ES6(2015) ES2017 cheat sheet

About The Author

Naftali Murgor is a software developer. Starting in late 2016, they have gained experience working for companies like ViewFin Canada and Bitgesell. They have also worked as a game tester and Q.A. correspondent for indie game franchises. Naftali Murgor is passionate about helping other developers get into full stack software development and is excited to share their knowledge with others.

1. Arrow Functions

Arrow function expressions provide an alternative to traditional functions using the `=>` syntax.

Read more about arrow functions from MDN

```
// Basic syntax
const aFunction = () => {
  // code to be executed
};

// A one-liner function with an implicit return
const add = (a, b) => a + b;
// This is the same as:
const add = (a, b) => {
  return a + b
}

console.log(add(2 , 6)) // prints 8

// A function with one parameter
const sayHello = name => console.log(`Hello, ${name}!`);
console.log(sayHello("Jerry")) // prints Hello, Jerry!

// A function with default parameter
// Default parameters come last in the parameter list
const multiplyNumbers = (a, b = 1) => a * b;
console.log(multiply(10)) // prints 10
```

2. Template Literals (``)

Template literals provide a powerful way to work with strings using the backticks syntax (``):

```
const firstName = 'John';
const lastName = 'Doe';

const fullName = `${firstName} ${lastName}`;

const message = `Hello ${fullName}, welcome to my website!`;

// Embed expressions in strings
const VAT = `the value added tax is ${100 * 0.16}%`
console.log(VAT) // prints: the value added tax is 16%
```

3. Destructuring assignment

Destructuring assignment makes it possible to "unpack" values from objects and arrays as individual variables:

```
// Array destructuring
const myArray = [1, 2, 3, 4, 5];
const [first, second, ...rest] = myArray;

// Object destructuring
const userObj = { name: 'John', age: 30 };
const { name, age } = userObj;
// Use name and age as separate variables

// Previously, this would have been:
const name = userObj.name
const age = userObj.age
```

4. Spread Operator

The spread operator allows an iterable (array, object) to be copied over to places where zero or more arguments are expected:

```
// Array spread operator
const myArray = [1, 2, 3];
const newArray = [...myArray, 4, 5];

// Object spread operator
const userObj = { name: 'John', age: 30 };
const newUserObj = { ...myObject, occupation: 'Developer' };

console.log(newUserObj) // prints {name: 'John', age: 30, occupation: 'Developer'}
```

5. Rest Parameters

Rest parameters provide a way of allowing a function to accept an indefinite number of arguments:

```
// sum can take as many arguments
function sum(...numbers) {
  return numbers.reduce((acc, curr) => acc + curr, 0);
}
console.log(sum(1,3,3,5,5)) // prints 17
```

7. Classes

NB: class syntax is "syntactical sugar" over prototype based inheritance. There are **no classes** in JavaScript:

```
class Person {
  constructor(name, occupation) {
    this.name = name;
    this.occupation = occupation;
  }
  greet() {
    console.log(`Hello, my name is ${this.name} and I'm a ${this.occupation} .`);
  }
}
const person = new Person('Roseland', 'Full stack developer');
person.greet() // Hello, my name is Roseland and I'm a Full stack developer.
```

8. Modules

ESM modules provide a more flexible way of handling reusability of code:

```
// person.js
export class Person {
  constructor(name) {
    this.name = name;
  }
}

// themes.js
export LIGHT_THEME = 'LIGHT_THEME'
export DARK_THEME = 'DARK_THEME'

// app.js
import { Person } from './person.js';
import { LIGHT_THEME, DARK_THEME} from './themes'

console.log(LIGHT_THEME) // LIGHT_THEME
console.log(DARK_THEME) // DARK_THEME

const john = new Person('John');
```

In HTML page:

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>My Website</title>
  </head>
  <body>
    <h1>Hello, world!</h1>
  </body>

<script type="module">
    import { Person } from './person.js';
    const john = new Person('John');
<script/>
</html>
```

9. Promises

Promises provide a better way of handling asynchronous code over callbacks:

```
// Create a Promise
const promise = new Promise((resolve, reject) => {
  // Do some asynchronous work
  setTimeout(() => {
    // If the work is successful, call resolve with the result
    resolve('Success!');
    // If the work fails, call reject with an error
    // reject(new Error('Something went wrong!'));
  }, 1000);
});

// Consume the Promise
promise
  .then(result => {
    console.log(result); // 'Success!'
  })
  .catch(error => {
    console.error(error); // 'Something went wrong!'
  });
```

NB: The `fetch` API is built on promises:

```
 fetch('https://dummyjson.com/posts')
          .then((data) => data.json())
          .then((json) => { console.log(json) });
```

ES2015 Array Methods

NB: Arrays in JavaScript and most Languages are Zero indexed. Array indexes begin at zero

Array Syntax

```
const names = ['Roseland', 'Naftali', 'Sebastian']
console.log(names[0]) // Roseland
console.log(names[1]) // Naftali
console.log(names[2]) // Sebastian
```

ES6 introduced new Array methods:

1. `Array.prototype.find()`

The **`.find()`** method returns the first element that satisfies the test function provided. If no element is found `undefined` is returned.

```
const numbers = [1, 2, 3, 4, 5];
const found = number.find(element => element > 3);
console.log(found); // 4

const found = number.find(element => element >= 3);
console.log(found) // 3
```

2. `Array.prototype.findIndex()`

The **`findIndex()`** method returns the index of the first element that satisfies the test provided function. If no element is found `-1` is returned.

```
const numbers = [1, 2, 3, 4, 5];

const index = numbers.findIndex(element => element > 3);

console.log(index); // 3

const notFound = numbers.findIndex(element => element > 10);

console.log(index); // -1
```

3. `Array.prototype.fill()`

The **`fill()`** method changes all elements in an array to a static value, from a start index (default `0`) to an end index (default `array.length`). It returns the modified array.

```
const numbers = [1, 2, 3, 4 ,5, 6, 7, 8, 9]
numbers.fill(0) // fill all indexes with 0 value
console.log(numbers) // [ 0, 0, 0, 0, 0,0, 0, 0, 0]
```

4. `Array.prototype.includes()`

The **`includes()`** method returns `true` if element exists in the array and `false` if element does not exist in the array:

```
const number = [1, 2, 3, 4, 5];

const includesThree = number.includes(3);
const includesSix = number.includes(6);

console.log(includesThree); // true
console.log(includesSix); // false
```

5. `Array.from()`

The `Array.from()` returns a shallow copied array from array-like iterable such as a string:

```
const helloWorld = 'Hello, world!';

const array = Array.from(helloWorld);

console.log(array); // ['H', 'e', 'l', 'l', 'o', ',', ' ', 'w', 'o', 'r', 'l', 'd', '!']
```

6. `Array.of()`

The `Array.of()` static method returns an array from the supplied elements:

```
const array = Array.of(1, 2, 3);
console.log(array); // [1, 2, 3]
```

7 `Array.prototype.entries()`

The `Array.prototype.entries()` method returns an iterator object that contains a key/value pair of indexes and the values:

```
const array = ['a', 'b', 'c'];

const iterator = array.entries();

console.log(iterator.next().value); // [0, 'a']
console.log(iterator.next().value); // [1, 'b']
console.log(iterator.next().value); // [2, 'c']
```

8 `Array.prototype.keys()`

The `Array.prototype.keys` returns an iterator with all the indexes:

```
const array = ['a', 'b', 'c'];

const iterator = array.keys();

console.log(iterator.next().value); // 0
console.log(iterator.next().value); // 1
console.log(iterator.next().value); // 2
```

9 `Array.prototype.values`

The `Array.prototype.values` method returns an iterator with all the values in the array:

```
const array = ['a', 'b', 'c'];

const iterator = array.values();

console.log(iterator.next().value); // 'a'
console.log(iterator.next().value); // 'b'
console.log(iterator.next().value); // 'c'
```

ES6 String Methods

ES6(E2015) introduced new methods for working with strings in JavaScript:

1. `String.prototype.startsWith()`

The **`startsWith`** method returns `true` if a string starts with given characters.

```
const string = 'Saturday Night Live!';

const startsWithSaturday = string.startsWith('Saturday');
const startsWithLive = string.startsWith('Live');

console.log(startsWithSaturday); // true
console.log(startsWithLive); // false
```

2. `String.prototype.endsWith()`

The **`endsWith()`** method returns `true` if string ends with given character and `false` otherwise.

```
const string = 'Hello, world!';

const endsWithHello = string.endsWith('Hello');
const endsWithWorld = string.endsWith('World');

console.log(endsWithHello); // false
console.log(endsWithWorld); // false
```

3. `String.prototype.includes()`

The **`includes()`** method performs a case-sensitive search to determine if the string contains the supplied string and returns `true` if found and `false` if the string is not found:

```
const string = 'Hello, world!';
const includesHello = string.includes('Hello');
const includesWorld = string.includes('World');

console.log(includesHello); // true
console.log(includesWorld); // false, World! is included and not 'World'
```

4. `String.prototype.repeat()`

The **`includes()`** method repeats a string with the supplied characters with the supplied number of times to repeat:

```
const string = 'Hello, world!';

const repeated = string.repeat(3);

console.log(repeated); // 'Hello, world!Hello, world!Hello, world!'
```

ES2017(ES8)

ES2017 introduced powerful features. Discover some of the features introduced to JavaScript:

1. `IIFE`

An IIFE(immediately Invoked Function Expression) is a function executed immediately after its declaration.

```
(function () {
  // code to be executed
})();

(() => {
 // code to be executed
})();

(async () => {
  // code to be executed
})();
```

2. `async/await`

Note: An alternative to `.then()` when consuming `Promises` and can be used to call multiple async functions

```
async function fetchData() {
  const response = await fetch('https://dummyjson.com/products');
  const data = await response.json();
  return data;
}

(async() {
 const todos = await fetchData()
 console.log(todos)
})()
```

3. `Object.entries()`

```
const myObject = { name: 'John', age: 30 };
const entries = Object.entries(myObject);
console.log(entries); // [['name', 'John'], ['age', 30]]
```

4. `Object.values`

The **`values()`** method returns an array of the values of an object:

```
const myObject = {
  name: 'John',
  age: 30,
  hobbies: ['Farming', 'Gaming', 'Reading']
 };

const values = Object.values(myObject);

console.log(values); // [ 'John', 30, [ 'Farming', 'Gaming', 'Reading' ] ]
```

5. `Object.getOwnPropertyDescriptors()`

The `Object.getOwnPropertyDescriptors()` static method returns all own property descriptors of a given object.

```
const myObject = { name: 'John', age: 30 };
const descriptors = Object.getOwnPropertyDescriptors(myObject);

console.log(descriptors);
/*
{
  name: {
    value: 'John',
    writable: true,
    enumerable: true,
    configurable: true
  },
  age: {
    value: 30,
    writable: true,
    enumerable: true,
    configurable: true
  }
}
*/
```

Immutable object

For an immutable object:

```
const myObject = { name: 'John', age: 30 };
const descriptors = Object.getOwnPropertyDescriptors(myObject);

const immutableObject = Object.create(null, descriptors);

// Make the object immutable
for (const key in descriptors) {
  descriptors[key].writable = false;
}

console.log(immutableObject.name); // 'John'
immutableObject.name = 'Jane';
console.log(immutableObject.name); // 'John'
```

Copying properties

```
const source = { name: 'John', age: 30 };
const destination = {};

Object.defineProperties(destination, Object.getOwnPropertyDescriptors(source));

console.log(destination); // { name: 'John', age: 30 }
```
