JS Modern Syntax

Learning Objectives

- [ ] **JS Modern Syntax**
  - [ ] understanding JS as a developing programming language
  - [ ] destructuring assignment
  - [ ] rest and spread syntax
  - [ ] optional chaining
  - [ ] nullish coalescing

A Developing Language: JavaScript

JavaScript isn't *done* – it is still being developed. The tc39 group is meeting up on a bi-monthly schedule to discuss proposals regarding ECMAScript specifications, which JavaScript is based on. Here you can find a list of active proposals that are still in the process of being reviewed. You can take a look at some of the finished proposals over here.

Destructuring Assignment

The destructuring assignment syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables. Destructuring doesn't mutate the original array or object.

Destructuring Arrays

```
const greekLetters = ["alpha", "beta", "gamma", "delta"];
```

If we want to declare the values of `greekLetters` as distinct variables, accessing them might turn out to be quite tedious.

Without Destructuring

```
const firstLetter = greekLetters[0];
const secondLetter = greekLetters[1];
const thirdLetter = greekLetters[2];
const fourthLetter = greekLetters[3];
// firstLetter → 'alpha'
// secondLetter → 'beta'
// etc.
```

With Destructuring

```
const [firstLetter, secondLetter, thirdLetter, fourthLetter] =
greekLetters;
// firstLetter → 'alpha'
// secondLetter → 'beta'
// etc.
```

The result of the two code snippets is the same but destructuring the array makes for more readable and concise code.

If you don't need or want the second and the fourth value in our initial array `greekLetters` there's an option to skip values during the process of destructuring.

```
const [firstLetter, , thirdLetter] = greekLetters;
// firstLetter → 'alpha'
// thirdLetter → 'gamma'
```

`thirdLetter` now contains the *third* value of `greekLetters`. The extra comma skips the value at the respective position in `greekLetters`. The fourth value is ignored because the destructuring assignment ended.

Skipping values with extra commas can quickly become unreadable. Imagine wanting to skip 5 values: `const [a, , , , , , g] = x;`. General advice is to be very mindful with this feature.

Destructuring Objects

```
const coachObject = {
  name: "Sam",
  mood: "great",
  skills: "amazing",
  score: 9999,
};
```

You can use destructuring assignment like this:

```
const { name, mood, skills, score } = coachObject;
// name → 'Sam'
// mood → 'great'
// etc.
```

Now you've got `name`, `mood`, `skills` and `score` available as distinct variables.

In contrast to array destructuring the variable names are not arbitrary and do not depend on the order. Here the names of the variables have to match the keys of the object properties.

You can rename the variable while destructuring:

```
const { name: firstName } = coachObject;
// firstName → 'Sam'
// name → undefined
```

You can also set a default value of a property in case it does not exist:

```
const { isAdmin = true } = coachObject;
// isAdmin → true
```

Rest and Spread Syntax

The rest and the spread syntax look deceptively similar. Both are identified by three dots: `...`. They do two different things though, depending in which context `...` is used.

Rest Syntax (`...`)

The rest syntax allows you to say: "Put the rest into this variable" when using destructuring assignment or declaring function parameters.

You can use it with arrays:

```
const greekLetters = ["alpha", "beta", "gamma", "delta"];
const [firstLetter, ...allTheOtherLetters] = greekLetters;
// firstLetter → "alpha"
// allTheOtherLetters → ["beta", "gamma", "delta"]
```

Or with objects:

```
const coachObject = {
  name: "Sam",
  mood: "great",
  skills: "amazing",
  score: 9999,
};

const { name, score, ...theRestOfTheCoachObject } = coachObject;
// name → "Sam"
// score → 9999
// theRestOfTheCoachObject → { mood: 'great', skills: 'amazing' }
```

And even with function parameters:

```

```

```
function logLetters(firstLetter, ...moreLetters) {
  console.log("the first letter is", firstLetter);
  console.log("even more letters", moreLetters);
}

logLetters("alpha", "beta", "gamma", "delta");
// logs:
// the first letter is alpha
// even more letters (3) ['beta', 'gamma', 'delta']
```

Spread Syntax (`...`)

The spread syntax allows you to say: "spread everything inside this variable into here" when declaring array or object literals or calling functions.

It works like this with array literal declarations:

```
const greekLetters = ["alpha", "beta", "gamma", "delta"];
const moreGreekLetters = [...greekLetters, "epsilon", "zeta"];
// moreGreekLetters → ['alpha', 'beta', 'gamma', 'delta', 'epsilon',
'zeta']
```

You can spread two (or more) arrays into another array ...

```
const redColors = ["crimson", "pink", "purple"];
const blueColors = ["navy", "teal", "sky"];
const mixedColors = [...redColors, ...blueColors];
// mixedColors → ['crimson', 'pink', 'purple', 'navy', 'teal', 'sky']
```

... and it matters where respectively you spread one array into another:

```
const cats = ["cat", "cat", "cat"];
const dogs = ["dog", "dog", "dog"];

const catsAndDogs = [...cats, ...dogs];
// catsAndDogs → ['cat', 'cat', 'cat', 'dog', 'dog', 'dog']

const dogsAndCats = [...dogs, ...cats];
// dogsAndCats → ['dog', 'dog', 'dog', 'cat', 'cat', 'cat']

const catsBetweenBirds = ["bird", ...cats, "bird"];
// catsBetweenBirds → ['bird', 'cat', 'cat', 'cat', 'bird']
```

Here is how the spread syntax works with object declarations:

```
const circle = { radius: 5, shape: "circle" };

const greenCircle = { ...circle, color: "green" };
// greenCircle → { radius: 5, shape: 'circle', color: 'green' }
```

Notice that the order of spread operations matters because you can override properties:

```
const circle = { radius: 5, shape: "circle" };

const largeCircle = { ...circle, radius: 20 };
// largeCircle → { radius: 20, shape: 'circle' }

const notALargeCircle = { radius: 20, ...circle };
// notALargeCircle → { radius: 5, shape: 'circle' }
```

The spread syntax is very helpful for creating a shallow copy of arrays and objects:

```
const book = { title: "Ulysses", author: "James Joyce" };
const shallowCopyOfBook = { ...book };
shallowCopyOfBook.year = "1920";

// book → { title: 'Ulysses', author: 'James Joyce' }
// shallowCopyOfBook → { title: 'Ulysses', author: 'James Joyce', year:
'1920' }
```

```
const greekLetters = ["alpha", "beta", "gamma", "delta", "epsilon",
"zeta"];
const sortedGreekLetters = [...greekLetters].sort((a, b) =>
a.localeCompare(b));
// this is an alternative to greekLetters.slice() which also creates a
shallow copy

// greekLetters → ['alpha', 'beta', 'gamma', 'delta', 'epsilon', 'zeta']
// sortedGreekLetters → ['alpha', 'beta', 'delta', 'epsilon', 'gamma',
'zeta']
```

You can also use the spread syntax when calling functions:

```
const numbers = [4534, 3411, 2455, 4952];
const smallestNumber = Math.min(...numbers);
// smallestNumber → 2455
```

Optional Chaining

The optional chaining operator `?.` is like the chaining operator `.`, except that instead of throwing an error if a reference is *nullish* (null or undefined), the expression 'short-circuits' with a value of `undefined`. This is useful when you're not sure if a property, method or index exists or if something is callable.

```
function feedCats(cats) {
  return cats.forEach((cat) => console.log(cat, "says meow"));
}

feedCats(["Garfield", "Tom", "Grumpy Cat"]);
// logs:
// Garfield says meow
// Tom says meow
// Grumpy Cat says meow

feedCats();
// throws:
// Uncaught TypeError: Cannot read properties of undefined (reading
'forEach')
```

```
function feedCats(cats) {
  return cats?.forEach((cat) => console.log(cat, "says meow"));
}

feedCats();
// logs nothing, but also does not throw
```

The `?.` short-circuits the expression and evaluates the return statement to `undefined`. This way the error is avoided without the need to explicitly use if statements or ternaries to check the values.

Optional chaining is useful when targeting a property in an object with a deeply nested structure.

```
const person = {
  name: "Sam",
  skills: [
    {
      name: "HTML",
      level: 9999,
      category: {
        name: "coding",
      },
    },
    {
      name: "Agile",
      level: 1337,
      category: {
        name: "projects",
```

```
      },
    },
  ],
};

console.log(person.skills[1].category.name);
// logs: projects

console.log(person.skills[2].level);
// throws: Uncaught TypeError: Cannot read properties of undefined
(reading 'level')

console.log(person.skills?.[2]?.level);
// logs: undefined

console.log(person.skills[0].partner.name);
// throws: Uncaught TypeError: Cannot read properties of undefined
(reading 'name')

console.log(person.skills[0].partner?.name);
// logs: undefined
```

Nullish Coalescing

The nullish coalescing operator `??` is a logical operator that returns its right-hand side operand when its left-hand side operand is null or undefined, and otherwise returns its left-hand side operand.

```
const chocolate = true;

function chocolateCheck() {
  return chocolate ?? "No chocolate :(";
}

const result = chocolateCheck();
```

In this example, our function `chocolateCheck` will return the value of `chocolate` (in this instance: `true`), because the value of `chocolate` is neither `null` or `undefined`.

```
const chocolate = undefined;

function chocolateCheck() {
  return chocolate ?? "No chocolate :(";
}

const result = chocolateCheck();
```

In this example, our function would return `'No chocolate :('`, seeing as how the value of `chocolate` is `undefined`. The same would happen if we declared the variable `chocolate` with the value `null`.

Written as an `if/else` statement, the code would look like this:

```
const chocolate = true;

function chocolateCheck() {
  if (chocolate === null || chocolate === undefined) {
    return "No chocolate :(";
  }

  return true;
}

const result = chocolateCheck();
```

Resources

- TC39 - Specifying JavaScript
- async function (MDN)
- Destructuring assignment (MDN)
- Spread Syntax (MDN)
- Rest parameters (MDN)