AkiraChix
SHE BUILDS • SHE SERVES • SHE LEADS

Outline

- Destructuring
- Arrow functions
- Modules (export and import)
- Callbacks and Promises
- Async/Await

AkiraChix
SHE BUILDS • SHE SERVES • SHE LEADS

Destructuring

The two most used data structures in JavaScript are *Object* { } and *Array* [ ].

Destructuring assignment is a special syntax that allows us to “unpack” arrays or objects into a bunch of variables.

- Array destructuring
- Object destructuring

Array destructuring

We have an array name with first and last name.

```
let name = ["Jane", "Doe"];
```

To derive firstName and lastName:

```
let firstName = arr[0];
let surname = arr[1];
```

Destructuring assignment is a shorter way to write the above:

```
let [firstName, lastName] = name;

alert(firstName);
alert(lastName);
```

NOTE: It's called "destructuring assignment," because it "destructurizes" by copying items into variables. But the array itself is not modified.

Destructuring assignment looks great when combined with split or other array-returning methods:

```
let [firstName, surname] = "Jane Doe".split(' ');
```

Unwanted elements of the array can also be thrown away via an extra comma:

```
// second element is not needed
let [firstName, , title] = ["Jane", "Doe", "Frontend developer", "AkiraChix"];
alert( title );  // Frontend developer
```

Object destructuring

We have a person object

```
let person = {
        name: 'John Doe',
        age: 25,
        profession: 'Frontend developer'
};
```

Object values are assigned to the corresponding key variables:

```
let {name, age, profession} = person;

alert(name);  // John Doe
alert(age);  // 25
alert(profession); // Frontend developer
```

If we want to assign a property to a variable with another name, for instance, person.name to go into the variable named fullName, then we can set it using a colon:

```
let {name: fullName, age, profession} = person;

alert(fullName); // John Doe
```

Default values

We can set default values by using "=", like this:

```
let person = {
        name: 'John Doe',
};

let {name, age=20, profession='Unknown'} = person;
alert(name);
alert(age);
alert(profession);
```

We can use it with destructuring to loop over keys-and-values of an object:

```
let user = {
  name: "Jane",
  age: 23
};

// loop over keys-and-values
for (let [key, value] of Object.entries(user)) {
  alert(`${key}:${value}`);   // name:Jane, then age:23
}
```

The rest operator ...

What if the object has more properties than we have variables? Can we take some and then assign the “rest” somewhere?

```
let user = {
        name: "John Doe",
        age: 30,
        gender: 'male
};
let {name, ...rest} = user;

// now name="John Doe", rest={age: 30, gender: 'male'}
alert(rest.age);  // 30
alert(rest.gender);   // 'male'
```

Arrow functions

Function Declaration vs Expression

Function Declaration: a function, declared as a separate statement, in the main code flow.

```
function sum(a, b) {

  return a + b;

}
```

Function Expression: a function, created inside an expression or inside another syntax construct. Here, the function is created at the right side of the “assignment expression” =:

```
let sum = function(a, b) {

  return a + b;

};
```

Arrow functions are a concise way for creating function expressions.

It's called "arrow functions", because it looks like this:

```
let func = (arg1, arg2, ...argN) => expression
```

This creates a function func that has arguments arg1..argN, evaluates the expression on the right side with their use and returns its result.

In other words, it's roughly the same as:

```
let func = function(arg1, arg2, ...argN) {
  return expression;
};
```

...But much more concise.

For instance:

```
let sum = (a, b) => a + b;
```

The arrow function is a shorter form of:

```
let sum = function(a, b) {
  return a + b;
};

alert( sum(1, 2) );   // 3
```

If we have only one argument, then parentheses can be omitted, making that even shorter:

```
let double = function(n) { return n * 2 }
```

Will be written as:

```
let double = n => n * 2;
alert( double(3) );  // 6
```

If there are no arguments, parentheses should be empty (but they should be present):

```
let sayHi = () => alert("Hello!");

sayHi();
```

Modules (export and import)

Types of exports

- Named exports (several per module)
- Default exports (one per module)
- Cyclical Dependencies

Name exports


```
// lib.js

export function square(x) {
    return x * x;
}

export function sum(a, b) {
    return a + b;
}
```

```
// main.js

import { square, sum } from 'lib';

console.log(square(5));  // 25

console.log(sum(4, 3));   // 7
```

Default exports (one per module)


```
// myFunc.js

export default function () {

        console .log(“Default export func”);

 };
```

```
// main1.js

import myFunc from 'myFunc';

myFunc();
```

Cyclical Dependencies


```
// lib.js

import Main from 'main';

var lib = {message: "This Is A Lib"};

export { lib as Lib };
```

```
// main.js

import { Lib } from 'lib';

export default class Main {

        // ....

}
```

AkiraChix
SHE BUILDS • SHE SERVES • SHE LEADS

Callbacks and Promises

Synchronous operations in JavaScript entails having each step of an operation waits for the previous step to execute completely.

> *This means no matter how long a previous process takes, subsequent process won't kick off until the former is completed.*

Asynchronous operations, on the other hand, defers operations.

> *Any process that takes a lot of time to process is usually run alongside other synchronous operation and completes in the future.*

How do you handle asynchronous operations?

Asynchronous Operations

Operations in JavaScript are traditionally synchronous and execute from top to bottom.

For instance, a farming operation that logs farming process to the console:

```
console.log("Plant kunde");
console.log("Water kunde");
console.log("Add fertilizer");
```

Now let's tweak that a bit so that watering the farm take longer than planting and fertilizing:

```
console.log("Plant kunde");
setTimeout(function() {
        console.log("Water kunde")
}, 3000)
console.log("Add fertilizer");
```

Why did the system go ahead to apply fertilizer and then water kunde after 3 seconds?

Callback Functions

Functions are First-Class Objects.

This means:

- They can be assigned to variables (and treated as a value)
- Have other functions in them
- Return other functions to be called later

When a function simply accepts another function as an argument, this contained function is known as a callback function.

Using callback functions is a core functional programming concept, and you can find them in most JavaScript code; either in simple functions like setInterval, event listening or when making API calls.

Callback functions are written like so:

```
setInterval(function() {
 console.log('hello');
}, 1000)
```

setInterval accepts a callback function as its first parameter and also a time interval.

Naming Callback functions

AkiraChix
SHE BUILDS • SHE SERVES • SHE LEADS

```
function greeting(name) {
        console.log(`Hello ${name},
        welcome to JS class`);
}
```

The above function is assigned a name greeting and has an argument of name.

We're also using an ES6 template string. Let's use this function as a callback function.

```
function introduction(firstName, lastName, callback) {
    const fullName = `${firstName} ${lastName}`;
    callback(fullName);
}
```

```
introduction('John', 'Doe', greeting);
```

NOTE: The callback function is not run unless called by its containing function, it is called back. Hence, the term *call back function*

Promises

> *I promise to do this whenever that is true. If it isn't true, then I won't.*

A promise is used to handle the asynchronous result of an operation.

JS is designed to not wait for an asynchronous block of code to completely execute before other synchronous parts of the code can run.

With Promises, we can defer execution of a code block until an async request is completed. This way, other operations can keep running without interruption.

Promises have three states:

- Pending: This is the initial state of the Promise before an operation begins
- Fulfilled: This means the specified operation was completed
- Rejected: The operation did not complete; an error value is usually thrown

Creating a Promise

The Promise object is created using the new keyword and contains the promise; this is an executor function which has a resolve and a reject callback.

As the names imply, each of these callbacks returns a value with the reject callback returning an error object.

```
const promise = new Promise(function(resolve, reject) {
        // do a thing, possibly async, then…

         if (/* everything turned out fine */) {
           resolve("Stuff worked!");
         }
         else {
           reject(Error("It broke"));
         }
});
```

Using Promises

Using a promise that has been created is relatively straightforward; we chain .then() and .catch() to our Promise like so:

```
.then(function(done) {
    // the content from the resolve() is here
})

.catch(function(error) {
    // the info from reject() is here
})
```

```
let promise = new Promise(function(resolve, reject) {

  // the function is executed automatically when the promise is constructed

  // after 1 second signal that the job is done with the result "done"

  setTimeout(() => resolve("done"), 1000);

});
```

We can see two things by running the code above:

1. The executor is called automatically and immediately (by the new Promise).
2. The executor receives two arguments: resolve and reject — these functions are pre-defined by the JavaScript engine. So we don’t need to create them. Instead, we should write the executor to call them when ready.

Async/Await

Async

An async function is a modification to the syntax used in writing promises.

You can call it syntactic sugar over promises. It only makes writing promises easier.

An async function returns a promise -- if the function returns a value, the promise will be resolved with the value, but if the async function throws an error, the promise is rejected with that value. Let’s see an async function:

```
async function myFavouriteFruit() {
    return 'mango';
}
```

and a different function that does the same thing but in promise format:

```
function myFavouriteFruit() {
    return Promise.resolve('mango');
}
```

Also when a promise is rejected, an async function is represented like this:

```
function foo() {
        return Promise.reject(25);
}

// is equal to
async function() {
        throw 25;
}
```

Await

Await is only used with an async function.

The await keyword is used in an async function to ensure that all promises returned in the async function are synchronized, ie. they wait for each other.

In using async and await, async is prepended when returning a promise, await is prepended when calling a promise.

Assignment

To be sent on slack channel.

Next class

- REVISION

Video of the week - https://www.youtube.com/watch?v=PoRJizFvM7s