Data Structure APIs

A brief overview of APIs as they relate to JavaScript data structures.

Data structures are all about choosing the right tool for the job. Do you need to store data in an ordered way, or do you just need to be able to store it and retrieve it quickly? What's more important to your use case: how fast the data structure performs, or how much memory it takes up? Different data structures all have advantages, disadvantages, and use cases, and that's the whole reason that there are different data structures!

Consider the `Array` in JavaScript. It's a really great data structure for storing ordered data because you can retrieve elements by index number. If you want the first element of an array, all you need to do is fetch it with index 0: `arrayName[0]`. It also provides all sorts of helpful methods for manipulating elements, such as `.push()`, `.pop()`, `.sort()`, and more. However, if you want to find out if a particular element exists in an array, you may need to iterate through the entire array.

What if I asked you to keep track of a series of numbers as I gave them to you, and then asked at the end whether I'd given you a particular number, you could probably do that in your memory. But if I asked you to do that in a computer program, you'd have to make choices about how to store the data. Let's look at two possibilities of how we'd build `storeNumber()` and `doYouHaveThisNumber()` functions. Given the following list of numbers:

```
1, 250, -42, 0.4, 17
```

How might you store these numbers if I gave you each at a time? You might use an array:

```
const listOfNumbers = [];

const storeNumber = num => listOfNumbers.push(num);

const doYouHaveThisNumber = num => listOfNumbers.includes(num);
```

In this program, `storeNumber()` adds a number to the array, and `doYouHaveThisNumber()` returns `true` if that number exists in the array, and `false` otherwise. Looks pretty good, but what if you had 10000000 numbers? `doYouHaveThisNumber()` might start getting pretty slow, since `Array.prototype.includes()` iterates through the entire array until it finds the input value.

Let's try using another built-in data type in JavaScript, the `Object`. Since all we want to keep track of is whether we received a particular number, we can just store those numbers in an object, and set their values to `true` if we received them:

```
const receivedNumbers = {};

const storeNumber = num => receivedNumbers[num] = true;

const doYouHaveThisNumber = num => receivedNumbers[num] === true;
```

In this case, we'll have the same result on the outside, but because retrieving a value from an object is much faster than iterating through an array, the overall result will be faster.

In both cases, the *public API* of the code, meaning the parts of the code that we want the end-user to interact with, remained the same: we had two functions, `storeNumber()` and `doYouHaveThisNumber()`. The underlying *implementation*, or the way the functionality was actually achieved, is what altered.

What is an API?

API is an acronym for application programming interface. An API allows end-users to access properties and methods of data structures easily and without needing to do the "behind the scenes" work.

For example, if you want to add a new element to the end of an array, you don't need to loop through the entire array, counting how many elements there are, and then setting `myArray[currentCount + 1]` equal to the new value. Instead, you can just call `.push()` with the value you want to add. As a JavaScript programmer, you don't actually need to know the actual strategy, or the *underlying implementation*, of how `.push()` added an element to the end of the array in order to use it.

The API of arrays provides lots of useful functionality, from adding and removing elements to the start and end of the array, to iterator methods that call a function on each element. If you wanted to find the smallest number in an array of numbers, however, you'd have to implement that functionality yourself.

Creating Your Own APIs

As you build your own data structures, you will implement the functionality to create public APIs. As in the example of `storeNumber()` and `doYouHaveThisNumber()`,

the same public API can be implemented in different ways, so it's important to think about the advantages and disadvantages of different implementations.

An API is like a message to end-users. Some languages have classes that can have methods or fields that are either public (can be called from anywhere) or private (can only be called from within the class). Public methods are the ones that end-users of that class can call, and private methods are only used by the class itself. JavaScript doesn't really support this concept, so properties that aren't meant to be public are often preceded by an underscore `_`. Let's look at an example where we want to build a data structure with a restricted API.

A *stack* is a data structure that only allows data to be added (*pushed*) or removed (*popped*) from the "top" of the stack. It just so happens that we could use an array as a stack, since it already has a `.push()` and `.pop()` method! However, arrays also allow you to add elements to the beginning or randomly access elements by index.

We're not going to cover all the ins and outs of the stack data structure right now, but to demonstrate public API vs implementation, let's build a quick custom `Stack` class:

```
class Stack {
  constructor() {
    this._array = [];
  }
}
```

In `Stack`, the array itself is stored as `_array`, so it's a signal to other developers that to use the `Stack` as intended, they shouldn't need to access it directly. From there, we can implement the `.push()` and `.pop()` methods:

```
class Stack {
  constructor() {
    this._array = [];
  }

  push(newValue) {
    this._array.push(newValue);
  }

  pop() {
    return this._array.pop();
  }
}
```

Now we've created a `Stack` data structure that limits direct interaction with the underlying data to `.push()` and `.pop()`. A developer could still access our underlying array to do other manipulation:

```
const stack = new Stack();
stack._array.unshift('value');
```

but they would then be breaking the intended behavior of the `Stack` class. The whole point of a public API is that we offer functionality to other end-users. If somebody were using our `Stack` class in a program, we could totally change the underlying implementation, and as long as the end-user API remained the same, their program should continue to function.

As you build your own classes and data structures, it's important to keep in mind this distinction between implementation (what does this need internally to do its job) and the outside API (how should users of this actually interact with it?).