11. callback function

1. HTML <script> 태그

1) 정의 및 특징
<script> 태그의 defer 속성은 페이지가 모두 로드된 후에 해당 외부 스크립트가 실행됨을 명시한다. defer 속성은 boolean 속성으로 명시하지 않으면 false 값을 가지게 되고, 명시하면 true 값을 가지게 된다. 이 속성은 <script> 요소가 외부 스크립트를 참조하는 경우에만 사용할 수 있으므로, src 속성이 명시된 경우에만 사용할 수 있다.

참조된 외부 스크립트 파일을 다음과 같이 여러 가지 방법으로 실행시킬 수 있다.
- ☞ async 속성이 명시된 경우 : 브라우저가 페이지를 파싱되는 동안에도 스크립트가 실행됨.
- ☞ async 속성은 명시되어 있지 않고 defer 속성만 명시된 경우 : 브라우저가 페이지의 파싱을 모두 끝내면 스크립트가 실행됨.
- ☞ async 속성과 defer 속성이 모두 명시되어 있지 않은 경우 : 브라우저가 페이지를 파싱하기 전에 스크립트를 가져와 바로 실행시킴.

```
//index.html
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <script src="async/callback.js" defer></script>
</head>

<body>

</body>

</html>
```

```
//callback.js
'use strict';

// JavaScript is synchronous
```

```
// Execute the code block by orger after hoisting
// hoisting : var, function declaration
// callback function : 우리가 전달한 함수를 나중에 불러주는 함수
console.log('1');
// setTimeout(function () {
//   console.log('2');
// }, 1000);
setTimeout(() => console.log('2'), 1000);
console.log('3');

// Synchronous callback
function printImmediately(print) {
  print();
}
printImmediately(() => console.log('Hello'));

// Asynchronous callback
function printWithDelay(print, timeout) {
  setTimeout(print, timeout);
}
printWithDelay(() => console.log('async callback'), 2000);


// Callback Hell example
class UserStorage {
  loginUser(id, password, onSuccess, onError) {
    setTimeout(() => {
      if (
        (id === 'smart01' && password === 'smu01') ||
        (id === 'ecom02' && password === 'smu02')
      ) {
        onSuccess(id);
      } else {
        onError(new Error('not found'));
      }
    }, 2000);
  }

  getRoles(user, onSuccess, onError) {
    setTimeout(() => {
      if (user === 'smart01') {
        onSuccess({ name: 'smart01', role: 'admin' });
      } else {
        onError(new Error('no access'));
```

```
      }
    }, 1000);
  }
}

const userStorage = new UserStorage();
const id = prompt('Enter your id');
const password = prompt('Enter your password');
userStorage.loginUser(
  id,
  password,
  user => {
    userStorage.getRoles(
      user,
      userWithRole => {
        alert(`Hello ${userWithRole.name}, you have a ${userWithRole.role} role.`);
      },
      error => {
        console.log(error);
      }
    );
  },
  error => {
    console.log(error);
  }
);
```

12. Promise

Promise는 비동기를 간편하게 처리할 수 있도록 도와주는 Object 이다. Promise는 정해진 시간에 기능을 수행하고 나서 정상적으로 기능이 수행되어 졌다면 성공의 메시지와 함께 처리된 결과값을 전달해 준다. 만약 기능을 수행하다가 예상하지 못한 문제가 발생하였다면 에러를 전달해 준다.

예를 들면, 수강신청 프로그램에서 개설된 교육과정이 있는데 언제 교육과정이 개설될지를 모르는 시점이며, 관심있는 A 학생은 이메일을 통해서 미리 등록할 수 있는 시스템이라고 가정한다. A 학생이 등록한 후 교육과정이 개설되면 등록한 A 학생에게 이메일로 공지를 하게 된다. 미리 등록한 A 학생은 교육과정이 개설되면 공지를 받을 수 있다.
B 학생은 교육과정이 개설된 후 늦게 교육과정이 개설된 공지를 발견하게 되었다. 뒤늦게 B 학생은 이메일을 입력하고 등록을 하게 되었다. 그러나 교육과정은 이미 개설되었기 때문에 기다릴 필요 없이 바로 B 학생에게 메일로 공지가 가서 수업에 바로 참여할 수 있게 된다.
자 이제부터 callback을 사용하지 않고 promise Object를 통해서 비동기 코드를 작성해 보도록 하자.

1) Promise 만들기

아래와 같이 Promise를 만들면 Promise를 만드는 순간 우리가 전달한 executor 라는 callback 함수가 바로 실행이 되는 것을 확인할 수 있다. 그러나 Promise 안에 네트워크 통신을 하는 코드를 작성하였다면 Promise가 만들어지는 순간 바로 네트워크 통신을 수행하게 된다. 만약 네트워크 요청을 사용자가 요구했을 때만 해야하는 경우에는 사용자가 원하지도 않을 경우에도 불필요한 네트워크 통신이 일어날 수도 있다.

//promise.js

```
'use strict';

// Promise is a JavaScript Object for asynchronous operation.
// State : pending -> fulfilled or rejected
// Producer vs Consumer

// 1. Producer
const promise = new Promise((resolve, reject) => {
  // doing some heavy work (network, read files)
  console.log('doing somethinmg...');
});
```

2) Promise 사용하기

```
//promise.js
'use strict';

// Promise is a JavaScript Object for asynchronous operation.
// State : pending -> fulfilled or rejected
// Producer vs Consumer

// 1. Producer
// when new Promise is created, the executor run automatically.
const promise = new Promise((resolve, reject) => {
  // doing some heavy work (network, read files)
  console.log('doing somethinmg...');
  setTimeout(() => {
    //resolve('smart01');
    reject(new Error('no network'));
  }, 2000);
});

// 2. Consumers : then, catch, finally -> chaining
promise //
  .then((value) => {
    console.log(value);
  })
  .catch(error => {
    console.log(error);
  })
  .finally(() => {
    console.log('finally');
  });
```

3) Promise 연결하기

```
// 3. Promise chaining
const fetchNumber = new Promise((resolve, reject) => {
  setTimeout(() => resolve(1), 1000);
});

fetchNumber //
  .then(num => num * 2)
  .then(num => num * 3)
```

```
  .then(num => {
    return new Promise((resolve, reject) => {
      setTimeout(() => resolve(num - 1), 1000);
    });
  })
  .then(num => console.log(num));
```

4) 오류를 잘 처리하자

```
// 4. Error Handling -1
const getHen = () =>
  new Promise((resolve, reject) => {
    setTimeout(() => resolve('□'), 1000);
  });
const getEgg = hen =>
  new Promise((resolve, reject) => {
    setTimeout(() => resolve(`${hen} => □`), 1000);
  });
const cook = egg =>
  new Promise((resolve, reject) => {
    setTimeout(() => resolve(`${egg} => □`), 1000);
  });

getHen() //
  .then(hen => getEgg(hen))
  .then(egg => cook(egg))
  .then(meal => console.log(meal));

// getHen()
// .then(getEgg)
// .then(cook)
// .then(console.log);


// 4. Error Handling -2
const getHen = () =>
  new Promise((resolve, reject) => {
    setTimeout(() => resolve('암닭'), 1000);
  });
const getEgg = hen =>
  new Promise((resolve, reject) => {
    //setTimeout(() => resolve(`${hen} => 계란`), 1000);
```

```
    setTimeout(() => reject(new Error(`error! ${hen} => 계란`)), 1000);
  });
const cook = egg =>
  new Promise((resolve, reject) => {
    setTimeout(() => resolve(`${egg} => 계란후라이`), 1000);
  });

// getHen()
// .then(hen => getEgg(hen))
// .then(egg => cook(egg))
// .then(meal => console.log(meal));

getHen() //
  .then(getEgg)
  .catch(error => {
    return '빵';
  })
  .then(cook)
  .then(console.log)
  .catch(console.log);
```

5) Callback Hell을 리펙토링하자

```
// Callback Hell example -> Refactoring
class UserStorage {
  loginUser(id, password) {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        if (
          (id === 'smart01' && password === 'smu01') ||
          (id === 'ecom02' && password === 'smu02')
        ) {
          resolve(id);
        } else {
          reject(new Error('not found'));
        }
      }, 2000);
    });
  }

  getRoles(user) {
    return new Promise((resolve, reject) => {
```

```
      setTimeout(() => {
        if (user === 'smart01') {
          resolve({ name: 'smart01', role: 'admin' });
        } else {
          reject(new Error('no access'));
        }
      }, 1000);
    });
  }
}

const userStorage = new UserStorage();
const id = prompt('Enter your id');
const password = prompt('Enter your password');

userStorage
  .loginUser(id, password)
  .then(userStorage.getRoles)
  .then(user => alert(`Hello ${user.name}, you have a ${user.role} role.`))
  .catch(console.log);
```

13. async, await

1. async

async와 await은 Promise를 좀 더 간결하고 간편하고 동기적으로 실행되는 것처럼 보이게 만들어 준다. 이것을 'syntactic sugar'라고 한다. 여기에는 class도 있다.

```
// async & await
// clear style of using promise :)

// 1. async -1
function fetchUser() {
  // do network request in 10 secs ...
  return 'smart01';
}

const user = fetchUser();
console.log(user);
```

```
// 1. async -2
```

☞ resolve와 reject를 호출하지 않고 이렇게 return을 하면 Promise가 pending 상태가 된다.

```
function fetchUser() {
  return new Promise((resolve, reject) => {
    // do network request in 10 secs ...
    //return 'smart01';
    resolve('smart01')  ;
  });
}

const user = fetchUser();
console.log(user);
```

☞ 따라서 resolve('smart01');를 해주어야 한다.

```
// 1. async -3
async function fetchUser() {
  // do network request in 10 secs ...
  return 'smart01';
}
```

```
const user = fetchUser();
user.then(console.log);
console.log(user);
```

☞ 이제는 Promise를 이용하지 않고 간편하게 비동기를 작설할 수 있다. 바로 함수 앞에 async를 붙여주면 된다.

2. await

```
// 2. await -1
function delay(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

async function getApple() {
  await delay(1000);
  return '사과';
}

async function getBanana() {
  await delay(1000);
  return '바나나';
}

// function getBanana(){
//   return delay(1000)
//   .then(() => "바나나");
// }

// function pickFruits() {
//   return getApple()
//     .then(apple => {
//       return getBanana().then(banana => `${apple} + ${banana}`);
//     });
// }

async function pickFruits() {
  const apple = await getApple();
  const banana = await getBanana();
  return `${apple} + ${banana}`;
}

pickFruits().then(console.log);
```

3. await 병렬처리

위의 코드의 문제점은 사과를 받는데 1초가 걸리고, 바나나를 받는데 1초가 걸리기 때문에 이렇게 순차적으로 진행하면 비효율적이 된다. 사과와 바나나를 받아오는데는 서로 연관이 되지 않기 때문에 서로 기다릴 필요가 전혀 없다. 따라서 Refactoring을 하도록 하자.
다음과 같이 처리를 하면 병렬적으로 처리할 수 있다.

```
// 2. await -2
function delay(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

async function getApple() {
  await delay(1000);
  return '사과';
}

async function getBanana() {
  await delay(1000);
  return '바나나';
}

// function getBanana(){
//   return delay(1000)
//   .then(() => "바나나");
// }

// function pickFruits() {
//   return getApple()
//     .then(apple => {
//       return getBanana().then(banana => `${apple} + ${banana}`);
//     });
// }

async function pickFruits() {
  const applePromise = getApple();
  const bananaPromise = getBanana();
  const apple = await applePromise;
  const banana = await bananaPromise;
  return `${apple} + ${banana}`;
}

pickFruits().then(console.log);
```

4. 유용한 Promise

이와 같이 서로 연관이 없이 병렬적으로 처리하는 경우에는 위와 같이 코드를 작성하지 않고 Promise에서 제공하는 유용한 API가 있다.

```
// 2. await -3
function delay(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

async function getApple() {
  await delay(2000);
  return '사과';
}

async function getBanana() {
  await delay(1000);
  return '바나나';
}

async function pickFruits() {
  const applePromise = getApple();
  const bananaPromise = getBanana();
  const apple = await applePromise;
  const banana = await bananaPromise;
  return `${apple} + ${banana}`;
}

pickFruits().then(console.log);


// 3. useful Promise APIs
function pickAllFruits() {
  return Promise.all([getApple(), getBanana()])
    .then(fruits => fruits.join(' + '));
}

pickAllFruits().then(console.log);

function pickOnlyOne() {
  return Promise.race([getApple(), getBanana()]);
}

pickOnlyOne().then(console.log);
```