はじめに

最近ECMAScriptの仕様を追いかけていなかったので勉強してみることにしました。
最新というよりは自分の知らない記法や、見たことあるけど意味は分からないというようなものをまとめてみたというかんじです。
仕様のソースは以下を参考にさせていただきました。

TC39のFinished Proposals

Stage

ECMAScriptはTC39という仕様策定委員会の監修のもと、以下の5つのステップを通過した上でリリースしていて、その段階をStageと呼称しています。

Stage	概要
0	Strawman - [アイデア]
1	Proposal - [提案]
2	Draft - [ドラフト]
3	Candidate - [実装候補]
4	Finished - [実装予定]

今回はES2016で実装された2つの仕様と、ES2017以降で実装されるであろう**Stage 4**の仕様の紹介をしていきたいと思います。

Array.prototype.includes()

配列内に求めている値が入っているかをtrue or falseで返しています。
今までの値チェックの方法は `indexOf()` とか使って値が入っているかどうかを確認していましたが、正直-1を返すとかが若干ややこしかったので返ってくる値がboolean型なのは可読性がぐっと上がるだろうなと感じたので、今後はこれを使っていこうと思いました。

```
const arr = ['hoge', 'fuga', 'piyo'];

console.log(arr.includes('hoge')); // true
console.log(arr.includes('hogera')); // false
```

仕様	リリース日
ES2016~	2016/06/17

参照元: https://github.com/tc39/Array.prototype.includes/

Exponentiation Operator

Exponentiation（冪乗・累乗）が簡単に使えるようになった模様。
この記法はPythonやRubyではよく使われるそうですが、果たしてJSではどのように使われるのでしょうか…。
ブラウザ上で使用する電卓とかですかね？笑

※この ** という記法は**べき乗演算子**というそうです。

```
console.log(2 ** 2); // 4
console.log(2 ** 8); // 256
console.log(2 ** 10); // 1024
```

仕様	リリース日
ES2016~	2016/06/17

参照元: https://github.com/rwaldron/exponentiation-operator

Object.values() & Object.entries()

- `Object.values()` はオブジェクトのvalueを配列にして返すメソッド。
- `Object.entries()` はオブジェクトを `[key, value]` という形で多次元配列にして返すメソッド。

`Object.keys()` というkeyを配列にして返すメソッドは存在していましたが、valueを配列にして返すメソッドがありませんでしたが、ES2017以降はそれが使用可能になります。

```
const obj = {
  name: 'yuyake0084',
  bloodType: 'B'
};

Object.keys(obj); // ["name", "bloodType"] <= ES2015から実装可能
Object.values(obj); // ["yuyake0084", "B"]
Object.entries(obj); // [["name", "yuyake0084"], ["bloodType","B"]]
```

仕様	リリース日
ES2017~	未定

参照元: https://github.com/tc39/proposal-object-values-entries

String padding

String.prototype.padStart()

指定した文字数になるように、先頭に指定した文字列を差し込む事ができるメソッド。
使い方としては、第一引数に指定文字数を入れ、第二引数では差し込む文字列を入れます。
差し込む文字数ともとの文字数の和が、指定した文字数に満たない場合は、
その数に到達するまで差し込む文字列を繰り返し差し込みます。
逆に、指定した文字数より、差し込む文字数ともとの文字数の和が多くなる場合は、

指定数からもとも文字数を引いた数だけ第二引数を差し込み、以降の文字列をカットして文字列の結合をします。

```
const str = 'hoge';

console.log(str.padStart(10));          //       hoge    全部で10文字になるよう デフォルトの空文字''を
console.log(str.padStart(10, 'fuga')); // fugafuhoge    全部で10文字になるよう fugaを繰り返す
console.log(str.padStart(5, 'piyo'));  // phoge         全部で 5文字になるよう piyoをカット
```

String.prototype.padEnd()

padStart() とは違い、末尾に文字列を結合します。

```
const str = 'hoge';

console.log(str.padEnd(10)); // hoge
console.log(str.padEnd(10, 'fuga')); // hogefugafu
console.log(str.padEnd(5, 'piyo')); // hogep
```

※2017年1月19日時点ではChromeでの動作は確認できませんでしたが、FireFoxでは動作確認できました。

仕様	リリース日
ES2017~	未定

参照元: https://github.com/tc39/proposal-string-pad-start-end

Object.getOwnPropertyDescriptors()

.....長っ！
なんて長いメソッド名でしょう。
これは覚えるのに少々時間がかかりそうなイメージですが、メソッド名の意味を紐解いていくと

「ああ、なるほどね」となるはずです。
Descriptorsは直訳すると**記述子**といってデータやファイルの性質という意味を持っています。つまり、『オブジェクトのプロパティのデータの性質を取得するよ』というメソッドと解釈しています。

じゃあどんなオブジェクトが返ってくるかを確認してみましょう！

```
const obj = {
  name: 'yuyake0084',
  get hello() {
    return `Hello my name is ${this.name}`;
  }
};
const descriptor = Object.getOwnPropertyDescriptors(obj);

console.log(descriptor);

// descriptor = {
//   hello: {
//     configurable: true,
//     enumerable: true,
//     get: hello(),
//     set: undefined
//   },
//   name: {
//     configurable: true,
//     enumerable: true,
//     value: "yuyake0084",
//     writable: true
//   }
// }
```

なるほど、たしかに name の情報がオブジェクトとして返ってきてるみたいです。
それぞれどういう意味なのか分からなかったのでググって表にしておきます。

プロパティ	**意味**
configurable	Descriptorを編集する事が可能かどうか
enumerable	valueをfor...inで列挙可能かどうか
get	getterの関数を返します

プロパティ	意味
set	setterの関数を返し、setがなければ undefind を返します
value	keyに対してのvalue
writable	値の書き換えが可能かどうか

以上6つのフィールドを持っていて、keyによって返ってくるオブジェクトの内容が変わってくるようです。
もともとES5にあったメソッドだったらしいのですが、ES2017から書き方がシンプルになって実装されるようです。

```
// ES5

const descriptor = Object.getOwnPropertyDescriptor(obj, 'hello');
const descriptor = Object.getOwnPropertyDescriptor(obj, 'name');
```

仕様	リリース日
ES2017~	未定

参照元: https://github.com/tc39/proposal-object-getownpropertydescriptors

Trailing commas in function parameter lists and calls

引数を改行で書いた時の最後のカンマを許容するようです。
地味に嬉しい仕様ですね。

```
function hoge(
  name,
  bloodType, // <= これ
) {
  return `My name is ${name}! Blood Type is ${bloodType} type.`;
}

hoge(
```

```
  'yuyake0084',
  'B',
);
```

仕様	リリース日
ES2017~	未定

参照元: https://github.com/tc39/proposal-trailing-function-commas

Async Functions

別名、**Async/Await**と呼ばれているもので、非同期処理を同期的に書く事ができる文法です。
以下はPromiseがES2015に実装された非同期処理の書き方の一例です。

```
// Promise

function theWorld(n) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log(`${n}秒経過`);
      resolve(n + 1);
    }, 1000);
  });
}

theWorld(1)
  .then(theWorld)
  .then(theWorld)
  .then(() => {
    setTimeout(() => {
      console.log('時は動き出す');
    }, 1000);
  });
```

theWorldという関数が実行され、3秒経過した後で『時は動き出す』というログを吐くというだけの処理ですね。
ではこれをAsync functionsで書くと以下のようになります。

```
// Async functions

function theWorld(n) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log(`${n}秒経過`);
      resolve(n + 1);
    }, 1000);
  });
}

async function StarPlatinumTheWorld(init) {
  let n;

   n = await theWorld(init);
   n = await theWorld(n);
   n = await theWorld(n);
   console.log('時は動き出す');
};

 StarPlatinumTheWorld(1);
```

なるほど、たしかに同期的っぽく書けますね。
Promiseも素敵ですけどAsync/Awaitの方が可読性が高いような気がします。

仕様	リリース日
ES2017~	未定

参照元: https://github.com/tc39/ecmascript-asyncawait

まとめ

斬新かつ実用性があるメソッドがたくさん出てきましたね！
ただ、完全に理解しきれていないのでこれからたくさん使っていって自身に落とし込んでいこうかなと思います。（特にAsync/Await。。。）
ではでは～

参考にさせていただいた記事

- 今年のうちに知っておきたい！ES2017に入る5つの新仕様
- JavaScriptのGetter/Setter 〜 JSおくのほそ道 #018