"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies."

—C.A.R. Hoare,
1980 ACM Turing Award Lecture

5

HIGHER-ORDER FUNCTIONS

A large program is a costly program, and not just because of the time it takes to build. Size almost always involves complexity, and complexity confuses programmers. Confused programmers, in turn, introduce mistakes (*bugs*) into programs. A large program then provides a lot of space for these bugs to hide, making them hard to find.

Let's briefly go back to the final two example programs in the introduction. The first is self-contained and six lines long.

```
let total = 0, count = 1;
while (count <= 10) {
  total += count;
  count += 1;
}
console.log(total);
```

The second relies on two external functions and is one line long.

```
console.log(sum(range(1, 10)));
```

Which one is more likely to contain a bug?

If we count the size of the definitions of `sum` and `range`, the second program is also big—even bigger than the first. But still, I'd argue that it is more likely to be correct.

It is more likely to be correct because the solution is expressed in a vocabulary that corresponds to the problem being solved. Summing a range of numbers isn't about loops and counters. It is about ranges and sums.

The definitions of this vocabulary (the functions `sum` and `range`) will still involve loops, counters, and other incidental details. But because they are expressing simpler concepts than the program as a whole, they are easier to get right.

Abstraction

In the context of programming, these kinds of vocabularies are usually called *abstractions*. Abstractions hide details and give us the ability to talk about problems at a higher (or more abstract) level.

As an analogy, compare these two recipes for pea soup. The first one goes like this:

> Put 1 cup of dried peas per person into a container. Add water until the peas are well covered. Leave the peas in water for at least 12 hours. Take the peas out of the water and put them in a cooking pan. Add 4 cups of water per person. Cover the pan and keep the peas simmering for two hours. Take half an onion per person. Cut it into pieces with a knife. Add it to the peas. Take a stalk of celery per person. Cut it into pieces with a knife. Add it to the peas. Take a carrot per person. Cut it into pieces. With a knife! Add it to the peas. Cook for 10 more minutes.

And this is the second recipe:

> Per person: 1 cup dried split peas, half a chopped onion, a stalk of celery, and a carrot.
>
> Soak peas for 12 hours. Simmer for 2 hours in 4 cups of water (per person). Chop and add vegetables. Cook for 10 more minutes.

The second is shorter and easier to interpret. But you do need to understand a few more cooking-related words, such as *soak*, *simmer*, *chop*, and, I guess, *vegetable*.

When programming, we can't rely on all the words we need to be waiting for us in the dictionary. Thus, we might fall into the pattern of the first recipe—work out the precise steps the computer has to perform, one by one, blind to the higher-level concepts that they express.

It is a useful skill, in programming, to notice when you are working at too low a level of abstraction.

Abstracting Repetition

Plain functions, as we've seen them so far, are a good way to build abstractions. But sometimes they fall short.

It is common for a program to do something a given number of times. You can write a `for` loop for that, like this:

```
for (let i = 0; i < 10; i++) {
  console.log(i);
}
```

Can we abstract "doing something *N* times" as a function? Well, it's easy to write a function that calls `console.log` *N* times.

```
function repeatLog(n) {
  for (let i = 0; i < n; i++) {
    console.log(i);
  }
}
```

But what if we want to do something other than logging the numbers? Since "doing something" can be represented as a function and functions are just values, we can pass our action as a function value.

```
function repeat(n, action) {
  for (let i = 0; i < n; i++) {
    action(i);
  }
}

repeat(3, console.log);
// → 0
// → 1
// → 2
```

We don't have to pass a predefined function to `repeat`. Often, it is easier to create a function value on the spot instead.

```
let labels = [];
repeat(5, i => {
  labels.push(`Unit ${i + 1}`);
});
console.log(labels);
// → ["Unit 1", "Unit 2", "Unit 3", "Unit 4", "Unit 5"]
```

This is structured a little like a `for` loop—it first describes the kind of loop and then provides a body. However, the body is now written as a function value, which is wrapped in the parentheses of the call to `repeat`. This

is why it has to be closed with the closing brace *and* closing parenthesis. In cases like this example, where the body is a single small expression, you could also omit the braces and write the loop on a single line.

Higher-Order Functions

Functions that operate on other functions, either by taking them as arguments or by returning them, are called *higher-order functions*. Since we have already seen that functions are regular values, there is nothing particularly remarkable about the fact that such functions exist. The term comes from mathematics, where the distinction between functions and other values is taken more seriously.

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 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.

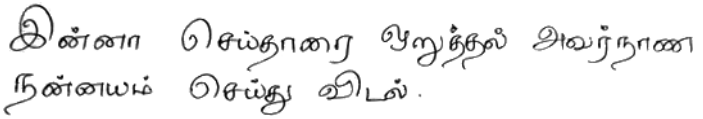
```
["A", "B"].forEach(l => console.log(l));
// → A
// → B
```

Script Data Set

One area where higher-order functions shine is data processing. To process data, we'll need some actual data. This chapter will use a data set about scripts—writing systems such as Latin, Cyrillic, or Arabic.

Remember Unicode from Chapter 1, the system that assigns a number to each character in written language? Most of these characters are associated with a specific script. The standard contains 140 different scripts—81 are still in use today, and 59 are historic.

Though I can fluently read only Latin characters, I appreciate the fact that people are writing texts in at least 80 other writing systems, many of which I wouldn't even recognize. For example, here's a sample of Tamil handwriting:



The example data set contains some pieces of information about the 140 scripts defined in Unicode. It is available in the coding sandbox for this chapter (*https://eloquentjavascript.net/code#5*) as the `SCRIPTS` binding. The binding contains an array of objects, each of which describes a script.

```
{
  name: "Coptic",
  ranges: [[994, 1008], [11392, 11508], [11513, 11520]],
  direction: "ltr",
  year: -200,
  living: false,
  link: "https://en.wikipedia.org/wiki/Coptic_alphabet"
}
```

Such an object tells us the name of the script, the Unicode ranges assigned to it, the direction in which it is written, the (approximate) origin time, whether it is still in use, and a link to more information. The direction may be `"ltr"` for left to right, `"rtl"` for right to left (the way Arabic and

Hebrew text are written), or "ttb" for top to bottom (as with Mongolian writing).

The ranges property contains an array of Unicode character ranges, each of which is a two-element array containing a lower bound and an upper bound. Any character codes within these ranges are assigned to the script. The lower bound is inclusive (code 994 is a Coptic character), and the upper bound is non-inclusive (code 1008 isn't).

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. The example defined the function only to show what it does internally. From now on, we'll use it like this instead:

```
console.log(SCRIPTS.filter(s => s.direction == "ttb"));
// → [{name: "Mongolian", ...}, ...]
```

Transforming with map

Say we have an array of objects representing scripts, produced by filtering the SCRIPTS array somehow. But we want an array of names, which is easier to inspect.

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. Our recurring example, summing a collection of numbers, is an instance of this. Another example is finding the script with the most characters.

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. When summing numbers, you'd start with the number zero and, for each element, add that to the sum.

The parameters to `reduce` are, apart from the array, a combining function and a start value. This function is a little less straightforward than `filter` and `map`, so take a close look at it:

```
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. The method will take the first element of the array as its start value and start reducing at the second element.

```
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.

The Han script has more than 89,000 characters assigned to it in the Unicode standard, making it by far the biggest writing system in the data set. Han is a script (sometimes) used for Chinese, Japanese, and Korean text. Those languages share a lot of characters, though they tend to write them differently. The (US-based) Unicode Consortium decided to treat them as a single writing system to save character codes. This is called *Han unification* and still makes some people very angry.

Composability

Consider how we would have written the previous example (finding the biggest script) without higher-order functions. The code is not that much worse.

```
let biggest = null;
for (let script of SCRIPTS) {
  if (biggest == null ||
      characterCount(biggest) < characterCount(script)) {
    biggest = script;
  }
}
console.log(biggest);
// → {name: "Han", ...}
```

There are a few more bindings, and the program is four lines longer. But it is still very readable.

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))));
// → 1188
console.log(Math.round(average(
  SCRIPTS.filter(s => !s.living).map(s => s.year))));
// → 188
```

So the dead scripts in Unicode are, on average, older than the living ones. This is not a terribly meaningful or surprising statistic. But I hope you'll agree that the code used to compute it isn't hard to read. You can see it as a pipeline: we start with all scripts, filter out the living (or dead) ones, take the years from those, average them, and round the result.

You could definitely also write this computation as one big loop.

```
let total = 0, count = 0;
for (let script of SCRIPTS) {
  if (script.living) {
    total += script.year;
    count += 1;
  }
}
console.log(Math.round(total / count));
// → 1188
```

But it is harder to see what was being computed and how. And because intermediate results aren't represented as coherent values, it'd be a lot more work to extract something like `average` into a separate function.

In terms of what the computer is actually doing, these two approaches are also quite different. The first will build up new arrays when running `filter` and `map`, whereas the second computes only some numbers, doing less work. You can usually afford the readable approach, but if you're processing huge arrays, and doing so many times, the less abstract style might be worth the extra speed.

Strings and Character Codes

One use of the data set would be figuring out what script a piece of text is using. Let's go through a program that does this.

Remember that each script has an array of character code ranges associated with it. So given a character code, we could use a function like this to find the corresponding script (if any):

```
function characterScript(code) {
  for (let script of SCRIPTS) {
    if (script.ranges.some(([from, to]) => {
      return code >= from && code < to;
    })) {
      return script;
    }
  }
  return null;
}

console.log(characterScript(121));
// → {name: "Latin", ...}
```

The `some` method is another higher-order function. It takes a test function and tells you whether that function returns true for any of the elements in the array.

But how do we get the character codes in a string?

In Chapter 1 I mentioned that JavaScript strings are encoded as a sequence of 16-bit numbers. These are called *code units*. A Unicode character code was initially supposed to fit within such a unit (which gives you a little over 65,000 characters). When it became clear that wasn't going to be enough, many people balked at the need to use more memory per character. To address these concerns, UTF-16, the format used by JavaScript strings, was invented. It describes most common characters using a single 16-bit code unit but uses a pair of two such units for others.

UTF-16 is generally considered a bad idea today. It seems almost intentionally designed to invite mistakes. It's easy to write programs that pretend code units and characters are the same thing. And if your language doesn't use two-unit characters, that will appear to work just fine. But as soon as someone tries to use such a program with some less common Chinese characters, it breaks. Fortunately, with the advent of emoji, everybody has started using two-unit characters, and the burden of dealing with such problems is more fairly distributed.

Unfortunately, obvious operations on JavaScript strings, such as getting their length through the `length` property and accessing their content using square brackets, deal only with code units.

```
// Two emoji characters, horse and shoe
let horseShoe = "🏇👟";
console.log(horseShoe.length);
// → 4
console.log(horseShoe[0]);
// → (Invalid half-character)
console.log(horseShoe.charCodeAt(0));
// → 55357 (Code of the half-character)
console.log(horseShoe.codePointAt(0));
// → 128052 (Actual code for horse emoji)
```

JavaScript's `charCodeAt` method gives you a code unit, not a full character code. The `codePointAt` method, added later, does give a full Unicode character. So we could use that to get characters from a string. But the argument passed to `codePointAt` is still an index into the sequence of code units. So to run over all characters in a string, we'd still need to deal with the question of whether a character takes up one or two code units.

In "Array Loops" on page 69, I mentioned that a `for/of` loop can also be used on strings. Like `codePointAt`, this type of loop was introduced at a time where people were acutely aware of the problems with UTF-16. When you use it to loop over a string, it gives you real characters, not code units.

```
let roseDragon = "🌹🐉";
for (let char of roseDragon) {
  console.log(char);
}
// → 🌹
// → 🐉
```

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.

Recognizing Text

We have a `characterScript` function and a way to correctly loop over characters. The next step is to count the characters that belong to each script. The following counting abstraction will be useful there:

```
function countBy(items, groupName) {
  let counts = [];
  for (let item of items) {
    let name = groupName(item);
    let known = counts.findIndex(c => c.name == name);
    if (known == -1) {
      counts.push({name, count: 1});
    } else {
      counts[known].count++;
    }
  }
```

```
  return counts;
}

console.log(countBy([1, 2, 3, 4, 5], n => n > 2));
// → [{name: false, count: 2}, {name: true, count: 3}]
```

The `countBy` function expects a collection (anything that we can loop over with `for/of`) and a function that computes a group name for a given element. It returns an array of objects, each of which names a group and tells you the number of elements that were found in that group.

It uses another array method—`findIndex`. This method is somewhat like `indexOf`, but instead of looking for a specific value, it finds the first value for which the given function returns true. Like `indexOf`, it returns –1 when no such element is found.

Using `countBy`, we can write the function that tells us which scripts are used in a piece of text.

```
function textScripts(text) {
  let scripts = countBy(text, char => {
    let script = characterScript(char.codePointAt(0));
    return script ? script.name : "none";
  }).filter(({name}) => name != "none");

  let total = scripts.reduce((n, {count}) => n + count, 0);
  if (total == 0) return "No scripts found";

  return scripts.map(({name, count}) => {
    return `${Math.round(count * 100 / total)}% ${name}`;
  }).join(", ");
}

console.log(textScripts('英国的狗说"woof", 俄罗斯的狗说"тяв"'));
// → 61% Han, 22% Latin, 17% Cyrillic
```

The function first counts the characters by name, using `characterScript` to assign them a name and falling back to the string `"none"` for characters that aren't part of any script. The `filter` call drops the entry for `"none"` from the resulting array since we aren't interested in those characters.

To be able to compute percentages, we first need the total number of characters that belong to a script, which we can compute with `reduce`. If no such characters are found, the function returns a specific string. Otherwise, it transforms the counting entries into readable strings with `map` and then combines them with `join`.

Summary

Being able to pass function values to other functions is a deeply useful aspect of JavaScript. It allows us to write functions that model computations with "gaps" in them. The code that calls these functions can fill in the gaps by providing function values.

Arrays provide a number of useful higher-order methods. You can use `forEach` to loop over the elements in an array. The `filter` method returns a new array containing only the elements that pass the predicate function. Transforming an array by putting each element through a function is done with `map`. You can use `reduce` to combine all the elements in an array into a single value. The `some` method tests whether any element matches a given predicate function. And `findIndex` finds the position of the first element that matches a predicate.

Exercises

Flattening

Use the `reduce` method in combination with the `concat` method to "flatten" an array of arrays into a single array that has all the elements of the original arrays.

Your Own Loop

Write a higher-order function `loop` that provides something like a `for` loop statement. It takes a value, a test function, an update function, and a body function. Each iteration, it first runs the test function on the current loop value and stops if that returns false. Then it calls the body function, giving it the current value. Finally, it calls the update function to create a new value and starts from the beginning.

When defining the function, you can use a regular loop to do the actual looping.

Everything

Analogous to the `some` method, arrays also have an `every` method. This one returns true when the given function returns true for *every* element in the array. In a way, `some` is a version of the `||` operator that acts on arrays, and `every` is like the `&&` operator.

Implement `every` as a function that takes an array and a predicate function as parameters. Write two versions, one using a loop and one using the `some` method.

Dominant Writing Direction

Write a function that computes the dominant writing direction in a string of text. Remember that each script object has a `direction` property that can be `"ltr"` (left to right), `"rtl"` (right to left), or `"ttb"` (top to bottom).

The dominant direction is the direction of a majority of the characters that have a script associated with them. The `characterScript` and `countBy` functions defined earlier in the chapter are probably useful here.