3

COMPOUND DATA TYPES

In the previous chapter we discussed JavaScript's primitive data types, which represent a single piece of data, like a number or a string. Now we'll look at JavaScript's *compound data types*, arrays and objects, which combine multiple pieces of data into a single unit. Compound data types are an essential part of programming because they allow us to organize and work with collections of data of any size. You'll learn how to create and manipulate arrays and objects, and how to combine them into more complex data structures.

Arrays

A JavaScript *array* is a compound data type that holds an ordered list of values. The elements of an array can be of any data type. They don't all have to be the same type, although they typically are. For instance, an array might function as a to-do list by holding a series of strings describing tasks that need to be performed, or it might hold a collection of numbers representing temperature readings taken at regular intervals from a particular location.

Arrays are perfect for these sorts of structures because they collect the related values together in one place, and they have the flexibility to grow and shrink as values are added or removed. If you had a fixed number of to-do items—say, four—then you might use separate variables to hold them, but using an array enables you to hold an unbounded, changing number of items and keep them in a fixed order. Also, once you have your elements together in a single array, you can write code to efficiently operate on each item in the array in turn, as you'll see in Chapter 4.

Creation and Indexing

To create an array, list its elements separated by commas inside a pair of square brackets:

```
let primes = [2, 3, 5, 7, 11, 13, 17, 19];
primes;
▸ (8) [2, 3, 5, 7, 11, 13, 17, 19]
```

This array contains the first eight prime numbers and is stored in the `primes` variable. When you enter **`primes;`** the Chrome console should print the length of the array (`8`) followed by its elements.

Every element in an array has an index number associated with it. Like strings, arrays are zero-indexed, so the first element is found at index `0`, the second at index `1`, and so on. To access an individual element of an array, place its index number in square brackets after the name of the array. Here, for example, we access the first element of the `primes` array:

```
primes[0];
2
```

Because arrays are zero-indexed, the index of the last element of the array is one less than the array's length. So, the last element of our eight-element `primes` array is at index `7`:

```
primes[7];
19
```

If you don't know how long an array is and you want to get its last element, you can first use dot notation to access its `length` property and look up the array's length, as we did with strings in Chapter 2:

```
primes.length;
8
primes[7];
19
```

Or, to do this in a single statement, you can simply subtract 1 from the length to get the element at the last index, like so:

```
primes[primes.length - 1];
19
```

If you use an index outside the range of the array, JavaScript returns `undefined`:

```
primes[10];
undefined
```

To replace an element in an array, assign the element a new value using indexing syntax:

```
primes[2] = 1;
primes;
▸ (8) [2, 3, 1, 7, 11, 13, 17, 19]
```

Here we add a `1` in the third position (index `2`) of `primes`, replacing the value that was previously at that index. The console output confirms that `1` is the new third element in the array.

Arrays of Arrays

Arrays can contain other arrays. These *multidimensional arrays* are often used to represent two-dimensional grids of points, or tables. To illustrate this, let's make a simple tic-tac-toe game. We'll create an array (we'll call this the *outer* array) containing three elements, each of which is another array (we'll call these the *inner* arrays) representing one of the rows of the tic-tac-toe board. Each inner array will contain three empty strings to represent the squares within that row:

```
let ticTacToe = [
  ["", "", ""],
  ["", "", ""],
  ["", "", ""]
];
```

In order to make the code easier to read, I've put each inner array on a new line. Usually when you press ENTER (commonly to start a new line), the JavaScript console will run the line of code you just entered, but in this case, it's clever enough to realize that the first line isn't finished, because there's no closing square bracket to match the opening bracket. It will

interpret everything up to the final closing bracket and semicolon as a single statement, even if you include additional brackets and carriage returns.

NOTE *The Chrome console automatically applies indentation to the inner arrays, to indicate that they're nested inside the outer array. Chrome and VS Code by default use four spaces for each level of indentation, but this is a matter of personal preference. Throughout this book I'll be using two spaces for indentation, both because this is more common in modern JavaScript code and because it helps some of the bigger listings fit on the page.*

I could have written this array on one line, as shown here, but this way it's harder to see its two-dimensionality:

```
let ticTacToeOneLine = [["", "", ""], ["", "", ""], ["", "", ""]];
```

Now let's see what happens when we ask the console for the value of the `ticTacToe` variable:

```
ticTacToe;
▸ (3) [Array(3), Array(3), Array(3)]
```

In this case, the length of the outer array is shown as `(3)`, indicating that it's an array with three elements. Each element of the array is `Array(3)`, which means each inner array is another three-element array.

To expand the view and see what's in those inner arrays, click the arrow on the left:

```
▾ (3) [Array(3), Array(3), Array(3)]
  ▸0: (3) ['', '', '']
  ▸1: (3) ['', '', '']
  ▸2: (3) ['', '', '']
   length: 3
  ▸[[Prototype]]: Array(0)
```

The first three lines show the values of the inner arrays at indexes `0`, `1`, and `2`. After these, the outer array's `length` property is shown, with its value of `3`. The final property, `[[Prototype]]`, is where the array's built-in methods come from (more on this in Chapter 6).

We've created our tic-tac-toe board, but it's empty. Let's set an `X` in the top-right corner. The first inner array represents the top row; we access it with `ticTacToe[0]`. The top-right corner is the third element of that row, or index `2` of the inner array. Because `ticTacToe[0]` returns an array, we can just add `[2]` on the end to access the element we want: `ticTacToe[0][2]`. Knowing this, we can set this element to `"X"` as follows:

```
ticTacToe[0][2] = "X";
```

Now let's look at the value of `ticTacToe` again, clicking the arrow to expand the outer array:

```
ticTacToe;
▾ (3) [Array(3), Array(3), Array(3)]
  ▸0: (3) ['', '', 'X']
  ▸1: (3) ['', '', '']
  ▸2: (3) ['', '', '']
   length: 3
  ▸[[Prototype]]: Array(0)
```

The top-right corner of the tic-tac-toe board now contains an `X`.

Next, let's set an `0` in the bottom-left corner. The bottom row is index `2` of the outer array, and the leftmost square of that row is index `0` of the inner array, so we enter the following:

```
ticTacToe[2][0] = "O";
ticTacToe;
▾ (3) [Array(3), Array(3), Array(3)]
  ▸0: (3) ['', '', 'X']
  ▸1: (3) ['', '', '']
  ▸2: (3) ['O', '', '']
   length: 3
  ▸[[Prototype]]: Array(0)
```

Now there's an `0` in the bottom-left corner of the board.

To summarize, if you want to access an element in a nested array, use one set of square brackets to select the element in the outer array (which returns one of the inner arrays), then a second set to select the element in the inner array.

TRY IT YOURSELF

3-1. Play a game of tic-tac-toe against yourself, using the `ticTacToe` array. Remember, the first index number should be the row of the board, and the second index number should be the column.

Array Methods

JavaScript has several useful methods for working with arrays. We'll look at a few important ones in this section. Some of these methods modify the array in question, which is known as *mutation*. Example mutations include adding or deleting array elements, or changing the elements' order. Other methods create and return a new array while leaving the original array unchanged, which is useful if you still need the original array for other purposes.

It's important to be aware of whether or not the method you're using will mutate the array. For example, say you have an array containing the months of the year listed chronologically, but one part of your program needs them in alphabetical order. You'd want to be sure that alphabetizing

the months doesn't inadvertently change the original, chronological array, or other parts of your program might start thinking April is the first month of the year. On the other hand, if you have an array representing a to-do list, you'd probably want the original array itself to be updated when a task is added or removed, rather than creating a new array.

Adding an Element to an Array

The `push` method mutates an array by adding a supplied element to the end of the array. The return value of the `push` method is the new length of the array. As an example, let's use `push` to build up an array of programming languages:

```
let languages = [];
languages.push("Python");
1
languages.push("Haskell");
2
languages.push("JavaScript");
3
languages.push("Rust");
4
languages;
▸ (4) ['Python', 'Haskell', 'JavaScript', 'Rust']
```

First we create a new array called `languages` and initialize it with `[]`, an empty array. The first time we call the `push` method, we pass the value `"Python"`. The method returns `1`, which means there's now one element in the array. We do this three more times, and finally ask for the value of `languages` by entering `languages;`. This returns the four languages we added to the array, in the order we added them.

To add an element to the beginning of the array rather than the end, use the `unshift` method, like so:

```
languages.unshift("Erlang");
5
languages.unshift("C");
6
languages.unshift("Fortran");
7
languages;
▸ (7) ['Fortran', 'C', 'Erlang', 'Python', 'Haskell', 'JavaScript', 'Rust']
```

Here we've added three more languages to the front of the `languages` array. Because each element is added to the beginning of the array, they end up in the opposite order to how they were added. Like `push`, calling `unshift` returns the new length of the array.

Removing Elements from an Array

To mutate an array by removing its last element, use the `pop` method. Here we call the `pop` method on the `languages` array, deleting its last element:

```
languages.pop();
'Rust'
languages;
▸ (6) ['Fortran', 'C', 'Erlang', 'Python', 'Haskell', 'JavaScript']
```

The method returns the value of the element being removed, in this case `"Rust"`. When we then check the array, it contains only six elements.

Because the `pop` method returns the array element being removed, it's particularly useful if you want to do something with that element as you're removing it. For example, here we delete another element from the `languages` array and use it in a message:

```
let bestLanguage = languages.pop();
let message = `My favorite language is ${bestLanguage}.`;
message;
'My favorite language is JavaScript.'
languages;
▸ (5) ['Fortran', 'C', 'Erlang', 'Python', 'Haskell']
```

This time when we call `languages.pop()` we store the method's return value in the `bestLanguage` variable, which we incorporate into a string using a template literal. When we print the resulting message, it includes the word *JavaScript*. This was the element removed from the array, which is now down to five languages.

To remove the *first* element from an array, rather than the last, use the `shift` method. Like `pop`, the `shift` method returns the removed element:

```
let worstLanguage = languages.shift();
message = `My least favorite language is ${worstLanguage}.`;
message;
'My least favorite language is Fortran.'
languages;
▸ (4) ['C', 'Erlang', 'Python', 'Haskell']
```

As with the previous example, we save the result of calling `shift` in a variable, this time called `worstLanguage`, and use it in a template literal. This variable contains the string `"Fortran"`, and `languages` is left with four elements.

The four methods we've looked at so far, `pop`, `unshift`, `push`, and `shift`, are commonly used to implement more specialized data structures, like queues. A *queue* is a data structure that resembles a line of people, where new items are added to the end and items are removed and processed from the beginning. This is useful when you want to process data in the order it arrives. For example, imagine a Q and A app, where many users can ask questions. You could use an array to store the list of questions, with the `push` method

adding each new question to the end of the array. When the answerer is ready to answer a question, they would use `shift` to get the first element in the array and remove it from the array. This ensures that only unanswered questions are in the array, and that they're answered in the order they were received.

TRY IT YOURSELF

3-2. Create a new empty array and save it in a variable called `rainbow` (see the section "Adding an Element to an Array" on page 42 to see how to create a new empty array). Your task is to add the colors of the rainbow (`"Red"`, `"Orange"`, `"Yellow"`, `"Green"`, `"Blue"`, `"Indigo"`, `"Violet"`) to this array, but with a twist: you must start by adding `"Green"`, and use `push` and `unshift` to add the rest. If you make a mistake, you can use `pop` or `shift` to remove the color you just added.

Combining Arrays

The `concat` method (short for *concatenate*) adds two arrays together. Here, for example, we start with two arrays, `fish` and `mammals`, and combine them into a new array, saving that into the `animals` variable:

```
let fish = ["Salmon", "Cod", "Trout"];
let mammals = ["Sheep", "Cat", "Tiger"];
let animals = fish.concat(mammals);
animals;
▸ (6) ['Salmon', 'Cod', 'Trout', 'Sheep', 'Cat', 'Tiger']
```

When you call `concat` on an array, a new array is created with all the elements from the first array (the array on which you called `concat`) followed by all the elements from the second array (the array passed as an argument to `concat`). The original arrays remain unchanged because, unlike the other methods we've looked at so far, `concat` isn't a mutating method. This is useful here, because we wouldn't want our `fish` array to suddenly contain the elements from `mammals`!

To combine three or more arrays, pass multiple arrays as arguments to `concat`, as in this example:

```
let originals = ["Hope", "Empire", "Jedi"];
let prequels = ["Phantom", "Clones", "Sith"];
let sequels = ["Awakens", "Last", "Rise"];
let starWars = prequels.concat(originals, sequels);
starWars;
▸ (9) ['Phantom', 'Clones', 'Sith', 'Hope', 'Empire', 'Jedi', 'Awakens', 'Last', 'Rise']
```

Here we create three separate arrays, `originals`, `prequels`, and `sequels`, representing the three sets of *Star Wars* movies. Then we use `concat` to combine them into a single nine-element `starWars` array. Notice that the elements in the combined array appear in the order in which the arrays were passed as arguments.

Finding the Index of an Element in an Array

To find out where a particular element is in an array, use the `indexOf` method. This method returns the index of the first occurrence of the specified element. If the element isn't found in the array, `indexOf` returns `-1`:

```
let sizes = ["Small", "Medium", "Large"];
sizes.indexOf("Medium");
1
sizes.indexOf("Huge");
-1
```

In this example, we want to check the position of `"Medium"` in the `sizes` array, and we get back the answer `1`. Then, because `"Huge"` isn't in the array, we get the answer `-1`.

If the array contains multiple instances of the given value, `indexOf` returns the index of the first matching element only. For example, here's an array with the colors of the flag of Argentina:

```
let flagOfArgentina = ["Blue", "White", "Blue"];
flagOfArgentina.indexOf("Blue");
0
```

Even though `"Blue"` is found twice in the array, `indexOf` returns only the index of the first occurrence.

Turning an Array into a String

The `join` method converts an array into a single string, joining all the elements together, as shown here:

```
let beatles = ["John", "Paul", "George", "Ringo"];
beatles.join();
'John,Paul,George,Ringo'
```

Notice how the separate strings in the `beatles` array are combined into one string. By default, `join` places a comma between each element to form the returned string. To change this, you can give your own separator as an argument to `join`. For example, if you want nothing in between each element, pass an empty string as an argument:

```
beatles.join("");
'JohnPaulGeorgeRingo'
```

You can pass any valid string as a separator. In the next example, we pass a space, an ampersand, and a newline escape character to set each element on its own line. As you learned in Chapter 2, we have to use `console.log` for the newlines to display correctly in Chrome:

```
console.log(beatles.join("&\n"));
John&
Paul&
George&
Ringo
```

Keep in mind that the separator appears only *between* array elements, not after each one. This is why there isn't an extra ampersand and newline after `Ringo`.

If you use `join` on an array containing non-string values, those values will be converted to strings, as in this example:

```
[100, true, false, "hi"].join(" - ");
'100 - true - false - hi'
```

As with the previous joins, the result is one long string, joined together by the separator (in this case, `" - "`). The difference is that the non-string values (the number `100` and the Booleans `true` and `false`) had to be automatically converted to strings before the join. This example also shows how you can call array methods directly on array literals, rather than having to save the array into a variable first.

TRY IT YOURSELF

3-3. Use the `join` method to convert the array `["X", "X", "X"]` into the string `"XoXoX"`.

Other Useful Array Methods

Here are some other useful array methods you might want to try out:

***arr*.includes(*elem*)** Returns `true` or `false` depending on whether a given *elem* is in the *arr* array.

***arr*.reverse()** Reverses the order of elements in the array. This is a mutating method, so it modifies the original array.

***arr*.sort()** Sorts the array elements, modifying the original array. If the elements are strings, they're sorted in alphabetical order. Otherwise, the sorting happens as if the elements were converted to strings.

***arr*.slice(*start*, *end*)** Creates a new array by extracting elements from the original array starting at index *start*, up to but not including index *end*.

This method is equivalent to the `slice` method on strings, introduced in the previous chapter. If you call `slice()` without any arguments, the entire array is copied into a new array. This is useful if you need to use a mutating method like `sort` but you don't want to mutate the original array.

***arr*.splice(*index*, *count*)** Removes *count* elements from the array, starting at *index*.

Objects

Objects are another compound data type in JavaScript. They're similar to arrays in that they hold a collection of values, but they differ in that objects use strings called *keys* instead of numeric indices to access the values. Each key is associated with a specific value, forming a *key-value pair*.

Whereas arrays are commonly used to store ordered lists of elements of the same data type, objects are usually used to store multiple pieces of information about a single entity. These pieces of information often are not all of the same data type. For example, an object representing a person might hold information like the person's name (a string), their age (a number), whether or not they're married (a Boolean), and so on. Objects are better suited for this purpose than arrays because each piece of information is given a meaningful name—its key—rather than a generic index number. It's much clearer what the values `35` and `true` mean if they're stored in a `person` object under the keys `"age"` and `"married"` than it would be if they were stored in a `person` array under the indices `1` and `2`.

Creating Objects

One way to create an object is with an *object literal*, which consists of a pair of braces ({ and }) enclosing a series of key-value pairs, separated by commas. Each key-value pair must have a colon between the key and the value. For example, here's an object literal called `casablanca` containing some information about that movie:

```
let casablanca = {
  "title": "Casablanca",
  "released": 1942,
  "director": "Michael Curtiz"
};
casablanca;
▸ {title: 'Casablanca', released: 1942, director: 'Michael Curtiz'}
```

Here we create a new object with three keys: `"title"`, `"released"`, and `"director"`. Each key has a value associated with it. I've written each key-value pair on its own line to make the object literal easier to read, but this isn't strictly necessary. As you'll see in later examples, the key-value pairs can also all be written on the same line.

All object keys are strings, but if your key is a valid identifier, it's common practice to omit the quotes. A *valid identifier* is any series of characters that can be used as a JavaScript variable name. An identifier can consist of letters, numbers, and the characters _ and $, but it can't start with a number. It also can't contain other symbols, like *, (, or #, nor can it include whitespace characters like spaces and newlines. These other characters *are* allowed in object keys, but only if the key is enclosed in quotes. For example:

```
let obj = { key1: 1, key_2: 2, "key 3": 3, "key#4": 4 };
obj;
▸ {key1: 1, key_2: 2, key 3: 3, key#4: 4}
```

Here `key1` and `key_2` are valid identifiers, so they don't need quotes. However, `key 3` contains a space and `key#4` contains a hash mark, making them invalid identifiers. They must be enclosed in quotes to be used as object keys.

Accessing Object Values

To get the value associated with a key, call the name of the object with the string key in square brackets:

```
obj["key 3"];
3
casablanca["title"];
'Casablanca'
```

This is just like the syntax for accessing an element from an array, but instead of using the numeric index, you use the string key.

For keys that are valid identifiers, you can use dot notation instead of square brackets, with the key name coming after the dot:

```
obj.key_2;
2
```

This doesn't work for keys that aren't valid identifiers. For example, you can't write `obj.key 3` because to JavaScript that looks like `obj.key` followed after the space by the number literal `3`.

Notice that this dot notation looks like the syntax we used for accessing the `length` property of strings (in Chapter 2) and arrays (earlier in this chapter). That's because it's the same thing! A property is just another name for a key-value pair. Behind the scenes, JavaScript treats strings like objects, and arrays, too, are actually a special kind of object. When we write something like `[1, 2, 3].length`, we say we're accessing the array's `length` property, but we could also say we're getting the value associated with the array's `length` key. Likewise, when we write something like `casablanca.title`, we often say we're accessing the object's `title` property instead of the value associated with its `title` key.

Setting Object Values

To add a new key-value pair to an object, use the same bracket or dot notation used to look up a value. Here, for example, we set up an empty `dictionary` object, then add two definitions:

```
let dictionary = {};
dictionary.mouse = "A small rodent";
dictionary["computer mouse"] = "A pointing device for computers";
dictionary;
▸ {mouse: 'A small rodent', computer mouse: 'A pointing device for computers'}
```

We first create a new, empty object using a pair of empty braces. We then set two new keys, `"mouse"` and `"computer mouse"`, giving each a definition as a value. As before, we can use dot notation with the valid identifier `mouse`, but we need bracket notation for `"computer mouse"` because it contains a space.

Changing the value associated with a key that already exists follows the same syntax:

```
dictionary.mouse = "A furry rodent";
dictionary;
▸ {mouse: 'A furry rodent', computer mouse: 'A pointing device for computers'}
```

The output confirms that the definition for `mouse` has been updated.

Working with Objects

JavaScript has plenty of methods for working with objects; we'll examine a few of the most common ones here. Unlike with arrays, where the methods are called directly on the array you want to operate on, object methods are called as static methods by entering `Object.`*`methodName`*`()` and passing the object you want to operate on as an argument inside the parentheses. Here, `Object` is a *constructor*, a type of function used to create objects, and *static methods* are methods defined directly on the constructor instead of on a particular object. We'll discuss constructors in more detail in Chapter 6.

Getting an Object's Keys

To get an array of all the keys of an object, use the static method `Object.keys`. For example, here's how you could retrieve the names of my cats:

```
let cats = { "Kiki": "black and white", "Mei": "tabby", "Moona": "gray" };
Object.keys(cats);
▸ (3) ['Kiki', 'Mei', 'Moona']
```

The `cats` object has three key-value pairs, where each key represents a cat name and each value represents that cat's color. `Object.keys` returns just the keys, as an array of strings.

`Object.keys` can be helpful in cases like this where the only pieces of information you need from an object are the names of its keys. For example, you might have an object tracking how much money you owe your friends, where the keys are your friends' names and the values are the amounts owed. With `Object.keys` you can list just the names of the friends that you're tracking, giving you a general sense of whom you owe money to.

You might be wondering why `keys` is a static method—that is, why we need to call it with `Object.keys(cats)` rather than with `cats.keys()`. To understand why this is the case, consider this `piano` object:

```
let piano = {
  make: "Steinway",
  color: "black",
  keys: 88
};
```

The object has a property named `"keys"` that represents the number of keys on the piano. If methods like `keys` could be called directly on the `piano` object itself, the property name and method name would conflict, which isn't allowed. JavaScript has many more built-in object methods besides `keys`, and it would be tedious to have to remember all of their names to make sure they don't conflict with any of your objects' property names. To avoid this issue, the designers of the language made these object methods static. They're attached to the overall `Object` constructor instead of to individual objects like `cat` or `piano`, so there's no possibility of a naming conflict.

NOTE *None of this is an issue with arrays. Method names must be valid identifiers, meaning they can't start with a number. Therefore, there's no way an array method could conflict with the array's numerical indices.*

Getting an Object's Keys and Values

To get an array of the keys *and* values of an object, use `Object.entries`. This static method returns an array of two-element arrays, where the first element of each inner array is a key and the second is its value. Here's how it works:

```
let chromosomes = {
  koala: 16,
  snail: 24,
  giraffe: 30,
  cat: 38
};
Object.entries(chromosomes);
▸ (4) [Array(2), Array(2), Array(2), Array(2)]
```

We create an object with four key-value pairs, showing how many chromosomes various animals have. `Object.entries(chromosomes)` returns an array containing four elements, each of which is a two-element array. To expand the outer array and view its full contents, click the arrow:

```
▾ (4) [Array(2), Array(2), Array(2), Array(2)]
  ▸0: (2) ['koala', 16]
  ▸1: (2) ['snail', 24]
  ▸2: (2) ['giraffe', 30]
  ▸3: (2) ['cat', 38]
   length: 4
  ▸[[Prototype]]: Array(0)
```

This shows that each inner array contains a key from the original object as its first element, and the associated value as its second element.

Converting an object into an array with `Object.entries` makes it easier to cycle through all of the object's key-value pairs and do something with each one in turn. We'll see how to do this with loops in Chapter 4.

Combining Objects

The `Object.assign` method lets you combine multiple objects into one. For example, say you have two objects, one giving the physical attributes of a book and the other describing its contents:

```
let physical = { pages: 208, binding: "Hardcover" };
let contents = { genre: "Fiction", subgenre: "Mystery" };
```

With `Object.assign`, you can consolidate these separate objects into one overall `book` object:

```
let book = {};
Object.assign(book, physical, contents);
book;
▸ {pages: 208, binding: 'Hardcover', genre: 'Fiction', subgenre: 'Mystery'}
```

The first argument to `Object.assign` is the *target*, the object that the keys from the other objects are assigned to. In this case, we use an empty object called `book` as the target. The remaining arguments are the *sources*, the objects whose key-value pairs are to be copied into the target. You can pass as many source objects after the initial target argument as you want—we're just doing two here. The method mutates and returns the target object with the key-value pairs copied from the source objects. The source objects themselves are untouched.

You don't have to create a new, empty object to use as the target for `Object.assign`, but if you don't, you'll end up modifying one of your source objects. For example, we could remove the first argument, `book`, from the previous call and still get an object with the same four key-value pairs:

```
Object.assign(physical, contents);
physical;
▸ {pages: 208, binding: 'Hardcover', genre: 'Fiction', subgenre: 'Mystery'}
```

The problem here is that `physical` is now the target object, so it gets mutated, gaining all the key-value pairs from `contents`. This usually isn't

what you want, as the original, separate objects are often still important to other parts of your application. For this reason, it's common practice to use an empty object as the first argument to `Object.assign`.

Nesting Objects and Arrays

As with arrays, we can nest objects in other objects. We can also nest objects in arrays, and arrays in objects, to create more sophisticated data structures. For example, you might want to make an object representing a person that contained a `children` property containing an array of objects representing that person's children. We build these nested structures in two ways: by creating an object or array literal with nested object or array literals inside, or by creating the inner elements, saving them to variables, and then building up the composite structures using the variables. We'll examine both of these techniques here.

Nesting with Literals

First, let's build a nested structure using literals. We'll create an array of objects representing different book trilogies:

```
let trilogies = [
❶ {
    title: "His Dark Materials",
    author: "Philip Pullman",
    books: ["Northern Lights", "The Subtle Knife", "The Amber Spyglass"]
  },
❷ {
    title: "Broken Earth",
    author: "N. K. Jemisin",
    books: ["The Fifth Season", "The Obelisk Gate", "The Stone Sky"]
  }
];
```

The variable `trilogies` contains an array of two elements, ❶ and ❷, each of which is an object with information about a particular trilogy. Notice that each object has the same keys, since we want to store the same pieces of information about each trilogy. One of those keys, `books`, itself contains an array of strings representing the book titles within the trilogy. We thus have an array within an object within an array.

Accessing an element from one of these inner arrays requires a combination of array indexing and object dot notation:

```
trilogies[1].books[0];
'The Fifth Season'
```

Here, `trilogies[1]` means we want the second object in the outer array, `.books` means we want the value of that object's `books` key (which is an

array), and `[0]` means we want the first element from that array. Putting it together, we get the first book from the second trilogy in the outer array.

Nesting with Variables

An alternative technique for making nested structures is to create objects containing the inner elements, assign those objects to variables, and then build the outer structure out of these variables. For example, say we want to create a data structure modeling the change in our pocket. We create four objects representing a penny, nickel, dime, and quarter, assigning each to its own variable:

```
let penny = { name: "Penny", value: 1, weight: 2.5 };
let nickel = { name: "Nickel", value: 5, weight: 5 };
let dime = { name: "Dime", value: 10, weight: 2.268 };
let quarter = { name: "Quarter", value: 25, weight: 5.67 };
```

Next, we use these variables to create an array representing the specific combination of coins in our pocket. For example:

```
let change = [quarter, quarter, dime, penny, penny, penny];
```

Notice that some of the coin objects appear in the array multiple times. This is one advantage of assigning the inner objects to variables before we create the outer array: an object can be repeated within the array without having to manually write out the object literal each time.

Accessing a value from one of the inner objects again requires a combination of array indexing and object dot notation:

```
change[0].value;
25
```

Here, `change[0]` gives us the first element of the `change` array (a `quarter` object) and `.value` gives us its `value` key.

An interesting consequence of building the array from object variables like this is that the repeated elements share a common identity. For example, `change[3]` and `change[4]` refer to the same `penny` object. If the US government decided to update the weight of a penny, we could update the `weight` property of the underlying `penny` object, and that update would be reflected in all the `penny` elements of the `change` array:

```
penny.weight = 2.49;
change[3].weight;
2.49
change[4].weight;
2.49
change[5].weight;
2.49
```

Here we change the `weight` property of `penny` from `2.5` to `2.49`. Then we check the weight of each `penny` in the array, confirming that the update has carried over to each one.

TRY IT YOURSELF

3-4. Try changing the `value` property of `quarter` and check to see if that change is reflected in the `change` array. Now, change the weight of `change[0]`. Do you see that change reflected in `quarter` as well?

Exploring Nested Objects in the Console

The Chrome console makes it easy to explore nested objects, like we did earlier in this chapter with the nested `ticTacToe` array. To illustrate, we'll create a deeply nested object and try to look inside:

```
let nested = {
  name: "Outer",
  content: {
    name: "Middle",
    content: {
      name: "Inner",
      content: "Whoa..."
    }
  }
};
```

Our `nested` object contains three layers of objects, each with a `name` and `content` property. The value of `content` for the outer and middle layers is another object. Getting the value of the innermost object's `content` property requires a long chain of dot notation:

```
nested.content.content.content;
'Whoa...'
```

This is equivalent to asking for the `content` property of the `content` property of the `content` property of the outermost object.

Now try viewing the value of `nested` as a whole:

```
nested;
▸ {name: 'Outer', content: {...}}
```

The console just gives an abbreviated version with the value of the outer object's `content` property shown as `{...}` to imply that there's an object here but there isn't room to display it. Click the arrow to expand the view of the outer object. Now the next nested object (with `name: "Middle"`) is shown in

abbreviated form. Click the arrow to expand this object, too, and then one more time to expand the object with `name: "Inner"`. You should now see the entire content of the object in the console:

```
▾ {name: 'Outer', content: {...}}
  ▾content:
    ▾content:
        content: "Whoa..."
        name: "Inner"
      ▸[[Prototype]]: Object
      name: "Middle"
    ▸[[Prototype]]: Object
    name: "Outer"
  ▸[[Prototype]]: Object
```

The `[[Prototype]]` properties refer to the `Object` constructor, which we've previously used to call object methods like `Object.keys` and `Object.assign`. We'll discuss prototypes in detail in Chapter 6.

Using the console like this to view complex objects is a very helpful debugging tool. You'll often be working with objects that come from different JavaScript libraries, or that contain data you fetch from a server, and you won't necessarily know the "shape" of the data—what properties the objects contain, how many levels of nesting they have, and the like. With the console, you can interactively explore the objects and see their contents.

Printing Nested Objects with JSON.stringify

Another way to view a nested object is to turn it into a JSON string. *JSON*, or *JavaScript Object Notation*, is a textual data format based on JavaScript object and array literals that's heavily used across the web and beyond to store and exchange information. The `JSON.stringify` method converts a JavaScript object into a JSON string. Let's pass it the `nested` object as an example:

```
JSON.stringify(nested);
'{"name":"Outer","content":{"name":"Middle","content":{"name":"Inner","content":"Whoa..."}}}'
```

The result is a string (it's enclosed in single quotes) containing a JSON representation of the `nested` object. Essentially, it's the equivalent of the original object literal we used to create `nested`. Just like JavaScript, JSON uses braces to enclose objects, colons to separate keys from values, and commas to separate different key-value pairs. All that's missing from this representation are the original line breaks and indentations we used to clarify the object literal's nested structure. To re-create those, we can pass `JSON.stringify` another argument representing the number of spaces to indent each new nested object:

```
nestedJSON = JSON.stringify(nested, null, 2);
console.log(nestedJSON);
```

```
{
  "name": "Outer",
  "content": {
    "name": "Middle",
    "content": {
      "name": "Inner",
      "content": "Whoa..."
    }
  }
}
```

The second argument to `JSON.stringify` lets you define a replacer function that can modify the output by replacing key-value pairs, but we don't have a need for that here, so we pass `null`. Passing `2` for the third argument modifies the behavior of `JSON.stringify` to add newlines after each property and after opening braces and brackets, and then two extra spaces of indentation for each additional level of nesting. If we viewed the result in the console directly, we'd see a bunch of `\n` escape characters for all the newlines. Instead, we store the result in a variable and pass it to `console.log`, giving us a well-formatted view of the object's nested hierarchy.

Calling `JSON.stringify` in this way is helpful for getting a quick visual representation of an object without having to repeatedly click the arrows in the console to expand each nested level. The method works on non-nested objects, too, but in that case the regular view of the object in the console is usually sufficient.

Summary

This chapter introduced you to JavaScript's compound data types, which allow you to combine multiple values into a single unit. By organizing data in this way, you can manipulate unbounded amounts of information more efficiently. You learned about arrays, which are ordered collections of values identified by numerical indices, usually all of the same data type, and about objects, which are collections of key-value pairs where each key is a string and the values are often of different data types. You've seen how arrays are useful for storing lists of similar values, such as a list of prime numbers or a list of programming languages. Meanwhile, objects are useful for collecting multiple pieces of information about a single entity, such as information about a particular book or movie.