에러 핸들링

try .. catch와 에러 핸들링

- 에러가 발생하면 스크립트는 **즉시 중단되고 콘솔에 에러가 출력**됨
- try .. catch 문법을 사용하면 스크립트가 죽는 걸 방지하고, 에러를 잡아서(catch) **필요한 복구 로직을 작성**할 수 있음

try .. catch 문법

- try .. catch 문법은 try와 catch라는 **두 개의 주요 블록으로 구성**

```
try {
    // 정상적으로 동작하도록 작성된 그러나 예외가 발생할 수도 있는 코드
    // ...
} catch (err) {
    // 전달된 err 객체는 에러에 대한 정보가 담긴 에러 객체
    // 여기에서 에러에 따르는 필요한 코드 작성 (에러 핸들링)
}
```

try .. catch 동작 알고리즘

```
1. try {...} 안의 코드가 실행됨
2. 에러가 없다면, try 안의 마지막 줄까지 실행되고, catch 블록은 건너뜀
3. 에러가 있다면, try 안 코드의 실행이 중단되고, catch 블록으로 제어 흐름이 넘어가게 됨,
그 과정에서 catch 블록의 파라미터로 무슨 일이 일어났는지에 대한 설명이 담긴 에러 객체
err(아무 이름이나 사용 가능)이 전달됨
```

- try 블록 안에서 에러가 발생해도 catch 블록을 추가했다면 해당 블록으로 제어권이 넘어가기 때문에 **스크립트는 중단되지 않고 계속 실행**됨
    - 단, 적절한 에러 처리가 이루어지는지 여부를 검사하지는 않음 (즉, catch 블록에서 에러에 따르는 처리를 아무것도 하지 않아도 스크립트가 계속 실행됨을 유의)

에러가 발생하지 않는 코드

```
try {
    // 어떠한 에러도 발생하지 않는 코드
    alert('try 블록 시작');
    alert('try 블록 끝');
} catch(err) {
    alert('에러가 없으므로, catch는 무시됩니다.');
}
```

에러가 발생하는 코드

```
try {
    alert('try 블록 시작');
    lalala; // "변수가 정의되지 않음" 에러 발생
    // 에러가 발생한 후 뒤의 코드는 실행되지 않음
    alert('try 블록 끝(절대 도달하지 않음)');
} catch(err) {
    // catch 블록으로 진입
    alert('에러가 발생했습니다!');
}
```

- try .. catch는 동기적으로 동작하므로 setTimeout처럼 **스케줄 된(scheduled) 코드에서 발생한 예외**는 try .. catch에서 잡아낼 수 없음을 유의

```
try {
    // 1초 후 함수가 동작하도록 예약
    setTimeout(function() {
        // 에러가 발생하고, 스크립트가 여기서 중단되도록 코드 작성
        noSuchVariable;
    }, 1000);
} catch(e) {
    // catch 블록으로 진입하지 않음을 유의!
    alert( "작동 멈춤" );
}
```

- setTimeout에 넘겨진 익명 함수는 엔진이 try .. catch를 떠난 다음에서야 실행되기 때문에 이러한 현상이 발생함
    - 스케줄 된 함수 내부의 예외를 잡으려면 다음과 같이 **try .. catch를 반드시 함수 내부에 구현**해야 함

```
setTimeout(function() {
    // try .. catch 블록을 전달 함수 내부에서 작성
    try {
        noSuchVariable; // 이제 try..catch에서 에러를 핸들링 할 수 있습니다!
    } catch {
        alert( "에러를 잡았습니다!" );
    }
}, 1000);
```

에러 객체

- 에러가 발생하면 자바스크립트는 **에러와 관련된 상세내용이 담긴 객체(에러 객체)**를 생성한 후, **catch 블록에 이 객체를 인수로 전달**
- 내장 에러 전체와 에러 객체는 **두 가지 주요 프로퍼티**를 가짐

```
name
에러 이름. 정의되지 않은 변수 때문에 발생한 에러라면 "ReferenceError"가 이름이 됨

message
에러 상세 내용을 담고 있는 문자 메시지

stack (비표준 프로퍼티)
현재 호출 스택. 에러를 유발한 중첩 호출들의 순서 정보를 가진 문자열로 디버깅 목적으로 사용
```

에러 객체 활용 코드

```
try {
    lalala; // 에러, 변수가 정의되지 않음!
} catch(err) {
    alert(err.name);    // ReferenceError
    alert(err.message); // lalala is not defined
    alert(err.stack);   // ReferenceError: lalala is not defined at ... (이후 호출 스택 출력)
    debugger
    // 에러 전체를 보여줄 수도 있습니다.
    // 이때, 에러 객체는 "name: message" 형태의 문자열로 변환됨
    alert(err); // ReferenceError: lalala is not defined
}
```

선택적 catch 바인딩

- 에러에 대한 자세한 정보가 필요하지 않으면(즉, 에러 객체가 필요하지 않으면) catch 블록의 에러 객체 전달 소괄호를 생략 가능

```
try {
    throw new Error("Hello");
} catch {
    // catch 블록에서 (err) 부분 생략 (여전히 catch 블록은 예외 발생시 동작)
    alert("에러 발생!");
}
```

잘못된 json 형식 문자열을 파싱하다 생긴 에러 잡기

```
let json = "{ bad json }";

try {
    let user = JSON.parse(json); // <= 여기서 에러가 발생했으므로
    alert(user.name); // 이 코드는 동작하지 않음
} catch (e) {
    // 에러가 발생하면 제어 흐름이 catch 블록으로 넘어오게 됨
    alert("데이터에 에러가 있어 재요청을 시도합니다.");
    alert(e.name);
    alert(e.message);
}
```

직접 에러를 만들어서 던지기

- throw 연산자를 사용하여 에러를 직접 발생시킬 수 있음

```
throw <에러 객체>;
```

- 이론적으로는 숫자, 문자열 같은 원시형 자료를 포함한 어떤 것이든 에러 객체(error object)로 사용할 수 있지만 **기본적으로 제공되는 내장 에러 객체와의 호환을 위해 되도록 에러 객체에 name과**

message 프로퍼티를 넣어주는 것이 권장됨

- 자바스크립트는 Error, SyntaxError, ReferenceError, TypeError등의 **표준 에러 객체 관련 생성자 함수를 지원**하므로 생성자 함수를 이용해서 에러 객체를 만들 수 있음

```
let error = new Error(message);
let error = new SyntaxError(message);
let error = new ReferenceError(message);
```

- 일반 객체가 아닌 내장 생성자를 사용해 만든 내장 에러 객체의 name 속성값은 생성자 함수의 이름과 동일하며, message 속성값은 생성자 함수로 전달한 문자열이 됨

```
// Error 생성자 함수 사용하여 내장 에러 객체를 생성
let error = new Error("이상한 일이 발생했습니다. o_O");

alert(error.name); // name은 생성자 함수의 이름인 Error
alert(error.message); // messsage는 전달한 인수 ("이상한 일이 발생했습니다. o_O")
```

throw 연산자를 사용해 에러 던지기

```
// 논리적으로는 불완전한 데이터 (name 속성이 반드시 있어야 한다고 가정)
let json = '{ "age": 30 }';

try {
    let user = JSON.parse(json);
    // 논리적으로 불완전한 데이터인지 검사 후 직접 에러 발생시키기
    if (!user.name) throw new SyntaxError("불완전한 데이터: 이름 없음");
    alert(user.name);
} catch(e) {
    // SyntaxError (message: 불완전한 데이터: 이름 없음)
    alert(`${e.name} (message: ${e.message})`);
}
```

에러 다시 던지기

- catch 블록에서는 **자신이 처리할 수 있는 에러만 처리하고 나머지 에러는 다시 던져서 외부에서 처리**할 수 있게 해야 함
    - 처리할 수 있는 에러 타입을 instanceof 연산자로 체크 가능

에러 다시 던지기 진행 과정

```
1. 먼저 catch 블록에서 모든 에러를 전달 받음
2. catch 블록 안에서 에러 객체를 분석
3. 에러 처리 방법을 알지 못하는 에러가 발생한 경우, throw <에러객체> 명령어를 통해 에러를 다시 발생시킴
```

에러 다시 던지기 방법 적용 코드

```
function readData() {
    let json = '{ "name": "John", "age": 30 }';

    try {
        let user = JSON.parse(json);
```

```
        // 불완전한 정보가 있는 것은 readData 함수 내부적으로 처리가 가능한 에러
        if (!user.name) throw new SyntaxError("불완전한 데이터: 이름 없음");
        // 함수 내부에서 발생하리라 예상하지 못한 에러가 발생했다고 가정
        blabla();
        alert( user.name );
    } catch(e) {
        // readData 함수 내부적으로 처리할 수 있는 에러(SyntaxError)만 처리
        // (이 과정에서 에러 타입을 확인하기 위해서 instanceof 명령어를 사용)
        if (e instanceof SyntaxError) {
            alert( "JSON Error: " + e.message );
        } else {
            // 처리할 수 없는 에러는 다시 던져서 외부(함수 호출 맥락)에서 처리하도록 함
            throw e;
        }
    }
}

try {
    readData();
} catch (e) {
    // 함수를 호출하는 쪽에서 나머지 예외에 대한 처리를 진행
    alert( "External catch got: " + e );
}
```

try .. catch .. finally

- try .. catch 에 finally라는 코드 블록을 하나 더 추가할 수 있으며 **finally 블록 안의 코드**는 다음과 같은 상황에서 실행됨

```
에러가 없는 경우: try 블록 실행이 끝난 후
에러가 있는 경우: catch 블록 실행이 끝난 후
```

try .. catch .. finally 문법

```
try {
    // ... 정상 코드 실행 ...
} catch(e) {
    // ... 에러 핸들링 ...
} finally {
    // ... 예외 발생 여부와 상관 없이 항상 실행 ...
}
```

```
try {
    alert("try 블록 시작");
    if(confirm("에러를 만드시겠습니까?")) {
        이상한_코드();
    }
} catch(e) {
    alert("catch");
} finally {
    alert("finally");
}
```

피보나치 함수의 수행 시간 구하는 코드

```
let num = +prompt("양의 정수를 입력해주세요.", 35);

function fib(n) {
    if (n < 0 || Math.trunc(n) != n) {
        throw new Error("음수나 정수가 아닌 값은 처리할 수 없습니다.");
    }
    return n <= 1 ? n : fib(n - 1) + fib(n - 2);
}

// try, catch, finally 블록은 모두 다른 블록 맥락을 생성하여 블록 내부에서 정의한 변수는 지역변수가 되어버리므로, 바깥에 같이 공유할 변수 선언
let diff, result;
let start = Date.now();

try {
    result = fib(num);
} catch (e) {
    result = 0;
} finally {
    // fib 함수 작동이 실패하더라도 연산 시간을 구할 수 있음
    diff = Date.now() - start;
}

alert(result || "에러 발생");
alert(`연산 시간: ${diff}ms`);
```

전역 catch

- try .. catch에서 처리하지 못한 에러를 잡는 것은 중요하기 때문에 자바스크립트 호스트 환경 대다수는 자체적으로 에러 처리 기능을 제공
- 브라우저 환경에선 **window 객체의 onerror 속성에 함수를 할당**하여 에러를 처리할 수 있으며 **예상치 못한(=try .. catch 불록에서 잡히지 못한) 에러가 발생했을 때 이 함수가 실행**됨 (즉, 전역 에러 catch 블록이라고 생각해도 무방)

global_catch.html

```
<script>
// 전역적으로 에러를 잡을 수 있도록 onerror 함수 정의
window.onerror = function(message, url, line, col, error) {
    // Uncaught ReferenceError: badFunc is not defined At 8:9 of file:///C:/Users/Mirim/Desktop/global_catch.html
    alert(`${message}\n At ${line}:${col} of ${url}`);
};

function readData() {
    // ReferenceError 에러 발생
    // (에러를 처리할 try .. catch 블록이 없으므로 에러가 처리되지 않고 스크립트가 중단되게 됨)
    badFunc();
}

readData();
```

```
</script>
```

커스텀 에러와 에러 확장

- 개발을 하다 보면 **자체 에러 클래스가 필요한 경우**가 종종 발생함
  - ex) 네트워크 관련 작업 중 에러가 발생했다면 HttpError, 데이터베이스 관련 작업 중 에러가 발생했다면 DbError, 검색 관련 작업 중 에러가 발생했다면 NotFoundError 클래스가 필요
- 직접 에러 클래스를 만든 경우 에러 객체는 message, name 프로퍼티를 지원하고 가능하다면 stack 프로퍼티도 지원해야 함
  - 필요한 경우 이러한 필수 프로퍼티 이외에도 **에러의 내용과 관련 있는 다른 프로퍼티를 추가 가능**
    - ex) HttpError 클래스의 객체에 statusCode 프로퍼티를 만들고 404나 403, 500같은 상태 코드 숫자를 설정하기
- throw의 인수엔 아무런 제약이 없기 때문에 커스텀 에러 클래스는 반드시 Error 클래스를 상속할 필요는 없으며 그냥 에러 객체를 던져도 무방함
  - 그러나 Error를 상속받아 커스텀 에러 클래스를 만들게 되면 instanceof(ex: obj instanceof Error)를 사용해서 **에러 객체를 식별할 수 있다는 장점**이 생김
    - 이런 장점 때문에 (객체 리터럴등을 이용하여) 임의의 커스텀 에러 객체를 만드는 것보다, **Error 클래스를 상속받도록 하여 에러 객체를 만드는 것이 권장**됨
- 커스텀 에러 클래스들의 계층 구조를 형성해주는 것도 가능함 (ex: HttpTimeoutError는 HttpError를 상속받기)

ValidationError 클래스 정의

```
// Error 클래스 상속 받으며 커스터 에러 정의
// (여기서는 폼의 입력 양식이 잘못된 경우 발생시킬 ValidationError 에러 클래스 정의)
class ValidationError extends Error {
    constructor(message) {
        // 부모 생성자를 호출하여 message 프로퍼티 정의
        super(message);
        // name 프로퍼티 정의 (보통 클래스 이름)
        this.name = "ValidationError";
    }
}

function test() {
    throw new ValidationError("에러 발생!");
}

try {
    test();
} catch(err) {
    alert(err.message); // 에러 발생!
    alert(err.name);    // ValidationError
    alert(err.stack);   // 각 행 번호가 있는 중첩된 호출들의 목록
}
```