Loops and Arrays

Foundations Course

Introduction

One of the challenges of programming is repetition. While some manual repetition is completely fine, sometimes we want to streamline repeated instructions to be more readable. Another challenge is dealing with large amounts of data. For example, if you want to store the names of all the students in your class, how would you do it? You could create a variable for each name, but that would be tedious and inefficient. It'll also be hard to manage and update. What if you want to change or access the names later?

Luckily, there are ways to handle these problems. In this lesson, you'll learn about loops, which are control structures that allow you to execute a block of code repeatedly. You'll also learn about arrays, which are data structures that can store multiple values in a single variable. Arrays are very useful for organizing and manipulating large amounts of data. Loops can often be very handy for performing the same operation on each element of an array. Finally, you'll be introduced to test-driven development (TDD), which is the practice of writing tests for your code before you write the code itself.

Lesson overview

This section contains a general overview of topics that you will learn in this lesson.

- Using loops.
- Using arrays.
- Using array methods such as `map`, `filter` and `reduce`.
- Getting your hands dirty with TDD exercises.

Loops

Computers don't get tired, and they're really, *really* fast! For that reason, they are well suited to solving problems that involve doing calculations multiple times. In some cases, a computer will be able to repeat a task *thousands* or even *millions* of times in just a few short seconds where it might take a human many hours (obviously, speed here depends on the complexity of the calculation and the speed of the computer itself).

To repeat a set of instructions multiple times, we can use a **loop**. Go through the following resources on loops. Some of them will contain examples that use arrays, which we will go through in more detail in the next section, but for now it will suffice to know that arrays are just lists of items.

1. Read MDN's Looping Code. It's a longer one, but make sure you tackle the exercises near the bottom of the page.
2. Once again, same info, slightly different context from JavaScript.info's "Loops: While and For". Be sure to do the exercises at the end of the page. You learn best by doing.

Arrays

Strings and numbers may be our building blocks, but as your scripts get more complex, you're going to need a way to deal with large quantities of them. Luckily, JavaScript has a couple of data types that are used for just that. An Array is an ordered collection of items (strings, numbers, or other things). You may recognize that some of the array methods demonstrated in some of the following resources do similar things to loops. Sometimes you may need more control via an actual loop but sometimes the appropriate array method does the job just fine and you prefer the way it reads.

1. This JavaScript Arrays crash course explains an overview of arrays in JavaScript in about 6 minutes.
2. Read through this intro to arrays from JavaScript.info. You do not need to do the exercises at the end of the page.
3. Read through JavaScript.info's array method guide for a deeper look at some further array methods in JavaScript. Again, you do not need to do the exercises at the end of the page; we will come back to these later.
4. Bookmark MDN's Array documentation. You do not need to memorize anything here. This is where you'll find all built-in array properties and methods and their documentation. This will likely be something you regularly revisit as you code and solve problems.

Some examples of array magic

Besides being a quick and handy way to store data, arrays also have a set of functions for manipulating that data in very powerful ways. Once you get more experience with these functions, you will start to see ways to use them all over the place! There are really only a handful of these functions, but as you'll soon see, the possibilities of what you can do with them are near endless.

As an example of what we mean, let's consider a `sumOfTripledEvens` function. It will:

1. Take in an array.
2. For every even number, it will triple it.
3. Then it will sum all those even numbers.

Can you think of how you could implement a function like that using pseudocode?

1. We need to perform an operation only on the even numbers.
2. We need to transform *those* numbers by multiplying them by 3.
3. Finally, we need to add the result up from the previous transformation.

So using that logic, you may end up implementing something like this:

```
function sumOfTripledEvens(array) {
  let sum = 0;
  for (let i = 0; i < array.length; i++) {
    // Step 1: If the element is an even number
    if (array[i] % 2 === 0) {
      // Step 2: Multiply this number by three
      const tripleEvenNumber = array[i] * 3;

      // Step 3: Add the new number to the total
      sum += tripleEvenNumber;
    }
  }
  return sum;
}
```

In the above code, there are 3 important snippets to consider:

- `if (array[i] % 2 === 0)`: checks if a given number is even.
- `const tripleEvenNumber = array[i] * 3`: gets triple that number.
- `sum += tripleEvenNumber`: increments the sum by that number.

Every single piece solves a crucial problem with our code. However, we can do the same thing with some array methods and sometimes, the result is a little easier to read and understand. Let's take a moment to see how we might be able to use some.

The map method

`map` is one such function. It expects a `callback` as an argument, which is a fancy way to say "I want you to pass another function as an argument to my function".

Let's say we had a function `addOne`, which takes in `num` as an argument and outputs that `num` increased by 1, and an array of numbers, `[1, 2, 3, 4, 5]`. Let's say we'd like to increment all of these numbers by 1 using our `addOne` function.

Instead of making a `for` loop and iterating over the above array, we could use our `map` array method instead, which **automatically** iterates over an array for us. We don't need to do any extra work aside from simply passing the function we want to use in:

```
function addOne(num) {
  return num + 1;
}
const arr = [1, 2, 3, 4, 5];
const mappedArr = arr.map(addOne);
console.log(mappedArr); // Outputs [2, 3, 4, 5, 6]
```

`map` returns a new array and does not change the original array.

```
// The original array has not been changed!
console.log(arr); // Outputs [1, 2, 3, 4, 5]
```

Using `map` in this way can be more elegant than writing a `for` loop and iterating over the array. But we can do even better. Since we're not using `addOne` anywhere else and it's a simple function, we can define it inline using an arrow function, right inside of `map` like so:

```
const arr = [1, 2, 3, 4, 5];
const mappedArr = arr.map((num) => num + 1);
console.log(mappedArr); // Outputs [2, 3, 4, 5, 6]
```

The filter method

`filter` is somewhat similar to `map`. It still iterates over the array and applies the callback function on every item. However, instead of transforming the values in the array, it returns a new array where each item is only included *if* the callback function returns `true` for it.

Let's say we had a function, `isOdd` that returns either `true` if a number is odd or `false` if it isn't.

The `filter` method expects the `callback` to return either `true` or `false`. If it returns `true`, the value is included in the output. Otherwise, it isn't. Consider the array from our previous example, `[1, 2, 3, 4, 5]`. If we wanted to remove all even numbers from this array, we could use `.filter()` like this:

```
function isOdd(num) {
  return num % 2 !== 0;
}
const arr = [1, 2, 3, 4, 5];
const oddNums = arr.filter(isOdd);
console.log(oddNums); // Outputs [1, 3, 5];
console.log(arr); // Outputs [1, 2, 3, 4, 5], original array is not
```

- `filter` will iterate through `arr` and pass **every value** into the `isOdd` callback function, one at a time.
- `isOdd` will return `true` when the value is odd, which means this value is included in the output.
- If it's an even number, `isOdd` will return `false` and not include it in the final output.

The reduce method

Finally, let's say that we wanted to multiply all of the numbers in our `arr` together like this: `1 * 2 * 3 * 4 * 5`. First, we'd have to declare a variable `total` and initialize it to 1. Then, we'd iterate through the array with a `for` loop and multiply the `total` by the current number.

But we don't actually need to do all of that; we have our `reduce` method that will do the job for us. Just like `.map()` and `.filter()`, it expects a callback function. However, there are two key differences with this array method:

- The callback function takes two arguments instead of one. The first argument is the `accumulator`, which is the current value of the result *at that point in the loop*. The first time through, this value will either be set to the `initialValue` (described in the next bullet point), or the first element in the array if no `initialValue` is provided. The second argument for the callback is the `current` value, which is the item currently being iterated on.
- `reduce` itself also takes in an `initialValue` as an optional second argument (after the callback), which helps when we don't want our initial value to be the first element in the array. For instance, if we wanted to sum all numbers in an array, we could call reduce without an `initialValue`, but if we wanted to sum all numbers in an array and add 10, we could use 10 as our `initialValue`.

```
const arr = [1, 2, 3, 4, 5];
const productOfAllNums = arr.reduce((total, currentItem) => {
  return total * currentItem;
}, 1);
console.log(productOfAllNums); // Outputs 120;
console.log(arr); // Outputs [1, 2, 3, 4, 5]
```

In the above function, we:

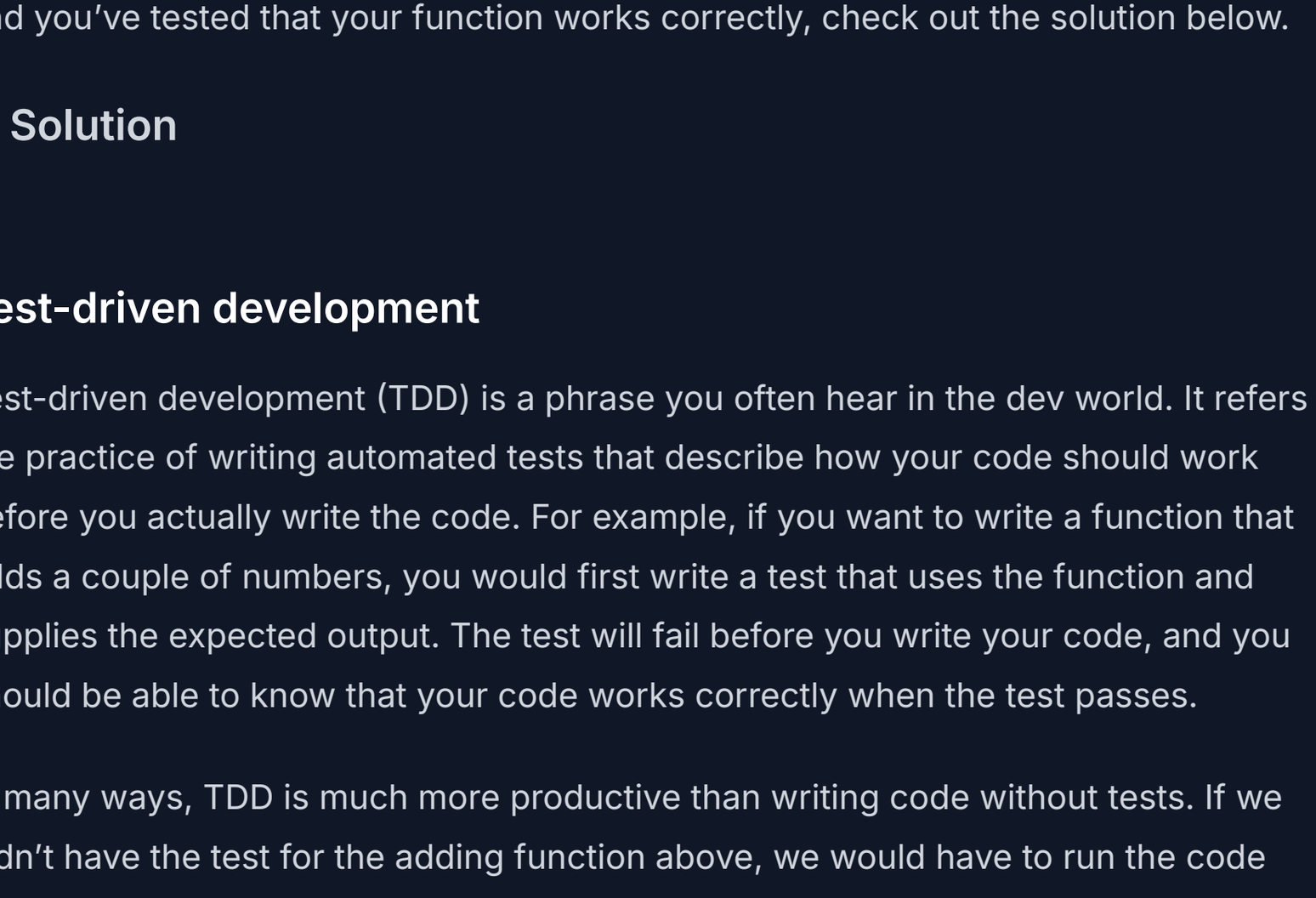
- Pass in a callback function, which is `(total, currentItem) => total * currentItem`.
- Initialize `total` to `1` in the second argument.

So what `.reduce()` will do is go through every element in `arr` and apply the `callback` function to it. It updates `total` without actually changing the array itself. After it's done, it returns `total`.

Summary

You've learnt about three powerful array methods which are `map`, `filter` and `reduce`. They can allow us to write potentially more readable code which in turn may be less prone to bugs.

For a quick recap of these array methods, consider this picture which should visually explain them in terms of sandwiches:



Let's do some quick practice before your assignment! Rewrite the `sumOfTripledEvens(array)` function using these three methods. Once you are finished and you've tested that your function works correctly, check out the solution below.

▶ Solution

Test-driven development

Test-driven development (TDD) is a phrase you often hear in the dev world. It refers to the practice of writing automated tests that describe how your code should work before you actually write the code. For example, if you want to write a function that adds a couple of numbers, you would first write a test that uses the function and supplies the expected output. The test will fail before you write your code, and you should be able to know that your code works correctly when the test passes.

In many ways, TDD is much more productive than writing code without tests. If we didn't have the test for the adding function above, we would have to run the code ourselves over and over, plugging in different numbers until we were sure that it was working... not a big deal for a basic `add(2, 2)`, but imagine having to do that for more complicated functions, like checking whether or not someone has won a game of tic tac toe: (`game_win(["o", null,"x",null,"x",null,"x", "o", "o"])`). If you didn't do TDD, then you might actually have to play multiple games against yourself just to test if the function was working correctly!

We will teach you the art of actually writing these tests later in the course. For now, you will continue to work on the JavaScript exercises from before.

Assignment

1. Go to the array methods exercises at the end of the JavaScript.info array methods article and do the following exercises only:
   - `Translate border-left-width to borderLeftWidth`
   - `Filter range`
   - `Filter range "in place"`
   - `Sort in decreasing order`
   - `Copy and sort array`
   - `Shuffle an array`
   - `Filter unique array members`
2. Go back to the JavaScript exercises repository's `foundations/` directory that we introduced in the Data Types and Conditionals assignment. Review each README file prior to completing the following exercises in order (it may have been a while, so take a moment to remind yourself of the repo's own README for general instructions):
   - `06_repeatString`
   - `07_reverseString`
   - `08_removeFromArray`
   - `09_sumAll`
   - `10_leapYears`
   - `11_tempConversion`

   Note: Solutions for these exercises can be found in the `solution` folder of each exercise.

Knowledge check

The following questions are an opportunity to reflect on key topics in this lesson. If you can't answer a question, click on it to review the material, but keep in mind you are not expected to memorize or master this knowledge.

- What are loops useful for?
- What is the break statement?
- What is the continue statement?
- What is an array?
- What are arrays useful for?
- How do you access or change an array element?
- What are some useful array methods?
- What is the advantage of writing automated tests?