

# 8월 23일 (월)

## 1. 백준 알고리즘 문제풀이 - 요세푸스 문제 1158번

### 요세푸스 문제



시간 제한	메모리 제한	제출	정답	맞은 사람	정답 비율
2 초	256 MB	48866	23750	16937	48.115%

### 문제

요세푸스 문제는 다음과 같다.

1번부터 N번까지 N명의 사람이 원을 이루면서 앉아있고, 양의 정수 K( $\leq N$ )가 주어진다. 이제 순서대로 K번째 사람을 제거한다. 한 사람이 제거되면 남은 사람들로 이루어진 원을 따라 이 과정을 계속해 나간다. 이 과정은 N명의 사람이 모두 제거될 때까지 계속된다. 원에서 사람들이 제거되는 순서를 (N, K)-요세푸스 순열이라고 한다. 예를 들어 (7, 3)-요세푸스 순열은 <3, 6, 2, 7, 5, 1, 4>이다.

N과 K가 주어지면 (N, K)-요세푸스 순열을 구하는 프로그램을 작성하시오.

### 입력

첫째 줄에 N과 K가 빈 칸을 사이에 두고 순서대로 주어진다. ( $1 \leq K \leq N \leq 5,000$ )

### 출력

예제와 같이 요세푸스 순열을 출력한다.

#### 예제 입력 1

복사

7 3

#### 예제 출력 1

복사

<3, 6, 2, 7, 5, 1, 4>

```
python3 basic1158.py > ...
1 # 백준 1158번 요세푸스 문제
2
3 N,K = map(int,input().split())
4 arr = [i for i in range(1,N+1)]      # 맨 처음에 원에 앉아있는 사람들
5
6 answer = []    # 제거된 사람들을 넣을 배열
7 num = 0        # 제거될 사람의 인덱스 번호
8
9 for t in range(N):
10     num += K-1
11     if num >= len(arr):    # 헛바퀴를 돌고 그다음으로 돌아올때를 대비해 값을 나머지로 바꿈
12         num = num%len(arr)
13
14     answer.append(str(arr.pop(num)))
15
16 print("<", " ".join(answer)[:], ">", sep='')
```

- 알게된 것

### array.pop([*i*])

배열에서 인덱스 *i*에 있는 항목을 제거하고 이를 반환합니다. 선택적 인자의 기본값은 -1이므로, 기본적으로 마지막 항목이 제거되고 반환됩니다.

array.pop( )의 인자에는 인덱스가 들어가야 한다. 디폴트 값은 -1이다.

## 2. Codecademy - TDD Fundamentals - Learn TDD with Mocha

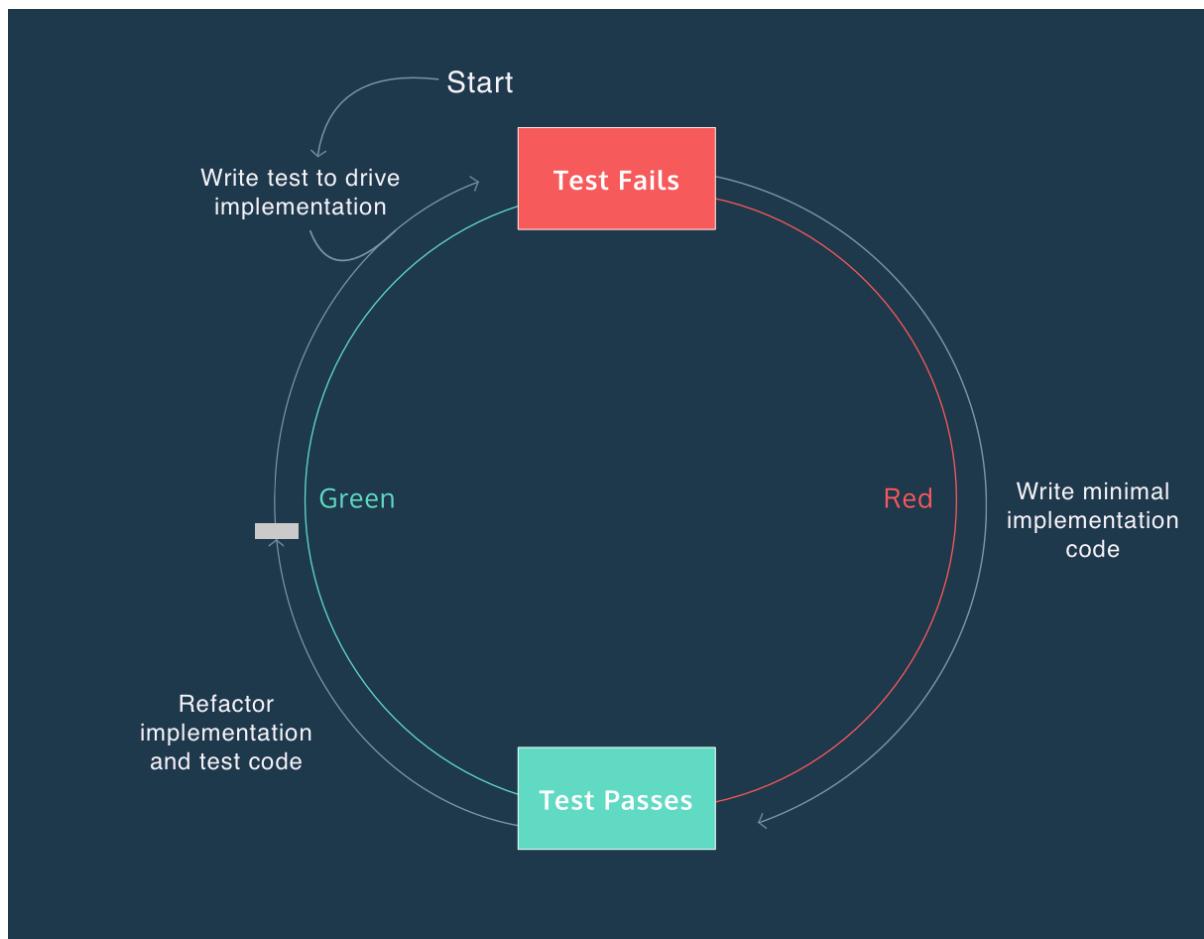
### Lesson 1. Learn TDD with Mocha

- Introduction
- Getting Into the Red -1-
- Red To Green -1-
- Refactor -1-
- Getting Into the Red -2-
- Red to Green -2-
- Refactor -2-
- Edge Case
- Review

- **Introduction**

## The Red-Green-Refactor Cycle

TDD 의 원동력중 하나는 red-green-refactor 사이클이다. 테스트시 red는 우리의 터미널에 나타나는 오류를 뜻하고, green 은 pass 를 뜻한다.



- Red - 구현코드의 의도된 행위를 설명하는 테스트를 작성하고, 구현 코드의 실제 결과와 기대 결과를 비교한다. 작성된 테스트는 처음엔 항상 실패해야한다. 왜냐하면 실패된

테스트에 맞춰서 올바른 코드를 쓸 수 있기 때문이다.

- Green - 테스트를 통과하기 위한 구현코드를 딱 알맞게 작성한다. 레드 단계를 거쳐 의도된 행위를 실행하기 때문에 그 테스트는 green (pass)을 출력해낸다.
- Refactor - characteristics of a good test 에 따라 코드를 정리하고 최적화한다. 리팩토링은 활동적으로 테스트와 구현코드를 고려하는 것과 코드 베이스를 수정하는 것을 포함한다. 테스트가 패스 되고, 모든 사이클 단계에서도 계속해서 패스되어야 한다.

이번 레슨에서는 배열내 숫자의 합을 계산하는 메서드를 짜기 위해 TDD 접근법을 사용할 것이다.

- **Getting Into the Red -1-**

레드 단계에서 의도적으로 에러를 띄우는 코드를 작성하면서, 퇴보하는 느낌이 들 수도 있다. 하지만 이러한 과정을 통해 에러를 제거함으로써 우리가 구현할 코드를 더욱 더 명확히 할 수 있다. 예시에서 우리는 Phrase 객체 내에 있는 .initials() 메서드(인자 안에 스트링을 모두 대문자로 변환)를 만들고 테스트 해볼 것이다.

#### Step 1. Write The Test

먼저 describe 와 it 을 사용하여 테스트 구문을 작성하자. 오류에 대한 메세지를 적는 것은 매우 중요하므로, 어떤 오류인 지 잘 알아볼 수 있도록 쓰자.

```
describe('Phrase', () => {
  describe('.initials', () => {
    it('returns the first letter of each word in a phrase.', () => {
      assert.equal(Phrase.initials('Nelson Mandela'), 'NM');
    })
  })
})
```

## Step 2. Run the test

```
Phrase
  .initials
    1) returns the first letter of each word
    in a phrase.

    0 passing (6ms)
    1 failing

    1) Phrase .initials returns the first letter
    of each word in a phrase.:
      ReferenceError: Phrase is not defined
```

에러가 뜰 것이다.

## Step 3. The test fails (yea!)

에러 메세지는 `Phrase.initials` 코드 블록과 관련이 있다고 우리에게 말하고 있다.  
`ReferenceError` 는 우리가 `Phrase` 객체를 가지고 있지 않기 때문에 에러가 났다고 말해준다.

## 실습 예제

```
1 const assert = require('assert');
2 const Calculate = require('../index.js')
3
4 ▼ describe('Calculate', () => {
5   ▼ describe('.sum', () => {
6     ▼ it('returns the sum of an array of numbers', () => {
7       // Code here
8       assert.equal(Calculate.sum([1,2,3]), 6);
9     });
10    });
11  });


```

- **Red To Green -1-**

레드 에러 메세지는 구현 코드의 실패를 설명한다. 이로써 테스트에 통과하지 못하게 하는 각각의 이유(문제)를 구체적으로 다를 수 있다. 다음 아래 Phrase.initials() 예시를 보자.

```
Phrase
```

```
.initials
```

```
1) returns the first letter of each word  
in a phrase.
```

```
0 passing (6ms)
```

```
1 failing
```

```
1) Phrase .initials returns the first letter  
of each word in a phrase.:
```

```
ReferenceError: Phrase is not defined
```

에러는 Phrase is not defined 라고 우리게 말하고 있다. 이는 우리가 Phrase 객체를 아직 생성하지 않았기 때문에 생기는 에러이다.

```
const Phrase = {}
```

다음 Phrase 객체를 만든 후 다시 테스트해보자.

```
Phrase
  .initials
    1) returns the first letter of each word in
    a phrase.

0 passing (6ms)
1 failing

1) Phrase .initials returns the first letter of
each word in a phrase.:
TypeError: Phrase.initials is not a function
```

이번엔 새로운 에러인 TypeError 가 등장했다. 에러는 객체 내에 initials 메서드가 없다고 하고 있다.

```
const Phrase = {
  initials() {
  }
}
```

다음과 같이 메서드를 객체내에 추가하자.

```
Phrase
```

```
.initials
```

```
 1) returns the first letter of each word  
in a phrase.
```

```
0 passing (7ms)
```

```
1 failing
```

```
1) Phrase .initials returns the first letter  
of each word in a phrase.:
```

```
AssertionError: undefined == 'NM'
```

또 다시 새로운 에러가 등장한다. 이것은 메서드가 return 을 쓰지 않았기에 undefined 를 출력한 것이다.

```
assert.equal(Phrase.initials('Nelson Mandela'), 'NM');
```

```
const Phrase = {  
  initials(phr) {  
    return 'NM';  
  }  
}
```

일단 테스트를 통과하기 위해 return 'NM' 을 작성해준다.

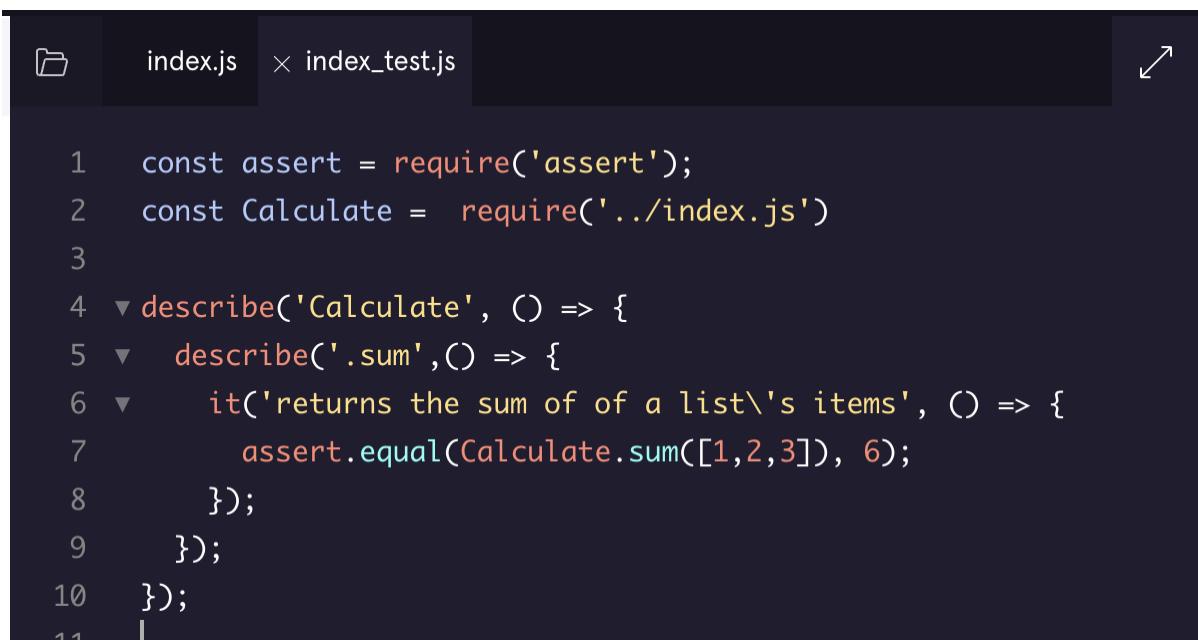
우리의 테스트는 통과하였다. 하지만 우리의 메서드가 의도한대로 작동한다는 것은 아니다. 만약 우리가 "Jody Williams" 를 입력하면 'JW' 를 리턴해야하는데 여전히 'NM' 를 리턴하고 있다. 이러한 이슈는 refactor 단계에서 다룰 것이다.

## 실습 예시



The screenshot shows a code editor interface with two tabs: 'index.js' and 'index\_test.js'. The 'index.js' tab is active, displaying the following code:

```
1 ▼ const Calculate = {
2   ▼   sum(arr) {
3       return 6;
4   }
5 };
6
7 module.exports = Calculate;
8
9
10
```



The screenshot shows a code editor interface with two tabs: 'index.js' and 'index\_test.js'. The 'index\_test.js' tab is active, displaying the following code:

```
1 const assert = require('assert');
2 const Calculate = require('../index.js')
3
4 ▼ describe('Calculate', () => {
5   ▼   describe('.sum', () => {
6       it('returns the sum of of a list\'s items', () => {
7           assert.equal(Calculate.sum([1,2,3]), 6);
8       });
9   });
10 });
11 |
```

- **Refactor -1-**

일단 테스트가 패스하면, 확신을 가지고 너의 코드를 리팩토링 할 수 있다. 즉, 외부적인 행위의 변경없이 재구성 및 향상 시킬 수 있다. 그러한 자신감은 우리의 테스트가 우리의 실수를 캐치 해줄거라는 것을 아는것으로부터 나온다.

리팩토링을 할 때, 테스트를 빨리, 종종하는 것이 중요하다. 즉, 만약 우리의 테스트가 레드로 변한다면, 리팩토리중에 무언가 잘못 되었다는 것을 알 수 있으며, 그것을 바탕으로 우리는 코드에 그러한 변화를 주지 않을 수 있다.

리팩토링을 하기에 좋은 시작은 좋은 테스트 4단계를 반영하는 테스트로 재구성하여 나누는 것이다. ( setup, exercise, verification, teardown )

아래 Phrase.initials 메서드를 4단계로 리팩토링 해보자.

```
describe('Phrase', () => {
  describe('.initials', () => {
    it('returns the first letter of each word in a phrase.', () => {
      // Setup
      const inputName = 'Nelson Mandela';
      const expectedInitials = 'NM';
      // Exercise
      const result = Phrase.initials(inputName);
      // Verification
      assert.equal(result, expectedInitials);
    });
  });
});
```

테스트 코드를 리팩토링한 것이다.

```
const Phrase = {
  initials(phr) {
    return 'NM';
  }
}
```

테스트 코드를 리팩토링 했으니, 구현 코드도 리팩토링 해주자.

```
const Phrase = {
  initials(inputName) {
    return 'NM';
  }
}
```

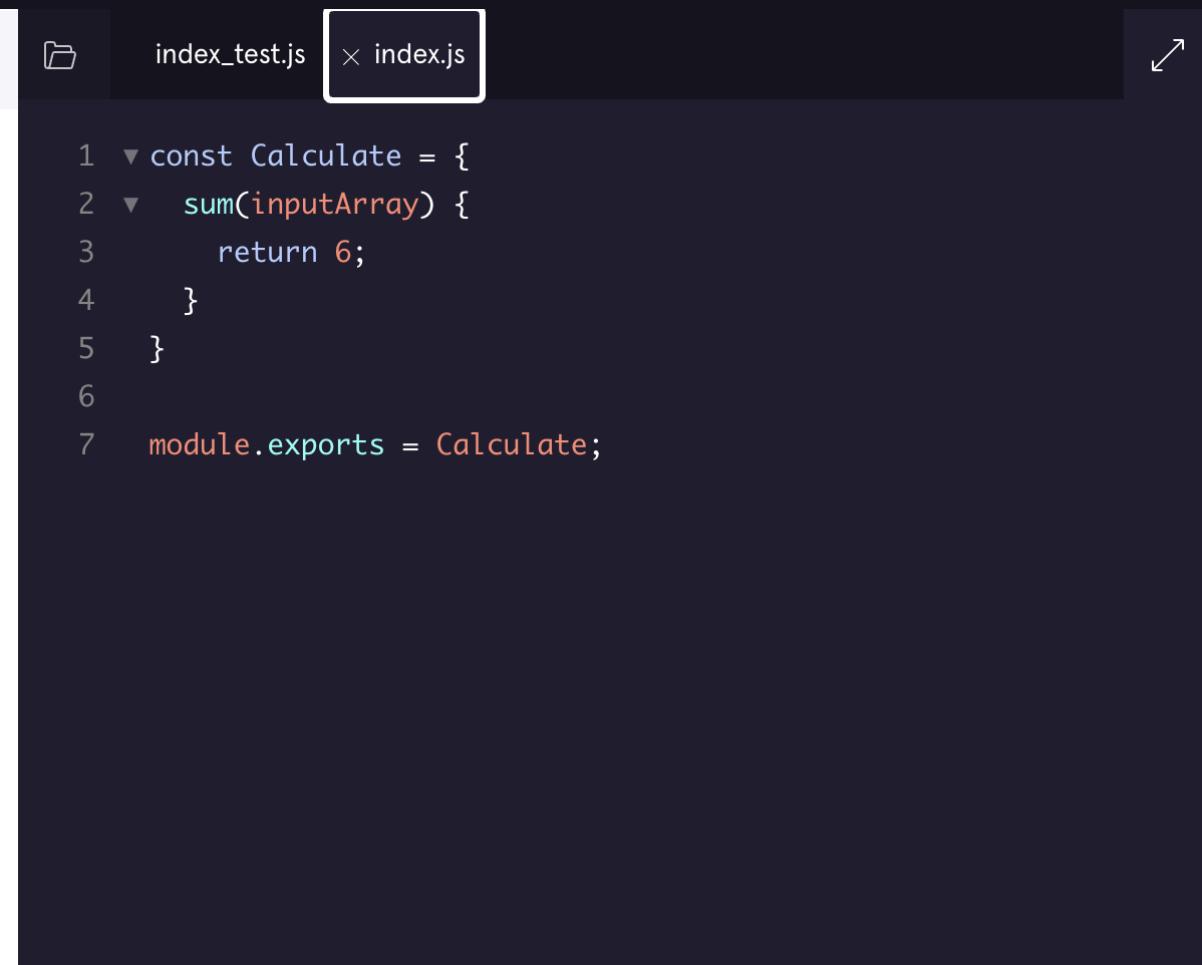
리팩토링은 각 프로젝트나 각 red-green-refactor 사이클에서 모두 다르게 보일 것이다. 몇몇 케이스에서는 리팩토링을 안해도 된다! 비록 리팩토링이 옵션이긴 하지만, 코드를 어떻게 향상할 수 있는지 항상 고려해야한다.

## 실습 예제



The screenshot shows a code editor interface with two tabs at the top: 'index\_test.js' and 'index.js'. The 'index\_test.js' tab is active, displaying the following Jest test code:

```
1 const assert = require('assert');
2 const Calculate = require('../index.js')
3
4 ▼ describe('Calculate', () => {
5   ▼ describe('.sum', () => {
6     ▼ it('returns the sum of an array of numbers', () => {
7       //Set up
8       const expectedResult = 6;
9       const inputArray = [1,2,3]
10      // exercise
11      const result = Calculate.sum(inputArray)
12      // verification
13      assert.equal(result, expectedResult);
14    });
15  });
16});
17
```



```
1 ▼ const Calculate = {
2   ▼   sum(inputArray) {
3       return 6;
4   }
5 }
6
7 module.exports = Calculate;
```

- **Getting Into the Red -2-**

축하합니다! TDD 를 사용하여 우리의 첫번째 red-green-refactor 사이클을 성공했다. 우리는 더 완벽한 메서드 구현 코드를 작성하기 위해 싸이클을 반복할 것이다.

일단 기능에 대한 베이스라인 테스트를 가졌다면, 더 나은 구현 코드를 작성하기 위한 추가적인 테스트 케이스 작성을 시작할 수 있다.

Phrase.initials 메서드 테스트 슈트를 고려해보자. 우리는 Phrase.initials("Nelson Mandela") 가 "NM"을 리턴하는지 체크하는 테스트를 가지고 있다.

```
describe('Phrase', () => {
  describe('.initials', () => {
    it('returns the first letter of each word in a phrase.', () => {
      const inputName = 'Nelson Mandela';
      const expectedInitials = 'NM';
      const result = Phrase.initials(inputName);
      assert.equal(result, expectedInitials);
    });
  });
});
```

NM 뿐만 아니라 다른 이름을 넣을 수 있는 테스트케이스를 작성해보자.

```
describe('Phrase', () => {
  describe('.initials', () => {

    . . .

    it('returns the initials of a name', () => {
      const nameInput = 'Juan Manuel Santos';
      const expectedInitials = 'JMS';

      const result = Phrase.initials(nameInput);

      assert.equal(result, expectedInitials);
    });
  });
});
```

비록 전 it 문과 비슷하지만, 두 가지의 경우의 수로 테스트를 통과할 수 있도록 하는 구문이다.

이처럼 TDD는 구현코드가 기대했던대로 수행할 것이라는 자신감을 가질 때까지 계속해서 테스트를 추가하고, red-green-refactor 과정을 겪어나가는 것이다.

## 실습 예제

```
1  const assert = require('assert');
2  const Calculate = require('../index.js')
3
4  describe('Calculate', () => {
5    describe('.sum', () => {
6      it('returns the sum of an array of numbers', () => {
7        const expectedResult = 6;
8        const inputArray = [1,2,3]
9
10       const result = Calculate.sum(inputArray)
11
12       assert.equal(result, expectedResult);
13     });
14
15     // Second test goes here
16   it('returns the sum of a four-item list', () => {
17     const expectedResult = 22;
18     const inputArray = [4, 5, 6, 7]
19
20     const result = Calculate.sum(inputArray)
21
22     assert.equal(result, expectedResult);
23   });
24
25 });
26 });
27 
```

```
$ npm test
> learn-mocha-tdd-tdd-with-mocha-red-ii@1.0.0 test /home/ccuser/workspace/learn-mocha-tdd-tdd-with-mocha-red-ii
> mocha test/**/*_test.js

  Calculate
    .sum
      ✓ returns the sum of an array of numbers
      1) returns the sum of a four-item list

      1 passing (7ms)
      1 failing

    1) Calculate .sum returns the sum of a four-item list:

        AssertionError: 6 == 22
        + expected - actual

        -6
        +22

        at Context.it (test/index_test.js:22:14)

npm ERR! Test failed. See above for more details.
```

첫 번째 it 블록 아래에 다른 테스트를 추가해준다. 두 번째 it 은 에러가 뜬다. 이는 구현코드의 리턴값이 그대로 6이기 때문이다.

- **Red to Green -2-**

1개 이상의 조건을 다루는 테스트를 거쳤으니, red 단계에 대한 오류를 고칠 차례이다.

```
1) Phrase .initials returns the initials of  
a name:
```

```
AssertionError: 'NM' == 'JMS'  
+ expected - actual  
  
-NM  
+JMS
```

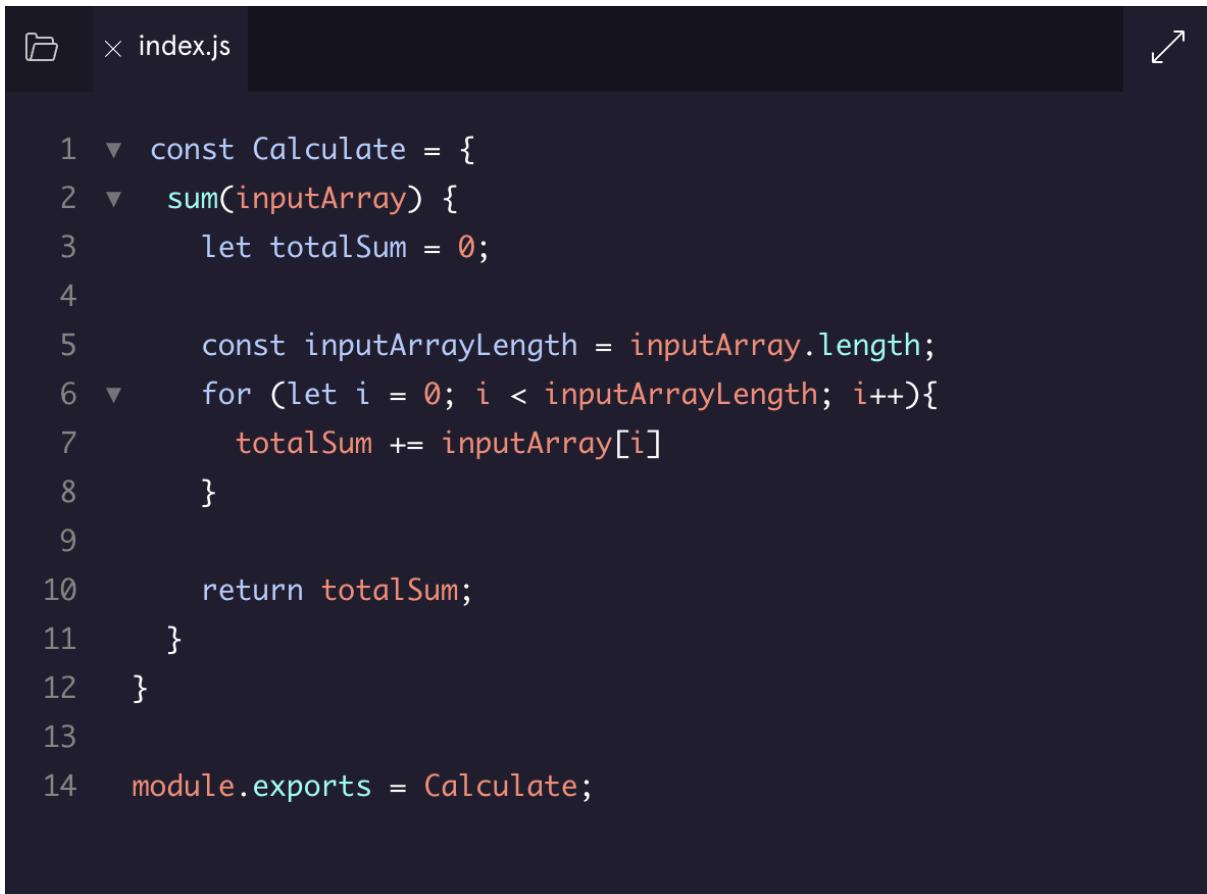
에러가 뜰 수 밖에 없는게 구현코드의 return 값이 여전히 "NM" 이기 때문이다.

우리는 구현코드를 다음과 같이 변경할 수 있다.

```
const Phrase = {  
  initials(inputName) {  
    // Create an empty array for initials  
    const initials = [];  
    // Create an array of strings  
    const words = inputName.split(" ");  
    // Iterate through the array of strings and push the  
    // first character of each to array  
    words.forEach((word) => {  
      initials.push(word.charAt(0));  
    });  
    // Return the initials as one string  
    return initials.join("");  
  }  
}
```

이 구현코드는 모든 인풋값에 대해 성공적으로 대문자들을 리턴할 것이다.

## 실습 예제



A screenshot of a code editor window titled "index.js". The code is a JavaScript function named "Calculate" that contains a "sum" method. The "sum" method takes an array ("inputArray") and calculates its total sum by iterating through it. The code is numbered from 1 to 14. The editor has a dark theme with syntax highlighting.

```
1  ▼ const Calculate = {
2   ▼   sum(inputArray) {
3       let totalSum = 0;
4
5       const inputArrayLength = inputArray.length;
6       for (let i = 0; i < inputArrayLength; i++){
7           totalSum += inputArray[i]
8       }
9
10      return totalSum;
11  }
12 }
13
14 module.exports = Calculate;
```

```
$ npm test

> learn-mocha-tdd-tdd-with-mocha-green-ii@1.0.0 test /home/c
cuser/workspace/learn-mocha-tdd-tdd-with-mocha-green-ii
> mocha test/**/*_test.js

Calculate
  .sum
    ✓ returns the sum of an array of numbers
    ✓ returns the sum of a four item list

  2 passing (6ms)

$ █
```

- **Refactor -2-**

축하합니다! 우리는 이제 다시 green 단계에 있습니다. 이제 우리는 코드를 리팩토링할 단계에 다시 오게 된 것이다. 이를 통해서 너의 테스트는 더 빠르고, 완벽하며, 믿음직스럽고, 고립적이며, 유지보수성을 지니고, 표현력을 가지게 되었다. 때로 리팩토링은 테스트와 구현코드에서 발생할 수 있으며 그렇지 않으면 아예 발생하지 않을 수 있다. 리팩토링의 목적은 불필요하고, 장황한 코드가 있는지 비판적으로 생각하여 코드를 더 깔끔하고 효율적으로 쓰는 것이다.

The screenshot shows a code editor window with a dark theme. The title bar says 'index.js'. The code in the editor is:

```
1  ▼ const Calculate = {  
2    ▼   sum(inputArray) {  
3      let totalSum = 0;  
4  
5      const inputArrayLength = inputArray.length;  
6      ▼   for (let i = 0; i < inputArrayLength; i++) {  
7          totalSum += inputArray[i]  
8      }  
9  
10     return totalSum;  
11   }  
12 }  
13  
14 module.exports = Calculate;
```

이전 연습에서 작성한 구현 코드이다.

```
1  ▼ const Calculate = {
2    ▼   sum(inputArray) {
3
4      ▼     return inputArray.reduce((accumulator, currentValue) => {
5          return accumulator + currentValue;
6        })
7      }
8    }
9
10   module.exports = Calculate;
11
12
```

다음 구현코드를 array.prototype.reduce( ) 메서드를 사용하여 이렇게 리팩토링이 가능하다. reduce( )는 이터레이터함수이며 배열의 누적값을 구하는데에 용이하다.

- **Edge Case**

지금까지 리팩토링을 했으니, 이제 edge case 를 고려할 차례이다.

엣지 케이스는 오직 극단적인(맥시멈 or 미니멈) 연산 파라미터에서 일어나는 문제나 상황이다. 즉, 너가 설명해야하는 특별한 케이스로서 생각하면 된다.

Phrase.initials( ) 의 예로 들어보자. 만약 인자에 스트링이 아니라 숫자를 넣는다면 어떤 일이 발생할까? 이럴 경우 오직 스트링만 사용하라고 지시하는 에러 메세지를 발생하도록 하는 행위를 테스트나 구현 코드에 작성할 수 있다.

인자에 숫자를 전달했을 때 에러를 던지는 테스트는 다음과 같이 작성할 수 있다.

```
it('raises an error if the parameter passed in is not a string', () => {
  // Setup
  const nameInput = 14;
  // Exercise
  const exercise = () => { Phrase.initials(nameInput) };
  // Verification
  assert.throws(exercise, /only use string/);
})
```

assert.throws 메서드를 사용한다.

다음은 구현 코드에 다음과 같은 에러 메세지 함수를 작성해준다.

```
if (typeof inputName !== "string") {
  throw new Error("ERROR: only use string");
}
```

```
const Phrase = {
  initials(inputName) {
    if (typeof inputName !== "string") {
      throw new Error("ERROR: only use string");
    }
    const initials = [];
    const words = inputName.split(" ");

    words.forEach((word) => {
      initials.push(word.charAt(0));
    });

    return initials.join("");
  }
}
```

이것이 엣지 케이스의 작성 예시이다.

실습 예제

```
0
1     assert.equal(result, expectedResult)
2 });
3
4 ▼ it('returns zero for an empty array', () => {
5     const expectedResult = 0
6     const inputArray = []
7
8     const result = Calculate.sum(inputArray)
9
10    assert.equal(result, expectedResult);
11 });
12
13});
```

테스트 코드에서 배열의 길이가 0이면 0을 출력하는 엣지 케이스를 추가하였다.

```
1 ▼ const Calculate = {
2 ▼   sum(inputArray) {
3 ▼     if(inputArray.length === 0) {
4       return 0;
5     }
6
7     ▼     return inputArray.reduce((sum, value) => {
8         return sum + value;
9     })
10   }
11 }
12
13 module.exports = Calculate;
14
```

구현 코드도 배열의 길이가 0이면 0을 출력하는 엣지 케이스를 추가하였다.

- Mocha 프레임워크 공식문서 : <https://mochajs.org/#getting-started>
- Chai assertion 라이브러리 공식문서 : <https://www.chaijs.com/guide/>

### 3. Async JavaScript and HTTP Requests - Basic of Asynchronous JavaScript

#### Article 1. General Asynchronous Programming Concepts ( 일반적인 비동기 프로그래밍 개념 )

- '비동기적'(Asynchronous) 이란?
  - Blocking code
  - Threads
  - Asynchronous code
- 
- '비동기적'(Asynchronous) 이란?

# '비동기적'(Asynchronous) 이란?

일반적으로, 프로그램의 코드는 순차적으로 진행됩니다. 한번에 한가지 사건만 발생하면서 말입니다. 만약 어떤 함수의 결과가 다른 함수에 영향을 받는다면, 그 함수는 다른 함수가 끝나고 값을 산출할 때까지 기다려야 합니다. 그리고 그 과정이 끝날 때 까지, 유저의 입장에서 보자면, 전체 프로그램이 모두 멈춘 것처럼 보입니다.

예를들면, 맥 유저라면 종종 회전하는 무지개색 커서(비치볼)를 본 적이 있을 것입니다. 이 커서는 오퍼레이팅 시스템이 이렇게 말하고 있는 것입니다. "당신이 지금 사용하고 있는 시스템은 지금 멈춰서서 뭔가가 끝나기를 기다려야만 합니다. 그리고 이 과정은 당신이 지금 무슨 일이 일어나고있는지 궁금해 할 만큼 오래 걸리고 있습니다."



이것은 당황스러운 경험이며, 특히 요즘과 같이 컴퓨터가 여러개 프로세서를 돌리는 시대에는 컴퓨터 성능을 효율적으로 쓰지 못하는 처사입니다. 당신이 다른 코어 프로세서에 다른 작업들을 움직이게 하고 작업이 완료되면 알려줄 수 있을 때, 무언가를 기다리는 것은 의미가 없습니다. 그 동안 다른 작업을 수행할 수 있고, 이것이 비동기 프로그래밍의 기본입니다. 이러한 작업을 비동기적으로 실행할 수 있는 API를 제공하는 것은 당신이 사용하고 있는 프로그래밍 환경(웹 개발의 경우 웹브라우저)에 달려 있습니다.

- - -

- **Blocking code**

## Blocking code

비동기 기법은 특히 웹 프로그래밍에 매우 유용합니다. 웹 앱이 브라우저에서 특정 코드를 실행하느라 브라우저에게 제어권을 돌려주지 않으면 브라우저는 마치 정지된 것처럼 보일 수 있습니다. 이러한 현상을 **blocking** 이라고 부릅니다. 자세히 정의하자면, 사용자의 입력을 처리하느라 웹 앱이 프로세서에 대한 제어권을 브라우저에게 반환하지 않는 현상입니다..

Blocking의 몇 가지 예를 살펴보겠습니다.

여기 [simple-sync.html](#) 예시가 있습니다. ([see it running live](#)), 하나의 버튼에 클릭 이벤트리스너를 지정하여 시 간이 오래 걸리는 처리를 하도록 하였습니다. (날짜를 천만번 계산하고 마지막에 콘솔에 날짜를 출력합니다.) 그리고 처리가 끝나면 페이지에 간단한 문장을 한 줄 출력합니다. :

```
const btn = document.querySelector('button');
btn.addEventListener('click', () => {
  let myDate;
  for(let i = 0; i < 10000000; i++) {
    let date = new Date();
    myDate = date
  }

  console.log(myDate);

  let pElem = document.createElement('p');
  pElem.textContent = 'This is a newly-added paragraph.';
  document.body.appendChild(pElem);
}) ;
```

이 예제를 실행할 때 JavaScript 콘솔을 열고 버튼을 클릭하면, 콘솔에 메시지가 출력되기 전 까지 페이지에 문장이 나타나지 않는다는 것을 알 수 있습니다. 코드는 위에서 아래로 순차적으로 실행되며, 아래쪽 코드는 위쪽 코드의 처리가 끝날 때 까지 실행되지 않습니다.

**Note:** 앞의 예제는 매우 비현실적입니다. 실제 웹 앱에서 날짜를 천만번 계산할 일은 없습니다. 실제로 보여주기 위해 극단적인 예시를 들었을 뿐입니다..

두 번째 예제를 살펴보겠습니다. [simple-sync-ui-blocking.html](#) ([see it live](#)), 페이지에 UI가 모두 표시되기 전 까지 사용자의 입력을 막는 좀 더 현실적인 예시입니다. 이번 예시에는 두 가지 버튼을 사용합니다. :

- "Fill canvas" 버튼을 클릭하면 `<canvas>` 태그에 100만개의 파란색 원을 채웁니다. (실행하면 원이 너무 많아서 원이 아니라 배경이 그냥 파란색으로 채워집니다.)
- "Click me for alert!" 버튼은 사용자에게 메시지를 출력합니다.

```
function expensiveOperation() {
  for(let i = 0; i < 1000000; i++) {
    ctx.fillStyle = 'rgba(0,0,255, 0.2)';
    ctx.beginPath();
    ctx.arc(random(0, canvas.width), random(0, canvas.height), 10, degToRad(0), degToRad(360), false);
    ctx.fill()
  }
}

fillBtn.addEventListener('click', expensiveOperation);

alertBtn.addEventListener('click', () =>
  alert('You clicked me!')
);
```

첫 번째 버튼을 클릭한 후 두 번째 버튼을 바로 클릭하면 경고 박스가 나타나지 않는 것을 확인할 수 있습니다. 첫 번째 버튼은 이벤트가 끝나기 전 까지 다음 작동을 막아버립니다.

**Note:** OK, in our case, it is ugly and we are faking the blocking effect, but this is a common problem that developers of real apps fight to mitigate all the time.

왜 이런 현상이 발생할까요? 답은 자바스크립트는 기본적으로 **single threaded**이기 때문입니다. 이 시점에서 **threads**의 개념을 소개할 필요가 있겠군요

## • Threads

## Threads

**Thread**는 기본적으로 프로그램이 작업을 완료하는데 사용할 수 있는 단일 프로세스입니다. 각 스레드는 한 번에 하나의 작업만 수행할 수 있습니다. :

```
Task A --> Task B --> Task C
```

위의 예시처럼 각 작업은 순차적으로 실행되며, 다음 작업을 시작하려면 앞의 작업이 완료되어야 합니다.

앞서 말했듯이, 많은 컴퓨터들이 현재 여러 개의 CPU코어를 가지고 있기 때문에, 한 번에 여러가지 일을 수행할 수 있습니다. Multiple thread를 지원하는 프로그래밍 언어는 멀티코어 컴퓨터의 CPU를 사용하여 여러 작업을 동시에 처리할 수 있습니다. :

```
Thread 1: Task A --> Task B  
Thread 2: Task C --> Task D
```

### JavaScript is single threaded

자바스크립트는 전통적으로 싱글 thread입니다. 컴퓨터가 여러 개의 CPU를 가지고 있어도 **main thread**라 불리는 단일 thread에서만 작업을 실행할 수 있습니다. 위의 예시는 아래처럼 실행됩니다. :

```
Main thread: Render circles to canvas --> Display alert()
```

JavaScript는 이러한 문제를 해결하기 위해 몇 가지 툴을 도입했습니다. [Web workers](#)는 여러 개의 JavaScript 청크를 동시에 실행할 수 있도록 worker라고 불리는 별도의 스레드로 보낼 수 있습니다. 따라서 시간이 오래 걸리는 처리는 worker를 사용해 처리하면 blocking 발생을 막을 수 있습니다..

```
Main thread: Task A --> Task C  
Worker thread: Expensive task B
```

위의 내용을 잘 기억하시고 다음 예제를 살펴보세요. [simple-sync-worker.html](#) (see it running live), JavaScript 콘솔을 함께 열어주세요. 이전 예시는 날짜를 천만 번 계산하고 페이지에 문장을 출력했지만, 이번엔 천만번 계산 전 문장을 페이지에 출력해줍니다. 더이상 첫 번째 작업이 두 번째 작업을 차단하지 않습니다.

- **Asynchronous code**

## Asynchronous code

Web worker는 꽤 유용하지만 이들도 한계가 있습니다. 주요한 내용은 Web worker는 [DOM](#)에 접근할 수 없습니다. — UI를 업데이트하도록 worker에게 어떠한 지시도 직접 할 수 없습니다. 두 번째 예시에서 worker는 100만개의 파란색 원을 만들 수 없습니다. 단순히 숫자만 계산합니다.

두 번째 문제는 worker에서 실행되는 코드는 차단되지 않지만 동기적으로 실행된다는 것 입니다. 이러한 문제는 함수를 사용할 때 발생합니다. 어떤 함수가 일의 처리를 위해 이전의 여러 프로세스의 결과를 return 받아야 할 경우입니다. 동기적으로 실행되면 함수 실행에 필요한 매개변수를 받아올 수 없는 경우가 생기므로 함수는 사용자가 원하는 기능을 제대로 실행할 수 없습니다.

```
Main thread: Task A --> Task B
```

이 예시에서 Task A는 서버로부터 이미지를 가져오고 Task B는 그 이미지에 필터를 적용하는 것과 같은 작업을 수행한다고 가정합니다. Task A를 실행하고 결과를 반환할 시간도 없이 Task B를 실행해버리면 에러가 발생할 것 입니다. 왜냐하면 Task A에서 이미지를 완전히 가져온 상태가 아니기 때문이죠.

```
Main thread: Task A --> Task B --> |Task D|
Worker thread: Task C -----> | |
```

이번 예시에선 Task D가 Task B와 Task C의 결과를 모두 사용한다고 가정합니다. Task B와 Task C가 동시에 아주 빠르게 결과를 반환하면 매우 이상적이겠지만, 현실은 그렇지 않습니다. Task D가 사용될 때 Task B, Task C 둘 중 어느 값이라도 입력이 되지 않을 경우 에러가 발생합니다.

이러한 문제를 해결하기 위해 브라우저를 통해 특정 작업을 비동기적으로 실행할 수 있습니다. 바로 [Promises](#)를 사용하는 것입니다. 아래 예시처럼 Task A가 서버에서 이미지를 가져오는 동안 Task B를 기다리게 할 수 있습니다. :

```
Main thread: Task A           Task B
Promise:      |__async operation__|
```

위의 작업은 다른 곳에서 처리가 되므로, 비동기 작업이 진행되는 동안 main thread가 차단되지 않습니다.

이번 문서에서 다룬 내용은 매우 중요한 내용입니다. 다음 문서에선 비동기 코드를 어떻게 쓸 수 있는지 살펴볼 계획이므로 끝까지 읽어주시면 좋겠습니다.

## • 결론

### 결론

현대의 소프트웨어 설계는 프로그램이 한 번에 두 가지 이상의 일을 할 수 있도록 비동기 프로그래밍을 중심으로 돌아가고 있습니다. 보다 새롭고 강력한 API를 이용하면서, 비동기로 작업해야만 하는 사례가 많아질 것입니다. 예전에는 비동기 코드를 쓰기가 힘들었습니다. 여전히 아직 어렵지만, 훨씬 쉬워졌습니다. 이 모듈의 나머지 부분에서는 비동기 코드가 왜 중요 한지, 위에서 설명한 일부 문제들을 방지하는 코드 설계 방법에 대해 자세히 알아봅시다.

## Article 2. Introducing Asynchronous JavaScript ( 비동기 자바스크립트 소개 )

- Synchronous JavaScript ( 동기적 자바스크립트 )
- Asynchronous JavaScript ( 비동기적 자바스크립트 )
- Async callbacks ( `async` 콜백함수 )
- Promises ( 프로미스 )
- The nature of asynchronous code ( 비동기적 코드의 특성 )
- Synchronous JavaScript ( 동기적 자바스크립트 )

## Synchronous JavaScript

[asynchronous \(en-US\)](#) JavaScript가 무엇인지 이해하려면, 우리는 [synchronous \(en-US\)](#) JavaScript가 무엇인지 알아야 합니다. 이 문서에선 이전 문서에서 본 정보의 일부를 요약하겠습니다.

이전의 학습 모듈에서 살펴본 많은 기능들은 동기식입니다. — 약간의 코드를 실행하면, 브라우저가 할 수 있는 한 빠르게 결과를 보여줍니다. 다음 예제를 살펴볼까요 ([see it live here](#) , and [see the source](#) ):

```
const btn = document.querySelector('button');
btn.addEventListener('click', () => {
  alert('You clicked me!');

  let pElem = document.createElement('p');
  pElem.textContent = 'This is a newly-added paragraph.';
  document.body.appendChild(pElem);
});
```



이 블럭에서 코드는 위에서 아래로 차례대로 실행됩니다. :

ing

1. DOM에 미리 정의된 `<button>` element를 참조합니다.
2. `click` 이벤트 리스너를 만들어 버튼이 클릭됐을 때 아래 기능을 차례로 실행합니다. :
  1. `alert()` 메시지가 나타납니다.
  2. 메시지가 사라지면 `<p>` element를 만듭니다.
  3. 그리고 `text content`를 만듭니다.
  4. 마지막으로 document body에 문장을 추가합니다.

각 작업이 처리되는 동안 렌더링은 일시 중지됩니다. [앞에서 말한 문서와 같이](#), [JavaScript는 single threaded](#)이기 때문입니다. 한 번에 한 작업만, 하나의 main thread에서 처리될 수 있습니다. 그리고 다른 작업은 앞선 작업이 끝나야 수행됩니다.

따라서 앞의 예제는 사용자가 OK 버튼을 누를 때까지 문장이 나타나지 않습니다. :

me-

  
This is a newly-added paragraph.

!

**Note:** 기억해두세요. `alert()` 는 동기 블럭을 설명하는데 아주 유용하지만, 실제 어플리케이션에서 사용하기엔 아주 꼼꼼합니다.

- **Asynchronous JavaScript ( 비동기적 자바스크립트 )**

# Asynchronous JavaScript

앞서 설명된 이유들 (e.g. related to blocking) 때문에 많은 웹 API기능은 현재 비동기 코드를 사용하여 실행되고 있습니다. 특히 일부 디바이스에서 어떤 종류의 리소스에 액세스하거나 가져오는 기능들에 많이 사용합니다. 예를 들어 네트워크에서 파일을 가져오거나, 데이터베이스에 접속해 특정 데이터를 가져오는 일, 웹 캠에서 비디오 스트림에 액세스하거나, 디스플레이를 VR 헤드셋으로 브로드캐스팅 하는 것 입니다.

동기적 코드를 사용하여 작업을 처리하는데 왜 이렇게 어려울까요? 다음 예제를 살펴보겠습니다. 서버에서 이미지를 가져오면 네트워크 환경, 다운로드 속도 등의 영향을 받아 이미지를 즉시 확인할 수 없습니다. 이 말은 아래 코드가 (pseudocode) 실행되지 않는다는 의미입니다. :

```
let response = fetch('myImage.png');
let blob = response.blob();
// display your image blob in the UI somehow
```

왜냐하면 앞서 설명했듯이 이미지를 다운로드 받는데 얼마나 걸릴지 모르기 때문입니다. 그래서 두 번째 줄을 실행하면 에러가 발생할 것 입니다. (이미지의 크기가 아주 작다면 에러가 발생하지 않을 수도 있습니다. 반대로 이미지의 크기가 크면 매번 발생할 것 입니다.) 왜냐하면 `response` 가 아직 제공되지 않았기 때문입니다. 따라서 개발자는 `response` 가 반환되기 전 까지 기다리게 처리를 해야합니다.

JavaScript에서 볼 수 있는 비동기 스타일은 두 가지 유형이 있습니다, 예전 방식인 callbacks 그리고 새로운 방식인 promise-style 코드입니다. 이제부터 차례대로 살펴보겠습니다.

자바스크립트의 비동기 스타일은 두 가지 유형이 있다. ⇒ 올드한 콜백함수와 뉴 프로미스

- **Async callbacks ( async 콜백함수 )**

## Async callbacks

Async callbacks은 백그라운드에서 코드 실행을 시작할 함수를 호출할 때 인수로 지정된 함수입니다. 백그라운드 코드 실행이 끝나면 callback 함수를 호출하여 작업이 완료됐음을 알리거나, 다음 작업을 실행하게 할 수 있습니다. callbacks를 사용하는 것은 지금은 약간 구식이지만, 여전히 다른 곳에서 사용이 되고 있음을 확인할 수 있습니다.

Async callback의 예시는 `addEventListener('click')` 옆의 두 번째 매개변수입니다. :

```
btn.addEventListener('click', () => {
  alert('You clicked me!');

  let pElem = document.createElement('p');
  pElem.textContent = 'This is a newly-added paragraph.';
  document.body.appendChild(pElem);
});
```

첫 번째 매개 변수는 이벤트 리스너 유형이며, 두 번째 매개 변수는 이벤트가 실행될 때 호출되는 콜백 함수입니다.

callback 함수를 다른 함수의 인수로 전달할 때, 함수의 참조를 인수로 전달할 뿐이지 즉시 실행되지 않고, 함수의 body에서 “called back”됩니다. 정의된 함수는 때가 되면 callback 함수를 실행하는 역할을 합니다.

[XMLHttpRequest API](#) ([run it live](#), and [see the source](#))를 불러오는 예제를 통해 callback 함수를 쉽게 사용해봅시다. :

```

function loadAsset(url, type, callback) {
  let xhr = new XMLHttpRequest();
  xhr.open('GET', url);
  xhr.responseType = type;

  xhr.onload = function() {
    callback(xhr.response);
  };

  xhr.send();
}

function displayImage(blob) {
  let objectURL = URL.createObjectURL(blob);

  let image = document.createElement('img');
  image.src = objectURL;
  document.body.appendChild(image);
}

loadAsset('coffee.jpg', 'blob', displayImage);

```

`displayImage()` 함수는 Object URL로 전달되는 Blob을 전달받아, URL이 나타내는 이미지를 만들어 <body>에 그립니다. 그러나, 우리는 `loadAsset()` 함수를 만들고 "url", "type" 그리고 "callback"을 매개변수로 받습니다. `XMLHttpRequest` (줄여서 "XHR")를 사용하여 전달받은 URL에서 리소스를 가져온 다음 `callback`으로 응답을 전달하여 작업을 수행합니다. 이 경우 `callback`은 `callback` 함수로 넘어가기 전, 리소스 다운로드를 완료하기 위해 XHR 요청이 진행되는 동안 대기합니다. (`onload` 이벤트 핸들러 사용)

Callbacks은 다재다능 합니다. 함수가 실행되는 순서, 함수간에 어떤 데이터가 전달되는지를 제어할 수 있습니다. 또한 상황에 따라 다른 함수로 데이터를 전달할 수 있습니다. 따라서 응답받은 데이터에 따라 (`processJSON()`, `displayText()` 등) 어떤 작업을 수행할지 지정할 수 있습니다.

모든 `callback`이 비동기인 것은 아니라는 것에 유의하세요 예를 들어 [Array.prototype.forEach\(\)](#) 를 사용하여 배열의 항목을 탐색할 때가 있습니다. ([see it live](#), and [the source](#)):

```

const gods = ['Apollo', 'Artemis', 'Ares', 'Zeus'];

gods.forEach(function (eachName, index){
  console.log(index + ' ' + eachName);
});

```

이 예제에선 그리스 신들의 배열을 탐색하여 인덱스 넘버와 그 값을 콘솔에 출력합니다. `forEach()` 매개변수는 callback 함수이며, callback 함수는 배열 이름과 인덱스 총 두 개의 매개변수가 있습니다. 그러나, 여기선 비동기로 처리되지 않고 즉시 실행됩니다..

- **Promises (프로미스)**

# Promises

Promises 모던 Web APIs에서 보게 될 새로운 코드 스타일입니다. 좋은 예는 `fetch()` API입니다. `fetch()`는 `XMLHttpRequest`보다 좀 더 현대적인 버전입니다. 아래 [Fetching data from the server](#) 예제에서 빠르게 살펴볼까요? :

```
fetch('products.json').then(function(response) {
  return response.json();
}).then(function(json) {
  products = json;
  initialize();
}).catch(function(err) {
  console.log('Fetch problem: ' + err.message);
});
```



Note: You can find the finished version on GitHub ([see the source here](#), and also [see it running live](#)).

`fetch()`는 단일 매개변수만 전달받습니다. — 네트워크에서 가지고 오고 싶은 리소스의 URL — 그리고 `promise`로 응답합니다. `promise`는 비동기 작업이 성공 했는지 혹은 실패했는지를 나타내는 하나의 오브젝트입니다. 즉 성공/실패의 분기점이 되는 중간의 상태라고 표현할 수 있죠. 왜 `promise`라는 이름이 붙었는지 잠깐 살펴보자면.. "내가 할수 있는 한 빨리 너의 요청의 응답을 가지고 돌아간다고 약속(promise)할게"라는 브라우저의 표현방식 이어서 그렇습니다.

이 개념에 익숙해 지기 위해서는 연습이 필요합니다.; 마치 [Schrödinger's cat](#) (슈뢰딩거의 고양이) 처럼 느껴질 수 있습니다. 위의 예제에서도 발생 가능한 결과 중 아직 아무것도 발생하지 않았기 때문에, 미래에 어떤 시점에서 `fetch()` 작업이 끝났을때 어떤 작업을 수행 시키기 위해 두 가지 작업이 필요합니다. 예제에선 `fetch()` 마지막에 세 개의 코드 블럭이 더 있는데 이를 살펴보겠습니다. :

- 두 개의 `.then()` 블럭: 두 함수 모두 이전 작업이 성공했을때 수행할 작업을 나타내는 callback 함수입니다. 그리고 각 callback함수는 인수로 이전 작업의 성공 결과를 전달받습니다. 따라서 성공했을때의 코드를 callback 함수 안에 작성하면 됩니다. 각 `.then()` 블럭은 서로 다른 promise를 반환합니다. 이 말은 `.then()` 을 여러개 사용하여 연쇄적으로 작업을 수행하게 할 수 있음을 말합니다. 따라서 여러 비동기 작업을 차례대로 수행할 수 있습니다.
- `.catch()` 블럭은 `.then()` 이 하나라도 실패하면 동작합니다. — 이는 동기 방식의 `try...catch` 블럭과 유사합니다. error 오브젝트는 `catch()` 블럭 안에서 사용할 수 있으며, 발생한 오류를 보고하는 용도로 사용할 수 있습니다. 그러나 `try...catch` 는 나중에 배울 `async/await`에서는 동작하지만, `promise`와 함께 동작할 수 없음을 알아두세요.

## The event queue

promise와 같은 비동기 작업은 **event queue**에 들어갑니다. 그리고 main thread가 끝난 후 실행되어 후속 JavaScript 코드가 차단되는것을 방지합니다. queued 작업은 가능한 빨리 완료되어 JavaScript 환경으로 결과를 반환해줍니다.

## Promises vs callbacks

Promises 는 old-style callbacks과 유사한 점이 있습니다. 기본적으로 callback을 함수로 전달하는 것이 아닌 callback 함수를 장착하고 있는 returned된 오브젝트 입니다.

그러나 Promise는 비동기 작업을 처리함에 있어서 old-style callbacks 보다 더 많은 이점을 가지고 있습니다. :

- 여러 개의 연쇄 비동기 작업을 할 때 앞서 본 예제처럼 `.then()` 을 사용하여 결과값을 다음 작업으로 넘길 수 있습니다. callbacks으로 이를 사용하려면 더 어렵습니다. 또한 "파멸의 피라미드" ([callback hell](#) ↗로 잘 알려진)을 종종 마주칠 수 있습니다.
- Promise callbacks은 event queue에 배치되는 임격한 순서로 호출됩니다.
- 에러 처리가 더 간결해집니다. — 모든 에러를 코드 블럭의 마지막 부분에 있는 단 한개의 `.catch()` 블럭으로 처리할 수 있습니다. 이 방법은 피라미드의 각 단계에서 에러를 핸들링하는 것 보다 더 간단합니다..
- 구식 callback은 써드파티 라이브러리에 전달될 때 함수가 어떻게 실행되어야 하는 방법을 상실하는 반면 Promise는 그렇지 않습니다.

- **The nature of asynchronous code ( 비동기적 코드의 특성 )**

## The nature of asynchronous code

코드 실행 순서를 완전히 알지 못할 때 발생하는 현상과 비동기 코드를 둘기 코드처럼 취급하려고 하는 문제를 살펴보면 서 비동기 코드의 특성을 더 살펴봅시다 아래 예제는 이전에 봤던 예제와 유사합니다. ([see it live](#)  and [the source](#) ). 한가지 다른 점은 코드가 실행순서를 보여주기 위해 `console.log()` 를 추가했습니다.

```
console.log('Starting');
let image;

fetch('coffee.jpg').then((response) => {
  console.log('It worked :)')
  return response.blob();
}).then((myBlob) => {
  let objectURL = URL.createObjectURL(myBlob);
  image = document.createElement('img');
  image.src = objectURL;
  document.body.appendChild(image);
}).catch((error) => {
  console.log('There has been a problem with your fetch operation: ' + error.message);
});

console.log('All done!');
```



브라우저는 코드를 실행하기 시작할 것이고, 맨 처음 나타난 (`Starting`) 글씨가 써진 `console.log()` 후 `image` 변수를 만들 것 입니다.

다음으로 `fetch()` block 으로 이동하여 코드를 실행하려고 할 것 입니다. `fetch()` 덕분에 blocking없이 비동기 적으로 코드가 실행되겠죠. 블럭 내에서 promise와 관련된 작업이 끝나면 다음 작업으로 이어질 것 입니다. 그리고 마지막으로 (`All done!`) 가 적혀있는 `console.log()` 에서 메시지를 출력할 것 입니다.

`fetch()` 블럭 작업이 작업을 완전히 끝내고 마지막 `.then()` 블럭에 도달해 애 `console.log()` 의 (`It worked :)`) 메시지가 나타납니다. 그래서 메시지의 순서는 코드에 적혀있는 대로 차례대로 나타나는게 아니라 순서가 약간 바뀌어서 나타납니다 :

- Starting
- All done!
- It worked :)

이 예가 어렵다면 아래의 다른 예를 살펴보세요 :

```
console.log("registering click handler");

button.addEventListener('click', () => {
  console.log("get click");
});

console.log("all done");
```

이전 예제와 매우 유사한 예제입니다. — 첫 번째와 세 번째 `console.log()` 메시지는 콘솔창에 바로 출력됩니다. 그러나 두 번째 메시지는 누군가가 버튼을 클릭하기 전엔 콘솔에 표시되지 않죠. 위의 예제의 차이라면 두 번째 메시지가 어떻게 잠시 blocking이 되는지입니다. 첫 예제는 Promise 체이닝 때문에 발생하지만 두 번째 메시지는 클릭 이벤트를 대기 하느라고 발생합니다.

less trivial 코드 예제에서 이러한 설정을 문제를 유발할 수 있습니다. — 비동기 코드 블럭에서 반환된 결과를 동기화 코드 블럭에 사용할 수 없습니다. 위의 예제에서 `image` 변수가 그 예시입니다. 브라우저가 동기화 코드 블럭을 처리하기 전에 비동기 코드 블럭 작업이 완료됨을 보장할 수 없습니다.

어떤 의미인지 확인을 하려면 [our example](#) 을 컴퓨터에 복사한 후 마지막 `console.log()` 를 아래처럼 고쳐보세요:

```
console.log ('All done! ' + image.src + ' displayed.');
```

고치고 나면 콘솔창에서 아래와 같은 에러가 뜨는 것을 확인할 수 있습니다. :

```
TypeError: image is undefined; can't access its "src" property
```

브라우저가 마지막 `console.log()` 를 처리할 때, `fetch()` 블럭이 완료되지 않아 `image` 변수에 결과가 반환되지 않았기 때문입니다.

비동기와 동기를 섞어서 사용하면 에러가 날 확률이 크다. 동기적 코드를 비동기로 포함하여 실행하자!

## Article 3. Cooperative Asynchronous JavaScript : Timeouts and Intervals ( 협동하는 비동기 자바스크립트 )

- **Introduction**
- **setTimeout( )**
- **setInterval( )**
- **setTimeout( ) 과 setInterval( )에서 주의해야 할 점**
- **requestAnimationFrame( )**

- **Introduction**

## Introduction

오랜 시간 동안 웹플랫폼은 자바스크립트 프로그래머가 일정한 시간이 흐른 뒤에 비동기적 코드를 실행할 수 있게하는 다양한 함수들을 제공해 왔다. 그리고 프로그래머가 중지시킬 때까지 코드 블록을 반복적으로 실행하기 위한 다음과 같은 함수들이 있다.

### `setTimeout()`

특정 시간이 경과한 뒤에 특정 코드블록을 한번 실행한다.

### `setInterval()`

각각의 호출 간에 일정한 시간 간격으로 특정 코드블록을 반복적으로 실행한다.

### `requestAnimationFrame()`

setInterval()의 최신 버전; 그 코드가 실행되는 환경에 관계없이 적절한 프레임 속도로 애니메이션을 실행시키면서, 브라우저가 다음 화면을 보여주기 전에 특정 코드블록을 실행한다.

이 함수들에 의해 설정된 비동기 코드는 메인 스레드에서 작동한다. 그러나 프로세서가 이러한 작업을 얼마나 많이 수행하는지에 따라, 코드가 반복되는 중간에 다른 코드를 어느 정도 효율적으로 실행할 수 있다. 어쨌든 이러한 함수들은 웹사이트나 응용 프로그램에서 일정한 애니메이션 및 기타 배경 처리를 실행하는 데 사용된다. 다음 섹션에서는 그것들을 어떻게 사용할 수 있는지 보여줄 것이다.

- **setTimeout( )**

## setTimeout()

앞에서 언급했듯이 `setTimeout()`은 지정된 시간이 경과 한 후 특정 코드 블록을 한 번 실행한다. 그리고 다음과 같은 파라미터가 필요하다.:

- 실행할 함수 또는 다른 곳에 정의된 함수 참조.
- 코드를 실행하기 전에 대기 할 밀리세컨드 단위의 시간 간격 (1000밀리세컨드는 1 초)을 나타내는 숫자. 값을 0으로 지정하면(혹은 이 값을 모두 생략하면) 함수가 즉시 실행된다. 왜 이 파라미터를 수행해야 하는지도 좀 더 살펴 볼 것이다.
- 함수가 실행될 때 함수에 전달해야 할 파라미터를 나타내는 0이상의 값.

**Note:** 타임아웃 콜백은 단독으로 실행되지 않기 때문에 지정된 시간이 지난 그 시점에 정확히 콜백 될 것이라는 보장은 없다. 그보다는 최소한 그 정도의 시간이 지난 후에 호출된다. 메인 스레드가 실행해야 할 핸들러를 찾기 위해 이런 핸들러들을 살펴보는 시점에 도달할 때까지 타임아웃 핸들러를 실행할 수 없다.

아래 예제에서 브라우저는 2분이 지나면 익명의 함수를 실행하고 경보 메시지를 띄울 것이다. ([see it running live](#), and [see the source code](#)):

```
let myGreeting = setTimeout(function() {
  alert('Hello, Mr. Universe!');
}, 2000)
```

지정한 함수가 꼭 익명일 필요는 없다. 함수에 이름을 부여 할 수 있고, 다른 곳에서 함수를 정의하고 `setTimeout()`에 참조 (reference)를 전달할 수도 있다. 아래 코드는 위의 코드와 같은 실행 결과를 얻을 수 있다.

```
// With a named function
let myGreeting = setTimeout(function sayHi() {
  alert('Hello, Mr. Universe!');
}, 2000)

// With a function defined separately
function sayHi() {
  alert('Hello Mr. Universe!');
}

let myGreeting = setTimeout(sayHi, 2000);
```

예를 들자면 timeout 함수와 이벤트에 의해 중복 호출되는 함수를 사용하려면 이 방법이 유용할 수 있다. 이 방법은 코드 라인을 깔끔하게 정리하는 데 도움을 준다. 특히 timeout 콜백의 코드라인이 여러 줄이라면 더욱 그렇다.

setTimeout ()은 나중에 타임아웃을 할 경우에 타임아웃을 참조하는데 사용하는 식별자 값을 리턴한다. 그 방법을 알아보려면 아래 [Clearing timeouts](#)을 참조하세요.

## setTimeout () 함수에 매개변수(parameter) 전달

setTimeout () 내에서 실행되는 함수에 전달하려는 모든 매개변수는 setTimeout () 매개변수 목록 끝에 추가하여 전달해야 한다. 아래 예제처럼, 이전 함수를 리팩터링하여 전달된 매개변수의 사람 이름이 추가된 문장을 표시할 수 있다.

```
function sayHi(who) {  
  alert('Hello ' + who + '!');  
}
```



Say hello의 대상이 되는 사람 이름은 setTimeout ()의 세 번째 매개변수로 함수에 전달된다.

```
let myGreeting = setTimeout(sayHi, 2000, 'Mr. Universe');
```



## timeout 취소

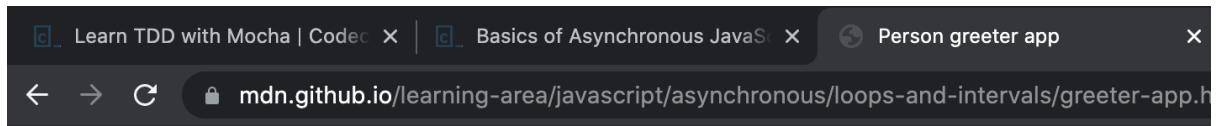
마지막으로 타임아웃이 생성되면 setTimeout()이 실행되면 특정시간이 경과하기 전에 [clearTimeout\(\)](#)을 호출하여 타임아웃을 취소할 수 있다. [clearTimeout\(\)](#)은 setTimeout () 콜의 식별자를 매개변수로 setTimeout ()에 전달한다. 위 예제의 타임아웃을 취소하려면 아래와 같이 하면 된다.

```
clearTimeout(myGreeting);
```



**Note:** 인사를 할 사람의 이름을 설정하고 별도의 버튼을 사용하여 인사말을 취소 할 수 있는 약간 더 복잡한 품양식 예제인 [greeter-app.html](#)을 참조하세요.

- clearTimeout() 예시

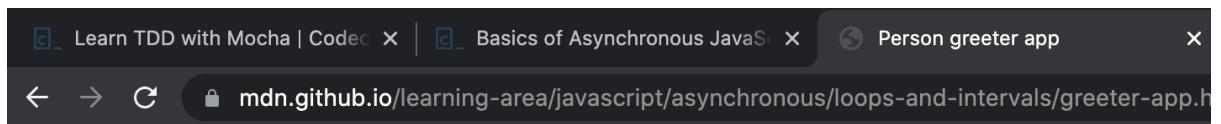


## Give someone a greeting!

Enter a name:

Greet!Cancel greeting

clearTimeout( )의 예제이다. 이름을 입력하고 Greet! 버튼을 누르면 2초 뒤 인삿말이 나오는 기능이다.  
Cancel 버튼은 아직 활성화되지 않았다.



## Give someone a greeting!

Enter a name:

Greet!Cancel greeting

Greet 버튼을 누르자 Cancel이 활성화 됐다. 인삿말이 나오기 전에 Cancel 버튼을 누르면 2초뒤에 나오는  
인삿말이 취소 된다.

- **setInterval()**

## setInterval()

setTimeout ()은 일정 시간이 지난 후 코드를 한 번 실행해야 할 때 완벽하게 작동합니다. 그러나 애니메이션의 경우와 같이 코드를 반복해서 실행해야 할 경우 어떨까요?

이럴 경우에 setInterval ()이 필요합니다. setInterval ()은 setTimeout ()과 매우 유사한 방식으로 작동합니다. 다만 setTimeout ()처럼 첫 번째 매개 변수(함수)가 타입이 웃 후에 한번 실행되는게 아니라 두 번째 매개 변수에 주어진 시간까지 반복적으로 실행되는 것이 차이점입니다. setInterval() 호출의 후속 파라미터로 실행 중인 함수에 필요한 파라미터를 전달할 수도 있다.

예를 들어 봅시다. 다음 함수는 새 Date() 객체를 생성한 후에 ToLocaleTimeString()을 사용하여 시간데이터를 문자열로 추출한 다음 UI에 표시합니다. 그리고 setInterval()을 사용하여 초당 한 번 함수(displayTime)를 실행하면 초당 한 번 업데이트되는 디지털 시계와 같은 효과를 만들어냅니다.([see this live](#), and also [see the source](#)):

```
function displayTime() {
  let date = new Date();
  let time = date.toLocaleTimeString();
  document.getElementById('demo').textContent = time;
}

const createClock = setInterval(displayTime, 1000);
```

setTimeout()과 같이 setInterval()도 식별자 값을 리턴하여 나중에 clearInterval()을 취소해야 할 때 사용한다.

### interval 취소

setInterval ()은 아무 조치를 취하지 않으면 끊임없이 계속 실행됩니다. 이 상태를 중지하는 방법이 필요합니다. 그렇지 않으면 브라우저가 추가 작업을 완료 할 수 없거나, 현재 처리 중인 애니메이션이 완료되었을 때 오류가 발생할 수 있습니다. setTimeout()과 같은 방식으로 setInterval () 호출에 의해 반환된 식별자를 clearInterval () 함수에 전달하여 이 작업을 취소할 수 있습니다.

```
const myInterval = setInterval(myFunction, 2000);

clearInterval(myInterval);
```

### • setTimeout( ) 과 setInterval( )에서 주의해야 할 점

## setTimeout()과 setInterval()에서 주의해야 할 것들

setTimeout()과 setInterval()에는 몇 가지 주의해야 할 것들이 있습니다. 어떤 것인지 한번 살펴보겠습니다.

### 순환 timeouts

setTimeout()을 사용하는 또 다른 방법입니다. 바로 setInterval()을 사용하는 대신 setTimeout()을 이용해 같은코드를 반복적으로 실행시키는 방법입니다.

The below example uses a recursive setTimeout() to run the passed function every 100 milliseconds:  
아래 예제에서는 setTimeout()이 주어진 함수를 100밀리세컨드마다 실행합니다.

```
let i = 1;

setTimeout(function run() {
  console.log(i);
  i++;
  setTimeout(run, 100);
}, 100);
```

위 예제를 아래 예제와 비교해 보세요. 아래 예제는 setInterval()을 사용하여 같은 결과를 얻을 수 있습니다.

```
let i = 1;

setInterval(function run() {
  console.log(i);
  i++;
}, 100);
```

그렇다면 순환 setTimeout()과 setInterval()은 어떻게 다를까요?  
두 방법의 차이는 미묘합니다.

- 순환 setTimeout ()은 실행과 실행 사이에 동일한 지연을 보장합니다. 위의 경우 100ms입니다. 코드가 실행 된 후 다시 실행되기 전에 100 ms 동안 대기하므로 간격은 코드 실행 시간에 관계없이 동일합니다.
- setInterval ()을 사용하는 예제는 약간 다르게 작동합니다. 설정된 간격에는 실행하려는 코드를 실행하는 데 걸리는 시간이 포함됩니다. 코드를 실행하는 데 40ms가 걸리다면 간격이 60ms에 불과합니다.
- setTimeout ()을 재귀적으로 사용할 때 각 반복은 다음 반복을 실행하기 전에 또 하나의 대기시간을 설정합니다. 즉, setTimeout ()의 두 번째 매개 변수의 값은 코드를 다시 실행하기 전에 대기 할 또 하나의 시간을 지정합니다.

코드가 지정한 시간 간격보다 실행 시간이 오래 걸리면 순환 setTimeout ()을 사용하는 것이 좋습니다. 이렇게하면 코드 실행 시간에 관계없이 실행 간격이 일정하게 유지되어 오류가 발생하지 않습니다.

### 즉시 timeouts

setTimeout()의 값으로 0을 사용하면 메인 코드 스레드가 실행된 후에 가능한 한 빨리 지정된 콜백 함수의 실행을 예약할 수 있다.

예를 들어 아래 코드 ([see it live](#)) 는 "Hello"가 포함된 alert를 출력 한 다음 첫 번째 경고에서 OK를 클릭하자마자 "World"가 포함된 alert를 출력합니다.

```
setTimeout(function() {
  alert('World');
}, 0);

alert('Hello');
```

이것은 모든 메인 스레드의 실행이 완료되자마자 실행되도록 코드 블록을 설정하려는 경우 유용할 할 수 있습니다. 비동기 이벤트 루프에 배치하면 곧바로 실행될 겁니다.

## clearTimeout() 와 clearInterval()의 취소기능

clearTimeout ()과 clearInterval ()은 모두 동일한 entry를 사용하여 대상 메소드 setTimeout () 또는 setInterval ()을 취소합니다. 흥미롭게도 이는 setTimeout () 또는 setInterval ()을 지우는데 clearTimeout ()과 clearInterval () 메소드 어느 것을 사용해도 무방합니다.

그러나 일관성을 유지하려면 clearTimeout ()을 사용하여 setTimeout () 항목을 지우고 clearInterval ()을 사용하여 setInterval () 항목을 지우십시오. 혼란을 피하는 데 도움이됩니다.

## • requestAnimationFrame() - setInterval() 의 최신버전

### requestAnimationFrame()

requestAnimationFrame ()은 브라우저에서 애니메이션을 효율적으로 실행하기 위해 만들어진 특수한 반복 함수입니다. 근본적으로 setInterval ()의 최신 버전입니다. 브라우저가 다음에 디스플레이를 다시 표시하기 전에 지정된 코드 블록을 실행하여 애니메이션이 실행되는 환경에 관계없이 적절한 프레임 속도로 실행될 수 있도록 합니다.

setInterval ()을 사용함에 있어 알려진 문제점을 개선하기 위해 만들어졌습니다. 예를 들어 장치에 최적화 된 프레임 속도로 실행되지 않는 문제, 때로는 프레임을 빠뜨리는 문제, 탭이 활성 탭이 아니거나 애니메이션이 페이지를 벗어난 경우에도 계속 실행되는 문제 등등이다. [CreativeJS에서 이에 대해 자세히 알아보십시오.](#)

**Note:** requestAnimationFrame() 사용에 관한 예제들은 이 코스의 여러곳에서 찾아볼 수 있습니다. [Drawing graphics](#) 와 [Object building practice](#)의 예제를 찾아 보세요.

이 메소드는 화면을 다시 표시하기 전에 호출 할 콜백을 인수로 사용합니다. 이것이 일반적인 패턴입니다. 아래는 사용예를 보여줍니다.

```
function draw() {
    // Drawing code goes here
    requestAnimationFrame(draw);
}

draw();
```

그 발상은 애니메이션을 업데이트하는 함수 (예 : 스프라이트 이동, 스코어 업데이트, 데이터 새로 고침 등)를 정의한 후 그 것을 호출하여 프로세스를 시작하는 것입니다. 함수 블록의 끝에서 매개 변수로 전달 된 함수 참조를 사용하여 requestAnimationFrame ()을 호출하면 브라우저가 다음 화면을 재표시할 때 함수를 다시 호출하도록 지시합니다. 그런 다음 requestAnimationFrame ()을 반복적으로 호출하므로 계속 실행되는 것입니다.

**Note:** 어떤 간단한 DOM 애니메이션을 수행하려는 경우, [CSS Animations](#) 은 JavaScript가 아닌 브라우저의 내부 코드로 직접 계산되므로 속도가 더 빠릅니다. 그러나 더 복잡한 작업을 수행하고 DOM 내에서 직접 액세스 할 수 없는 객체 (예 : 2D Canvas API or WebGL objects)를 포함하는 경우 대부분의 경우 requestAnimationFrame ()이 더 나은 옵션입니다.

## 여러분의 애니메이션의 작동속도는 얼마나 빠른가요?

부드러운 애니메이션을 구현은 직접적으로 프레임 속도에 달려 있으며, 프레임속도는 초당 프레임 (fps)으로 측정됩니다. 이 숫자가 높을수록 애니메이션이 더 매끄럽게 보입니다.

일반적으로 화면 재생률은 60Hz이므로 웹 브라우저를 사용할 때 여러분이 설정할 수 있는 가장 빠른 프레임 속도는 초당 60 프레임 (FPS)입니다. 이 속도보다 빠르게 설정하면 과도한 연산이 실행되어 화면이 더듬거리고 띄엄띄엄 표시될 수 있습니다. 이런 현상을 프레임 손실 또는 쟁크라고 합니다.

재생률 60Hz의 모니터에 60FPS를 달성하려는 경우 각 프레임을 렌더링하기 위해 애니메이션 코드를 실행하려면 약 16.7ms(1000/60)가 필요합니다. 그러므로 각 애니메이션 루프를 통과 할 때마다 실행하려고 하는 코드의 양을 염두에 두어야합니다.

requestAnimationFrame()은 불가능한 경우에도 가능한한 60FPS 값에 가까워 지려고 노력합니다. 실제로 복잡한 애니메이션을 느린 컴퓨터에서 실행하는 경우 프레임 속도가 떨어집니다. requestAnimationFrame ()은 항상 사용 가능한 것을 최대한 활용합니다.

## requestAnimationFrame()○] setInterval(), setTimeout()과 다른점은?

requestAnimationFrame () 메소드가 이전에 살펴본 다른 메소드와 어떻게 다른지에 대해 조금 더 이야기하겠습니다. 위의 코드를 다시 살펴보면;

```
function draw() {  
    // Drawing code goes here  
    requestAnimationFrame(draw);  
}  
  
draw();
```

setInterval()을 사용하여 위와 같은 작업을 하는 방법을 살펴봅시다.

```
function draw() {  
    // Drawing code goes here  
}  
  
setInterval(draw, 17);
```

앞에서 언급했듯이 requestAnimationFrame ()은 시간 간격을 지정하지 않습니다. requestAnimationFrame ()은 현재 상황에서 최대한 빠르고 원활하게 실행됩니다. 어떤 이유로 애니메이션이 화면에 표시되지 않으면 브라우저는 그 애니메이션을 실행하는 데 시간을 낭비하지 않습니다.

반면에 setInterval ()은 특정 시간간격을 필요로 합니다. 1000 ms/60Hz 계산을 통해 최종값 16.6에 도달 한 후 반올림 (17)했습니다. 이때 반올림하는 것이 좋습니다. 그 이유는 반내림(16)을하면 60fps보다 빠르게 애니메이션을 실행하려고 하게 되지만 애니메이션의 부드러움에 아무런 영향을 미치지 않기 때문입니다. 앞에서 언급했듯이 60Hz가 표준 재생률입니다.

## timestamp를 포함하기

requestAnimationFrame() 함수에 전달 된 실제 콜백에는 requestAnimationFrame()이 실행되기 시작한 이후의 시간을 나타내는 timestamp를 매개변수로 제공할 수 있습니다. 정치 속도에 관계없이 특정 시간과 일정한 속도로 작업을 수행 할 수 있으므로 유용합니다. 사용하는 일반적인 패턴은 다음과 같습니다.

```
let startTime = null;

function draw(timestamp) {
  if(!startTime) {
    startTime = timestamp;
  }

  currentTime = timestamp - startTime;

  // Do something based on current time

  requestAnimationFrame(draw);
}

draw();
```



## 브라우저 지원

requestAnimationFrame()은 setInterval() / setTimeout()보다 좀 더 최신 브라우저에서 지원됩니다. 가장 흥미롭게도 Internet Explorer 10 이상에서 사용할 수 있습니다. 따라서 별도의 코드로 이전 버전의 IE를 지원해야 할 필요가 없다면, requestAnimationFrame()을 사용하지 않을 이유가 없습니다.

## requestAnimationFrame() call의 취소

cancelAnimationFrame() 메소드를 호출하여 requestAnimationFrame()을 취소할 수 있다. (접두어가 "clear"가 아니고 "cancel"임에 유의) requestAnimationFrame()에 의해서 리턴된 rAF 변수값을 전달받아 취소한다.

```
cancelAnimationFrame(rAF);
```



**setInterval( ) 은 쓰지않고 requestAnimationFrame( ) 을 사용해라 !!**

유튜브 requestAnimationFrame( ) 강의 : <https://www.youtube.com/watch?v=9XnqDSabFjM>

## 오늘의 단어

- driving force : 원동력
- just enough : 딱 알맞게

- Clean up : 정리하다.
- backwards : 거꾸로, 뒤로, 퇴보하는 방향으로
- redundant : 장황한