13 How to Write General Loops

Function likes **map()** and **reduce()** are used over and over in modern JavaScript programming to process collections of data. They are highly reusable, because they do a commonly needed looping pattern, but let you, the programmer, specify the details of what should happen in the loop.

The key to this reusability is that **map()** and **reduce()** take functions are arguments. This is called *higher-order functional programming*. Once, only a few languages, such as Lisp, could do this, but now most modern languages support this feature. JavaScript did from the start. This ability is used not only for looping in JavaScript, but also for things like *asynchronous calls* to pass *callbacks* to functions like fetch.

Defining a function that takes functions as arguments is not hard. No new syntax is needed. You just have to remove a mental assumption you might have about what's possible.

For example, while **map()** and **reduce()** let you apply a function to a single array, sometimes you want to apply a function to two arrays. For example, you would like to take two arrays and return a new array that is as long as the shortest of the two arrays, and contains the sum of the corresponding elements. E.g,.

```
addArrays([1, 2, 3, 4], [5, 6, 7])
```

should return [6, 8, 10]. Because one list may be shorter than the other, we need to use a more complicated loop than a simple **map()**. Here's one possible solution:

```
function addArrays(a, b) {
  var lst = [];
  var len = Math.min(a.length, b.length);
  for (var i = 0; i < len; ++i) {
    lst.push(a[i] + b[i]);
  }
  return lst;
}
```

If we wanted to collect a list of products, we'd have to write another function

```
function multiplyArrays(a, b) {
  var lst = [];
  var len = Math.min(a.length, b.length);
  for (var i = 0; i < len; ++i) {
    lst.push(a[i] * b[i]);
  }
  return lst;
}
```

If we have some other way of combining, we'd have to write the same loop again.

But if we define a function that takes the function to apply to the two arrays, we can write this loop once, and never have to worry about it again. Using the same kind of code, the function might look like this:

```
function mapArrays(fn, a, b) {
  var lst = [];
  var len = Math.min(a.length, b.length);
  for (var i = 0; i < len; ++i) {
    lst.push(fn(a[i], b[i]));
  }
  return lst;
}
```

Here the parameter variable **fn** is assumed to be a function of two arguments, because, inside the loop, we call it like we call any function that takes two arguments. As long as we pass a function of two arguments as the first argument to **mapArrays()**, this code should work.

Now we can add two arrays with:

```
mapArrays(function(a, b) { return a + b; }, [1, 2, 3, 4], [5, 6, 7])
```

Similarly we can multiply two arrays with:

```
mapArrays(function(a, b) { return a * b; }, [1, 2, 3, 4], [5, 6, 7])
```

Any function will work. Try this:

```
mapArrays(function(a, b) { return [a, b]; }, [1, 2, 3, 4], [5, 6, 7])
```