* 예습
1. 6/5

42. 비동기 프로그래밍

TABLE OF CONTENTS

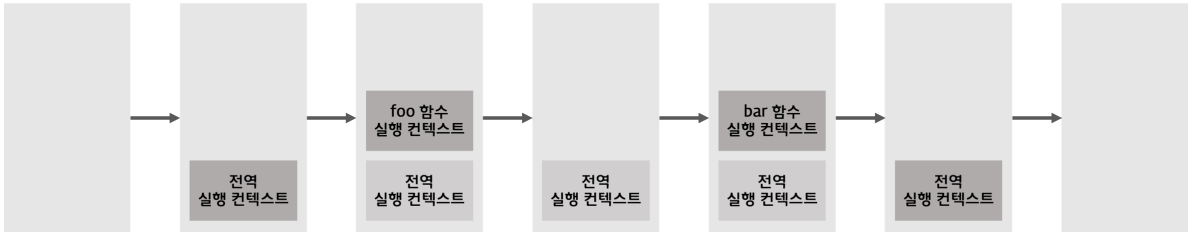
#1. 동기식 처리 모델과 비동기식 처리 모델

"23. 실행 컨텍스트"에서 살펴본 바와 같이 함수를 호출하면 함수 코드가 평가되어 함수의 실행 컨텍스트가 생성된다. 이때 생성된 함수의 실행 컨텍스트는 실행 컨텍스트 스택(콜 스택이라고도 부른다)에 푸시되고 함수 코드가 실행된다. 함수 코드의 실행이 종료하면 함수의 실행 컨텍스트는 실행 컨텍스트 스택에서 팝되어 제거된다.

다음 예제의 foo 함수와 bar 함수는 호출된 순서대로 스택 자료구조인 실행 컨텍스트 스택에 푸시되어 실행된다.

JAVASCRIPT

```
const foo = () => {};
const bar = () => {};

foo();
bar();
```



실행 컨텍스트 스택

함수가 실행되려면 함수 코드 평가 과정에서 생성된 함수 실행 컨텍스트가 실행 컨텍스트 스택에 푸시되어야 한다. 다시 말해, 실행 컨텍스트 스택에 함수 실행 컨텍스트가 푸시되는 것은 바로 함수의 실행을 의미한다. 함수가 호출된 순서대로 순차적으로 실행되는 이유는 함수가 호출된 순서대로 함수 실행 컨텍스트가 실행 컨텍스트 스택에 푸시되기 때문이다. 이처럼 함수의 실행 순서는 실행 컨텍스트 스택으로 관리한다.

자바스크립트 엔진은 단 하나의 실행 컨텍스트 스택을 갖는다. 이는 함수를 실행할 수 있는 창구가 단 하나이며 동시에 2개 이상의 함수를 동시에 실행할 수 없다는 것을 의미한다. 실행 컨텍스트 스택의 최상위 스택(실행 중인 실행 컨텍스트)을 제외한 모든 실행 컨텍스트는 모두 실행 대기 중인 태스크(task)들이다. 대기 중인 태스크들은 현재 실행 중인 실행 컨텍스트가 팝되어 실행 컨텍스트 스택에서 제거되면, 다시 말해 현재 실행 중인 함수가 종료하면 비로소 실행되기 시작한다.

이처럼 자바스크립트 엔진이 동작하는 브라우저 환경이나 Node.js 환경은 한번에 하나의 태스크만 실행할 수 있는 **싱글 스레드**(single thread) 방식으로 동작한다. 싱글 스레드 방식은 한번에 하나의 태스크만 실행할 수 있기 때문에 처리에 시간이 걸리는 태스크를 실행하는 경우 **블로킹**(blocking, **작업 중단**)이 발생한다. 예를 들어, setTimeout 함수와 유사하게 일정 시간이 경과한 이후에 콜백 함수를 호출하는 sleep 함수를 구현해 보자.

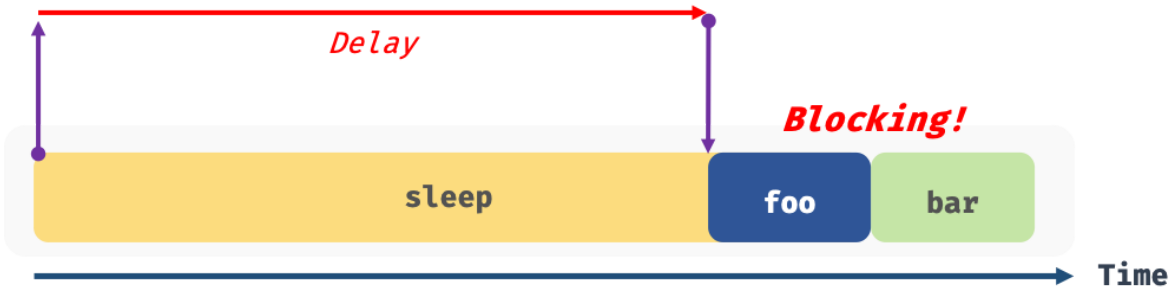
JAVASCRIPT

```
// sleep 함수는 일정 시간(delay)이 경과한 이후에 콜백 함수(func)를 호출한다.
function sleep(func, delay) {
  // Date.now()는 현재 시간을 숫자(ms)로 반환한다.("30.2.1. Date.now" 참고)
  const delayUntil = Date.now() + delay;

  // 현재 시간(Date.now())에 delay를 더한 delayUntil이 현재 시간보다 작으면 계속 반복한다.
  while (Date.now() < delayUntil);
  func();
}

function foo() {
  console.log('foo');
}
```

```
function bar() {
  console.log('bar');
}

// 일정 시간이 경과한 이후에 콜백 함수 foo를 호출하므로 다음에 호출될 bar가 블로킹된다.
sleep(foo, 3 * 1000);
bar();
// (3초 경과후) foo → bar
```

위 예제는 일정 시간이 경과한 이후에 foo 함수를 호출하고 그 후에 bar 함수를 호출한다. 이때 foo 함수는 3초 후에 호출되므로 bar 함수는 foo 함수의 실행 시간 + 3초 후에 호출된다. 이처럼 현재 실행중인 태스크가 종료할 때까지 다음 실행될 태스크가 대기하는 방식을 **동기식 처리 모델**(synchronous processing model)이라고 한다. 동기식 처리 모델은 태스크를 순차적으로 하나씩 처리하는 방식이다.

동기식 처리 모델은 태스크를 순서대로 처리하므로 실행하므로 처리 순서가 보장된다는 장점이 있다. 하지만 앞선 태스크가 종료할 때까지 이후 태스크들이 블로킹(작업 중단)되는 단점이 있다. 위 예제의 sleep 함수는 3초간 대기하다가 인수로 전달받은 콜백 함수 foo를 호출한다. 따라서 sleep 함수가 종료된 이후 실행될 태스크인 bar 함수는 3초 이상(sleep 함수의 실행 시간 3초와 콜백 함수 foo의 실행 시간) 블로킹 된다.



동기식 처리 모델

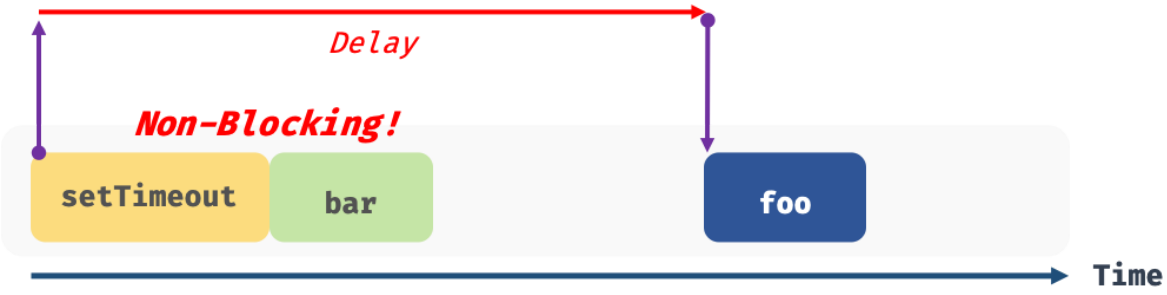
위 예제를 타이머 함수인 setTimeout을 사용하여 수정해 보자.

JAVASCRIPT

```
function foo() {
  console.log('foo');
}

function bar() {
  console.log('bar');
}
```

```
// 타이머 함수 setTimeout는 일정 시간이 경과한 이후에 콜백 함수 foo를 호출한다.
// 타이머 함수 setTimeout는 bar 함수를 블로킹하지 않는다.
setTimeout(foo, 3 * 1000);
bar();
// bar → (3초 경과후) foo
```

타이머 함수 setTimeout은 앞서 살펴본 sleep 함수와 유사하게 일정 시간이 경과한 이후에 콜백 함수를 호출하지만 setTimeout 이후의 태스크를 블로킹하지 않고 곧바로 실행한다. 이처럼 현재 실행중인 태스크가 종료되지 않은 상태라 하더라도 다음 태스크를 곧바로 실행하는 방식을 **비동기식 처리 모델** (asynchronous processing model)이라고 한다.



비동기식 처리 모델

동기식 처리 모델은 태스크를 순서대로 처리하므로 실행하므로 처리 순서가 보장된다는 장점이 있지만 앞선 태스크가 종료할 때까지 이후 태스크들이 블로킹되는 단점이 있었다. 비동기식 처리 모델은 블로킹이 발생하지 않는다는 장점이 있지만 태스크의 처리 순서가 보장되지 않는 단점이 있다.

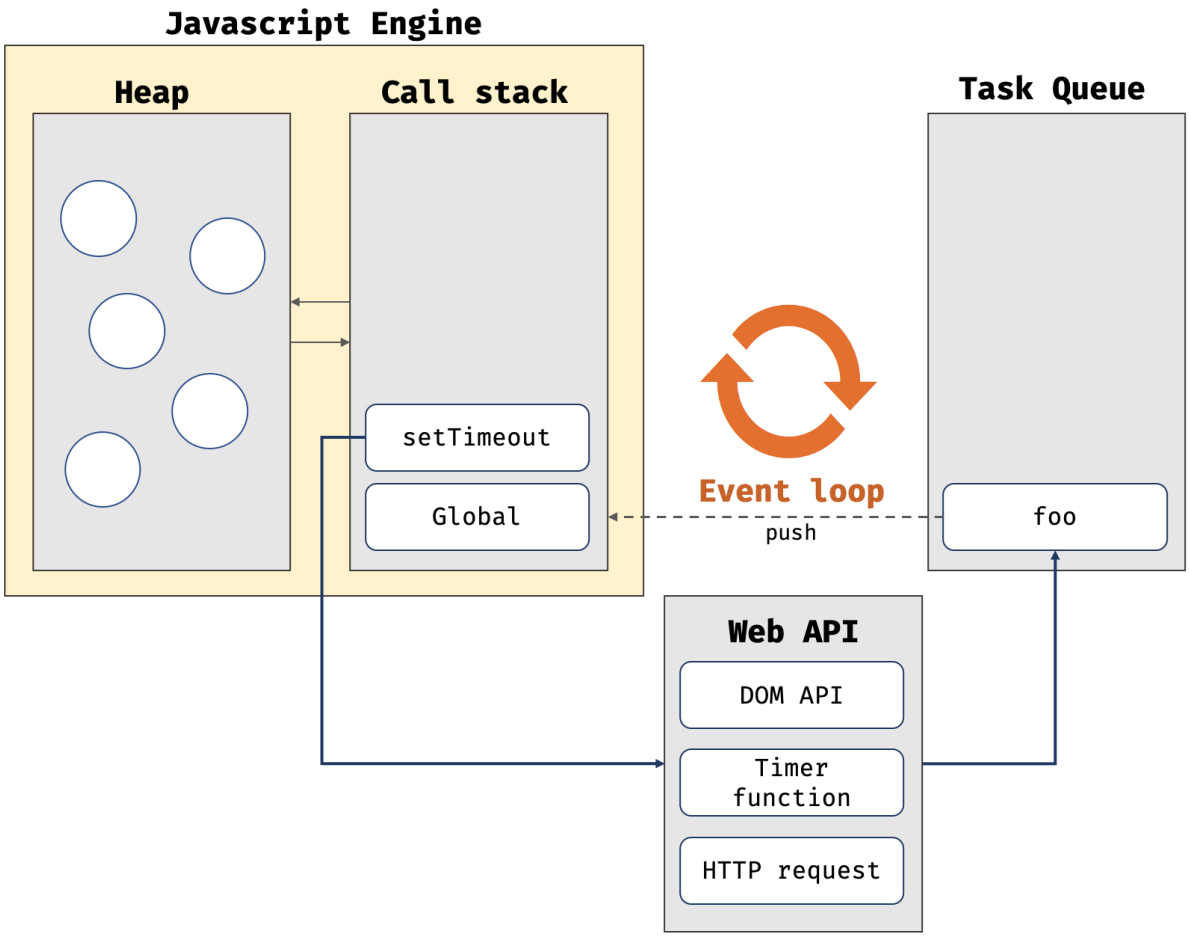
따라서 비동기 처리 과정에서 순차적인 처리가 필요한 경우 일반적으로 콜백 패턴을 사용했다. 비동기 처리를 위한 콜백 패턴은 콜백 헬을 발생시켜 가독성을 나쁘게 하고, 비동기 처리 중 발생한 에러의 예외 처리가 곤란하며, 여러 개의 비동기 처리를 한번에 처리하는 것도 한계가 있다. 이에 대해서는 “45. 프로미스”에서 자세히 살펴보자.

타이머 함수인 setTimeout과 setInterval, HTTP 요청은 비동기식 처리 모델로 동작한다. 비동기식 처리 모델은 자바스크립트에 동시성(concurrency)을 부여한다. 동시성은 이벤트 루프로 구현된다.

2. 이벤트 루프와 동시성

자바스크립트의 특징 중 하나는 싱글 스레드로 동작한다는 것이다. 앞서 살펴본 바와 같이 싱글 스레드 방식은 한번에 하나의 태스크만 처리할 수 있다는 것을 의미한다. 하지만 브라우저가 동작하는 것을 살펴보면 많은 태스크가 동시에 처리되는 것처럼 느껴진다.

예를 들어, HTML 요소가 애니메이션 효과를 통해 움직이면서 이벤트를 처리하기도 하고, HTTP 요청을 통해 서버로부터 데이터를 가지고 오면서 렌더링을 하기도 한다. 이처럼 자바스크립트의 동시성(concurrency)을 지원하는 것이 바로 **이벤트 루프**(event loop)이다. 브라우저 환경을 그림으로 표현하면 다음과 같다.



이벤트 루프와 브라우저 환경

구글의 V8 자바스크립트 엔진을 비롯한 대부분의 자바스크립트 엔진은 크게 2개의 영역으로 구분할 수 있다.

- 콜 스택(call stack, 실행 컨텍스트 스택)
  "23. 실행 컨텍스트"에서 살펴본 바와 같이 소스코드(전역 코드나 함수 코드 등)의 평가에 의해 생성된 실행 컨텍스트가 추가(push)되고 제거(pop)되는 스택 자료구조인 실행 컨텍스트 스택이 바로 콜 스택이다.
  함수를 호출하면 함수 실행 컨텍스트가 순차적으로 콜 스택에 푸시되어 순차적으로 실행된다. 자바스크립트 엔진은 단 하나의 콜 스택을 사용하기 때문에 최상위 실행 컨텍스트(실행 중인 실행 컨텍스트)가 종료되어 콜 스택에서 제거되기 이전까지는 다른 어떤 태스크도 수행되지 않는다.
- 힙(heap)
  할당해야 할 메모리 공간의 크기를 런타임에 결정(동적 할당)해야 하는 객체가 저장되는 메모리 공간으로 구조화되어 있지 않다. 콜 스택의 요소는 힙에 저장된 객체를 참조한다.

이와 같이 자바스크립트 엔진은 단순히 태스크가 요청되면 콜 스택을 통해 요청된 작업을 순차적으로 실행할 뿐이다. 앞에서 언급한 동시성을 지원하기 위해 필요한 비동기 요청 처리(예를 들어, setTimeout의 호출 스케줄링을 위한 타이머 설정과 콜백 함수 등록)는 자바스크립트 엔진을 구동하는 환경 즉 브라우저 또는 Node.js가 담당한다. 이를 위해 브라우저 환경은 태스크 큐와 이벤트 루프를 제공한다.

- 태스크 큐(task queue/event queue/callback queue)
  타이머 함수인 setTimeout이나 setInterval과 같은 비동기 처리 함수의 콜백 함수 또는 이벤트 핸들러가 일시적으로 보관되는 영역이다.
- 이벤트 루프(event loop)
  콜 스택에 현재 실행중인 실행 컨텍스트가 있는지 그리고 태스크 큐에 대기 중인 함수(콜백 함수, 이벤트 핸들러 등)가 있는지 반복해서 확인한다. **콜 스택이 비어졌을 때 태스크 큐에서 대기 중인 함수는 이벤트 루프에 의해 순차적**(FIFO, First In First Out)**으로 콜 스택으로 이동되어 실행된다**. 즉, 태스크 큐에 일시 보관된 함수들은 비동기식 처리 모델로 동작한다.

브라우저 환경에서 다음 예제가 어떻게 동작할지 살펴보도록 하자.

JAVASCRIPT

```
function foo() {
  console.log('foo');
}

function bar() {
  console.log('bar');
}

setTimeout(foo, 0); // 0초 후(실제는 4ms)에 foo 함수가 호출된다.
bar();
// bar → foo
```

1. 전역 코드가 평가되어 전역 실행 컨텍스트가 생성되고 콜 스택에 푸시된다.
2. 전역 코드가 실행되기 시작하여 타이머 함수 setTimeout이 호출된다. 이때 setTimeout의 함수 실행 컨텍스트가 생성되고 콜 스택에 푸시되어 현재 실행중인 실행 컨텍스트가 된다. 브라우저의 Web API(호스트 객체)인 타이머 함수도 함수이므로 함수 실행 컨텍스트를 생성한다.
3. 타이머 함수 setTimeout에 의해 타이머가 설정된다. setTimeout**은 브라우저의** Web API**이므로 타이머 설정 처리는 자바스크립트 엔진이 아니라 브라우저가 수행한다**. 이후 브라우저에 설정된 타이머가 만료되면 콜백 함수 foo가 태스크 큐에 푸시된다. 위 예제의 경우 지연 시간(delay)이 0이지만 지연 시간이 4ms 이하인 경우 최소 지연 시간 4ms가 지정된다. 따라서 4ms **후에 콜백 함수** foo**가**

태스크 큐에 푸시되어 대기하게 된다. 이 처리 또한 자바스크립트 엔진이 아니라 브라우저가 수행한다. 이후 setTimeout 함수가 종료되어 콜 스택에서 팝된다.
이처럼 setTimeout으로 호출 스케줄링한 콜백 함수는 정확히 지연 시간 후에 호출된다는 보장은 없다. 지연 시간 이후에 콜백 함수가 태스크 큐에 푸시되어 대기하게 되지만 콜 스택이 비어야 호출되므로 약간의 시간차가 발생할 수 있기 때문이다.

4. setTimeout이 종료되어 콜 스택에서 팝되고 bar 함수가 호출된다. **이때 foo 함수는 아직 태스크 큐에서 대기 중이다**. bar 함수가 호출되면 bar 함수의 함수 실행 컨텍스트가 생성되고 콜 스택에 푸시되어 현재 실행중인 실행 컨텍스트가 된다. 이후 bar 함수가 종료되어 콜 스택에서 팝된다.
5. 전역 코드 실행이 종료되고 전역 실행 컨텍스트가 콜 스택에서 팝된다. 이로서 콜 스택은 아무런 실행 컨텍스트도 존재하지 않게 된다.
6. 이벤트 루프에 의해 콜 스택이 비어 있음이 감지되고 3에서 setTimeout에 의해 태스크 큐에 푸시되어 대기 중인 콜백 함수 foo가 이벤트 루프에 의해 콜 스택에 푸시된다. 다시 말해, 콜백 함수 foo의 함수 실행 컨텍스트가 생성되고 콜 스택에 푸시되어 현재 실행중인 실행 컨텍스트가 된다. 이후 foo함수가 종료되어 콜 스택에서 팝된다.

이처럼 **비동기 함수인 setTimeout의 콜백 함수는 태스크 큐에 푸시되어 대기하다가 콜 스택이 비게 되면, 다시 말해 전역 코드 및 명시적으로 호출된 함수가 모두 종료하면 비로소 콜 스택에 푸시되어 실행된다**.

자바스크립트는 싱글 스레드 방식으로 동작한다. 이때 **싱글 스레드 방식으로 동작하는 것은 브라우저가 아니라 자바스크립트 엔진이라는 것에 주의하기 바란다**. **만약 모든 자바스크립트 코드가 자바스크립트 엔진에서 싱글 스레드 방식으로 동작한다면 자바스크립트는 비동기식으로 동작할 수 없다**.

예를 들어, 타이머 함수 setTimeout의 모든 처리가 자바스크립트 엔진에서 수행된다고 가정해 보자. 이때 setTimeout의 호출 스케줄링을 위한 타이머 설정도 자바스크립트 엔진에서 수행될 것이므로 대기 시간 동안 어떤 태스크도 실행할 수 없다(앞에서 살펴본 sleep 함수를 떠올려 보자). 즉, setTimeout의 타이머 설정은 자바스크립트 엔진에서 싱글 스레드 방식으로 동작해서는 비동기식으로 동작할 수 없다.

브라우저는 자바스크립트 엔진 이외에도 렌더링 엔진과 Web API를 제공한다. Web API는 ECMAScript 사양에 정의된 함수가 아니라 브라우저에서 제공하는 API이며 DOM API와 타이머 함수, HTTP 요청(Ajax)와 같은 비동기 처리를 포함한다. 위 예제에서 살펴봤듯이 브라우저의 Web API인 타이머 함수 setTimeout이 호출되면 콜 스택에 푸시되어 실행된다. 이때 setTimeout의 2가지 기능인 타이머 설정과 타이머가 만료하면 콜백 함수를 태스크 큐에 등록하는 처리는 자바스크립트 엔진이 아니라 브라우저가 실행한다. 즉, 브라우저와 자바스크립트 엔진이 협력하여 setTimeout을 수행한다.