ES2018: Async Iteration and More

This lesson covers the new features introduced in ES2018

WE'LL COVER THE FOLLOWING

- Rest / Spread for Objects
- Asynchronous iteration
- `Promise.prototype.finally()`
- RegExp features
  - `s (dotAll)` flag for regular expression
  - RegExp named capture groups
  - RegExp Lookbehind Assertions
  - RegExp Unicode Property Escapes
- Lifting template literals restriction

In this chapter we will look at what was introduced with ES2018.

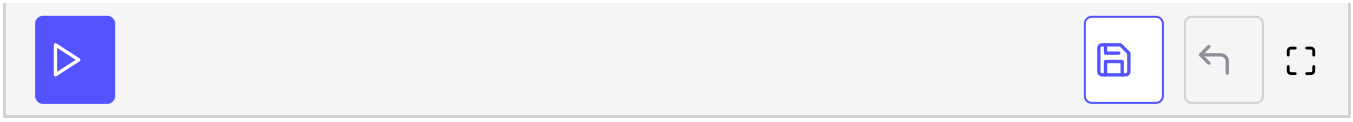
Rest / Spread for Objects

Remember how ES6 (ES2015) allowed us to do this?

```
const veggie = ["tomato","cucumber","beans"];
const meat = ["pork","beef","chicken"];

const menu = [...veggie, "pasta", ...meat];
console.log(menu);
// Array [ "tomato", "cucumber", "beans", "pasta", "pork", "beef", "chicken" ]
```

Now we can use the rest/spread syntax for objects too, let's look at how:

```
let myObj = {
  a:1,
  b:3,
  c:5,
  d:8,
}

// we use the rest operator to grab everything else left in the object.
let { a, b, ...z } = myObj;
console.log(a);     // 1
console.log(b);     // 3
console.log(z);     // {c: 5, d: 8}

// using the spread syntax we cloned our Object
let clone = { ...myObj };
console.log(clone);
// {a: 1, b: 3, c: 5, d: 8}
myObj.e = 15;
console.log(clone)
// {a: 1, b: 3, c: 5, d: 8}
console.log(myObj)
// {a: 1, b: 3, c: 5, d: 8, e: 15}
```



With the spread operator we can easily create a clone of our `Object` so that when we modify the original `Object`, the clone does not get modified, similarly to what we saw when we talked about arrays.

Asynchronous iteration

With asynchronous iteration we can iterate asynchronously over our data.

From the documentation:

> An **async** iterator is much like an iterator, except that its `next()` method returns a promise for a `{ value, done }` pair.

To do so, we will use a `for-await-of` loop which works by converting our iterables to a promise, unless they already are one.

```
const iterables = [1,2,3];
```

```
async function test() {
    for await (const value of iterables) {
        console.log(value);
    }
}
test();
// 1
// 2
// 3
```




> During execution, an async iterator is created from the data source using the `[Symbol.asyncIterator]()` method. Each time we access the next value in the sequence, we implicitly await the promise returned from the iterator method.

`Promise.prototype.finally()`

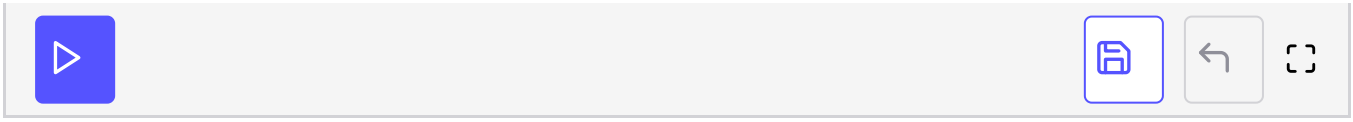
After our promise has finished we can invoke a callback.

```
const myPromise = new Promise((resolve,reject) => {
  resolve();

})
myPromise
  .then(result => {
    console.log('still working');
  })
  .catch(error => {
    console.log('there was an error');
  })
  .finally(()=> {
    console.log('Done!');
  })
```




`.finally()` will also return a `Promise` so we can chain more `then` and `catch` after it, but those promises will fulfill based on the `Promise` they were chained onto.

```
const myPromise = new Promise((resolve,reject) => {
  resolve();
})
myPromise
.then( () => {
    console.log('still working');
    return 'still working';
  })
  .finally(()=> {
    console.log('Done!');
    return 'Done!';
  })
  .then( res => {
    console.log(res);
  })
// still working
// Done!
// still working
```



As you can see the `then` chained after `finally` returned the value that was returned by the `Promise` created not by `finally` but by the first `then`.

RegExp features

Four new RegExp related features made it to the new version of ECMAScript. They are:

- `s(dotAll)` flag for regular expressions
- RegExp named capture groups
- RegExp lookbehind assertions
- RegExp unicode property escapes

`s (dotAll)` flag for regular expression

This introduces a new `s` flag for ECMAScript regular expressions that makes `.` match any character, including line terminators.

```
console.log(/foo.bar/s.test('foo\nbar'));
// true
```

RegExp named capture groups

From the documentation:

> Numbered capture groups allow one to refer to certain portions of a string that a regular expression matches. Each capture group is assigned a unique number and can be referenced using that number, but this can make a regular expression hard to grasp and refactor.
>
> For example, given `/(\d{4})-(\d{2})-(\d{2})/` that matches a date, one can't be sure which group corresponds to the month and which one is the day without examining the surrounding code. Also, if one wants to swap the order of the month and the day, the group references should also be updated.
>
> A capture group can be given a name using the `(?<name>...)` syntax, for any identifier `name`. The regular expression for a date then can be written as `/(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})/u`. Each name should be unique and follow the grammar for ECMAScript IdentifierName.
>
> Named groups can be accessed from properties of a `groups` property of the regular expression result. Numbered references to the groups are also created, just as for non-named groups. For example:

```
let re = /(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})/u;
let result = re.exec('2015-01-02');
// result.groups.year === '2015';
// result.groups.month === '01';
// result.groups.day === '02';

// result[0] === '2015-01-02';
// result[1] === '2015';
// result[2] === '01';
// result[3] === '02';

let {groups: {one, two}} = /^(?<one>.*):(?<two>.*)$/u.exec('foo:bar');
console.log(`one: ${one}, two: ${two}`);
// one: foo, two: bar
```

RegExp Lookbehind Assertions

From the documentation:

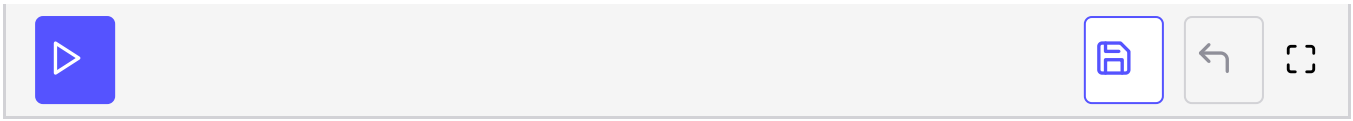
> With lookbehind assertions, one can make sure that a pattern is or isn't preceded by another, e.g. matching a dollar amount without capturing the dollar sign.
>
> Positive lookbehind assertions are denoted as `(?<=...)` and they ensure that the pattern contained within precedes the pattern following the assertion. For example, if one wants to match a dollar amount without capturing the dollar sign, `/(?<=\$)\d+(\.\d*)?/` can be used, matching `'$10.53'` and returning `'10.53'`. This, however, wouldn't match `€10.53`.
>
> Negative lookbehind assertions are denoted as `(?<!...)` and, on the other hand, make sure that the pattern within doesn't precede the pattern following the assertion. For example, `/(?<!\$)\d+(?:\.\d*)/` wouldn't match `'$10.53'`, but would `'€10.53'`.

RegExp Unicode Property Escapes

From the documentation:

> This brings the addition of Unicode property escapes of the form `\p{…}` and `\P{…}`. Unicode property escapes are a new type of escape sequence available in regular expressions that have the `u` flag set. With this feature, we could write:

```
const regexGreekSymbol = /\p{Script=Greek}/u;
console.log(regexGreekSymbol.test('π'));
// true
```



Lifting template literals restriction

When using *tagged* template literals, the restriction on escape sequences are removed.

You can read more here.

Now onto a quiz before the next topic!