Higher-order functions

Wednesday, April 3, 2024

11:25 AM

Functions that operate on other functions, either by taking them as arguments or by returning them, are called ***higher-order functions***.

***Higher-order functions allow us to abstract over actions, not just values.*** They come in several forms. For example, we can have functions that create new functions.

*function greaterThan(n) { return m => m > n; }*

*let greaterThan10 = greaterThan(10);*

*console.log(greaterThan10(11));*

*// → true*

And we can have functions that change other functions.

*function noisy(f) {*

*return (...args) => {*

*console.log("calling with", args);*

*let result = f(...args);*

*console.log("called with", args, ", returned", result); return result; };*

*}*

*noisy(Math.min)(3, 2, 1);*

*// → calling with [3, 2, 1]*

*// → called with [3, 2, 1] , returned 1*

We can even write functions that provide new types of control flow.

*function repeat(n, action) {*

*for (let i = 0; i < n; i++) { action(i); }*

*}*

*function unless(test, then) {*

*if (!test) then();*

*}*

*repeat(3, n => {*

*unless(n % 2 == 1, () => {*

*console.log(n, "is even");*

*});*

*});*

*// → 0 is even*

*// → 2 is even*

There is a built-in array method, **forEach,** that provides something like a for/of loop as a higher-order function.

*["A", "B"].forEach(l => console.log(l));*

*// → A*

*// → B*

**Filtering arrays**

To find the scripts in the data set that are still in use, the following function might be helpful. It filters out the elements in an array that don’t pass a test.

*function filter(array, test) {*

*let passed = [];*

*for (let element of array) {*

*if (test(element)) {*

*passed.push(element);*

*}*

*}*

*return passed;*

*}*

*console.log(filter(SCRIPTS, script => script.living));*

*// → [{name: "Adlam", …}, …]*

The function uses the argument named test, a function value, to fill a “gap” in the computation—the process of deciding which elements to collect.

Note how the filter function, rather than deleting elements from the existing array, builds up a new array with only the elements that pass the test. This function is ***pure***. It does not modify the array it is given.

Like **forEach**, **filter** is a standard array method

*console.log(SCRIPTS.filter(s => s.direction == "ttb"));*

*// → [{name: "Mongolian", …}, …*

**Transforming with map**

The **map** method transforms an array by applying a function to all of its elements and building a new array from the returned values. The new array will have the same length as the input array, but its content will have been mapped to a new form by the function.

*function map(array, transform) {*

*let mapped = [];*

*for (let element of array) {*

*mapped.push(transform(element));*

*}*

*return mapped;*

*}*

*let rtlScripts = SCRIPTS.filter(s => s.direction == "rtl");*

*console.log(map(rtlScripts, s => s.name));*

*// → ["Adlam", "Arabic", "Imperial Aramaic", …]*

Like **forEach** and **filter**, **map** is a standard array method.

**Summarizing with reduce**

Another common thing to do with arrays is to compute a single value from them.

The higher-order operation that represents this pattern is called **reduce** (sometimes also called fold). It builds a value by repeatedly taking a single element from the array and combining it with the current value.

The parameters to reduce are, apart from the array, a combining function and a start value.

*function reduce(array, combine, start) {*

*let current = start;*

*for (let element of array) {*

*current = combine(current, element);*

*} return current;*

*}*

*console.log(reduce([1, 2, 3, 4], (a, b) => a + b, 0));*

*// → 10*

The standard array method **reduce**, which of course corresponds to this function, has an added convenience. If your array contains at least one element, you are allowed to leave off the start argument.

*console.log([1, 2, 3, 4].reduce((a, b) => a + b));*

*// → 10*

To use **reduce** (twice) to find the script with the most characters, we can write something like this:

*function characterCount(script) {*

*return script.ranges.reduce((count, [from, to]) => {*

*return count + (to - from);*

*}, 0);*

*}*

*console.log(SCRIPTS.reduce((a, b) => {*

*return characterCount(a) < characterCount(b) ? b : a;*

*}));*

*// → {name: "Han", …}*

The characterCount function reduces the ranges assigned to a script by summing their sizes. Note the use of destructuring in the parameter list of the reducer function. The second call to reduce then uses this to find the largest script by repeatedly comparing two scripts and returning the larger one.

**Composability**

Higher-order functions start to shine when you need to ***compose operations***. As an example, let’s write code that finds the average year of origin for living and dead scripts in the data set

*function average(array) {*

*return array.reduce((a, b) => a + b) / array.length;*

*}*

*console.log(Math.round(average( SCRIPTS.filter(s => s.living).map(s => s.year))));*

*// → 1165*

*console.log(Math.round(average( SCRIPTS.filter(s => !s.living).map(s => s.year))));*

*// → 204*

**Strings and character code**

If you have a character (which will be a string of one or two code units), you can use **codePointAt(0)** to get its code.