

COMBINING ARRAYS AND OBJECTS

So far, we've looked only at arrays and objects that contain simple types like numbers and strings. But there's nothing stopping you from using another array or object as a value in an array or object.

For example, an array of dinosaur objects might look like this:

```
var dinosaurs = [  
  { name: "Tyrannosaurus Rex", period: "Late Cretaceous" },  
  { name: "Stegosaurus", period: "Late Jurassic" },  
  { name: "Plateosaurus", period: "Triassic" }  
];
```

To get all the information about the first dinosaur, you can use the same technique we used before, entering the index in square brackets:

```
dinosaurs[0];  
{ name: "Tyrannosaurus Rex", period: "Late Cretaceous" }
```

If you want to get only the name of the first dinosaur, you can just add the object key in square brackets after the array index:

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

Or, you can use dot notation, like this:

```
dinosaurs[1].period;  
"Late Jurassic"
```

NOTE

You can use dot notation only with objects, not with arrays.

AN ARRAY OF FRIENDS

Let's look at a more complex example now. We'll create an array of friend objects, where each object also contains an array. First, we'll make the objects, and then we can put them all into an array.

```
var anna = { name: "Anna", age: 11, luckyNumbers: [2, 4, 8, 16] };  
var dave = { name: "Dave", age: 5, luckyNumbers: [3, 9, 40] };  
var kate = { name: "Kate", age: 9, luckyNumbers: [1, 2, 3] };
```

First, we make three objects and save them into variables called `anna`, `dave`, and `kate`. Each object has three keys: `name`, `age`, and `luckyNumbers`. Each `name` key has a string value assigned to it, each `age` key has a single number value assigned to it, and each `luckyNumbers` key has an array assigned to it, containing a few different numbers.

Next we'll make an array of our friends:

```
var friends = [anna, dave, kate];
```

Now we have an array saved to the variable `friends` with three elements: `anna`, `dave`, and `kate` (which each refer to objects). You can retrieve one of these objects using its index in the array:

```
friends[1];  
{ name: "Dave", age: 5, luckyNumbers: Array[3] }
```

This retrieves the second object in the array, `dave` (at index 1). Chrome prints out `Array[3]` for the `luckyNumbers` array, which is just its way of saying, “This is a three-element array.” (You can use Chrome to see what’s in that array; see “Exploring Objects in the Console” on page 71.) We can also retrieve a value within an object by entering the index of the object in square brackets followed by the key we want:

```
friends[2].name  
"Kate"
```

This code asks for the element at index 2, which is the variable named `kate`, and then asks for the property in that object under the key `"name"`, which is `"Kate"`. We could even retrieve a value from an array that’s inside one of the objects inside the `friends` array, like so:

```
friends[0].luckyNumbers[1];  
4
```

Figure 4-2 shows each index. `friends[0]` is the element at index 0 in the `friends` array, which is the object `anna`. `friends[0].luckyNumbers` is the array `[2, 4, 8, 16]` from the object called `anna`. Finally, `friends[0].luckyNumbers[1]` is index 1 in that array, which is the number value 4.

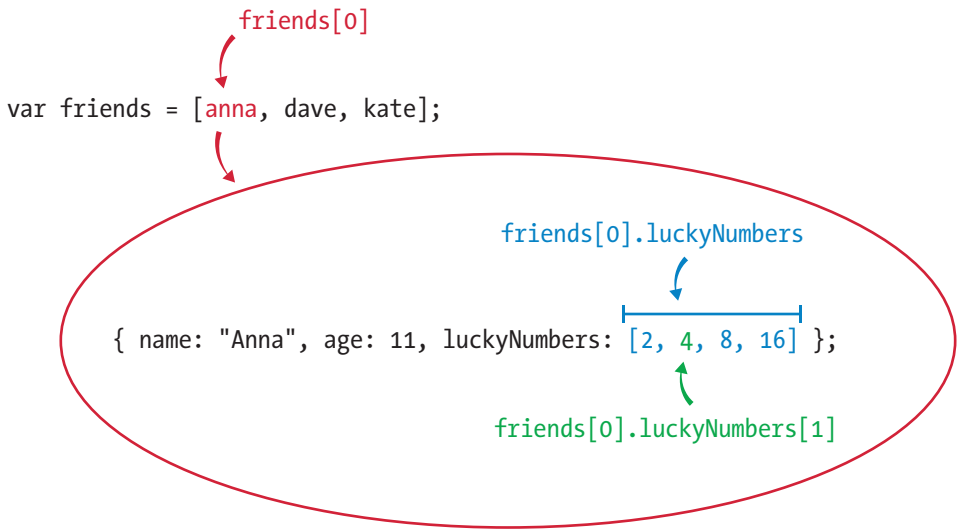


Figure 4-2: Accessing nested values

EXPLORING OBJECTS IN THE CONSOLE

Chrome will let you dig into objects that you print out in the console. For example, if you type . . .

```
friends[1];
```

Chrome will display the output shown in Figure 4-3.

```
friends[1];  
► Object {name: "Dave", age: 5, LuckyNumbers: Array[3]}
```

Figure 4-3: How an object is displayed in the Chrome interpreter

The triangle on the left means that this object can be expanded. Click the object to expand it, and you'll see what's shown in Figure 4-4.

```
friends[1];  
▼ Object {name: "Dave", age: 5, LuckyNumbers: Array[3]} ⓘ  
  age: 5  
  ► luckyNumbers: Array[3]  
    name: "Dave"  
  ► __proto__: Object
```

Figure 4-4: Expanding the object

You can expand luckyNumbers, too, by clicking it (see Figure 4-5).

```
friends[1];
▼ Object {name: "Dave", age: 5, luckyNumbers: Array[3]} ⓘ
  age: 5
  ▼ luckyNumbers: Array[3]
    0: 3
    1: 9
    2: 40
    length: 3
  ► __proto__: Array[0]
  name: "Dave"
  ► __proto__: Object
```

Figure 4-5: Expanding an array within the object

Don't worry about those `__proto__` properties—they have to do with the object's *prototype*. We'll look at prototypes later, in Chapter 12. Also, you'll notice that the interpreter shows the value of the array's `length` property.

You can also view the entire `friends` array and expand each element in the array, as shown in Figure 4-6.

```
friends
[▼ Object ⓘ, ▼ Object ⓘ, ▼ Object ⓘ]
  age: 11
  ► luckyNumbers: Array[4]
  name: "Anna"
  ► __proto__: Object
  age: 5
  ► luckyNumbers: Array[3]
  name: "Dave"
  ► __proto__: Object
  age: 9
  ► luckyNumbers: Array[3]
  name: "Kate"
  ► __proto__: Object
```

Figure 4-6: All three objects from the `friends` array, as shown in the Chrome interpreter

USEFUL THINGS TO DO WITH OBJECTS

Now that you know a few different ways to create objects and add properties to them, let's put what we've learned to use by trying out some simple programs.

KEEPING TRACK OF OWED MONEY

Let's say you've decided to start a bank. You lend your friends money, and you want to have a way to keep track of how much money each of them owes you.

You can use an object as a way of linking a string and a value together. In this case, the string would be your friend's name, and the value would be the amount of money he or she owes you. Let's have a look.

```
❶ var owedMoney = {};  
❷ owedMoney["Jimmy"] = 5;  
❸ owedMoney["Anna"] = 7;  
❹ owedMoney["Jimmy"];  
5  
❺ owedMoney["Jinen"];  
undefined
```

At ❶, we create a new empty object called `owedMoney`. At ❷, we assign the value 5 to the key "Jimmy". We do the same thing at ❸, assigning the value 7 to the key "Anna". At ❹, we ask for the value associated with the key "Jimmy", which is 5. Then at ❺, we ask for the value associated with the key "Jinen", which is undefined because we didn't set it.

Now let's imagine that Jimmy borrows some more money (say, \$3). We can update our object and add 3 to the amount Jimmy owes with the plus-equals operator (`+=`) that you saw in Chapter 2.



```
owedMoney["Jimmy"] += 3;  
owedMoney["Jimmy"];  
8
```

This is like saying `owedMoney["Jimmy"] = owedMoney["Jimmy"] + 3`. We can also look at the entire object to see how much money each friend owes us:

```
owedMoney;  
{ Jimmy: 8, Anna: 7 }
```

STORING INFORMATION ABOUT YOUR MOVIES

Let's say you have a large collection of movies on DVD and Blu-ray. Wouldn't it be great to have the information about those movies on your computer so you can find out about each movie easily?

You can create an object to store information about your movies, where every key is a movie title, and every value is another object containing information about the movie. Values in objects can be objects themselves!

```
var movies = {  
  "Finding Nemo": {  
    releaseDate: 2003,  
    duration: 100,  
    actors: ["Albert Brooks", "Ellen DeGeneres", "Alexander Gould"],  
    format: "DVD"  
  },  
  "Star Wars: Episode VI - Return of the Jedi": {  
    releaseDate: 1983,  
    duration: 134,  
    actors: ["Mark Hamill", "Harrison Ford", "Carrie Fisher"],  
    format: "DVD"  
  },  
  "Harry Potter and the Goblet of Fire": {  
    releaseDate: 2005,  
    duration: 157,  
    actors: ["Daniel Radcliffe", "Emma Watson", "Rupert Grint"],  
    format: "Blu-ray"  
  }  
};
```

You might have noticed that I used quotes for the movie titles (the keys in the outer object) but not for the keys in the inner objects. That's because the movie titles need to have spaces—otherwise, I'd have to type each title like `StarWarsEpisodeVIReturnOfTheJedi`, and that's just silly! I didn't need quotes for the keys in the inner objects, so I left them off. It can make code look a bit cleaner when there aren't unnecessary punctuation marks in it.



Now, when you want information about a movie, it's easy to find:

```
var findingNemo = movies["Finding Nemo"];
findingNemo.duration;
100
findingNemo.format;
"DVD"
```

Here we save the movie information about *Finding Nemo* into a variable called `findingNemo`. We can then look at the properties of this object (like `duration` and `format`) to find out about the movie.

You can also easily add new movies to your collection:

```
var cars = {
  releaseDate: 2006,
  duration: 117,
  actors: ["Owen Wilson", "Bonnie Hunt", "Paul Newman"],
  format: "Blu-ray"
};
movies["Cars"] = cars;
```

Here we create a new object of movie information about *Cars*. We then insert this into the `movies` object, under the key "Cars".

Now that you're building up your collection, you might want to find an easy way to list the names of all your movies. That's where `Object.keys` comes in:

```
Object.keys(movies);
["Finding Nemo", "Star Wars: Episode VI - Return of the Jedi", "Harry Potter and the Goblet of Fire", "Cars"]
```

WHAT YOU LEARNED

Now you've seen how objects work in JavaScript. They're a lot like arrays, because you can use them to hold lots of pieces of information together in one unit. One major difference is that you use strings to access elements in an object and you use numbers to access elements in an array. For this reason, arrays are ordered, while objects are not.

We'll be doing a lot more with objects in later chapters, once we've learned about more of JavaScript's features. In the next chapter, we'll look at *conditionals* and *loops*, which are both ways of adding structure to our programs to make them more powerful.