📖

4주차 (4/25)

1번 실습 (콜백 함수)

콜백 함수란?

콜백 함수는 파라미터로 함수를 전달받아, 함수의 내부에서 실행하는 함수입니다.

콜백 함수는 이미 우리가 자바스크립트를 배우면서 많이 써봤기 때문에 금방 이해가 갈겁니다.

```
let names = ["Daniel", "John", "Doe"];

names.forEach(function(x){
  console.log(x)
});

/* output
Daniel
John
Doe
*/
```

위의 예시를 보면 forEach 함수 안에 함수를 전달해 forEach문에 작동하는 걸 볼 수 있습니다.

setTimeout() {}

```
setTimeout(() => {console.log("5초후에 넌 죽는다")}, 5000);
```

`setTimeout(함수, 시간)` 메서드는 뒤에 특정 시간이 만료된 후 함수나 지정한 코드 조각을 실행하는 타이머를 설정합니다. setTimeout은 비동기 함수이기 때문에 실행 순서와 상관없이 지정한 시간이 만료될 때 결과값을 반환합니다.

▼ 실습 정답 코드

```
// 지시사항에 따라 코드를 작성합니다.
function countDownThree() {
  console.log(3)
  setTimeout(() => console.log(2), 1000)
  setTimeout(() => console.log(1), 2000)
  setTimeout(() => console.log("끝"), 3000)
};
```

2번 실습 (alert)

Alert()

`Window.alert()` 메서드는 확인 버튼을 가지며 메시지를 지정할 수 있는 경고 대화 상자를 띄웁니다.

```
alert("Hello world!");
```

setTimeout의 안좋은 예시

```
// 안 좋은 예 1
setTimeout(alert("문구"), 1000)

// 좋은 예 1
setTimeout(() => alert("문구"), 1000)

// 안 좋은 예 2
setTimeout(myFunc(), 1000)

// 좋은 예 2
setTimeout(myFunc, 1000)
```

💡 `e.preventDefault()` 해주는걸 잊지 마세요!

▼ 실습 정답 코드

```
// 자유롭게 코드를 작성하여, 예시 화면이 구현되도록 해 보세요.
function alertAfter1Second(e) {
  // 새로고침 방지 (form의 버튼은 default 행위가 새로고침인데, 이 default 행위를 막아야 함)

  e.preventDefault()

  const name = nameElem.value
  setTimeout(() => alert(`입력된 이름: ${name}`), 2000)
}

const nameElem = document.querySelector('#inputName')
const buttonElem = document.querySelector('#buttonSubmit')

buttonElem.addEventListener("click", alertAfter1Second)
```

3번 실습 (Debouncing)

> 연속으로 호출되는 함수들 중에 마지막에 호출되는 함수(또는 제일 처음 함수)만 실행되도록 하는 것

예를 들어 아래의 코드가 있다고 가정해 봅시다.

```
document.querySelector('input').addEventListener('input', function(e) {
    console.log('입력 내용: ', e.target.value);
});
```

위 코드는 우리가 input 창에 타이핑을 칠때마다 요청이 실행됩니다.

지금은 별 문제가 안될 수 있지만 만약 우리가 유료 API를 사용한다고 가정하면 API 사용 비용이 만만치 않을 겁니다.

우리가 이때 사용할 수 있는 방법이 바로 Debouncing 입니다.

```
let timer;

document.querySelector('#input').addEventListener('input', function(e) {
    if(timer) {
        clearTimeout(timer); -> timer의 setTimeout을 취소시킴
    }
    timer = setTimeout(function() {
        // 실행 코드 내용
    }, 200);
});
```

위 코드를 설명해보자면 이벤트가 발생 할 때마다 이전에 설정해둔 내용을 clear시키고 다시 새로 timer를 설정해줍니다. 설정해둔 시간내에 이벤트가 다시 발생하지 않으면 이벤트 발생이 끝난 것으로 판단하고 실행할 코드가 실행됩니다.

Throttling

위의 개념과 반대되는 개념이 쓰로틀링입니다. 쓰로틀링은 설정한 특정 시간 주기로 계속 실행이 됩니다. 보통 스크롤 이벤트에 많이 사용됩니다.

```
let timer;
document.querySelector('.body').addEventListener('scroll', function (e) {
  if (!timer) {
    timer = setTimeout(function() {
      timer = null;
      // 실행할 코드 내용
    }, 200);
  }
});
```

▼ 실습 정답 코드

```
// 자유롭게 코드를 작성하여, 예시 화면이 구현되도록 해 보세요.
let alertTimer;

const nameElem = document.querySelector("#inputName");

nameElem.addEventListener("input", () => {
  if (alertTimer) {
    clearTimeout(alertTimer);
  }

  const name = nameElem.value;
  alertTimer = setTimeout(() => alert(`입력된 이름: ${name}`), 1000);
});
```

4번 실습 (promise)

Promise

`Promise` 객체는 비동기 작업이 맞이할 미래의 완료 또는 실패와 그 결과 값을 나타냅니다. `Promise`는 보통 서버에서 데이터를 받아올 때 사용합니다. 서버에서 데이터를 요청했는데 오기도 전에 데이터를 사용하려고 하면 에러가 뜰테니, 이를 방지하기 위해 `Promise`가 사용됩니다.

💡 **비동기란?: 순서에 상관없이 동시에 코드가 실행되는 상태**

A작업이 실행될 때, B 작업도 동시에 시작되며, 각 작업이 끝나는대로 결과값을 출력한다.

(a) 동기

(b) 비동기

다음의 대화를 자바스크립트로 구현하고 싶다고 가정합시다.

```
야 트와이스에 모모 예쁘지 않냐?
// 2초후
모모? 예쁘지.
// 2초후
그니깐. 어제 봤는데 실물도 이쁘더라.
```

만약 아래의 코드를 돌려보면 어떻게 나올까요?

```
function function1() {
  setTimeout(() => {
    console.log("야 트와이스에 모모 예쁘지 않냐?");
  }, 2000);
}

function function2() {
  setTimeout(() => {
    console.log("모모? 예쁘지.");
  }, 2000);
}

function function3() {
  setTimeout(() => {
    console.log("그니깐. 어제 봤는데 실물도 이쁘더라.");
  }, 2000);
}

function1();
function2();
function3();
```

한꺼번에 코드가 실행되는 걸 볼 수 있습니다. 이건 setTimeout 함수가 비동기 함수이기 때문에 그렇습니다. 각 함수가 끝날때까지 기다리는것이 아니기 때문에 결과값이 2초후 한꺼번에 출력되는 겁니다.

아래는 Promise를 이용한 코드입니다. 한번 돌려볼까요?

```
function Promise1() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
```

```
      console.log("야 트와이스 모모 이쁘지 않냐?");
      resolve();
    }, 2000);
  });
}

function Promise2() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log("모모? 이쁘지.");
      resolve();
    }, 2000);
  });
}

function Promise3() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log("그니깐. 어제 봤는데 실물도 이쁘더라.");
      resolve();
    }, 2000);
  });
}

const result = Promise1().then(Promise2).then(Promise3);
```

정상적으로 나오는 걸 볼 수 있습니다.

이제 이 코드가 돌아가는 원리를 한번 알아봅시다. 아래는 `Promise`가 적용된 코드의 기본적인 구조입니다.

```
function getData(callback) {
  // new Promise() 추가
  return new Promise(function(resolve, reject) {
    //데이터를 받아오는 함수
    함수('url 주소', function(response) {
      // 데이터를 받으면 resolve() 호출
      if (data == 호출을 성공하면) {
        resolve(response);
      } else {
        reject(new Error("오류가 발생했습니다"));
      }
    });
  });
}

getData().then(function() {
  renderData()
}).catch((err) => console.log(err))
```

일단 코드는 나중에 살펴보도록 하고 프로미스의 3가지 상태에 대해서 알아봅시다.

프로미스의 3가지 상태

프로미스는 대략 3가지 상태, 좀 더 쉽게 이야기하자면 처리 과정을 가집니다.

- Pending (대기) → 비동기 처리 로직이 완료되지 않았을 때
- Fulfilled (완료) → 비동기 처리가 결과 값을 반환해 주었을 때
- Rejected (실패) → 비동기 처리가 실패했거나 오류가 발생했을 때

Pending

우리가 `new Promise()` 메서드를 호출하면 기본적으로 `Pending` 상태가 됩니다.

`new Promise()` 메서드를 호출할 때 콜백 함수에 인자 `resolve`와 `reject`를 기본적으로 넣어줍니다.

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

Fullfilled

우리가 위에서 인자로 보내줬던 `resolve`가 아래와 같이 실행되면 Fullfilled 상태가 됩니다.

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

여기서 더 나아가 `resolve` 안에 우리가 `return` 할 값을 넣어주면 나중에 `then()`을 이용해서 값을 다시 받을 수 있습니다.

```
new Promise((resolve, reject) => {
  let data = ["Daniel", "John", "Doe"];
  resolve(data);
}).then(resolvedData => resolvedData.forEach(x => console.log(x)))
```

Rejected

new Promise()로 만약에 `reject`를 호출하면 Rejected 상태가 됩니다. catch()를 이용하면 오류를 출력 할 수 있습니다.

```
new Promise((resolve, reject) => {
  reject(new Error("Request is failed"));
}).then().catch((err) => console.log(err));
```

Promise Chaining

프로미스의 재미있는 점은 여러개의 프로미스를 연결해서 쓸 수 있다는 점입니다.

```
new Promise(function(resolve, reject){
  setTimeout(function() {
    resolve(1);
  }, 2000);
})
.then(function(result) {
  console.log(result); // 1
  return result + 10;
})
.then(function(result) {
  console.log(result); // 11
  return result + 20;
})
.then(function(result) {
  console.log(result); // 31
});
```

Promise Chaining은 실제 웹 서비스에서 어떻게 사용되냐면:

```
getData(userInfo)
  .then(parseValue)
  .then(setData(data))
  .then(display);
```

▼ 실슬 정답 코드

```
// 지시사항에 따라 코드를 작성합니다.
function resolveWhenPositiveNumber(n) {
  return new Promise((resolve, reject) => {
    if (n > 0) {
      resolve(`${n}은 양수입니다!`)
    } else {
      reject(`${n}은 음수입니다...`)
    }
  })
}
```

5번 실습

fetch

`fetch()` 함수는 첫번째 인자로 URL, 두번째 인자로 옵션 객체를 받고, Promise 타입의 객체를 반환합니다. 반환된 객체는, API 호출이 성공했을 경우에는 응답(response) 객체를 resolve하고, 실패했을 경우에는 예외(error) 객체를 reject합니다.

```
fetch(데이터요청 할 서버 url, 아니면 json 파일)
  // 데이터를 json 객체로 바꾸고 resolve 해줌
  .then(response => response.json())
  .then(data => {
    //데이터 처리 부분
  }
```

💡 `e.preventDefault()` 해주는걸 잊지 마세요!

💡 Hint

```
// 자유롭게 코드를 작성하여, 예시 화면이 구현되도록 해 보세요.

function showHexaCode(e) {
  // 새로고침 방지

  e.preventDefault()
  // 유저 입력값 가져오기

  // fetch -> response.json() 하는 과정은 고정입니다 (번거롭지만, 매번 해 주어야 함).
  // fetch를 활용하여 data.json에서 데이터를 가져온다
    //then response를 .json()을 이용하여 JSON 데이터를 js 객체로 변환해준다
    //then datas를 이용하여
      // find() 메서드를 이용하여 배열의 요소 중, color 가, 사용자가 입력한 색과 일치하는 요소를 찾고 foundData 변수에 저장.
      //hexaCodeElem innerHTML을 이용하여 foundData의 value를 넣어준다

}


const inputElem = document.querySelector('#inputColor')
const buttonElem = document.querySelector('#buttonSubmit')
const hexaCodeElem = document.querySelector('#hexaCode')

buttonElem.addEventListener("click", showHexaCode)
```

▼ 실습 정답 코드

```
// 자유롭게 코드를 작성하여, 예시 화면이 구현되도록 해 보세요.

function showHexaCode(e) {
  // 새로고침 방지

  e.preventDefault()

  const userInputColor = inputElem.value

  // fetch -> response.json() 하는 과정은 고정입니다 (번거롭지만, 매번 해 주어야 함).
  // 물론 response 대신 res 등으로 변수명은 바꾸어도 됩니다.
  fetch('data.json')
    .then(res => res.json())
    .then(datas => {
      // 배열의 요소 중, color 가, 사용자가 입력한 색과 일치하는 요소를 찾음.
      const foundData = datas.find(data => data.color === userInputColor)

      // 찾은 요소는 객체인데, 그 value값이 헥사코드임.
      hexaCodeElem.innerHTML = foundData.value

      // 참고로, 아래 코드를 추가하면 색상도 반영됨 (채점과는 무관합니다).
      hexaCodeElem.style.color = foundData.value
    })

}
```

```
const inputElem = document.querySelector('#inputColor')
const buttonElem = document.querySelector('#buttonSubmit')
const hexaCodeElem = document.querySelector('#hexaCode')

buttonElem.addEventListener("click", showHexaCode)
```

6번 실습

💡 Hint

```
//posts 변수를 수정하지 마세요.
const posts = [
  { title: "Post 1", body: "첫번째 게시글입니다." },
  { title: "Post 2", body: "두번째 게시글입니다." },
  { title: "Post 3", body: "세번째 게시글입니다." },
  { title: "Post 4", body: "네번째 게시글입니다." },
  { title: "Post 5", body: "다섯번째 게시글입니다." },
];

//getPosts() 함수를 작성하세요.

function getPosts() {
  //setTimeout()를 사용해서 1초 후에 posts element를 rendering 합니다.
  setTimeout(() => {
    let output = "";

    //위에 정의된 posts 내의 게시글 제목과 내용을 forEach()을 사용해서 rendering 합니다.

    //rendering 된 게시글을 document.body.innerHTML을 사용해서 html에 띄어줍니다.

  }, 1000);
}

//createPost() 함수를 작성하세요.
function createPost(post) {
  //Promise를 생성해서 resolve와 reject를 인자로 받습니다.

    //Promise 내에 setTimeout으로 비동기 처리하는데,
        //createPost()함수에 인자로 받아온 post를 push할 때 에러없이 성공적으로 호출되면(if(!error)) `resolve`를 실행하고, 그렇지 않으면 에러를 받아들이는 `


}

//createPost()를 이용해 데이터를 추가해보세요.
//title은 "Post N", body는 "N번째 게시글입니다."로 설정하세요.
    //then getPosts
    //catch err
```

▼ 실습 정답 코드

```
//posts 변수를 수정하지 마세요.
const posts = [
  { title: "Post 1", body: "첫번째 게시글입니다." },
  { title: "Post 2", body: "두번째 게시글입니다." },
  { title: "Post 3", body: "세번째 게시글입니다." },
  { title: "Post 4", body: "네번째 게시글입니다." },
  { title: "Post 5", body: "다섯번째 게시글입니다." },
];

function getPosts() {
  setTimeout(() => {
    let output = "";
    posts.forEach((post, index) => {
      output = output + `<li>${post.title}<br> 내용: ${post.body} </li> `;
    });
    document.body.innerHTML = output;
    //postMessage.forEach((post, resolve))
  }, 1000);
```

```
}

function createPost(post) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      posts.push(post);
      const error = false;
      if (!error) {
        resolve();
      } else {
        reject("error: wrong");
      }
    }, 2000);
  });
}

createPost({ title: "Post N", body: "N번째 게시글입니다." })
  .then(getPosts)
  .catch((err) => console.log(err));
```