Seoul Software ACademy

웹 개발자 부트캠프 과정

SeSAC x CODINGOn

With. 팀 뤼쳐드

setTimeout()

```
setTimeout(code, delay); // delay 동안 기다리다가 code 함수를 실행
```

```
console.log(1);
setTimeout(function () {
  console.log(2);
}, 2000);
console.log(3);
```

🚀 **실행 결과**

1
3
2

1 -> 2 -> 3 이 아닌 **1 -> 3 -> 2 로 출력됨!!**

비동기 처리란?

- 특정 코드의 연산이 끝날 때까지 코드의 실행을 **멈추지 않고 다음 코드를 먼저 실행**하는 자바스크립트의 특성
- 왜 비동기 처리일까?
  - 서버로 데이터를 요청 시, 서버가 언제 그 요청에 대한 응답을 줄지도 모르는데 마냥 다른 코드를 실행 안 하고 기다릴 순 없음!
  - 비동기 처리가 아니고 동기 처리라면 코드 실행하고 기다리고, 실행하고 기다리고..
  - 웹을 실행하는데 수십 분이 걸릴 수도…?

비동기 처리

- 아래 코드에서 문제점은?

```
// 편의점에 들어가서 음료수를 사고 나오는 상황
function goMart() {
  console.log('마트에 가서 어떤 음료를 살지 고민한다.');
}

function pickDrink() {
  setTimeout(function () {
    console.log('고민 끝!!');
    product = '제로 콜라';
    price = 2000;
  }, 3000);
}

function pay(product, price) {
  console.log(`상품명: ${product}, 가격: ${price}`);
}
```

```
let product;
let price;
goMart();
pickDrink();
pay(product, price);
```

실행 결과

```
마트에 가서 어떤 음료를 살지 고민한다.
상품명: undefined, 가격: undefined
고민 끝!!
```

왜 undefined가 뜰까??

비동기 코드를 처리하기 위한 3가지 방법

1. callback 함수
2. promise
3. async await

Callback 함수

콜백(callback) 함수

- JavaScript는 함수를 인자로 받고 다른 함수를 통해 반환될 수 있는데, **인자(매개변수)로 대입되는 함수**를 콜백함수라고 한다.

- 즉, **다른 함수가 실행을 끝낸 뒤 실행되는 함수**

- 함수를 선언할 때는 **parameter(인자, 매개변수)로 함수를** 받아서 쓸 수 있다.

콜백(callback) 함수

Q. 콜백 함수를 **왜 사용할까?**

A. 어떤 함수가 다른 함수의 실행을 끝낸 뒤 실행되는 것을 보장하기 위해

독립적으로 수행되는 작업도 있는 반면 **응답을 받은 이후 처리되어야 하는 종속적인 작업**도 있을 수 있으므로 그에 대한 대응 방법이 필요

```html
<button type="button">Click Me!</button>
```

```js
const button = document.querySelector('button');
function sayHello() {
  console.log('hello~~~');
}
button.addEventListener('click', sayHello);
```

콜백 함수!!

sayHello 함수는 button 요소에 **"클릭 이벤트가 발생했을 때" 실행**되어야 한다.

콜백(callback) 함수

*용어 정리: 파라미터 = 매개변수 = 인자

- 보통 함수를 선언한 뒤에 **함수 타입 파라미터**를 맨 마지막에 하나 더 선언해 주는 방식으로 정의

```
function mainFunc(param1, param2, callback) {
    callback(result);
}

function callbackFunc(param){
    console.log("콜백함수 입니다");
}

mainFunc(1, 2, callbackFunc);
```

```
// mainFunc(1, 2, callbackFunc);
// callbackFunc(result);
```

콜백함수 사용 예시

콜백(callback) 함수

```
function goMart() {
  console.log('마트에 가서 어떤 음료를 살지 고민한다.');
}

function pickDrink(callback) {
  setTimeout(function () {
    console.log('고민 끝!!');
    product = '제로 콜라';
    price = 2000;
    callback(product, price);
  }, 3000);
}

function pay(product, price) {
  console.log(`상품명: ${product}, 가격: ${price}`);
}
```

```
let product;
let price;
goMart();
pickDrink(pay);
```

실행 결과

```
마트에 가서 어떤 음료를 살지 고민한다.
고민 끝!!
상품명: 제로 콜라, 가격: 2000
```

콜백 함수를 이용하니 값이 제대로 들어온다!!

콜백(callback) 함수

```
function goMart() {
  console.log('마트에 가서 어떤 음료를 살지 고민한다.');
}

function pickDrink(callback) {
  setTimeout(function () {
    console.log('고민 끝!!');
    product = '제로 콜라';
    price = 2000;
    callback(product, price);
  }, 3000);
}
```

이전 페이지와 동일한 코드!

🚀 실행 결과

```
마트에 가서 어떤 음료를 살지 고민한다.
고민 끝!!
상품명: 제로 콜라, 가격: 2000
```

```
let product;
let price;
goMart();
pickDrink(function pay(product, price) {
  console.log(`상품명: ${product}, 가격: ${price}`);
}); // pickDrink(pay);
```

콜백 지옥(Callback Hell)

- 비동기 프로그래밍 시 발생하는 문제
- 함수의 매개변수로 넘겨지는 콜백 함수가 반복되어 코드의 들여쓰기가 너무 깊어지는 현상
- **가독성 ↓ 코드 수정 난이도 ↑**

콜백 지옥 

이런 코드.. 만나고 싶으신가요..? 😫

```
step1(function (value1) {
    step2(function (value2) {
        step3(function (value3) {
            step4(function (value4) {
                step5(function (value5) {
                    step6(function (value6) {
                        // Do something with value6
                    });
                });
            });
        });
    });
});
```

```
step1(function (err, value1) {
    if (err) {
        console.log(err);
        return;
    }
    step2(function (err, value2) {
        if (err) {
            console.log(err);
            return;
        }
        step3(function (err, value3) {
            if (err) {
                console.log(err);
                return;
            }
            step4(function (err, value4) {
                // 정신 건강을 위해 생략
            });
        });
    });
});
```

Promise

Promise

- 비동기 처리를 할 수 있는 **객체**
- 비동기 작업이 맞이할 미래의 완료 또는 실패와 그 결과 값을 나타냄
  - 그래서 promise 라는 용어를 사용
- **성공**과 **실패**를 분리하여 반환
- 비동기 작업이 완료된 이후에 **다음 작업을 연결시켜 진행**할 수 있는 기능을 가짐

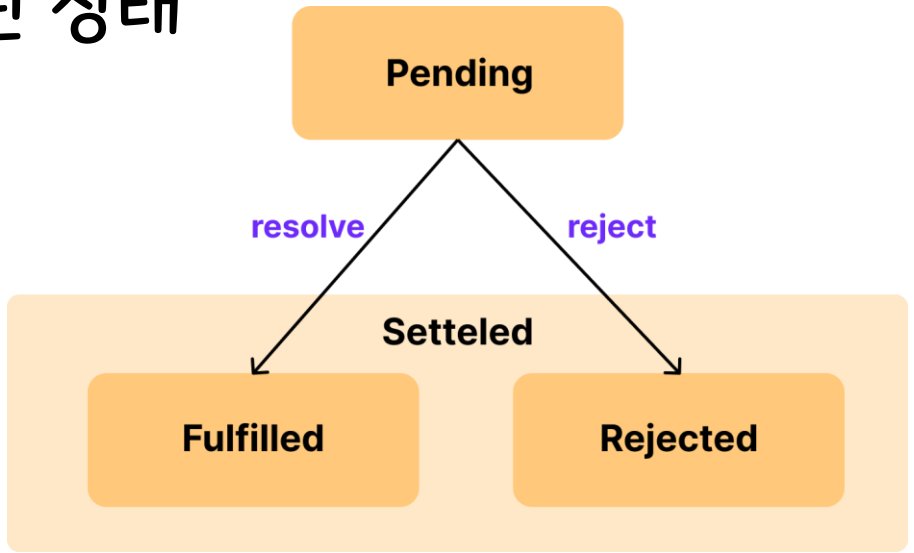
Promise

- ES6 에서 추가된 JS 문법
- new Promise 로 만들어서 사용!
  - new Promise가 만들어질 때, executor(실행 함수)가 자동으로 실행
  - executor의 인수인 reject 와 resolve
  - Promise가 생성되면 작업을 실행하고, 작업의 완료 여부를 executor의 매개변수를 통해서 전달

- executor 매개변수 (resolve와 reject 모두 callback)
  - resolve : 비동기 작업이 성공했을 때
  - reject : 비동기 작업이 실패했을 때

Promise의 상태

- **Pending**(대기) : Promise를 수행 중인 상태
- **Fulfilled**(이행) : Promise가 **Resolve** 된 상태 (성공)
- **Rejected**(거부) : Promise가 지켜지지 못한 상태. **Reject** 된 상태 (실패)
- **Settled** : fulfilled 혹은 rejected로 결론이 난 상태



Promise 사용법

Promise 객체를 반환하는 **promise1 함수 정의**

```
function promise1(flag) {
  return new Promise(function (resolve, reject) {
    if (flag) {
      resolve('promise 상태는 fulfilled!!! then으로 연결됩니다.\n 이때의 flag가 true입니다.');
    } else {
      reject('promise 상태는 rejected!!! catch로 연결됩니다. \n 이때의 flag는 false입니다.');
    }
  });
}
```

- Promise는 **두 가지 콜백 함수**를 가짐
  - **resolve(**value**)** : 작업이 **성공**(fulfilled)한 경우, 그 결과를 value와 함께 호출
  - **reject(**error**)**: **에러**(rejected) 발생 시 에러 객체를 나타내는 error와 함께 호출

Promise 사용법

promise1 함수 호출 **(성공)**

```
promise1(true)
  .then(function (result) {
    console.log(result);
  })
  .catch(function (err) {
    console.log(err);
  });
```

promise 상태는 fulfilled!!! then으로 연결됩니다.
이때의 flag가 true입니다.

promise1 함수 호출 **(실패)**

```
promise1(false)
  .then(function (result) {
    console.log(result);
  })
  .catch(function (err) {
    console.log(err);
  });
```

promise 상태는 rejected!!! catch로 연결됩니다.
이때의 flag는 false입니다.

- **resolve() -> then 메서드 실행**
- **reject() -> catch 메서드 실행**

Promise 사용법

```
function goMart() {
  console.log('마트에 가서 어떤 음료를 살지 고민한다.');
}

function pickDrink() {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      console.log('고민 끝!!');
      product = '제로 콜라';
      price = 2000;
      resolve();
    }, 3000);
  });
}

function pay() {
  console.log(`상품명: ${product}, 가격: ${price}`);
}
```

p.8에서 “콜백함수”를 이용해 동기 처리한 것을

“promise”를 이용해 동기 처리해보자

```
let product;
let price;
goMart();
pickDrink().then(pay);
```

Promise 사용법

```
function goMart() {
  console.log('마트에 가서 어떤 음료를 살지 고민한다.');
}

function pickDrink() {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      console.log('고민 끝!!');
      product = '제로 콜라';
      price = 2000;
      resolve();
    }, 3000);
  });
}
```

이전 페이지와 동일한 코드!

```
let product;
let price;
goMart();
pickDrink().then(function () {
  console.log(`상품명: ${product}, 가격: ${price}`);
});
```

Promise 체이닝 사용 안한 경우

```
function add(n1, n2, cb) {
  setTimeout(function () {
    let result = n1 + n2;
    cb(result);
  }, 1000);
}

function mul(n, cb) {
  setTimeout(function () {
    let result = n * 2;
    cb(result);
  }, 700);
}

function sub(n, cb) {
  setTimeout(function () {
    let result = n - 1;
    cb(result);
  }, 500);
}
```

// 함수를 이용해 (4+3)*2-1 = 13 연산을 해보자!
// 연산 순서: 덧셈 -> 곱셈 -> 뺄셈

콜백함수 사용시 -> **콜백 지옥!!!!** ✕

```
add(4, 3, function (x) {
  console.log('1: ', x);
  mul(x, function (y) {
    console.log('2: ', y);
    sub(y, function (z) {
      console.log('3: ', z);
    });
  });
});
```

🚀 **실행 결과**

```
1:  7  // 4+3
2:  14 // (4+3)*2
3:  13 // (4+3)*2-1
```

Promise 체이닝 사용한 경우

```
function add(n1, n2) {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      let result = n1 + n2;
      resolve(result);
    }, 1000);
  });
}

function mul(n) {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      let result = n * 2;
      resolve(result);
    }, 700);
  });
}

function sub(n) {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      let result = n - 1;
      resolve(result);
    }, 500);
  });
}
```

// 함수를 이용해 (4+3)*2-1 = 13 연산을 해보자!
// 연산 순서: 덧셈 -> 곱셈 -> 뺄셈

```
add(4, 3)
  .then(function (result) { // result: 7
    console.log('1: ', result);
    return mul(result);
  })
  .then(function (result) { // result: 14
    console.log('2: ', result);
    return sub(result);
  })
  .then(function (result) { // result: 13
    console.log('3: ', result);
  });
```

Promise 체이닝 장점(1)

```
function add(n1, n2) {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      let result = n1 + n2;
      resolve(result);
    }, 1000);
  });
}

function mul(n) {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      let result = n * 2;
      resolve(result);
    }, 700);
  });
}

function sub(n) {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      let result = n - 1;
      resolve(result);
    }, 500);
  });
}
```

- **then 메서드 연속 사용** → 순차적인 작업 가능
- 콜백 지옥에서 탈출!

```
add(4, 3)
  .then(function (result) { // result: 7
    console.log('1: ', result);
    return mul(result);
  })
  .then(function (result) { // result: 14
    console.log('2: ', result);
    return sub(result);
  })
  .then(function (result) { // result: 13
    console.log('3: ', result);
  });
```

🚀 **실행 결과**

```
1:  7  // 4+3
2:  14 // (4+3)*2
3:  13 // (4+3)*2-1
```

Promise 체이닝 장점(2)

```
function add(n1, n2) {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      let result = n1 + n2;
      resolve(result);
    }, 1000);
  });
}

function mul(n) {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      let result = n * 2;
      resolve(result);
    }, 700);
  });
}

function sub(n) {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      let result = n - 1;
      // resolve(result);
      reject(new Error('의도적으로 에러를 일으켜봤음!'));
    }, 500);
  });
}
```

- 예외 처리 간편
- 마지막 **catch** 구문에서 **한 번에 에러 처리** 가능

```
add(4, 3)
  .then(function (result) { // result: 7
    console.log('1: ', result);
    return mul(result);
  })
  .then(function (result) { // result: 14
    console.log('2: ', result);
    return sub(result);
  })
  .then(function (result) {
    console.log('3: ', result);
  })
  .catch(function (err) {
    console.log('실패!');
    console.log(err);
  });
```

실행 결과

```
1:  7
2:  14
실패!
Error: 의도적으로 에러를 일으켜봤음!
    at index2.js:159:14
```

Async / Await

async/await

- Promise 도 chaning을 하다보면 then().then()… 처럼 꼬리를 물게 되어 코드의 가독성이 떨어질 수 있습니다.

- Promise 보다 직관적인 코드를 위해 등장한 async와 await!
- 기능이 추가된 것이 아닌, Promise를 다르게 사용하는 것

async/await

프로미스 기반 코드를 좀 더 쓰기 쉽고 읽기 쉽게 하기 위해 등장!!

비동기 처리 패턴 중 가장 최근에 나온 문법

- **async**
  - 함수 앞에 붙여 Promise를 반환한다.
  - 프로미스가 아닌 값을 반환해도 프로미스로 감싸서 반환한다.

- **await**
  - '기다리다' 라는 뜻을 가진 영단어
  - 프로미스 앞에 붙여 프로미스가 다 처리될 때까지 기다리는 역할을 하며 결과는 그 후에 반환한다.

async/await

- Async : 비동기 실행되는 게 있음을 알림
- Await : 실행 다 될 때까지 기다려주세요!

```
async function f1() {
  return 1;
}

async function f2() {
  return Promise.resolve(1);
}
```

  - async가 붙은 함수는 항상 Promise를 반환합니다.
  - async가 있는 함수에서만 await 키워드가 사용 가능합니다.

Promise -> async / await

프로미스 기반 코드를 좀 더 쓰기 쉽고 읽기 쉽게 하기 위해 등장!!

비동기 처리 패턴 중 가장 최근에 나온 문법

```
function goMart() {
  console.log('마트에 가서 어떤 음료를 살지 고민한다.');
}

function pickDrink() {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      console.log('고민 끝!!');
      product = '제로 콜라';
      price = 2000;
      resolve();
    }, 3000);
  });
}

function pay() {
  console.log(`상품명: ${product}, 가격: ${price}`);
}
```

```
async function exec() {
  goMart();
  await pickDrink();
  pay();
}

let product;
let price;
exec();
```