JavaScript ES6

ITS290F

What is ES6?

- **E**CMA**S**cript (or **ES**) is a trademarked scripting-language specification standardized by the **E**uropean **C**omputer **M**anufacturers **A**ssociation (ECMA) International.
- The 6th edition (**ES6**) of the scripting language is officially known as ECMAScript 2015, was finalized in June 2015.
- JavaScript is compatible with ECMAScript
- Some people like to call ES6 **JavaScript 6**

Reference: https://en.wikipedia.org/wiki/ECMAScript

Why ES6?

- JavaScript is a very powerful programming language that runs on a wide range of platforms, especially with the advent of JavaScript runtimes like Node.js. The adoption of the language is increasing among programmers of different categories and levels.
- As with most things, there have been a quite a few changes across various versions of the language since its creation. However, the ES6 specification of the language added a lot of syntax improvements and new features. This makes writing JavaScript programs more efficient, less error-prone, and so much interesting.
- Let's look at *some* of the useful / interesting ES6 syntax

Reference: https://blog.logrocket.com/javascript-es6-5-new-abstractions-to-improve-your-code-54a369e82407

Variables

- In addition to `var`, you can also use `let` and `const`.
- `let` has a smaller scope than `var`—it is limited to the block surrounding it.
- `const` is the same as `let`, but the value cannot be reassigned.

Reference: https://medium.com/@anton_ivanov/five-minute-crash-course-in-es6-es7-and-node-5eb2f16ce8a5

Variables

```
// The scope of `let` is limited to the {} braces surrounding it.
if (true) {
  let a = 10;
} else {
  let a = 20;
}
console.log(a); // ReferenceError - out of scope

// Constants cannot be reassigned:
const MAGIC = 42;
MAGIC = 43; // Error!

// Properties of constant objects can be modified,
// but the object itself cannot be reassigned:
const credentials = {
 login: 'anton',
 password: '123',
};

credentials.password = '456'; // works fine
credentials = null; // Error!
```

Reference: https://medium.com/@anton_ivanov/five-minute-crash-course-in-es6-es7-and-node-5eb2f16ce8a5

Template literals

- Prior to ES6, strings are delimited by either a pair of single quotes('string') or a pair of double quotes("string"). In ES6, strings can also be delimited by a pair of back-ticks(`string`). Such strings are called template literals.

```
const greeting = `Good morning!`;
const shortcut = `\`cmd\` + \`shift\` + \`G\``;

console.log(greeting); // "Good morning!"
console.log(shortcut); // "`cmd` + `shift` + `G`"
```

Multiline strings

- Prior to ES6, strings in JavaScript were limited to a single line. However, ending a line with a backward slash(\) before beginning a newline made it possible to create seeming multiline strings even though the newlines are not output in the string:

```
const message = "Hello Glad, \
Your meeting is scheduled for noon today.";

console.log(message);
// Hello Glad, Your meeting is scheduled for noon
today.
```

String substitution

- Prior to ES6, string concatenation was heavily relied on for creating dynamic strings.

```
const price = 24.99;
console.log("The item costs $" + price + " on the online store.");
// The item costs $24.99 on the online store.
```

- Using ES6 template literals, the substitution can be done as follows:

```
const price = 24.99;
console.log(`The item costs ${price} on the online store.`);
// The item costs $24.99 on the online store.
```

- A string substitution is delimited by an opening ${ and a closing } and can contain any valid JavaScript expression in between.

```
const price = 24.99;
const discount = 10;

console.log(`The item costs ${(price * (100 - discount) / 100).toFixed(2)}
on the online store.`);
// The item costs $22.49 on the online store.
```

Arrow functions

- Another very important syntax improvement in ES6 is the introduction of **arrow functions**. Arrow functions make use of a completely new syntax and offer a couple of great advantages when used in ways they are best suited for.
- The syntax for arrow functions omits the function keyword. Also the function parameters are separated from the function body using an arrow (=>), hence the name arrow functions.

```
// USING REGULAR FUNCTION
const getTimestamp = function() {
  return new Date;
}

// USING ARROW FUNCTION
const getTimestamp = () => {
  return new Date;
}
```

Arrow functions with parameters

```
const n1 = 10;
const n2 = -10;
const addNumbers = (a,b) => {
    return a+b;
}
const msg = `Add ${n1} to ${n2} gives you ${addNumbers(n1,n2)}`;
console.log(msg);

// Add 10 to -10 gives you 0
```

Arrow functions with *default* parameters

```
const addNumbers = (a = 0, b = 0) => {
    return a+b;
}
console.log(addNumbers());
console.log(addNumbers(10));
console.log(addNumbers(null,20));

// 0
// 10
// 20
```

Arrow functions with *rest* parameters

- let's write a simple **variadic** function `containsAll` that checks whether a string contains a number of substrings. For example,

  `containsAll("banana", "b", "nan")` would return `true`, and
  `containsAll("banana", "c", "nan")` would return `false`.

```
function containsAll(haystack) {
  for (var i = 1; i < arguments.length; i++) {
    var needle = arguments[i];
    if (haystack.indexOf(needle) === -1) {
      return false;
    }
  }
  return true;
}
```

- the magical **`arguments`** object, an array-like object containing the parameters passed to the function.

Reference: https://hacks.mozilla.org/2015/05/es6-in-depth-rest-parameters-and-defaults/

Arrow functions with *rest* parameters

```
function containsAll(haystack, ...needles) {
  for (var needle of needles) {
    if (haystack.indexOf(needle) === -1) {
      return false;
    }
  }
  return true;
}

console.log(containsAll("banana","b"));
console.log(containsAll("banana","b","nan"));
console.log(containsAll("banana","x","b","nan"));

// true
// true
// false
```

- The argument **`haystack`** is filled with parameter that is passed first, namely `"banana"`
- The argument **`needles`** is set to `["b"]`, `["b","nan"]` and `["x","b","nan"]`

Reference: https://hacks.mozilla.org/2015/05/es6-in-depth-rest-parameters-and-defaults/

Restrictions of arrow functions

Although arrow functions are more compact and shorter than regular functions, they are significantly different from regular functions in some ways that define how they can be used:

1. Arrow functions cannot be used as constructors and they have no prototype. Hence, using the `new` keyword with an arrow function will usually result in an error.
2. Array function does not have `arguments` object. Duplicate named parameters are also not allowed.
3. The `this` binding inside an arrow function cannot be modified, and it always points up to the closest non-arrow parent function.

Destructuring

- Destructuring lets you assign multiple variables at once.

```
// Before:
var width = 200;
var height = 400;

// After:
let [width, height] = [200, 400];
```

- Also works with functions that return arrays:

```
function get() {
  return ['42', 'success'];
}

// Return two variables from function at the same time:
let [count, status] = get();
```

Reference: https://medium.com/@anton_ivanov/five-minute-crash-course-in-es6-es7-and-node-5eb2f16ce8a5

for…of; entries()

- No more clumsy looping over arrays…

```
var towns = ["Causeway Bay", "Central", "Diamond Hill"];

// Loop over each item of the array,
// supporting `return`, `break` and `continue`:
for (let town of towns) {
  if (town === "Central")
    return;
}
```

- If you need the `i` variable, use `.entries()`:

```
var towns = ["Causeway Bay", "Central", "Diamond Hill"];

// Get the index variable, just like in `for` loops:
for (let [i, town] of towns.entries()) {
  towns[i] = `The great city of ${town}!`;
}
```

Reference: https://medium.com/@anton_ivanov/five-minute-crash-course-in-es6-es7-and-node-5eb2f16ce8a5

Array map

```
var numbers = [4, 9, 16, 25];
var items = ['banna','coffee','eggs'];

x = document.getElementById("x");
x.innerHTML = numbers.map(Math.sqrt);

const toUpper = (x) => {
     return x.toUpperCase();
}
y = document.getElementById("y");
y.innerHTML = items.map(toUpper);
```

Expand arrays

- Three dots before an array expands it into separate values:

```
let numbers = [1, 5, 20];

// Before:
let result = sum(numbers[0], numbers[1], numbers[2]);

// After:
let result = sum(...numbers);
```

- You can use this to quickly concatenate several arrays:

```
var even = [2, 4, 6];
var odd = [1, 3, 5];

var all = [...odd, ...even]; // [1, 3, 5, 2, 4, 6]
```

Reference: https://medium.com/@anton_ivanov/five-minute-crash-course-in-es6-es7-and-node-5eb2f16ce8a5