ES5 함수와 ES6 함수의 차이점

1. 화살표 함수의 선언

화살표 함수는 function 키워드 대신 화살표(⇒)를 사용하여 보다 간략한 방법으로 함수를 선언할 수 있다.

```
// 매개변수 지정 방법
    () => { ... } // 매개변수가 없을 경우
     x => { ... } // 매개변수가 한 개인 경우, 소괄호를 생략할 수 있다.
(x, y) => { ... } // 매개변수가 여러 개인 경우, 소괄호를 생략할 수 없다.

// 함수 몸체 지정 방법
x => { return x * x }  // single line block
x => x * x             // 함수 몸체가 한줄의 구문이라면 중괄호를 생략할 수 있으며 암묵적으로 return된다. 위 표현과 동일하다.
```

```
() => { return { a: 1 }; }
() => ({ a: 1 })  // 위 표현과 동일하다. 객체 반환시 소괄호를 사용한다.

() => {           // multi line block.
  const x = 10;
  return x * x;
};
```

2. 화살표 함수의 호출

화살표 함수는 익명 함수로만 사용할 수 있다. 따라서 화살표 함수를 호출하기 위해서는 함수 표현식을 사용한다.

```
// ES5
var pow = function (x) { return x * x; };
console.log(pow(10)); // 100

// ES6
const pow = x => x * x;
console.log(pow(10)); // 100
```

콜백 함수로 사용할 수 있다. 일반적인 함수 표현식보다 표현이 간결하다.

```
// ES5
var arr = [1, 2, 3];
var pow = arr.map(function (x) { // x는 요소값
  return x * x;
});

console.log(pow); // [ 1, 4, 9 ]

// ES6
const arr = [1, 2, 3];
const pow = arr.map(x => x * x);

console.log(pow); // [ 1, 4, 9 ]
```

3. this

function 키워드로 생성한 일반 함수와 화살표 함수의 가장 큰 차이점은 this 이다.

3.1. 일반 함수의 this

자바스크립트의 경우 함수 호출 방식에 의해 this에 바인딩할 어떤 객체가 동적으로 결정된다. 다시 말해, 함수를 선언할 때 this에 바인딩할 객체가 정적으로 결정되는 것이 아니고, **함수를 호출할 때 함수가 어떻게 호출되었는지에 따라** this에 바인딩할 객체가 동적으로 결정된다.

콜백 함수 내부의 this는 전역 객체 window를 가리킨다.

```
function Prefixer(prefix) {
  this.prefix = prefix;
}

Prefixer.prototype.prefixArray = function (arr) {
  // (A)
  return arr.map(function (x) {
    return this.prefix + ' ' + x; // (B)
  });

  return arr.map(x => this.prefix + ' ' + x);
};

var pre = new Prefixer('Hi');
console.log(pre.prefixArray(['Lee', 'Kim']));
```

(A) 지점에서의 this는 생성자 함수 Prefixer가 생성한 객체, 즉 생성자 함수의 인스턴스(위 예제의 경우 pre)이다.

(B) 지점에서 사용한 this는 아마도 생성자 함수 Prefixer가 생성한 객체(위 예제의 경우 pre)일 것으로 기대하였겠지만, 이곳에서 this는 전역 객체 window를 가리킨다. 이는 생성자 함수와 객체의 메소드를 제외한 모든 함수(내부 함수, 콜백 함수 포함) 내부의 this는 전역 객체를 가리키기 때문이다.

위 설명이 잘 이해되지 않는다면 this를 참조하기 바란다.

콜백 함수 내부의 this가 메소드를 호출한 객체(생성자 함수의 인스턴스)를 가리키게 하려면 아래의 3가지 방법이 있다.

```
// Solution 1: that = this
function Prefixer(prefix) {
  this.prefix = prefix;
}

Prefixer.prototype.prefixArray = function (arr) {
  var that = this;  // this: Prefixer 생성자 함수의 인스턴스
  return arr.map(function (x) {
    return that.prefix + ' ' + x;
  });
};

var pre = new Prefixer('Hi');
```

```
console.log(pre.prefixArray(['Lee', 'Kim']));

// Solution 2: map(func, this)
function Prefixer(prefix) {
  this.prefix = prefix;
}

Prefixer.prototype.prefixArray = function (arr) {
arr.map(() => {}, this);
  return arr.map(function (x) {
    return this.prefix + ' ' + x;
  }, this); // this: Prefixer 생성자 함수의 인스턴스
};

var pre = new Prefixer('Hi');
console.log(pre.prefixArray(['Lee', 'Kim']));

// ES5에 추가된 Function.prototype.bind()로 this를 바인딩한다.
// Solution 3: bind(this)
function Prefixer(prefix) {
  this.prefix = prefix;
}

Prefixer.prototype.prefixArray = function (arr) {
  return arr.map(function (x) {
    return this.prefix + ' ' + x;
  }.bind(this)); // this: Prefixer 생성자 함수의 인스턴스
};

var pre = new Prefixer('Hi');
console.log(pre.prefixArray(['Lee', 'Kim']));
```

3.2. 화살표 함수의 this

일반 함수는 함수를 선언할 때 this에 바인딩할 객체가 정적으로 결정되는 것이 아니고, 함수를 호출할 때 함수가 어떻게 호출되었는지에 따라 this에 바인딩할 객체가 동적으로 결정된다고 하였다.

화살표 함수는 함수를 선언할 때 this에 바인딩할 객체가 정적으로 결정된다. 동적으로 결정되는 일반 함수와는 달리 **화살표 함수의 this 언제나 상위 스코프의 this를 가리킨다.** 이를 **Lexical this**라 한다. 화살표 함수는 앞서 살펴본 Solution 3의 Syntactic sugar이다.

> 💡 화살표 함수의 this 바인딩 객체 결정 방식은 함수의 상위 스코프를 결정하는 방식인 렉시컬 스코프와 유사하다.

```
function Prefixer(prefix) {
  this.prefix = prefix;
}
```

```
Prefixer.prototype.prefixArray = function (arr) {
  // this는 상위 스코프인 prefixArray 메소드 내의 this를 가리킨다.
  return arr.map(x => `${this.prefix}  ${x}`);
};

const pre = new Prefixer('Hi');
console.log(pre.prefixArray(['Lee', 'Kim']));
```

화살표 함수는 call, apply, bind 메소드를 사용하여 this를 변경할 수 없다.

```
window.x = 1;
const normal = function () { return this.x; };
const arrow = () => this.x;

console.log(normal.call({ x: 10 })); // 10
console.log(arrow.call({ x: 10 }));  // 1
```

4. 화살표 함수를 사용해서는 안되는 경우

화살표 함수는 Lexical this를 지원하므로 콜백 함수로 사용하기 편리하다. 하지만 화살표 함수를 사용하는 것이 오히려 혼란을 불러오는 경우도 있으므로 주의하여야 한다.

4.1. 메소드

화살표 함수로 메소드를 정의하는 것은 피해야 한다. 화살표 함수로 메소드를 정의하여 보자.

```
// Bad
const person = {
  name: 'Lee',
  sayHi: () => console.log(`Hi ${this.name}`)
};

person.sayHi(); // Hi undefined
```

위 예제의 경우, 메소드로 정의한 화살표 함수 내부의 this는 메소드를 소유한 객체, 즉 메소드를 호출한 객체를 가리키지 않고 상위 컨택스트인 전역 객체 window를 가리킨다. 따라서 화살표 함수로 메소드를 정의하는 것은 바람직하지 않다.

이와 같은 경우는 메소드를 위한 단축 표기법인 ES6의 축약 메소드 표현을 사용하는 것이 좋다.

```
// Good
const person = {
  name: 'Lee',
  sayHi() { // === sayHi: function() {
    console.log(`Hi ${this.name}`);
  }
};

person.sayHi(); // Hi Lee
```

4.2. prototype

화살표 함수로 정의된 메소드를 prototype에 할당하는 경우도 동일한 문제가 발생한다. 화살표 함수로 정의된 메소드를 prototype에 할당하여 보자.

```
// Bad
const person = {
  name: 'Lee',
};

Object.prototype.sayHi = () => console.log(`Hi ${this.name}`);

person.sayHi(); // Hi undefined
```

화살표 함수로 객체의 메소드를 정의하였을 때와 같은 문제가 발생한다. 따라서 prototype에 메소드를 할당하는 경우, 일반 함수를 할당한다.

```
// Good
const person = {
  name: 'Lee',
};

Object.prototype.sayHi = function() {
  console.log(`Hi ${this.name}`);
};

person.sayHi(); // Hi Lee
```

4.3. 생성자 함수

화살표 함수는 생성자 함수로 사용할 수 없다. 생성자 함수는 prototype 프로퍼티를 가지며 prototype 프로퍼티가 가리키는 프로토타입 객체의 constructor를 사용한다. 하지만 화살표 함수는 prototype 프로퍼티를 가지고 있지 않다.

```
const Foo = () => {};

// 화살표 함수는 prototype 프로퍼티가 없다
console.log(Foo.hasOwnProperty('prototype')); // false

const foo = new Foo(); // TypeError: Foo is not a constructor
```

4.4. addEventListener 함수의 콜백 함수

addEventListener 함수의 콜백 함수를 화살표 함수로 정의하면 this가 상위 컨택스트인 전역 객체 window를 가리킨다.

```
// Bad
var button = document.getElementById('myButton');

button.addEventListener('click', () => {
  console.log(this === window); // => true
  this.innerHTML = 'Clicked button';
});
```

따라서 addEventListener 함수의 콜백 함수 내에서 this를 사용하는 경우, function 키워드로 정의한 일반 함수를 사용하여야 한다. 일반 함수로 정의된 addEventListener 함수의 콜백 함수 내부의 this
는 이벤트 리스너에 바인딩된 요소(currentTarget)를 가리킨다.

```
// Good
var button = document.getElementById('myButton');

button.addEventListener('click', function() {
  console.log(this === button); // => true
  this.innerHTML = 'Clicked button';
});
```