

9일 (금)

1. Codecademy JavaScript - Promises

- **Introduction**

비동기 연산은 컴퓨터가 비동기적 연산을 완료할 때까지 다른 과제로 이동하는 것을 허용하는 것을 말한다.

우리의 일상에서 자주 볼 수 있다. 집안 청소를 예를 들어보자. 우리는 세탁기에 세탁을 넣고 세탁이 모두 완료될 때까지 기다리지 않아도 된다. 세탁이 다 되는 동안 설거지를 할 수 있고 화분에 물을 줄 수도 있다. 이것이 바로 비동기적 프로그래밍이다. 어떤 일이 다 마치기전에 기다리지 않고 다른 동작을 수행하는 것이다.

웹 개발에서 비동기 예는 네트워크 요청을 보내거나 데이터베이스 쿼리잉을 완료 하는 동안 다른 작업을 할 수 있도록 하는 것이다.

ES6의 Promise 객체를 이용해서 비동기적 프로그래밍을 해보자.

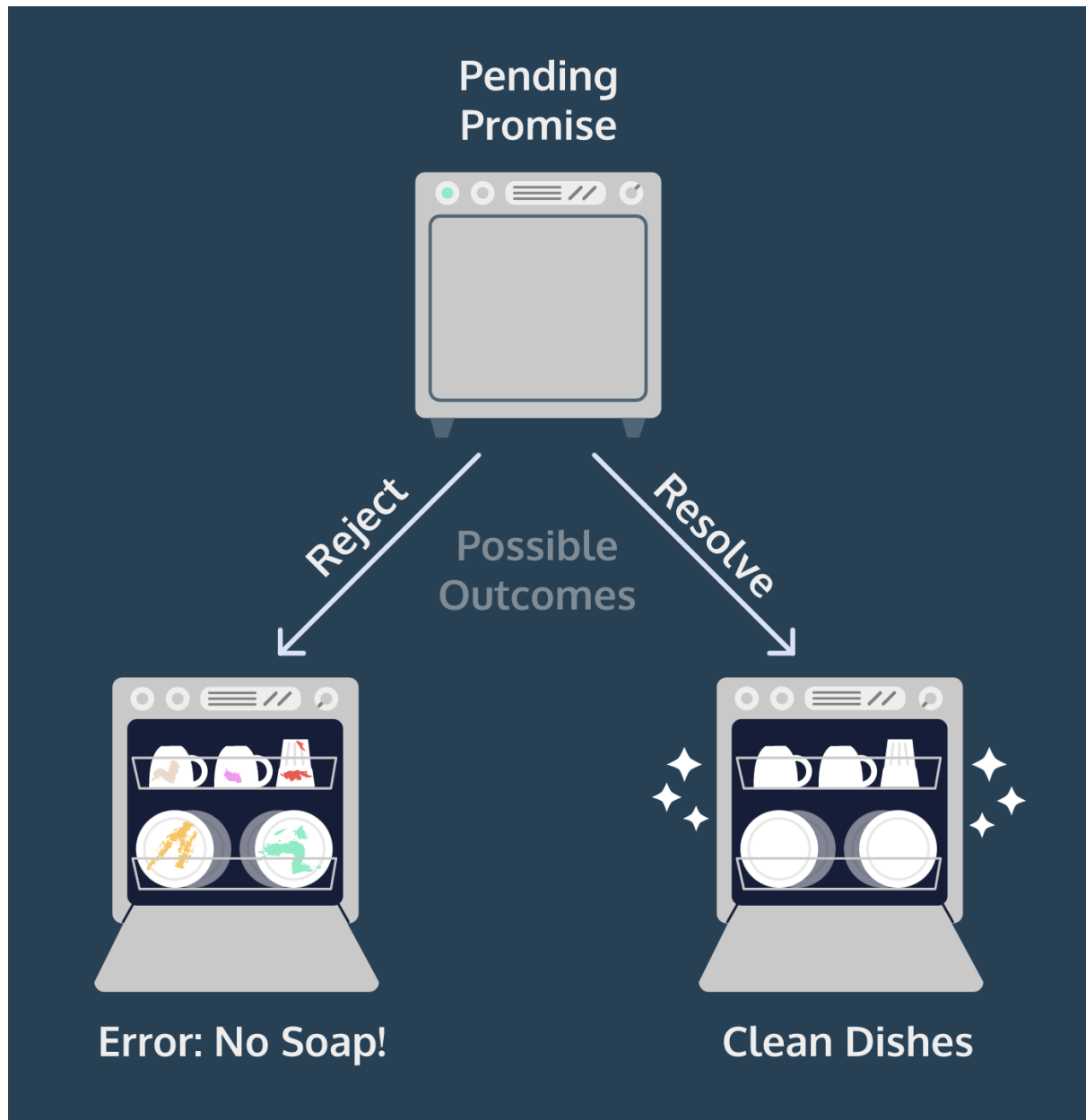
- **What is a Promise?**

promise 는 비동기 연산의 최종적인 결과를 나타내는 객체이다. promise 객체는 세가지 상태중 하나가 될 수 있다.

⇒ Pending(초기 상태) : 동작이 아직 완료되지 않은 상태

⇒ Fulfilled : 동작이 성공적으로 완료된 상태이며 promise 는 resolved value 를 가진다.

⇒ Rejected : 동작은 실패하고 promise 는 실패의 이유를 가진다. 이 이유는 보통 Error 이다.



프로미스는 결국 fulfilled 이든 rejected 든 정착한다. 이러한 결과는 앞으로 우리가 어떤 로직을 적어야할 지 가능하게 한다.

디시워시 예시에서 만약 Resolve 되면 다음 로직은 접시들을 빼서 잘 정리하는 로직일 수 있고, Reject 즉 error 가 난다면 디시웨시를 다시 돌리는 로직일 수 있는 것이다. 즉, 프로그램의 방향성을 제시한다.

- **Constructing a Promise Object (프로미스 객체 짜기)**

```
const executorFunction = (resolve, reject) => { };  
const myFirstPromise = new Promise(executorFunction);
```

프로미스를 위해 new 키워드와 Promise 생성자함수에 넣을 executor 함수가 필요하다. executor 함수는 보통 비동기적 동작을 시작하고 promise가 어떤 상태에 정착해야하는지 말해준다.

executor 함수는 resolve() 와 reject() 의 두가지 함수 인자를 갖는다. 두 함수 인자는 프로그래머에 의해 정의되지 않는다. Promise Constructor이 실행될 때, 자바스크립트는 자신의 resolve() 와 reject() 함수를 executor 함수에 패스 할 것이다.

⇒ resolve() 함수는 프로미스의 상태를 pending 에서 fulfilled 로 바꾸는 함수이다. 그리고 프로미스의 해결된 값은 resolve() 안에 전달된 인자로 지정될 것이다.

⇒ reject() 함수는 에러나 이유를 인자로서 갖는 함수이다. 이 함수는 프로미스의 상태를 pending 에서 rejected로 바꾼다. 그리고 프로미스의 거절 이유는 reject() 함수안에 전달된 인자로 지정될 것이다.

```
const executorFunction = (resolve, reject) => {  
  if (someCondition) {  
    resolve('I resolved!');  
  } else {  
    reject('I rejected!');  
  }  
}  
const myFirstPromise = new Promise(executorFunction);
```

예시

```

1 ▼ const inventory = {
2   sunglasses: 1900,
3   pants: 1088,
4   bags: 1344
5 };
6
7 // Write your code below:
8 ▼ const myExecutor = (resolve, reject) => {
9   ▼ if (inventory.sunglasses > 0) {
10     resolve('Sunglasses order processed.');

```

선글라스 구매 예시

```
+ × bash ↗  
  
$ node app.js  
Promise { 'Sunglasses order processed.'  
  }  
$
```

- **Node setTimeout() 함수**

프로미스를 어떻게 건설하는지 아는 것은 유용하지만, 프로미스를 어떻게 소비하거나 사용할지가 더 중요하다. 우리는 비동기 동작의 결과값을 리턴하는 프로미스 객체를 다룰 것이다. 프로미스는 pending 에서 시작하고 결국 rejected 나 fulfilled 에 정착할 것이다.

이제 약간의 시간이 흐른 뒤 settle하는 프로미스를 리턴하는 함수와 함께 시뮬레이팅을 해보자. 이를 위해 setTimeout() 메소드를 사용할 것이다. setTimeout() 메소드는 콜백 함수가 약간의 딜레이 후 스케줄된 과제를 수행할 수 있도록 하는 Node API 이다. 이는 콜백 함수와 딜레이 시간 두 가지 인자를 가진다.

```
const delayedHello = () => {  
  console.log('Hi! This is an asynchronous greeting!');  
};  
  
setTimeout(delayedHello, 2000);
```

초는 밀리세컨즈이다. 1000은 1초를 나타낸다.

위 예시에서 delayedHello()는 최소한 2초 뒤에(더 길어질 수 있음) 실행될 것이다. 왜 정확히 2초가 아니라 더 늦어질 수 있나?

이 딜레이는 비동기적으로 수행된다. 즉, 우리의 나머지 프로그램이 딜레이동안 수행을 멈추지 않을 수 있다. 비동기적 자바스크립트는 event-loop 라는 것을 사용한다. 2초 뒤에 delayedHello() 는 실행되길 기다리는 코드 라인에 추가된다. 이것이 실행될 수 있기 전에 프로그램의 어느 다른 비동기적 코드는 실행될 것이다. 다음 순서로 , 그 코드 앞에 있는 다른 코드가 된다. 이것이 delayedHello() 가 실제 2초보다 더 걸릴 수 있다는 것을 설명한다.

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

const prom = returnPromiseFunction();
```

위 예시에서는 우리는 프로미스를 리턴하는 returnPromiseFunction()을 실행했다. 이 프로미스를 prom 변수에 할당한다. prom 의 초기상태는 pending 이며, 1초 뒤에 fulfilled로 변경된다.

```

1 console.log("This is the first line of
  code in app.js.");
2
3
4 ▼ const usingST0 = () => {
5   console.log("Hello World");
6 }
7
8
9 setTimeout(usingST0, 2000);
10
11
12 console.log("This is the last line of
   code in app.js.");

```

```

$ node.js
bash: node.js: command not found
$ node app.js
This is the first line of
  code in app.js.
This is the last line of
  code in app.js.
Hello World
$

```

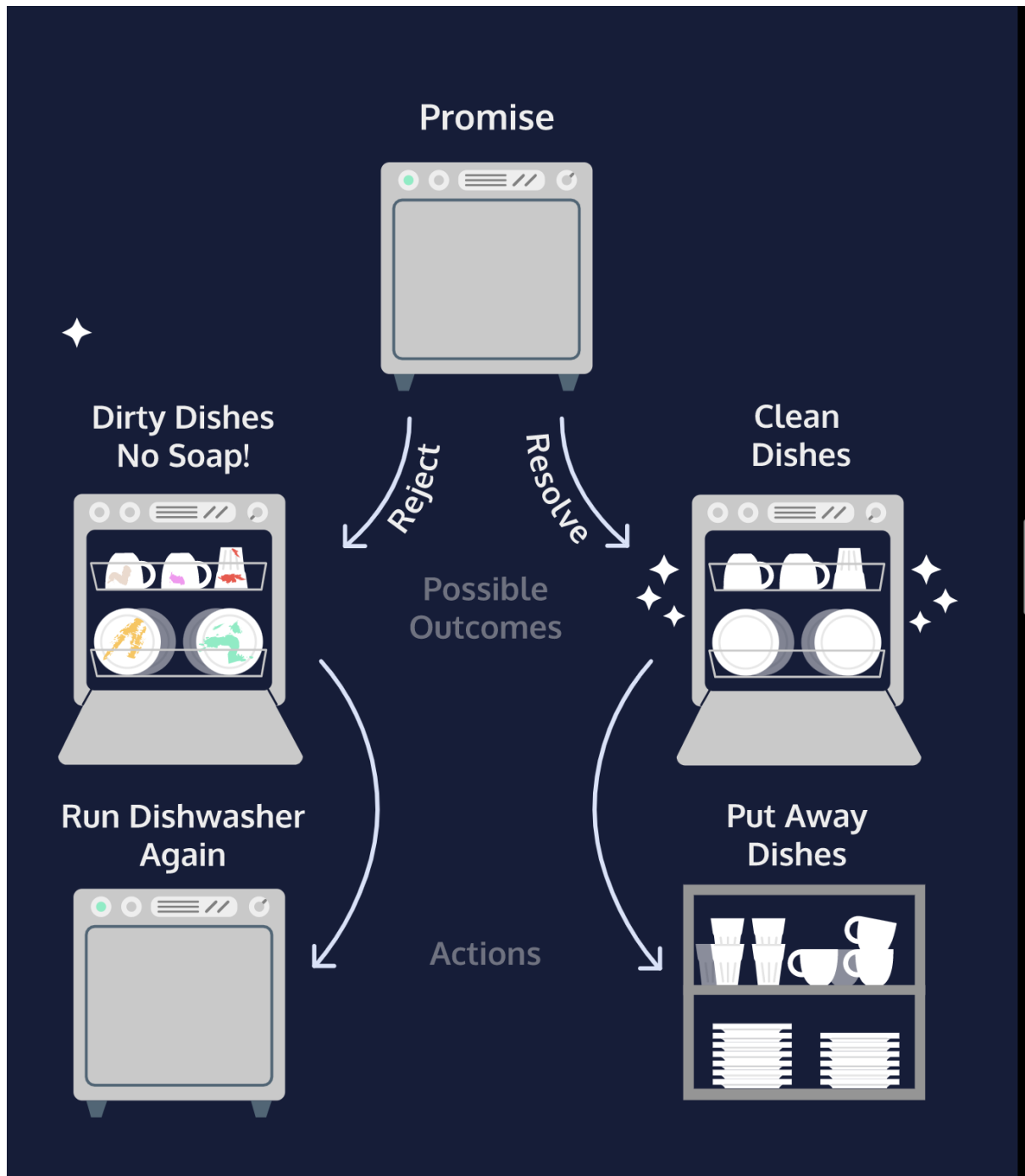
원래 맨위 콘솔로그 뒤 `setTimeout()` 함수뒤에 다른 콘솔로그가 출력되어야 하는데 `setTimeout()`이 비동기적 실행으로 가장 마지막에 출력되었다. 이런 식으로 아직 프로그램이 끝나지 않은 상태에서 다른 명령을 실행하는 것이 비동기적 프로그래밍이다.

• Consuming Promises - `.then()` 키워드

비동기적 프로미스의 초기상태는 `pending` 이며, 우리는 그것이 어디든 정착한다는 것을 보장한다. 그렇다면 그 이후에 어떤 것을 해야하는지 컴퓨터에게 어떻게 말해야할까? 바로 `.then()` 을 이용하면 된다. `promise` 객체는 `.then()` 를 가지고 있다.

`then()` 은 두개의 콜백함수를 인자로 갖는 고차함수이다. 이는 `handler` 라고 불린다. `onFulfilled` 핸들러, `onRejected` 핸들러로 나뉜다. `onFulfilled` 핸들러는 `fulfilled` 된 후 실행해야는 동작이며, `onRejected` 는 그 반대이다.

우리는 `then()` 을 0개 또는 하나 또는 두개로 유연하게 설정할 수 있다. 그러나 이것은 복잡한 디버깅을 요구할 수 있다. `then` 은 에러를 띄우는 대신, 정착된 값의 프로미스를 리턴한다. `then`의 중요한 특징 한 가지는 항상 프로미스를 리턴한다는 것이다.



- **Success and Failure Callback Functions**


```
const prom = new Promise((resolve, reject) => {
  resolve('Yay!');
});

const handleSuccess = (resolvedValue) => {
  console.log(resolvedValue);
};

prom.then(handleSuccess); // Prints: 'Yay!'
```

resolve 만 있는 예시

prom 은 'Yay!'를 resolve 하는 하나의 프로미스이다.

handleSuccess()는 전달된 인자를 프린트하는 함수이다.

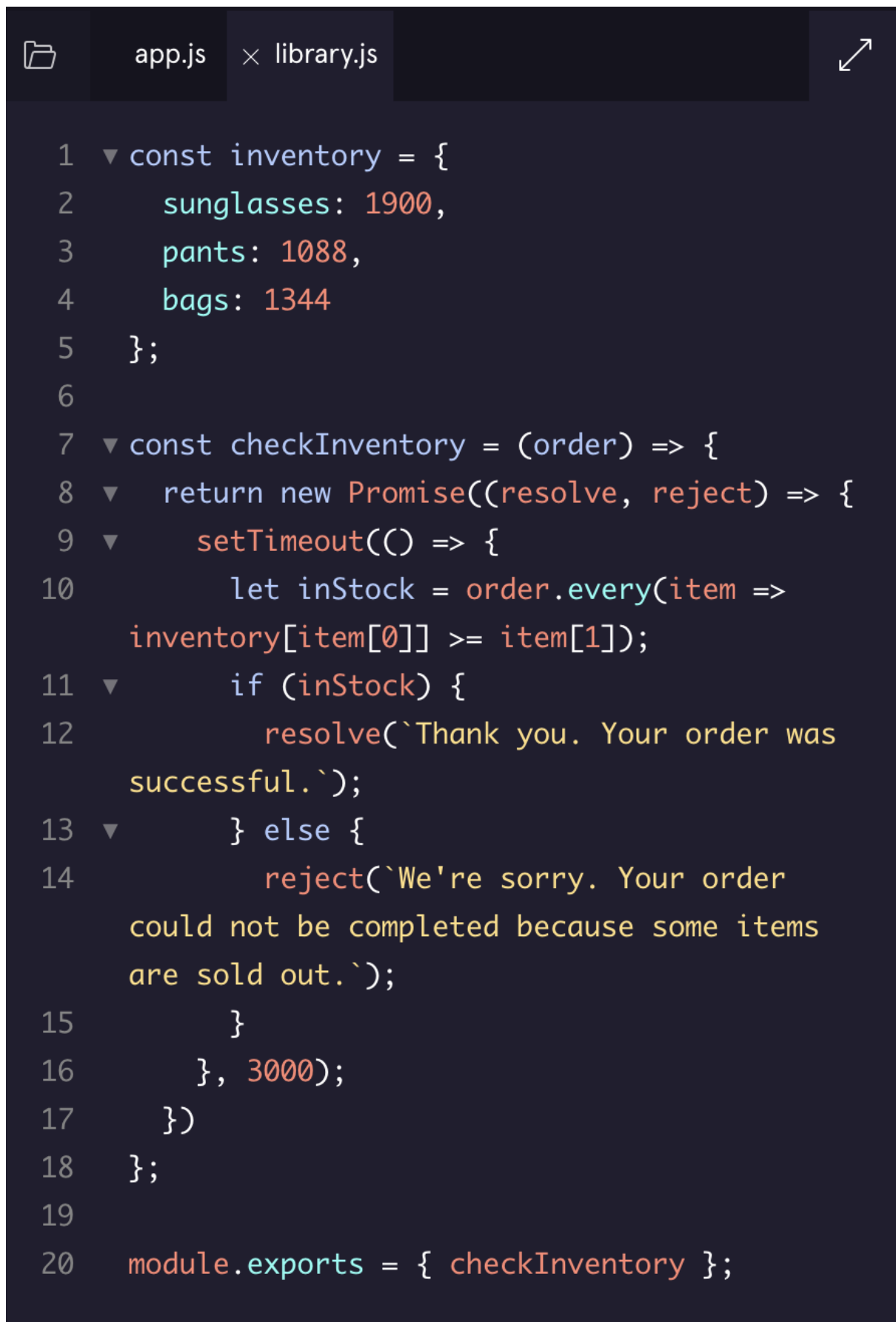
```
let prom = new Promise((resolve, reject) => {
  let num = Math.random();
  if (num < .5 ){
    resolve('Yay!');
  } else {
    reject('Ohhh noooo!');
  }
});

const handleSuccess = (resolvedValue) => {
  console.log(resolvedValue);
};

const handleFailure = (rejectionReason) => {
  console.log(rejectionReason);
};

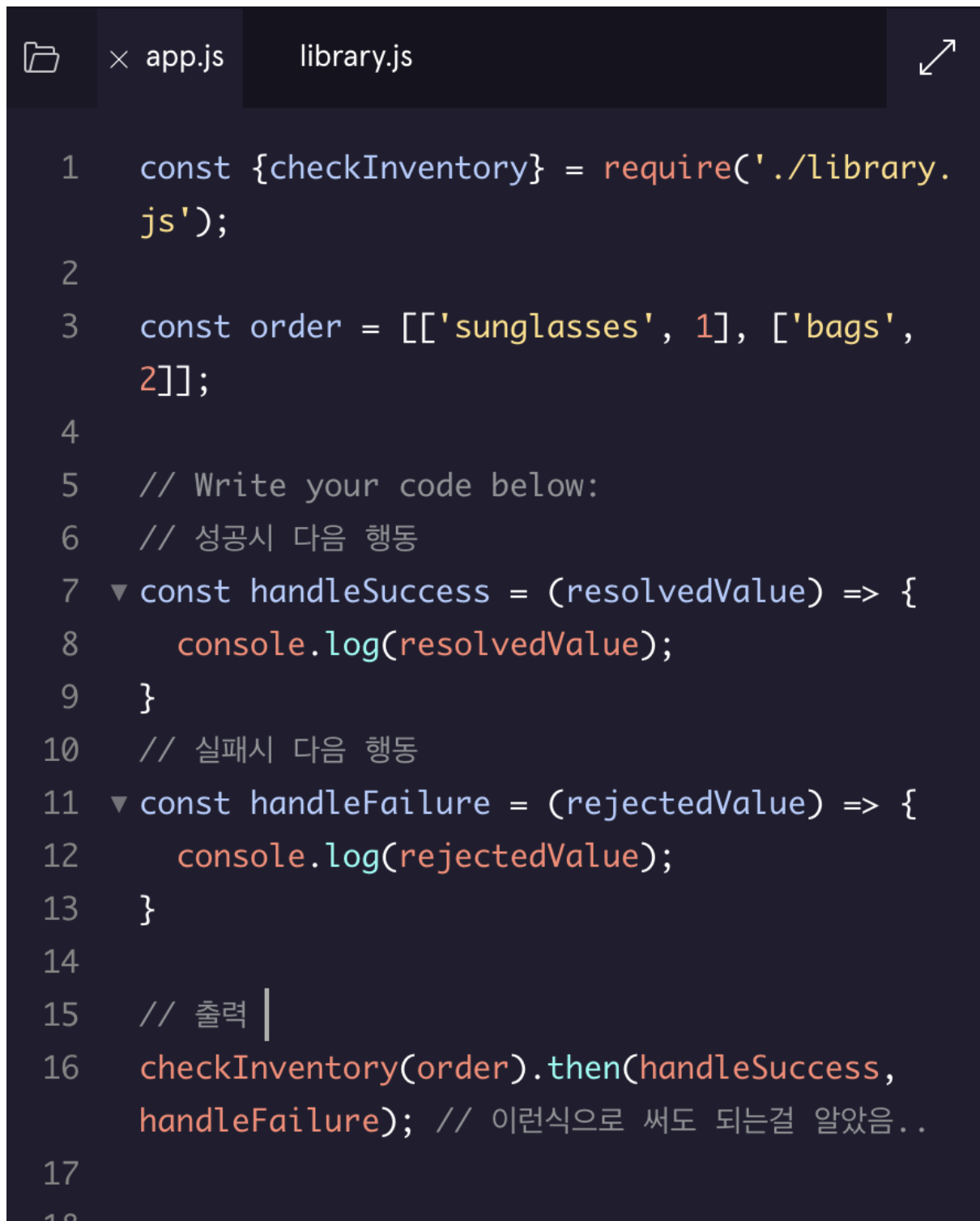
prom.then(handleSuccess, handleFailure);
```

실습



```
1 ▼ const inventory = {
2     sunglasses: 1900,
3     pants: 1088,
4     bags: 1344
5 };
6
7 ▼ const checkInventory = (order) => {
8 ▼     return new Promise((resolve, reject) => {
9 ▼         setTimeout(() => {
10             let inStock = order.every(item =>
              inventory[item[0]] >= item[1]);
11 ▼             if (inStock) {
12                 resolve(`Thank you. Your order was
                  successful.`);
13 ▼             } else {
14                 reject(`We're sorry. Your order
                  could not be completed because some items
                  are sold out.`);
15             }
16             }, 3000);
17         })
18     };
19
20     module.exports = { checkInventory };
```

promise 를 리턴하는 checkInventory() 모듈 메서드를 app.js 에 부른다.



```
1  const {checkInventory} = require('./library.  
js');  
2  
3  const order = [['sunglasses', 1], ['bags',  
2]];  
4  
5  // Write your code below:  
6  // 성공시 다음 행동  
7  ▼ const handleSuccess = (resolvedValue) => {  
8      console.log(resolvedValue);  
9  }  
10 // 실패시 다음 행동  
11 ▼ const handleFailure = (rejectedValue) => {  
12     console.log(rejectedValue);  
13 }  
14  
15 // 출력 |  
16 checkInventory(order).then(handleSuccess,  
    handleFailure); // 이런식으로 써도 되는걸 알았음..  
17  
18
```

require 후 핸들러만 정의하고 then 함수와 바로 실행하였다.

```
+ × bash

$ node app.js
Thank you. Your order was successful.
```

- **Using catch () with Promises**

더 깨끗한 코드를 쓰기위한 한 가지 방법으로 separation of concerns(별거) 라고 불리는 원리를 따르는 것이다. 이것은 분명한 섹션안에 특정한 과제를 수행하는 각각의 핸들링을 잘 정리한 코드를 의미한다. 이것의 장점은 우리의 코드를 빠르게 찾고, 어디에서 오류가 났는지 쉽게 알게 해준다.

하나의 then 안에 두개의 인자를 넣는것보단 따로 따로 넣는 것이 가독성에 훨씬 좋을 것이다.

```
prom
  .then((resolvedValue) => {
    console.log(resolvedValue);
  })
  .then(null, (rejectionReason) => {
    console.log(rejectionReason);
  });
```

자바스크립트는 공백을 무시하기때문에 이러한 형태가 가능하다.

하지만 인간의 욕심은 끝이 없는 법.. 가독성을 더욱더 높여줄 메소드가 있다.

```

prom
  .then((resolvedValue) => {
    console.log(resolvedValue);
  })
  .catch((rejectionReason) => {
    console.log(rejectionReason);
  });

```

바로 .catch() 메소드이다. 이는 인자를 onRejected 만 받는다. 마치 if 문에서 else 와 같은 느낌이다.

실습

```

1  const {checkInventory} = require('./library.js');
2
3  const order = [['sunglasses', 1], ['bags', 2]];
4
5  ▼ const handleSuccess = (resolvedValue) => {
6    console.log(resolvedValue);
7  };
8
9  ▼ const handleFailure = (rejectedValue) => {
10   console.log(rejectedValue);
11 };
12
13 // Write your code below:
14
15 checkInventory(order)
16   .then(handleSuccess)
17   .catch(handleFailure);
18

```

require 한 inventory 의 선글라스 재고가 0으로 설정되었다는 가정

```
$ node app.js
We're sorry. Your order could not be completed because some items are sold out.
$
```

- **Chaining Mutiple Promises**

우리가 비동기 프로그래밍에서 보게될 패턴은 특정한 순서로 실행되어야하는 것과 각각 따로 실행되어야하는 여러개의 동작이다. 데이터베이스에 하나의 요청을 하고 반환된 그 값으로 또 다른 요청을 할 수 있는 것이다. 즉 연쇄적으로 요청을 할 수 있다.

이를 옷을 빨는것에 비유를 해보자.

우리는 옷을 세탁기에 집어넣는다. 만약 옷이 다 빨리면 **.then**

세탁된 옷을 건조기에 넣는다. 옷이 다 마르면 **.then** 옷을 갠다.

이 연쇄된 프로미스의 과정은 composition 이라고 불린다. 프로미스는 composition으로 설계되었다!

```
firstPromiseFunction()
  .then((firstResolveVal) => {
    return secondPromiseFunction(firstResolveVal);
  })
  .then((secondResolveVal) => {
    console.log(secondResolveVal);
  });
```

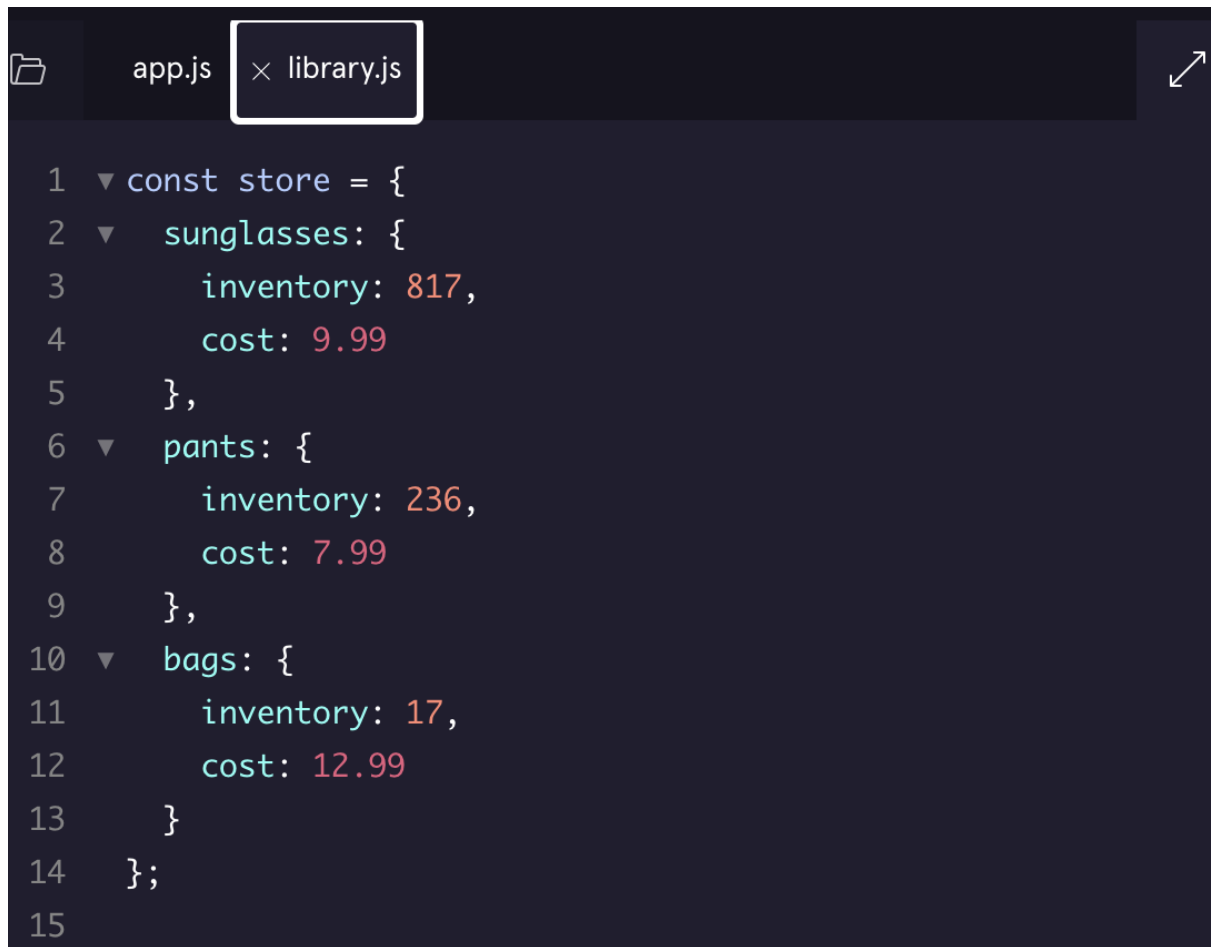
예시이다.

체인이 적절히 작동되기 위해서는 새로운 프로미스를 반환하는 함수에 꼭 return 을 넣어줘야한다.

실습

```

1  const {checkInventory, processPayment, shipOrder} =
    require('./library.js');
2
3  ▼ const order = {
4      items: [['sunglasses', 1], ['bags', 2]],
5      giftcardBalance: 79.82
6  };
7
8  checkInventory(order)
9  ▼ .then((resolvedValueArray) => {
10      // Write the correct return statement here:
11      return processPayment(resolvedValueArray);
12  })
13  ▼ .then((resolvedValueArray) => {
14      // Write the correct return statement here:
15      return shipOrder(resolvedValueArray);
16  })
17  ▼ .then((successMessage) => {
18      console.log(successMessage);
19  })
20  ▼ .catch((errorMessage) => {
21      console.log(errorMessage);
22  });
23
```



```
1 ▼ const store = {  
2   ▼ sunglasses: {  
3     inventory: 817,  
4     cost: 9.99  
5   },  
6   ▼ pants: {  
7     inventory: 236,  
8     cost: 7.99  
9   },  
10  ▼ bags: {  
11    inventory: 17,  
12    cost: 12.99  
13  }  
14 };  
15
```



```

15
16 ▼ const checkInventory = (order) => {
17 ▼   return new Promise ((resolve, reject) => {
18     setTimeout(()=> {
19       const itemsArr = order.items;
20       let inStock = itemsArr.every(item => store[item[0]].
         inventory >= item[1]);
21
22 ▼     if (inStock){
23       let total = 0;
24 ▼     itemsArr.forEach(item => {
25       total += item[1] * store[item[0]].cost
26     });
27     console.log(`All of the items are in stock. The
         total cost of the order is ${total}.`);
28     resolve([order, total]);
29 ▼   } else {
30     reject(`The order could not be completed because
         some items are sold out.`);
31   }
32 ▼ }, generateRandomDelay());
33   });
34   };

```

```

35
36 ▼ const processPayment = (responseArray) => {
37     const order = responseArray[0];
38     const total = responseArray[1];
39 ▼   return new Promise ((resolve, reject) => {
40       setTimeout(()=> {
41         let hasEnoughMoney = order.giftcardBalance >= total;
42         // For simplicity we've omitted a lot of functionality
43         // If we were making more realistic code, we would
         want to update the giftcardBalance and the inventory
44 ▼     if (hasEnoughMoney) {
45         console.log(`Payment processed with giftcard.
         Generating shipping label.`);
46         let trackingNum = generateTrackingNumber();
47         resolve([order, trackingNum]);
48 ▼     } else {
49         reject(`Cannot process order: giftcard balance was
         insufficient.`);
50     }
51
52 ▼ }, generateRandomDelay());
53     });
54 };
55

```

```

56
57 ▼ const shipOrder = (responseArray) => {
58     const order = responseArray[0];
59     const trackingNum = responseArray[1];
60 ▼   return new Promise ((resolve, reject) => {
61 ▼     setTimeout(()=> {
62         resolve(`The order has been shipped. The tracking
        number is: ${trackingNum}.`);
63 ▼     }, generateRandomDelay());
64     });
65   };
66
67
68   // This function generates a random number to serve as a
   "tracking number" on the shipping label. In real life
   this wouldn't be a random number
69 ▼ function generateTrackingNumber() {
70     return Math.floor(Math.random() * 1000000);
71   }
72
73   // This function generates a random number to serve as
   delay in a setTimeout() since real asynchronous operations
   take variable amounts of time
74 ▼ function generateRandomDelay() {
75     return Math.floor(Math.random() * 2000);
76   }
77
78   module.exports = {checkInventory, processPayment,
   shipOrder};

```

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/21a74a09-b948-4346-99d3-a03d3b72a8bc/__2021-07-10_12.31.32.mov

터미널 출력 내용

- **Avoiding Common Mistakes**

프로미스 컴포지션은 중첩된 콜백 함수보다 훨씬 더 좋은 가독성을 제공한다. 그러나 실수할 수 있는 여지가 있다. 프로미스 composition 의 두가지 공통된 mistakes 를 살펴보자.

⇒ 첫번째 mistake : 체이닝 프로미스가 아닌 중첩된 프로미스 사용

```
returnsFirstPromise()
  .then((firstResolveVal) => {
    return returnsSecondValue(firstResolveVal)
      .then((secondResolveVal) => {
        console.log(secondResolveVal);
      })
  })
})
```

두번째 then 은 첫번째 then 의 결과값을 갖는다?

⇒ 두번째 mistake : return 까먹기

```
returnsFirstPromise()
  .then((firstResolveVal) => {
    returnsSecondValue(firstResolveVal)
  })
  .then((someVal) => {
    console.log(someVal);
  })
```

두번째 then 은 처음 then의 결과값을 받아야하는데 returnsFirstPromise 의 프로미스를 받게 된다.

리턴을 까먹어도 에러를 내놓지 않는다. 이는 디버깅하는데 꽤 골치 아플 것이다. 그러니 return 을 꼭 확인하자!!

또한 then 다음 then 이 있다면 ';' 을 쓰면 안된다 !!

```

8 // Refactor the code below:
9
10 ▼ checkInventory(order)
11 ▼   .then((resolvedValueArray) => {
12       return processPayment(resolvedValueArray)}))
13 ▼   .then((resolvedValueArray) => {
14       return shipOrder(resolvedValueArray)}))
15 ▼   .then((successMessage) => {
16       console.log(successMessage);
17   });
18

```

- **Using Promise.all()**

promise composition 의 순서를 제어할 수 없을까?

우리의 집을 청소하는 것을 예로 들어보자. 우리는 옷이 마르고, 쓰레기통이 비워지고 디시워시가 돌아가길 원한다. 우리는 이 모든 과업들이 순서에 상관없이 완료되는 것을 원한다. 게다가 그들은 모두 비동기적으로 완료되기 때문에, 그들은 동시에 일어나야한다.

효율성을 극대화하기 위해 우리는 많은 비동기적 연산이 함께 일어나는 "concurrency" 를 사용해야한다. 프로미스들과 함께 우리는 Promise.all() 메서드를 이용하여 이것들을 할 수 있다.

Promise.all() 은 인자로써 프로미스들의 배열을 받는다. 그리고 하나의 프로미스를 리턴한다. 그 리턴되는 하나의 프로미스는 다음 두가지 방법중 하나로 정착될 것이다.

⇒ 1. 만약 인자안의 모든 프로미스가 resolve 라면, promise.all()은 각각 프로미스의 resolve 값이 포함된 하나의 배열을 resolve 할 것이다.

⇒ 2. 만약 인자안의 프로미스 중 reject가 있다면, Promise.all() 은 즉시 reject의 이유와 함께 reject 한다. 이러한 것을 ' *failing fast* ' 라고 부른다.

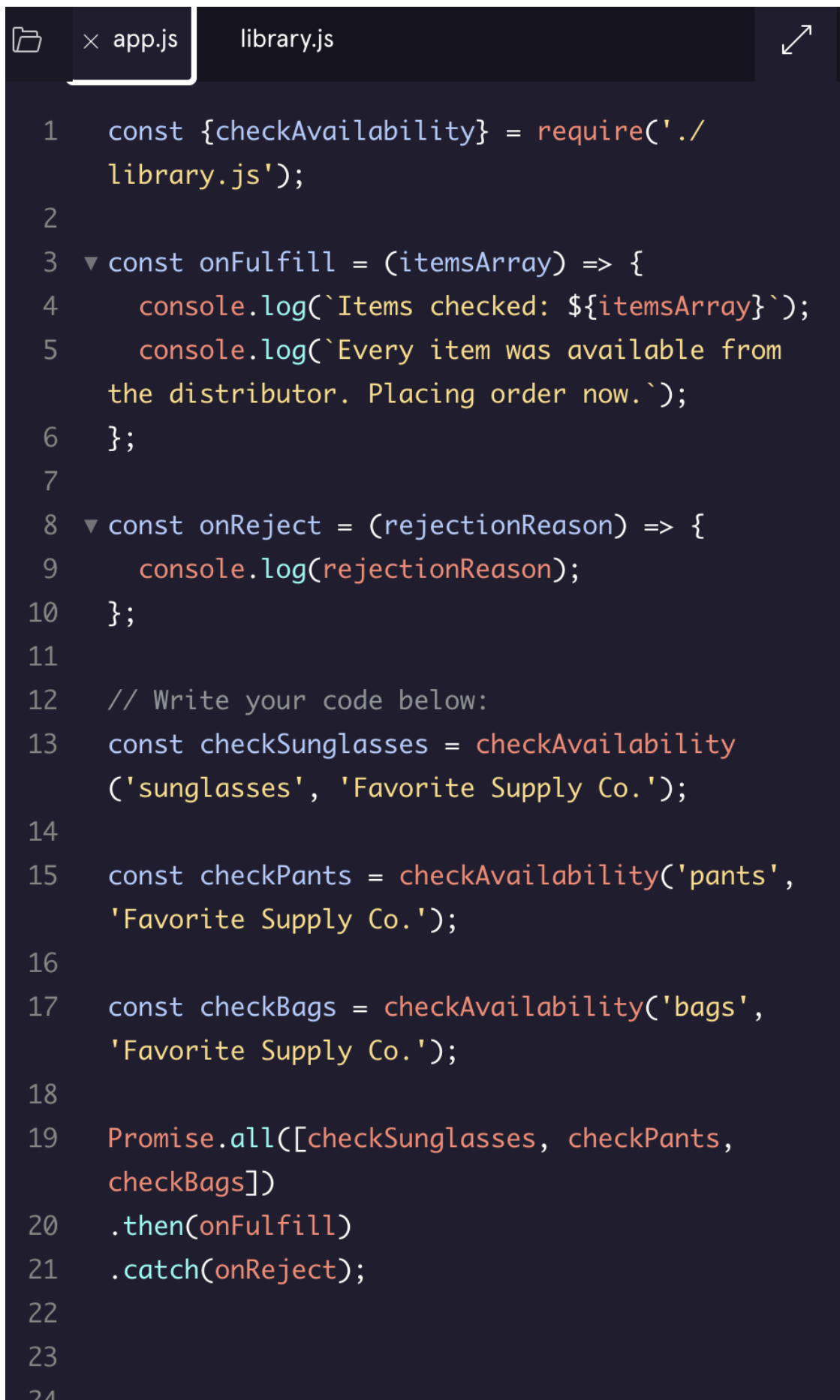
```
let myPromises = Promise.all([returnsPromOne(),
returnsPromTwo(), returnsPromThree()]);

myPromises
  .then((arrayOfValues) => {
    console.log(arrayOfValues);
  })
  .catch((rejectionReason) => {
    console.log(rejectionReason);
  });
```

실습

```
app.js × library.js ↗

1  const checkAvailability = (itemName,
  ▼ distributorName) => {
2      console.log(`Checking availability of $
    {itemName} at ${distributorName}...`);
3  ▼  return new Promise((resolve, reject) => {
4  ▼      setTimeout(() => {
5  ▼          if (restockSuccess()) {
6              console.log(`${itemName} are
    in stock at ${distributorName}`)
7              resolve(itemName);
8  ▼          } else {
9              reject(`Error: ${itemName} is
    unavailable from ${distributorName} at this
    time.`);
10         }
11         }, 1000);
12     });
13 };
14
15 module.exports = { checkAvailability };
16
17
18 // This is a function that returns true 80%
    of the time
19 // We're using it to simulate a request to
    the distributor being successful this often
20 ▼ function restockSuccess() {
21     return (Math.random() > .2);
22 }
```



```
1  const {checkAvailability} = require('./
  library.js');
2
3  ▼ const onFulfill = (itemsArray) => {
4    console.log(`Items checked: ${itemsArray}`);
5    console.log(`Every item was available from
  the distributor. Placing order now.`);
6  };
7
8  ▼ const onReject = (rejectionReason) => {
9    console.log(rejectionReason);
10 };
11
12 // Write your code below:
13 const checkSunglasses = checkAvailability
  ('sunglasses', 'Favorite Supply Co.');
```

```
14
15 const checkPants = checkAvailability('pants',
  'Favorite Supply Co.');
```

```
16
17 const checkBags = checkAvailability('bags',
  'Favorite Supply Co.');
```

```
18
19 Promise.all([checkSunglasses, checkPants,
  checkBags])
20 .then(onFulfill)
21 .catch(onReject);
22
23
24
```



```
+ × bash ↗  
  
$ node app.js  
Checking availability of sunglasses at Favorite Supply Co....  
Checking availability of pants at Favorite Supply Co....  
Checking availability of bags at Favorite Supply Co....  
sunglasses are in stock at Favorite Supply Co.  
pants are in stock at Favorite Supply Co.  
bags are in stock at Favorite Supply Co.  
Items checked: sunglasses,pants,bags  
Every item was available from the distributor. Placing order now.  
$
```

- **Review**

프로미스는 숙련된 개발자에게도 어려운 개념이다. 지금까지 매우 잘해왔다!

⇒ 프로미스는 비동기적 연산의 최종적인 결과를 나타내는 자바스크립트의 객체이다.

⇒ 프로미스는 세가지 중 하나의 상태가 될 수 있다. (pending, resolved, rejected)

⇒ 프로미스는 resolved , rejected 로 둘 중 하나로 정착된다.

⇒ 우리는 new 키워드를 이용해서 프로미스를 생성한다. 그리고 executor 함수를 Promise 생성자 함수에 넣는다.

⇒ setTimeout() 는 event - loop 를 사용하면서 콜백함수의 실행을 딜레이 시키는 노드 함수이다.

⇒ .then() 함수는 다음에 무엇을 해야하는지 로직을 결정하는 성공적인 핸들러 콜백과 함께 사용한다.

⇒ .catch() 함수는 다음에 무엇을 해야하는지 로직을 결정하는 failure 함수와 함께 쓰인다.

⇒ 프로미스 composition 은 복잡하고 비동기적인 코드를 읽기 쉽도록 작성하게 해준다. 다수의 then 과 catch 를 체이닝하여 composition 을 형성한다.

⇒ 프로미스 composition 을 정확하게 사용하기 위해, then() 안에 return 을 꼭 써야 하는 것과 중첩된 then 을 쓰지 말아야하는 것을 기억해라.

⇒ 동시동작의 이점을 얻기 위해, Promise.all() 을 사용할 수 있다.

오늘의 단어

- notorious : 악명이 높은
- eventual : 최종적인
- come with : ~이 딸려있다.
- pat : 토닥이다.