ES6(ECMAScript 표준의 6번째 에디션, ECMAScript2015)에 대한 이야기를 하기 전에 자바스크립트와 ECMAScript에 대한 것부터 간략히 소개한다.

넷스케이프(Netscape)에서 1995년 개발한 자바스크립트(javascript)는 웹 브라우저에서 동적인 기능을 제공하기 위한 언어다. 현재는 대부분의 브라우저에서 이 언어를 제공하고 있다. 그런데 표준 규격없이 여러 브라우저에서 독자적인 특성이 추가되면서 호환성 문제가 발생하기 시작했다. 이에 ECMA 국제 기구에서 "ECMAScript Standard"라는 표준을 만들게 되었다. 정확히 이야기 하자면 현재의 자바스크립트는 ECMAScript와 BOM(Browser Object Model)와 DOM(Document Object Model)을 포괄하는 개념이다.

개별적인 설명에 앞서 개발자가 필히 알아야 할 ES6 10가지 기능을 나열하자면 아래와 같다.

- 기본 매개 변수 (Default Parameters)
- 멀티 라인 문자열 (Multi-line Strings)
- 비구조화 할당 (Destructuring Assignment)
- 향상된 객체 리터럴 (Enhanced Object Literals)
- 화살표 함수 (Arrow Functions)
- Promises
- 블록 범위 생성자 Let 및 Const (Block-Scoped Constructs Let and Const)
- 클래스 (Classes)
- 모듈 (Modules)

이제부터 하나씩 알아보자.

1. 기본 매개 변수 (Default Parameters)

```
var link = function (height, color, url) {
    var height = height || 50
    var color = color || 'red'
    var url = url || 'http://azat.co'
    ...
}
```

함수에 넘겨주는 인자값에 대한 default 처리를 위해 위와 같이 처리 했었다면 ES6에서는 아래와 같이 간단히 처리할 수 있다.

```
var link = function(height = 50, color = 'red', url = 'http://azat.co') {
  ...
}
```

단, 주의해야 할 점이 있다. 인자값으로 0 또는 false가 입력될 때 두 예시의 결과는 다르다. ES5에서는 || 처리 시 0 또는 false 값이 입력 되어도 거짓이 되므로 기본값으로 대체된다. 하지만 ES6의 기본 매개 변수를 사용하면 undefined 를 제외한 입력된 모든 값(0, false, null 등)을 인정한다.

2. 템플릿 리터럴 (Template Literals)

ES5에서는 아래와 같이 문자열을 처리해야 했다.

```
var name = 'Your name is ' + first + ' ' + last + '.'
var url = 'http://localhost:3000/api/messages/' + id
```

하지만 ES6에서는 템플릿 리터럴을 제공하므로 "`" (back-ticked) 문자열 안에 ${NAME}라는 새로운 구문을 사용해서 아래와 같이 간단히 처리할 수 있다.

var name = `Your name is ${first} ${last}.` var url = `http://localhost:3000/api/messages/${id}`

3. 멀티 라인 문자열 (Multi-line Strings)

ES5에서는 멀티 라인 문자열을 처리하기 위해 아래와 같은 방법들을 사용해야 했다.

```
var roadPoem = 'Then took the other, as just as fair,\n\t'
    + 'And having perhaps the better claim\n\t'
    + 'Because it was grassy and wanted wear,\n\t'
    + 'Though as for that the passing there\n\t'
    + 'Had worn them really about the same,\n\t'

var fourAgreements = 'You have the right to be you.\n\
    You can only be you when you do your best.'
```

하지만 ES6에서는 "`" (back-ticked) 문자열을 이용해서 아래와 같이 간단히 처리할 수 있다.

```
var roadPoem = `Then took the other, as just as fair,
    And having perhaps the better claim
    Because it was grassy and wanted wear,
    Though as for that the passing there
    Had worn them really about the same,`

var fourAgreements = `You have the right to be you.
    You can only be you when you do your best.`
```

4. 비구조화 할당 (Destructuring Assignment)

ES5에서는 구조화된 데이터를 변수로 받기 위해 아래와 같이 처리해야 했다.

```
// browser
var data = $('body').data(), // data has properties house and mouse
  house = data.house,
  mouse = data.mouse

// Node.js
var jsonMiddleware = require('body-parser').json

var body = req.body, // body has username and password
  username = body.username,
  password = body.password
```

하지만 ES6에서는 비구조화 할당을 사용해 아래와 같이 처리할 수 있다.

```
var {house, mouse} = $('body').data() // we'll get house and mouse variables

var {jsonMiddleware} = require('body-parser')

var {username, password} = req.body
```

주의할 점은 var로 할당하려는 변수명과 구조화된 데이터의 property명이 같아야 한다. 또한 구조화된 데이터가 아니라 배열의 경우 {} 대신 []를 사용해서 위와 유사하게 사용할 수 있다.

var [col1, col2] = $('.column'), [line1, line2, line3, , line5] = file.split('\n')

5. 향상된 객체 리터럴 (Enhanced Object Literals)

ES5에서는 아래와 같이 JSON을 사용해서 객체 리터럴을 만들 수 있었다.

```
var serviceBase = {port: 3000, url: 'azat.co'},
    getAccounts = function(){return [1,2,3]}

var accountServiceES5 = {
  port: serviceBase.port,
  url: serviceBase.url,
  getAccounts: getAccounts,
  toString: function() {
    return JSON.stringify(this.valueOf())
  },
  getUrl: function() {return "http://" + this.url + ':' + this.port},
  valueOf_1_2_3: getAccounts()
```

```
}
```

위 예시와 달리 serviceBase를 확장하길 원한다면 Object.create 로 프로토타입화하여 상속 받을 수 있다.

```
var accountServiceES5ObjectCreate = {
  getAccounts: getAccounts,
  toString: function() {
    return JSON.stringify(this.valueOf())
  },
  getUrl: function() {return "http://" + this.url + ':' + this.port},
  valueOf_1_2_3: getAccounts()
}
accountServiceES5ObjectCreate.__proto__ = Object.create(serviceBase)
```

accountServiceES5ObjectCreate와 accountServiceES5는 동일하게 사용할 수 있으나 다른 구조를 가진다. accountServiceES5ObjectCreate는 accountServiceES5와 다르게 **proto** 에 port 와 url 속성을 가진 객체를 담고 있다.

ES6에서는 아래와 같이 처리할 수 있다.

```
var serviceBase = {port: 3000, url: 'azat.co'},
    getAccounts = function(){return [1,2,3]}
var accountService = {
    __proto__: serviceBase,
    getAccounts,
    toString() {
     return JSON.stringify((super.valueOf()))
    },
    getUrl() {return "http://" + this.url + ':' + this.port},
    [ 'valueOf_' + getAccounts().join('_') ]: getAccounts()
};
```

위 예시에 대해 ES5와의 차이를 요약하면 아래와 같다.

- **proto** 속성을 사용해서 바로 프로토타입을 설정할 수 있다.
- getAccounts: getAccounts, 대신 getAccounts, 를 사용할 수 있다 (변수명으로 속성 이름을 지정).
- ['valueOf_' + getAccounts().join('_')] 와 같이 동적으로 속성 이름을 정의할 수 있다. 조금 더 자세한 내용을 보고 싶다면 gsfe/es2015features 를 참고하자.

6. 화살표 함수 (Arrow Functions)

화살표 함수는 항상 익명 함수이며 this의 값을 현재 문맥에 바인딩 시킨다.

아래의 예시는 화살표 함수가 지원되지 않는 ES5에서 this를 사용하기 위한 처리 예시다.

```
var _this = this
$('.btn').click(function(event){
  _this.sendData()
})
다음은 위 예시를 화살표 함수로 대체한 ES6 예시이다.

$('.btn').click((event) => {
  this.sendData()
})
```

다음은 ES5에서 call을 사용하여 context를 logUpperCase() 함수에 전달하는 또 다른 예제다.

```
var logUpperCase = function() {
  var _this = this

  this.string = this.string.toUpperCase()
  return function () {
    return console.log(_this.string)
  }
}
logUpperCase.call({ string: 'es6 rocks' })()
```

ES6에서는 화살표 함수를 사용하면 _this 를 사용할 필요가 없다.

```
var logUpperCase = function() {
  this.string = this.string.toUpperCase()
  return () => console.log(this.string)
}
```

logUpperCase.call({ string: 'es6 rocks' })() 화살표 함수가 한 줄의 명령문과 함께 사용되면 표현식이 되어 명령문의 결과를 암시적으로 반환한다.

ES5에서의 처리 예시.

```
var ids = ['5632953c4e345e145fdf2df8','563295464e345e145fdf2df9']
var messages = ids.map(function (value) {
  return "ID is " + value // explicit return
});
```

ES6에서의 처리 예시.

```
var ids = ['5632953c4e345e145fdf2df8','563295464e345e145fdf2df9']
var messages = ids.map(value => `ID is ${value}`) // implicit return
```

여러 개의 인자를 사용하는 경우는 변수 목록을 () 로 감싸줘야 한다.

ES5에서의 처리 예시.

```
var ids = ['5632953c4e345e145fdf2df8', '563295464e345e145fdf2df9'];
var messages = ids.map(function (value, index, list) {
  return 'ID of ' + index + ' element is ' + value + ' ' // explicit return
});
```

ES6에서의 처리 예시.

```
var ids = ['5632953c4e345e145fdf2df8','563295464e345e145fdf2df9']
var messages = ids.map((value, index, list) => `ID of ${index} element is ${value}
`) // implicit return
```

또한 본문을 괄호로 감싸 객체 표현식을 반환할 수 있으며 ... 을 이용해 가변 파라미터를 사용할 수도 있다.

```
var ids = ['5632953c4e345e145fdf2df8','563295464e345e145fdf2df9']
var messages = ids.map((value, index, ...abc) => ({v:value, i:index, a:abc}))
```

7. Promises

ES6에서는 표준 Promise가 제공된다.

아래는 setTimeout 을 이용한 지연된 비동기 실행에 대한 ES5 예시다.

```
setTimeout(function(){
  console.log('Yay!')
}, 1000)
위 예시를 ES6에서 Promise를 사용해서 재작성하면 아래와 같다.

var wait1000 =  new Promise(function(resolve, reject) {
```

```
  setTimeout(resolve, 1000)
}).then(function() {
  console.log('Yay!')
})
```

위 예시를 화살표 함수를 사용해 재작성한 예시는 아래와 같다.

```
var wait1000 =  new Promise((resolve, reject)=> {
  setTimeout(resolve, 1000)
}).then(()=> {
  console.log('Yay!')
})
```

ES5 보다 ES6의 Promise를 사용한 예시가 더 복잡해 보이지만 아래와 같이 중첩된 setTimeout 예시를 보면 Promise의 이점을 확인할 수 있다.

```
setTimeout(function(){
  console.log('Yay!')
  setTimeout(function(){
    console.log('Wheeyee!')
  }, 1000)
}, 1000)
```

아래는 ES6 Promise 로 작성된 예시.

```
var wait1000 =  ()=> new Promise((resolve, reject)=> {setTimeout(resolve, 1000)})
wait1000()
    .then(function() {
        console.log('Yay!')
        return wait1000()
    })
    .then(function() {
        console.log('Wheeyee!')
    });
```

조금 더 자세한 내용을 보고 싶다면 Introduction to ES6 Promises – The Four Functions You Need To Avoid Callback Hell 또는 gsfe/es2015features 를 참고하자.

8. 블록 범위 생성자 Let 및 Const (Block-Scoped Constructs Let and Const)

let과 const는 중괄호("{}")로 정의된 블록으로 유효 범위(스코프)를 지정하는 새로운 var이다. 단, let은 변수를 const는 상수를 선언한다.

```
function calculateTotalAmount (vip) {
  var amount = 0
  if (vip) {
    var amount = 1
  }
  { // more crazy blocks!
    var amount = 100
    {
      var amount = 1000
      }
  }
  return amount
}
console.log(calculateTotalAmount(true))
```

위 예시의 결과는 1000 이다. var는 전역 또는 함수 내부로 유효 범위를 갖기 때문에 예시에 사용된 함수 내부의 "{}" 들은 아무런 역할을 하지 못한다. 아래는 위 예시에서 var를 let으로 바꾼 ES6 예시다.

```
function calculateTotalAmount (vip) {
  var amount = 0 // probably should also be let, but you can mix var and let
  if (vip) {
    let amount = 1 // first amount is still 0
  }
  { // more crazy blocks!
    let amount = 100 // first amount is still 0
    {
      let amount = 1000 // first amount is still 0
      }
  }
  return amount
}
console.log(calculateTotalAmount(true))
```

이 예시의 결과는 0 이다. let 으로 선언된 변수는 "{}" 블록 내부로 유효 범위가 한정되므로 100, 1000으로 할당된 변수는 해당 블록 내부에서만 유효하기 때문이다. if 블록 내부에서 let으로 선언된 amount 또한 해당 if 블록 내에서만 유효하므로 아무런 변경이 일어나지 않는다.

아래의 예시는 const를 사용한 예시다. const는 상수를 선언하는 것으로 여러번 선언될 수 없지만 let과 같이 블록 내부로 유효 범위가 한정되므로 아래의 예시는 오류가 발생하지 않는다.

```
function calculateTotalAmount (vip) {
  const amount = 0
  if (vip) {
    const amount = 1
  }
  { // more crazy blocks!
    const amount = 100
    {
      const amount = 1000
      }
  }
  return amount
}
console.log(calculateTotalAmount(true))
```

9. 클래스 (Classes)

ES6에는 class 키워드가 추가되어 ES5의 prototype 기반 상속보다 명확하게 class를 정의할 수 있다. get 과 set 키워드 외에도 static 키워드를 사용해 static 메소드를 정의하는 것도 가능하다.

```
class baseModel {
  constructor(options = {}, data = []) { // class constructor
        this.name = 'Base'
    this.url = 'http://azat.co/api'
        this.data = data
    this.options = options
    }

    getName() { // class method
        console.log(`Class name: ${this.name}`)
    }
}
```

constructor 는 class 내부에 하나만 존재할 수 있으며 메소드 정의에 function 또는 콜론(":")이 더이상 필요하지 않다. 단, property의 경우 메소드와 달리 생성자에서 값을 할당해야 한다.

또한 아래의 예시와 같이 class NAME extends PARENT_NAME 형식으로 상속이 가능하다. 상속시 부모 생성자를 호출하기 위해 super() 를 사용할 수 있다. 생성자가 아닌 메소드에서는 super 키워드를 사용해서 부모 메소드에 접근한다.

```
class AccountModel extends baseModel {
    constructor(options, data) {
      super({private: true}, ['32113123123', '524214691']) //call the parent
```

```
method with super
      this.name = 'Account Model'
      this.url +='/accounts/'
    }

    get accountsData() { //calculated attribute getter
      // ... make XHR
      return this.data
    }
}
```

class 는 get 과 set 키워드를 사용할 수 있으며 선언된 함수는 아래와 같이 사용할 수 있다.

```
let accounts = new AccountModel(5)
accounts.getName()
console.log('Data is %s', accounts.accountsData)
```

위 예시를 실행하면 아래와 같은 결과를 얻을 수 있다.

```
Class name: Account Model
Data is %s 32113123123,524214691
```

10. 모듈 (Modules)

ES6 에서 모듈을 공식적으로 제공하기 전까지는 CommonJS, AMD, RequireJS 등의 비공식 모듈 스펙을 사용해 왔다. ES6에서 제공하는 모듈 스펙은 기존과 유사하지만 차이가 있다.

ES5에서 CommonJS를 이용해서 모듈을 사용하는 예시는 아래와 같다(module.js).

```
module.exports = {
  port: 3000,
  getAccounts: function() {
    ...
  }
}
```

main.js 파일에서 위에서 정의한 모듈을 불러서 사용하는 예시는 아래와 같다.

```
var service = require('module.js')
```

```
console.log(service.port) // 3000
```

여기서 부터는 ES6의 import 와 export 를 사용해서 유사한 기능을 구현한 예시다(module.js).

```
export var port = 3000
export function getAccounts(url) {
  ...
}
```

main.js 파일에서는 import 를 사용해서 module.js 모듈을 불러올 수 있다.

```
import {port, getAccounts} from 'module'
console.log(port) // 3000
```

위와 유사하지만 export 된 모든 변수를 아래와 같이 하나의 구조화된 데이터로 받을 수도 있다.

```
import * as service from 'module'
console.log(service.port) // 3000
```

ES6 당장 사용할 수 있는 방법 (Babel) ES6는 확정되었지만 아직 모든 브라우저에서 완전하게 지원되지 않는다. 따라서 지금 당장 ES6 사용하고 싶다면 Babel과 같은 컴파일러를 사용해야 한다. Babel은 독립 실행형 도구로 실행하거나 빌드 시스템에서 사용할 수 있다. Grunt, Gulp 및 Webpack 용 Babel 플러그인이 있다.

ES6의 기타 특징 참고로 이 외에도 여러가지 특징이 있으니 관심이 있다면 git.io/es6features를 번역한 ECMAScript 6 Features를 참고하면 된다.