마지막에 작성
- default 는 위에 선언된 case 값에 만족하지 않는 경우 실행될 구문을 명시하기 위해서 사용

```
switch(데이터){  
    case 값1: {  
        실행 구문  
    }  
    case 값2: {  
        실행 구문  
    }  
    default: {  
        실행 구문  
    }  
}
```

Basic Syntax – 제어문

switch

- case 값 조건에 만족하게 되면 그 위치부터 아래에 선언된 모든 case 와 default 부분이 실행

```
switch(data % 3){ → 0
  case 0: { ←
    console.log('나머지는 0입니다.')
  }
  case 1: {
    console.log('나머지는 1입니다.')
  }
  default: {
    console.log('default 부분이 실행되었습니다.')
  }
}
```

- 특정 위치의 case 값에 일치하는 경우 그 위치의 case 만 실행되게 하고자 한다면 break 를 사용

Basic Syntax – 제어문

조건문 – 다양한 타입의 데이터로 참 거짓 판단

- 논리타입의 데이터가 이용되는 곳에 true, false 이외에 1, 'hello' 등의 데이터가 사용 가능

```
let data = 1
if(data){
  console.log('data is true')
}else {
  console.log('data is false')
}
//data is true
```

- 자바스크립트에 Boolean이라는 객체로 다양한 타입의 데이터를 논리타입으로 판단 가능

```
console.log(Boolean('hello'))//true
console.log(Boolean(1))//true
console.log(Boolean(0))//false
console.log(Boolean(-1))//true
console.log(Boolean(null))//false
console.log(Boolean(undefined))//false
```

Basic Syntax – 제어문

for

- 반복문은 for, while 문이 주로 사용되며 드물게 do – while 문이 사용
- for 문을 작성하는 방법은 특정 변수 값을 증감시켜 조건에 만족하는 동안 반복적으로 실행

```
①      ②      ③  
for( 초기값 ; 반복조건 ; 증감식 ){  
④body, 반복적으로 실행될 구문  
}
```

Basic Syntax – 제어문

while

- while 문은 증감조건만 명시, 조건이 참인 경우 계속 실행

```
while( 반복조건 ) {  
    body - 반복적으로 실행되는 구문  
}
```

do – while

- while() {} 은 while문 앞 부분에 조건을 명시하지만 do – while 문은 do {} while() 형태로 작성해서 do – while 문 마지막 부분에 조건을 명시
- do – while 은 body 부분이 최소 1번은 실행

```
do {  
    body - 반복적으로 실행되는 구문  
} while( 반복조건 )
```

Basic Syntax – 제어문

break, continue

- break 와 continue 는 반복문 내에 작성되어 반복문의 실행 흐름을 제어
- break 는 switch – case 문에도 사용되며 switch 문에 break 를 사용하면 switch 를 벗어나게 되어 제어하는 역할로도 사용
- continue 를 만나게 되면 continue 아랫 부분은 실행되지 않으며 다시 반복 조건을 판단
- break 문은 break 를 만나게 되면 반복문을 끝내게 됩니다.

```
for(let i=0; i<10; i++){
    if(i % 2 === 0){
        continue
    }
    console.log(`for body : ${i}`)
    if(i === 7){
        break
    }
}
```

Basic Syntax – 제어문

break, continue

- 중첩 구조의 경우 가장 가까운 loop 문을 벗어나거나 다시 조건 판단
- 특정 위치의 반복문이 제어되게 하고 싶다면 라벨을 이용
- 라벨이란 개발자가 지정하는 임의 식별자

```
for(let no1=0; no1<2; no1++){
    console.log(`position 1 : ${no1}`)
    for(let no2=0; no2<2; no2++){
        console.log(`position 2 : ${no1}, ${no2}`)
        break
    }
}
```

```
myloop : for(let no1=0; no1<2; no1++){
    console.log(`position 1 : ${no1}`)
    for(let no2=0; no2<2; no2++){
        console.log(`position 2 : ${no1}, ${no2}`)
        break myloop
    }
}
```

Basic Syntax – 함수

선언과 이용

- 함수를 선언하기 위해서는 function이라는 예약어를 이용

함수 선언 {
function myFun(){
 console.log('hello world')
}

함수 호출 ————— myFun()

Basic Syntax – 함수

매개변수와 반환 값

- 인수(argument)는 함수를 호출하는 곳에서 그 함수에 전달되는 값
- 반환 값은 함수내용이 실행되고 함수를 호출한 곳에 전달되는 값
- 함수를 호출한 곳에 데이터를 반환하고 싶다면 return 예약어를 이용

```
function myFun(arg1, arg2){  
    ...  
    return 10  
}  
  
let result = myFun(10, 20)
```

The diagram illustrates the execution flow of the code. It shows a function definition 'myFun' with parameters 'arg1' and 'arg2'. Inside the function, there is a 'return' statement followed by the value '10'. A dashed oval encircles the value '10'. An arrow points from this oval up to the 'return' keyword. Another arrow points from the same oval down to the argument '10' in the line 'let result = myFun(10, 20)'. This visualizes how the function's return value becomes the value assigned to the variable 'result'.

Basic Syntax – 함수

default parameter

- 매개변수에 인수를 전달하지 않으면 undefined
- default parameter 가 선언이 되었다면 인수가 전달되지 않는 경우 그 매개변수는 선언하면서 지정한 값을 가지게 됩니다.

```
function myFun(arg1, arg2 = 0){  
    console.log(`arg1 : ${arg1}, arg2 : ${arg2}`)  
}  
myFun()//arg1 : undefined, arg2 : 0  
myFun(10)//arg1 : 10, arg2 : 0  
myFun(10, 20)//arg1 : 10, arg2 : 20
```

Basic Syntax – 함수

rest parameter

- Rest Parameter 란 나머지 매개변수라는 의미
- 외부에서 전달하는 데이터 개수가 너무 많거나 개수가 예측이 안되는 경우에 사용
- Rest parameter 는 ... 으로 선언되는 매개변수
- Rest Parameter 로 선언한 매개변수는 그 함수의 매개변수들 중 가장 마지막에 위치
- 함수 내부에서는 Rest Parameter 의 매개변수를 배열로 이용

```
function myFun(arg1, ...arg2){  
    console.log(arg2)  
    if(arg2.length > 0){  
        for(let i=0; i<arg2.length; i++){  
            console.log(`arg2[${i}] = ${arg2[i]}`)  
        }  
    }  
}
```

Basic Syntax – 함수

함수 표현식

- 함수 표현식이란 함수를 선언하는데 이 선언된 함수를 변수에 대입할 수 있다는 의미

```
//함수 선언문
function myFun1() {
    console.log('myFun1 call')
}

//함수 표현식
const myFun2 = function(){
    console.log('myFun2 call')
}

myFun1() //myFun1 call
myFun2() //myFun2 call
```

Basic Syntax – 함수

화살표 함수

- 화살표 함수(Arrow Function)은 다른 소프트웨어 언어에서는 람다함수(Lambda Function)이라고 불리우는 함수
- 화살표 함수는 함수를 선언하면서 function 예약어를 사용하지 않으며 함수명도 주지 않습니다.
- 매개변수를 선언하는 () 와 함수 내용을 담는 {} 사이를 화살표(=>)로 구분하여 선언

```
const myFun = (arg1) => {
    console.log(`myFun, arg1 : ${arg1}`)
    return 20
}
```

Basic Syntax – 함수

화살표 함수

- 만약 화살표 함수의 내용이 한줄이라면 함수 내용을 묶는 {} 을 생략할 수 있습니다.

```
const myFun = (arg1) => arg1 * 10
```

- => 왼쪽의 매개변수가 하나라면 () 를 생략할 수도 있습니다.

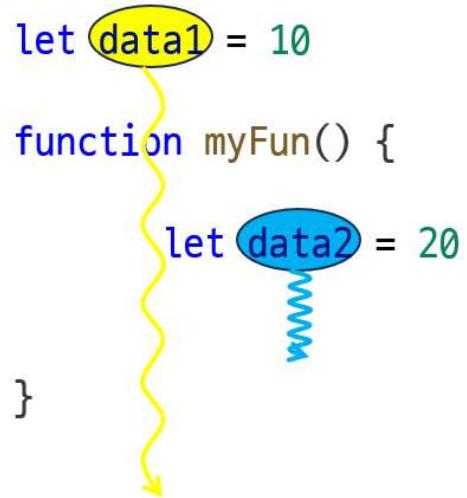
```
const myFun1 = (arg1) => arg1 * 10
```

```
const myFun2 = arg1 => arg1 * 10
```

Basic Syntax – 함수

지역변수와 전역변수

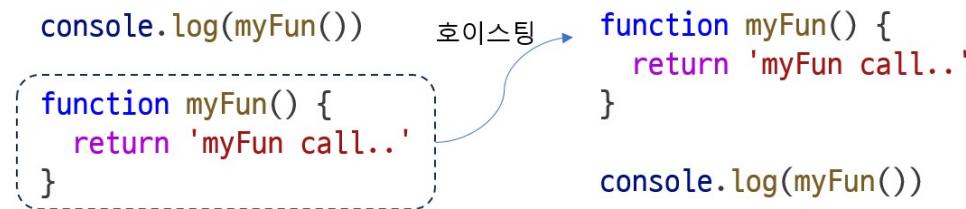
- 변수 선언이 함수 외부에 작성되어 있다면 이 변수는 전역변수(Global Variable)
- 어떤 함수 내부에 선언되어 있다면 이 변수는 지역변수(Local Variable)



Basic Syntax – 함수

함수 호이스팅

- 함수 호이스팅은 아랫부분에 선언된 함수를 코드 위 부분에서 호출할 수 있게 지원
- 함수의 호이스팅은 함수 선언식에서 제공되며 표현식 함수는 호이스팅이 지원되지 않습니다.



```
console.log(myFun());//error .. Cannot access 'myFun'
before initialization
const myFun = () => {
  return 'myFun call...'
}
```

Basic Syntax – 함수

익명함수

- 익명함수란 특별한 규칙이 있는 것이 아니라 함수를 선언하면서 함수명을 지정하지 않는 함수

```
//일반함수
function myFun1() {
}

//익명함수
const myFun2 = function(){
}

////익명함수 - 화살표 함수
const myFun3 = () => {

}
```

Basic Syntax – 함수

인수 및 반환 값으로 함수 이용

- 익명함수를 선언하는 대표적인 사례 중 하나가 함수를 어떤 다른 함수의 매개변수 혹은 반환값으로 이용하는 경우

```
function test1(){ console.log('test1') }

function test2(){ console.log('test2') }

function myFun1(arg) {
    arg()
    return test2
}

let resultFun = myFun1(test1)
resultFun()

//test1
//test2
```

Basic Syntax – 함수

인수 및 반환 값으로 함수 이용

- 인수 혹은 반환값으로 활용되는 함수들은 코드 간략화 차원에서 대부분 익명함수로 작성

```
function myFun1(arg) {  
    arg()  
    return ()=>{console.log('test2')}  
}  
  
let resultFun = myFun1(()=>{console.log('test1')})  
resultFun()  
  
// //test1  
// //test2
```

이벤트

이벤트 종류

- 프런트엔드 웹 애플리케이션에서 이벤트는 두가지로 구분
 - 브라우저 이벤트
 - 사용자 이벤트
- 사용자 이벤트는 이벤트 종류에 따라 3가지로 구분
 - 마우스 이벤트
 - 키 이벤트
 - HTML FORM 관련 이벤트

이벤트

이벤트 처리 구조

- 애플리케이션에서 이벤트 처리를 하기 위해서는 이벤트 소스와 이벤트 핸들러를 리스너로 연결



- 이벤트 소스 : 이벤트 발생 객체
- 이벤트 핸들러 : 이벤트 처리 내용
- 리스너 : 이벤트 소스와 이벤트 핸들러를 연결
- 이벤트 소스와 이벤트 핸들러를 이벤트 리스너로 연결, 실제 이벤트 소스에서 이벤트 발생되었을 때 리스너로 등록해 놓은 이벤트 핸들러가 실행

이벤트 명
↓
[window].addEventListener('load', () => {})
event source listener event handler

이벤트

이벤트 핸들러 등록 1 – addEventListener()

- 자바스크립트에서 이벤트를 등록하는 가장 기본적인 방법은 addEventListener()라는 함수를 이용

이벤트소스 . addEventListener(이벤트 명, 이벤트 핸들러)

```
//HTML 문서 로딩 완료 이벤트
window.addEventListener('load', () => {
    console.log('HTML 문서 로딩이 완료 되었습니다.')
})

//버튼 클릭 이벤트
let button = document.querySelector('#button1')
button.addEventListener('click', () => {
    console.log('button 이 클릭 되었습니다.')
})
```

이벤트

이벤트 핸들러 등록 2 – DOM 노드에서 이벤트 등록

- HTML 문서의 태그에서 직접 이벤트 핸들러를 등록

```
<button onclick="myEventHandler()">click me</button>
```

이벤트 핸들러 등록 3 – 자바스크립트에서 onXXX로 이벤트 등록

- addEventListener() 함수를 이용하지 않고 이벤트 소스에 이벤트를 onXXX로 등록 가능

```
window.onload = () => {
    console.log('HTML 문서 로딩이 완료 되었습니다.')
}

let button = document.querySelector('#button1')
button.onclick = () => {
    console.log('button 이 클릭 되었습니다.')
}
```

이벤트

마우스 이벤트

| 이벤트 종류 | 설명 |
|-------------------|----------------------|
| click | 마우스 클릭 이벤트 |
| dblclick | 마우스 더블 클릭 이벤트 |
| mousedown | 마우스 버튼을 누르는 순간의 이벤트 |
| mouseup | 마우스 버튼을 떼는 순간 이벤트 |
| mousemove | 마우스 이동 이벤트 |
| mouseenter | 마우스 포인터가 들어오는 순간 이벤트 |
| mouseleave | 마우스 포인터가 나가는 순간 이벤트 |
| mouseover | 마우스 포인터가 들어오는 순간 이벤트 |
| mouseout | 마우스 포인터가 나가는 순간 이벤트 |

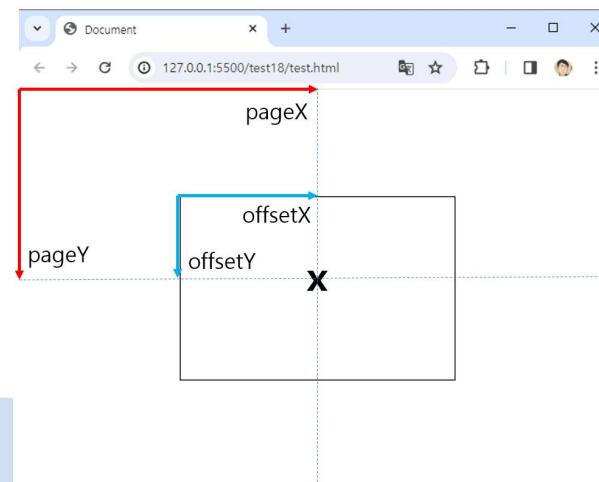
이벤트

마우스 이벤트

- 모든 마우스 이벤트들은 이벤트 핸들러에 MouseEvent 타입의 객체가 전달되며 이 객체의 정보를 이용해 이벤트가 발생한 지점의 좌표값을 얻을 수 있습니다.

```
area.addEventListener('mousemove', (e) => {
    console.log(`offset(${e.offsetX}, ${e.offsetY}), page(${e.pageX}, ${e.pageY})`)
})
```

- offsetX, offsetY는 이벤트가 발생한 DOM 노드 내에서의 좌표이며 pageX, pageY는 브라우저 창의 좌표



이벤트

마우스 이벤트

- DOM 노드에 마우스가 들어가거나 나가는 순간의 이벤트를 처리하기 위한 것이 mouseenter, mouseleave 도 있고 mouseover, mouseout 도 있습니다.
- mouseenter, mouseleave 는 이벤트가 발생한 DOM 노드에서만 이벤트 처리가 가능하며 버블링이 발생하지 않습니다
- mouseover, mouseout 은 버블링이 발생

이벤트

키 이벤트

| 이벤트 종류 | 설명 |
|----------------|----------------|
| keydown | 키를 누르는 순간의 이벤트 |
| keyup | 키를 떼는 순간의 이벤트 |

- 이벤트 핸들러 함수의 매개변수에 KeyboardEvent 타입의 객체가 전달되며 이 객체의 key 변수를 이용하여 이벤트가 발생한 키를 알아낼 수 있습니다.

```
textarea.addEventListener("keydown", (e) => {
    console.log(`key "${e.key}" down`);
});
```

이벤트

window 이벤트

- HTML 문서가 출력되는 브라우저 창을 지칭하는 객체는 window

| 이벤트 | 설명 |
|---------------|----------------------------|
| copy | 브라우저에서 복사했을 때의 이벤트 |
| cut | 브라우저에서 잘라내기 했을 때의 이벤트 |
| paste | 브라우저에서 붙여넣기 했을 때의 이벤트 |
| load | 브라우저에 문서 로딩이 완료 되었을 때의 이벤트 |
| error | 브라우저에서 문서 로딩에 실패했을 때의 이벤트 |
| resize | 브라우저 창의 크기가 변경될 때의 이벤트 |

```
window.addEventListener('load', () => {
  console.log('load event')
})
```

이벤트

window 이벤트

- window 객체는 브라우저 내에서 생략이 가능
- resize 이벤트는 브라우저 창의 크기가 변경되는 순간의 이벤트이며 이벤트 처리 함수에서 innerWidth, innerHeight로 브라우저 창의 크기를 획득

```
addEventListener('resize', () => {
  console.log(`width: ${innerWidth}, height:
${innerHeight}`)
})
```

이벤트

Form 관련 이벤트

| 이벤트 | 설명 |
|---------------|-----------------------------|
| submit | form 데이터가 submit 되는 순간의 이벤트 |
| reset | form 데이터가 reset 되는 순간의 이벤트 |

```
let form = document.querySelector('#form')
form.addEventListener('submit', (e) => {
  e.preventDefault()
  console.log('submit event')
})
form.addEventListener('reset', () => {
  console.log('reset event')
})
```

이벤트

Form 관련 이벤트

- <input type='text' /> 등 다양한 입력 태그들이 추가 될 수 있으며 이 입력 태그에 포커스가 추가되는 순간, 데이터가 변경되는 순간 등의 이벤트

| 이벤트 | 설명 |
|--------|---------------------------|
| change | 입력 데이터가 변경되는 순간의 이벤트 |
| focus | 입력 요소가 포커스를 가지는 순간의 이벤트 |
| blur | 입력 요소가 포커스를 잃어버리는 순간의 이벤트 |

자바스크립트 내장 객체

내장객체

- 자바스크립트 소프트웨어 언어에서 제공되는 객체
- 브라우저에서 제공되는 객체
- 백엔드 애플리케이션이라면 Node.js에서 제공되는 객체

자바스크립트 내장 객체 - Array

Array – 배열 선언 – [] 이용

- 배열은 여러 개의 데이터를 하나의 변수로 활용하기 위한 프로그래밍 기법이며 자바스크립트에서는 Array 타입의 객체
- 배열 객체를 선언하고 싶다면 대괄호([])을 이용

```
let products1 = ['나이키 운동화', '아식스 가방']  
let products2 = []
```

```
console.log(products1 instanceof Array)//true  
console.log(products2 instanceof Array)//true
```

자바스크립트 내장 객체 - Array

배열 객체 선언 – Array 생성자 이용

- Array 생성자를 이용해 배열 객체를 생성할 수 있습니다.

```
let products1 = new Array('나이키 운동화', '아식스 가방')
let products2 = new Array()
console.log(products1 instanceof Array)//true
console.log(products2 instanceof Array)//true
```

자바스크립트 내장 객체 - Array

알아두면 유용한 배열의 함수들

| 함수 | 설명 |
|------------------|---------------------------------------|
| concat() | 두 배열을 합쳐 하나의 배열을 만드는 함수 |
| join() | 배열 데이터를 구분자로 연결해 문자열로 만드는 함수 |
| push() | 배열에 데이터를 추가하는 함수 |
| unshift() | 배열에 맨 앞에 데이터를 추가하는 함수 |
| pop() | 배열에서 데이터를 제거하는 함수 |
| shift() | 맨 앞의 데이터를 제거하는 함수 |
| splice() | 지정된 위치의 데이터 삭제, 변경, 추가 |
| slice() | 특정 위치의 데이터 획득 |
| forEach() | 배열의 데이터 개수 만큼 함수 반복 실행 |
| filter() | 조건에 만족하는 배열 데이터만 추출 |
| every() | 배열의 데이터가 특정 조건에 모두 만족하는지 판단 |
| map() | 배열의 데이터로 함수를 실행, 반환 값을 모아서 배열로 만드는 함수 |

자바스크립트 내장 객체 - Array

concat()

- 두 배열을 결합시켜 새로운 배열을 만들어 주는 함수

```
let array1 = [10, 20]
let array2 = [100, 200]
let array3 = array1.concat(array2)
console.log(array3)//[10, 20, 100, 200]
```

join()

- 배열에 있는 여러건의 데이터를 하나로 묶어서 문자열로 반환
- 데이터를 매개변수의 구분자로 구분

```
let array = [10, 20, 30, 40]
let result = array.join('-')
console.log(result)//10-20-30-40
```

자바스크립트 내장 객체 - Array

push(), unshift()

- 배열에 데이터를 추가하는 함수는 push() 와 unshift()
- push() 함수는 마지막 위치에 데이터가 추가
- unshift() 는 맨 처음 위치에 데이터가 추가

```
let array = [10, 20]
array.push(100)
array.unshift(200)
console.log(array)//[200, 10, 20, 100]
```

pop(), shift()

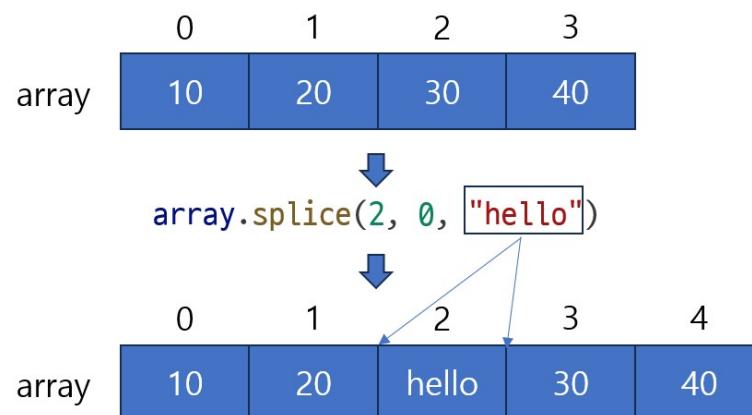
- 배열에 있는 데이터를 제거하기 위해 pop(), shift() 함수가 제공
- pop() 함수는 배열의 마지막 데이터가 제거
- shift() 는 맨 앞의 데이터가 제거

```
let array = [10, 20, 30, 40]
array.pop()
console.log(array)//[10, 20, 30]
array.shift()
console.log(array)//[20, 30]
```

자바스크립트 내장 객체 - Array

splice()

- 중간 특정 위치에 데이터를 추가하거나, 제거 혹은 교체
- . splice() 함수의 첫번째 매개변수에 데이터가 추가되어야 하는 위치를 인덱스로 지정
- 두번째 매개변수는 교체하고자 하는 데이터 개수, 이 값을 0으로 지정하면 교체하는 데이터 개수가 0임으로 기존의 데이터는 변경되지 않고 데이터 추가



자바스크립트 내장 객체 - Array

splice()

- splice() 함수를 이용해 특정 위치에 데이터 여러 개를 한꺼번에 추가할때는 추가하고자 하는 데이터를 세번 째 매개변수부터 여러 개를 나열

```
let array = [10, 20, 30, 40]
array.splice(2, 0, "hello", "world")
```

- 두번째 매개변수 값을 1로 지정하면 1개를 교체하게 되며 2라고 지정하면 2개를 한꺼번에 교체

```
let array = [10, 20, 30, 40]
array.splice(1, 2, "hello", "world", "js")
console.log(array)//[10, 'hello', 'world', 'js', 40]
```

- 데이터를 삭제하기 위해서는 인덱스 위치와 삭제하고자 하는 데이터 개수만 지정

```
let array = [10, 20, 30, 40]
array.splice(1, 2)
console.log(array)//[10, 40]
```

자바스크립트 내장 객체 - Array

slice()

- slice() 함수는 배열의 데이터를 획득할 때 사용
- slice() 를 사용하면 두개의 인덱스를 지정하고 그 인덱스 사이의 데이터 여러 개여 한꺼번에 획득
- slice() 는 획득한 데이터를 배열로 반환

```
let array = [10, 20, 30, 40, 50]
let resultArray = array.slice(1, 4)
console.log(resultArray)//[20, 30, 40]
```

- slice() 함수를 이용하면서 매개변수를 1개만 지정하게 되면 그 인덱스 위치의 데이터부터 모든 데이터가 획득

자바스크립트 내장 객체 - Array

forEach()

- 배열의 모든 데이터를 순차적으로 획득해서 특정 로직을 실행

```
let array = [1, 2, 3]
array.forEach((value) => {
  console.log(` ${value} * ${value} = ${Math.pow(value, 2)} `)
})
```

filter()

- filter() 함수는 배열의 데이터 중 조건에 맞는 데이터만 추출하고자 할 때 사용하는 함수

```
let array = [11, 3, 20, 15, 5]
let resultArray = array.filter((value) => {
  return value > 10
})
console.log(resultArray)//[11, 20, 15]
```

자바스크립트 내장 객체 - Array

every()

- 특정 조건에 모두 만족하는 데이터인지를 판단하기 위해서 사용

```
let array = [11, 3, 20, 15, 5]
let result = array.every((value) => {
    return value > 4
})
console.log(result)//false
```

map()

- 배열의 데이터로 특정 로직을 실행하고 그 결과를 다시 모아서 반환

```
let array = [1, 2, 3]
let resultArray = array.map((value) => {
    return value * 2
})
console.log(resultArray)//[2, 4, 6]
```

자바스크립트 내장 객체 - Date

- 시간/날짜 데이터는 애플리케이션에서 이용할 때 다양한 형태로 이용되며 다양한 연산이 필요한데 이를 도와주기 위한 자바스크립트 내장 객체가 Date 객체
- 매개변수를 지정하지 않고 생성하면 현재 시간

```
let date1 = new Date()
console.log(date1.toString())
//Tue Apr 23 2024 11:32:02 GMT+0900 (한국 표준시)
console.log(date1.toLocaleString())
//2024. 4. 23. 오전 11:32:02
```

- 특정 시간을 설정하고 싶다면 매개변수에 시간을 지정

```
let date2 = new Date("2024-10-09T10:10:10")
console.log(date2.toLocaleString())//2024. 10. 9. 오전 10:10:10
let date3 = new Date(2024, 10-1, 9, 10, 10, 10);
console.log(date3.toLocaleString())//2024. 10. 9. 오전 10:10:10
```

자바스크립트 내장 객체 - Date

원하는 데이터만 추출

- `getFullYear()` : 년도를 반환
- `getMonth()` : 월을 반환, 0이 1월
- `getDate()` : 일을 반환
- `getDay()` : 요일을 반환, 0이 일요일
- `getHours()` : 시간을 반환
- `getMinutes()` : 분을 반환
- `getSeconds()` : 초를 반환
- `getTime()`: 타임스탬프값 반환

자바스크립트 내장 객체 – Math

- Math.PI – 원주율 값을 가지는 프로퍼티
- Math.abs() – 절대값을 반환하는 함수
- Math.ceil() – 올림 값을 반환하는 함수
- Math.floor() – 내림 값을 반환하는 함수
- Math.round() – 반올림 값을 반환하는 함수
- Math.max() – 최대값을 구하는 함수
- Math.min() – 최소값을 구하는 함수
- Math.pow() – 제곱근을 구하는 함수
- Math.random() – 난수를 구하는 함수

자바스크립트 내장 객체 – Math

난수 발생

- Math.random() 함수를 이용하면 발생하는 난수는 0 과 1 사이의 실수 값

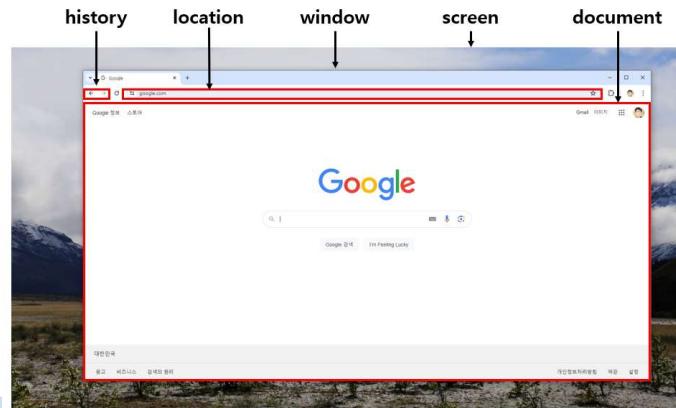
```
//0 < random < 1  
console.log(Math.random())//0.6958603372854415
```

```
//0 ~ 100 사이의 난수  
console.log(Math.random() * 100)//73.27559075370138
```

```
//51~70 사이의 난수  
let max = 70;  
let min = 51;  
console.log(Math.random() * (max - min) + min)//63.48809433223897
```

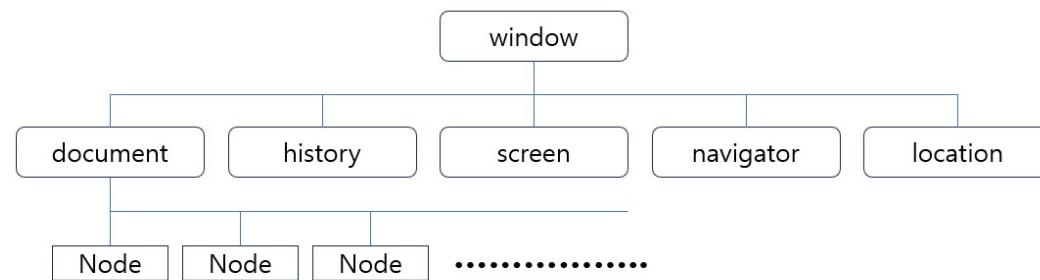
브라우저 내장 객체

- 브라우저 내장 객체는 브라우저 창의 정보를 획득하거나 브라우저를 제어하기 위한 객체
 - window : 브라우저 창을 지칭하는 객체
 - document : 브라우저에 출력되는 HTML 문서를 지칭하는 객체
 - history : 브라우저의 인터넷 방문 기록을 지칭하는 객체
 - screen : 브라우저가 띠 있는 스크린 창을 지칭하는 객체
 - navigator : 브라우저에 대한 정보를 제공하는 객체
 - location : 브라우저의 출력 URL 을 지칭하는 객체



브라우저 내장 객체

- 브라우저의 객체들은 모두 window 의 객체들로 window 부터 계층 구조로 구성



```
window.document.querySelector('#a1')
```

```
document.querySelector('#a1')
```

브라우저 내장 객체 - window

- window 객체는 브라우저 창을 지칭하는 객체
- 브라우저에서 실행되는 애플리케이션 모든 곳에서 이용이 가능한 객체
- alert() 함수도 window 객체의 함수
- console 도 window 에 선언된 객체
- document, location, navigator, history, screen 도 window 에 선언된 객체

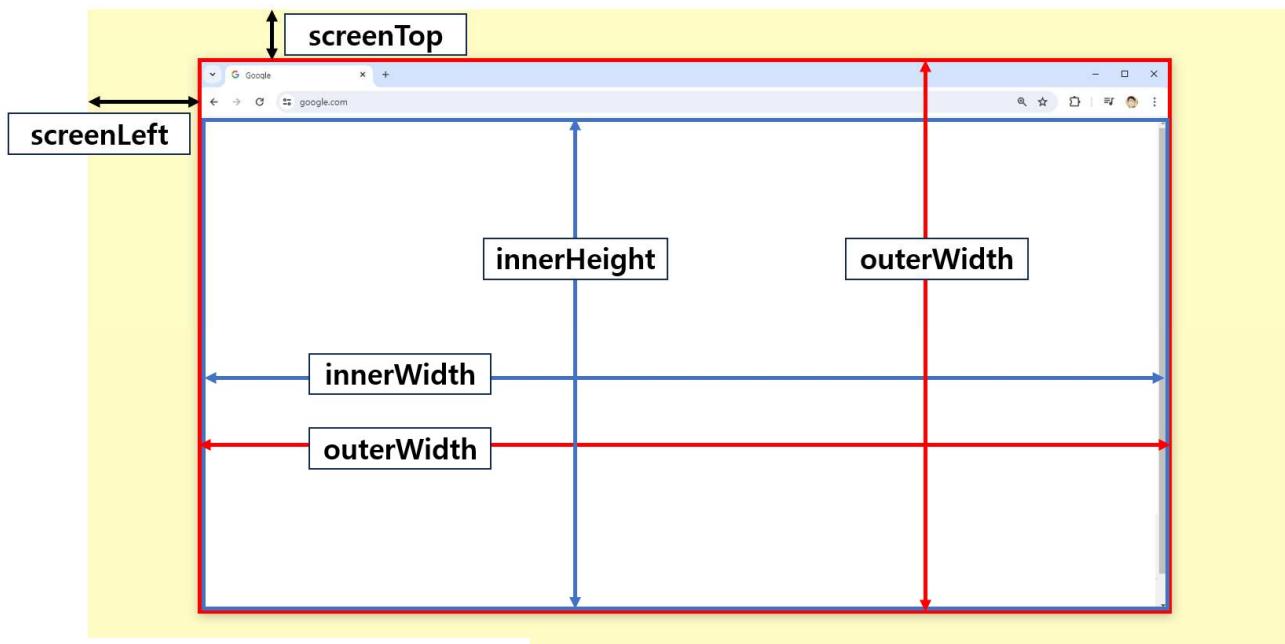
브라우저 내장 객체 - window

window 프로퍼티

| 프로퍼티 | 설명 |
|--------------------|--------------------------|
| document | HTML 문서를 지칭하는 객체 |
| location | 브라우저의 URL 정보를 지칭하는 객체 |
| screen | 스크린 창을 지칭하는 객체 |
| history | 브라우저의 인터넷 방문 기록을 지칭하는 객체 |
| navigator | 브라우저의 다양한 정보를 가지는 객체 |
| console | 브라우저 콘솔 창, 주로 로그 출력에 이용 |
| innerWidth | HTML 문서가 출력되는 부분의 가로 사이즈 |
| innerHeight | HTML 문서가 출력되는 부분의 세로 사이즈 |
| outerWidth | 브라우저 창의 가로 사이즈 |
| outerHeight | 브라우저 창의 세로 사이즈 |
| screenLeft | 스크린 왼쪽에서 브라우저가 위치한 거리 |
| screenTop | 스크린 위에서 브라우저가 위치한 거리 |
| scrollX | 브라우저의 가로방향 스크롤 위치 |
| scrollY | 브라우저의 세로방향 스크롤 위치 |

브라우저 내장 객체 - window

window 프로퍼티



```
console.log(window.innerWidth, window.innerHeight)
console.log(window.outerWidth, window.outerHeight)
console.log(window.screenLeft, window.screenTop)
window.addEventListener('scroll', () => {
  console.log(window.scrollX, window.scrollY)
})
```

브라우저 내장 객체 - window

window 함수

| 함수 | 설명 |
|-------------------------|-------------------------|
| <code>alert()</code> | 알림 다이얼로그 띄우기 |
| <code>confirm()</code> | 확인, 취소 버튼이 있는 다이얼로그 띄우기 |
| <code>prompt()</code> | 사용자에게 입력받는 다이얼로그 띄우기 |
| <code>open()</code> | 새로운 URL 문서를 띄운다. |
| <code>close()</code> | 브라우저 창 닫기 |
| <code>scrollBy()</code> | 스크롤 시키기 |

브라우저 내장 객체 - window

open() 함수

- open(url) : url 만 지정해서 open() 함수 호출
- open(url, target) : url 과 target 을 지정해서 open() 함수 호출
- open(url, target, windowFeatures) : url, target 과 다양한 정보를 설정해서 open() 함수 호출
- target 정보는 아래의 정보 중 하나를 지정
 - _black : 새 창이 열리고 그곳에 url 출력
 - _parent : 부모 창에 url 출력
 - _self : 자신 창에 url 출력

```
window.open("http://www.google.com", "_self")
```

브라우저 내장 객체 - window

open() 함수

- windowFeatures 에 창에 대한 정보를 설정
 - width, height : 새로운 브라우저 창의 사이즈
 - left, top : 새로운 브라우저 창이 뜨는 위치

```
window.open(  
    "http://www.naver.com",  
    "_blank",  
    "left=100,top=100,width=300,height=400"  
)
```

브라우저 내장 객체 – location

- location 객체는 브라우저의 URL 을 다루기 위한 객체

location 프로퍼티

| 프로퍼티 | 설명 |
|-----------------|---------------------------|
| href | 브라우저의 URL 정보 |
| protocol | URL 의 프로토콜 정보 |
| host | URL 의 호스트 정보, 호스트명과 포트 정보 |
| hostname | URL 의 호스트명정보 |
| port | URL 의 포트 정보 |
| pathname | URL 의 경로 정보 |
| search | URL 의 쿼리 문자열 정보 |
| hash | URL 의 해시 정보 |
| origin | URL 의 백엔드를 식별하기 위한 정보 |

- hostname 은 도메인 정보이며 hostname 과 port 를 포함하는 정보가 host 입니다. 그리고 origin 은 백엔드를 식별하기 위한 정보로 protocol + host 까지의 정보

브라우저 내장 객체 – location

location 함수

| 함수 | 설명 |
|------------------------|----------------|
| <code>reload()</code> | 현재의 URL로 새로 요청 |
| <code>replace()</code> | 주어진 URL로 이동 |

- `reload()` 함수는 현재의 URL로 다시 요청하는 것임으로 마치 브라우저의 새로고침 버튼을 누른것과 동일한 효과
- `location.href` 로 이동하면 현재의 URL 정보가 브라우저 히스토리에 유지
- `location.replace()` 로 이동을 하면 현재의 URL 정보는 유지되지 않습니다.

브라우저 내장 객체 – history

history 프로퍼티

| 프로퍼티 | 설명 |
|---------------|-----------------|
| length | 히스토리에 등록된 정보 개수 |

- 1이 반환된다면 브라우저의 히스토리 정보에는 현재 화면에 출력된 URL 정보만 있는 것입니다.

history 함수

| 프로퍼티 | 설명 |
|------------------|-----------------|
| back() | 이전 화면으로 이동 |
| forward() | 이후 화면으로 이동 |
| go() | 히스토리의 특정 위치로 이동 |

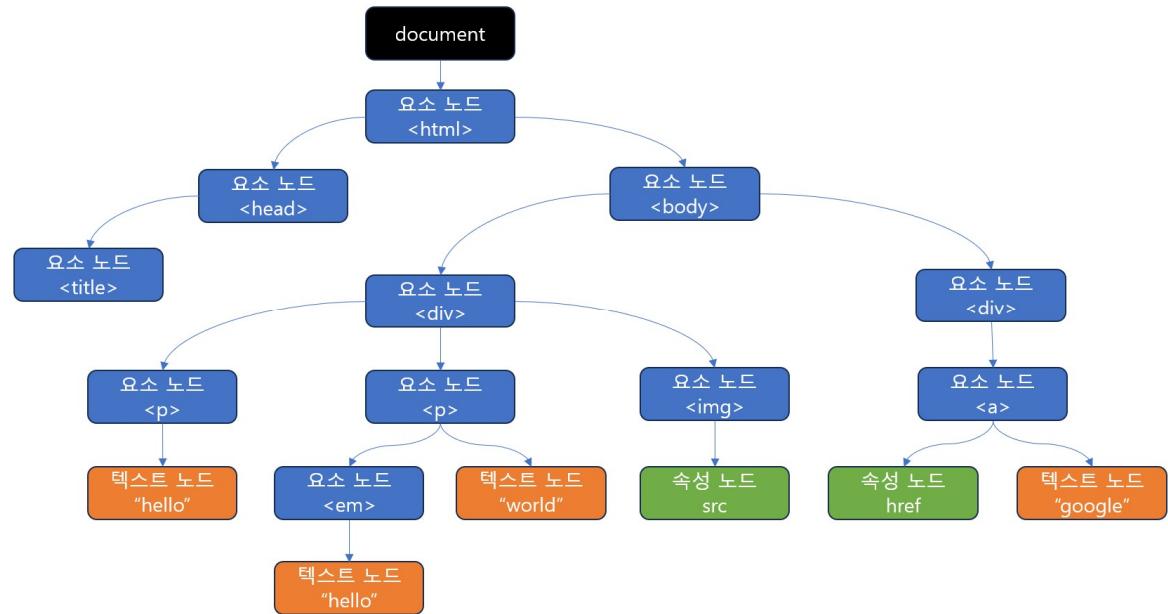
- go()함수의 매개변수에 숫자를 지정하면 그 단계의 앞, 뒤로 이동이 가능

브라우저 내장 객체 – document

- 브라우저에 출력되는 HTML 문서 자체를 지칭하는 객체
- document 객체에 만들어지는 노드는 HTML 문서의 어느 구성 요소를 지칭하는지에 따라 구분
 - 요소 노드 : 태그를 지칭하는 노드
 - 속성 노드 : 태그의 속성을 지칭하는 노드
 - 텍스트 노드 : 태그의 바디에 작성된 글을 지칭하는 노드
 - 주석 노드 : 문서의 주석을 지칭하는 노드

브라우저 내장 객체 – document

```
<html>
  <head>
    <title>Document</title>
  </head>
  <body>
    <div>
      <p>hello</p>
      <p>
        <em>hello</em>world
      </p>
      
    </div>
    <div>
      <a href="http://www.google.com">google</a>
    </div>
  </body>
</html>
```



브라우저 내장 객체 – document

getElementById()

- 노드를 획득하는 방법은 여러가지가 있지만 가장 기본이 되는 방법이 id 값을 이용해 획득하는 것
- id 는 모든 태그에 추가할 수 있는 속성이며 태그에 의해 출력되는 화면과 관련없이 태그를 식별할 목적으로 추가되는 것임

```
let oneNode = document.getElementById('one')
console.log(oneNode.innerText)//hello
```

브라우저 내장 객체 – document

getElementsByName()

- getElementsByName() 함수는 태그명을 매개변수에 설정하고 그 태그명에 해당되는 노드를 획득
- 반환 값은 획득된 노드 객체 여러 개가 담겨있는 HTMLCollection 객체

```
let divNodes = document.getElementsByTagName('div')

for(let i=0; i<divNodes.length; i++){
    console.log(divNodes[i].innerText)
}
```

브라우저 내장 객체 – document

getElementsByClassName()

- 태그에 적용된 CSS 클래스명으로 획득
- 클래스명이 선언된 모든 노드 객체가 HTMLCollection 타입으로 반환

```
let enableNodes = document.getElementsByClassName('enable')
for(let i=0; i<enableNodes.length; i++){
    console.log(enableNodes[i].innerText)
}
```

getElementsByName()

- name 속성 값으로 획득
- 반환 값은 HTMLCollection

```
<input type="text" name="id" />

let idInput = document.getElementsByName('id')
console.log(idInput[0].value)
```

브라우저 내장 객체 – document

innerHTML vs innerText

- 노드 객체 하위에 선언된 자식 노드를 얻거나 변경
- 어떤 노드의 자식 노드를 이용할 때 innerHTML 과 innerText 를 이용

```
<div id="one">hello</div>
```

```
let divNode = document.getElementById('one')
console.log(divNode.innerHTML)//hello
console.log(divNode.innerText)//hello
```

브라우저 내장 객체 – document

innerHTML vs innerText

- innerHTML 은 어떤 노드의 하위에 선언된 모든 구성요소를 포함시키지만 innerText 는 텍스트 노드만 포함

```
<div id="two"><a href="#">google</a></div>
```

```
let twoNode = document.getElementById('two')
console.log(twoNode.innerHTML)//<a href="#">google</a>
console.log(twoNode.innerText)//google
```

result1.innerHTML = 'google'



google

google

result2.innerText = 'google'

브라우저 내장 객체 – document

속성 이용

- getAttribute() : 노드의 속성 값 획득
- setAttribute(): 노드의 속성 값 변경
- removeAttribute() : 노드의 속성 삭제
- hasAttribute() : 노드에 속성이 있는지 판단

```
<body>  
  <a id="link1" href="http://www.google.com">google</a><br/>  
  <a id="link2">link</a><br/>  
  <a id="link3" href="http://www.google.com">link</a>  
</body>
```

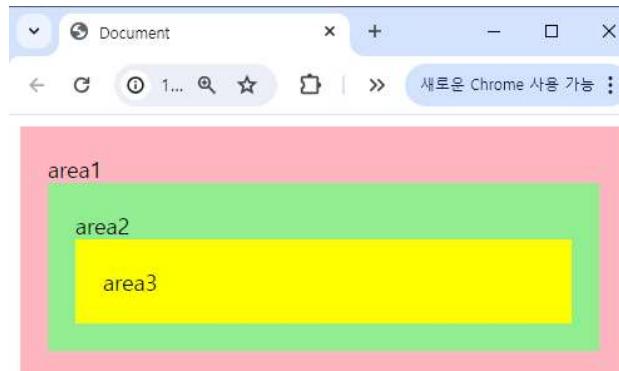
```
let link1 = document.getElementById('link1')  
console.log(link1.href)//http://www.google.com  
console.log(link1.getAttribute('href'))//http://www.google.com  
  
let link2 = document.getElementById('link2')  
//link2.href = 'http://www.naver.com'  
link2.setAttribute('href', 'http://www.naver.com')  
  
let link3 = document.getElementById('link3')  
link3.removeAttribute('href')  
console.log(link3.hasAttribute('href'))//false
```

브라우저 내장 객체 – document

타겟팅과 버블링

- 노드에서 발생하는 이벤트를 처리하기 위해서는 이벤트 캡처링과 버블링에 대한 이해가 필요

```
<body>  
  <div id="area1">  
    area1  
    <div id="area2">  
      area2  
      <div id="area3">  
        area3  
      </div>  
    </div>  
  </div>
```

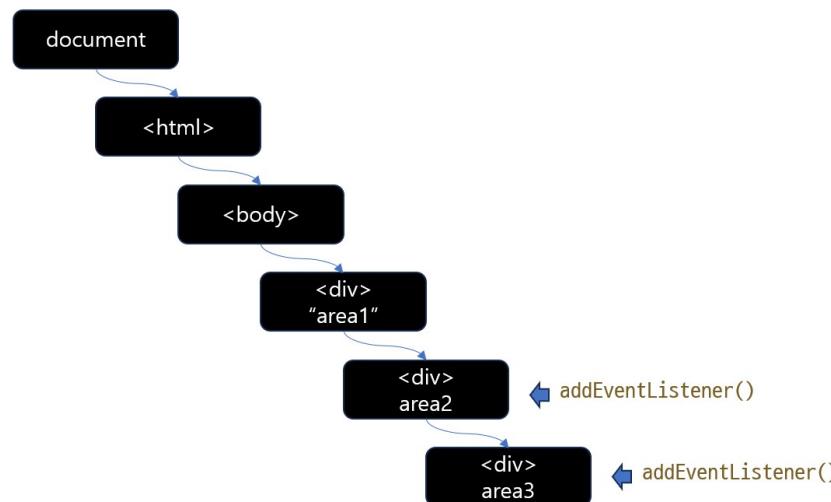


- 만약 사용자가 area3 를 클릭하면 이 이벤트는 area3 이기도 하지만 area2, area1 이기도 합니다.

브라우저 내장 객체 – document

타겟팅과 버블링

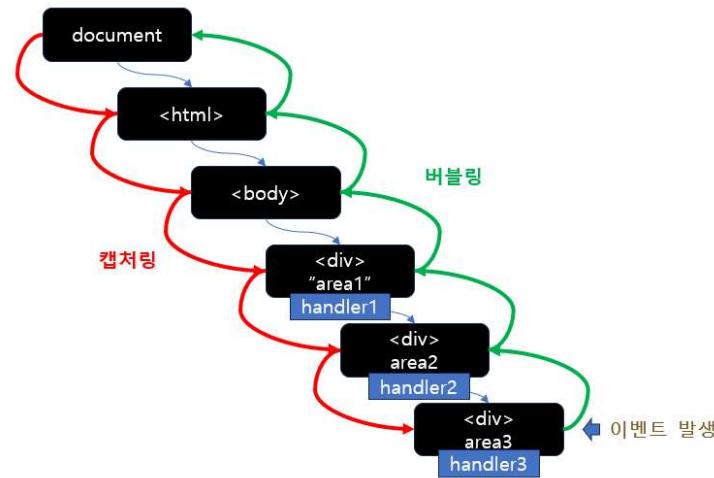
- 중첩되어 있는 경우라고 하더라도 코드에서 area3 에만 이벤트 핸들러를 등록했다면 문제될 것은 없습니다.
- 하지만 중첩구조에서 여러 노드에 이벤트 핸들러를 등록하게 되는 경우
- 둘 중 어느것이 먼저 실행되어야 하는지 실행 순서를 제어되어야 한다면?
- 먼저 실행되는 이벤트 핸들러에서 그다음 순서의 이벤트 핸들러 실행을 취소시키고 싶다면?



브라우저 내장 객체 – document

타겟팅과 버블링

- 캡처링은 이벤트가 발생했을 때 상위 노드부터 하위노드로 이벤트가 전파되는 단계
- 버블링은 이벤트가 발생한 노드부터 상위 노드로 이벤트가 전파되는 단계



- 이벤트 처리를 캡처링 단계에서 할것인지 버블링 단계에서 할것인지 제어는 이벤트를 등록

브라우저 내장 객체 – document

타겟팅과 버블링

- addEventListener(type, listener, useCapture)
- 3번째 매개변수는 true/false 값을 가지는데 이 값이 false 이면 버블링 단계에서 이벤트가 처리됨을 의미하며 true 이면 캡처링 단계에서 이벤트가 처리됨을 의미
- 보통의 경우 노드의 이벤트는 버블링 단계에서 처리하며 addEventListener() 의 세번째 매개변수를 지정하지 않으면 기본값이 false 임

브라우저 내장 객체 – document

이벤트 중단

- 핸들러에서 어떤 처리를 한 후에 이 이벤트에 의해 다른 것이 실행되지 못하게 해야 하는 경우
- preventDefault() 함수를 이용하는 방법과 stopPropagation() 함수를 이용하는 방법

preventDefault()

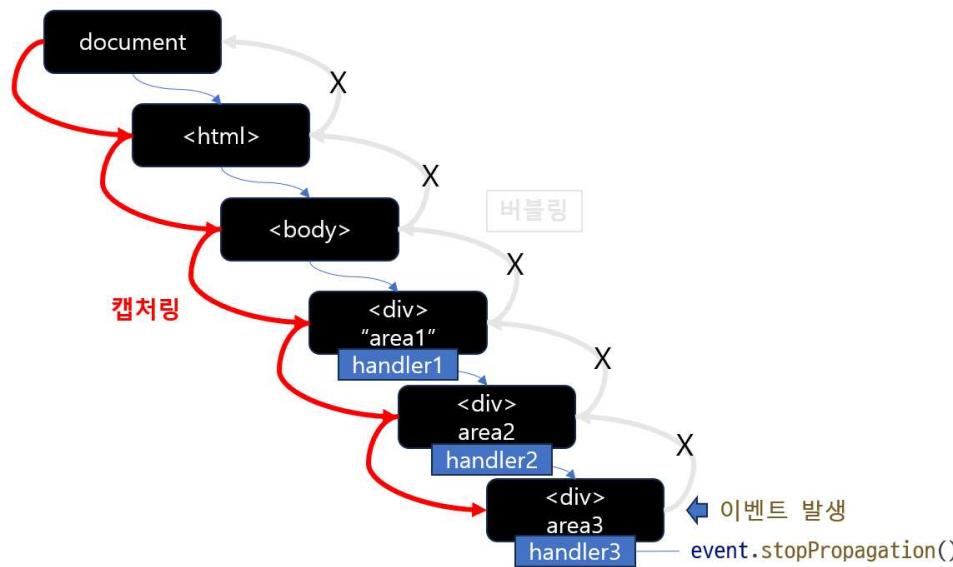
- preventDefault() 는 노드에 기본으로 등록된 이벤트가 처리되지 않게 하기 위해서 사용
- <a> 태그, <form> 태그가 선언되어 있는데 이 태그들은 자바스크립트에서 별도의 이벤트 처리하지 않아도 각 태그의 기본 동작 이벤트가 있습니다.

```
link1.addEventListener('click', function(e) {  
    console.log('link click..')  
    e.preventDefault()  
})
```

브라우저 내장 객체 – document

stopPropagation()

- stopPropagation() 은 이 캡처링과 버블링을 중단시키는 함수



브라우저 내장 객체 – document

스타일 이용

- 노드의 style 프로퍼티

```
<div id="area1" style="width: 200px; height: 200px; background-color: green;"></div>
```

```
let area1 = document.getElementById('area1')
```

```
console.log(area1.style.width)//200px  
console.log(area1.style.height)//200px  
console.log(area1.style.backgroundColor)//green
```

- camel case 명명 규칙으로 이용

브라우저 내장 객체 – document

스타일 이용

- 태그의 style 속성이 아닌 외부에서 정의해서 적용된 스타일을 획득하고자 한다면 노드의 style 속성을 이용 할 수 없으며 getComputedStyle() 함수를 이용

```
console.log(getComputedStyle(area1).width)//200px
```

브라우저 내장 객체 – document

노드 추가

- 자바스크립트에서 HTML 문서에 새로운 노드를 추가하는 방법은 이미 살펴본 innerHTML 을 이용

노드 객체 생성

- createElement() : 요소 노드 생성
- createAttribute() : 속성 노드 생성
- createTextNode() : 텍스트 노드 생성

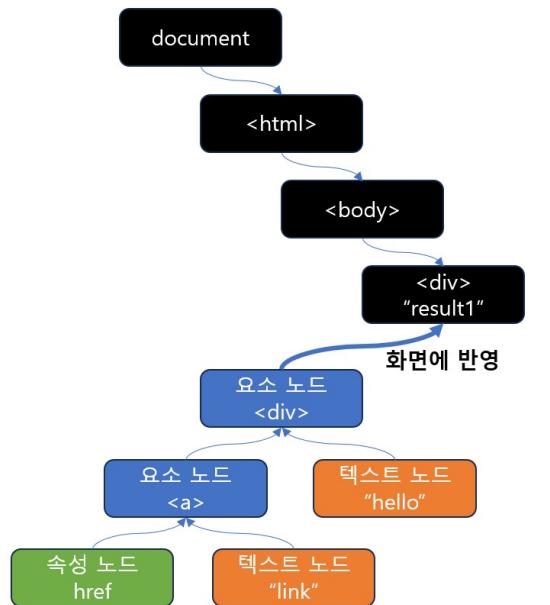
```
//<div><a href="#">link</a>hello</div>
let newDiv = document.createElement('div')
let newA = document.createElement('a')
```

```
let newHref = document.createAttribute('href')
newHref.value = '#'
```

```
let newAText = document.createTextNode('link')
let newDivText = document.createTextNode('hello')
```

브라우저 내장 객체 – document

appendChild()로 노드 객체 추가



`newA.setAttributeNode(newHref)`

`newA.appendChild(newAText)`

`newDiv.appendChild(newA)`

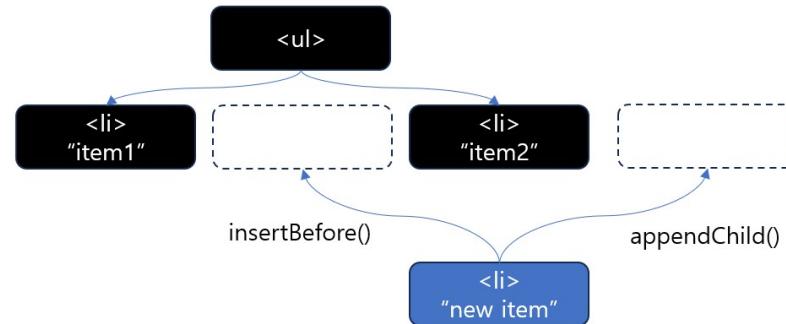
`newDiv.appendChild(newDivText)`

`result1.appendChild(newDiv)`

브라우저 내장 객체 – document

insertBefore() 함수로 노드 추가

- appendChild() 함수를 이용하면 새로 추가하는 노드는 마지막 부분에 추가
- 특정 위치에 추가하고 싶다면 insertBefore() 함수를 이용



노드 객체 삭제

- 자바스크립트에서 특정 노드를 삭제하고 싶다면 removeChild() 함수를 이용

객체 리터럴

객체 = 관련된 변수나 함수를 하나로 묶어서 사용하려고 묶어놓은 단위

자바스크립트에서 객체를 만드는 방법

- 객체 리터럴
- 생성자 함수
- 클래스

객체 리터럴

- 객체 리터럴은 객체의 내용을 중괄호({})로 묶고 그 안에 객체의 내용을 등록
- 객체의 프로퍼티는 키:값 형태를 띠며 여러 프로퍼티를 콤마(,)로 구분해서 등록하는 방식

```
let user = {  
    name: '홍길동',  
    age: 20,  
    isMember: true,  
    order: {  
        productId: 2,  
        count: 10  
    }  
}  
  
console.log(user.name)//홍길동  
console.log(user.order.productId)//2
```

객체 리터럴

- 함수도 데이터로 활용이 가능함으로 프로퍼티에 함수 대입도 가능

```
let user = {  
    ... ... ... ... ... ...  
    sayHello: function(){  
        console.log(`Hello, ${this.name}`)//Hello, 홍길동  
    }  
}  
user.sayHello()
```

객체 리터럴

객체의 this

- 객체에 선언된 다른 멤버를 이용하는 경우 this 사용

```
let user = {  
    name: '홍길동',  
    age: 20,  
    isMember: true,  
    sayHello: function(){  
        console.log(`Hello, ${this.name} - age = ${age}`)//error  
    }  
}
```

- this.name 이라고 작성하면 이 함수가 등록된 객체, 즉 user 객체내의 name
- age 라고 사용하면 이 age 는 window 객체의 age

객체 리터럴

객체의 this

- 함수는 function 예약어로 선언될 수도 있고 화살표 함수로 선언될 수도 있습니다.
- function 으로 함수를 선언하면 this 는 객체 리터럴
- 화살표 함수로 선언하면 this 는 window 객체

```
let user = {
    name: '홍길동',
    age: 20,
    sayHello1: function() {
        console.log(` ${this.name}, ${this.age}`)
    },
    sayHello2: () => {
        console.log(` ${this.name}, ${this.age}`)
    }
}

user.sayHello1() //홍길동, 20
user.sayHello2() //, undefined
```

객체 리터럴

축약형으로 프로퍼티 등록

- 등록하고자 하는 키의 이름과 대입하고자 하는 값의 변수명이 동일한 경우

```
let user = {  
    name,  
    age,  
    sayHello: function () {  
        console.log(` ${this.name}, ${this.age}`);  
    },  
};
```

객체 리터럴

외부에서 객체 멤버 등록

- 객체를 선언하고 이후에 그 객체에 멤버를 추가

```
let user = {
    name: '홍길동',
    sayHello1: function() {
        console.log(` ${this.name}, ${this.age}`)
    }
}
user.age = 20
user.sayHello2 = function(){
    console.log(` ${this.name}, ${this.age}`)
}
```

생성자 함수

- 생성자 함수는 객체를 만드는 역할을 하는 함수
- 일반 함수와 문법적인 차이는 없습니다.
- 객체를 생성하는 목적으로 선언된 함수라는 의미에서 생성자 함수라고 부릅니다.
- 생성자 함수는 관습상 일반 함수와 구분하기 위해서 첫 글자를 대문자로 선언

```
//객체의 모형을 생성자 함수로 선언
function User(name, age){
    this.name = name;
    this.age = age;
}

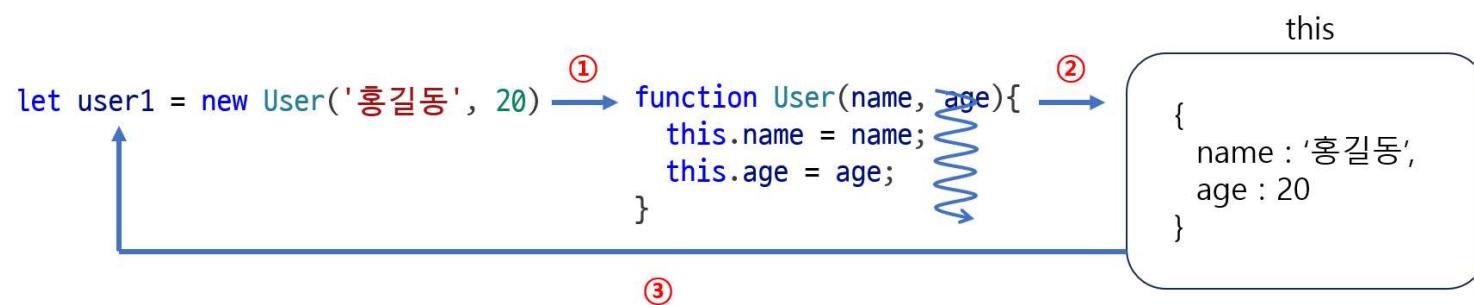
//생성자 함수를 이용해 객체 생성
let user1 = new User('홍길동', 20)
let user2 = new User('김길동', 30)

console.log(user1)//User {name: '홍길동', age: 20}
console.log(user2)//User {name: '김길동', age: 30}
```

생성자 함수

new 연산자를 이용해 객체 생성

- 생성자 함수를 호출해서 객체를 생성할 때는 new 연산자를 이용
- new 연산자를 이용하게 되면 함수 호출과 동시에 객체가 자동으로 만들어지고 그 객체에 함수내에 this로 선언한 멤버들이 등록
- 만들어진 객체가 최종 생성한 곳에 자동으로 반환



생성자 함수

new 연산자를 이용해 객체 생성

- new 를 이용하지 않고 함수를 호출하게 되면 명시적으로 객체를 생성하겠다고 선언한 것이 아님으로 객체가 자동으로 만들어지지 않습니다.

```
function User(name, age){  
    this.name = name;  
    this.age = age;  
}  
  
//new를 사용하지 않고 호출  
let user2 = User('김길동', 30)//error  
console.log(user2.name, user2.age)
```

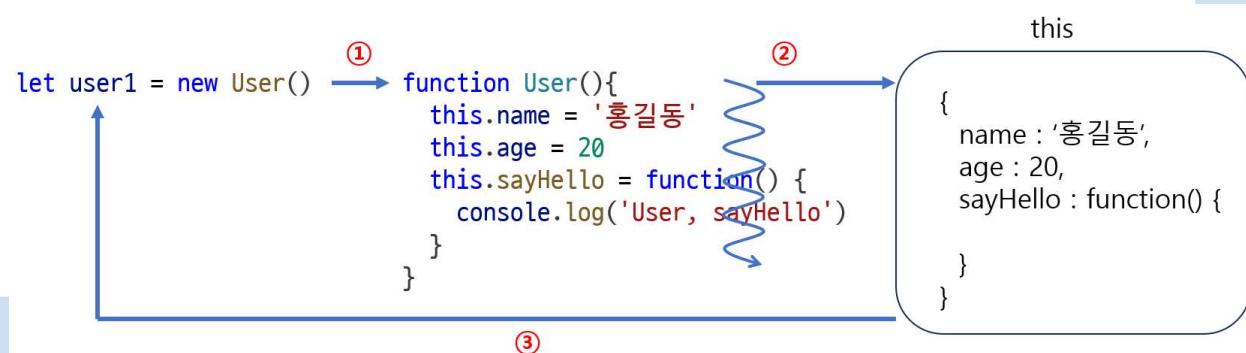
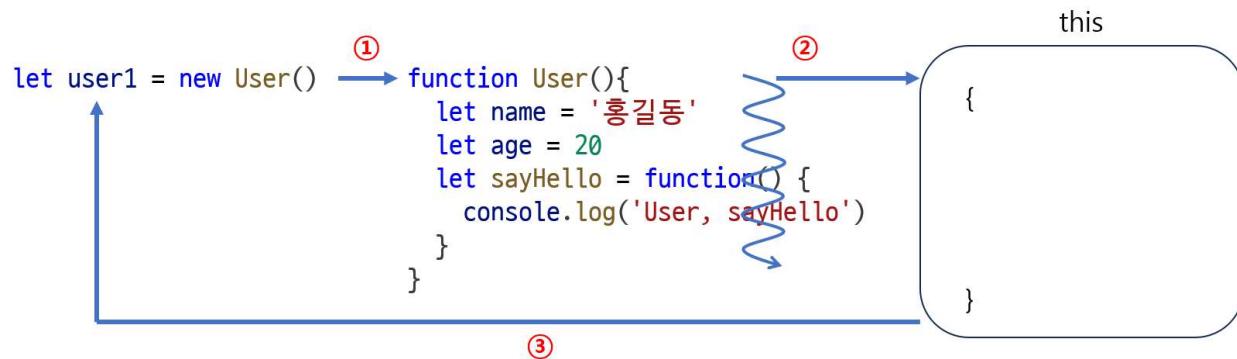
let user1 = User('홍길동', 20) → function User(name, age){
 this.name = name;
 this.age = age;
}

① this ?

생성자 함수

this 예약어로 객체 멤버 등록

- 객체에 포함될 변수, 함수는 this 예약어로 선언
- this 예약어를 사용하지 않은 변수, 함수는 생성되는 객체와 관련이 없습니다.



생성자 함수

생성자 함수의 리턴 값

- 생성자 함수는 객체 생성이 주 목적임으로 new 연산자로 함수를 호출하게 되면 자동으로 만들어지는 객체가 return 예약어를 사용하지 않아도 자동으로 반환
- return 구문을 생성자 함수에서 사용한다고 문제되지는 않습니다.
- 단지 return 에 의해 반환되는 값이 기초 데이터인지 객체인지에 따라 차이가 있습니다.
- 기초 데이터를 return 시키면 이 return 은 무시되며 자동으로 만들어진 객체가 반환

```
function User(name, age){  
    this.name = name  
    this.age = age  
    return 10  
}  
  
let result = new User('홍길동', 20)  
console.log(result)//User {name: '홍길동', age: 20}  
console.log(result.name, result.age)//홍길동 20
```

생성자 함수

생성자 함수의 리턴 값

- 생성자 함수에서 명시적으로 return 을 이용해 특정 객체를 반환한다면 자동으로 생성된 this 객체는 무시

```
function User(name, age){  
    this.name = name  
    this.age = age  
    return {  
        prodId: 1,  
        prodName: '노트북'  
    }  
}  
  
let result = new User('홍길동', 20)  
console.log(result)//{prodId: 1, prodName: '노트북'}  
console.log(result.name, result.age)//undefined undefined
```

생성자 함수

외부에서 멤버 추가

- 생성자 함수로 객체를 생성한 후에 원한다면 그 객체에 변수, 함수등의 멤버를 추가할 수 있습니다.

```
function User(name, age){  
    this.name = name  
    this.age = age  
}  
  
let user1 = new User('홍길동', 20)  
let user2 = new User('김길동', 30)  
  
console.log(user1)//User {name: '홍길동', age: 20}  
console.log(user2)//User {name: '김길동', age: 30}  
  
user1.address = 'seoul'  
user2.phone = '010-1111-1111'  
  
console.log(user1)//User {name: '홍길동', age: 20, address: 'seoul'}  
console.log(user2)//User {name: '김길동', age: 30, phone: '010-1111-1111'}
```

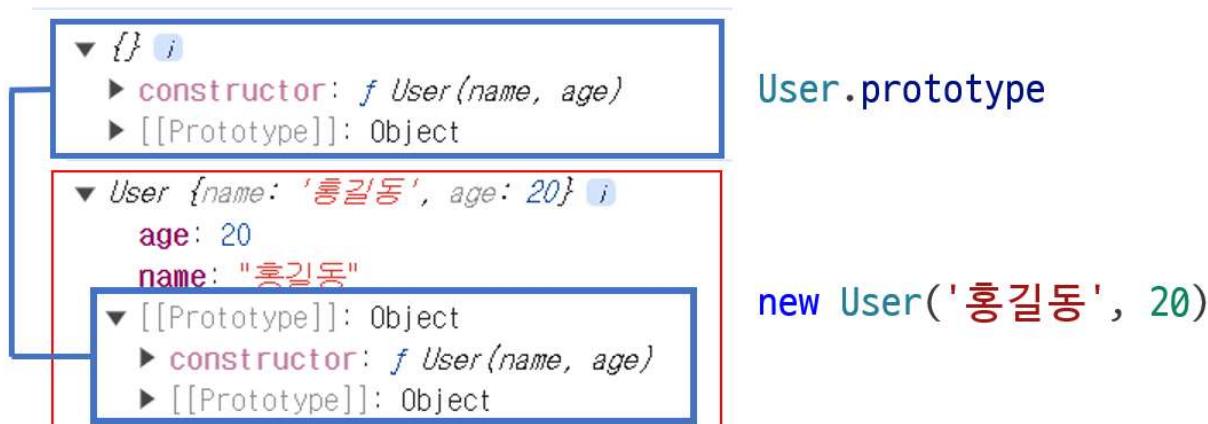
프로토타입

- “프로토타입”이라는 단어의 일반적인 의미는 어떤 제품을 만들기 전의 시제품이라는 의미
- 자바스크립트에서 프로토타입은 객체의 시제품 정도로 이해해 주면 됩니다.
- 자바스크립트에서 함수를 선언하면 자동으로 그 함수를 위한 프로토타입 객체가 만들어 집니다.
- 함수를 이용해 객체를 생성해야 만들어지는 것이 아니라 함수를 선언하는 것만으로 자동으로 프로토타입 객체가 만들어 지는 것입니다.
- 프로토타입 객체를 참조할 수 있게 함수내에 prototype이라는 프로퍼티가 추가

```
function User(name, age){  
    this.name = name  
    this.age = age  
}  
  
console.log(User.prototype)  
console.log(new User('홍길동', 20))
```

프로토타입

- 프로토타입 객체에 자동으로 constructor 라는 생성자가 추가
- 자바스크립트에서 함수를 이용해 객체를 생성한다는 의미는 사실 내부적으로 프로토타입의 생성자를 이용하는 것
- new User() 로 객체를 생성하게 되면 내부적으로는 User 함수의 프로토타입 객체내의 constructor() 라는 생성자가 이용되고 이 생성자에 의해 객체가 생성



프로토타입

vs this

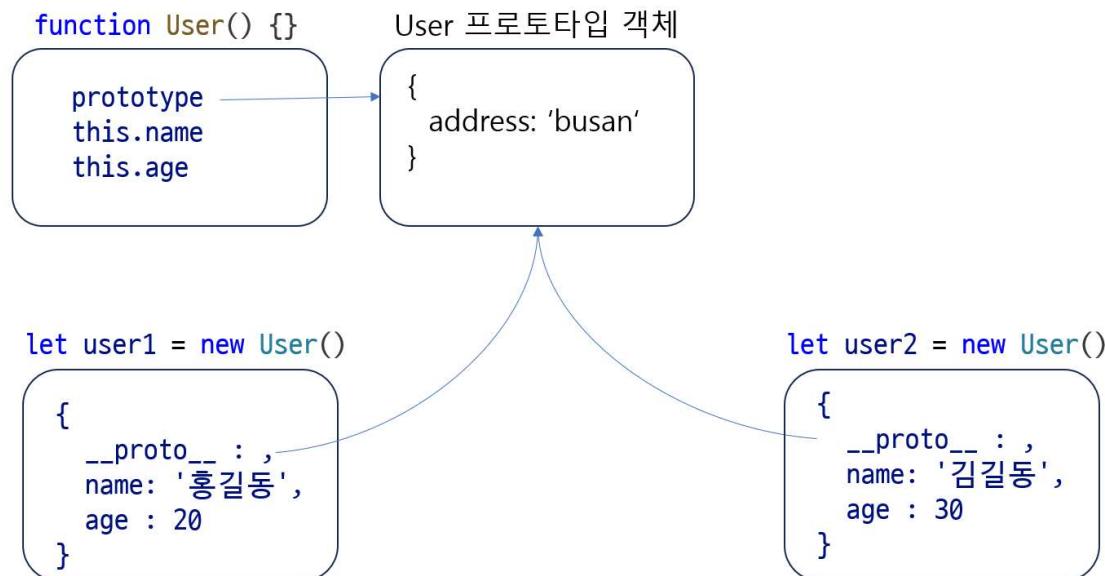
- 객체를 위한 프로퍼티와 함수가 선언되어야 하는데 객체의 this 멤버로 선언할 수도 있고 프로토타입 멤버로 선언할 수도 있습니다.
- this 멤버나 프로토타입 멤버를 모두 객체에서 이용할 수는 있지만 이 둘은 차이가 있습니다.

```
function User(name, age, address){  
    this.name = name  
    this.age = age  
    User.prototype.address = address  
}  
  
let user1 = new User('홍길동', 20, 'seoul')  
let user2 = new User('김길동', 30, 'busan')  
  
console.log(user1.name, user1.age, user1.address)//홍길동 20 busan  
console.log(user2.name, user2.age, user2.address)//김길동 30 busan
```

프로토타입

vs this

- this 는 객체가 생성될 때마다 객체별로 메모리가 별도로 할당되기 때문에 객체별로 다른 값을 유지
- 프로토타입은 여러 개의 객체가 만들어진다고 하더라도 객체끼리 공유



프로토타입

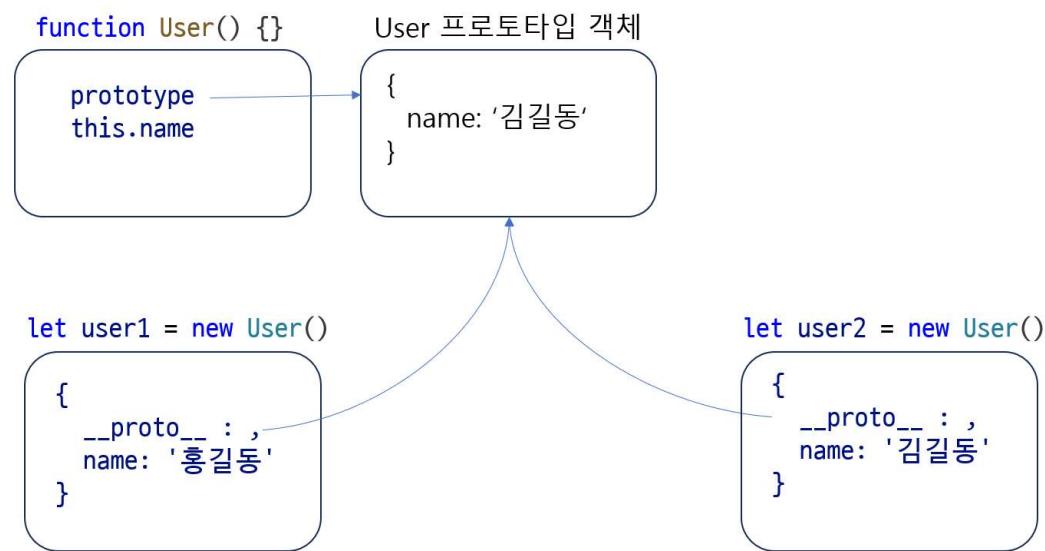
vs this

- this 와 프로토타입에 동일 이름의 멤버 선언이 가능
- 자신의 메모리에서 찾게 되고 없는 경우에만 내부 참조를 이용해 프로토타입의 멤버에 접근

```
function User(name){  
    this.name = name  
    User.prototype.name = name  
}  
  
let user1 = new User('홍길동')  
let user2 = new User('김길동')  
  
console.log(user1.name, user1.__proto__.name)//홍길동 김길동  
console.log(user2.name, user2.__proto__.name)//김길동 김길동
```

프로토타입

vs this



프로토타입

vs this

- 객체들끼리 공유해야 하는 멤버를 프로토타입으로 선언한 후에 어떤 특정 객체에서 프로토타입에 선언한 멤버를 다시 자신의 this에 선언해서 특정 객체만 다른 값 혹은 로직을 가지도록 할 수 있습니다.

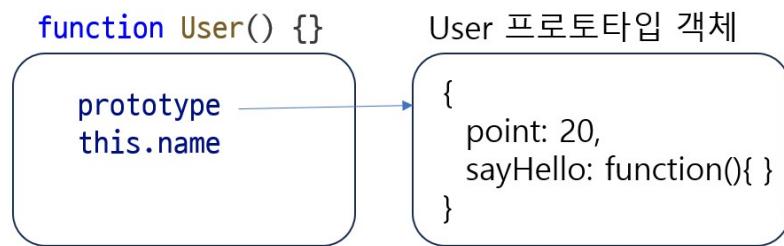
```
function User(name){  
  this.name = name  
  User.prototype.point = 20  
  User.prototype.sayHello = function(){  
    console.log(`Hello ${this.name}, point : ${this.point}`)  
  }  
}
```

```
let user1 = new User('honggildong')  
user1.sayHello() //Hello honggildong, point : 20
```

```
let user2 = new User('이길동')  
user2.point = 30  
user2.sayHello = function() {  
  console.log(`안녕하세요 ${this.name}, 포인트 : ${this.point}`)  
}  
  
user2.sayHello() //안녕하세요 이길동, 포인트 : 30  
  
let user3 = new User('kimgildong')  
user3.sayHello() //Hello kimgildong, point : 20
```

프로토타입

vs this



```
let user1 = new User()  
{  
  __proto__: ,  
  name: 'honggildong'  
}
```

```
let user2 = new User()  
{  
  __proto__: ,  
  name: '이길동',  
  point: 30,  
  sayHello: function() {}  
}
```

```
let user3 = new User()  
{  
  __proto__: ,  
  name: 'kimgildong'  
}
```

프로토타입

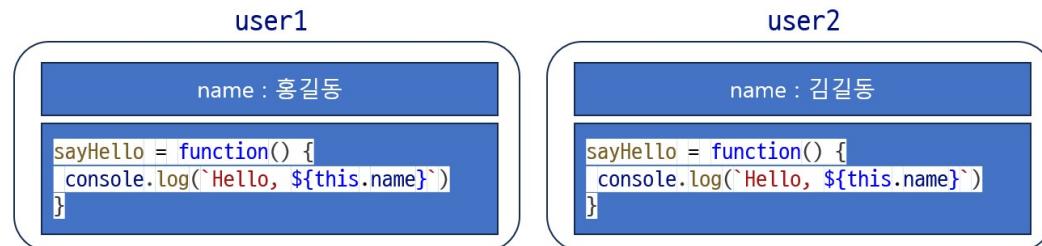
메모리 효율성

- 프로토타입을 이용하게 되면 메모리 효율성 측면에서도 이점을 얻을 수 있습니다.
- 주로 이 경우는 프로퍼티보다는 함수를 선언하는 경우
- 동일 타입의 객체가 여러 개 생성되는 경우 각 객체에 등록되는 함수는 거의 대부분 로직이 동일한 것이 일반적입니다.
- 함수를 `this` 에 선언하게 되면 동일한 동작을 하는 함수가 객체별로 매번 메모리에 할당되어 메모리가 불필요하게 점유되는 상황

프로토타입

메모리 효율성

```
function User(name){  
    this.name = name  
    this.sayHello = function() {  
        console.log(`Hello, ${this.name}`)  
    }  
}  
  
let user1 = new User('홍길동')  
let user2 = new User('김길동')  
  
user1.sayHello() // Hello, 홍길동  
user2.sayHello() // Hello, 김길동  
  
console.log(user1.sayHello == user2.sayHello) // false
```



프로토타입

메모리 효율성

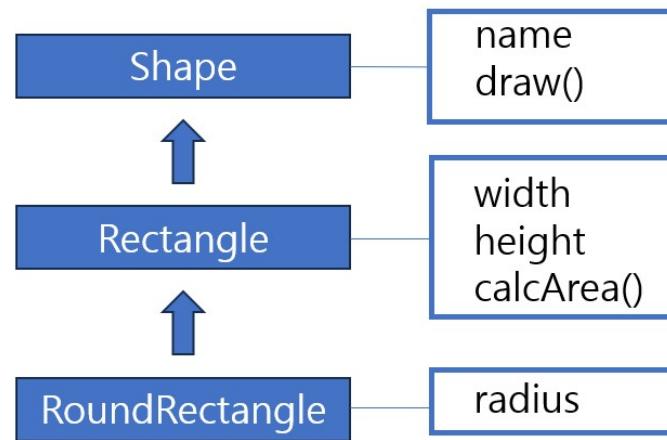
```
function User(name){  
    this.name = name  
    User.prototype.sayHello = function() {  
        console.log(`Hello, ${this.name}`)  
    }  
  
    let user1 = new User('홍길동')  
    let user2 = new User('김길동')  
    user1.sayHello() // Hello, 홍길동  
    user2.sayHello() // Hello, 김길동  
  
    console.log(user1.sayHello == user2.sayHello) // true
```



프로토타입

상속 구현

- 자바스크립트에서 상속을 이용한 코드의 재사용도 프로토타입을 이용해 구현할 수 있습니다.



프로토타입

상속 구현 - 상위 객체를 하위 프로토타입으로 지정

- 프로토타입도 객체입니다. 상위 함수의 객체를 하위 함수의 프로토타입으로 지정해서 상위에 선언된 것을 하위에서 이용하게 할 수 있습니다.

```
function Shape(name){ this.name = name }
Shape.prototype.draw = function(){
  console.log(`#${this.name} 도형을 그립니다.`)
}

function Rectangle(name, width, height){
  //부모의 this 를 자식의 this 에 바인딩(복제)
  Shape.apply(this, [name])
  this.width = width
  this.height = height
}
```

```
Rectangle.prototype = new Shape()
Rectangle.prototype.calcArea = function(){
  console.log(`area : ${this.width} * ${this.height} = ${this.width * this.height}`)
}

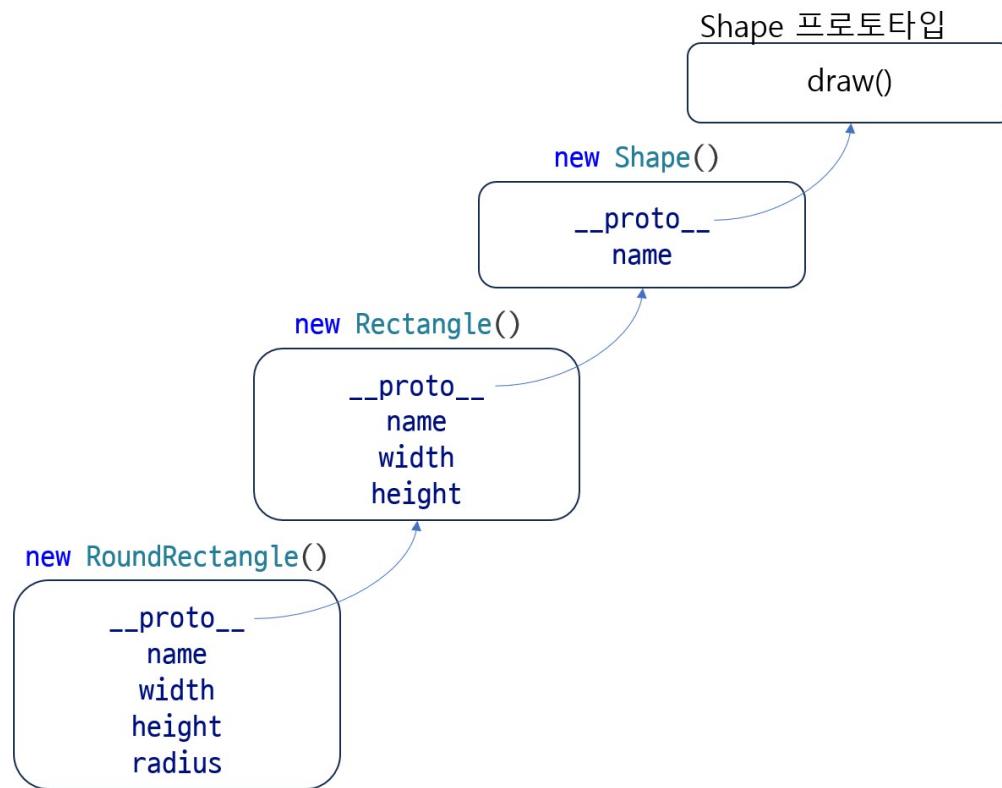
let rect = new Rectangle('사각형1', 10, 20)

function RoundRectangle(name, width, height, radius){
  Rectangle.apply(this, [name, width, height])
  this.radius = radius
}

RoundRectangle.prototype = new Rectangle()
```

프로토타입

상속 구현 - 상위 객체를 하위 프로토타입으로 지정



프로토타입

상속 구현 - 상위의 프로토타입을 하위 프로토타입으로 지정

- 상위 객체를 하위 프로토타입으로 지정은 상위 객체가 생성되어야 합니다. 불 필요하게 객체가 생성될 수도 있는 방법입니다.
- 상위의 프로토타입을 그대로 하위에서 프로토타입으로 이용하게 할 수도 있습니다.

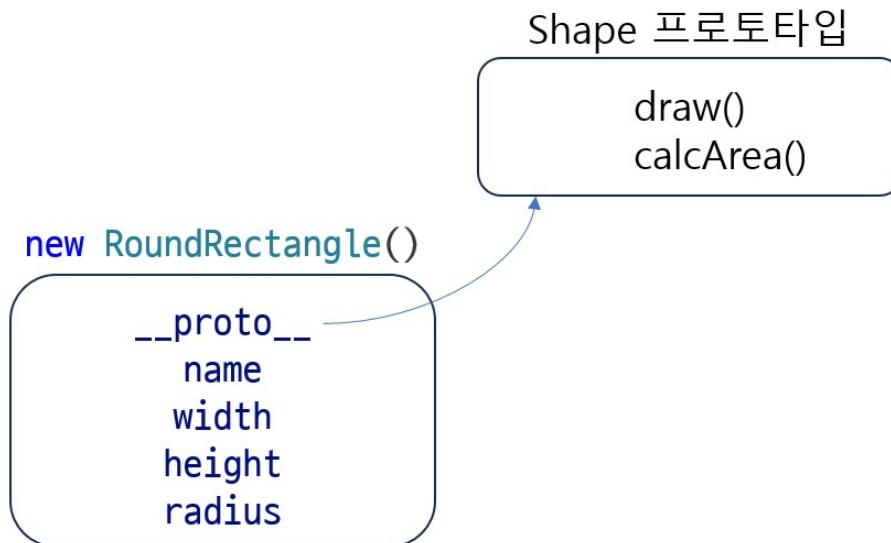
```
function Shape(name){  
    this.name = name  
}  
  
Shape.prototype.draw = function(){  
    console.log(` ${this.name} 도형을 그립니다.`)  
}  
  
  
function Rectangle(name, width, height){  
    Shape.apply(this, [name])  
    this.width = width  
    this.height = height  
}
```

```
Rectangle.prototype = Shape.prototype  
  
Rectangle.prototype.calcArea = function(){  
    console.log(`area : ${this.width} * ${this.height} =  
    ${this.width * this.height}`)  
}  
  
let rect = new Rectangle('사각형1', 10, 20)  
  
  
function RoundRectangle(name, width, height, radius){  
    Rectangle.apply(this, [name, width, height])  
    this.radius = radius  
}  
  
RoundRectangle.prototype = Rectangle.prototype
```

프로토타입

상속 구현 - 상위의 프로토타입을 하위 프로토타입으로 지정

- Shape 객체를 생성하든, Rectangle 객체를 생성하든, 아니면 RoundRectangle 객체를 생성하든 객체 생성을 위한 프로토타입은 동일한 프로토타입을 이용



다양한 객체 이용 기법 - typeof

- typeof 연산자는 타입을 확인하기 위한 연산자

```
function User(){}
let user1 = new User()

console.log(typeof 10)//number
console.log(typeof "hello")//string
console.log(typeof true)//boolean
console.log(typeof User)//function
console.log(typeof [10, 20])//object
console.log(typeof user1)//object
```

다양한 객체 이용 기법 - instanceof

- instanceof 는 객체의 타입이 특정 타입인지를 판단하기 위한 연산자
- 왼쪽의 객체가 오른쪽의 생성자로 만들어진 객체인지를 판단하는 연산자

```
console.log(10 instanceof Number)//false  
console.log("hello" instanceof String)//false  
console.log(true instanceof Boolean)//false
```

```
console.log(new Number(10) instanceof Number)//true  
console.log(new String("hello") instanceof String)//true  
console.log(new Boolean(true) instanceof Boolean)//true  
console.log(new Number(10) instanceof String)//false
```

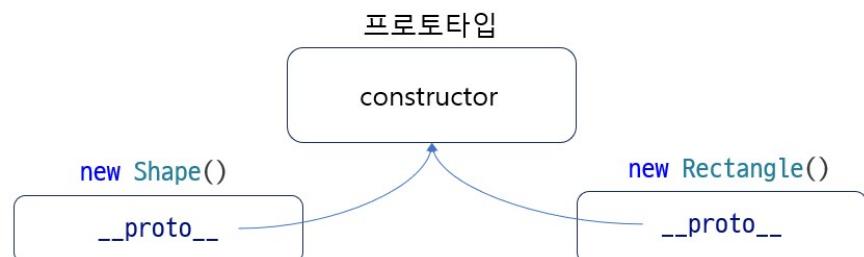
다양한 객체 이용 기법 - instanceof

- 다른 함수의 프로토타입을 그대로 자신의 프로토타입으로 지정하는 경우

```
function Shape(){}
function Rectangle(){}
Rectangle.prototype = Shape.prototype
```

```
let shape1 = new Shape()
let rect1 = new Rectangle()
```

```
console.log(shape1 instanceof Shape)//true
console.log(shape1 instanceof Rectangle)//true
console.log(rect1 instanceof Shape)//true
console.log(rect1 instanceof Rectangle)//true
```



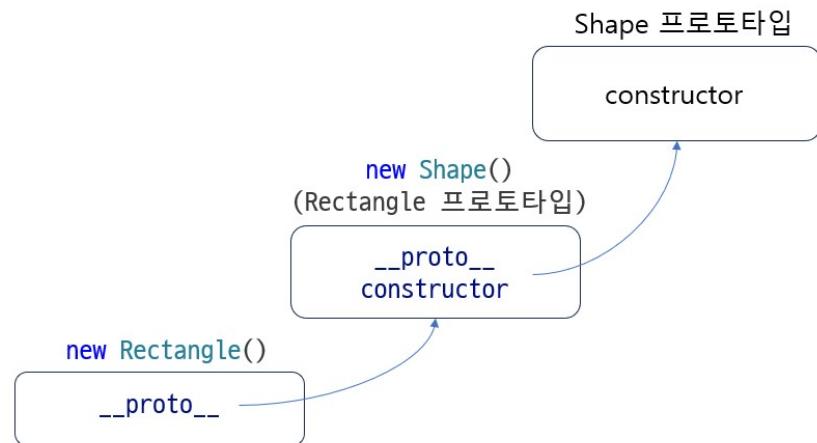
다양한 객체 이용 기법 - instanceof

- 상위 객체를 생성해서 하위 프로토타입으로 지정하는 경우

```
function Shape(){}
function Rectangle(){}
Rectangle.prototype = new Shape()

let shape1 = new Shape()
let rect1 = new Rectangle()

console.log(shape1 instanceof Shape)//true
console.log(shape1 instanceof Rectangle)//false
console.log(rect1 instanceof Shape)//true
console.log(rect1 instanceof Rectangle)//true
```



다양한 객체 이용 기법 – 프로퍼티 Descriptor

프로퍼티 Descriptor

- 객체에 프로퍼티에는 설명자(Descriptor)라는 정보가 있습니다.
- 설명자를 이용해 객체의 프로퍼티 값이 변경되지 않게 하거나 열거로 사용되지 않게 하는 등 원하는데로 사용되게 할 수 있습니다.
- 객체를 선언하면서 프로퍼티의 설명자를 따로 지정하지 않는다고 하더라도 기본으로 프로퍼티에 설명자가 추가되게 됩니다.

다양한 객체 이용 기법 - 프로퍼티 Descriptor

- 프로퍼티의 설명자를 확인하려면 Object.getOwnPropertyDescriptor() 를 이용
 - value : 프로퍼티에 대입된 값
 - writable: 프로퍼티 값을 수정할 수 있는지에 대한 여부
 - enumerable : 프로퍼티가 열거형으로 이용이 가능한지에 대한 여부
 - configurable : 프로퍼티의 설명자를 변경할 수 있는지에 대한 여부

```
let obj = {
    name: '홍길동',
    age: 10
}
console.log(Object.getOwnPropertyDescriptor(obj, 'name'))
//{value: '홍길동', writable: true, enumerable: true, configurable: true}
```

다양한 객체 이용 기법 – 프로퍼티 Descriptor

- 프로퍼티의 설명자 값을 변경하고 싶다면 `Object.defineProperty()` 를 이용
- `writable` 은 프로퍼티의 값을 변경하지 못하게 할 때 이용
- 객체를 선언할 때 지정한 값으로만 사용되어야 하거나 특정 업무가 진행되는 동안 값 변경이 불가능하게 하고자 할 때 이용

```
Object.defineProperty(obj, 'age', {writable: false})
```

다양한 객체 이용 기법 – 프로퍼티 Descriptor

- enumerable 은 프로퍼티가 열거로 이용이 가능한지를 설정
- 어떤 객체에 등록된 프로퍼티들을 순서대로 나열해서 사용하는 것을 의미하며 특정 프로퍼티가 이 열거로 이용되지 않게 하고자 할 때 enumerable 을 false 로 지정

```
let obj = {
    name: '홍길동',
    age: 10,
    address: 'seoul'
}
Object.defineProperty(obj, 'age', {enumerable: false})
console.log(obj.age)//10
console.log(Object.keys(obj))//['name', 'address']
console.log(Object.values(obj))//['홍길동', 'seoul']
console.log(Object.entries(obj))
//0:['name', '홍길동']
//1:['address', 'seoul']
```

다양한 객체 이용 기법 – 프로퍼티 Descriptor

- configurable 은 프로퍼티의 설명자를 재설정 할 수 있는지에 대한 정보
- 어떤 프로퍼티의 값 변경을 못하게 하기 위해서 writable:false 로 했다고 하더라도 누군가가 다시 Object.defineProperty() 로 writable: true 로 설정해 놓고 프로퍼티의 값을 변경할 수도 있습니다. 이 경우 configurable: false 로 지정해서 설명자를 변경하지 못하게 해주어야 합니다.

다양한 객체 이용 기법 – Object.create()

- 자바스크립트의 모든 객체는 그 객체를 만드는 프로토타입이 있어야 합니다.
- 객체 리터럴로 객체를 생성하는 것은 new Object() 방법으로 객체를 생성하는 것과 동일
- 객체 리터럴로 만든 객체의 프로토타입은 Object 의 프로토타입
- 경우에 따라 객체를 생성하면서 Object의 프로토타입이 아닌 다른 프로토타입을 지정
- 상속 개념을 적용해 객체 리터럴을 만들때 주로 사용

```
function Shape(name){  
    this.name = name  
}  
  
Shape.prototype.draw = function(){  
    console.log(` ${this.name} 을 그립니다.`)  
}
```

```
let rect1 = Object.create(Shape.prototype, {  
    name: {value: 'rect1'},  
    width: {value: 10},  
    height: {value: 10}  
})  
  
rect1.draw() // rect1 을 그립니다.  
console.log(rect1) // Shape {name: 'rect1', width: 10, height: 10}
```

다양한 객체 이용 기법 – this 동적 바인딩

- 동적 바인딩이란 실행시점에 함수의 this 를 지정할 수 있다는 의미
- 동적 바인딩 기법을 이용해 함수의 this 역할을 하는 객체를 바꿔서 이용할 수 있습니다.

bind()

- bind() 함수는 함수에 this 역할을 하는 객체를 바인딩하여 새로운 함수를 반환하는 역할

```
let sayHello = function(){
    console.log(`Hello, ${this.name}`)//error
}

sayHello()

let obj = {
    name: '홍길동'
}

let sayHello = function(){
    console.log(`Hello, ${this.name}`)//Hello, 홍길동
}

let newSayHello = sayHello.bind(obj)

newSayHello()
```

다양한 객체 이용 기법 – this 동적 바인딩

- bind 되는 함수에 매개변수를 지정하고 새로 만들어지는 함수를 호출하면서 매개변수 값을 대입할 수 있습니다.

```
let obj = {
    name: '홍길동'
}
let sayHello = function(arg1, arg2){
    console.log(`Hello, ${this.name}, ${arg1}, ${arg2}`)//Hello, 홍길동, 30, 40
}
let newSayHello = sayHello.bind(obj)
newSayHello(30, 40)
```

다양한 객체 이용 기법 – this 동적 바인딩

- bind()로 함수에 객체를 바인딩 시키면서 매개변수 값을 지정 가능
- 이렇게 되면 바인딩 시점의 매개변수와 호출 시점의 매개변수를 모두 이용 가능

```
let obj = {  
    name: '홍길동'  
}  
  
let sayHello = function(...args){  
    console.log(`Hello, ${this.name}, ${args}`)//Hello, 홍길동, 10,20,30,40  
}  
  
let newSayHello = sayHello.bind(obj, 10, 20)  
newSayHello(30, 40)
```

다양한 객체 이용 기법 – this 동적 바인딩

call(), apply()

- bind()는 새로운 함수를 만드는 역할이지 함수를 호출하는 역할은 아닙니다.
- call(), apply() 를 이용하면 함수에 객체를 바인딩 시키고 그 함수를 호출까지 해줍니다.
- call(), apply() 의 반환 값은 새로운 함수가 아니라 함수를 호출한 결과 값

```
let obj = {  
    name: '홍길동'  
}  
  
let sayHello = function(){  
    console.log(`Hello, ${this.name}`)//Hello, 홍길동  
    return 100  
}  
  
console.log(sayHello.call(obj))//100
```

다양한 객체 이용 기법 – this 동적 바인딩

call(), apply()

- apply() 함수가 call() 과 차이가 있는 것은 함수를 호출하면서 전달하는 매개변수 값을 지정하는 방법입니다.
- apply() 는 전달하는 매개변수를 배열로 지정

```
console.log(sayHello.call(obj, 10, 20)) //100
```

```
console.log(sayHello.apply(obj, [10, 20]))
```

다양한 객체 이용 기법 – getter/setter

- 외부에서는 이 함수를 변수처럼 이용한다는 개념입니다.
- 이런 함수들을 흔히 getter/setter 함수라고 부릅니다.
- 자바스크립트에서 프로퍼티로 이용되는 getter/setter 함수를 만들기 위해서는 get, set이라는 예약어로 함수가 선언되어 있어야 합니다.
- get 예약어로 선언된 함수를 getter라고 부르며 프로퍼티 값을 참조할 때 호출이 됩니다.
- set 예약어로 선언된 함수는 매개변수를 가지고 있어야 하며 이 프로퍼티 값을 변경할 때 호출이 됩니다.

```
let obj = {  
    _num: 0,  
    get num() {  
        return this._num  
    },  
    set num(value){  
        this._num = value  
    }  
}  
  
obj.num = 10  
console.log(obj.num)//10
```

다양한 객체 이용 기법 – getter/setter

- get 함수만 선언할 수도 있습니다. 이렇게 되면 함수를 프로퍼티로 활용할 수 있지만 get 만 있는 프로퍼티가 됨으로 값 참조만 되고 변경은 불가능한 프로퍼티가 됩니다.
- set 만 선언된 함수를 만들 수도 있습니다. 이렇게 되면 값 대입만 되는 프로퍼티가 됩니다.

```
let obj = {
  _num: 0,
  set num(value) {
    this._num = value
  }
}

obj.num = 10
console.log(obj.num)//undefined
```

클래스

- 생성자 함수를 이용하는 방법은 자바스크립트 초기부터 제공하던 전통적인 방법
- 클래스를 이용하는 방법은 ECMA2015부터 추가된 방법
- 클래스는 class 라는 예약어로 선언되며 이 클래스내에 객체의 멤버인 프로퍼티와 함수를 선언
- 객체를 생성하기 위해서는 new 연산자를 이용

```
class User {  
    name = '홍길동'  
    sayHello(){  
        console.log(`Hello ${this.name}`)  
    }  
}  
  
let obj = new User()  
obj.sayHello()
```

클래스 - 생성자

- 생성자는 객체 생성시에 호출되어 클래스의 프로퍼티, 메서드를 메모리에 할당하는 역할
- 객체 생성을 위해 new 연산자를 이용할 때 호출되는 것은 생성자
- 개발자가 코드에서 클래스를 선언하면서 명시적으로 생성자를 추가하지 않았다면 자동으로 기본 생성자 (Default Constructor) 가 추가
- 자바스크립트에서 클래스의 생성자는 constructor 예약어로 선언되는 함수
- 클래스내에 constructor() 로 생성자를 정의할 수 있는데 하나의 클래스내에 하나의 생성자만 추가할 수 있습니다.

```
class User {  
    name = '홍길동'  
    constructor(){ }  
    sayHello(){  
        console.log(`Hello ${this.name}`)  
    }  
}  
let obj = new User()
```

클래스 – 객체 멤버

- 클래스에 멤버를 선언할 때는 프로퍼티의 경우 생성자내에서 "this.프로퍼티명" 형태로 선언하고 메서드의 경우 생성자 밖에 클래스 영역에 선언하는 것이 일반적
- 클래스 내에 멤버를 추가할 때는 일반적으로 변수를 선언하기 위한 let, var, const 예약어와 함수를 선언하기 위한 function 예약어는 사용할 수 없습니다.

```
class User {  
    constructor(name, age) {  
        this.name = name  
        this.age = age  
    }  
    sayHello(){  
        console.log(`Hello ${this.name}, age = ${this.age}`)  
    }  
}  
  
let obj = new User('홍길동', 20)  
obj.sayHello() //Hello 홍길동, age = 20
```

클래스 – 객체 멤버

- 프로퍼티를 생성자에서 선언하지 않고 클래스 영역에서 선언 가능
- 일반 메서드 내에서 클래스 멤버를 this.xxx 처럼 선언 가능
- 객체 생성후에 클래스 외부에서 객체에 멤버를 추가 가능

```
class User {  
    name  
    constructor(name, age) {  
        this.name = name  
        this.age = age  
        this.sayHello1 = function(){  
        }  
    }  
    sayHello2(){  
        this.address = 'seoul'  
        console.log(`Hello2 ${this.name}, age = ${this.age}`)  
    }  
}  
  
let obj = new User('홍길동', 20)  
obj.phone = '0101111'
```

클래스 – 객체 멤버 – private

- 클래스에 선언된 프로퍼티, 메서드를 클래스 외부에서 이용 못하게 해야 하는 경우
- 자바스크립트에서는 일반 소프트웨어 언어에서 제공하는 접근제한자를 제공하지 않습니다.
- 클래스 내부에 선언된 멤버가 외부에서 사용되지 않게 하려면 이름을 #으로 시작하는 이름을 지정
- #으로 시작하는 이름의 멤버는 클래스 영역에 멤버가 선언될 때만 사용할 수 있습니다. 생성자내에서 멤버를 선언할때는 사용할 수 없습니다.

```
class User {  
    #name  
    age  
    constructor(name, age){  
        this.#name = name  
        this.age = age  
    }  
    #myFun(){ console.log('myFun call....') }  
    sayHello(){  
        console.log(`Hello ${this.#name}, age = ${this.age}`)  
        this.#myFun()  
    }  
}  
  
let obj = new User('홍길동', 20)  
// obj.#name = '김길동'//error  
obj.age = 30  
obj.sayHello()//Hello 김길동, age = 30  
// obj.#myFun()//error
```

클래스 – static

- 클래스에 선언된 멤버는 객체 멤버와 static 멤버로 구분



- static 멤버는 static 예약어로 선언된 프로퍼티, 메서드이며 객체 멤버는 static 예약어로 선언되지 않은 프로퍼티, 메서드
- static 멤버는 클래스내에 선언되기는 하지만 객체로 이용되기 위한 멤버가 아닙니다. 그럼으로 static 멤버는 객체 생성 없이 클래스명으로 이용이 가능합니다.

클래스 – static

```
class MyClass {  
    data1 = 10  
    static data2 = 20  
  
    myFun1() {  
        console.log('myFun1 call..')  
    }  
    static myFun2() {  
        console.log('myFun2 call..')  
    }  
}  
  
MyClass.myFun2() //myFun2 call..  
console.log(MyClass.data2) //20  
  
// MyClass.myFun1() //error  
console.log(MyClass.data1) //undefined  
  
let obj = new MyClass()  
obj.myFun1() //myFun1 call..  
console.log(obj.data1) //10  
  
// obj.myFun2() //error  
console.log(obj.data2) //undefined
```

클래스 - 상속

- extends 예약어로 다른 클래스를 상속받아 작성

```
class Shape {  
    name = '도형'  
    x = 0  
    y = 0  
    draw(){  
        console.log(` ${this.name} 을 ${this.x}, ${this.y} 에 그립니다.`)  
    }  
}  
class Rect extends Shape{  
    width = 0  
    height = 0  
}  
  
let obj = new Rect()  
obj.name = '사각형1'  
obj.x = 10  
obj.y = 10  
obj.width = 30  
obj.height = 30  
obj.draw() //사각형1을 10, 10에 그립니다.
```

클래스 - 상속 - super

- super 는 상위 클래스를 지칭하는 예약어

super 로 상위 생성자 호출

- 하위 클래스 생성자가 호출될 때 상위 클래스의 생성자가 호출되지 않는 경우는 존재할 수 없습니다.
- super() 로 상위의 생성자를 호출하는 구문은 생성자 내에서 가장 첫 줄에 한번만 작성 가능

```
class Shape {  
    constructor(){  
  
    }  
}  
  
class Rect extends Shape {  
    constructor(){  
  
    }  
}  
  
let obj = new Rect() //error
```

```
class Shape {  
    constructor(){  
  
    }  
}  
  
class Rect extends Shape {  
    constructor(){  
        super()  
    }  
}  
  
let obj = new Rect()
```

클래스 - 상속 - super

super로 상위 멤버 이용

- 경우에 따라 상위에 선언된 멤버를 하위에서 동일이름으로 다시 정의하는 경우가 있습니다. 이를 용어로 오버라이딩(Overriding) 이라고 합니다.
- 필요에 의해 상위에 선언된 동일이름의 멤버를 이용해야 하는 경우가 있습니다. 이때 super라는 예약어로 명시적으로 상위의 멤버를 지칭하게 됩니다.

```
class Shape {  
    constructor(name, x, y){  
        this.name = name  
        this.x = x  
        this.y = y  
    }  
    calcArea(){  
        console.log(` ${this.name}의 면적을 계산합니다.`)  
    }  
}
```

```
class Rect extends Shape {  
    constructor(name, x, y, width, height){  
        super(name, x, y)  
        this.width = width  
        this.height = height  
    }  
    calcArea(){  
        super.calcArea()  
        console.log(`면적은 ${this.width * this.height} 입니다.`)  
    }  
}
```

클로저

- 클로저(Closure)란 함수와 함수가 선언되었을 때의 렉시컬 환경(Lexical environment)의 조합을 의미합니다.
- 자바스크립트 뿐만 아니라 함수를 객체로 사용하는 대부분의 소프트웨어 언어에서 제공되는 개념
- 클로저는 자바스크립트에서 함수를 이용하기 위해서 자동으로 제공되는 개념이며 클로저를 위해 개발자가 어떤 코드적인 프로그램을 작성해야 하는 것은 아닙니다.
- 함수의 실행 컨텍스트와 렉시컬 환경을 구분해서 정리 필요

클로저

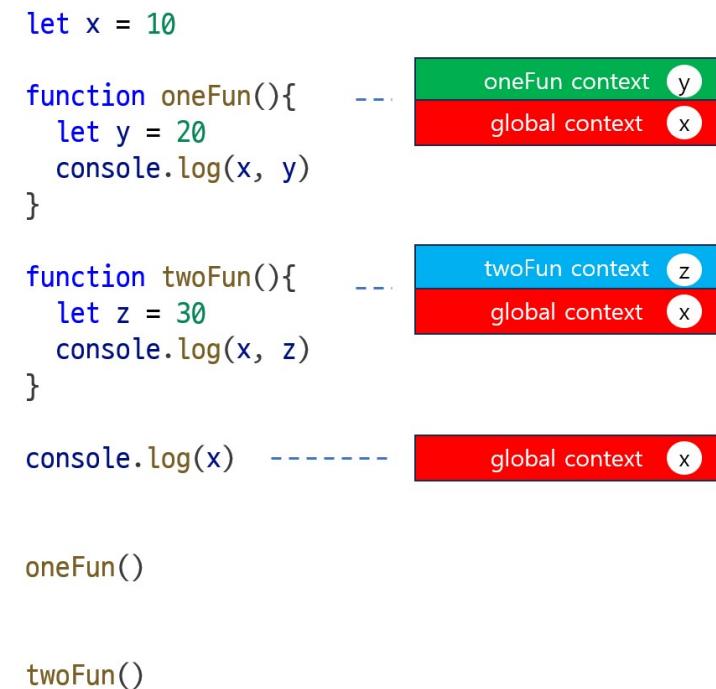
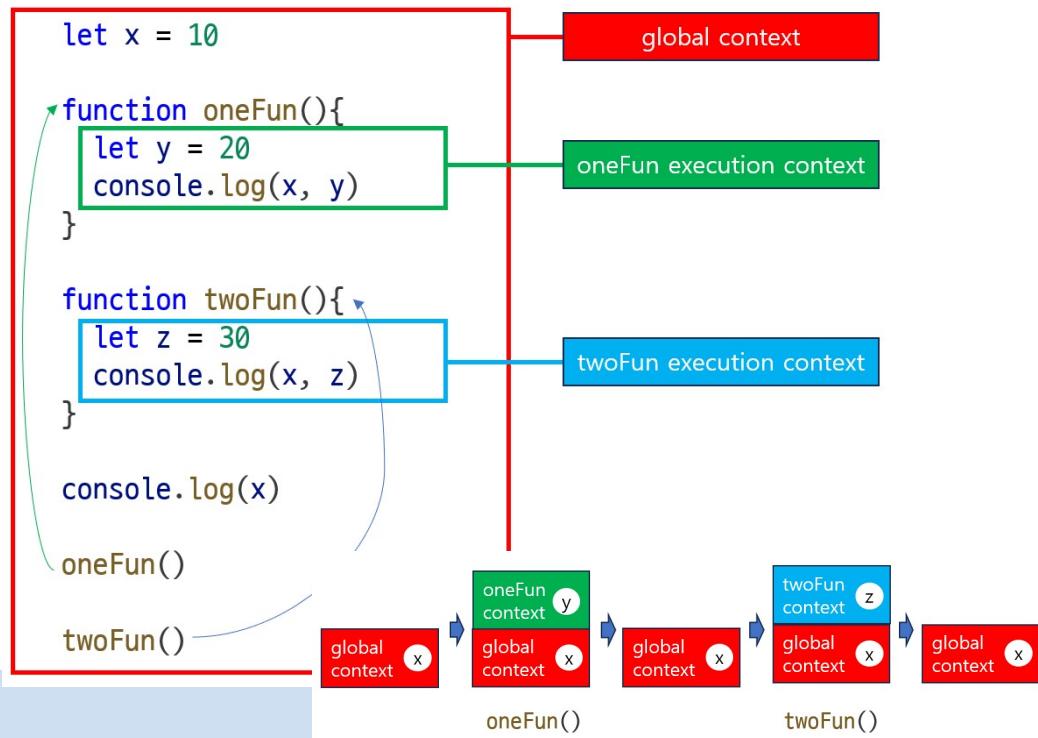
실행 컨텍스트

- 실행 컨텍스트란 함수의 실행 환경이며 함수가 실행되기 위한 정보
- 함수가 호출이 되면 자동으로 그 함수를 위한 실행 컨텍스트가 만들어지고 이 실행 컨텍스트내에 그 함수를 위한 매개변수 값, 로컬 변수등이 저장
- 함수내에서 매개변수를 참조하거나 로컬의 변수를 참조할 때 자동으로 함수의 실행 컨텍스트내에 등록된 것들이 이용되는 구조

클로저

실행 컨텍스트

- 자바스크립트 코드가 실행되면 자동으로 Global Context 가 만들어지고 그 곳에 전역 멤버가 등록
- 실행 컨텍스트는 함수 호출 순서대로 스택 구조로 유지



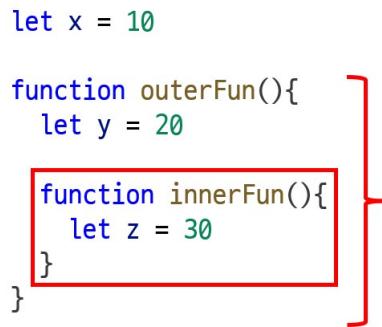
클로저

렉시컬 환경이란

- 렉시컬(Lexical) 이란 영어 단어 뜻은 '허휘', '구문'이라는 뜻
- 렉시컬 환경 혹은 렉시컬 스코프라고 하는데 이 의미는 함수가 선언된 위치를 의미
- 함수가 실행되는 동적인 환경이 아닌 코드로 함수를 작성한 위치를 의미

```
let x = 10

function outerFun(){
  let y = 20
  function innerFun(){
    let z = 30
  }
}
```



- innerFun() 을 outerFun() 함수 내에 선언했음으로 innerFun() 의 렉시컬 환경은 outerFun() 입니다.

클로저

클로저가 왜 필요한가?

- 함수를 선언하고 그 함수를 호출해서 실행시키는 모든 상황에서 클로저가 만들어지지 않습니다.
- 클로저가 필요한 경우는 함수를 호출하는 곳이 함수가 선언된 렉시컬 환경 내부가 아닌 경우입니다.

```
let x = 10

function outerFun(){
    let y = 20
    function innerFun(){
        let z = 30
        console.log(x, y, z)
    }
    innerFun()
}

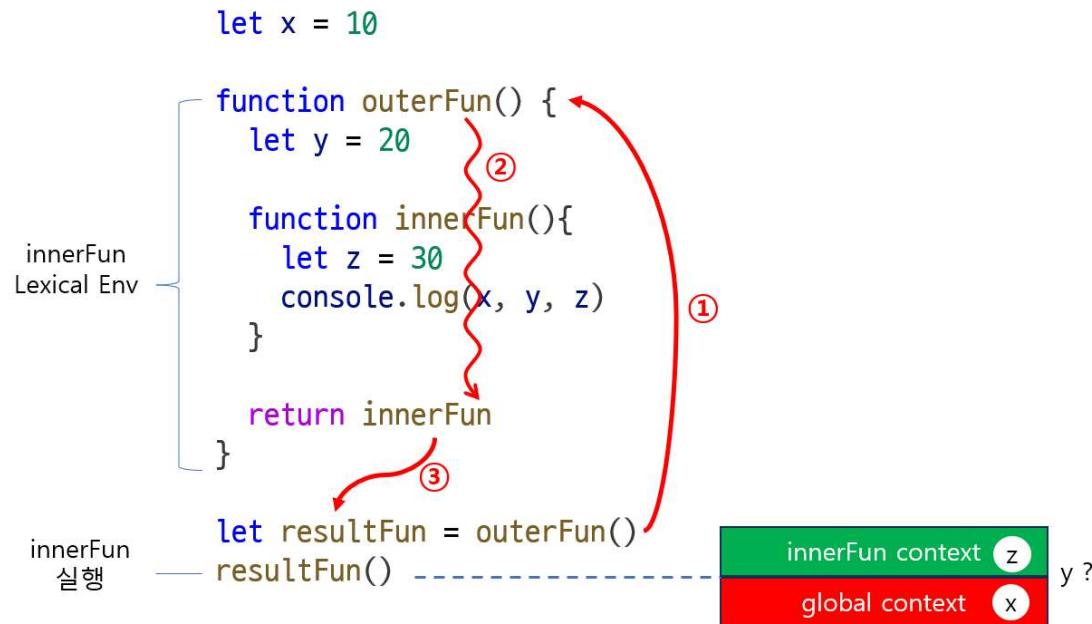
outerFun()
```



- 위의 경우는 함수의 실행이 자신이 선언된 렉시컬 환경내에서 실행된 경우입니다.
- 이 경우에는 클로저는 필요가 없습니다.

클로저

클로저가 왜 필요한가?

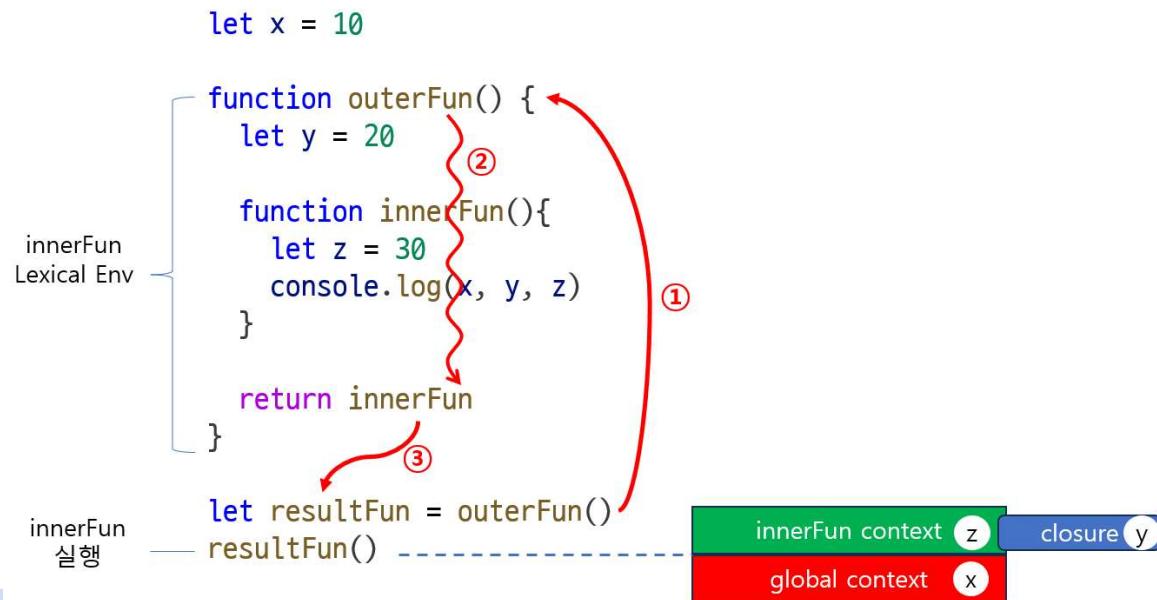


- 예에서 중요한 점은 `innerFun` 함수가 자신이 선언된 렙시컬 환경(`outerFun`)에서 실행되지 않고 외부(`outerFun` 밖)에서 실행

클로저

클로저가 왜 필요한가?

- 클로저(Closure)란 함수와 함수가 선언되었을 때의 렙시컬 환경(Lexical environment)의 조합
- 함수가 선언된 위치 외부에 전전달되 실행될 때 자신이 선언된 렙시컬 환경의 정보를 이용할 수 있게 클로저가 자동으로 만들어져서 함수에 추가되게 됩니다.



클로저 - 캡슐화

- 클로저를 활용해 구현하는 것 중 하나가 캡슐화입니다.

```
function User() {  
    this.name = '홍길동'  
}  
  
let user1 = new User()  
console.log(user1.name)//홍길동  
user1.name = '김길동'  
console.log(user1.name)//김길동
```

- 생성자 함수 외부에서 name값을 획득하거나 수정할 수 있는 하지만 name 프로퍼티를 직접 이용하지 못하게 하려면 어떻게 해야 할까요?

클로저 - 캡슐화

```
function User() {  
    let name = '홍길동'  
    this.getName = function(){  
        return name  
    }  
    this.setName = function(value){  
        name = value  
    }  
}  
  
let user1 = new User()  
console.log(user1.getName())//홍길동  
user1.setName('김길동')  
console.log(user1.getName())//김길동
```

클로저 - 커링

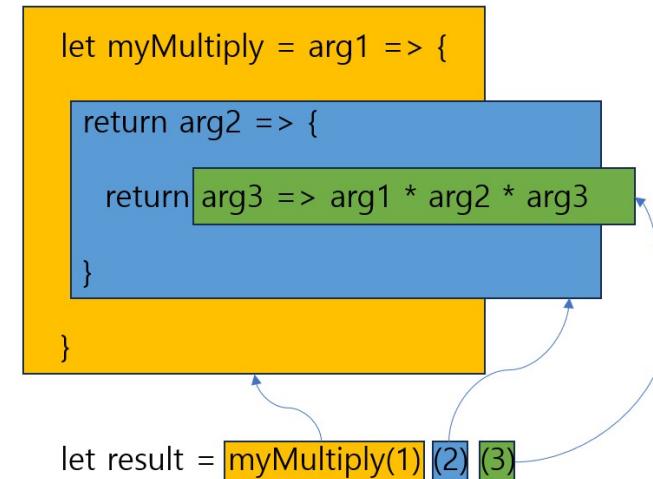
- 클로저가 활용되는 사례의 하나로 커링이 있습니다.
- 커링이란 매개변수를 여러 개 가지는 하나의 함수를 각각의 매개변수를 가지는 여러 개의 함수로 나누어 개발하는 함수형 프로그래밍 기법을 지칭하는 용어입니다.
- 커링은 함수와 관련된 기법이며 함수를 객체로 활용하도록 지원하는 소프트웨어 언어에서 대부분 제공하는 기법입니다.

```
function myMultiply(arg1, arg2, arg3){  
    return arg1 * arg2 * arg3  
}  
  
let result = myMultiply(1, 2, 3)  
console.log(result)//6
```

- 이 함수를 커링 기법을 적용해 여러 함수로 분리시켜서 동일하게 동작하도록 작성하고자 한다면?

클로저 - 커링

```
let myMultiply = arg1 => {
    return arg2 => {
        return arg3 => arg1 * arg2 * arg3
    }
}
let result = myMultiply(1)(2)(3)
console.log(result)//6
```



- 함수가 분리 되어 있다면 이 함수들은 한꺼번에 호출하지 않고 따로 호출이 가능해 집니다.

```
let fun2 = myMultiply(1)
let fun3 = fun2(2)
let result = fun3(3)
console.log(result)//6
```

비동기 – setTimeout(), setInterval()

- 함수 호출을 예약해야 하는 경우
- 1초 후에 함수를 호출해야 하거나 1초마다 반복적으로 함수를 호출해야 하는 경우가 있는데 이를 "호출 스케줄링"이라고 하며 호출 스케줄링을 위해 setTimeout() 과 setInterval() 을 제공합니다.
- setTimeout() : 일정 시간이 지난 후 함수 실행
- setInterval() : 일정 시간 간격으로 반복적 함수 실행

비동기 – setTimeout(), setInterval()

setTimeout()

- setTimeout() 은 시간을 지정하고 그 시간이 지난 후 함수를 실행시키기 위해 사용됩니다.
- setTimeout(functionRef, delay, param1, param2, /* ..., */ paramN)
- 자연 시간을 설정하지 않으면 즉시 sayHello() 함수가 호출

```
function sayHello(){  
    console.timeEnd()  
    console.log('Hello')  
}  
  
console.time()  
setTimeout(sayHello, 1000)  
//default: 1007.23291015625 ms  
//Hello
```

비동기 – setTimeout(), setInterval()

setTimeout()

- setTimeout() 으로 함수를 호출하면서 매개변수 값을 전달해야 하는 경우가 있는데 이 전달할 값은 setTimeout() 의 세번째 매개변수부터 나열

```
function sayHello(arg1, arg2, arg3){  
    console.log(`Hello, ${arg1}, ${arg2}, ${arg3}`)//Hello, 흥길동, 10, true  
}  
  
setTimeout(sayHello, 1000, '흥길동', 10, true)
```

- 함수 호출 예약 취소는 clearTimeout() 을 이용

```
function sayHello(){  
    console.log('Hello')  
}  
  
let id = setTimeout(sayHello, 1000)  
clearTimeout(id)
```

비동기 – setTimeout(), setInterval()

setInterval()

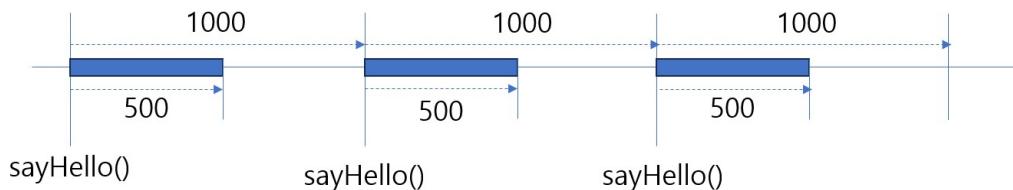
- setInterval()은 반복적으로 함수 호출을 예약
- setInterval()으로 예약된 함수 호출을 취소하고 싶다면 clearInterval()을 이용

```
function sayHello(name){  
    console.log(`Hello ${name}`)  
}  
  
let id = setInterval(sayHello, 1000, '홍길동')  
setTimeout(() => clearInterval(id), 3000)  
//Hello 홍길동  
//Hello 홍길동  
//Hello 홍길동
```

비동기 – setTimeout(), setInterval()

setInterval()

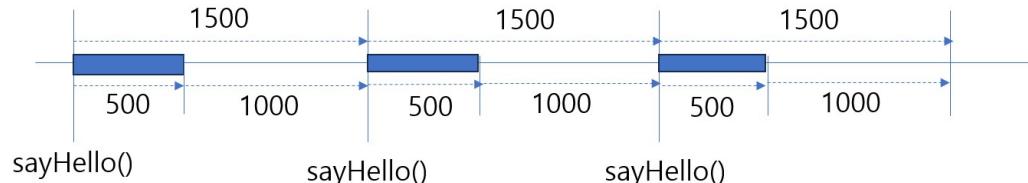
- setInterval()에서 반복 주기 시간은 함수가 호출되는 주기를 의미



- 함수가 실행이 끝난 다음 지연시간을 지정하고 이 지연시간이 지나면 다시 호출하게 하고자 한다면 setTimeout()을 중첩으로 구현

```
let sayHello = async () => {
  console.log('Hello')
  setTimeout(sayHello, 1000)
}

setTimeout(sayHello, 1000)
```



비동기 – Promise

- 프라미스는 영어 단어 뜻으로 약속이며 비동기 업무를 진행하는 곳과 그 업무의 실행 결과를 받아야 하는 곳이 상호 어떻게 연동하는지의 약속을 정의한 객체
- 비동기 업무가 진행된 곳에서 나의 업무가 끝났음을, 업무에 대한 결과가 무엇인지를 알려줄 규칙이 필요한데 이 규칙(약속)을 정의한 것이 프라미스입니다.

```
function myFun(){  
    return new Promise((resolve, reject) => {  
        setTimeout(() => resolve(10), 1000)  
    })  
}  
  
console.log('step1')  
  
let promise = myFun()  
  
promise.then(result => {  
    //step1  
    console.log(`result : ${result}`)  
    //step2  
    //result : 10  
})  
  
console.log(`step2`)
```

비동기 – Promise

프라미스 생성 – executor

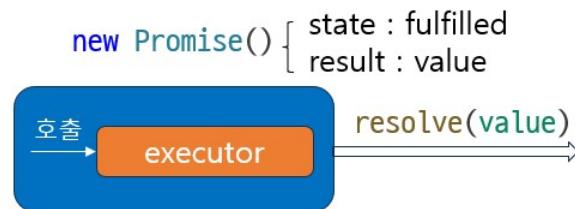
- Promise 객체 생성할 때 생성자 매개변수에 함수를 지정해야 하는데 이 함수를 executor 라고 합니다.
- 프라미스에 executor 가 지정이 되면 프라미스는 이 executor 함수를 비동기적으로 즉시 실행시켜 줍니다.
- 프라미스 객체 내부에 state 와 result 프로퍼티가 유지되며 state 는 프라미스의 상태값을, result 는 프라미스에 의해 발생된 결과값을 가집니다.
- Promise 객체를 생성하게 되면 state 값은 pending 이며 pending 은 아직 실행이 끝나지 않았음을 의미합니다. 그리고 result 는 아직 발생하지 않았음으로 undefined 입니다.



비동기 – Promise

프라미스 결과 발생 – resolve

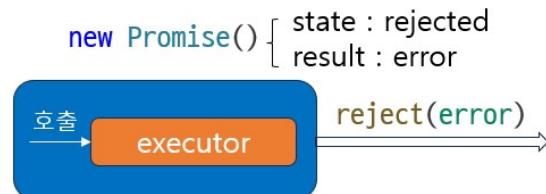
- executor 함수의 매개변수를 이용해 외부에 비동기 업무가 종료되거나 발생한 결과 데이터를 전달할 수 있습니다.
- executor 의 매개변수에 전달된 것은 함수이며 resolve 함수라고 부릅니다.
- resolve 함수를 호출하게 되면 executor 의 업무 처리가 종료된 것이며 resolve 함수 호출시 매개변수에 지정한 값이 프라미스에 의해 발행되는 결과 값입니다.
- executor 의 resolve 함수를 호출하게 되면 프라미스의 state 는 fulfilled 상태가 되며 result 는 발생한 값이 됩니다.



비동기 – Promise

프라미스 결과 발생 – reject

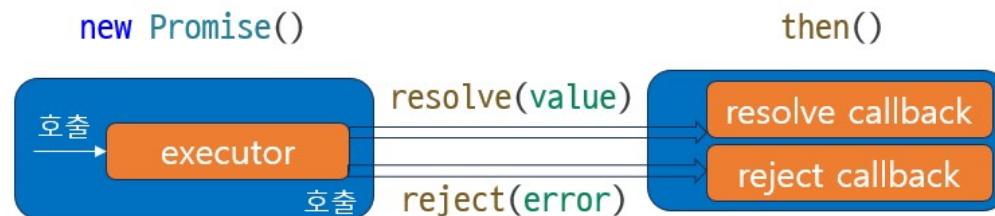
- executor 함수에서 에러가 발생했음을, 발생한 에러 정보를 외부에 발행할 필요가 있을 수도 있습니다.
- executor 의 매개변수로 전달되는 reject 함수를 이용
- reject 함수가 호출되게 되면 프라미스의 state 는 rejected 상태가 되며 result 는 발행한 에러 내용이 됩니다.



비동기 – Promise

프라미스 콜백

- 프라미스의 executor 함수에서 resolve 를 이용해 결과를 발행하거나 reject 를 이용해 에러를 발행하는데 이렇게 프라미스에서 발행한 값을 프라미스를 이용하는 외부에서 전달받아야 합니다.
- then() 의 매개변수에 콜백함수를 등록해 놓으면 프라미스에 의해 resolve 가 호출되거나 reject 이 호출되는 순간 콜백함수가 실행되게 됩니다.
- then() 함수에는 콜백 두개를 등록할 수 있는데 첫번째 매개변수로 지정한 콜백함수가 resolve 에 의해 실행되는 함수이며 두번째 매개변수로 지정한 콜백함수가 reject 에 의해 실행될 함수입니다.



비동기 – Promise

프라미스 콜백

- eject에 의해 실행될 결과를 원한다면 then()이 아닌 catch()로 등록 할 수 있습니다.
- finally()를 이용해 콜백을 등록하면 resolve, reject 모든 경우에 실행할 코드를 등록할 수도 있습니다.

```
function myFun(num){  
    return new Promise((resolve, reject) => {  
        setTimeout(() => {  
            if(num>0) {resolve(num*num)}  
            else {reject('0보다 큰수를 지정하세요')}  
        })  
    })  
}
```

```
myFun(10)  
.then((value) => console.log(`결과 데이터는 ${value} 입니다.`))  
.catch((error) => console.log(error))  
.finally(() => console.log('finally 부분이 실행됩니다.'))
```

//결과 데이터는 100입니다.
//finally 부분이 실행됩니다.

```
myFun(0)  
.then((value) => console.log(`결과 데이터는 ${value} 입니다.`))  
.catch((error) => console.log(error))  
.finally(() => console.log('finally 부분이 실행됩니다.'))
```

//0보다 큰수를 지정하세요

//finally 부분이 실행됩니다.

비동기 – async, await

- async await 는 프라미스를 대체하기 위한 개념이 아니며 프라미스를 이용하는 코드를 조금 더 간결하게 작성하기 위해서 제공되는 기법
- async await 는 단지 코드 작성 기법의 차이인 것 뿐이지 내부적으로 프라미스를 이용

```
function getData(id){  
    return new Promise((resolve, reject) => {  
        setTimeout(() => resolve(`${id} 데이터`, 1000))  
    })  
}
```

```
function myFun(){  
    getData(1)  
    .then((value) => { return getData(2) })  
    .then((value) => { return getData(3) })  
    .then((value) => { console.log(value) })  
}  
  
myFun()
```

```
async function myFun(){  
    console.log(await getData(1))  
    console.log(await getData(2))  
    console.log(await getData(3))  
}  
  
myFun()
```

비동기 – async, await

- async 는 함수 선언 부분에 사용되는 예약어
- async 로 선언된 함수는 내부적으로 Promise 를 반환
- 함수 선언에 async 만 추가 해 놓고 일반 함수에서 결과를 반환하듯 return 으로 데이터를 반환해 주면 됩니다.
- 명시적으로 Promise 를 사용하지 않았다고 하더라도 resolve(2) 로 반환한 것과 동일해 집니다.

```
function myFun1(){
    return new Promise((resolve, reject) => {
        resolve(1)
    })
}

myFun1().then((value) => console.log(value))//1
```

```
async function myFun2(){
    return 2
}
```

비동기 – async, await

- await 는 함수 내에 사용되는 예약어
- 일반 함수에서는 사용할 수 없으며 async 함수내에서만 사용이 가능
- 실행이 await 로 선언된 구문이 실행이 완료 될때까지 기다리게 하는 역할

```
async function myFun2() {  
    let data = await getData(2)  
    console.log(data)  
}
```

Ajax

- Ajax(Asynchronous Javascript + XML)
- 자바스크립트와 XML에 기반한 비동기 통신기법

XMLHttpRequest

- 서버와 비동기 통신을 하는 객체
- new XMLHttpRequest()

| 메소드 | 설명 |
|--|---|
| <code>open (method, url, async)</code> | 요청의 초기화로 요청방식, 요청 URL, 비동기 여부를 지정 method : HTTP 요청방식. GET 또는 POST값을 주로 사용 url : 요청하는 자원의 URL. async : true나 생략할 경우 비동기, false일 경우 동기방식 |
| <code>send(data)</code> | 요청 전송 data: POST 방식일 경우에 쿼리스트링으로 name1=value1&name2=value2 형태 |
| <code>abort()</code> | 요청 취소 |

Ajax

응답 처리

- 서버에 요청을 보내기 전에 XMLHttpRequest 객체의 onload 속성에 응답 받을 함수를 지정
- 응답이 도착한 후에 해당 함수가 호출

```
xhr= new XMLHttpRequest();
xhr.onload = function(){
    .....
};
xhr.open("GET", "/time", true);
xhr.send();
```

Ajax

XMLHttpRequest 속성

| 속성 | 설명 |
|--------------|--|
| readyState | 요청의 상태를 나타내는 정수 값 <ul style="list-style-type: none">- 0 : 초기화 이전 상태. open() 호출 전- 1 : 로드되지 않은 상태. send() 메서드 호출 전- 2 : 로드된 상태. send() 메서드 호출 후 응답헤더와 상태 받음- 3 : 상호작용 상태. 데이터를 받고 있는 상태- 4 : 완료 상태. 모든 데이터를 받은 상태 |
| status | 서버로부터 받은 응답의 상태를 나타내는 HTTP 상태코드 <ul style="list-style-type: none">- 200, 404, 500 등 |
| statusText | 서버로부터 받은 응답의 상태 메세지 |
| responseText | 응답으로 받은 문자열 |
| responseXML | 응답으로 받은 XML DOM 객체 |
| onload | 응답이 도착하면 발생하는 이벤트 등록 |

Ajax

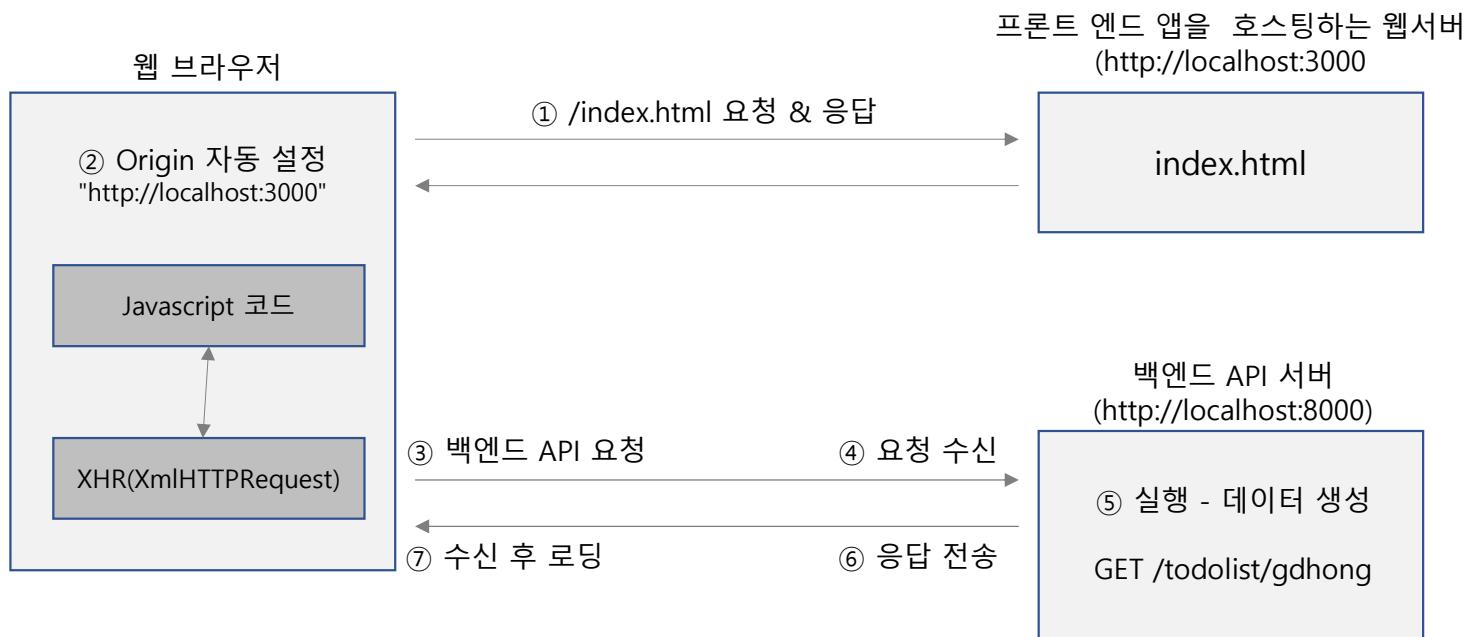
CORS – 크로스 오리진 문제



Ajax

CORS – 크로스 오리진 문제

- SOP : Same Origin Policy, 브라우저의 기본 보안 정책

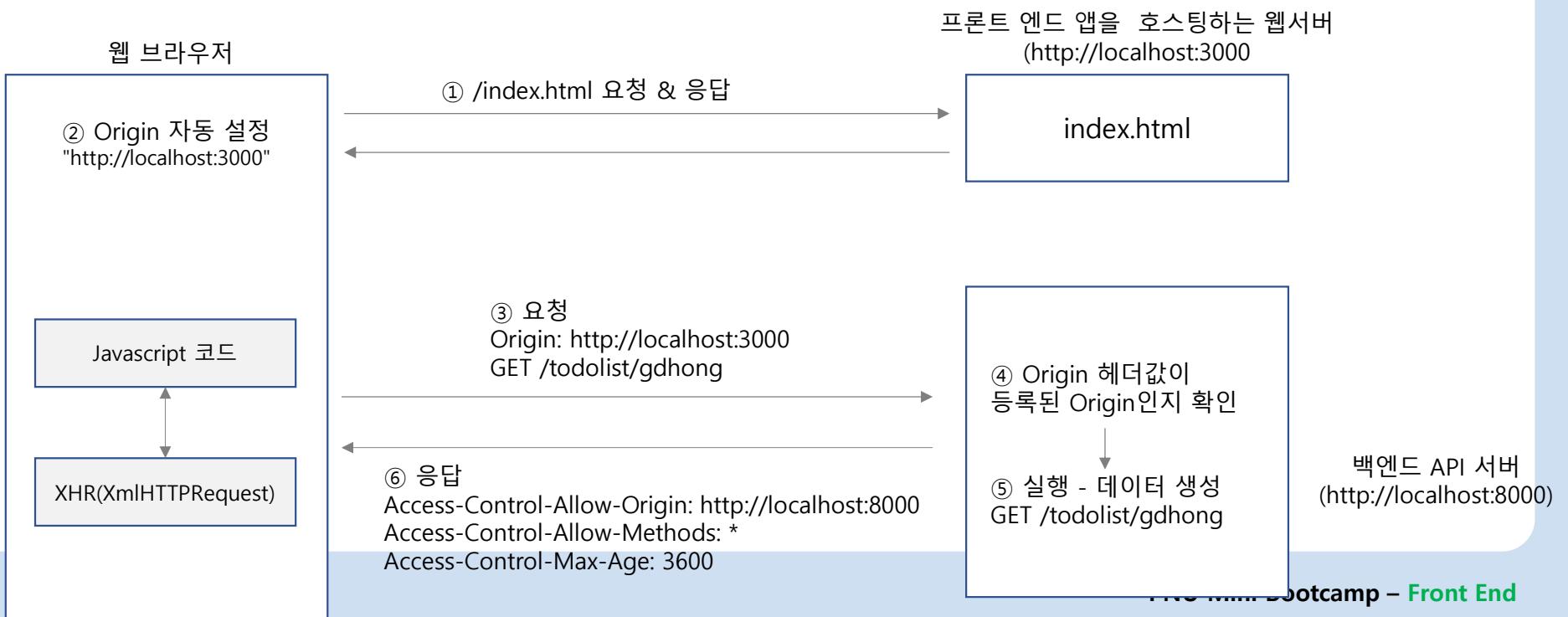


Browser의 Origin : "http://localhost:3000"
백엔드 API 서버의 Origin : "https://localhost:8000"

Ajax

크로스 오리진 문제 해결 방법

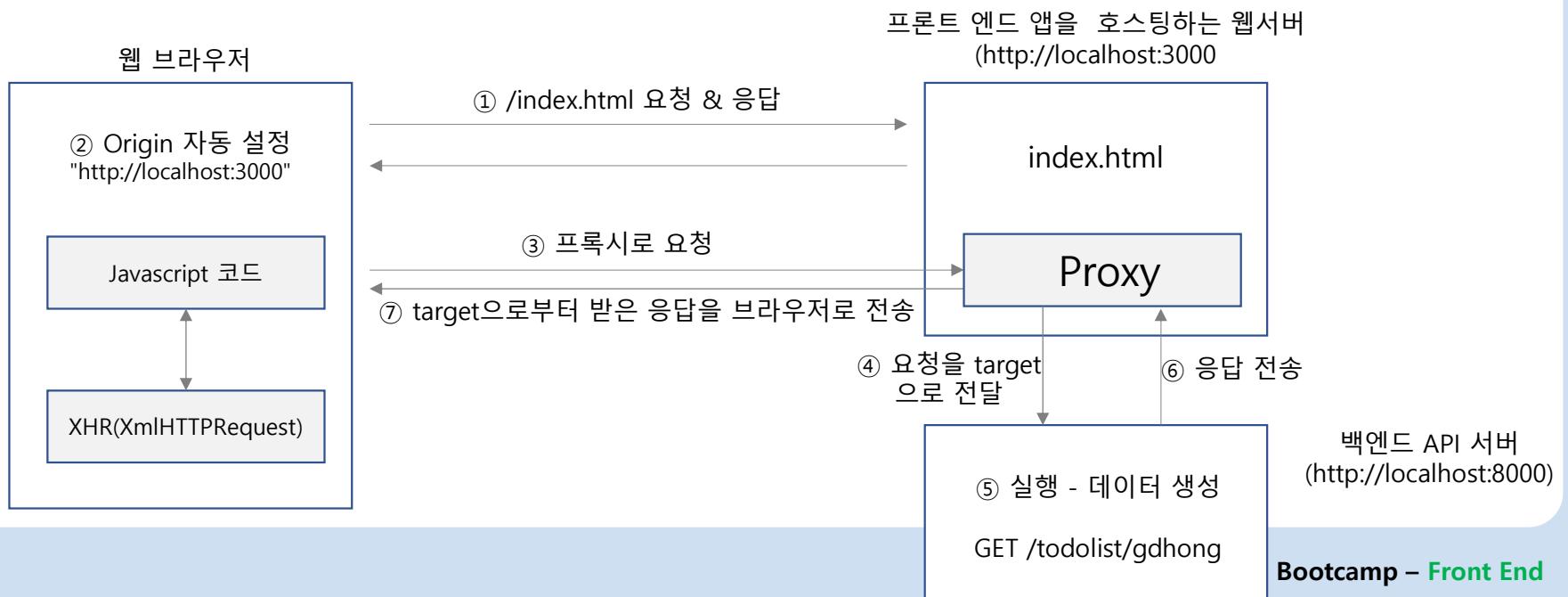
- 백엔드 API 서비스 제공자측에서 CORS기능을 제공
- Cross Origin Resource Sharing
- 서버에서 Access-Control-Allow-Origin HTTP 헤더로 브라우저의 오리진을 응답하여 브라우저가 통신 및 데이터 로딩을 할 수 있게 허용하는 방법이다.



Ajax

크로스 오리진 문제 해결 방법

- Proxy를 이용한 Cross Domain 문제 해결 방법
- 프런트엔드에서 해결하는 방법이다.
- 우리의 html을 서비스하는 오리진 서버에 proxy를 설정(혹은 설치)하는 방법이다.
- 브라우저에서 request를 오리진 서버에 넘기면 오리진 서버가 다른 서버랑 통신하는 구조이다.



Ajax

axios

- HTTP 통신 라이브러리
- axios 이외에도 jquery ajax, fetch, superagent 등이 있음
- 이 중 axios가 최근에 가장 많이 사용됨.

제공 기능

- node.js, 브라우저에서 XMLHttpRequest 객체 사용
- Promise API를 제공함
- 요청 데이터와 응답 데이터의 ContentType에 의한 자동 변환
- 요청 취소(Abort) 기능 제공
- npm install axios

Ajax

axios

[저수준 API]

```
axios(config)
```

```
axios(url, config)
```

[각 메소드별 별칭]

```
axios.get(url[, config])
```

```
axios.delete(url[, config])
```

```
axios.post(url[, data[, config]])
```

```
axios.put(url[, data[, config]])
```

```
axios.head(url[, config])
```

```
axios.options(url[, config])
```

Ajax

- axios 저수준 메서드와 get 메서드

```
axios({
  method : 'GET',
  url : '/contacts',
  params : { pageno : 1, pagesize:5 }
})
.then((response) => {
  console.log(response);
})
.catch((error)=> {
  console.log("ERROR!!!! : ", error);
})
```

```
axios.get('/contacts', {
  params : { pageno:1, pagesize:5 }
})
.then(...)
.catch(...)
```

Ajax

```
axios.post('/api/todolist/gdhong',
  { todo:"독서하기", desc:"인문서적 1권 이번주까지" })
.then((response) => {
  if (response.data.status !== "success") {
    throw new Error("데이터 추가 실패!!!");
  }
  console.log(response.data);
})
```

```
axios.put('/api/todolist/gdhong/123456789',
  { todo:"TypeScript 공부", desc:"이번주까지", done:true })
.then((response) => {
  if (response.data.status !== "success") {
    throw new Error("데이터 변경 실패!!!");
  }
  console.log(response.data);
})
```

Ajax

axios config 옵션

- baseURL : 이 옵션을 이용해 공통적인 URL의 앞부분을 미리 등록해두면 요청 시 나머지 부분만을 요청 URL로 전달하면 됩니다. 가능하다면 axios.defaults.baseURL 값을 미리 바꾸는 편이 좋습니다.
- transformRequest : 요청 데이터를 서버로 전송하기 전에 데이터를 변환하기 위한 함수를 등록합니다.
- transformResponse : 응답 데이터를 수신한 직후에 데이터를 변환하기 위한 함수를 등록합니다.
- headers : 요청시에 서버로 전달하고자 하는 HTTP 헤더 정보를 설정합니다.

```
// todolist_long 은 1초의 의도적 지연시간을 일으키는 엔드포인트임
axios.defaults.baseURL = '/api/todolist_long';
//timeout에 설정된 시간내에 응답이 오지 않으면 연결을 중단(abort)시킴
axios.defaults.timeout = 2000;
```



감사합니다

단단히 마음먹고 떠난 사람은
산꼭대기에 도착할 수 있다.
산은 올라가는 사람에게만 정복된다.



윌리엄 셰익스피어
William Shakespeare