11월 22일 (월)

1. Codecademy - Async JavaScript and HTTP Requests - Learn JavaScript Syntax: Async-Await

Lesson 1. Async Await

- **Introduction**

생성된 프로미스를 좀 더 가독성있고 동기적 연산을 가능하게 해주는 것이 async await 이다. 즉, 이번 레슨에서도 프로미스는 계속 사용해야한다.

웹 개발에서 우리는 종종 비동기적인 행동을 다뤄야한다. 우리는 네트워크, 데이터베이스, 숫자와 같은 연산에 요청을 한다. 자바스크립트는 non-blocking 이며, 그 요청을 기다리는 동안 코드의 실행을 멈추는 대신에 자바스크립트는 비동기 연산이 완료될 때까지 기다리는 동안 다른 과제를 효과적으로 실행하도록하는 event-loop 를 사용한다.

원래 자바스크립트는 비동기적 연산에 콜백함수를 사용했지만, ES6 에서 promise 가 나오게 되어 코드를 훨씬 더 가독성있게 짤 수 있었다. 우리가 이번에 배울 async...await 는 ES8 이 제공하는 더욱더 발전한 문법이다. 이는 전통적인 비동기적 코드가 더욱 더 가독성 있도록 해주는 기능을 갖고 있다.

async...await 문법은 언어에 새로운 기능을 도입하지 않고, 프로미스와 제네레이터를 기반으로 구축된 syntactic sugar 이다. async 와 await 는 이미 언어에 구축 되어 있다.

- **The async Keyword**

async 키워드는 비동기적 연산을 다루는 함수를 작성할 때 쓰인다. 우리는 async 가 앞에 써있는 함수 안에 비동기적 로직을 써넣는다.

```
async function myFunc() {
  // Function body here
```

```
};

myFunc();
```

```
const myFunc = async () => {
  // Function body here
};

myFunc();
```

async 함수는 항상 프로미스를 리턴한다. 이는 우리가 ES6의 프로미스의 .then( ) 과 .catch( )의 기능을 async 함수를 통해서 사용할 수 있다는 것을 의미한다.

async 함수는 다음 세가지 방식 중 하나로 결과값을 리턴할 것이다.

- 만약 함수로부터 아무것도 리턴되지 않는다면, 하나의 undefined 값으로 resolved 된 프로미스를 리턴할 것이다.
- 만약 함수로부터 프로미스가 아닌 값을 리턴받는다면, 그것은 값으로 resolved 된 프로미스를 리턴할 것이다.
- 만약 함수로부터 프로미스가 리턴된다면, 그것은 그것은 간단하게 그 프로미스를 리턴할 것이다.

```
async function fivePromise() {
  return 5;
}

fivePromise()
.then(resolvedValue => {
    console.log(resolvedValue);
  })  // Prints 5
```

실습 예시

```
// 프로미스로 작성된 코드
function withConstructor(num){
  return new Promise((resolve, reject) => {
    if (num === 0){
      resolve('zero');
    } else {
      resolve('not zero');
    }
  });
}

withConstructor(0)
  .then((resolveValue) => {
  console.log(` withConstructor(0) returned a promise which resolved to: ${resolveValue}.`);
});


// async 로 작성된 코드

async function withAsync(num) {
  if (num === 0) {
    return 'zero';
  } else {
    return 'not zero';
  }
};



withAsync(100)
  .then((resolveValue) => {
```

```
  console.log(` withAsync(100) returned a promise which resolved to: ${resolveValue}.`);
})
```

- **The await Operator**

async 함수는 혼자 쓰이지 않고 거의 await 랑 같이 쓰인다. await 는 async 함수블록 내부에 쓴다.

await 은 오퍼레이터로 하나의 프로미스의 resolved 된 값을 리턴한다.

프로미스는 가늠할 수 없는 시간을 통해 resolve 하기 때문에, await 키워드는 주어진 promise 가 resolved 될 때까지 async 함수의 실행을 정지하거나 멈춘다.

수업 대부분의 상황에서 우리는 함수로부터 리턴되는 프로미스를 다룰 것이다. 이러한 함수들은 일반적으로 라이브러리에서 가져온다.

```
async function asyncFuncExample(){
  let resolvedValue = await myPromise();
  console.log(resolvedValue);
}

asyncFuncExample(); // Prints: I am resolved now!
```

실습 예제

```
/*
this is the brainstormDinner function.
It's a little silly.
It returns a promise that uses a series of
setTimeout() functions to simulate a time-consuming asynchronous action.
It's a good example of "callback hell" or "the pyramid of doom,"
two ways people describe how confusing a bunch of nested callback functions
can become.
*/

const brainstormDinner = () => {
  return new Promise((resolve, reject) => {
  console.log(`I have to decide what's for dinner...`)
  setTimeout(() => {
    console.log('Should I make salad...?');
    setTimeout(() => {
      console.log('Should I make ramen...?');
      setTimeout(() => {
        console.log('Should I make eggs...?');
        setTimeout(() => {
          console.log('Should I make chicken...?');
          resolve('beans');
        }, 1000);
      }, 1000);
    }, 1000);
  }, 1000);
});
};

module.exports = brainstormDinner;
```

```
const brainstormDinner = require('./library.js');
```

```
// 네이티브 프로미스 버전:
function nativePromiseDinner() {
  brainstormDinner().then((meal) => {
    console.log(`I'm going to make ${meal} for dinner.`);
  });
}


// async/await 버전:
async function announceDinner() {

  let meal = await brainstormDinner();
  console.log(`I'm going to make ${meal} for dinner.`);

}


announceDinner(); // 꼭 실행을 해줘야 한다.
```

- **Writing async Functions**

우리는 지금까지 프로미스가 더이상 pending 이 아닐 때까지 async 함수의 실행을 중단하는 await 키워드를 배웠다. 이것은 명백하게 보일지 몰라도, 다루기 힘든 실수가 있을 수 있다. 왜냐하면 우리의 함수는 여전히 실행되고 있기 때문이다. 즉, 종종 우리가 바라던 결과를 출력하지 않을 것이다.

```
let myPromise = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('Yay, I resolved!')
    }, 1000);
  });
}
```

```
async function noAwait() {
 let value = myPromise();
 console.log(value);
}

async function yesAwait() {
 let value = await myPromise();
 console.log(value);
}

noAwait(); // Prints: Promise { <pending> }
yesAwait(); // Prints: Yay, I resolved!
```

await 을 안 쓰면 함수 실행은 정지하지 않는다. 이 함수는 myPromise( ) 가 resolve 하기전에 아직 pending 상태로 남아있을때 이미 console.log( ) 를 실행하여 다음과 같은 값을 출력한 것이다. 즉, promise 객체 그대로를 value 에 할당한 것이다.

실습 예제

```
const shopForBeans = require('./library.js');

function getBeans() {
  console.log(`1. Heading to the store to buy beans...`);
  let value = shopForBeans();
  console.log(`3. Great! I'm making ${value} beans for dinner tonight!`);
}

getBeans();
```

```
$ node app.js
1. Heading to the store to buy beans...
3. Great! I'm making [object Promise] beans for dinner tonig
ht!
2. I bought black beans because they were on sale.
$
```

위와 같이 순서가 뒤 바뀌었다.

```
const shopForBeans = require('./library.js');

async function getBeans() {
  console.log(`1. Heading to the store to buy beans...`);
  let value = await shopForBeans();
  console.log(`3. Great! I'm making ${value} beans for dinner tonight!`);
}

getBeans();
```

```
$ node app.js
1. Heading to the store to buy beans...
2. I bought fava beans because they were on sale.
3. Great! I'm making fava beans for dinner tonight!
$
```

결과가 순서대로 잘 나오는 것을 볼 수 있다.

- **Handling Dependent Promises**

async await 의 실제 아름다움은 서로 의존하는 일련의 비동기적 액션을 가질 때 비로소 느낄 수 있다. 예를 들어 데이터베이스의 쿼리에 기반한 네트워크 요청을 한다고 할 때, 우리는 데이터베이스로부터 오는 결과를 우리가 얻을 때까지 네트워크 요청을 기다려야 할 것이다. 네이티브 프로미스 문법에서 우리는 .then( ) 함수 체이닝을 사용했다.

```
function nativePromiseVersion() {
  returnsFirstPromise()
    .then((firstValue) => {
      console.log(firstValue);
      return returnsSecondPromise(firstValue);
    })
   .then((secondValue) => {
      console.log(secondValue);
    });
}
```

```
async function asyncAwaitVersion() {
  let firstValue = await returnsFirstPromise();
  console.log(firstValue);
  let secondValue = await returnsSecondPromise(firstValue);
  console.log(secondValue);
}
```

비록 async await 문법을 사용하는 것은 타이핑해야하는 수를 줄여줄 수 있지만, 사실 코드의 길이 감소가 주된 포인트가 아니다. 두 버전의 코드를 고려해보자면, async await 버전이 동기적코드와 가깝고, 이것은 개발자가 유지보수하고 디버그하기에 아주 쉽다. 또한 복잡한 코드에서 프로미스를 체이닝할 때 저장과 참조를 굉장히 쉽게 하게 해준다.

실습 예제

```
/*
This is the shopForBeans function from the last exercise
*/

const shopForBeans = () => {
  return new Promise((resolve, reject) => {
  const beanTypes = ['kidney', 'fava', 'pinto', 'black', 'garbanzo'];
  setTimeout(()=>{
    let randomIndex = Math.floor(Math.random() * 5);
    let beanType = beanTypes[randomIndex];
    console.log(`I bought ${beanType} beans because they were on sale.`);
   resolve(beanType);
  }, 1000)
})
}

let soakTheBeans = (beanType) => {
   return new Promise((resolve, reject) => {
     console.log('Time to soak the beans.');
    setTimeout(()=>{
      console.log(`... The ${beanType} beans are softened.`);
      resolve(true);
      }, 1000);
  });
}

let cookTheBeans = (isSoftened) => {
  return new Promise((resolve, reject) => {
    console.log('Time to cook the beans.');
    setTimeout(()=>{
      if (isSoftened) {
        console.log('... The beans are cooked!');
        resolve('\n\nDinner is served!');
      }
    }, 1000);
  });
}
```

```
module.exports = {shopForBeans, soakTheBeans, cookTheBeans};
```

```
const {shopForBeans, soakTheBeans, cookTheBeans} = require('./library.js');

// Write your code below:

async function makeBeans() {
  let type = await shopForBeans();
  let isSoft = await soakTheBeans(type);
  let dinner = await cookTheBeans(isSoft);
  console.log(dinner);
}


makeBeans();
```

```
$ node app.js
I bought kidney beans because they were on sale.
Time to soak the beans.
... The kidney beans are softened.
Time to cook the beans.
... The beans are cooked!


Dinner is served!
$
```

실행 결과

- **Handling Errors**

긴 프로미스 체이닝에서 .catch( ) 함수가 사용될 때, 어떤 체인에서 에러가 던져졌는지 모른다. 이것은 디버깅을 어렵게 만들곤 한다.

async await 에서는 에러 핸들링을 위해 try ... catch 를 사용한다. 이 문법을 사용함으로써 에러를 동기적 코드로 다룰 수 있을 뿐만 아니라, 비동기적, 동기적 코드 상관없이 모든 에러를 잡아낼 수 있다. 이는 디버깅을 더 쉽게 만든다.

```
async function usingTryCatch() {
 try {
   let resolveValue = await asyncFunction('thing that will fail');
   let secondValue = await secondAsyncFunction(resolveValue);
 } catch (err) {
   // Catches any errors in the try block
   console.log(err);
 }
}
```

```
usingTryCatch();
```

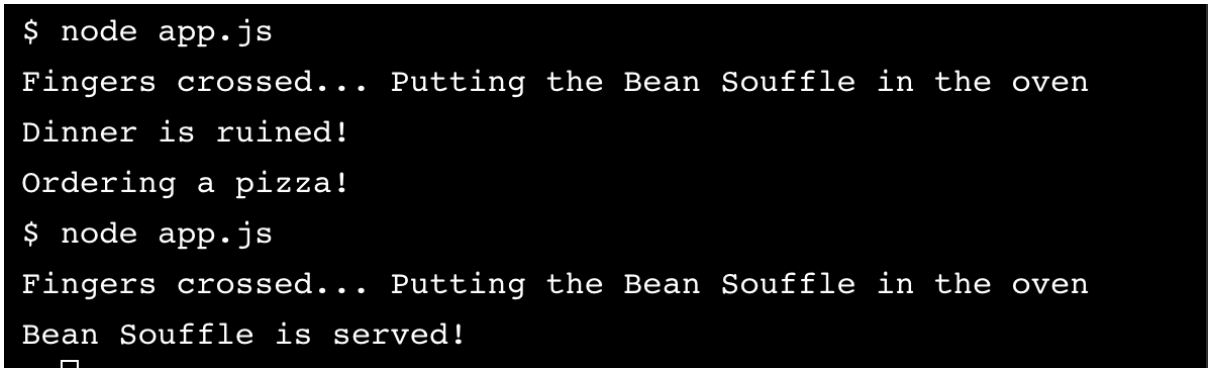
```
async function usingPromiseCatch() {
   let resolveValue = await asyncFunction('thing that will fail');
}

let rejectedPromise = usingPromiseCatch();
rejectedPromise.catch((rejectValue) => {
console.log(rejectValue);
})
```

실습 예제

지금까지 rejected 가 없는 쉬운 예시만 봐왔지만 이번엔 rejected 가 추가된 예시를 볼 것이다.

```
// This function returns true 50% of the time.
let randomSuccess = () => {
 let num = Math.random();
 if (num < .5 ){
   return true;
 } else {
   return false;
 }
};

// This function returns a promise that resolves half of the time and rejects half of the time
let cookBeanSouffle = () => {
 return new Promise((resolve, reject) => {
   console.log('Fingers crossed... Putting the Bean Souffle in the oven');
   setTimeout(()=>{
     let success = randomSuccess();
     if(success){
       resolve('Bean Souffle');
     } else {
       reject('Dinner is ruined!');
     }
   }, 1000);
 });
};

module.exports = cookBeanSouffle;
```

```
const cookBeanSouffle = require('./library.js');

// Write your code below:
async function hostDinnerParty() {
  try {
      let status = await cookBeanSouffle();
      console.log(`${status} is served!`);
  } catch(error) {
      console.log(error);
      console.log('Ordering a pizza!');
  }
}


hostDinnerParty();
```



위와 같은 결과를 출력한다.

- **Handling Independent Promises (독립적인 프로미스 다루기 - 동시성으로 다루기)**

async await 은 종속적인 프로미스들을 동기적으로 다룰 수 있도록 해주는 편리한 기능이라는 것을 배웠다.

만약 async 함수가 서로 결과에 종속적이지 않은(=영향을 끼치지않는 ex. 결과값을 다시 다음 인자에 넣는 액션) 프로미스인 다수의 독립적 프로미스를 포함하고 있다면 어떻게 할까?

```
async function waiting() {
 const firstValue = await firstAsyncThing();
 const secondValue = await secondAsyncThing();
 console.log(firstValue, secondValue);
}

async function concurrent() {
 const firstPromise = firstAsyncThing();
 const secondPromise = secondAsyncThing();
console.log(await firstPromise, await secondPromise);
}
```

wating( ) 함수에서는 첫번째 프로미스가 resolve 할때까지 우리의 함수를 정지한다. 그 다음 두번째 프로미스를 실행한다. 두개 모두 resolve 하면, resolved 된 두 값을 콘솔에 출력한다.

concurrent( ) 함수에서는 두 프로미스는 await 없이 구성되었다. 대신 콘솔로그 안에 프린트하기 위한 반환된 값에 await 을 넣어줬다.

concurrent 함수에서 두 프로미스의 비동기적 연산은 동시적으로 실행될 수 있다. 가급적이면, 우리는 각 비동기연산이 가능한 빨리 시작되길 원한다. async 함수에서 우리는 여전히 비동기적 연산을 동시에 수행하는 능력인 동시성(concurrency)을 이용해야한다.

****Note: 만약 우리가 완전히 병렬적으로 실행하고 싶은 다수의 독립적인 프로미스를 실제로 가지고 있다면, 우리는 각각에 .then( ) 함수를 사용해야한다. 그리고 await 키워드로 우리의 실행을 멈추는 것을 피해야한다. 즉, 무조건 async await 만 쓰는 것이 아니라 네이티브 프로미스 .then( ) .catch( ) 또한 위와 같은 상황에 따라 알맞게 쓰인다.**

실습 예제

```
let cookBeans = () => {
  return new Promise ((resolve, reject) => {
   setTimeout(()=>{
     resolve('beans');
   }, 1000);
 });
}

let steamBroccoli = () => {
 return new Promise ((resolve, reject) => {
   setTimeout(()=>{
     resolve('broccoli');
   }, 1000);
 });
}

let cookRice = () => {
 return new Promise ((resolve, reject) => {
   setTimeout(()=>{
     resolve('rice');
   }, 1000);
 });
}

let bakeChicken = () => {
 return new Promise ((resolve, reject) => {
   setTimeout(()=>{
     resolve('chicken');
   }, 1000);
 });
}

module.exports = {cookBeans, steamBroccoli, cookRice, bakeChicken};
```

- **실습예제** - 동시성이 있는 독립적인 프로미스 - 출력에 await

```
let {cookBeans, steamBroccoli, cookRice, bakeChicken} = require('./library.js');

// Write your code below:

async function serveDinner() {
  let vegetablePromise = steamBroccoli();   // 여기에 await 을 쓰지 않는다.
  let starchPromise = cookRice();
  let proteinPromise = bakeChicken();
  let sidePromise = cookBeans();
  console.log(`Dinner is served. We're having ${await vegetablePromise}, ${await starchPromise}, ${await proteinPromise}, and ${await
} // 여기에 await 을 써서 콘솔 내용을 동시에 출력가능하도록 한다.


serveDinner();
```

```
$ node app.js
Dinner is served. We're having broccoli, rice, chicken, and
beans.
$
```

위 예시의 결과값 - 동시성을 이루어 1초만에 결과값을 반환했다.

- **실습 예제** - await 을 아예 안넣은 프로미스

```
let {cookBeans, steamBroccoli, cookRice, bakeChicken} = require('./library.js');

// Write your code below:

async function serveDinner() {
  let vegetablePromise = steamBroccoli();
  let starchPromise = cookRice();
  let proteinPromise = bakeChicken();
  let sidePromise = cookBeans();
  console.log(`Dinner is served. We're having ${vegetablePromise}, ${starchPromise}, ${proteinPromise}, and ${sidePromise}.`);
}


serveDinner();
```

```
$ node app.js
Dinner is served. We're having [object Promise], [object Pro
mise], [object Promise], and [object Promise].
$
```

만약 콘솔로그안에 await 을 안쓰면 생기는 결과값

- **실습예제** - 동시성이 없는 독립적 프로미스 - 변수 할당에서 await

```
let {cookBeans, steamBroccoli, cookRice, bakeChicken} = require('./library.js');

// Write your code below:

async function serveDinner() {
  let vegetablePromise = await steamBroccoli();
  let starchPromise = await cookRice();
  let proteinPromise = await bakeChicken();
  let sidePromise = await cookBeans();
  console.log(`Dinner is served. We're having ${vegetablePromise}, ${starchPromise}, ${proteinPromise}, and ${sidePromise}.`);
}


serveDinner();
```

```
$ node app.js
Dinner is served. We're having broccoli, rice, chicken, and
beans.
$
```

결과는 똑같이 나오지만 각 함수당 1초 씩 총 4초의 시간이 걸렸다. 즉, 동시성이 없다!! 맨 위 예시에서 waiting( ) 함수와 같다.

- **Await Promise.all( )**

동시다발적으로 실행될 수 있는 다수의 프로미스를 가질 때, 동시성을 이용하기 위한 또 다른 방법은 await Promise.all( ) 함수를 사용하는 것이다.

우리는 네이티브 프로미스 파트에서 배운 것과 같이 Promise.all( ) 함수의 인자에 프로미스 배열을 전달할 수 있다. 그리고 그것은 단 하나의 프로미스를 리턴할 것이다. 이 Promise.all( ) 은 배열내 모든 프로미스가 resolved 되어야만 resolve 할 것이다. 하나라도 rejected 되면 이 Promise.all( ) 도 rejected 가 된다. 이 프로미스의 resolve 값은 인자로 전달된 배열내 각각 프로미스의 resolved 된 값을 포함하는 배열이 될 것이다.

```
async function asyncPromAll() {
  const resultArray = await Promise.all([asyncTask1(), asyncTask2(), asyncTask3(), asyncTask4()]);
  // 각 프로미스의 resolved 된 값을 콘솔로그에 출력하는 반복문
  //(resultArray 변수는 각 프로미스의 resolved 값이 저장된 배열이다.rejected 가 하나라도 있음 reject)
  for (let i = 0; i<resultArray.length; i++){
    console.log(resultArray[i]);
  }
}
```

Promise.all( )은 4개의 비동기적 과제를 동시적(Concurrency)으로 접근할 수 있게 하는 비동기성을 이용할 수 있도록 해주는 함수이다.(전 연습때 출력에 await 을 넣은 것과 같은 결과) 이 await Promise.all( )는 만약 하나라도 rejected 된다면 나머지 작업은 보지도 기다리지도 않고 에러의 이유와 함께 rejected 을 반환하는 ***falling fast*** 의 이점 또한 가지고 있다.

네이티브 프로미스로 작업할때와 마찬가지로, Promise.all( )은 만약 다수의 비동기적 과제가 모두 요청된다면 해당함수를 사용하는 것은 좋은 선택이지만, 실행하기 전에 다른 작업을 기다릴 필요는 없다.

실습 예제

위 라이브러리 함수와 같다.

```
let {cookBeans, steamBroccoli, cookRice, bakeChicken} = require('./library.js');

// Write your code below:

async function serveDinnerAgain() {
  const foodArray = await Promise.all([steamBroccoli(), cookRice(), bakeChicken(), cookBeans()]);
  let vegetable = foodArray[0];
  let starch = foodArray[1];
  let protein = foodArray[2];
  let side = foodArray[3];

  console.log(`Dinner is served. We're having ${vegetable}, ${starch}, ${protein}, and ${side}.`);

}


serveDinnerAgain();



// 내가 했던 방법 ( 프로미스 결과값을 변수로 지정하지 않고 foodArray 값을 인덱스로 바로 넣어주었다. )

let {cookBeans, steamBroccoli, cookRice, bakeChicken} = require('./library.js');
```

```
// Write your code below:

async function serveDinnerAgain() {
  const foodArray = await Promise.all([steamBroccoli(), cookRice(), bakeChicken(), cookBeans()]);

  console.log(`Dinner is served. We're having ${foodArray[0]}, ${foodArray[1]}, ${foodArray[2]}, and ${foodArray[3]}.`);

}


serveDinnerAgain();
```

```
let {cookBeans, steamBroccoli, cookRice, bakeChicken} = require('./library.js');

// Write your code below:

async function serveDinner() {
  let vegetablePromise = steamBroccoli();   // 여기에 await 을 쓰지 않는다.
  let starchPromise = cookRice();
  let proteinPromise = bakeChicken();
  let sidePromise = cookBeans();
  console.log(`Dinner is served. We're having ${await vegetablePromise}, ${await starchPromise}, ${await proteinPromise}, and ${await
} // 여기에 await 을 써서 콘솔 내용을 동시에 출력가능하도록 한다.


serveDinner();
```

```
$ node app.js
Dinner is served. We're having broccoli, rice, chicken, and beans.
$
```

결과값

- **Review**
- async await 는 네이티브 자바스크립트 프로미스와 제네레이터를 기반으로 구축된 syntactic sugar 이다.
- async 함수는 async 키워드를 사용하여 선언한다.
- async 함수안에서 비동기적 연산이 완료되고 더이상 pending 이 없을 때까지 함수의 실행을 멈추는 await 연산자를 사용한다.
- await 은 await 된 프로미스의 resolved 된 값을 리턴한다.
- 동기적인 코드처럼 읽을 수 있는 코드를 생산하기위해 다수의 await 문을 쓸 수 있다.
- async 함수에서 에러를 잡기 위해 try...catch 문을 사용한다.
- 가능한 언제든지 동시적으로 발생하는 비동기적인 액션을 허용하는 async 함수를 작성함으로써 동시성을 이용해야한다.

퀴즈

What does the function `example()` return?

```
async function example() { }
```

A promise

A function

`undefined`

An `async` function returns a promise. `example()` will return a promise which resolves to the value `undefined`.

A string

What will this code print to the console?

```
async function myFunction() {
  return 'hello world';
}

myFunction()
.then((resolvedValue) => {
  console.log(resolvedValue);
})
```

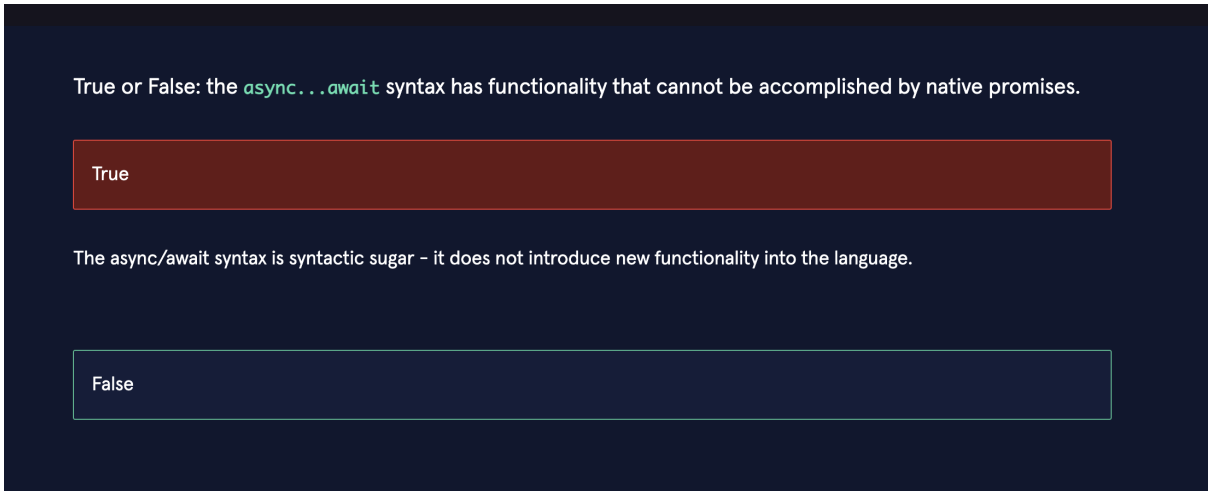
`Promise { 'hello world' }`

`hello world`

👏 Right!

It won't print anything.

`Promise { <pending> }`



오늘의 단어

- imperative : 피할수없는, 반드시 해야하는, 긴요한
- indeterminate : 불확실한, 가늠할 수 없는
- resolution : 결심, 결단 ; 결의 ; 단호함 ; 해상도 ; 분석 ; 해답 , 해명
- If possible : 가급적이면
- take advantage of : ~ 을 이용하다.
- As ~ : ~와 마찬가지로, 마치, ~처럼, ~때문에(으니, 하니까), ~라고해서,