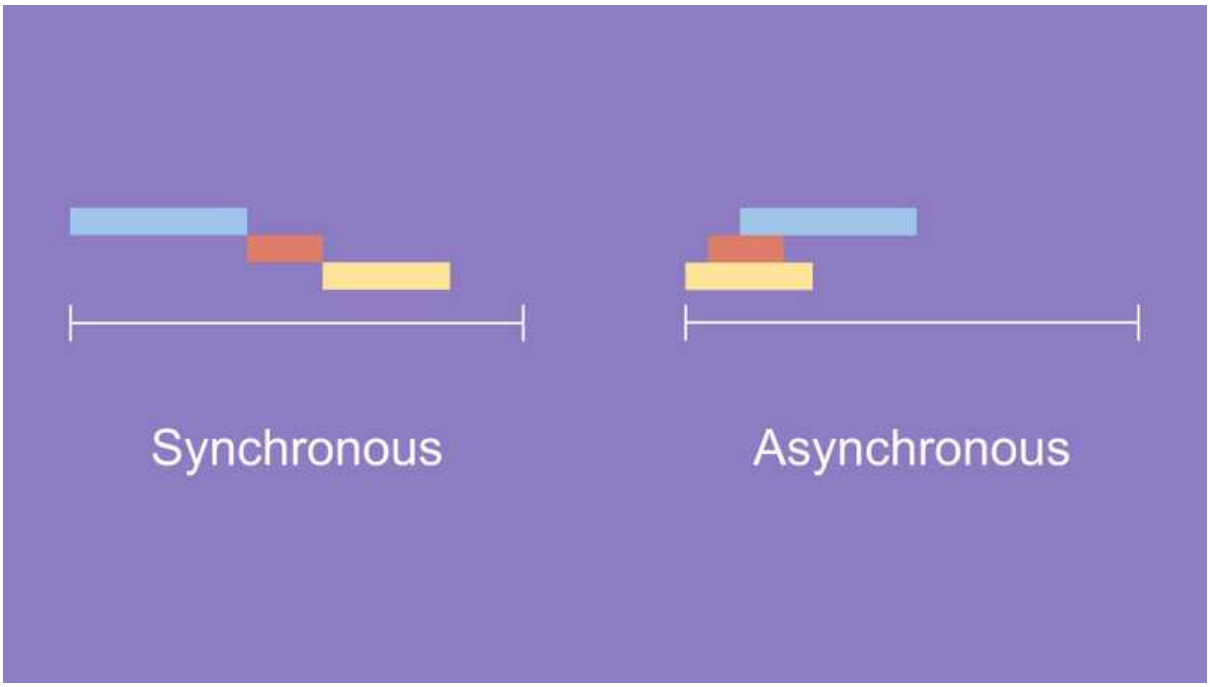
221219_2반_실습

JavaScript

동기(Synchronous)/비동기(*Asynchronous*)

동기(Synchronous)와 비동기(*Asynchronous*)란?



- 동기(Synchronous) : ‘동시에 일어나는’ 의미
  - **한 작업이 실행되는 동안 다른 작업은 멈춘 상태를 유지하고 자신의 차례를 기다리는 것**

    - **단일 스레드(싱글 스레드), 동기(Synchronous)**라고 부른다.
- 비동기(*Asynchronous*) : '동시에 일어나지 않는' 의미
    - **어떠한 요청을 보내면 그 요청이 끝날 때까지 기다리는 것이 아니라, 응답에 관계없이 바로 다음 동작이 실행되는 방식**

- 위 그림상에서 둘은 반대 개념인 것처럼 느껴진다. 여기서 쟁점은 **동시성의 발생을 어디로 보느냐**의 차이다.
- **동기는 요청의 결과가 그 자리에서 동시에 일어나야 하지만, 비동기는 그렇지 않다.**
- 따라서 동기적 주문의 경우 그 자리에서 커피를 받아야만 다음 작업을 처리할 수 있고,
- 비동기적 주문의 경우 진동벨을 받는 방식이라 한 번에 여러 주문을 처리할 수 있어 속도가 빨라지게 된다.

비동기 처리가 필요한 이유

- 비동기로 처리하는 방식은 효율성을 상승시켜 처리 속도를 높여준다.
- **서버와 통신할 때 가장 많은 시간이 소요되므로 네트워크 관련 작업들은 비동기적으로 구현되어 있다.**
- 자바스크립트 외적인 것, 예를 들어 DOM에 접근하는 메서드나 브라우저가 제공하는 웹 API 같은 것은 이미 비동기로 만들어져 있다.

- 예를들어, 웹 페이지가 로딩되거나, 어떠한 동작(Event) 하나가 30초 이상이 걸린다고 상상해보자. 그렇게 되면, 웹 페이지는 이 동작이 끝날 때까지 화면에 나타나지 않거나 다음 동작을 수행하는데 지장을 주게 된다. 또, 요즘 사용자들은 느리고 응답이 없는 웹 사이트를 원하지 않는다. 그렇기 때문에 자바스크립트가 웹 사이트에서 동작할 때, 비동기적으로 동작할 수 있어야 한다.

동기(Synchronous)/비동기(*Asynchronous*) 동작 원리

1. 동기(Synchronous) 동작 원리
    a. 코드가 실행되면 순서대로 Call Stack에 실행할 함수가 쌓인다.(push)
    b. 쌓인 반대 순서로 함수가 실행된다.(LIFO)
    c. 실행이 된 함수는 Call Stack에서 제거된다(pop)
2. 비동기(*Asynchronous*) 동작 원리
    a. Call Stack에서 비동기 함수가 호출되면 Call Stack에 먼저 쌓였다가 Web API(혹은 백그라운드라고도 한다)로 이동한 후 해당 함수가 등록되고 Call Stack에서 사라진다.
    b. Web API(백그라운드)에서 비동기 함수의 이벤트가 발생하면, 해당 콜백 함수는 Callback Queue에 push(이동) 된다.
    c. 이제 Call Stack이 비어있는지 이벤트 루프(Event Loop)가 확인을 하는데 만약 비어있으면, Call Stack에 Callback Queue에 있는 콜백 함수를 넘겨준다.(push)
    d. Call Stack에 들어온 함수는 실행이 되고 실행이 끝나면 Call Stack에서 사라진다.

JavaScript의 비동기 예제

- 기본적으로 JavaScript는 동기식 언어

1. 동기 예제

```
console.log("1번")
console.log("2번")
console.log("3번")
console.log("4번")
// 1번 -> 2번 -> 3번 -> 4번
```

2. 비동기 예제1

```
setTimeout(() => {console.log("1번")}, 5000);
setTimeout(() => {console.log("2번")}, 3000);
setTimeout(() => {console.log("3번")}, 1000);
```

```
console.log("4번")
// 4번->(1초)->3번->(2초)->2번->(2초)->1번
```

- 적힌 순서대로 1번 2번 3번 4번으로 될 것 같지만, **setTimeout은 비동기 함수이기 때문에 완전히 다른 결과가 나오게 된다.**
- setTimeout이 만약 동기적으로 처리됐다면 5초뒤 1번이, 3초뒤 2번이, 1초뒤 호출되어 총 8초가 걸렸겠지만, 비동기적으로 처리됐기 때문에 전체 걸린 시간은 5초가 된 것이다.

3. 비동기 예제 2

```
function hello(){
console.log('hello');
niceTo();
}

function niceTo(){
setTimeout(()=>{console.log('niceTo')}, 2000)
meetYou();
}

function meetYou(){
console.log('meetYou')
}
hello();
// hello meetYou niceTo
```

- 우리가 예상한 결과 : hello niceTo meetYou
    1. hello() 실행 → hello
    2. niceTo() 실행 → (2초 대기 후) niceTo
    3. meetYou() 실행 → meetYou
- 그러나 현실은 : hello meetYou niceTo
    1. hello() 실행 → hello //여기까지는 비슷
    2. niceTo() 실행 → (’niceTo’는 2초 대기 후 예약), meetYou();
    3. meetYou() 실행 → meetYou
    4. (2초 후) → niceTo
- 그동안 배웠던대로면 순서대로 함수가 실행되는 것이 맞는데, 출력된 결과값은 영문법이 맞지 않는 문장이다. 왜 그런것일까? 그 이유는 바로 setTimeout 메서드에

있다. 이 친구는 **자바스크립트 내장 함수가 아닌, 브라우저에서 제공하는 웹 API이자 비동기 함수이기 때문**이다.

- hello() 함수가 호출되면 해당 함수는 콜스택에 쌓인다. 이 함수가 niceTo()함수를 호출하므로 niceTo()도 콜스택에 쌓이지만 setTimeout의 콜백함수는 즉시 실행되지 않고, 지정 대기 시간만큼 기다린 후 이벤트 발생시 큐로 이동하여 콜스택이 비어졌을때가 되서야 다시 콜스택으로 이동되어 실행된다.

setTimeout()

- setTimeout() 메소드의 기능은 일정 시간이 지난 후 정해진 코드를 실행하게 합니다.
- 타임아웃 아이디(Timeout ID)라고 불리는 숫자를 반환한다.
- 문법

```
var timeoutID = setTimeout(function[, delay, arg1, arg2, ...]);
var timeoutID = setTimeout(function[, delay]);
var timeoutID = setTimeout(code[, delay]);
```

  - `function` : 타이머가 끝나고 실행할 함수
  - `code` : 함수 대신 문자열을 지정하는 대체 구문으로, 타이머가 끝나고 코드로 컴파일하여 실행. `eval()`의 보안 취약점과 같은 이유로 사용을 권장하지 않는다.
  - `delay` : 주어진 함수나 코드를 실행하기 전에 대기할 단위 시간(ms, milli seconds)

    💡 1s = 100ms

  - `arg1, ..., argN` : 함수에 전달할 매개변수
- 예제

```
setTimeout(() => {console.log("첫 번째 메시지")}, 5000);
setTimeout(() => {console.log("두 번째 메시지")}, 3000);
setTimeout(() => {console.log("세 번째 메시지")}, 1000);

// 콘솔 출력:

// 세 번째 메시지
// 두 번째 메시지
// 첫 번째 메시지
```

JavaScript에서 비동기 구현 방식

1. `Callback Function`
   - 특정함수에 매개변수로 전달된 함수를 의미합니다.즉, 나중에 호출될 함수들이라고 볼 수 있습니다.
   - 음식점을 예약하는 행위와 빗대어 설명이 가능하다.대기자가 있는 맛집을 찾아갔다고 가정해보자. 사람이 많아서 음식점을 예약하고 나면 실제 예약시간에 도달하기 전까지는 다양한 일들을 할 수 있다.(근처에서 쇼핑을 하거나 카페를 가거나 등등) 이 때 음식점으로부터 내 차례가 되었다고 전화가 옵니다. 이때를 콜백함수가 호출되는 시기라 볼 수 있습니다. 자리가 났다는 조건을 만족할때 저희는 밥을먹는 행위를 수행할 수 있으며, 조건이 만족되기 전까지는 자유롭게 다른 일들을 할 수 있습니다. 즉 데이터가 준비된 시점에서만 저희가 원하는 동작을 수행할 수 있습니다.
   - 예제

```
function Callback(callback){
    console.log('callback function');
    callback();
}
Callback(function(){
    console.log('callback 호출');
})
```

2. `Promise`
   - Promise 객체는 자바스크립트에서 비동기적인 프로그래밍을 위해 사용된다.
   - promise는 기본적으로 콜백과 하는일 동일하다.하지만 아래와 같이 .then()을 호출한다.이는 연속적으로 메소드들을 호출할 수 있다는 장점이 존재한다.따라서 콜백과 기능은 동일하지만 훨씬 간결하고 가독성이 좋아진다.
   - 예제

```
function getData() {
  return new Promise(function(resolve, reject) {
    reject(new Error("Request is failed"));
  });
}

// reject()의 결과 값 Error를 err에 받음
getData().then().catch(function(err) {
  console.log(err); // Error: Request is failed
});
```

3. `Async/Await`
    - 가장 최근 자바스크립트 문법에 추가된 비동기 처리 패턴입니다.기존의 콜백과 Promise의 단점을 보완하기위해 태어났다고 볼 수 있습니다.
    - 특히 가독성 부분을 집중적으로 보완한 문법입니다. 즉, 깔끔하게 Promise를 사용할수 있게 해주는 역할을 수행합니다.
    - Async와 Await을 사용하려면 우선 사용할 함수 앞에 async라는 키워드를 붙여 사용해야 하며 선언된 async 함수 안에서만 await 키워드를 사용할 수 있다.
    - 예제

```
async function 함수명() {
  await 비동기_처리_메서드_명();
}
```

```
function add10(a){
    return new Promise(resolve => setTimeout(() => resolve(a + 10),100));
}

async function f1() {
  const a = await add10(10);
  const b = await add10(a);
  console.log(a, b)
}
f1();
```

Callback Function

Callback Function 이란?

- 콜백함수는 함수를 활용하는 방법중 하나로 **파라미터로 전달받은 함수**를 말한다.
- 파라미터로 콜백함수를 전달받고 함수 내부에서 필요할 때 콜백함수를 호출할 수 있다.
- 간단하게 말하면 **함수 안에서 실행하는 또 다른 함수**이다.
- 파라미터로 변수가 아닌 함수를 전달하는 것을 말하며, 또한 함수이름 없이 익명으로도 전달 가능한 함수를 말한다.
- 콜백이 유용한 이유는, 콜백 함수만을 바꿔줌으로서 **하나의 함수를 여러가지로 응용**할 수 있기 때문이다.
- 하지만 콜백을 너무 남용하게 되면 우리가 흔히 부르는 콜백 지옥에 빠질 수가 있다.
- 또한 에러처리도 힘들 뿐더러 여러 개의 비동기 처리를 한번에 하는데 한계가 있다.

Callback Function 사용

1. 익명의 함수 사용

```
let number = [1, 2, 3, 4, 5];

number.forEach(function(x) {
    console.log(x * 2);
});
```

   - 콜백함수는 이름이 없는 '**익명의 함수**'를 사용한다. 함수의 내부에서 실행되기 때문에 이름을 붙이지 않아도 된다.

2. 함수의 이름(만) 넘기기

```
function whatYourName(name, callback) {
    console.log('name: ', name);
    callback();
}

function finishFunc() {
    console.log('finish function');
}

whatYourName('miniddo', finishFunc);

/*
name: miniddo
finish function
*/
```

   - 자바스크립트는 null과 undefined 타입을 제외하고 모든 것을 객체로 다룬다.
   - 함수를 변수 or 다른 함수의 변수처럼 사용할 수 있다.
   - 함수를 콜백함수로 사용할 경우, 함수의 이름만 넘겨주면 된다.
   - 위의 예제에서, 함수를 인자로 사용할 때 callback, finishFunc 처럼 () 를 붙일 필요가 없다는 것이다.

3. 전역변수, 지역변수를 콜백함수의 파라미터로 전달

```
let fruit = 'apple';  // Global Variable

function callbackFunc(callback) {
    let vegetable = 'tomato'; // Local Variable
    callback(vegetable);
}
```

```
function eat(vegetable) {
    console.log(`fruit: ${fruit} / vegetable: ${vegetable}`);
}

callbackFunc(eat);

// fruit: apple / vegetable: tomato
```

Callback Function 예제

1. 예제1
    - 일반 함수로 사용할 경우

```
function add(a, b) {
  return a + b;
}

function sayResult(value) {
  console.log(value);
}

sayResult(add(3, 4));
```

    - callback 함수로 사용할 경우

```
function add(a, b, callback) {
  callback(a + b);
}

function sayResult(value) {
  console.log(value);
}

add(3, 4, sayResult);
```

2. 예제2
    - 비동기 처리

```
function callBackTestFunc(callback) {
    setTimeout(callback, 1000)
    console.log('Hello')
}

callBackTestFunc(() => {
    console.log('waited 1 second')
})
```

```
//결과
//Hello
//waited 1 second
```

3. 예제3

```
function introduce (lastName, firstName, callback) {
    var fullName = lastName + firstName;

    callback(fullName);
}

function say_hello (name) {
    console.log("안녕하세요 제 이름은 " + name + "입니다");
}

function say_bye (name) {
    console.log("지금까지 " + name + "이었습니다. 안녕히계세요");
}

introduce("홍", "길동", say_hello);
// 결과 -> 안녕하세요 제 이름은 홍길동입니다

introduce("홍", "길동", say_bye);
// 결과 -> 지금까지 홍길동이었습니다. 안녕히계세요
```

- 위와 같이 다른 동작을 수행하는 함수 say_hello 와 say_bye 를 정의해두고 introduce 함수에 인풋으로 사용해주면, introduce라는 함수에서 받아들이는 같은 인풋들을 가지고 다른 동작을 수행하는 것이 가능해진다. 즉, 이런식으로 **함수를 나눠줌으로써** 코드를 재활용 할 수 있고, 관리도 더 쉬워지게 되는 것이다.

Callback Hell

콜백 지옥은 JavaScript를 이용한 비동기 프로그래밍시 발생하는 문제로서, 함수의 매개 변수로 넘겨지는 콜백 함수가 반복되어 코드의 들여쓰기 수준이 감당하기 힘들 정도로 깊어지는 현상을 말한다.

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

```
            });
        });
    });
});
```

- 예제
    - callback hell

```
class UserStorage{
    loginUser(id, password, onSuccess, onError){
      setTimeout(() =>{
        if(
            (id === 'seul' && password === '123') ||
            (id === 'kim' && password === '456')
         ) {
            onSuccess(id);
         } else{
            onError(new Error('error'));
         }
        }, 2000);
    }

    getRoles(user, onSuccess, onError){
        setTimeout(()=> {
            if (user === 'seul') {
                onSuccess({name: 'seul', role: 'admin'});
            } else {
                onError(new Error('error'));
            }
        }, 1000);
    }
};

const userStorage = new UserStorage();
const id = prompt('아이디를 입력해 주세요!');
const password = prompt('비밀번호를 입력해 주세요!!');

userStorage.loginUser(
    id,
    password,
    user => {
        userStorage.getRoles(
            user,
            userWithRole => {
                alert(`hello ${userWithRole.name}, you have a ${userWithRole.role}
role`)
            },
            error => {
                console.log('에러2')
            }
            );
        },
    error => {console.log('에러1')}
);
```

- promise

```
class UserStorage{
    loginUser(id, password){
        return new Promise((resolve, reject) => {
           setTimeout(() =>{
               if(
                   (id === 'seul' && password === '123') ||
                   (id === 'kim' && password === '456')
                 ) {
                    resolve(id);
                 } else{
                    reject(new Error('에러1'));
                 }
           }, 2000);
       })
    }

    getRoles(user){
       return new Promise((resolve, reject) => {
          setTimeout(()=> {
              if (user === 'seul') {
                resolve({name: 'seul', role: 'admin'});
              } else {
                reject(new Error('에러2'));
              }
           }, 1000);
       })
    }
};

const userStorage = new UserStorage();
const id = prompt('아이디를 입력해 주세요!');
const password = prompt('비밀번호를 입력해 주세요!!');

userStorage.loginUser(id, password)
.then(userStorage.getRoles)
// .then(user => userStorage.getRoles(user)); 인자가 똑같으니 생략 가능하다.
.then(user => alert(`hello ${user.name}, you have a ${user.role} role`))
.catch(console.log);
```

- sync/await

```
class UserStorage{
    loginUser(id, password){
        return new Promise((resolve, reject) => {
            setTimeout(() => {
                if(
                    (id === 'seul' && password === '123') ||
                    (id === 'kim' && password === '456')
                 ) {
```

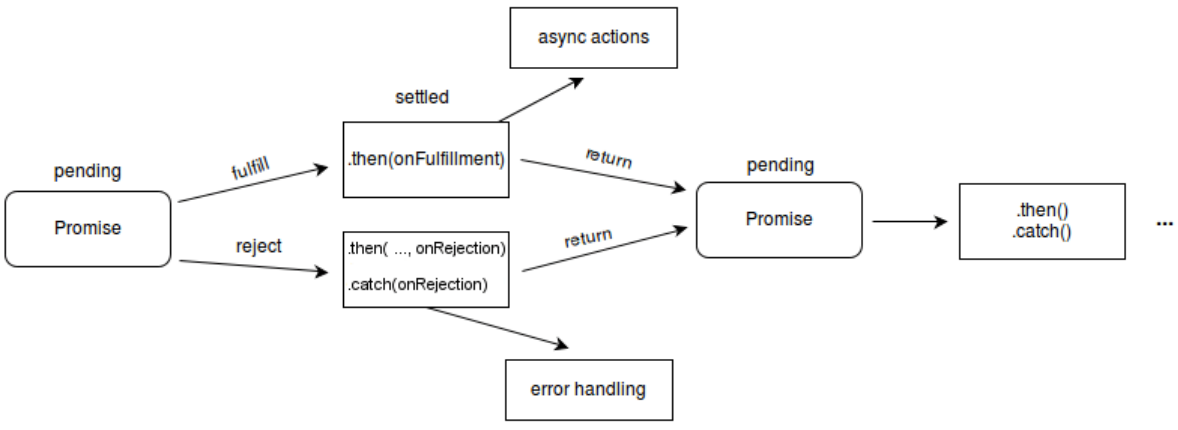
```
                    resolve(id);
                } else{
                    reject(new Error('에러1'));
                }
            }, 2000);
        })
    }
    getRoles(user){
        return new Promise((resolve, reject) => {
            setTimeout(()=> {
                if (user === 'seul') {
                    resolve({name: 'seul', role: 'admin'});
                } else {
                    reject(new Error('에러2'));
                }
            }, 1000);
        })
    }
};

const userStorage = new UserStorage();
const id = prompt("아이디를 입력해 주세요!");
const password = prompt("비밀번호를 입력해 주세요!!");

async function checkUser() {
  try {
    const userId = await userStorage.loginUser(id, password);
    const user = await userStorage.getRoles(userId);
    alert(`Hello ${user.name}, you have a ${user.role}`);
  } catch (error) {
    console.log(error);
  }
}
checkUser();
```

Promise

Promise 개요



1. Promise 란?
    - 콜백 함수의 단점을 보완하며 비동기 처리에 사용되는 객체를 **프로미스(Promise)** 라 한다.
    - `Promise` 객체는 비동기 작업이 맞이할 미래의 완료 또는 실패와 그 결과 값을 나타냅니다.
    - 프로미스는 객체이기 때문에 생성자 함수를 호출하여 인스턴스화할 수 있다.
    - 프로미스의 생성자 함수는 `resovle`와 `reject`함수를 인자로 전달받는 콜백함수를 인자로 전달받습니다.
    - 프로미스는 인자로 전달받은 콜백 함수를 내부에서 비동기 처리 합니다.
    - 프라미스 내부에서 **성공을 하면** `resolve`를 호출하고 **실패하면** `reject`를 호출

    > 💡 `Promise`는 프로미스가 생성된 시점에는 알려지지 않았을 수도 있는 값을 위한 대리자로, 비동기 연산이 종료된 이후에 결과 값과 실패 사유를 처리하기 위한 처리기를 연결할 수 있습니다. 프로미스를 사용하면 비동기 메서드에서 마치 동기 메서드처럼 값을 반환할 수 있습니다. 다만 최종 결과를 반환하는 것이 아니고, 미래의 어떤 시점에 결과를 제공하겠다는 '약속'(프로미스)을 반환합니다.

    - Promise의 상태
        - 대기(*pending*): 이행하지도, 거부하지도 않은 초기 상태. - 작업 시작 전(보류 됨)
        - 이행(*fulfilled*): 연산이 성공적으로 완료됨. - 작업 성공(처리 완료)
        - 거부(*rejected*): 연산이 실패함. - 작업 실패(거부 됨)
        - 완료(*settled*) : 처리 완료 혹은 거부 됨 - (해결 됨)
2. Promise 장점
    - 비동기 처리 시점을 명확하게 표현할 수 있다.
    - 연속된 비동기 처리 작업을 수정, 삭제, 추가하기 편하고 유연하다.
    - 비동기 작업 상태를 쉽게 확인할 수 있다.
    - 코드의 유지 보수성이 증가한다

Promise 사용

1. Promise vs Callback Function
    - Callback Function

```
var loading = function(path, done) {
  console.log("전달받은 경로 : ", path);
  done(path + "sample.txt");
}

loading('/folder/texrt/', function(result){
  console.log("완료! : ", result);
})
```

    - Promise

```
var loading = function(path) {
  return new Promise(function(resolve, reject) {
      console.log('전달받은 경로 : ', path);
      resolve(path + 'sample.txt');
  })
}

loading('/folder/text/').then(function(result){
    console.log(result);
}).catch(function(error)) {
    console.log('error', error);
}
```

    - 코드의 형태는 변화되었지만, 기능은 동일
2. Promise 객체

```
new Promise(executor)

let promise = new Promise(function(resolve, reject) {
  // executor (성공, 실패)
});
```

    - `resolve`와 `reject`는 자바스크립트에서 자체 제공하는 콜백함수이다
    - Promise 는 클래스 문법과 같이 `new` 키워드와 생성자를 사용해 만든다.
    - 생성자는 매개변수로 executor라는 실행함수를 받는다.
    - executor의 인자로 `resolve` , `reject` 로 받으며 이 두개도 각 함수이다.
    - Promise를 즉시 실행할 수 도있고 `then` 으로 호출하여 필요시에 사용할 수 도있다.

3. Promise()
    - 새로운 `Promise` 객체를 생성, 주로 프로미스를 지원하지 않는 함수를 감쌀 때 사용한다.
4. then()
    - then 메소드는 두 개의 콜백 함수를 인자로 전달 받습니다.
    - 첫 번째 콜백 함수는 성공(fulfilled, resolve 함수가 호출된 경우)시에 실행됩니다.
    - 두 번째 콜백 함수는 실패(rejected, reject 함수가 호출된 경우)시에 실행됩니다.
    - then 메소드는 기본적으로 프로미스를 반환합니다.
5. catch()
    - catch 메소드는 비동기 처리 혹은 then 메소드 실행 중 발생한 에러(예외)가 발생하면 호출됩니다.
    - catch 메소드 역시 프로미스를 반환합니다.

Promise 예제

1. 예제 1

```
const promise = () => new Promise((resolve, reject) => {
    let a = 1 + 1

    if(a == 2) {
        resolve('success')
    } else {
        reject('failed')
    }
})

promise().then((message) => {
    console.log('This is in the then ' + message)
}).catch((message) => {
    console.log('This is in the catch' + message)
})
```