코어 자바스크립트

- 프라미스와 async, await
- 모듈

콜백: 비동기 처리

프라미스와 async, await

- setTimeout이 스케줄링에 사용되는 대표적 비동기 함수
- 스크립트, 모듈 로딩 등등 여러 비동기 동작이 존재

```
function loadScript(src) {
  // <script> 태그를 만들고 페이지에 태그를 추가합니다.
  // 태그가 페이지에 추가되면 src에 있는 스크립트를 로딩하고 실행합니다.
  let script = document.createElement('script');
  script.src = src;
  document.head.append(script);
}

// 해당 경로에 위치한 스크립트를 불러오고 실행함
loadScript('/my/script.js');

// loadScript 아래의 코드는
// 스크립트 로딩이 끝날 때까지 기다리지 않습니다.
// ...
```

콜백: 비동기 처리

프라미스와 async, await

- 스크립트를 읽어올 시간이 확보되지 않아 에러 발생

```
loadScript('/my/script.js'); // script.js엔 "function newFunction() {…}"이 있습니다.

newFunction(); // 함수가 존재하지 않는다는 에러가 발생합니다!
```

콜백: 비동기 처리

프라미스와 async, await

- 로딩이 끝난 후 실행되는 콜백 함수 이용

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(script);

  document.head.append(script);
}
```

```
loadScript('/my/script.js', function() {
  // 콜백 함수는 스크립트 로드가 끝나면 실행됩니다.
  newFunction(); // 이제 함수 호출이 제대로 동작합니다.
  ...
});
```

콜백: 에러 핸들링

프라미스와 async, await

- 로딩이 실패하는 경우 에러를 추적 & 핸들링

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`${src}를 불러오는 도중에 에러가 발생했습니

  document.head.append(script);
}
```

```
loadScript('/my/script.js', function(error, script) {
  if (error) {
    // 에러 처리
  } else {
    // 스크립트 로딩이 성공적으로 끝남
  }
});
```

콜백: 중첩 콜백

프라미스와 async, await

- 여러 스크립트 로딩이 필요한 경우 콜백을 중첩으로 구성
- 꼬리에 꼬리를 무는 비동기 동작일 경우 중첩 방식의 코딩은 좋지 않음

```
loadScript('1.js', function(error, script) {

  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', function(error, script) {
      if (error) {
        handleError(error);
      } else {
        // ...
        loadScript('3.js', function(error, script) {
          if (error) {
            handleError(error);
          } else {
            // 모든 스크립트가 로딩된 후, 실행 흐름이 이어집니다. (*)
```

콜백: 중첩 콜백

프라미스와 async, await

- 각 함수들을 독립적으로 코딩할 수 있지만, 실제 작동은 의존적이고 코드 가독성이 나쁨

```
loadScript('1.js', step1);

function step1(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', step2);
  }
}

function step2(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('3.js', step3);
  }
}

function step3(error, script) {
  if (error) {
```

프라미스: 기본 개념

프라미스와 async, await

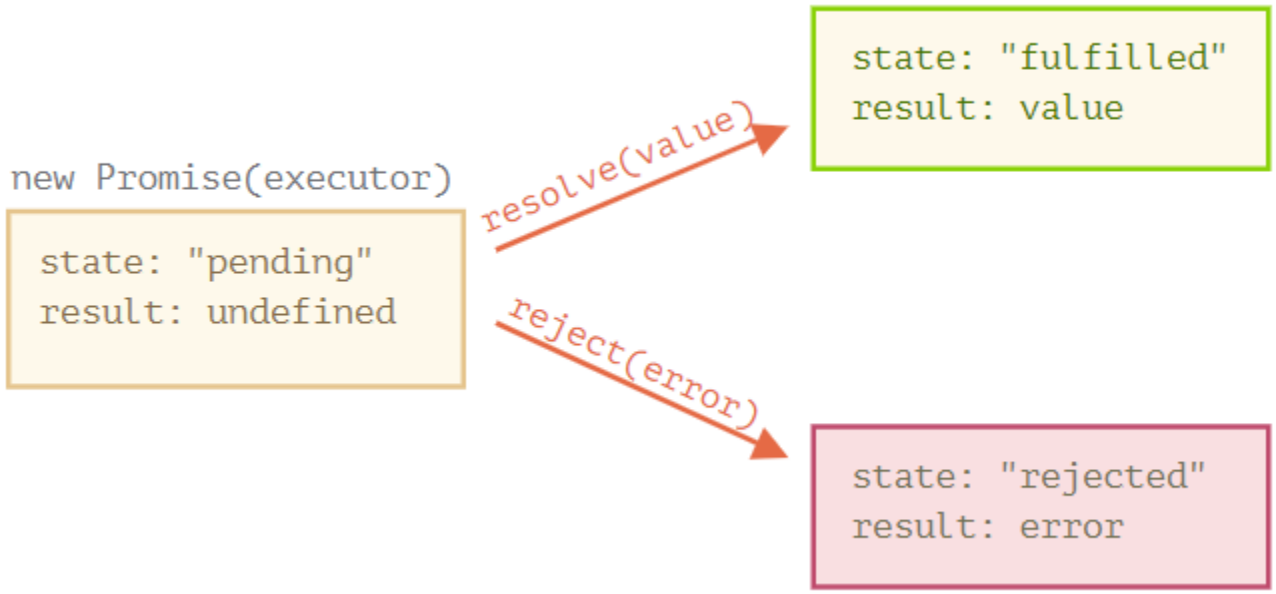
```
let promise = new Promise(function(resolve, reject) {
  // executor (제작 코드, '가수')
});
```

- new Promise에 전달되는 함수: executor(실행자, 실행함수)
- executor는 new Promise가 만들어질 때 자동 실행
- resolve와 reject는 자바스크립트가 자체적으로 제공하는 콜백
- Promise 구현 시 resolve, reject 콜백 중 하나는 호출해야 함
- resolve(value) / reject(error)
- Promise 내부 property: state와 result

프라미스: 기본 개념

프라미스와 async, await

- Promise 내부 property: state와 result
- executor에 의해 변경된 상태는 더 이상 변하지 않음
- 처리가 끝난 Promise에 resolve / reject를 호출해도 무시됨



프라미스: 메서드

프라미스와 async, await

- .then의 첫 번째 인수는 Promise가 이행되었을 때 실행
- .then의 두 번째 인수는 Promise가 거부되었을 때 실행

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("done!"), 1000);
});

// resolve 함수는 .then의 첫 번째 함수(인수)를 실행합니다.
promise.then(
  result => alert(result), // 1초 후 "done!"을 출력
  error => alert(error) // 실행되지 않음
);
```

프라미스: 메서드

프라미스와 async, await

- 작업이 성공적으로 처리된 경우만을 다루고 싶은 경우

```
let promise = new Promise(resolve => {
  setTimeout(() => resolve("done!"), 1000);
});

promise.then(alert); // 1초 뒤 "done!" 출력
```

- 에러가 발생한 경우만을 다루고 싶은 경우

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("에러 발생!")), 1000);
});

// .catch(f)는 promise.then(null, f)과 동일하게 작동합니다
promise.catch(alert); // 1초 뒤 "Error: 에러 발생!" 출력
```

프라미스: 메서드

프라미스와 async, await

- 결과가 무엇이든 마무리가 필요한 경우
- finally 핸들러에는 인수가 없으므로 Promise의 상태를 확인할 수 없음
- 자동으로 다음 핸들러에 결과와 에러를 전달함

```
new Promise((resolve, reject) => {
  setTimeout(() => resolve("결과"), 2000)
})
  .finally(() => alert("프라미스가 준비되었습니다."))
  .then(result => alert(result)); // <-- .then에서 result를 다룰 수 있음
```

프라미스 체이닝

프라미스와 async, await

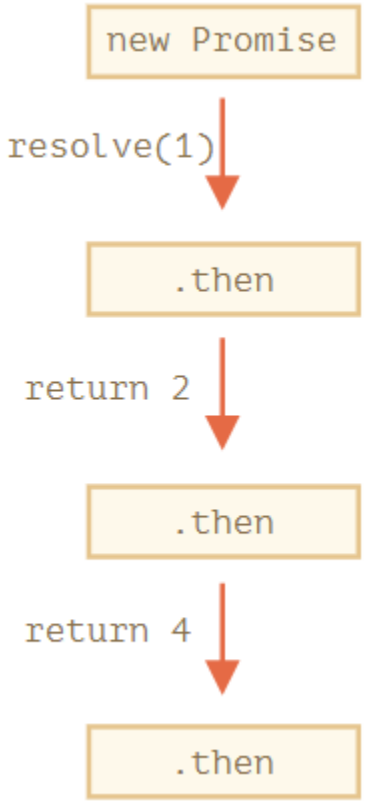
- 순차적으로 실행해야 하는 비동기 작업 처리

```
new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000); // (*)

}).then(function(result) { // (**)

  alert(result); // 1
  return result * 2;

}).then(function(result) { // (***)

  alert(result); // 2
  return result * 2;

}).then(function(result) {

  alert(result); // 4
  return result * 2;

});
```

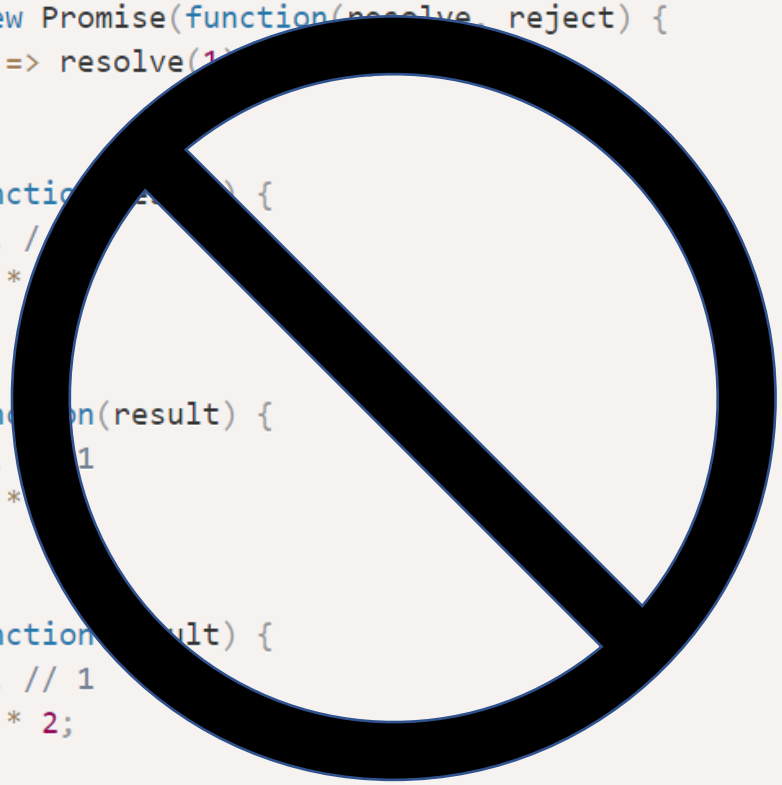
프라미스 체이닝

프라미스와 async, await

- promise.then을 호출하면 Promise가 호출되기 때문에 가능
- 핸들러가 값을 return할 땐 이 값이 Promise의 result가 되어 다음 핸들러가 이 값을 이용해 호출됨




```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve(1
});

promise.then(functi
  alert(result); /
  return result *
});

promise.then(fun
  alert(result);
  return result *
});

promise.then(functio
  alert(result); // 1
  return result * 2;
});
```



프라미스 체이닝

프라미스와 async, await

- 핸들러가 Promise를 생성하여 반환하는 경우도 가능

```
new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000);

}).then(function(result) {

  alert(result); // 1

  return new Promise((resolve, reject) => { // (*)
    setTimeout(() => resolve(result * 2), 1000);
  });

}).then(function(result) { // (**)

  alert(result); // 2

  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(result * 2), 1000);
  });

}).then(function(result) {
```

프라미스 체이닝: thenable

- 핸들러는 thenable라는 객체를 반환하기도 함
- .then을 가진 객체는 모두 thenable 객체라고 부름

```
class Thenable {
  constructor(num) {
    this.num = num;
  }
  then(resolve, reject) {
    alert(resolve); // function() { 네이티브 코드 }
    // 1초 후 this.num*2와 함께 이행됨
    setTimeout(() => resolve(this.num * 2), 1000); // (**)
  }
}

new Promise(resolve => resolve(1))
  .then(result => {
    return new Thenable(result); // (*)
  })
  .then(alert); // 1000밀리 초 후 2를 보여줌
```

프라미스 체이닝: thenable

- 핸들러는 thenable라는 객체를 반환하기도 함
- .then을 가진 객체는 모두 thenable 객체라고 부름

```
class Thenable {
  constructor(num) {
    this.num = num;
  }
  then(resolve, reject) {
    alert(resolve); // function() { 네이티브 코드 }
    // 1초 후 this.num*2와 함께 이행됨
    setTimeout(() => resolve(this.num * 2), 1000); // (**)
  }
}

new Promise(resolve => resolve(1))
  .then(result => {
    return new Thenable(result); // (*)
  })
  .then(alert); // 1000밀리 초 후 2를 보여줌
```

프라미스 체이닝: fetch와 함께

- 네트워크 요청 시 사용하는 fetch와 체이닝 함께 응용 예시

```
// user.json에 요청을 보냅니다.
fetch('/article/promise-chaining/user.json')
  // 응답받은 내용을 json으로 불러옵니다.
  .then(response => response.json())
  // GitHub에 요청을 보냅니다.
  .then(user => fetch(`https://api.github.com/users/${user.name}`))
  // 응답받은 내용을 json 형태로 불러옵니다.
  .then(response => response.json())
  // 3초간 아바타 이미지(githubUser.avatar_url)를 보여줍니다.
  .then(githubUser => {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => img.remove(), 3000); // (*)
  });
```

프라미스 체이닝: fetch와 함께

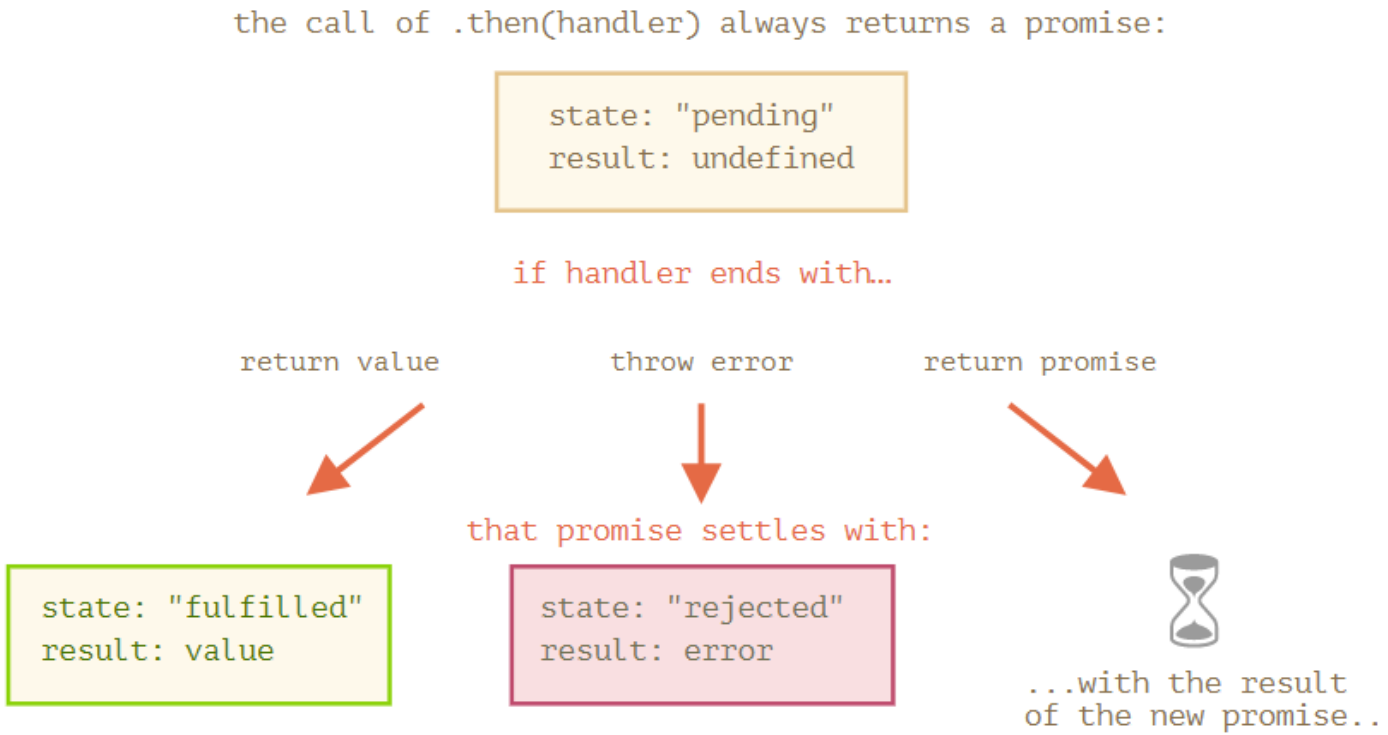
- 이미지를 보여준 후 무언가를 하기 위해서는 아래와 같이 코드 작성을 해야함

```
fetch('/article/promise-chaining/user.json')
  .then(response => response.json())
  .then(user => fetch(`https://api.github.com/users/${user.name}`))
  .then(response => response.json())
  .then(githubUser => new Promise(function(resolve, reject) { // (*)
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => {
      img.remove();
      resolve(githubUser); // (**)
    }, 3000);
  }))
  // 3초 후 동작함
  .then(githubUser => alert(`Finished showing ${githubUser.name}`));
```

프라미스 체이닝: 요약

프라미스와 async, await

- 핸들러가 Promise를 return하면 나머지 체인은 처리될 때까지 대기
- 처리 완료되면 Promise의 result가 다음 체인으로 전달됨



프라미스와 에러 핸들링

프라미스와 async, await

- Promise executor와 핸들러 주위에는 암시적 try/catch문이 존재
- 예외가 발생하면 try/catch에서 예외를 잡고, 이를 거부상태의 Promise로 변경함
- .catch문은 Promise에서 발생한 모든 에러를 다룸

```
new Promise((resolve, reject) => {
  throw new Error("에러 발생!");
}).catch(alert); // Error: 에러 발생!
```

```
new Promise((resolve, reject) => {
  reject(new Error("에러 발생!"));
}).catch(alert); // Error: 에러 발생!
```

```
new Promise((resolve, reject) => {
  resolve("ok");
}).then((result) => {
  throw new Error("에러 발생!"); // 프라미스가 거부됨
}).catch(alert); // Error: 에러 발생!
```

프라미스와 에러 핸들링: 중첩

프라미스와 async, await

- 처리할 수 없는 에러일 경우 catch문에서 에러를 던질 수 있음

```
// 실행 순서: catch -> catch
new Promise((resolve, reject) => {

  throw new Error("에러 발생!");

}).catch(function(error) { // (*)

  if (error instanceof URIError) {
    // 에러 처리
  } else {
    alert("처리할 수 없는 에러");

    throw error; // 에러 다시 던지기
  }

}).then(function() {
  /* 여기는 실행되지 않습니다. */
}).catch(error => { // (**)

  alert(`알 수 없는 에러가 발생함: ${error}`);
  // 반환값이 없음 => 실행이 계속됨

});
```

프라미스 API: all

프라미스와 async, await

- 여러 개의 Promise를 동시에 실행시키고 모든 Promise가 준비될 때까지 기다림
- 요소 전체가 Promise인 배열을 받고 새로운 Promise를 return 함
- 배열 result의 요소 순서는 Promise.all에 전달되는 Promise 순서와 상응함
- 전달되는 Promise 중 하나라도 거부되면 Promise.all 전체가 거부되며 .catch가 실행됨

```
Promise.all([
  new Promise(resolve => setTimeout(() => resolve(1), 3000)), // 1
  new Promise(resolve => setTimeout(() => resolve(2), 2000)), // 2
  new Promise(resolve => setTimeout(() => resolve(3), 1000))  // 3
]).then(alert); // 프라미스 전체가 처리되면 1, 2, 3이 반환됩니다. 각 프라미스는
```

프라미스 API: allSettled

프라미스와 async, await

- 각 Promise의 상태와 값 또는 에러를 받고자 하는 경우

```
let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
  'https://no-such-url'
];

Promise.allSettled(urls.map(url => fetch(url)))
  .then(results => { // (*)
    results.forEach((result, num) => {
      if (result.status == "fulfilled") {
        alert(`${urls[num]}: ${result.value.status}`);
      }
      if (result.status == "rejected") {
        alert(`${urls[num]}: ${result.reason}`);
      }
    });
  });
```

```
[
  {status: 'fulfilled', value: ...응답...},
  {status: 'fulfilled', value: ...응답...},
  {status: 'rejected', reason: ...에러 객체...}
]
```

프라미스 API: race

- Promise.all과 유사하지만, 가장 먼저 처리되는 Promise의 결과 혹은 에러를 return 함
- 전달된 Promise 중 어느 Promise가 처리되면 다른 Promised의 결과 혹은 에러는 무시됨

```
Promise.race([
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("에러 발생!")),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).then(alert); // 1
```

프라미스 API: resolve

- 호환성을 위해 함수가 Promise를 return하도록 하는 경우
- 함수를 호출하면 Promise가 return되는 것이 보장되므로 .then을 사용 가능

```
let cache = new Map();

function loadCached(url) {
  if (cache.has(url)) {
    return Promise.resolve(cache.get(url)); // (*)
  }

  return fetch(url)
    .then(response => response.text())
    .then(text => {
      cache.set(url,text);
      return text;
    });
}
```

프라미스 API: reject

- 결과값이 error인 거부 상태 Promise를 생성함
- 실무에서 이 메서드를 쓸 일은 거의 없음

```
let promise = new Promise((resolve, reject) => reject(error));
```

프라미스화

프라미스와 async, await

- 콜백을 받는 함수를 Promise를 return하는 함수로 바꾸는 과정

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`${src}를 불러오는 도중에 에러가

  document.head.append(script);
}
```

```
let loadScriptPromise = function(src) {
  return new Promise((resolve, reject) => {
    loadScript(src, (err, script) => {
      if (err) reject(err)
      else resolve(script);
    });
  })
}
```

프라미스화

프라미스와 async, await

- 콜백 함수를 promisification하는 헬퍼 함수

```
function promisify(f) {
  return function (...args) { // 래퍼 함수를 반환함
    return new Promise((resolve, reject) => {
      function callback(err, result) { // f에 사용할 커스텀 콜백
        if (err) {
          reject(err);
        } else {
          resolve(result);
        }
      }

      args.push(callback); // 위에서 만든 커스텀 콜백을 함수 f의 인수 끝에 추가합니다.

      f.call(this, ...args); // 기존 함수를 호출합니다.
    });
  };
};
```

마이크로태스크 프라미스와 async, await

- Promise가 즉시 이행되더라도 .then/catch/finally 아래에 있는 코드가 먼저 실행됨

```
let promise = Promise.resolve();

promise.then(() => alert("프라미스 성공!"));

alert("코드 종료"); // 이 얼럿 창이 가장 먼저 나타납니다.
```

- 비동기 작업은 항상 microtask queue를 통과함
- 실행할 것이 남아있지 않은 상태에만 microtask queue에 있는 작업이 FIFO에 맞추어 수행됨
- 여러 개의 핸들러를 사용해 만든 체인의 경우, 각 핸들러는 비동기적으로 실행됨

async와 await: async 함수

프라미스와 async, await

- 함수 앞에 async를 붙이면 해당 함수는 항상 Promise를 return함
- Promise가 아닌 값을 return하더라도 이행된 Proimse로 값을 감싸 return함

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

f().then(alert); // 1
```

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

f().then(alert); // 1
```

async와 await: await

프라미스와 async, await

- async 함수 안에서만 작동
- Promise가 처리될 때까지 기다린 후 결과를 return함
- Promise가 처리되길 기다리는 동안에 엔진이 다른 작업을 할 수 있음

```
async function f() {

  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("완료!"), 1000)
  });

  let result = await promise; // 프라미스가 이행될 때까지 기다림 (*)

  alert(result); // "완료!"
}

f();
```

async와 await: await

프라미스와 async, await

- promise.then을 async/await를 사용하는 예시

```
fetch('/article/promise-chaining/user.json')
  .then(response => response.json())
  .then(user => fetch(`https://api.github.com/users/${user.name}`))
  .then(response => response.json())
  .then(githubUser => new Promise(function(resolve, reject) { // (*)
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => {
      img.remove();
      resolve(githubUser); // (**)
    }, 3000);
  }))
  // 3초 후 동작함
  .then(githubUser => alert(`Finished showing ${githubUser.name}`));
```

async와 await: await 프라미스와 async, await

- promise.then을 async/await를 사용하는 예시

```
async function showAvatar() {

  // JSON 읽기
  let response = await fetch('/article/promise-chaining/user.json');
  let user = await response.json();

  // github 사용자 정보 읽기
  let githubResponse = await fetch(`https://api.github.com/users/${user.name}`);
  let githubUser = await githubResponse.json();

  // 아바타 보여주기
  let img = document.createElement('img');
  img.src = githubUser.avatar_url;
  img.className = "promise-avatar-example";
  document.body.append(img);

  // 3초 대기
  await new Promise((resolve, reject) => setTimeout(resolve, 3000));

  img.remove();

  return githubUser;
}
```

async와 await: await

프라미스와 async, await

- promise.then처럼 await에도 thenable 객체를 사용 가능

```
class Thenable {
  constructor(num) {
    this.num = num;
  }
  then(resolve, reject) {
    alert(resolve);
    // 1000밀리초 후에 이행됨(result는 this.num*2)
    setTimeout(() => resolve(this.num * 2), 1000); // (*)
  }
};

async function f() {
  // 1초 후, 변수 result는 2가 됨
  let result = await new Thenable(1);
  alert(result);
}
```

async와 await: 에러 핸들링

프라미스와 async, await

- Promise가 거부되면 throw문을 작성한 것처럼 에러가 던져짐

```
async function f() {
  await Promise.reject(new Error("에러 발생!"));
}
```

```
async function f() {
  throw new Error("에러 발생!");
}
```

- awiat가 던진 에러는 일반 try/catch문으로 잡을 수 있음
- await를 사용하면 .then이 거의 필요하지 않다는 점과 일반 try/catch문을 사용할 수 있는 점이 장점

async와 await: 활용 예시

- 일반 함수에서 async 함수를 호출하고 그 결과를 사용하는 경우

```
async function wait() {
  await new Promise(resolve => setTimeout(resolve, 1000));

  return 10;
}

function f() {
  // ...코드...
  // async wait()를 호출하고 그 결과인 10을 얻을 때까지 기다리려면 어떻게 해야 할까요?
  // f는 일반 함수이기 때문에 여기선 'await'를 사용할 수 없다는 점에 주의하세요!
}
```

async와 await: 활용 예시

- 일반 함수에서 async 함수를 호출하고 그 결과를 사용하는 경우

```
async function wait() {
  await new Promise(resolve => setTimeout(resolve, 1000));

  return 10;
}

function f() {
  // shows 10 after 1 second
  wait().then(result => alert(result));
}

f();
```

모듈 소개 모듈

- 클래스 하나 혹은 특정한 목적을 가진 복수의 함수로 구성된 라이브러리
- 모듈 내보내기: export 지시자를 사용하여 외부 모듈에서 해당 변수나 함수에 접근 가능
- 모듈 가져오기: import 지시자를 사용하여 외부 모듈의 기능을 가져올 수 있음

```
// 📁 sayHi.js
export function sayHi(user) {
  alert(`Hello, ${user}!`);
}
```

```
// 📁 main.js
import {sayHi} from './sayHi.js';

alert(sayHi); // 함수
sayHi('John'); // Hello, John!
```

모듈 소개 모듈

- 브라우저에서 모듈이 작동하는 예시

```
<!doctype html>
<script type="module">
  import {sayHi} from './say.js';

  document.body.innerHTML = sayHi('John');
</script>
```

모듈 소개: 핵심 기능 모듈

- 엄격 모드로 실행됨: 선언되지 않은 변수에 값을 할당하는 등의 코드는 에러를 발생

```
<script type="module">
  a = 5; // 에러
</script>
```

- 모듈 레벨 스코프: 모듈 내부에서 정의한 변수/함수는 다른 스크립트에서 접근 불가
- 공개하려는 모듈은 export하고, 내보내진 모듈을 가져와 사용하려면 import 해야함

모듈 소개: 핵심 기능 모듈

- 동일 모듈이 여러 곳에서 사용되더라도 최초 호출 시 단 한 번만 실행됨

```
// 📂 1.js
import {admin} from './admin.js';
admin.name = "Pete";

// 📂 2.js
import {admin} from './admin.js';
alert(admin.name); // Pete

// 1.js와 2.js 모두 같은 객체를 가져오므로
// 1.js에서 객체에 가한 조작을 2.js에서도 확인할 수 있습니다.
```

모듈 소개: 브라우저 특정 기능 모듈

- 모듈 스크립트는 항상 지연 실행됨
- HTML 문서 준비가 완료된 후 모듈 스크립트가 실행됨
- 문서 상 위쪽의 스크립트부터 차례로 실행되어 상대적 순서가 유지됨

```
<script type="module">
  alert(typeof button); // 모듈 스크립트는 지연 실행되기 때문에 페이지가 모두 로드
  // 얼럿창에 object가 정상적으로 출력됩니다. 모듈 스크립트는 아래쪽의 button 요소
</script>

하단의 일반 스크립트와 비교해 봅시다.

<script>
  alert(typeof button); // 일반 스크립트는 페이지가 완전히 구성되기 전이라도 바로
  // 버튼 요소가 페이지에 만들어지기 전에 접근하였기 때문에 undefined가 출력되는 것
</script>

<button id="button">Button</button>
```

모듈 소개: 브라우저 특정 기능 모듈

- 일반 스크립트에서 async 속성은 외부 스크립트를 불러올 때만 유효함
- 모듈 스크립트에서 async 속성은 다른 스크립트/HTML이 처리되길 기다리지 않고 바로 실행됨
- 이런 특징은 광고 노출과 같이 어디에도 종속되지 않은 기능을 구현할 때 유용하게 사용 가능

```
<!-- 필요한 모듈(analytics.js)의 로드가 끝나면 -->
<!-- 문서나 다른 <script>가 로드되길 기다리지 않고 바로 실행됩니다.-->
<script async type="module">
  import {counter} from './analytics.js';

  counter.count();
</script>
```

모듈 소개: 브라우저 특정 기능 모듈

- src 속성 값이 동일한 외부 스크립트인 경우 한 번만 실행됨

```
<!-- my.js는 한번만 로드 및 실행됩니다. -->
<script type="module" src="my.js"></script>
<script type="module" src="my.js"></script>
```

- Webpack과 같은 빌드 툴로 모듈을 한 데 묶어 관리함

모듈 내보내고 가져오기 모듈

- 선언부 앞에 export 붙이기

```
// 배열 내보내기
export let months = ['Jan', 'Feb', 'Mar','Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];

// 상수 내보내기
export const MODULES_BECAME_STANDARD_YEAR = 2015;

// 클래스 내보내기
export class User {
  constructor(name) {
    this.name = name;
  }
}
```

모듈 내보내고 가져오기 모듈

- 선언부와 떨어진 곳에 export 붙이기

```
// 📁 say.js
function sayHi(user) {
  alert(`Hello, ${user}!`);
}

function sayBye(user) {
  alert(`Bye, ${user}!`);
}

export {sayHi, sayBye}; // 두 함수를 내보냄
```

모듈 내보내고 가져오기 모듈

- import *

```
// 🗀 main.js
import {sayHi, sayBye} from './say.js';

sayHi('John'); // Hello, John!
sayBye('John'); // Bye, John!
```

```
// 🗀 main.js
import * as say from './say.js';

say.sayHi('John');
say.sayBye('John');
```

모듈 내보내고 가져오기 모듈

- import ‘as’ / export ‘as’

```
// 🗀 main.js
import {sayHi as hi, sayBye as bye} from './say.js';

hi('John'); // Hello, John!
bye('John'); // Bye, John!
```

```
// 🗀 say.js
...
export {sayHi as hi, sayBye as bye};
```

모듈 내보내고 가져오기 모듈

- 개체 하나만 선언되어 있는 방식으로 모듈을 만드는 걸 선호
- export default를 사용하면 해당 모듈엔 개체가 하나만 있다는 사실을 명확히 할 수 있음
- default를 붙여서 모듈을 내보내면 중괄호 {} 없이 모듈을 가져올 수 있음

```
// 📁 user.js
export default class User { // export 옆에 'default'를 추가해보았습니다.
  constructor(name) {
    this.name = name;
  }
}
```

```
// 📁 main.js
import User from './user.js'; // {User}가 아닌 User로 클래스를 가져왔습니다.

new User('John');
```

모듈 내보내고 가져오기: 다시 내보내기

- 특정 모듈을 외부에 공개할 경우, 패키지 안의 내부 구조를 건드리지 않도록 할 경우

```
// 📁 auth/index.js

// login과 logout을 가지고 온 후 바로 내보냅니다.
import {login, logout} from './helpers.js';
export {login, logout};

// User를 가져온 후 바로 내보냅니다.
import User from './user.js';
export {User};
...
```

```
// 📁 auth/index.js
// login과 logout을 가지고 온 후 바로 내보냅니다.
export {login, logout} from './helpers.js';

// User 가져온 후 바로 내보냅니다.
export {default as User} from './user.js';
```

모듈 내보내고 가져오기: 동적 작동 모듈

- import/export는 코드 구조의 중심을 잡아주기 때문에 정적인 방식으로 작동

```
import ... from getModuleName(); // 모듈 경로는 문자열만 허용되기 때문에 에러가 발생
```

```
if(...) {
  import ...; // 모듈을 조건부로 불러올 수 없으므로 에러 발생
}

{
  import ...; // import 문은 블록 안에 올 수 없으므로 에러 발생
}
```

모듈 내보내고 가져오기: 동적 작동 모듈

- import(module) 표현식은 모듈을 읽고, export하는 것들을 포함한 객체를 담은 Promise를 return함

```
// 📁 say.js
export function hi() {
  alert(`안녕하세요.`);
}

export function bye() {
  alert(`안녕히 가세요.`);
}
```

```
<script>
  async function load() {
    let say = await import('./say.js');
    say.hi(); // 안녕하세요.
    say.bye(); // 안녕히 가세요.
    say.default(); // export default한 모듈을 불러왔습니다!
  }
</script>
<button onclick="load()">클릭해주세요,</button>
```