# 3

# ARRAYS

So far we've learned about numbers and strings, which are types of data that you can store and use in your programs. But numbers and strings are kind of boring. There's not a lot that you can do with a string on its own. JavaScript lets you create and group together data in more interesting ways with *arrays*. An array is just a list of other JavaScript data values.

For example, if your friend asked you what your three favorite dinosaurs were, you could create an array with the names of those dinosaurs, in order:

```
var myTopThreeDinosaurs = ["T-Rex", "Velociraptor", "Stegosaurus"];
```

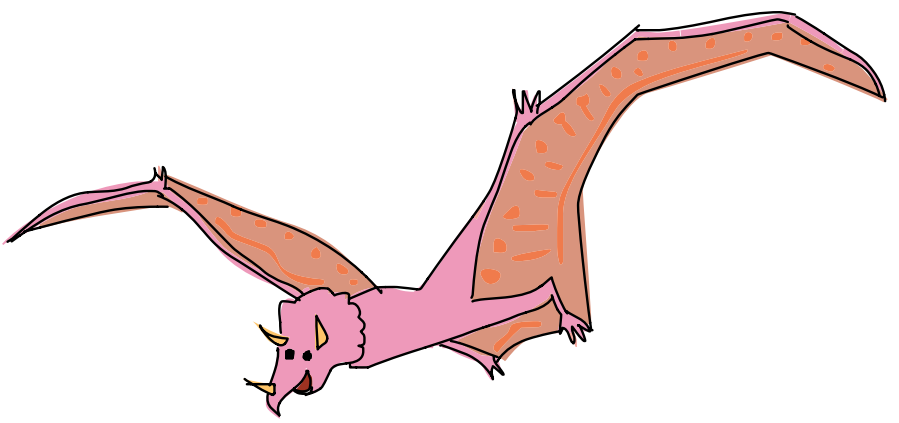
So instead of giving your friend three separate strings, you can just use the single array `myTopThreeDinosaurs`.

## WHY SHOULD YOU CARE ABOUT ARRAYS?

Let's look at dinosaurs again. Say you want to use a program to keep track of the many kinds of dinosaurs you know about. You could create a variable for each dinosaur, like this:

```
var dinosaur1 = "T-Rex";
var dinosaur2 = "Velociraptor";
var dinosaur3 = "Stegosaurus";
var dinosaur4 = "Triceratops";
var dinosaur5 = "Brachiosaurus";
var dinosaur6 = "Pteranodon";
var dinosaur7 = "Apatosaurus";
var dinosaur8 = "Diplodocus";
var dinosaur9 = "Compsognathus";
```

This list is pretty awkward to use, though, because you have nine different variables when you could have just one. Imagine if you were keeping track of 1000 dinosaurs! You'd need to create 1000 separate variables, which would be almost impossible to work with.



It's like if you had a shopping list, but every item was on a different piece of paper. You'd have one piece of paper that said "eggs," another piece that said "bread," and another piece that said "oranges." Most people would write the full list of things they want to buy on a single piece of paper. Wouldn't it be much easier if you could group all nine dinosaurs together in just one place?

You can, and that's where arrays come in.

## CREATING AN ARRAY

To create an array, you just use square brackets, `[]`. In fact, an empty array is simply a pair of square brackets, like this:

```
[];
[]
```

But who cares about an empty array? Let's fill it with our dinosaurs!

To create an array with values in it, enter the values, separated by commas, between the square brackets. We can call the individual values in an array *items* or *elements*. In this example, our elements will be strings (the names of our favorite dinosaurs), so we'll write them with quote marks. We'll store the array in a variable called `dinosaurs`:

```
var dinosaurs = ["T-Rex", "Velociraptor", "Stegosaurus", ↵
"Triceratops", "Brachiosaurus", "Pteranodon", "Apatosaurus", ↵
"Diplodocus", "Compsognathus"];
```

**NOTE** *Because this is a book and the page is only so wide, we can't actually fit the whole array on one line. The ↵ is to show where we've put the code onto an extra line because the page is too narrow. When you type this into your computer, you can type it all on one line.*

Long lists can be hard to read on one line, but luckily that's not the only way to format (or lay out) an array. You can also format an array with an opening square bracket on one line, the

list of items in the array each on a new line, and a closing square bracket, like this:

```
var dinosaurs = [
  "T-Rex",
  "Velociraptor",
  "Stegosaurus",
  "Triceratops",
  "Brachiosaurus",
  "Pteranodon",
  "Apatosaurus",
  "Diplodocus",
  "Compsognathus"
];
```

If you want to type this into your browser console, you'll need to hold down the SHIFT key when you press the ENTER key for each new line. Otherwise the JavaScript interpreter will think you're trying to execute the current, incomplete, line. While we're working in the interpreter, it's easier to write arrays on one line.

Whether you choose to format the items in an array on one line or on separate lines, it's all the same to JavaScript. However many line breaks you use, JavaScript just sees an array—in this example, an array containing nine strings.

## ACCESSING AN ARRAY'S ELEMENTS

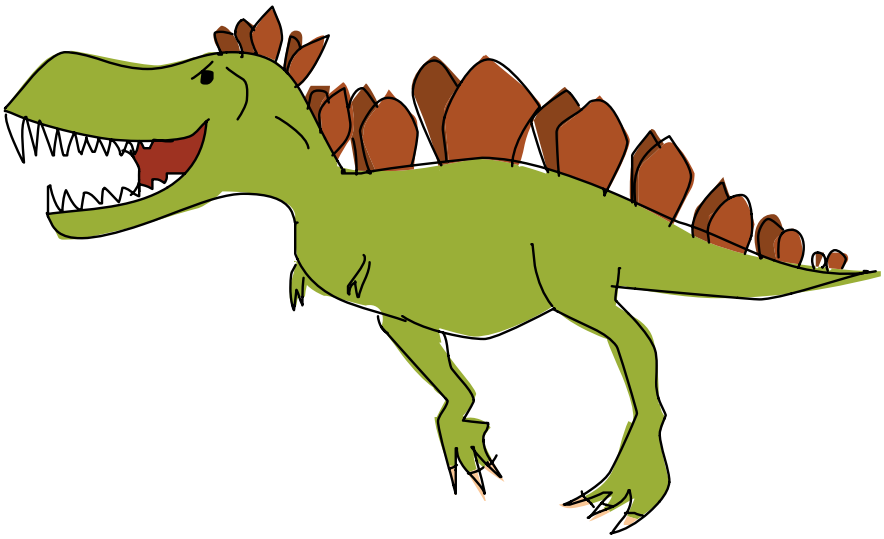
When it's time to access elements in an array, you use square brackets with the *index* of the element you want, as you can see in the following example:

```
dinosaurs[0];
"T-Rex"
dinosaurs[3];
"Triceratops"
```

An *index* is the number that corresponds to (or matches) the spot in the array where a value is stored. Just as with strings, the first element in an array is at index 0, the second is at index 1, the third at index 2, and so on. That's why asking for index 0 from the `dinosaurs` array returns `"T-Rex"` (which is first in the list), and index 3 returns `"Triceratops"` (which is fourth in the list).

It's useful to be able to access individual elements from an array. For example, if you just wanted to show someone your absolute favorite dinosaur, you wouldn't need the whole `dinosaurs` array. Instead you would just want the first element:

```
dinosaurs[0];
"T-Rex"
```



## SETTING OR CHANGING ELEMENTS IN AN ARRAY

You can use indexes in square brackets to set, change, or even add elements to an array. For example, to replace the first element in the `dinosaurs` array (`"T-Rex"`) with `"Tyrannosaurus Rex"`, you could do this:

```
dinosaurs[0] = "Tyrannosaurus Rex";
```

After you've done that, the `dinosaurs` array would look like this:

```
["Tyrannosaurus Rex", "Velociraptor", "Stegosaurus", "Triceratops",
"Brachiosaurus", "Pteranodon", "Apatosaurus", "Diplodocus",
"Compsognathus"]
```

You can also use square brackets with indexes to add new elements to an array. For example, here's how you could create the `dinosaurs` array by setting each element individually with square brackets:

```
var dinosaurs = [];
dinosaurs[0] = "T-Rex";
dinosaurs[1] = "Velociraptor";
dinosaurs[2] = "Stegosaurus";
dinosaurs[3] = "Triceratops";
dinosaurs[4] = "Brachiosaurus";
dinosaurs[5] = "Pteranodon";
dinosaurs[6] = "Apatosaurus";
dinosaurs[7] = "Diplodocus";
dinosaurs[8] = "Compsognathus";

dinosaurs;
["T-Rex", "Velociraptor", "Stegosaurus", "Triceratops",
"Brachiosaurus", "Pteranodon", "Apatosaurus", "Diplodocus",
"Compsognathus"]
```

First we create an empty array with `var dinosaurs = []`. Then, with each following line we add a value to the list with a series of `dinosaurs[]` entries, from index 0 to index 8. Once we finish the list, we can view the array (by typing `dinosaurs;`). We see that JavaScript has stored all the names ordered according to the indexes.

You can actually add an element at any index you want. For example, to add a new (made-up) dinosaur at index 33, you could write the following:

```
dinosaurs[33] = "Philosoraptor";

dinosaurs;
["T-Rex", "Velociraptor", "Stegosaurus", "Triceratops",
"Brachiosaurus", "Pteranodon", "Apatosaurus", "Diplodocus",
"Compsognathus", undefined × 24 "Philosoraptor"]
```

The elements between indexes 8 and 33 will be `undefined`. When you output the array, Chrome helpfully tells you how many elements were `undefined`, rather than listing them all individually.

## MIXING DATA TYPES IN AN ARRAY

Array elements don't all have to be the same type. For example, the next array contains a number (3), a string ("dinosaurs"), an array (["triceratops", "stegosaurus", 3627.5]), and another number (10):

```
var dinosaursAndNumbers = [3, "dinosaurs", ["triceratops", ↵
"stegosaurus", 3627.5], 10];
```

To access an individual element in this array's inner array, you would just use a second set of square brackets. For example, while `dinosaursAndNumbers[2];` returns the entire inner array, `dinosaursAndNumbers[2][0];` returns only the first element of that inner array, which is "triceratops".

```
dinosaursAndNumbers[2];
["triceratops", "stegosaurus", 3627.5]
dinosaursAndNumbers[2][0];
"triceratops"
```

When we type `dinosaursAndNumbers[2][0];`, we tell JavaScript to look at index 2 of the array `dinosaursAndNumbers`, which contains the array `["triceratops", "stegosaurus", 3627.5]`, and to return the value at index 0 of that second array. Index 0 is the first value of the second array, which is "triceratops". Figure 3-1 shows the index positions for this array.

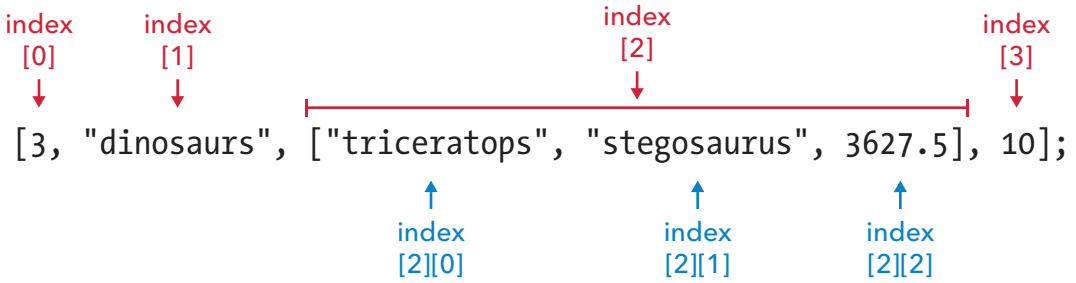



*Figure 3-1: The index positions of the main array are labeled in red, and the indexes of the inner array are labeled in blue.*

# WORKING WITH ARRAYS

*Properties* and *methods* help you work with arrays. Properties generally tell you something about the array, and methods usually do something to change the array or return a new array. Let's have a look.

## FINDING THE LENGTH OF AN ARRAY

Sometimes it's useful to know how many elements there are in an array. For example, if you kept adding dinosaurs to your `dinosaurs` array, you might forget how many dinosaurs you have.

The `length` property of an array tells you how many elements there are in the array. To find the length of an array, just add `.length` to the end of its name. Let's try it out. First we'll make a new array with three elements:

```
var maniacs = ["Yakko", "Wakko", "Dot"];
maniacs[0];
"Yakko"
maniacs[1];
"Wakko"
maniacs[2];
"Dot"
```

To find the length of the array, add `.length` to `maniacs`:

```
maniacs.length;
3
```

JavaScript tells us that there are 3 elements in the array, and we already know they have the index positions 0, 1, and 2. This gives us a useful piece of information: the last index in an array is always the same number as the length of the array minus 1. This means that there is an easy way to access the last element in an array, however long that array is:

```
maniacs[maniacs.length - 1];
"Dot"
```

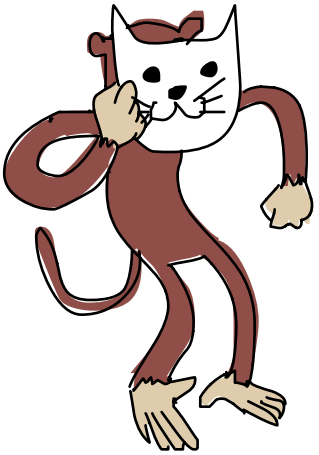
Here, we're asking JavaScript for an element from our array. But instead of entering an index number in the square brackets, we use a little bit of math: the length of the array minus 1. JavaScript finds `maniacs.length`, gets 3, and then subtracts 1 to get 2. Then it returns the element from index 2—the last maniac in the array, `"Dot"`.

## ADDING ELEMENTS TO AN ARRAY

To add an element to the end of an array, you can use the `push` method. Add `.push` to the array name, followed by the element you want to add inside parentheses, like this:

```
var animals = [];
animals.push("Cat");
1
animals.push("Dog");
2
animals.push("Llama");
3
animals;
["Cat", "Dog", "Llama"]
animals.length;
3
```

Here we create an empty array with `var animals = [];`, and then use the `push` method to add `"Cat"` to the array. Then, we use `push` again to add on `"Dog"` and then `"Llama"`. When we display `animals;`, we see that `"Cat"`, `"Dog"`, and `"Llama"` were added to the array, in the same order we entered them.



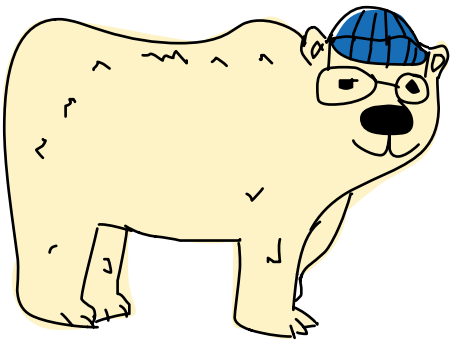
The act of running a method in computer-speak is known as *calling* the method. When you call the `push` method, two things happen. First, the element in parentheses is added to the array. Second, the new length of the array is returned. That's why you see those numbers printed out every time you call `push`.

To add an element to the beginning of an array, you can use `.unshift(element)`, like this:

```
animals;
["Cat", "Dog", "Llama"]
❶ animals[0];
"Cat"
animals.unshift("Monkey");
4
animals;
["Monkey", "Cat", "Dog", "Llama"]
animals.unshift("Polar Bear");
5
animals;
["Polar Bear", "Monkey", "Cat", "Dog", "Llama"]
animals[0];
"Polar Bear"
❷ animals[2];
"Cat"
```

Here we started with the array that we've been using, `["Cat", "Dog", "Llama"]`. Then, as we add the elements `"Monkey"` and `"Polar Bear"` to the beginning of the array with `unshift`, the old values get pushed along by one index each time. So `"Cat"`, which was originally at index 0 ❶, is now at index 2 ❷.

Again, `unshift` returns the new length of the array each time it is called, just like `push`.



## REMOVING ELEMENTS FROM AN ARRAY

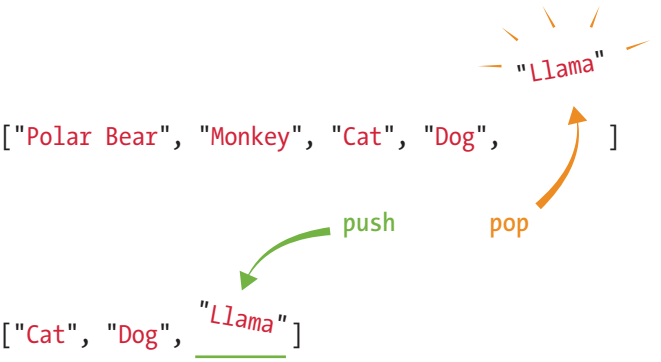
To remove the last element from an array, you can pop it off by adding `.pop()` to the end of the array name. The `pop` method can be particularly handy because it does two things: it removes the last element, *and* it returns that last element as a value. For example, let's start with our `animals` array, `["Polar Bear", "Monkey", "Cat", "Dog", "Llama"]`. Then we'll create a new variable called `lastAnimal` and save the last animal into it by calling `animals.pop()`.

```
animals;
["Polar Bear", "Monkey", "Cat", "Dog", "Llama"]
❶ var lastAnimal = animals.pop();
lastAnimal;
"Llama"
animals;
["Polar Bear", "Monkey", "Cat", "Dog"]
❷ animals.pop();
"Dog"
animals;
["Polar Bear", "Monkey", "Cat"]
❸ animals.unshift(lastAnimal);
4
animals;
["Llama", "Polar Bear", "Monkey", "Cat"]
```

When we call `animals.pop()` at ❶, the last item in the `animals` array, `"Llama"`, is returned and saved in the variable `lastAnimal`. `"Llama"` is also removed from the array, which leaves us with four animals. When we call `animals.pop()` again at ❷, `"Dog"` is removed from the array and returned, leaving only three animals in the array.

When we used `animal.pop()` on `"Dog"`, we didn't save it into a variable, so that value isn't saved anywhere anymore. The `"Llama"`, on the other hand, was saved to the variable `lastAnimal`, so we can use it again whenever we need it. At ❸, we use `unshift(lastAnimal)` to add `"Llama"` back onto the front of the array. This gives us a final array of `["Llama", "Polar Bear", "Monkey", "Cat"]`.

Pushing and popping are a useful pair because sometimes you care about only the end of an array. You can push a new item onto the array and then pop it off when you're ready to use it. We'll look at some ways to use pushing and popping later in this chapter.



To remove and return the first element of an array, use `.shift()`:

```
animals;
["Llama", "Polar Bear", "Monkey", "Cat"]
var firstAnimal = animals.shift();
firstAnimal;
"Llama"
animals;
["Polar Bear", "Monkey", "Cat"]
```

`animals.shift()` does the same thing as `animals.pop()`, but the element comes off the beginning instead. At the start of this example, `animals` is `["Llama", "Polar Bear", "Monkey", "Cat"]`. When we call `.shift()` on the array, the first element, `"Llama"`, is returned and saved in `firstAnimal`. Because `.shift()` removes the first element as well as returning it, at the end `animals` is just `["Polar Bear", "Monkey", "Cat"]`.

You can use `unshift` and `shift` to add and remove items from the beginning of an array just as you'd use `push` and `pop` to add and remove items from the end of an array.