코어 자바스크립트

4강 콜백 함수

참고문헌

- 코어자바스크립트
- 모던자바스크립트 DeepDive

콜백함수(callback function)?

콜백 함수는 다른 코드(함수 또는 메서드)에게 인자로 넘겨줌으로써 그 **제어권**도 함께 위임한 함수

(콜백 함수를 위임 받은 코드는 자체적인 내부 로직에 의해 콜백 함수를 적절한 시점에 실행)

제어권 호출 시점

```
const intervalId = setInterval(func[, delay, param1, param2, ...]);
clearInterval(intervalId);
```

*** setInterval / clearInterval**

setInterval 함수는 두 번째 인수(delay)로 전달받은 시간(ms)으로 반복 동작하는 타이머를 생성함. (delay도 생략 가능하고, 생략 시 0으로 동작)

이후 타이머가 만료될 때마다 첫 번째 인수로 전달받은 콜백 함수가 반복 호출됨.

return값은 생성된 타이머를 식별할 수 있는 고유한 타이머 id (브라우저 환경은 숫자, Node.js환경은 객체)

[param1, param2…]는 콜백함수의 인자가 된다.

return값을 clearInterval 함수로 전달하면 타이머를 취소할 수 있음.

브라우저 환경

```
> console.log(timeoutId);
  2
```

Node.js 환경

```
> console.log(timeoutId)
Timeout {
  _idleTimeout: -1,
  _idlePrev: null,
  _idleNext: null,
  _idleStart: 61220,
  _onTimeout: null,
  _timerArgs: undefined,
  _repeat: 1000,
  _destroyed: true,
  [Symbol(refed)]: true,
  [Symbol(kHasPrimitive)]: false,
  [Symbol(asyncId)]: 473,
  [Symbol(triggerId)]: 5
}
```

제어권 호출 시점

```
var count = 0;
var cbFunc = function () {
  console.log(count);
  if(++count > 4) clearInterval(timer);
};
var timer = setInterval(cbFunc, 300);
```

```
//0.3s마다 count 출력
0
1
2
3
4
```

code	호출 주체	제어권
cbFunc();	사용자	사용자
setInterval(cbFunc, 300);	setInterval	setInterval

cbFunc();로 함수를 실행하면 호출 주체와 제어권은 사용자에게 있습니다.

하지만 setInterval안에 인자로 cbFunc 함수를 넘겨주면,

제어권은 setInterval로 넘어가고 setInterval이 판단하여 0.3초마다 익명 함수를 실행합니다.

➡ 콜백 함수의 제어권을 넘겨받은 코드는 **콜백 함수 호출 시점**에 제어권을 갖는다!

제어권 인자

```
var newArr = [10, 20, 30].map(function (currentValue, index)
{
  console.log(currentValue, index);
  return currentValue + 5;
});
console.log(newArr);
```

```
//실행결과
10 0
20 1
30 2
[15, 25, 35]
```

```
Array.prototype.map(callback[, thisArg])
callback: function(currentValue[, index[, array])
```

map 메서드는 첫 번째 인자로 콜백 함수를 인자로 받고, thisArg를 생략하면 this는 전역객체를 가리킴.

map메서드는 배열의 모든 요소들을 처음부터 끝까지 하나씩 꺼내어 콜백 함수를 반복 호출하고,

콜백 함수의 실행결과(retur값)를 모아 새로운 배열을 만든다.

currentValue는 배열의 요소 중 현재 값이, index는 현재 값의 인덱스가, array에는 map메서드의 대상이 되는 배열 자체가 담긴다.

➡ 콜백 함수의 제어권을 넘겨받은 코드는 콜백 함수를 호출할 때 인자에 **어떤 값을 어떤 순서로 넘길 것인지에** 대한 제어권을 갖음.

제어권 this

* 콜백 함수도 함수이기 때문에 기본적으로는 this가 전역객체를 참조하지만, 제어권을 넘겨받을 코드에서 콜백 함수에 별도로 this가 될 대상을 지정한 경우에는 그 대상을 참조하게 됨

map메서드 구현

```
Array.prototype.map = function (callback,
thisArg) {
  var mapperArr = [];
  for (var i = 0; i< this.length; i++) {
    var mappedValue =
callback.call(thisArg || window, this[i],
i, this);
    mapperArr[i] = mappedValue;
  }
  return mapperArr;
};
```

제어권을 넘겨받을 코드에서 콜백 함수 내부에서의 this가 될 대상을 명시적으로 바인딩

```
setTimeout(function() { console.log(this); },
300); // Window{ ... }
[1, 2, 3, 4, 5].forEach(function (x)
{ console.log(this); }); // Window{ ... }

document.body.innerHTML += '<button
id="a">클릭</button>'
document.body.querySelector('#a')
    .addEventListener('click', function(e)
{ console.log(this, e); });
// <button id="a">클릭</button>
// MouseEvent { isTrusted: true, ...}
```

setTimeout과 forEach에서는 this가 전역객체

addEventListener에서는 this가 호출주체인 HTML 엘리먼트

콜백 함수는 함수

*함수는 그 자체로 독립적인 기능을 수행하고, 메서드는 자신을 호출한 객체에 관한 동작을 수행한다.

어떤 객체의 메서드를 콜백 함수로 전달해도 그 메서드는 메서드가 아닌 함수로서 호출됨!

```
var obj = {
  vals: [1, 2, 3],
  logValues: function(v, i) {
    console.log(this, v, i);
  }
};
obj.logValues(1, 2); // ................(1)
[4, 5, 6].forEach(obj.logValues); //....(2)
```

(1)의 경우 this는 obj이다.

* 출력결과: { vals: [1, 2, 3], logValues: f } 1 2

(2)의 경우는 forEach의 인자로 obj의 메서드인 logValues를 준 경우이다. 이 때의 this는 전역객체이다.

* 출력결과

Window { ... } 4 0

Window { ... } 5 1

Window { ... } 6 2

➡ 어떤 함수의 인자에 객체의 메서드를 전달해도 이는 메서드가 아닌 함수!

콜백 함수 내부의 this에 다른 값 바인딩하기

객체의 메서드로 콜백함수를 전달했을 때, this가 객체를 바라보게 하고 싶다면?

```
var obj = {
  vals: [1, 2, 3],
  logValues: function(v, i) {
    console.log(this, v, i);
  }
};
[4, 5, 6].forEach(obj.logValues);
```

```
Window { ... } 4 0
Window { ... } 5 1
Window { ... } 6 2
```

```
{ vals: [1, 2, 3], logValues: f }  4 0
{ vals: [1, 2, 3], logValues: f } 5 1
{ vals: [1, 2, 3], logValues: f } 6 2
```

1) this를 다른 변수에 담아 콜백 함수로 활용할 함수에서는 this 대신 변수를 사용
(전통적인 방식, 클로저를 만들 때 많이 사용)

2) bind 메서드

콜백 함수 내부의 this에 다른 값 바인딩하기 변수 사용

```
var obj1 = {
  name: 'obj1',
  func: function () {
    var self = this;
    return function () {
      console.log(self.name);
    };
  }
};
var callback = obj1.func();
setTimeout(callback, 1000);
```

```
var obj1 = {
  name: 'obj1',
  func: function () {
    console.log(obj1.name);
  }
};
setTimeout(obj1.func, 1000);
```

실제로 this를 사용하지 않고 번거로움!

this를 사용하지 않고 바로 객체의 프로퍼티에 접근하는 방법이 간결

BUT 재활용은 불가능

콜백 함수 내부의 this에 다른 값 바인딩하기 변수 사용

기존 예제

```
var obj1 = {
    name: 'obj1',
    func: function () {
      var self = this;
      return function () {
        console.log(self.name);
      };
    }
  };
  var callback = obj1.func();
  setTimeout(callback, 1000);

  var obj2 = {
    name: 'obj2',
    func: obj1.func
  };
  var callback2 = obj2.func();
  setTimeout(callback2, 1500);

  var obj3 = { name: 'obj3' };
  var callback3 = obj1.func.call(obj3);
  setTimeout(callback3, 2000);
```

obj1의 func를 재활용, this는 새로 만든 객체가 되게 하고 싶음!

1) obj2의 경우

obj1의 func를 복사하여 이를 메서드를 갖는 obj2 객체를 생성하고, obj2를 실행한 결과를 담아 콜백으로 사용

2) obj3의 경우

obj3의 콜백함수는 obj1의 func를 실행할 때 this를 obj3가 되도록 지정한 것. (call 메서드 사용)

콜백 함수 내부의 this에 다른 값 바인딩하기 bind 메서드

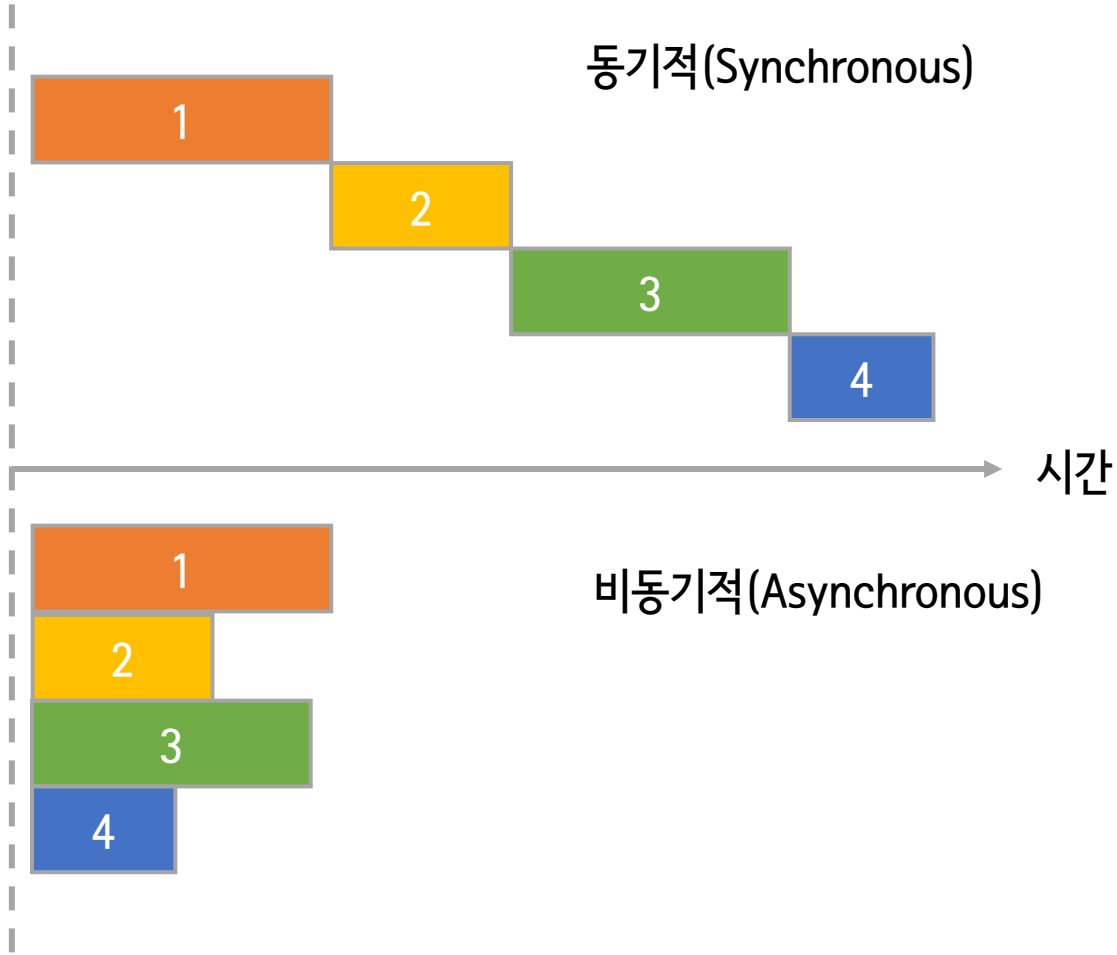
```
var obj1 = {
  name: 'obj1',
  func: function () {
    console.log(this.name);
  }
};
setTimeout(obj1.func.bind(obj1), 1000);

var obj2 = { name: 'obj2' };
setTimeout(obj1.func.bind(obj2), 1500);
```

ES5의 bind메서드를 이용하면 깔끔!

* call/apply 메서드는 함수를 즉시 실행하고,
bind 메서드는 새로운 함수를 반환하기 때문에
위 코드에서 콜백 함수로 인자를 주는 것이 가능!

콜백 지옥과 비동기 제어 비동기 vs 동기



동기적인 코드는 실행 중인 코드가 완료되어야 다음 코드를 실행하는 방식

- 즉시 처리 가능한 코드는 대부분 동기적인 코드
- 계산식이 복잡해 시간이 오래 걸려도 동기적인 코드

비동기적인 코드는 현재 실행 중인 코드의 완료 여부와 무관하게 즉시 다음 코드를 실행하는 방식

- 사용자의 요청, 실행 대기, 보류 등과 관련된 코드는 비동기적인 코드

ex) setTimeout, addEventListner, XMLHttpRequest*

콜백 지옥과 비동기 제어 콜백 지옥

- 콜백 함수를 익명 함수로 전달하는 과정이 반복되어 코드의 들여쓰기 수준이 감당하기 힘들 정도로 깊어지는 현상
- 이벤트 처리, 서버 통신 등 비동기적인 작업을 수행할 때 자주 등장
- 0.5s 주기마다 커피 목록을 수집하고 출력

```
setTimeout(function (name) {
  var coffeeList = name;
  console.log(coffeeList);

  setTimeout(function (name) {
    coffeeList += ', ' + name;
    console.log(coffeeList);

    setTimeout(function (name) {
      coffeeList += ', ' + name;
      console.log(coffeeList);

      setTimeout(function (name) {
        coffeeList += ', ' + name;
        console.log(coffeeList);
      }, 500, '카페라떼');
    }, 500, '카페모카')
  }, 500, '아메리카노');
}, 500, '에스프레소');
```

콜백 지옥과 비동기 제어 콜백 지옥

```
setTimeout(function (name) {
  var coffeeList = name;
  console.log(coffeeList);

  setTimeout(function (name) {
    coffeeList += ', ' + name;
    console.log(coffeeList);

    setTimeout(function (name) {
      coffeeList += ', ' + name;
      console.log(coffeeList);

      setTimeout(function (name) {
        coffeeList += ', ' + name;
        console.log(coffeeList);
      }, 500, '카페라떼');
    }, 500, '카페모카')
  }, 500, '아메리카노');
}, 500, '에스프레소');
```

가독성↑

```
var coffeeList = '';
var addEspresso = function (name) {
  coffeeList = name;
  console.log(coffeeList);
  setTimeout(addAmericano, 500, '아메리카노');
};
var addAmericano = function (name) {
  coffeeList += ', ' + name;
  console.log(coffeeList);
  setTimeout(addMocha, 500, '카페모카');
};
var addMocha = function (name) {
  coffeeList += ', ' + name;
  console.log(coffeeList);
  setTimeout(addLatte, 500, '카페라떼');
};
var addLatte = function (name) {
  coffeeList += ', ' + name;
  console.log(coffeeList);
};
setTimeout(addEspresso, 500, '에스프레소');
```

콜백 지옥과 비동기 제어 콜백 지옥

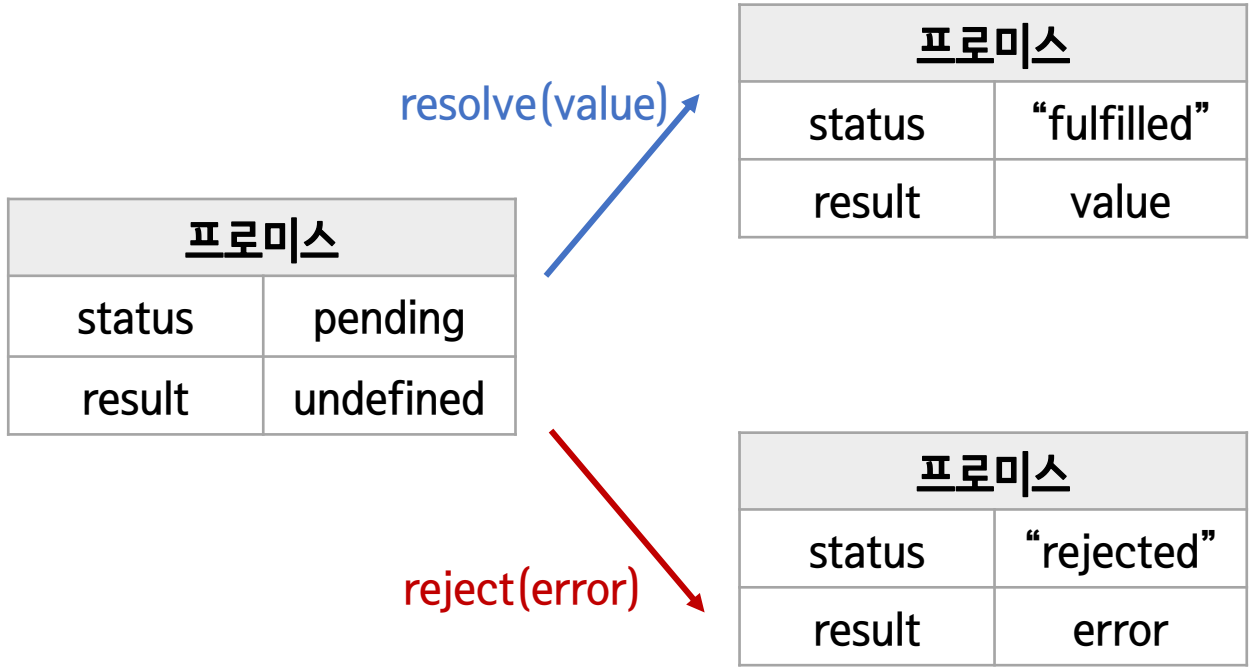
비동기적인 작업을 동기적으로, 혹은 동기적을 보이게끔 처리해주는 장치

1) Promise(ES6)
2) Generator(ES6)
3) async/await(ES2017)

프로미스(Promise)?

Promise 객체는 비동기 작업이 맞이할 미래의 완료 또는 실패와 그 결과 값을 나타냄!

```
//프로미스 생성
const promise = new Promise((resolve, reject) => {
  // Promise 함수의 콜백 함수 내부에서 비동기 처리를 수행
  if(/* 비동기 처리 성공 */){
    resolve('result');
  } else{/* 비동기 처리 실패 */
    reject('failure reason')
  }
});
```



프로미스(Promise)후속 처리 메서드

Promise.prototype.then 메서드

첫번째 인수는 프로미스가 fulfilled일 때 호출하고, 이때 콜백함수는 프로미스의 비동기 처리 결과를 인수로 받음. 언제나 프로미스 반환

```
new Promise(resolve => resolve('fulfilled')).then(v => console.log(v), e => console.error(e));
new Promise((_, reject) => reject('rejected')).then(v => console.log(v), e => console.error(e));
```

Promise.prototype.catch 메서드

한 개의 콜백 함수를 인수로 전달받음. 프로미스가 rejected 상태인 경우만 호출. 언제나 프로미스 반환

```
new Promise((_, reject) => reject(new Error('rejected'))).catch(e => console.error(e));
```

Promise.prototype.finally 메서드

한 개의 콜백 함수를 인수로 전달받음. 프로미스 성공/실패 관계없이 한 번만 호출. 언제나 프로미스 반환

```
new Promise(() => {}).finally(() => console.log('finally'));
```

비동기 작업의 동기적 표현 프로미스(Promise)

```
new Promise(function (resolve) {
  setTimeout(function () {
    var name = '에스프레소';
    console.log(name);
    resolve(name);
  }, 500);
}).then(function (prevName){
  return new Promise(function (resolve) {
    setTimeout(function () {
      var name = prevName + ', 아메리카노';
      console.log(name);
      resolve(name);
    }, 500);
  });
})
```

생략

비동기 작업의 동기적 표현 프로미스(Promise)

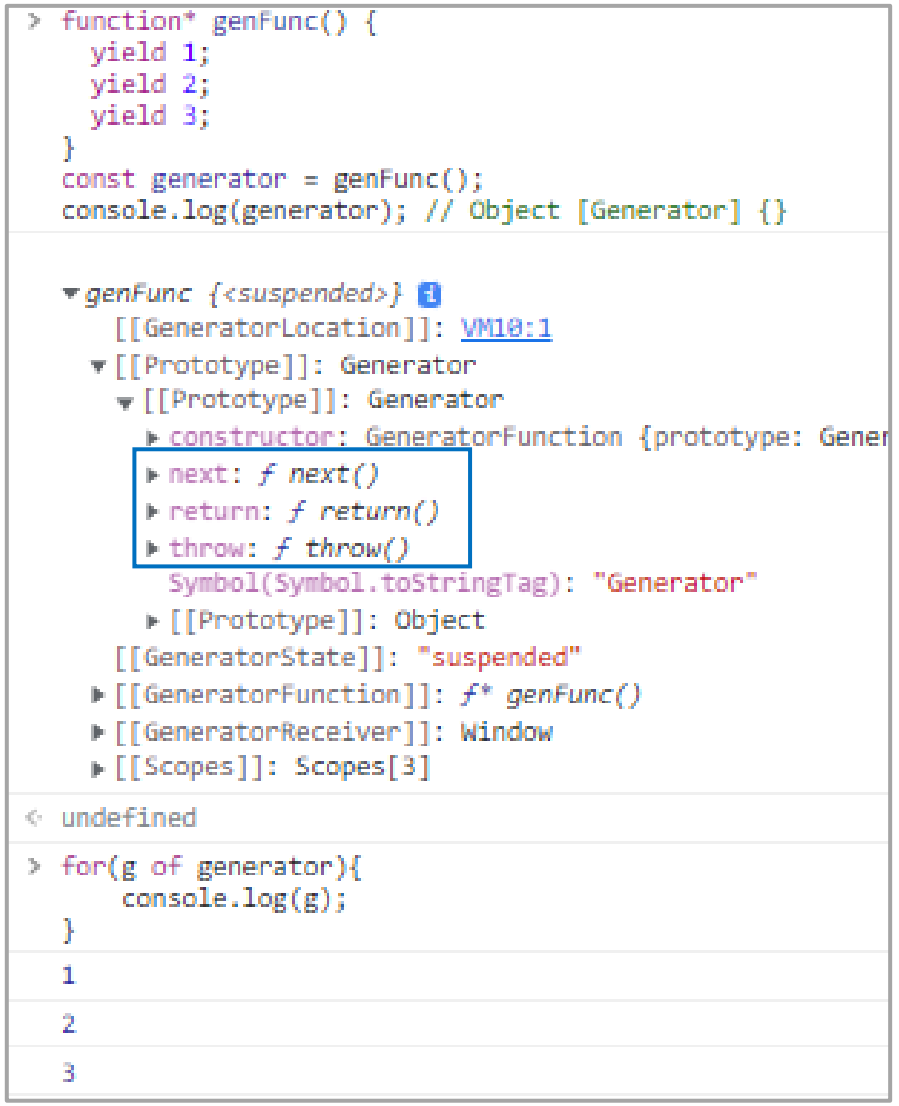
```
var addCoffee = function (name) {
  return function (prevName){
    return new Promise(function (resolve) {
      setTimeout(function () {
        var newName = prevName ? (prevName + ', ' + name) : name;
        console.log(newName);
        resolve(newName);
      }, 500);
    });
  };
}
addCoffee('에스프레소')()
    .then(addCoffee('아메리카노'))
    .then(addCoffee('카페모카'))
    .then(addCoffee('카페라떼'));
```

클로저를 활용해 반복적인 내용을 함수화하여 짧게 표현

제너레이터(Generator)?

- 코드 블록의 실행을 일시 중지 했다가 필요한 시점에 재개할 수 있는 특수한 함수
(한 번에 코드 블록의 모든 코드를 일괄 실행하는 것이 아니고, yield 표현식까지만 실행)
- function* 키워드로 선언하고 하나 이상의 yield 표현식을 포함해야 함(화살표 함수 정의X, new 연산자 사용X)
- iterable이면서 iterator인 제너레이터 객체를 반환

* iterable : 반복 가능한 객체 (for..of문, spread operator 등 사용 가능)
* iterator : iterable객체에서 반복을 실행하는 반복기, next 메서드가 있고 iterator result object를 반환하는데 이 객체는 done과 value를 프로퍼티로 가져야 함.



제너레이터객체 메서드

Genrator.prototype.next 메서드

yield 표현식을 통해 yield된 값을 반환함. next메서드를 호출하면 yield표현식까지 실행되고 일시 중지 됨.

value는 제네레이터로 보낼 값. variable = yield 식 에서 value는 variable에 할당됨.

done(반복이 끝났을 때만 true)과 value(제너레이터가 생성,반환한 값)를 프로퍼티로 갖는 iterator result object를 반환.

```
generatorObject.next(value);
```

Genrator.prototype.return 메서드

주어진 값을 반환하고 제네레이터 종료

```
generatorObject.return(value);
```

Genrator.prototype.throw 메서드

제네레이터에 오류 발생

```
generatorObject.throw(exception);
```

비동기 작업의 동기적 표현 제너레이터(Generator)

```
var addCoffee = function (prevName, name) {
  setTimeout(function () {
    coffeeMaker.next(prevName? prevName + ', ' + name : name);
  }, 500);
};
var coffeeGenerator = function* () {
  var espresso = yield addCoffee('', '에스프레소');
  console.log(espresso);
  var americano = yield addCoffee(espresso, '아메리카노');
  console.log(americano);
  var mocha = yield addCoffee(americano, '카페모카');
  console.log(mocha);
  var latte = yield addCoffee(mocha, '카페라떼');
  console.log(latte);
};
var coffeeMaker = coffeeGenerator();
coffeeMaker.next();
```

async/await?

async함수

async 키워드를 사용해 정의하며 언제나 프로미스를 반환. 명시적으로 프로미스를 반환하지 않아도 암묵적으로 반환값을 resolve하는 프로미스를 반환.(class의 constructor는 인스턴스를 반환해야 하므로 async 메서드가 되지 못한다.

await 키워드

프로미스가 비동기 처리가 수행된 상태(settled)가 될 때까지 대기하다가 프로미스가 resolve한 처리 결과를 반환.

async함수 배부에서 사용되어야 하며, 반드시 프로미스 앞에 사용해야 함

*async/await에서의 에러 처리

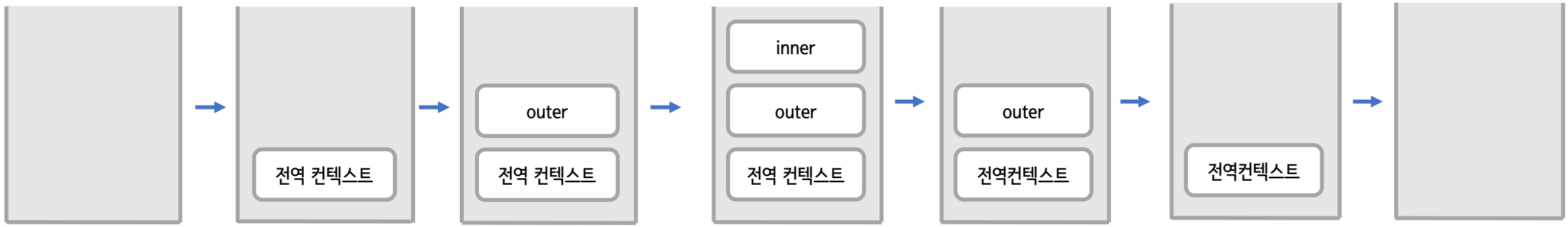
try...catch문으로 처리 가능하고, 에러 처리를 하지 않으면 에러를 reject하는 프로미스를 반환.

비동기 작업의 동기적 표현 Promise + async/await

```
var addCoffee = function (name) {
  return new Promise(function (resolve) {
    setTimeout(function () {
      resolve(name);
    }, 500);
  });
};
var coffeeMaker = async function () {
  var coffeeList = '';
  var _addCoffee = async function (name) {
    coffeeList += (coffeeList ? ',' : '') + await addCoffee(name);
  };
  await _addCoffee('에스프레소');
  console.log(coffeeList);
  await _addCoffee('아메리카노');
  console.log(coffeeList);
  await _addCoffee('카페모카');
  console.log(coffeeList);
  await _addCoffee('카페라떼');
  console.log(coffeeList);
};
coffeeMaker();
```

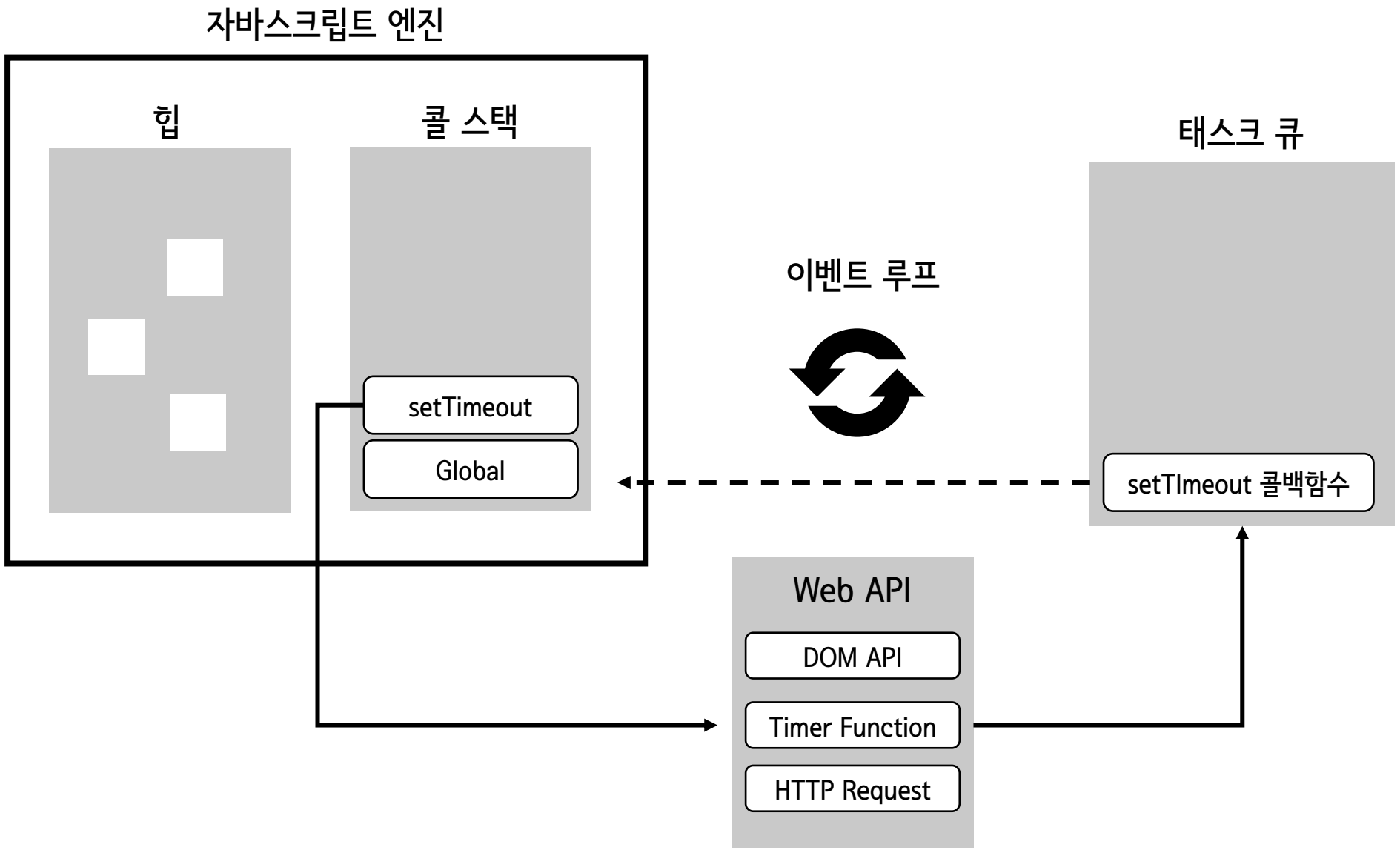
* 참고 JS에서 비동기 처리가 가능한 이유?

```
var a = 1;
function outer() {
  function inner() {
    console.log(a);//undefined
    var a = 3;
  }
  inner();
  console.log(a); //1
}
outer();
console.log(a); //1
```

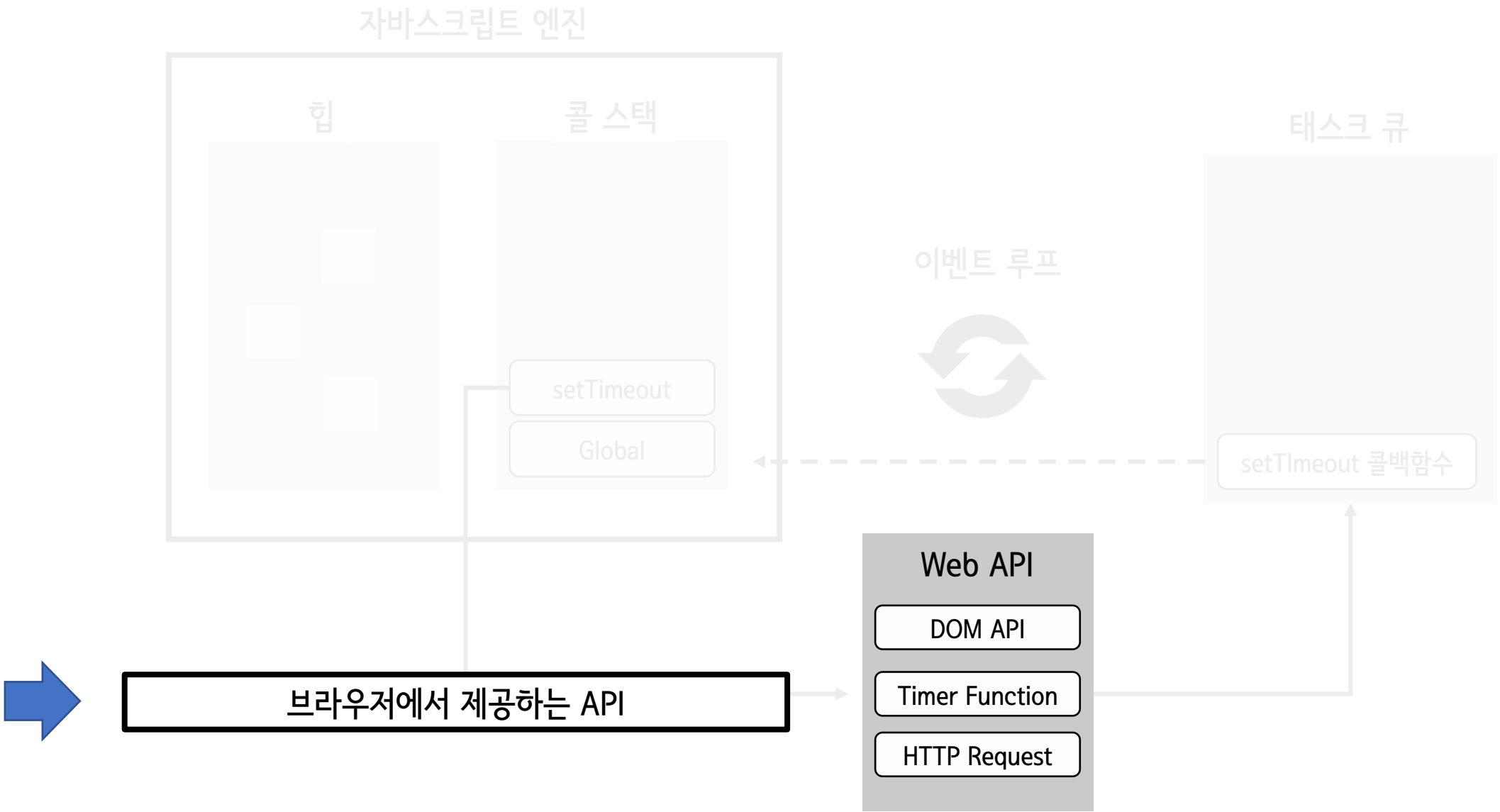


콜스택에서 실행순서를 관리하고 함수는 2개 이상 동시에 실행할 수 없는데, 비동기 처리는 어떻게 가능한 걸까?

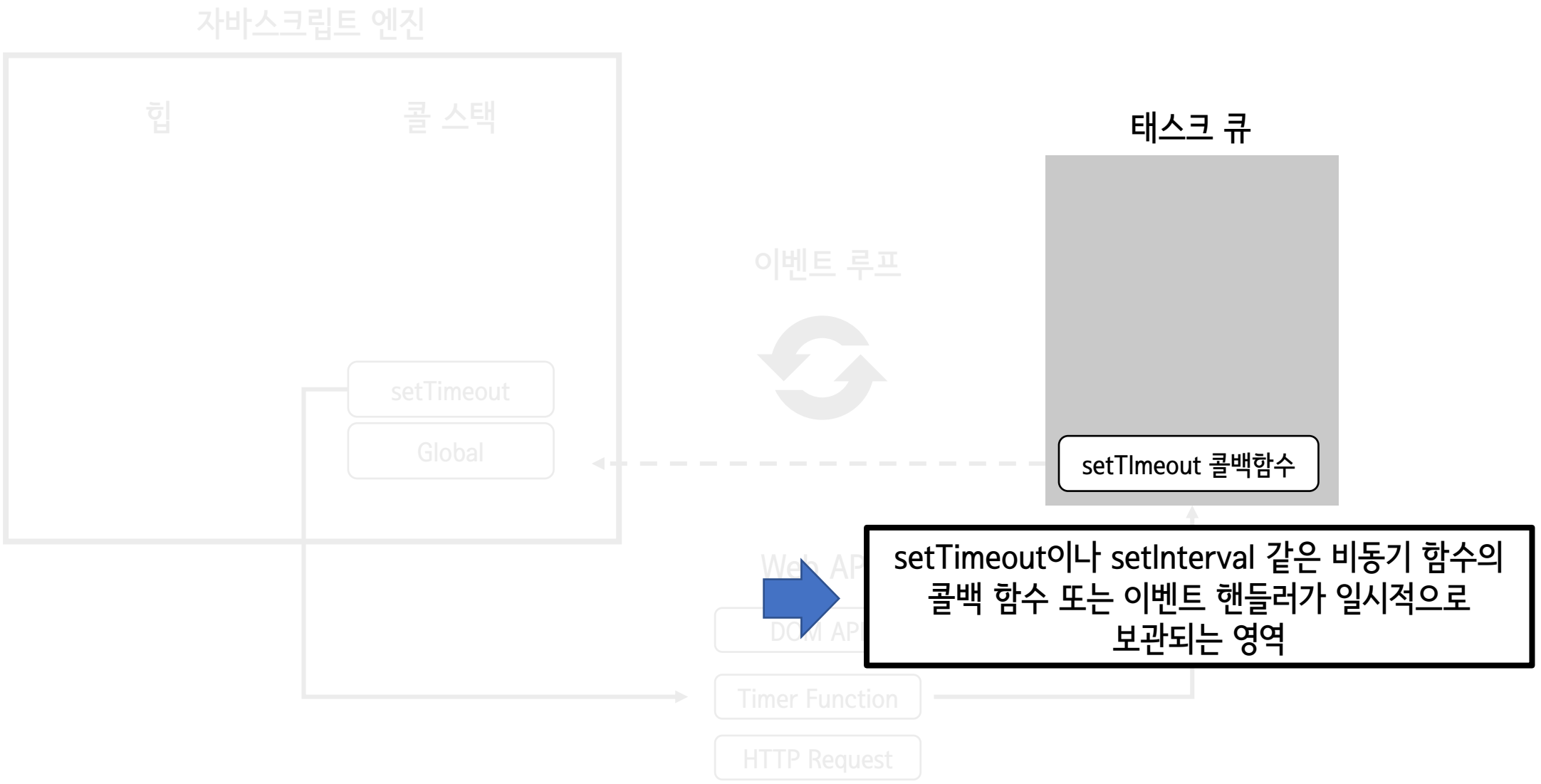
* 참고 JS에서 비동기 처리가 가능한 이유?



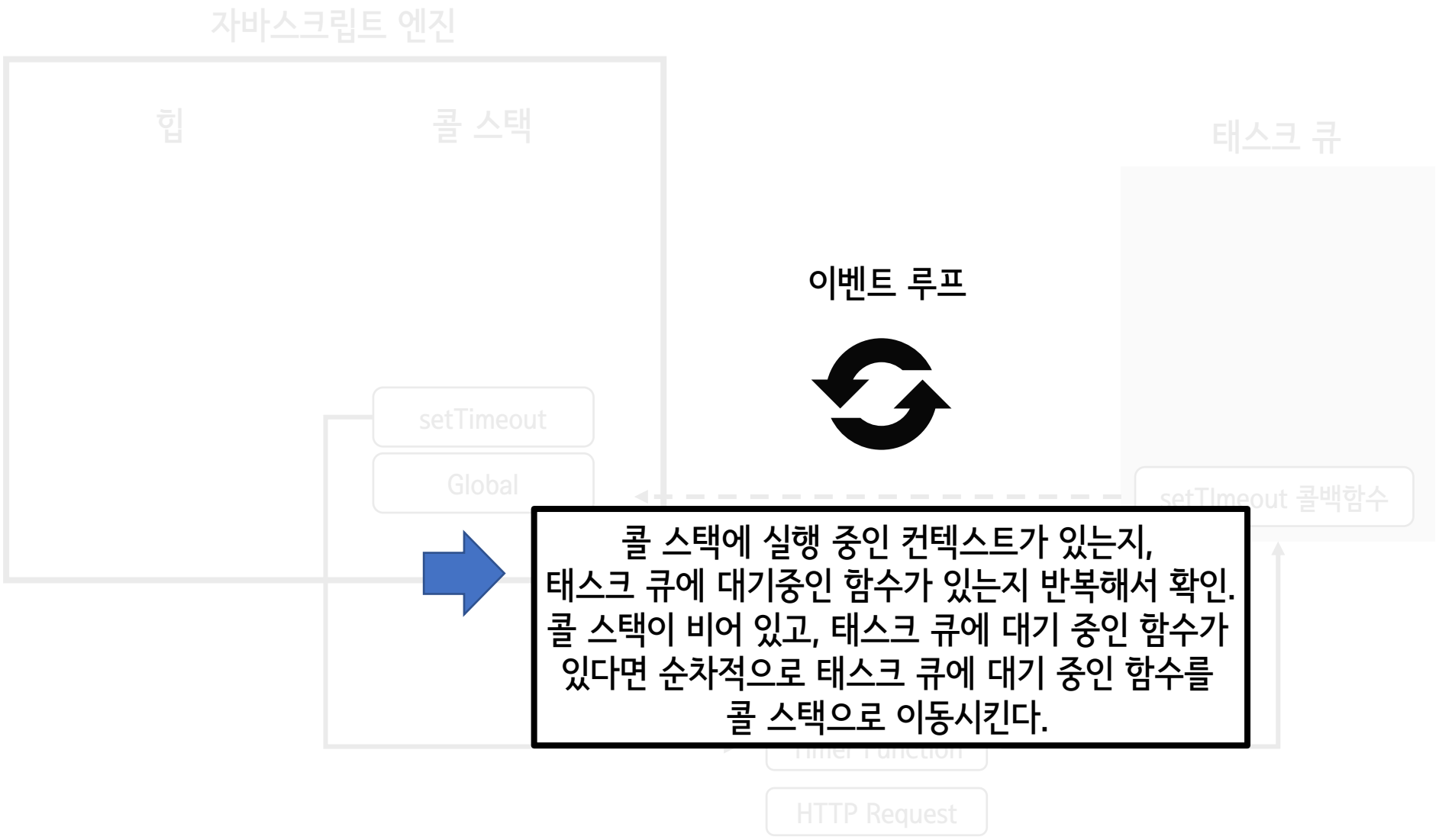
* 참고 JS에서 비동기 처리가 가능한 이유?



* 참고 JS에서 비동기 처리가 가능한 이유?



* 참고 JS에서 비동기 처리가 가능한 이유?



* 참고 자바스크립트는 싱글스레드 언어인데, 비동기 처리가 가능한 이유가 뭔가요?(면접 질문)

자바스크립트는 싱글 스레드 방식으로 동작함.

* 싱글 스레드: 하나의 프로그램에서 동시에 하나의 코드만 실행 가능

자바스크립트 엔진 밖에도 web API, 태스크 큐, 이벤트 루프가 자바스크립트 실행에 관여하기 때문이다. 비동기 함수는 web API에 위임하고 web API는 실행 후 콜백함수를 태스크 큐에 삽입한다. 이벤트 루프는 콜스택이 비었는지, 태스크 큐에 대기중인 함수가 있는지 계속해서 확인하는데, 콜스택이 비었고, 태스크 큐에 대기중인 함수가 있다면 콜백함수를 큐에서 빼서 콜스택에서 실행한다.

QnA 지수님 스터디 자료 참고

- 콜백함수란 무엇인가요?
- 제어권에는 어떤 게 있나요?
- 콜백함수에서의 this는 무엇을 가리키며, 이를 의도적으로 명시하기 위한 방법에는 무엇이 있나요?
- 콜백지옥이란 무엇이며, 이를 해결하기 위한 방법에는 무엇이 있나요?
- 동기와 비동기에 대해서 설명해보세요.
- async, await의 내부는 어떻게 구현되어 있을까요?
- 콜백지옥을 ES6 이전에는 어떻게 해결했나요?
- Promise는 어떤 식으로 구현이 되어있을까요?
- setTimeout의 두번째 인자가 0일 때, 콜백이 언제 발생하나요?