ECMAScript 6+

ES6/ES2017 Basics

2019.03

박재성

이 문서는 나눔글꼴로 작성되었습니다. 설치하기

목차

1. ECMAScript
2. 환경구성: Polyfill, Babel, Webpack
3. 변수: let, const
4. Function - 디폴트 파라미터
5. Arrow function
6. Object 표현식
7. Destructuring
8. Rest parameter / Spread operator
9. Template Literals / Tagged Template
10. Class
11. Modules
12. Promise
13. Async/Await
14. Fetch API

목차

1. ECMAScript
2. 환경구성: Polyfill, Babel, Webpack
3. 변수: let, const
4. Function - 디폴트 파라미터
5. Arrow function
6. Object 표현식
7. Destructuring
8. Rest parameter / Spread operator
9. Template Literals / Tagged Template
10. Class
11. Modules
12. Promise
13. Async/Await
14. Fetch API

ECMAScript

- 제안된 스펙이 '표준'이 되기 위해선, 다음의 5단계를 거쳐 리뷰 된다.
- 제안에 대한 논의는 기술위원회인 TC39를 통해 결정된다.

Stage	Description
Stage 0	Strawman - 논의단계
Stage 1	Proposal - 제안단계
Stage 2	Draft - 초안단계
Stage 3	Candidate - 후보단계
Stage 4	Finished - 완성단계

https://tc39.github.io/process-document/

ECMAScript

- 제안된 표준이 완성되면, 어떤 릴리스에 포함될지 확인할 수 있다.
- 대체로 브라우저 벤더들에서 새로운 제안이 많이 이뤄지며, 때에 따라 이미 브라우저에 '구현'되어 있을 수도 있다.
- 당장 많은 브라우저에서 native 지원되지 않더라도 transpiler를 통해 바로 사용할 수도 있다.

Finished Proposals

Finished proposals are proposals that have reached stage 4, and are included in the latest draft of the specification.

Proposal	Author	Champion(s)	TC39 meeting notes	Expected Publication Year
`Array.prototype.includes`	Domenic Denicola	Domenic Denicola Rick Waldron	November 2015	2016
Exponentiation operator	Rick Waldron	Rick Waldron	January 2016	2016
`Object.values` / `Object.entries`	Jordan Harband	Jordan Harband	March 2016	2017
String padding	Jordan Harband	Jordan Harband Rick Waldron	May 2016	2017

https://github.com/tc39/proposals/blob/master/finished-proposals.md

목차

1. ECMAScript
2. 환경구성: Polyfill, Babel, Webpack
3. 변수: let, const
4. Function - 디폴트 파라미터
5. Arrow function
6. Object 표현식
7. Destructuring
8. Rest parameter / Spread operator
9. Template Literals / Tagged Template
10. Class
11. Modules
12. Promise
13. Async/Await
14. Fetch API

Polyfill (폴리필)

Polyfill은 코드 조각(또는 플러그인) 등을 통해, 브라우저에서 native하게 지원되지 않는 기능을 해당 브라우저에서 사용 가능한 코드로 구현하는 것을 말한다.

- " 문자열 ".trim(); 은 문자열의 앞뒤 공백 문자를 제거해 준다.
- .trim()을 지원하지 않는 브라우저에서는 아래 코드를 통해 동일한 메서드를 사용할 수 있게 된다.

```
if (!String.prototype.trim) {
  String.prototype.trim = function() {
    return this.replace(/^[₩s₩uFEFF₩xA0]+|[₩s₩uFEFF₩xA0]+$/g, "");
  };
}
```

Polyfill 이란?
→ https://remysharp.com/2010/10/08/what-is-a-polyfill

Babel

트랜스파일러는 특정 syntax를 낮은 버전의 syntax로 '변환'해 준다.

```
// ES6 arrow function code
var sum = (num1, num2) => num1 + num2;

// transpiled to ES5
var sum = function(num1, num2) { return num1 + num2; }
```

장점:

- 개발자는 최신 syntax를 사용하더라도, transpiler를 통해 코드를 여러 브라우저 환경에서 실행되는 코드로 변환할 수 있기 때문에, 브라우저 버전의 중요성이 감소하게 된다.

단점:

- 변환 작업을 위한 도구의 설정 및 매번 변환(Transpiling/Compile) 단계를 거쳐야 한다.
- '변환'된 코드는 작성된 코드와는 다르기 때문에 디버깅과 코드의 가독성이 어려워 질 수 있다.

Babel 설치

아래의 명령어를 통해 기본적으로 필요한 Babel 패키지들을 설치한다.

```
$ npm install --save-dev @babel/core @babel/cli @babel/preset-env
```

- **@babel/core**: 기본 Babel 코어 라이브러리
  → https://babeljs.io/docs/en/babel-core

- **@babel/cli**: CLI 도구
  → https://babeljs.io/docs/en/babel-cli

- **@babel/preset-env**: 기 정의된 변환할 ES6+ syntax 플러그인 셋
  → https://babeljs.io/docs/en/babel-preset-env

Babel 설정파일

babel.config.js 파일을 생성한다.

```
module.exports = {
    presets: [
        [
            "@babel/preset-env", {
                targets: {  // 변환될 코드가 실행될 환경(target)을 지정
                    browsers: [
                        "last 2 versions",  // 마지막 2개 버전까지
                        "ie >= 9",  // ie는 IE9+
                        "iOS >= 8"  // iOS는 iOS8+
                    ]
                },

                // https://babeljs.io/docs/en/babel-preset-env#usebuiltins-usage-experimental
                // polyfill 사용이 필요한 경우, polyfill을 import로 추가한다.
                // usage로 지정하는 경우, 코드에서 import "@babel/polyfill"할 필요 없다.
                useBuiltIns: "usage"
            }
        ]
]};
```

→ https://babeljs.io/docs/en/babel-preset-env#docsNav

Babel 설정파일

package.json 파일에 ‘scripts’ 항목을 추가하고 ‘compile’ task에 babel 컴파일을 수행할 명령어를 추가한다.

```
{
  "name": "babel",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
      "compile": "babel src -d dist"
   },
   …
}
```

Babel 실습

1) src 폴더내에, arrow-func.js로 아래의 코드를 저장한다.

```
const sum = (val1, val2) => val1 + val2;

console.log(sum(5, 5));
```

2) 아래와 같은 명령어를 수행해 transpiler를 수행한다.

```
$ npm run compile
```

3) dist 폴더에서 결과 값을 확인해 본다.

Webpack

다음의 명령어를 통해 webpack 패키지를 설치한다.

- webpack 패키지는 코어 패키지
- webpack-cli 패키지는 CLI에서 webpack을 실행하는 도구이다.

```
$ npm install webpack webpack-cli --save-dev
```

Webpack 설정파일

webpack.config.js 파일을 통해, webpack 설정을 기술한다.

```
module.exports = {
    entry: "./src/index.js",
    output: {
        path: __dirname + "/dist",
        filename: "webapp.bundle.js",
        publicPath: "dist"
    },
    module: {
        rules: [
            { test: /\.txt$/, use: 'raw-loader' }
        ]
    },
    plugins: [
        new HtmlWebpackPlugin({template: './src/index.html'})
    ],
   devtool: "cheap-module-source-map",  // source-map 파일 생성
};
```

→ 웹팩 설정문서: https://webpack.js.org/configuration/

Webpack 설정파일

```
module.exports = {
    entry: "./src/index.js",
    output: { … },
    module: { … },
    plugins: [ … ],
    devtool: "cheap-module-source-map",  // source-map 파일 생성
};
```

- **entry**: 번들링을 위한 entry point 파일의 위치를 기술한다.
- **output**: 번들링의 결과가 어느 위치에, 어떤 이름으로 저장할지를 지정한다.
- **loaders:** 별도의 로더(패키지)를 통해, entry를 통해 포함되는 원래의 코드에서 다른 형태의 코드로 변환하는 등의 작업을 한다.
- **plugins:** 번들링된 코드의 결과에 대한 추가 작업을 수행한다. 플러그인 종류에 따라 추가 작업은 달라질 수 있다.
- **debugging**: 번들링된 파일 디버깅을 위해, 결과 output 파일과 original 코드를 매핑해 주는 파일인 source-map 파일을 생성한다.

Webpack 실행

package.json 파일에 ‘scripts’ 항목에 ‘build’ task에 webpack 명령어를 추가한다.

```
{
  "name": "babel",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
     "compile": "babel src -d dist",
     "build": "webpack"
   },
   …
}
```

아래의 명령어를 통해 webpack을 통한 번들링을 실행한다.

```
$ npm run build
```

Webpack에 babel 설정하기

- webpack을 사용해, babel을 사용해 transpiling 되도록 하기 위해선, 관련 로더 추가가 필요
- babel은 webpack에서 사용할 수 있는 'babel-loader' 패키지를 제공한다.

```
$ npm install babel-loader --save-dev
```

```
// webpack.config.js
module.exports = {
    ...
    module: {
        rules: [
            {
                test: /₩.m?js$/,
                exclude: /(node_modules)/,
                use: {
                    loader: "babel-loader",
                    options: {
                        presets: ["@babel/preset-env"]
                    }
                }
            }
        ]
    }
};
```

- webpack 설정 파일에 babel 로더가 코드를 변환하도록 설정을 추가한다.
- 'npm run build'를 수행하면 번들링 파일에 트랜스파일된 코드가 생성된다.

개발 서버: watch 기능

- 개발 서버를 사용하면 코드가 수정되면, 감시(watch)하고 있다가 자동으로 번들링을 수행한다.
- 개발 진행중인 상태에서 매번 코드를 수정하고 컴파일 하고 확인하는 번거로움을 덜어준다.

```
$ npm install webpack-dev-server --save-dev
```

기본적으로 webpack.config.js의 설정을 사용한다.
별도의 개발서버 설정이 필요한 경우, 아래와 같이 'devServer' 항목을 통해 가능하다.

```
module.exports = {
    entry: "./src/index.js",
    …
    // https://webpack.js.org/configuration/dev-server/#devserver
    devServer: {
        contentBase: "./dist",
        compress: true,
        port: 9000
    },
};
```

개발 서버: watch 기능

- package.json에 개발서버 실행 명령어를 추가한다.
- ‘npm run start’를 실행한다.

```
{
  …
  "scripts": {
     "start": "webpack-dev-server --open"
```

개발/배포 버전 번들링

webpack은 'mode' 옵션을 통해, 개발 또는 배포(압축) 버전의 번들링을 지원한다.

- development: 개발 버전에 최적화된 번들링 파일 생성
- production: 배포 버전에 최적화된 번들링 파일 생성
- none: 아무런 최적화도 수행하지 않음

```
// webpack.config.js
module.exports = {
  …
  mode: "none" | "development" | "production" (default)
};
```

개발/배포 버전 번들링을 위해 아래와 같이 명령어를 구분해 구성한다.

```
// package.json
{
  "scripts": {
      "build": "webpack",
      "build:prod": "webpack --mode production"
  },
  …
}
```

개발/배포 버전 번들링

webpack 설정 파일을 아래와 같이 변경해, 파라미터를 전달받아 처리하도록 한다.

```
// webpack.config.js
var path = require("path");

module.exports = function(env, arg) {
    ...
    mode: arg.mode || "none",
}
```

이후, 각각 다음을 실행을 통해 개발 버전과 배포 버전 번들링이 되는지 확인한다.

```
# 개발버전 - 파일은 압축되지 않아 디버깅에 용이 하다.
$ npm run build

# 배포 버전 - 파일은 minify(압축) 된다.
$ npm run build:prod
```

CLI 스크립트 환경 변수

OS 환경에 따라, 환경 변수(environment var)의 사용이 각각 다른데, 이 경우 cross-env와 같은 패키지를 설치해 해결할 수 있다.

```
$ npm install cross-env --save-dev
```

```
// package.json
"scripts": {
   "build": "cross-env NODE_ENV=production webpack --config build/webpack.config.js"
 }


// webpack.config.js
{
   ...
   mode: process.env.NODE_ENV
}
```

목차

1. ECMAScript
2. 환경구성: Polyfill, Babel, Webpack
3. 변수: let, const
4. Function - 디폴트 파라미터
5. Arrow function
6. Object 표현식
7. Destructuring
8. Rest parameter / Spread operator
9. Template Literals / Tagged Template
10. Class
11. Modules
12. Promise
13. Async/Await
14. Fetch API

실습 파일 다운로드

→ 다운로드 주소는 수업에서 공지

새로운 변수선언자: let

let 선언자는 var 선언자와 같지만, 스코프는 선언된 블럭내에서만 유효성을 갖는다.

```
if (condition) {
        let value = "blue";

        return value;
    } else {
        // 이 블럭내에서 value 변수는 존재하지 않음
    }
}
```

https://leanpub.com/understandinges6/read

새로운 변수선언자: const

const 선언자는 상수 값에 대한 선언자로, 스코프는 선언된 블럭내에서만 유효성을 갖으며 값의 재 할당이 불가능하다.

```
const max = 5;

max = 6;      // 오류를 발생

const config = {
    value1: "abcd",
    value2: 3
};

config.value2 = 24;  // 오류 발생되지 않음
```

목차

1. ECMAScript
2. 환경구성: Polyfill, Babel, Webpack
3. 변수: let, const
4. Function - 디폴트 파라미터
5. Arrow function
6. Object 표현식
7. Destructuring
8. Rest parameter / Spread operator
9. Template Literals / Tagged Template
10. Class
11. Modules
12. Promise
13. Async/Await
14. Fetch API

Function: 디폴트 파라미터

함수 정의 시, 전달받을 파라미터 들의 기본값 정의가 가능하다.

```
function makeRequest(url, timeout, callback) {
    timeout = timeout || 2000;
    callback = callback || function() {};

     ...
}

function makeRequest(url, timeout = 2000, callback = function() {}) {
    timeout;
    callback;

     ...
}
```

목차

1. ECMAScript
2. 환경구성: Polyfill, Babel, Webpack
3. 변수: let, const
4. Function - 디폴트 파라미터
5. Arrow function
6. Object 표현식
7. Destructuring
8. Rest parameter / Spread operator
9. Template Literals / Tagged Template
10. Class
11. Modules
12. Promise
13. Async/Await
14. Fetch API

Arrow function (화살표 함수)

Arrow function은 "=>" 문법을 사용해 정의되는 함수 정의이다.

```
// 일반 함수 정의
function sum(val1, val2) {
    return val1 + val2;
}

// Arrow function을 사용한 경우
const sum = (val1, val2) => val1 + val2;

const double = val => val * 2;  // 파라미터가 한 개인 경우, 괄호 생략 가능
const getName = () => "John Doe";  // 파라미터가 없는 경우

// 블록을 사용하는 경우
const replaceValue = str => {
    str =+ " string";

    return str;
}
```

Arrow function vs Function #1

Arrow function은 심플한 문법을 통해 간단하게 함수를 정의할 수 있지만,
일반 함수와 다른 몇 가지 제약 사항들이 존재한다.

this, super, arguments 및 new.target는 arrow function이 속해 있는
가까운 lexical(어휘적) 스코프(선언된 시점의)를 따른다.

```
var handler = {
    id: "123456",
    init1: function() {
        document.addEventListener("click", function(e) { this.doSomething(e.type); }, false);
    },

    init2: function() {
        document.addEventListener("click", e => this.doSomething(e.type), false);
    },

    doSomething: function(type) {
        console.log("Event " + type  + " fired. Id is " + this.id);
    }
};

handler.init1();  // error
handler.init2();
```

Arrow function vs Function #2

- new 키워드를 사용해 인스턴스를 생성할 수 없다.
- 인스턴스가 생성되지 않으므로, prototype 또한 존재하지 않는다.

```
var name = "Donovan";
var getName = () => this.name;

new getName();  // error

// this가 변경되지 않으므로, 'Donovan'을 반환
getName.bind({
    name: "Michel"
})();
```

Arrow function vs Function #3

- this 키워드는 바인딩을 통해 다른 값으로 변경될 수 없다.
- arguments 인자 객체를 사용할 수 없다.

```
function sum1() {
    return arguments[0] + arguments[1];
}

var sum2 = () => arguments[0] + arguments[1];

sum1(1, 2);  // 3
sum2(1, 2);  // error
```

목차

1. ECMAScript
2. 환경구성: Polyfill, Babel, Webpack
3. 변수: let, const
4. Function - 디폴트 파라미터
5. Arrow function
6. Object 표현식
7. Destructuring
8. Rest parameter / Spread operator
9. Template Literals / Tagged Template
10. Class
11. Modules
12. Promise
13. Async/Await
14. Fetch API

Object 표현식: 속성

변수명이 key명과 같은 경우, key 값을 생략해 표현할 수 있다.

```
let name: "John Doe",
let age: 32;

const person = {
    "name" : name,
    "age": age
};

// key를 생략할 수 있다.
const person = {
    name, age
};
```

Object 표현식: 속성

대괄호([])를 통해 변수를 지정하면, key명으로 변수의 값을 지정할 수 있다.

```
var name = "middle";

var person = {
    "surname": "Doe",
    [name]: "John"
};

console.log(person.middle);  // John
```

Object 표현식: 함수

function 키워드를 생략할 수 있다.

```
var person = {
    name: "Nicholas",
    sayName: function() {
        console.log(this.name);
    }
};

var person = {
    name: "Nicholas",
    sayName() {
        console.log(this.name);
    }
};
```

목차

1. ECMAScript
2. 환경구성: Polyfill, Babel, Webpack
3. 변수: let, const
4. Function - 디폴트 파라미터
5. Arrow function
6. Object 표현식
7. Destructuring
8. Rest parameter / Spread operator
9. Template Literals / Tagged Template
10. Class
11. Modules
12. Promise
13. Async/Await
14. Fetch API

Destructuring

Destructing은 특정 변수의 값을 원하는 형태의 변수에 나누어 할당할 수 있는 방법을 제공한다.

Object의 경우:

```
let person = {name: "John Doe", age: 32};

// 객체로 부터 각 요소들을 별도의 변수에 할당하는 경우
let name = person.name;
let age = person.age;

// destructing을 통해 변수를 할당하는 경우
// 변수명과 같은 key를 갖는 값이 person 객체에 존재해야 한다.
// let {a, b} = person → a와 b는 undefined가 된다.
let { name, age } = person;

// destructing 변수만 선언할 수는 없다.
let { name, age };

// 값은 순차적으로 할당되며, 값이 없는 경우 sex는 undefined가 된다.
let { name, age, sex } = person;

// 새로운 변수명으로 할당하는 경우. aliasName과 aliasAge 변수에 할당된다.
let {name: aliasName, age: aliasAge} = person;
```

Destructuring

Array의 경우:

```
let colors = [ "red", "green", "blue" ];

let [ firstColor, secondColor ] = colors;

console.log(firstColor);        // "red"
console.log(secondColor);       // "green"

// 값의 swapping
let a = 1;
let b = 2;

[ a, b ] = [ b, a ];

// 변수 초기화를 같이 포함할 수도 있다.
let animals = [ "cow", "tiger" ];
let [ first, second, third = "dog" ];

first;  // cow
second;  // tiger
third;  // dog
```

목차

1. ECMAScript
2. 환경구성: Polyfill, Babel, Webpack
3. 변수: let, const
4. Function - 디폴트 파라미터
5. Arrow function
6. Object 표현식
7. Destructuring
8. Rest parameter / Spread operator
9. Template Literals / Tagged Template
10. Class
11. Modules
12. Promise
13. Async/Await
14. Fetch API

Rest parameter

'...'(3개의 점)으로 표현되는 파라미터로, 배열 형태로 '**나머지**' 파라미터들이 포함되어 전달된다.

```
function checkArgs(...args) {
    console.log(args.length);
    console.log(arguments.length);
    console.log(args[0], arguments[0]);
    console.log(args[1], arguments[1]);
}

checkArgs("a", "b");

// → 2
// → 1
// → a a
// → b b

// rest parameter 뒤에 parameter 기술은 허용되지 않는다.
function sum(val1, ...val, val3) { ... }
```

Spread operator

'...'(3개의 점)으로 표현되는 '**펼침**' 인자값으로, 개별적인 인자로 호출되는 대신 배열의 형태로 전달해 호출한다.

```
let value1 = 25;
let value2 = 50;

Math.max(value1, value2);

// spread operator로 인자값 전달시
Math.max(...[value1, value2]);
```

목차

1. ECMAScript
2. 환경구성: Polyfill, Babel, Webpack
3. 변수: let, const
4. Function - 디폴트 파라미터
5. Arrow function
6. Object 표현식
7. Destructuring
8. Rest parameter / Spread operator
9. Template Literals / Tagged Template
10. Class
11. Modules
12. Promise
13. Async/Await
14. Fetch API

Template Literals

- backticks(`)으로 감싸진 문자열로, 다중 줄바꿈과 변수등을 포함하는 문자열 템플릿 표현식
- 변수 및 표현식 등의 결과는 "${변수/표현식}"과 같이 '${ }'를 사용해 표현될 수 있다.

```
let text = `Line1 text
Line2 text
Line3 text`;

console.log(text);  // Line1 text Line2 text Line3 text

let year = 2018;
let month = 11;
let day = 20;

console.log(`Date: ${year}-${month}-${day}`);  // Date: 2018-11-20
```

Tagged Template

- backticks(`)으로 감싸진 template 문자열의 값을 처리하는 함수 tag

```
function tag(literals, ...substitutions) {
  // literals
  // ["The unit price is ", " and the amount is ", ".",]

  // substitutions
  // [10, 2.5]
  return `The total price is $${substitutions[0] * substitutions[1]}.`;
}

const unit = 10;
const price = 0.25;
const message = tag`The unit price is ${unit} and the amount is ${price}.`;

console.log(message);  // The total price is $2.5.
```

목차

1. ECMAScript
2. 환경구성: Polyfill, Babel, Webpack
3. 변수: let, const
4. Function - 디폴트 파라미터
5. Arrow function
6. Object 표현식
7. Destructuring
8. Rest parameter / Spread operator
9. Template Literals / Tagged Template
10. Class
11. Modules
12. Promise
13. Async/Await
14. Fetch API

Class

- 클래스를 사용하기 위한 문법이 존재하지 않던, JavaScript에서 class를 선언하고 사용할 수 있게 한다.
- 그러나 실제로 내부 동작은 prototype과 동일하며, 단순한 문법적 선언이다.
- 강제성은 없으나, 클래스 명은 일반적으로 대문자로 시작하는 것이 일반적

```
// ES5 or less
function Person(name) {
    this.name = name;
}

Person.prototype.sayName = function() {
    console.log(this.name);
};


let person = new Person("Nicholas");
person.sayName();   // "Nicholas"
```

```
// ES6+
class Person {
    constructor(name) {
        this.name = name;
    }

    sayName() {
        console.log(this.name);
    }
}
// 아래와 같은 표현식도 가능하다.
// let Person = class { … };

let person = new Person("Nicholas");
person.sayName();   // "Nicholas"
```

Class 상속

클래스의 상속은 'extends' 키워드를 사용

```
class Rectangle {
    constructor(length, width) {
        this.length = length;
        this.width = width;
    }

    getArea() {
        return this.length * this.width;
    }
}
```

```
// Rectangle 클래스를 상속
class Square extends Rectangle {
    constructor(length, width) {

        // Rectangle.call(this, length, width)과 동일
        super(length, width);
    }
}

var square = new Square(3);
console.log(square.getArea());  // 9
```

Class static method

- 정적 메서드는 'static' 키워드를 사용해 정의한다.
- 정적 메서드는 인스턴스를 생성하지 않고 사용될 수 있는 메서드이다.

```
class Person {
    constructor(name) {
        this.name = name;
    }

    sayName() {
        console.log(this.name);
    }

    static factory(name) {
        return new Person(name);
    }
}

let person = Person.factory("Nicholas");
person.sayName();  // Nicholas
```

Class accessor 속성

- accessor 속성을 사용하면, 값을 얻을 때(getter)와 설정할 때(setter) 필요한 작업을 하도록 구성할 수 있다.
- Getter는 "**get**" 키워드로, setter는 "**set**" 키워드를 지정한다.

```
class Person {
    constructor(value) {
        this.value = value;
    }

    get name() {
        console.log("getter called");
        return this.value;
    }

    set name(value) {
        console.log("setter called");
        this.value = value;
    }
}

var person = new Person("Hong");
person.name;  // getter called, Hong
person.name = "Kim";  // setter called, Kim
```

목차

1. ECMAScript
2. 환경구성: Polyfill, Babel, Webpack
3. 변수: let, const
4. Function - 디폴트 파라미터
5. Arrow function
6. Object 표현식
7. Destructuring
8. Rest parameter / Spread operator
9. Template Literals / Tagged Template
10. Class
11. Modules
12. Promise
13. Async/Await
14. Fetch API

Modules: basic

Module은 JavaScript 코드를 모듈 단위로 분리하고 사용할 수 있게 한다.

- 모듈을 불러들이는 측에선 "import" 키워드를 사용
- 모듈로 내보내는 측에선 "export" 키워드를 사용

```
// A.js
// 함수를 export
export function sum(num1, num2) {
    return num1 + num1;
}

// B.js
// A.js에서 expor된 sum을 import
import { sum } from "./A.js";

sum(10, 20);  // 30
```

Modules: basic

<script> 태그를 통해 사용되는 경우

```
<!-- load a module JavaScript file -->
<script type="module" src="module.js"></script>

<!-- include a module inline -->
<script type="module">
    import { sum } from "./example.js";

    let result = sum(1, 2);
</script>
```

Modules: export

변수, 함수, 클래스 등의 단위로 export 할수 있다.

```
export var color = "red";  // export data
export function sum() { ... }
export class Person { ... }

// 또는 정의된 값들을 나중에 export 할수도 있다.
var name = "John Doe";
function abcd() { ... };

export {name as name2, abcd};

// default export는 이름을 지정할 필요가 없다.
export default function(num1, num2) {
    return num1 + num2;
}
```

Modules: import

import는 export 요소들을 바인딩 해 사용한다.

```
// single biding import
import { sum } from "./A.js";

// multiple binding import
import { sum, division, substraction } from "./A.js";

// rename binding import
import { sum as calc } from "./A.js";

// namespace import
import * as util from "./A.js";

// 바인딩 없이 import 하는 경우는
// 전역 영역에서 실행되는 코드를 포함하고자 하는 경우이다.
import "./A.js";
```

Modules: re-export

Import된 값을 다시 export 할수도 있다.

```
// import를 다시 export 하는 경우
import { sum } from "./example.js";
export { sum }

// import를 사용하지 않고 바로 export 할수도 있다.
export { sum } from "./example.js";
export * from "./example.js";
```

Modules: default export

default export는 해당 모듈의 '기본' export를 지정할 때 사용한다.
'기본'이기 때문에, 별도의 name을 지정할 필요가 없다.

```
// A.js
export default function(num1, num2) {
    return num1 + num2;
}

// B.js
import sum from "./A.js"

// named export와 default export가 포함된 경우
export let color = "red";
export default function() { ... }

import sum, { color } from "./A.js";
```

목차

1. ECMAScript
2. 환경구성: Polyfill, Babel, Webpack
3. 변수: let, const
4. Function - 디폴트 파라미터
5. Arrow function
6. Object 표현식
7. Destructuring
8. Rest parameter / Spread operator
9. Template Literals / Tagged Template
10. Class
11. Modules
12. Promise
13. Async/Await
14. Fetch API

Promise

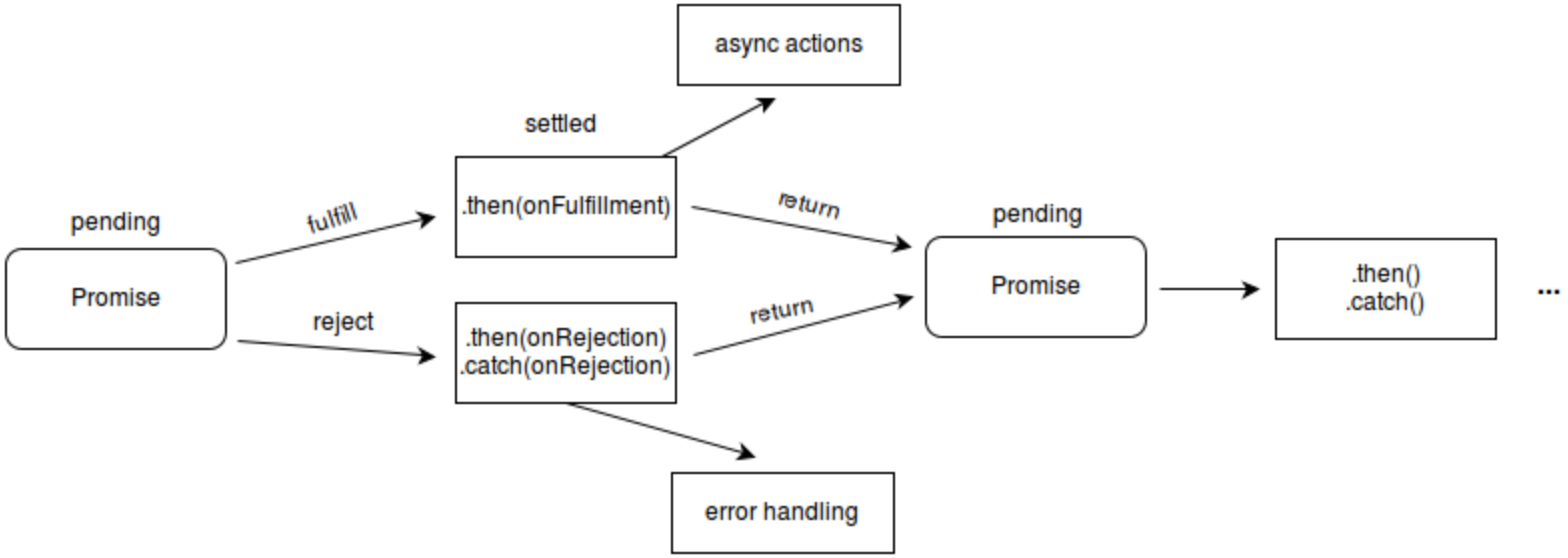
- 전통적인 비동기 처리 방식인 구독(subscription) 또는 콜백(callback) 함수 등의 코드 작성 형태로 인한 복잡성을 제거하는 방법을 제공한다.
- Promise로 함수를 감싸, promise 객체를 생성한다. Promise로 감싸지 않아도 Promise를 반환하는 API들도 있다. (ex. Fetch API)

```
// Promise로 전달되는 함수는 executor라 한다.
new Promise(function(resolve, reject) {
  …
  // 작업이 성공한 경우
  if (success) {
    resolve(someValue); // fulfilled

  // 작업이 실패한 경우, reject를 호출
  } else {
    reject("failure reason"); // rejected
  }
})
// 작업이 성공한 경우
.then(function(value) { … }, function(value) { … })
// 오류가 발생한 경우 - 이전 체이닝에서 rejection 함수가 없는 경우 수행
.catch(function(msg) { … });
```

Promise

Promise는 다음 중 하나의 상태를 갖는다.
- pending(대기): 이행되거나 거부되지 않은 초기 상태
- fulfilled(이행): 연산이 성공적으로 완료된 상태
  → .then()이 수행
- rejected(거부): 연산/작업이 실패한 상태



Promise: Babel Polyfill

- @babel/polyfill 패키지의 설치가 필요하다.
  → https://babeljs.io/docs/en/babel-polyfill

번들링 파일에 포함되어야 하므로, dependency 항목에 설치되어야 한다.

```
npm install --save @babel/polyfill
```

Entry point 파일 상단에 import 되도록 해, polyfill이 애플리케이션 코드보다 먼저 실행 되게 한다.

```
// index.js
import "@babel/polyfill";

new Promise( ... );
```

Promise

- 비동기 작업에 대해 'Promise'를 반환, 작업의 완료/오류 발생에 대해 정돈된 형태의 코드를 작성할 수 있다.

```
// 콜백의 경우
let promise = readFile("example.txt", function(contents) {
    console.log(contents);
}, function(err) {
    console.log(err.message);
});

// Promise를 사용하는 경우
let promise = readFile("example.txt");  // Promise를 반환

promise.then(function(contents) {       // ← resolved callback
    console.log(contents);
}, function(err) {                      // ← rejected callback
    console.error(err.message);
})
.then( … ).then(…).catch(…);
```

Promise.race()

인자로 주어진 Promise들 중 빨리 resolve/reject 되는 쪽을 사용

```
var p1 = new Promise(function(resolve, reject) {
    setTimeout(reject, 500, "one");
});
var p2 = new Promise(function(resolve, reject) {
    setTimeout(reject, 100, "two");
});

Promise.race([p1, p2]).then(function(value) {
    console.log(value); // "two"
}, function(msg) {
    console.log(msg);
});
```

Promise.all()

- 여러 개의 Promise들의 resolve 상태를 한번에 처리
- 주어진 것들 중, 어느 한 곳에서 reject 발생시 즉시 reject 된다.

```
var promise1 = Promise.resolve(3);
var promise2 = 42;
var promise3 = new Promise(function(resolve, reject) {
                setTimeout(resolve, 100, 'foo');
});

Promise.all([promise1, promise2, promise3])
    .then(function(values) {
        console.log(values);  // 3, 42, "foo"
    });
```

Promise.resolve() / .reject()

- .resolve()는 resolve 되는 Promise를 반환
- .reject()는 reject 되는 Promise를 반환

```
Promise.resolve("성공").then(function(value) {
    console.log(value);  // "성공"
});

Promise.reject("거부").then(function(reason) {
    // 호출되지 않음
}, function(reason) {
    console.log(reason);  // "거부"
});
```

목차

1. ECMAScript
2. 환경구성: Polyfill, Babel, Webpack
3. 변수: let, const
4. Function - 디폴트 파라미터
5. Arrow function
6. Object 표현식
7. Destructuring
8. Rest parameter / Spread operator
9. Template Literals / Tagged Template
10. Class
11. Modules
12. Promise
13. Async/Await
14. Fetch API

Async/Await

- ES2016/ES7에 속한 스펙이다.
- Async/Await는 비동기 처리를 위해 함수 선언문과 같이 사용된다.
- 비동기 처리를 순차적 코드로 표현되며, 코드의 가독성과 흐름의 파악이 쉽다.

```
async function getAPI() {
    // 비동기 호출
    let data = await some-async-call();

    console.log(data);
}
```

async 함수 선언자

- async는 비동기 처리를 포함하는 함수를 선언할 때 사용되며, Promise를 반환한다.
- 여러 promise의 작업들을 동기스럽게(synchronous) 사용할 수 있도록 한다.

```
async function getAPI() {
    …
}
```

await 선언자

- await는 기다릴 Promise 값이거나, 또는 그외 값을 지정할 수 있다.
- await는 async 함수 내에서만 사용 가능하다.
- await가 지정된 경우, Promise가 작업을 완료(fulfill) 될때까지 기다린 후, 완료되면 값을 반환한다.
- Promise가 아닌 경우라면, 해당 값은 resolved된 Promise가 반환된다.

```
async function getAPI() {
    // 비동기 호출
    let data = await some-async-call();
    let value = await 50;  // resolved Promise를 반환

    console.log(data);
}
```

Async/Await 예제

- await는 기다릴 Promise 값이거나, 또는 그외 값을 지정할 수 있다.

```
function resolveAfter(wait) {
  return new Promise(resolve => {
      setTimeout(() => {
          resolve(`resolved after ${wait}ms.`);
      }, wait);
  });
}

async function asyncCall() {
    let result = await resolveAfter(2000);

    console.log(result);  // resolved after 2000ms.
}

asyncCall();
```

목차

1. ECMAScript
2. 환경구성: Polyfill, Babel, Webpack
3. 변수: let, const
4. Function - 디폴트 파라미터
5. Arrow function
6. Object 표현식
7. Destructuring
8. Rest parameter / Spread operator
9. Template Literals / Tagged Template
10. Class
11. Modules
12. Promise
13. Async/Await
14. Fetch API

Fetch API

- Fetch API는 Promise를 반환하는 XMLHttpRequest(XHR) 통신을 위한 기능을 제공한다.
- 일반적인 XHR의 중첩된 콜백으로 인한 callback hell 이슈를 해결할 수 있다.
- ES6의 명세에 포함되지 않으며, HTML의 living standard 명세에 속한다.

```
fetch(url, {
    method: "POST",
    body: JSON.stringify(data),
    headers: {
      "Content-Type": "application/json"
    },
    credentials: "same-origin"
}).then(function(response) {
    response.status      //=> number 100–599
    response.statusText //=> String
    response.headers     //=> Headers
    response.url         //=> String

    return response.text()
}, function(error) {
    error.message //=> String
});
```

Fetch API

Fetch API와 async/await를 같이 사용하면, 아래와 같이 비동기 XHR 호출의 처리를 아주 단순하게 그리고 우아한 방식으로 처리할 수 있다.

```
async function() {
  let a = await fetch("./some-api");

  // XHR을 통해 반환된 값
  console.log(await a.text());
}
```

Fetch API: Polyfill

Fetch는 대다수의 모던 브라우저에서 지원되나, IE 지원을 위해선 polyfill이 필요하다.
→ https://caniuse.com/#feat=fetch

번들링 파일에 포함되어야 하므로, dependency 항목에 설치되어야 한다.

```
npm install whatwg-fetch --save
```

Entry point 파일 상단에 import 되도록 해, polyfill이 애플리케이션 코드보다 먼저 실행 되게 한다.

```
// index.js
import "whatwg-fetch";

fetch("some-url", function(response) { ... })
    .then( ... );
```

고맙습니다.