はじめに

今更ですが、ES2015(ES6)で追加された機能や構文に関する備忘録です。
「JSは書けるけどES2015(ES6)はわからないっす...!」といった人達向けの記事です。
入門記事のためイテレータやジェネレータ等のわかりづらかったり、
説明が長くなりそうな機能や構文は割愛しております。

ES2015(ES6)とは

ECMASCriptの6th Editionのこと。ECMAScript 6th editionの6を取ってES6と呼ばれていたが、
2015年に標準化されたため正式名称はES2015になった。

正式名称がES2015ならES6という名称を使うのは間違いなのか

どちらの名称でも問題はない。
ES6の名称の方がエンジニアコミュニティに中では浸透しているらしく、ES6と記載されていることが多い。

詳細は以下を参考。
ES6 or ES2015？〜WEB+DB PRESS Vol.87 ES6特集に寄せて〜

正式名称はES2015のため本記事ではES2015と表記する。

ES2015の対応状況

以下は各ブラウザのES2015の対応状況（ES2015のコードが動くかどうか）。
ECMAScript 6 compatibility table

全てのブラウザで対応しているわけではないため、Babel等のトランスパイラを利用して
ES2015で書いたコードを各ブラウザで動くコードであるES5のコードにトランスパイル（変換）する必要がある。

記事によっては「Babelはコンパイラで ES2015をコンパイルするもの」と書かれていることがあるが、
どちらの言い方でも良い。

詳細は以下を参考（自分が調べて自己解決した記事です）。
Babelは「ES2015をコンパイルするコンパイラ」なのか、それとも「ES2015をトランスパイルするトランスパイラ」 なのか

何故ES2015で書くのか

- 便利な機能、構文が追加され、従来より簡潔かつ明瞭な構文で記述できるようになったから。
- 主要な機能、構文を覚える程度なら学習コストはそこまで高くないから。
- フレームワーク等と違ってすぐに廃れるものではないから。
- ES2015で記述されているコードを理解するため。

ES2015で追加された機能、構文

`let` と `const`

`let` と `const` で変数を宣言できる。
`let` は**再宣言が不可能**な宣言であり、 `const` は**再宣言と再代入が不可能**な宣言。
間違えて変数を二重に定義しまったり、予期せぬ再代入を防ぐことができる。

従来の var での変数宣言

```
var name = 'soarflat';
console.log(name); // => soarflat

name = 'SoarFlat'; // 再代入する
console.log(name); // => SoarFlat

var name = 'SOARFLAT'; // 再宣言をする
console.log(name); // => SOARFLAT
```

let での変数宣言

```
let name = 'soarflat';
console.log(name); // => soarflat

name = 'SoarFlat'; // 再代入する
console.log(name); // => SoarFlat

let name = 'SOARFLAT'; // 再宣言するとエラーになる
```

const での変数宣言

```
const NAME = 'soarflat';
console.log(NAME); // => soarflat

NAME = 'SOARFLAT'; // 再代入するとエラーになる
```

let と const の使い分け

再代入が必要な変数のみ let で宣言し、それ以外は const で宣言する。

var は利用する理由がないため利用しない。

カーリーブラケット {} によるブロックスコープ

カーリーブラケット {} によるブロックスコープが有効。
ブラケット内に `let` か `const` で変数を宣言することによって、そのブラケット内のみで変数が有効になる。
変数の影響範囲を狭めることができる。

```
if (true) {
    var a = 1;
    let b = 2;
    const c = 3;
    console.log(a); // => 1
    console.log(b); // => 2
    console.log(c); // => 3
}

console.log(a); // => 1
console.log(b); // => undefined
console.log(c); // => undefined
```

アロー関数

アロー関数は無名関数の省略形の関数。
従来より簡潔な記述で関数を定義できる。

従来の無名関数

```
var fn = function (a, b) {
  return a + b;
};
```

アロー関数

```
const fn = (a, b) => {
  return a + b;
```

```
};

// 単一式の場合はブラケットやreturnを省略できる
const fn = (a, b) => a + b;

// ブラケットやreturnを省略してオブジェクトを返したい場合は`()`で囲む
const fn = (a, b) => ({ sum: a + b });
```

従来の無名関数と this の参照が異なる

従来の無名関数は関数の呼び方によって this の参照先が異なるが、
アロー関数は、**関数が定義されたスコープ内の this を参照する。**

以下は Counter コンストラクタのメンバ変数 this.count の加算を期待した処理だが、
setTimeout 関数内の this がグローバルオブジェクトを参照しているため、意図しない処理になっている。

```
// グローバルオブジェクト
window.count = 10;

var Counter = function() {
  this.count = 0;
};

Counter.prototype.increment = function() {
  setTimeout(function(){
    // thisはグローバルオブジェクトを参照しているため、window.countに加算される。
    this.count++;
    console.log(this.count); // => 11
  }, 1000);
};

var counter = new Counter().increment();
```

そのため、以下のように別の変数に this を参照させておく必要がある。

```
// グローバルオブジェクト
window.count = 10;
```

```
var Counter = function () {
  this.count = 0;
};

Counter.prototype.increment = function() {
  var self = this;
  setTimeout(function(){
    self.count++;
    console.log(self.count); // => 1
  }, 1000);
};

var counter = new Counter().increment();
```

アロー関数は関数が定義されたスコープ内の `this` を参照するため期待通りの処理になる。
今回の場合はインスタンス自身を参照している。

```
const Counter = function () {
  this.count = 0;
};

Counter.prototype.increment = function() {
  setTimeout(() => {
    this.count++;
    console.log(this.count);  // 1
  }, 1000);
};

const counter = new Counter().increment();
```

Class構文

Class構文はprototypeベースのクラス定義構文の糖衣構文。
従来のprototypeベースの構文より、簡潔かつ明瞭な記述でクラスを定義できる。

prototypeベースの構文

```
var Person = function(name) {
  this.name = name;
};

Person.prototype.sayHello = function() {
  console.log("Hello, I'm " + this.getName());
};

Person.prototype.getName = function() {
  return this.name;
};
```

Class構文

```
class Person {
  constructor(name) {
    this.name = name;
  }
  sayHello() {
    console.log("Hello, I'm " + this.getName());
  }
  getName() {
    return this.name;
  }
}
```

extends でクラスの継承

extends でクラスの継承ができる。

```
class Person {
  constructor(name) {
    this.name = name;
  }
  sayHello() {
    console.log("Hello, I'm " + this.getName());
  }
  getName() {
    return this.name;
```

```
  }
}

// Personを敬承
// 同じメソッド名であるsayHelloは敬承先のものに上書きされる。
class Teacher extends Person {
  sayHello() {
    console.log("Hi, I'm " + this.getName());
  }
}

const teacher = new Teacher('soarflat');
teacher.sayHello(); // => Hi, I'm soarflat
```

関数のデフォルト引数

関数にデフォルト引数を指定できる。

```
// 引数の値を乗算する関数
function multiply (a = 5) {
  return a * a;
}

console.log(multiply()) // => 25
console.log(multiply(10)) // => 100
```

分割代入

配列やオブジェクトからデータを取り出して個別の変数に代入できる。
従来より簡潔な記述で変数に代入できる。

従来の構文

```
var a = 1;
var b = 2;
```

```
var c = {d: 3};
var d = 3;

console.log(a, b, c, d); // => 1 2 {d: 3} 3
```

配列の分割代入の構文

```
const [a, b, c, d] = [1, 2, {d: 3}, 3];

console.log(a, b, c, d);  // => 1 2 {d: 3} 3
```

オブジェクトの分割代入の構文

```
const object = {a: 1, b: 2, c: {d: 3}};
const {a, b, c, c: {d}} = object;

console.log(a, b, c, d); // => 1 2 {d: 3} 3
```

```
const person = {
  name: 'soarflat',
  country: 'Japan'
};
const {name, country} = person;

console.log(name); // => soarflat
console.log(country); // => Japan
```

```
function fullName ({first, last}) {
  return first + last;
}

console.log(fullName({ first: 'John', last: 'Doe'})); // => JohnDoe
```

上記のような関数の引数に分割代入する構文は以下のメリットがある。

- 引数の順番を考慮する必要がなくなるため、引数の追加や削除が楽。
- 関数を実行する側は引数の順番を知らなくて良いので依存関係が減る。

- 引数を渡す時に何を渡しているのかがわかりやすい。

テンプレート文字列

``（バッククオート）で文字列を囲むと、 ${} で文字列内に変数展開ができ、改行も反映できる。
従来の文字列での構文より簡潔かつ明瞭に記述できる。

従来の文字列での構文

```
const name = 'soarflat';

console.log('My name is ' + name); // => My name is soarflat
console.log("My name is " + name); // => My name is soarflat
```

テンプレート文字列での構文

```
const name = 'soarflat';

console.log(`My name is ${name}`); // => My name is soarflat
console.log(`My
name
is
${name}`); // => My
           //    name
           //    is
           //    soarflat
```

スプレッド構文 ...

> スプレッド構文 は、複数の引数 (関数呼び出しのため) または複数の要素 (配列リテラルのため)、あるいは複数の値 (分割代入のため) が置かれるところで式が展開されます。

スプレッド構文 - JavaScript | MDN

スプレッド構文を利用すれば、従来の構文より簡潔に記述できる。
言葉だと説明しづらいため、実際に動作を見ていく。

可変長引数（レストパラメータ）

関数の仮引数にスプレッド構文を利用すると可変長引数を配列で受け取ることができる。

```
function func (...r) {
  console.log(r);
  console.log(r[0]);
}

func(1, 2, 3, 4, 5); // => [1, 2, 3, 4, 5]
                     // => 1
```

```
function func (a, ...r) {
  console.log(a);
  console.log(r);
}

func(1, 2, 3, 4, 5); // => 1
                     // => [2, 3, 4, 5]
```

配列の展開

スプレッド構文を利用すると配列の展開ができる。

従来の構文

```
var array = [1, 10, 3];

console.log(array);                    // => [1, 10, 3]
console.log(Math.max(1, 10, 3));       // => 10
```

```
console.log(Math.max(array));             // => NaN
console.log(Math.max.apply(null, array)); // => 10
```

スプレッド構文を利用した構文

```
const array = [1, 10, 3];

console.log(...array);           // => 1 10 3
console.log(Math.max(1, 10, 3)); // => 10
console.log(Math.max(array));    // => NaN
console.log(Math.max(...array)); // => 10



function func (x, y, z) {
  console.log(x);
  console.log(y);
  console.log(z);
}

const array = [1, 2, 3]
func(...array); // => 1
                // => 2
                // => 3
```

分割代入時に複数の値を1つの配列にまとめる

分割代入時にスプレッド構文を利用すると複数の値を1つの配列にまとめられる。

```
const [a, b, ...c] = ['a', 'b', 'c', 'C'];

console.log(a); // => a
console.log(b); // => b
console.log(c); // => ["c", "C"]
```

配列の結合

スプレッド構文を利用すると配列の結合が簡単にできる。

従来の構文

```
var array = [1, 2, 3];
var array2 = array.concat([4, 5, 6]);

console.log(array2); // => [1, 2, 3, 4, 5, 6]
```

スプレッド構文を利用した構文

```
const array = [1, 2, 3];
const array2 = [...array, 4, 5, 6];

console.log(array2); // => [1, 2, 3, 4, 5, 6]
```

for...of

反復可能なオブジェクト（配列等）に対して反復処理（ループ処理）ができる。
for...in は配列のインデックスやオブジェクトのプロパティ等を処理できるのに対し
for...of は配列の値（要素）等を処理できる。

for...in を利用した構文

```
const array = [65, 55, 80, 100];

for(const index in array) {
  console.log(index);
}
// => 0
// => 1
// => 2
// => 3

for(const index in array) {
  console.log(array[index]);
}
```

```
// => 65
// => 55
// => 80
// => 100
```

```
const obj = {name: 'soarflat', country: 'Japan'}

for(const key in obj) {
  console.log(key);
}
// => name
// => country
```

for...of を利用した構文

```
const array = [65, 55, 80, 100];

for(const value of array) {
  console.log(value);
}
// => 65
// => 55
// => 80
// => 100
```

for...of でオブジェクトのプロパティの値は処理できないのか

通常ではできない。

従来通りに for...in を利用する。

```
const obj = {name: 'soarflat', country: 'Japan'}

for(const key in obj) {
  console.log(obj[key]);
}
// => soarflat
// => Japan
```

Promise

Promiseは非同期処理を扱うオブジェクト。

Promiseを利用すれば従来より簡潔かつ明瞭に非同期処理を記述できる。

従来の構文（ネストが深くなりコールバック地獄と呼ばれている）

```
function timer(number, callback) {
  setTimeout(function() {
    callback(number * 2);
  },1000);
}

timer(100, function(number) {
  timer(number, function(number) {
    timer(number, function(number) {
      timer(number, function(number) {
        timer(number, function(number) {
          console.log(number); // => 3200
        });
      });
    });
  });
});
```

Promiseを利用した構文

```
function timer(number) {
  return new Promise(function(resolve, reject) {
    setTimeout(function() {
      resolve(number * 2);
    },1000);
  });
}

timer(100)
  .then(timer)
  .then(timer)
  .then(timer)
  .then(timer)
  .then(function onFulfilled(value) {
```

```
    console.log(value) // => 3200
});
```

※ES2017の async/await を利用することで、非同期処理を更に簡潔に書けます。詳細は別の記事に書きましたので、こちらを参考にしてください。

async/await 入門（JavaScript）

終わり

上記の機能、構文は学習コストもそこまで高くはないため、触っていればすぐに慣れると思います。
いずれES2015で書こうと思っており、トランスパイルの面倒臭さよりES2015の魅力が勝っていると感じたなら
この瞬間からES2015を始めてみましょう。

以下を読めば、本記事では記載していない機能、構文に関して知ることができます。

- Learn ES2015（英語）
- JavaScript Promiseの本
- イテレーターとジェネレーター - JavaScript | MDN
- JavaScript の イテレータ を極める！
- JavaScript の ジェネレータ を極める！

お知らせ

Udemy で webpack の講座を公開したり、Kindle で技術書を出版しています。

Udemy:
webpack 最速入門（~~**10,800 円**~~ **-> 2,000 円**）

Kindle（Kindle Unlimited だったら無料）:

React 実践入門（500 円）

React Hooks 入門（500 円）

興味を持ってくださった方はご購入いただけると大変嬉しいです。よろしくお願いいたします。