

프라이미스와 비동기 함수 및 await 연산자 (***)

콜백

- 자바스크립트 호스트 환경이 제공하는 여러 함수를 사용하면 비동기(asynchronous) 동작을 스케줄링 가능
 - 동기 vs 비동기 => 동기는 순차적으로 작동, 비동기는 순차적으로 작동하지 않음
 - setTimeout은 스케줄링에 사용되는 가장 대표적인 함수
- 스크립트나 모듈을 로딩하는 것 또한 비동기적으로 동작함

```
function loadScript(src) {  
  // <script> 태그를 만들고 페이지에 태그를 추가  
  let script = document.createElement('script');  
  // 태그가 페이지에 추가되면 src에 있는 스크립트를 파일을 로딩 후 실행  
  script.src = src;  
  document.head.append(script);  
}
```

loadScript 함수 호출 코드

```
loadScript('/my/script.js');  
// loadScript 아래의 코드는 스크립트 로딩이 끝날 때까지 기다리지 않고 바로 실행됨  
const hello = "world";  
const sum = 1 + 2;  
// ...
```

- 스크립트 로딩이 끝나자마자 스크립트를 사용해서 무언가를 해야하는 상황을 가정해보기
 - loadScript를 호출하자마자 스크립트에서 정의한 함수를 호출하면, (스크립트 불러오기가 비동기적으로 동작하므로) 원하는 대로 작동하지 않음

```
// script.js에 newFunction이 정의되어있다고 가정  
loadScript('/my/script.js');  
// 이 시점에서는 스크립트 로딩이 완료되지 않았으므로, 함수가 존재하지 않는다는 에러가 발생함  
newFunction();
```

- 위와 같은 상황을 방지하기 위해서 loadScript의 두 번째 인수로 스크립트 로딩이 끝난 후 실행될 함수인 콜백(callback) 함수를 추가하도록 수정

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

  // 추가할 스크립트의 로딩이 끝나고 실행될 onload 함수를 설정
  // https://developer.mozilla.org/en-US/docs/Web/API/Window/load_event
  script.onload = () => callback(script);

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

- 두 번째 인수로 전달된 함수(대개 익명 함수)는 원하는 동작(여기서는 외부 스크립트를 불러오는 것)이 완료되었을 때 실행됨

```
loadScript('/my/script.js', function() {
  // 전달한 익명 함수(=콜백 함수)는 스크립트 로드가 끝나면 실행되므로 함수 호출이 제대로
  동작함
  newFunction();
  // ...
});
```

- 이런 방식을 **콜백 기반(callback-based) 비동기 프로그래밍**이라고 함

콜백 속 콜백

- 특정 스크립트에 의존성을 가지는 경우 중첩된 콜백 함수를 이용해서 코드 작성 가능

```
loadScript('/my/script.js', function(script) {
  // script2는 script에 의존함 (즉, script에 정의된 함수나 클래스를 이용해서 실행됨)
  loadScript('/my/script2.js', function(script) {
    // script3는 script2에 의존함
    loadScript('/my/script3.js', function(script) {
      // 세 개의 스크립트 파일 로딩이 모두 끝난 후 실행됨 (이 시점 이후 script,
      script2, script3 코드가 모두 실행되었다고 가정 가능)
      // ... 필요한 코드 작성 ...
    });
  });
});
```

- loadScript 함수 내용을 스크립트 로딩에 성공하면 callback(null, script)을 실패하면 callback(error)을 호출하도록 개선할 수 있음

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

  // 성공할 경우 callback(null, script) 호출
  script.onload = () => callback(null, script);
  // 실패할 경우 에러 객체 전달
  script.onerror = () => callback(new Error(`${src}를 불러오는 도중에 에러가 발생했습니다.`));

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


오류를 우선 처리하는 콜백(error-first callback) 함수를 전달하는 코드

```
loadScript('/my/script.js', function(error, script) {
  if (error) {
    // 에러 처리 관련 코드 작성
    // ...
  } else {
    // 스크립트 로딩이 성공적으로 끝났으므로 필요한 코드 작성
    // ...
  }
});
```

멸망의 피라미드

- 만약 특정 작업에 의존적인 비동기 동작이 많아질 경우 아래와 같은 코드 작성이 불가피함
 - 계속해서 들여쓰기가 발생하여 코드 가독성이 떨어지게 되는 callback hell, pyramid of doom 현상이 발생

```
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 {
            // ...
          }
        });
      }
    });
  }
});
```



위의 코드에서 발생한 문제를 해결한 코드

```

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) {
    handleError(error);
  } else {
    // 모든 스크립트가 로딩되면 다른 동작을 수행합니다. (*)
  }
};

```

- 동작상의 문제는 없고 콜백 지옥 현상이 해결되었지만, 코드의 가독성이 여전히 떨어지고 억지로 만든 함수(step"N")들이 여럿 존재하게 되는 불편함이 존재

프라이미스

- **제작 코드(producing code)**는 원격에서 스크립트를 불러오는 것 같은 시간이 걸리는 비동기 작업을 진행하는 코드
- **소비 코드(consuming code)**는 제작 코드의 결과를 기다렸다가 이를 소비하는 코드
- **프라이미스(promise)**는 "제작 코드"와 "소비 코드"를 연결해 주는 특별한 자바스크립트 객체

프라이미스 객체를 생성하는 코드

```

// Promise 생성자를 통해 함수 전달
let promise = new Promise(function(resolve, reject) {
  // ...
});

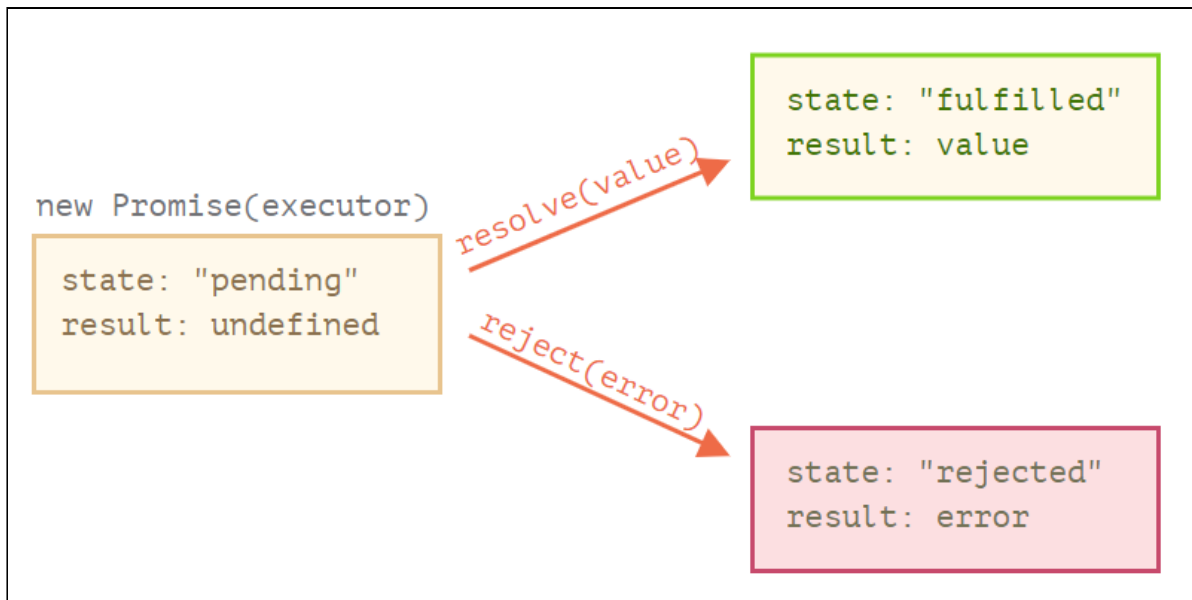
```

- Promise 생성자 함수로 전달하는 함수를 **executor(실행자, 실행 함수)**라고 부름
 - executor의 인수 resolve와 reject는 자바스크립트가 자체적으로 제공하는 콜백 함수
 - 개발자는 resolve와 reject를 신경 쓰지 않고 **executor 함수 내부의 코드(비동기 작업을 진행하는 코드)**를 작성
- 대신 executor 함수 내에서 결과를 즉시 얻든, 늦게 얻든 상관없이 **상황(성공 or 실패)에 따라 인수로 넘겨준 콜백 중 하나를 반드시 호출**해야 함

resolve, reject 함수의 역할

`resolve(value)` => (일이 성공적으로 끝난 경우) 결괏값을 전달하며 호출
`reject(error)` => (일이 실패한 경우 혹은 에러 발생 시) 에러 객체를 전달하며 호출

- `executor` 함수는 자동으로 실행되므로 코드 내부에서 **성공, 실패 여부에 따라 `resolve`나 `reject`를 호출**하면 됨
 - `executor` 함수 내부에서는 **`resolve`나 `reject` 함수 중 하나를 반드시 호출**해야 하며, 이때 변경된 상태(`state`)는 더 이상 변하지 않게 됨
 - 즉, 한 번 처리가 끝난 프라미스에 `resolve`와 `reject` 함수를 호출하면 무시됨
 - `resolve`나 `reject` 함수에는 **인수를 하나만 전달 가능**
 - 단, 상황에 따라 인수값을 전달안해도 무방하며 그냥 성공 여부만 알아도 괜찮다면 `resolve`를 인수 없이 호출해도 됨
- Promise 생성자 함수 호출 결과로 반환된 프라미스 객체는 다음과 같은 (접근 불가능한) 내부 숨김 속성을 가짐
 - `state` 속성 => 처음엔 **pending(보류)**이었다가 `resolve`가 호출되면 **fulfilled(이행)**, `reject`가 호출되면 **rejected(거부)**로 변경됨
 - `result` 속성 => 처음엔 `undefined`이었다가 `resolve(value)`가 호출되면 `value` 값으로, `reject(error)`가 호출되면 `error` 값으로 변경됨



resolve, reject 함수 호출에 따르는, 프라미스 객체의 내부 상태 변화

- `executor` 함수 내부의 작업이 잘 마무리된 경우에는 다음과 같이 `resolve` 함수를 호출해야 함

```
let promise = new Promise(function(resolve, reject) {
  // 프라미스가 만들어지면 executor 함수는 자동으로 실행됩니다.

  // 1초 뒤에 결괏값("done")을 전달하며 resolve 함수를 호출하여 작업이 성공적으로 끝났음을 알림
  setTimeout(() => resolve("done"), 1000);
});
```

- 작업이 성공적으로 처리되었을 때의 프라미스를 **fulfilled promise(약속이 이행된 프라미스)**라고 부름
- 만약 작업이 잘못된 경우에는 다음과 같이 `reject` 함수를 호출

```
let promise = new Promise(function(resolve, reject) {
  // 1초 뒤에 에러 객체를 전달하며 reject 함수를 호출하여 작업 실패를 알림
  setTimeout(() => reject(new Error("에러 발생!")), 1000);
});
```

- reject 함수 호출시 전달할 인수값의 타입에는 제한이 없으므로, 반드시 에러 객체를 전달해야 할 필요는 없지만 **보통 Error 객체 혹은 Error를 상속받은 객체를 전달함**
- 이행(resolved)되거나 거부(rejected)된 상태(즉, 어떤식으로든 작업이 마무리된 상태)의 프라미스는 **처리된(settled) 프라미스**라고 부르며 아직 작업이 진행중인 프라미스는 **대기 중인(pending) 프라미스**라고 부름

소비자 : then, catch, finally

- 프라미스 객체는 **executor** 함수를 통해 처리된 결과나 에러를 받을 소비 함수를 이어주는 역할을 함
 - 프라미스 객체에서 제공하는 **then, catch, finally 메서드**를 통해 결과나 에러를 소비 가능

then 메서드

- then의 첫 번째 인수는 **프라미스가 이행되었을 때 실행되는 소비 함수**이고, 여기서 결과값을 받게 됨
- then의 두 번째 인수는 **프라미스가 거부되었을 때 실행되는 소비 함수**이고, 여기서 에러 객체를 받게 됨

```
promise.then(
  function(result) { /* 결과값을 이용한 코드 작성 */ },
  function(error) { /* 에러 처리 관련 코드 작성 */ }
);
```

resolve 함수 실행하는 시나리오의 코드

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

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

reject 함수 실행하는 시나리오의 코드

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

// reject 함수는 then의 두 번째 함수를 실행
promise.then(
  result => alert(result),    // 실행되지 않음
  error => alert(error)      // 1초 후 "Error: 에러 발생!"를 출력
);
```

- 작업이 성공적으로 처리된 경우에만 특정 코드를 실행하고 싶으면 then에 첫 번째 함수만 전달하면 됨

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

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

상황에 따라 성공 및 실패 여부가 달라지는 일반적인 용례를 보여주는 코드

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => {
    // 특정 작업을 진행한 후 이후 랜덤하게 성공, 실패 여부가 결정된다고 가정
    let success = Math.random() + .5 >> 0;

    if(success) {
      // 성공하면 resolve 함수 호출하며 결과값 전달
      resolve({ result: "Hello" });
    } else {
      // 실패하면 reject 함수 호출하며 에러 객체 전달
      reject(new Error("실패"));
    }
  }, 1000);
});

promise.then(
  result => alert(JSON.stringify(result)),
  error => alert(error)
);
```

catch 메서드

- 에러가 발생한 경우만 다루고 싶다면 then 메서드의 첫 번째 인수로 null을 전달하고 두 번째 인수로 에러 처리 코드를 담은 함수 전달하거나 **catch 메서드**를 사용할 수 있음
 - catch는 코드를 간결하게 작성할 수 있다는 점을 제외하면 **then(null, f)**과 완벽하게 같음

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

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

finally 메서드

- 프라미스가 처리되면(즉, 이행되거나 거부되면) **finally 메서드로 전달한 함수가 무조건 실행됨**
- 쓸모가 없어진 로딩 인디케이터(loading indicators)를 멈추는 경우같이, 결과와 상관없이 실행해야 할 **마무리 작업 코드가 필요하면 finally 메서드를 활용** 가능 (try - catch - finally 문의 finally와 같은 역할 수행)
- finally 핸들러(핸들러 => 전달한 함수)엔 전달 받을 인수가 없으며 프라미스가 이행되었는지, 내부에서 거부되었는지 여부도 알 수 없음
 - finally 핸들러에서는 **절차를 마무리하는 보편적 동작을 수행하기** 때문에 성공, 실패 여부를 몰라도 됨
- finally 핸들러는 자동으로 다음 핸들러에 결괏값 혹은 에러값을 전달함

```
new Promise((resolve, reject) => {
  /* 시간이 걸리는 어떤 일을 수행하고, 그 후 resolve 혹은 reject 함수를 호출 */
})
// finally로 전달한 함수는 성공, 실패 여부와 상관없이 프라미스가 처리되면 무조건 실행됨
.finally(() => /* 로딩 인디케이터 중지 코드 실행 */)
.then(result => result와 err 보여줌 => error 보여줌)
```

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

```
new Promise((resolve, reject) => {
  throw new Error("에러 발생!");
})
.finally(() => alert("프라미스가 준비되었습니다."))
.catch(err => alert(err)); // <-- catch에서 에러 객체를 다룰 수 있음
```

- 프라미스가 대기(pending) 상태일 때, then/catch/finally 핸들러는 프라미스가 처리(settled)되길 기다림. 반면, **프라미스가 이미 처리된 상태라면 핸들러가 즉각 실행됨**

```
// 아래 프라미스는 생성과 동시에 이행된 상태로 처리됨
let promise = new Promise(resolve => resolve("완료!"));

// 기다리지 않고 바로 "완료!" 출력
promise.then(alert);
```

프라미스 체이닝

- 프라미스 체이닝은 **순차적으로 처리해야 하는 비동기 작업이 여러 개 있는 상황**에 적용할 수 있는 방법

프라미스 체이닝이 적용된 코드

```
new Promise(function(resolve, reject) {
  // (*)
  setTimeout(() => resolve(1), 1000);
}).then(function(result) {
  // resolve 호출의 결과로 전달된 값(1)이 result로 넘어옴 (**)
  alert(result); // 1
  // 여기서 2가 반환되지만, 내부적으로 반환값을 resolve 함수로 전달하는 Promise 객체를
  // 생성하여 반환하므로, 결과적으로 연쇄적인 then 메서드 호출이 가능함
  return result * 2;
  // 위의 코드는 내부적으로는 아래와 같은 작업을 실행하는 코드로 바뀌어서 실행됨을 유의!
  // return new Promise(resolve => resolve(result * 2));
}).then(function(result) {
  // (***)
  alert(result); // 2
  return result * 2;
}).then(function(result) {
  alert(result); // 4
  return result * 2;
});
```

위의 코드가 실행되는 양상

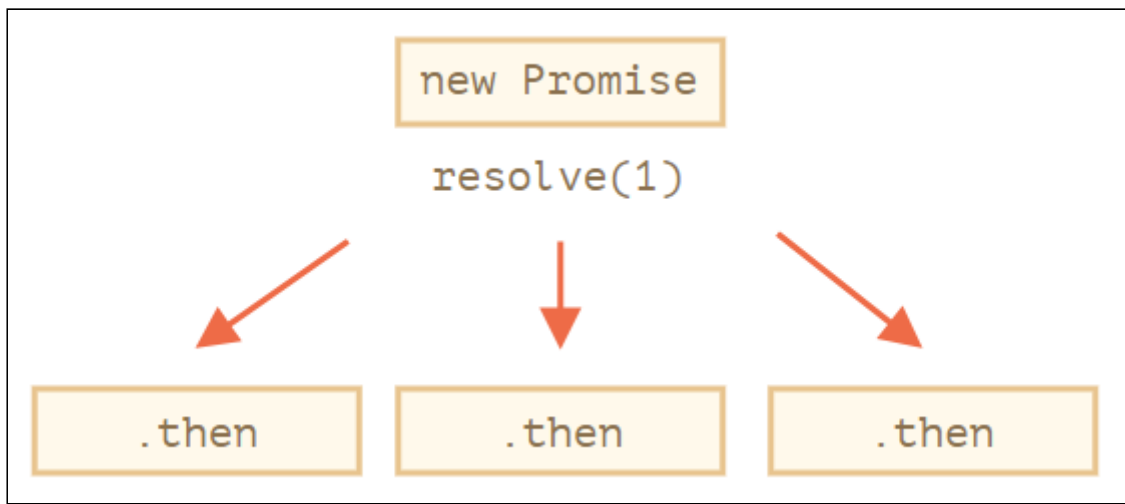
- 1초 후 최초 프라미스가 이행됨 - (*)
- 이후 then 핸들러가 호출됨 - (**)
- 2에서 반환한 값이 다음 then 핸들러로 전달됨 - (***)
- 이런 과정이 계속 이어지며 then 핸들러가 실행됨

- 프라미스 체이닝이 가능한 이유는 **then 메서드의 호출 결과로 프라미스 객체 반환되기 때문**이며, 이러한 이유로 반환된 프라미스 객체의 then 메서드를 연이어서 호출할 수 있음

잘못된 프라미스 체이닝 사용 방법 (체이닝이 일어나지 않게 됨)

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve(1), 1000);
});

// 동일한 프라미스 객체에 여러번 then 메서드를 호출했으므로 같은 result값(1)이 전달됨 (체이닝 효과가 생기지 않음)
promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});
promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});
promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});
```



동일한 프라미스 객체의 then 핸들러를 여러번 호출

프라미스 반환하기

- then 핸들러 내부에서 새로운 프라미스 객체를 생성하여 반환할 수도 있음

```
new Promise(function(resolve, reject) {
  setTimeout(() => resolve(1), 1000);
}).then(function(result) {
  alert(result); // 1

  // then 핸들러 내부에서 또 다른 "비동기 작업"을 수행할 프라미스 객체를 생성하여 반환
  // (즉, 이러한 방식을 통해 연속적인 비동기 작업을 수행할 수 있음)
  return new Promise((resolve, reject) => { // (*)
    // 1초 뒤 resolve 호출 (이후의 then 핸들러가 실행됨)
    setTimeout(() => resolve(result * 2), 1000);
  });
}).then(function(result) {
  // 여기서의 then 핸들러는 위의 then 핸들러 내부에서 생성한 비동기 객체와 연결된 then 핸
  // 들러임을 유의
  alert(result); // 2

  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(result * 2), 1000);
  });
}).then(function(result) {
  alert(result); // 4
});
```

fetch 함수와 체이닝 함께 응용하기

- fetch 함수는 네트워크 요청(대표적인 비동기 작업)을 보내는 함수로 요청 결과로 프라미스 객체를 반환

fetch 활용 네트워크 요청 코드

```

fetch('/article/promise-chaining/user.json')
  // 원격 서버가 응답하면 then 핸들러가 실행됨
  .then(function(response) {
    // response.text()는 응답 텍스트 전체가 다운로드되면 결괏값(응답 텍스트)을
    resolve 함수로 전달하는 이행 프라미스 객체를 생성 후 반환함
    return response.text();
  })
  .then(function(text) {
    // 원격에서 받아온 파일의 내용
    alert(text); // {"name": "iliakan", "isAdmin": true}
  });

```

json 메서드를 통해 텍스트가 아닌 객체를 전달받도록 수정한 코드

```

fetch('/article/promise-chaining/user.json')
  .then(response => response.json()) // 위 코드와 동일한 기능을 하지만
  response.json()은 원격 서버에서 불러온 내용을 JSON으로 변경
  .then(user => alert(user.name)); // iliakan, got user name

```

연속적인 비동기 작업을 수행하는 GitHub API 호출 코드

```

// 유저 정보 API 호출
fetch('/article/promise-chaining/user.json')
  // 객체로 변환
  .then(response => response.json())
  // 객체 정보를 통해서 GitHub API 호출
  .then(user => fetch(`https://api.github.com/users/${user.name}`))
  // 다시 객체로 변환
  .then(response => response.json())
  // 이미지 요소를 생성하여 일정 시간 보여주는 프라미스 객체 반환
  .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);
  })))
  // setTimeout 함수 호출의 결과로 3초후 alert 메시지 출력
  .then(githubUser => alert(`Finished showing ${githubUser.name}`));

```

프라미스와 에러 핸들링

- (문제가 발생하여) 프라미스가 거부되면 제어 흐름이 제일 가까운 catch 핸들러로 넘어가게 됨

```

// 존재하지 않는 주소이므로, fetch 내부에서 reject 함수가 실행됨
fetch('https://no-such-server.blabla')
  .then(response => response.json())
  // 가장 가까운 reject 함수 핸들러 실행
  .catch(err => alert(err)) // TypeError: failed to fetch

```

암시적 try .. catch 블록

- executor 함수와 프라미스 핸들러의 코드 주위엔 보이지 않는 try .. catch 블록이 존재하며, 예외가 발생하면 암시적 try .. catch 블록에서 예외를 잡고, reject 함수를 호출한 것과 같은 처리를 함 (=> 잡힌 에러 객체를 reject 함수로 전달하며 호출)

```
new Promise((resolve, reject) => {
  // 예외 발생 (이 경우 reject 함수를 호출하며 해당 예외 객체를 전달한 것과 똑같은 결과가 발생함)
  throw new Error("에러 발생!");

  // 다음과 같이 바깥에 보이지 않는 try - catch 블록이 존재한다고 생각하기
  /*
  try {
    // ...
  } catch(e) {
    reject(new Error("에러 발생!"));
  }
  */
})
.catch(alert); // Error: 에러 발생!
```

위의 코드와 똑같은 역할을 하는 코드

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

- then 핸들러 안에서 throw 문을 사용해 에러를 던지면, 이 자체가 거부된 프라미스를 생성하여 반환한 것과 같은 결과를 만들고, 따라서 제어 흐름이 가장 가까운 에러 핸들러로 넘어가게 됨

```
new Promise((resolve, reject) => {
  // 여기서는 작업이 잘 이행됨
  resolve("ok");
}).then((result) => {
  // 중간 단계에서 에러 발생 (거부된 프라미스 객체 반환)
  throw new Error("에러 발생!");
})
.catch(e => {
  // Error: 에러 발생!
  alert(e);
});
```

- throw문이 만든 에러뿐만 아니라 모든 종류의 에러가 암시적 try .. catch에서 처리됨

```

new Promise((resolve, reject) => {
  resolve("ok");
}).then((result) => {
  // 존재하지 않는 함수 호출
  blabla();
})
.catch(alert); // ReferenceError: blabla is not defined

```

다시 던지기

- 체인 마지막의 catch 메서드는 타 언어의 catch 블록과 유사한 역할을 할 수 있음
 - then 핸들러를 원하는 만큼 사용하다 마지막에 catch 핸들러 하나만 붙이면 해당 핸들러에서 then 핸들러에서 발생한 모든 에러를 처리함

```

// 작업 도중 에러가 발생할 경우 맨 밑의 catch 핸들러 실행
new Promise((resolve, reject) => {
  // throw new Error("에러 발생 1");
  resolve(1);
})
.then(result => {
  // throw new Error("에러 발생 2");
  return result * 2;
})
.then(result => {
  // throw new Error("에러 발생 3");
  return result * 2;
})
.then(alert) // 중간 작업들이 모두 성공
.catch(alert); // 도중 한 번이라도 실패

```

- 에러를 잡고, 처리할 수 없는 에러라고 판단되면 throw 문으로 다시 에러를 던져서 뒤의 catch 핸들러에서 처리하도록 할 수 있음

```

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 (이후의 가장 가까운 catch 핸들러 실행)
  }
}).then(function() {
  /* 여기는 실행되지 않습니다. */
}).catch(error => { // (**)
  alert(`알 수 없는 에러가 발생함: ${error}`);
  // 반환값이 없음 => 실행이 계속됨
});

```

처리되지 못한 거부

```
<script>
// html 파일 만들어서 실행
// https://stackoverflow.com/questions/40026381/unhandledrejection-not-working-in-chrome
window.addEventListener('unhandledrejection', function(event) {
    // 이벤트엔 두 개의 특별 프로퍼티가 있습니다.
    alert(event.promise); // [Object Promise] - 에러를 생성하는 프라미스
    alert(event.reason); // Error: 에러 발생! - 처리하지 못한 에러 객체
});

// 성공 상태의 프라미스를 처리하는 핸들러만 존재하고, 실패를 처리할 catch 핸들러가 없는 상황
new Promise(function() {
    // 존재하지 않는 함수 호출로 인한 에러 발생
    noSuchFunction();
})
.then(() => {});
</script>
```

- 에러가 발생하면 프라미스는 거부상태가 되고 실행 흐름은 가장 가까운 catch 핸들러로 넘어감
 - 그런데 위의 코드에서는 예외를 처리해 줄 핸들러가 없어서 **에러가 갇혀버리게 됨**
- 자바스크립트 엔진은 프라미스 거부를 추적하다가 에러를 처리할 코드가 없는 상황이 발생하면 **전역 에러를 생성함**
 - 브라우저 환경에선 이렇게 발생한 에러와 에러를 발생시킨 프라미스 객체를 **unhandledrejection 이벤트**를 통해서 전달받을 수 있음

