Arrow function vs Function

> 💡 **Arrow function이와 function 키워드의 가장 큰 차이점은 `this` 의 차이이다.**

Q. 왜 이벤트 리스너에서 일반적으로 arrow function을 사용하면 안될까?

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>
<body>
    <button id="function">function</button>
    <button id="arrow">arrow function</button>
    <script src="https://unpkg.com/axios/dist/axios.min.js"></script>
    <script>
        const functionButton = document.querySelector('#function')
        const arrowButton = document.querySelector('#arrow')

        functionButton.addEventListener('click', function(event) {
            console.log('=====function=====')
            console.log(this)
        })

        arrowButton.addEventListener('click', event => {
            console.log('=====arrow fuction=====')
            console.log(this)
        })

    </script>
</body>
</html>
```

각각의 버튼을 눌러서 확인을 해보자.

```
=====function=====
<button id="function">function</button>
=====arrow fuction=====
Window
```

2. this

자바스크립트의 this는 일반적인 프로그래밍 언어에서의 this와 조금 다르게 동작한다.

java this와 python self의 인스턴스의 호출한 대상의 현재 객체를 뜻하는 것(참조)이었다.

> 💡 **자바스크립트의 function 키워드 함수에서는 함수가 어떻게 호출 되었는지에 따라 다르게 동작한다.**
> **(동적으로 결정)**

브라우저 콘솔에 `this`를 입력 해보자.

```
> this
< ▶Window {parent: Window, postMessage: ƒ, blur: ƒ, focus: ƒ, close: ƒ, …}
```

2-1. window : 함수 호출, 함수 내 함수

window는 브라우저에서의 최상위 객체를 의미한다. (전역 객체)

기본적인 함수 선언을 하고 호출한다면, 이 경우에는 전역에서 호출 하였으므로 전역 객체가 바인딩된다.

```
const greeting = function() {
  console.log(this)
}
greeting() // window
```

- 함수 내 함수는 추후에 다루겠다.

2-2. 객체 : 메소드 호출

메소드로 선언하고 호출한다면, 오브젝트의 메소드이므로 오브젝트가 바인딩된다.

```
const you = {
  name: 'neo',
  greeting
}

you.greeting() // {name: 'neo', greeting: f} : this는 해당 오브젝트(객체)
```

2-3. arrow function

arrow function에서의 this는 어떻게 동작할까?

> 💡 **arrow function에서는 호출과 위치와 상관없이 상위 스코프 this를 가리킨다. (Lexical this)**

따라서, 메소드 선언을 arrow 함수로 하게 된다면, 해당 오브젝트의 상위 스코프인 전역 객체인 window가 바인딩된다.

```
const arrowGreeting = () => {
    console.log(this)
}

const me = {
  name: 'me',
  arrowGreeting
}

arrowGreeting() // window
me.arrowGreeting() // window
```

메소드 선언은 function 키워드를 활용하자!

그러면 ES6에서 왜 arrow function을 언제 활용하면 편할까?

```
const num = {
    numbers:  [1],
    print: function() {
        console.log(this) // {numbers: Array(1), print: f}
        console.log(this.numbers) // [1]
        this.numbers.forEach(function(num) {
            console.log(num) // 1
            console.log(this) // window
        })
    }
}
num.print()
```

- **내부 함수에서는 어디든 상관 없이 항상 전역객체를 바인딩한다.**
- 뒤에 이야기 한다고 했던 함수 내의 함수 상황의 예시가 위와 같다. 이때 arrow function을 쓰면 좋다!
- print 메소드의 내에 있는 콜백함수(forEach)에서의 상위 스코프는 num2 오브젝트이다.

- 따라서, this가 아래와 같이 해당 오브젝트가 바인딩 된다.

```
const num2 = {
    numbers:  [1],
    print: function() {
        console.log(this) // {numbers: Array(1), print: f}
        console.log(this.numbers) // [1]
        this.numbers.forEach(num => {
            console.log(num) // 1
            console.log(this) // {numbers: Array(1), print: f}
        })
    }
}
num2.print()
```

다시 이벤트 리스너로 돌아와서,

`addEventListener` 에서의 콜백 함수는 특별하게 function키워드의 경우에는 이벤트 리스너를 호출한 대상을 (`event.target`) 뜻한다. 따라서, 호출한 대상을 원한다면 `this` 를 활용할 수 있다.

다만, arrow function의 경우 상위 스코프를 지칭하기 때문에 window 객체가 출력된 것이다.

3. 정리

Arrow Function을 쓰면 안되는 대표적인 경우는 다음과 같다.

1. object의 메소드 정의
    - this가 전역 객체(window)를 나타낸다.
2. 생성자 함수
    - 생성자 함수는 object를 생성하는 또다른 방법이다.

```
const MyInfo = {
  name: 'tak',
  phoneNumber : '010-1234-5678',
  greeting() {
    console.log(this.name + 'hi')
  }
}

const Person = function (name) {
  this.name = name
  this.greeting = function() {
    console.log(this.name + 'hi')
```

```
  }
}

const justin = new Person('justin')
```

- Arrow function을 사용하면 에러가 발생한다.

```
const Animal = name => {
  this.name = name
}

const dog = new Animal('dog')
// Uncaught TypeError: Animal is not a constructor
```

3. addEventListener 함수의 콜백 함수
    - this가 전역 객체(window)를 나타낸다.