JavaScript Closure by Example

JULY 16, 2019 BY ROBIN WIERUCH - EDIT THIS POST

Follow on Twitter 18k Follow on Facebook

Eventually you will come across the concept of a JavaScript Closure. I want to give you a step by step walkthrough on how to implement a JavaScript Closure. Along the way, you will find out yourself why it makes sense to implement certain things with JavaScript Closures. The whole source code can be found on GitHub. If you want to code along the way, make sure to set up a JavaScript project before.

WHY A JAVASCRIPT CLOSURE?

Let's say we have the following JavaScript function which just returns an object for us. The object's properties are based on the incoming function's arguments.

```
}

const employeeOne = getEmployee('Robin', 'Germany');
const employeeTwo = getEmployee('Markus', 'Canada');

const employees = [employeeOne, employeeTwo];
```

In our case, the function creates an object for an employee object. The function can be used to create multiple objects one by one. It's up to you what you are doing with these objects afterwards. For instance, put them in an array to get a list of your company's employees.

In order to distinguish our employees, we should give them an employee number (identifier). The identifier should be assigned **internally** -- because from the outside when calling the function, we don't want to care about the number.

```
function getEmployee(name, country) {
  let employeeNumber = 1;
  return { employeeNumber, name, country };
}

const employeeOne = getEmployee('Robin', 'Germany');
const employeeTwo = getEmployee('Markus', 'Canada');

const employees = [employeeOne, employeeTwo];

console.log(employees);

// [
//   { employeeNumber: 1, name: 'Robin', country: 'Germany' },
//   { employeeNumber: 1, name: 'Markus', country: 'Canada' },
// ]
```

At the moment, every employee has an employee number of 1 which isn't right. It should be a unique identifier. Usually an employee number just increments by one for every joining employee in a company. However, without being able to do something from the outside, the function doesn't know how many employees it has created already. **It doesn't keep track of the state.**

Since **a function doesn't keep any internal state**, we need to move the variable outside of the function, to increment it within the function with every created employee. We keep track of the state by incrementing the number every time the function gets called.

```
let employeeNumber = 1;
```

```
}

const employeeOne = getEmployee('Robin', 'Germany');
const employeeTwo = getEmployee('Markus', 'Canada');

const employees = [employeeOne, employeeTwo];

console.log(employees);

// [
//   { employeeNumber: 1, name: 'Robin', country: 'Germany' },
//   { employeeNumber: 2, name: 'Markus', country: 'Canada' },
// ]
```

Note: The ++ operator (called Increment Operator) increments an integer by one. If it is used postfix (e.g. `myInteger++`), it increments the integer but returns the value from before incrementing it. If it is used prefix (e.g. `++myInteger`), it increments the integer and returns the value after incrementing it. In contrast, there exists an Decrement Operator in JavaScript too.

There is one crucial step we did to implement this feature: We moved the variable outside of the **function's scope** in order to keep track of its state. Before it was internally managed by the function and thus only the function knew about this variable. Now we moved it outside and made it available in the **global scope**.

Now it's possible to mess up things with the new **global scope of the variable**:

```
let employeeNumber = 1;

function getEmployee(name, country) {
  return { employeeNumber: employeeNumber++, name, country };
}

const employeeOne = getEmployee('Robin', 'Germany');
employeeNumber = 50;
const employeeTwo = getEmployee('Markus', 'Canada');

const employees = [employeeOne, employeeTwo];

console.log(employees);

// [
//   { employeeNumber: 1, name: 'Robin', country: 'Germany' },
//   { employeeNumber: 50, name: 'Markus', country: 'Canada' },
// ]
```

though our feature works, the previous code snippet clearly shows that we have a potential pitfall here.

Everything we have done in our previous code snippets was changing the scope of our variable from a function's scope to a global scope. A JavaScript Closure will fix the problem of our variable's scope, making it inaccessible from the outside of the function, but making it possible for the function to track its internal state. Fundamentally, the existence of scopes in programming give closures the air to breathe.

JAVASCRIPT CLOSURE BY EXAMPLE

A JavaScript Closure fixes the problem of our variable's scope. A closure makes it possible to track internal state with a variable in a function, without giving up the local scope of this variable.

```
function getEmployeeFactory() {
  let employeeNumber = 1;
  return function(name, country) {
    return { employeeNumber: employeeNumber++, name, country };
  };
}

const getEmployee = getEmployeeFactory();

const employeeOne = getEmployee('Robin', 'Germany');
const employeeTwo = getEmployee('Markus', 'Canada');

const employees = [employeeOne, employeeTwo];

console.log(employees);

// [
//   { employeeNumber: 1, name: 'Robin', country: 'Germany' },
//   { employeeNumber: 2, name: 'Markus', country: 'Canada' },
// ]
```

The new function became a higher-order function, because the first time calling it returns a function. This returned function can be used to create our employee as we did before. However, since **the surrounding function creates a stateful environment around the returned function** -- in this case the stateful employee number -- it is called a closure.

docs)

From the outside, it's not possible to mess with the employee number anymore. It's not in the global scope, but in the closure of our function. Once you create your `getEmployee` function, which you can give any name, the employee number is kept internally as state.

Note: It's worth to mention that the previous implementation of a JavaScript Closure for our example is also called "factory pattern" in software development. Basically the outer function is our factory function and the internal function our function to create an "item" (here employee) out of this factory's specification.

. . .

I hope this brief walkthrough has helped you to understand a JavaScript Closure by example. We started with our problem – the scoping of variables and the keeping track of internal state of a function – and got rid of the problem by implementing a closure for it.

Show Comments

KEEP READING ABOUT TOOLING

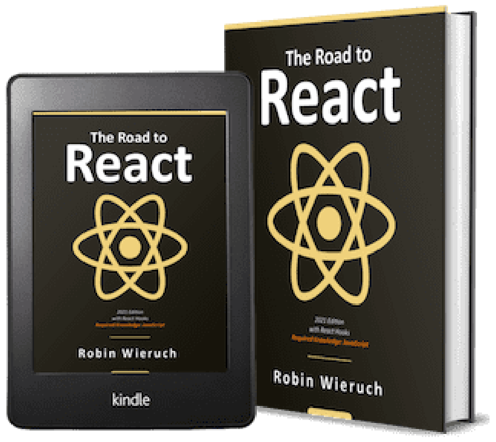
HOW TO USE PRETTIER IN VS CODE

A brief step by step tutorial on how to install and use Prettier in VS Code. Prettier is an opinionated code formatter which ensures one unified code format. It can be used within VS Code by...

HOW TO USE ESLINT IN WEBPACK 5 - SETUP TUTORIAL

So far, you should have a working JavaScript with Webpack application. In this tutorial, we will take this one step further by introducing ESLint for an enforced unified code style without code smells...

rwieruch ABOUT HIRE BLOG COURSES RSS



THE ROAD TO REACT

Learn React by building real world applications. No setup configuration. No tooling. Plain React in 200+ pages of learning material. Learn React like **50.000+ readers**.

GET THE BOOK >

Get it on Amazon.

TAKE PART

NEVER MISS AN ARTICLE ABOUT WEB DEVELOPMENT AND JAVASCRIPT.

✔ Join 50.000+ Developers

✓ Learn JavaScript

✓ Access Tutorials, eBooks and Courses

✓ Personal Development as a Software Engineer

SUBSCRIBE

View our Privacy Policy.