4 OBJECTS

Objects in JavaScript are very similar to arrays, but objects use strings instead of numbers to access the different elements. The strings are called *keys* or *properties*, and the elements they point to are called *values*. Together these pieces of information are called *key-value pairs*. While arrays are mostly used to represent lists of multiple things, objects are often

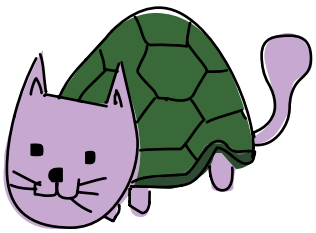
used to represent single things with multiple characteristics, or *attributes*. For example, in Chapter 3 we made several arrays that listed different animal names. But what if we wanted to store different pieces of information about one animal?

CREATING OBJECTS

We could store lots of information about a single animal by creating a JavaScript object. Here's an object that stores information about a three-legged cat named Harmony.

```
var cat = {
  "legs": 3,
  "name": "Harmony",
  "color": "Tortoiseshell"
};
```

Here we create a variable called `cat` and assign an object to it with three key-value pairs. To create an object, we use curly brackets, {}, instead of the straight brackets we used to make arrays. In between the curly brackets, we enter key-value pairs. The curly brackets and everything in between them are called an *object literal*. An object literal is a way of creating an object by writing out the entire object at once.



NOTE

We've also seen array literals (for example, `["a", "b", "c"]`), number literals (for example, `37`), string literals (for example, `"moose"`), and Boolean literals (`true` and `false`). Literal *just means that the whole value is written out at once, not built up in multiple steps.*

For example, if you wanted to make an array with the numbers 1 through 3 in it, you could use the array literal `[1, 2, 3]`. Or you could create an empty array and then use the `push` method to add 1, 2, and 3 to the array. You don't always know at first what's going to be in your array or object, which is why you can't always use literals to build arrays and objects.

Figure 4-1 shows the basic syntax for creating an object.

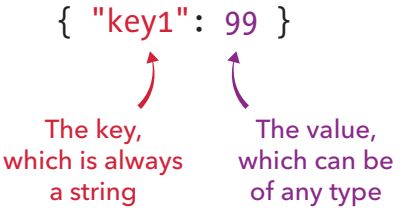


Figure 4-1: The general syntax for creating an object

When you create an object, the key goes before the colon (:), and the value goes after. The colon acts a bit like an equal sign—the values on the right get assigned to the names on the left, just like when you create variables. In between each key-value pair, you have to put a comma. In our example, the commas are at the ends of the lines—but notice that you don't need a comma after the last key-value pair (`color: "Tortoiseshell"`). Because it's the last key-value pair, the closing curly bracket comes next, instead of a comma.

KEYS WITHOUT QUOTES

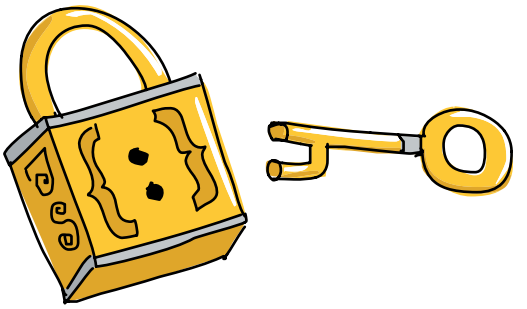
In our first object, we put each key in quotation marks, but you don't necessarily need quotes around the keys—this is a valid `cat` object literal as well:

```
var cat = {
  legs: 3,
  name: "Harmony",
  color: "Tortoiseshell"
};
```

JavaScript knows that the keys will always be strings, which is why you can leave out the quotes. If you don't put quotes around the keys, the unquoted keys have to follow the same rules as variable names: spaces aren't allowed in an unquoted key, for example. If you put the key in quotes, then spaces are allowed:

```
var cat = {
  legs: 3,
  "full name": "Harmony Philomena Snuggly-Pants Morgan",
  color: "Tortoiseshell"
};
```

Note that, while a key is always a string (with or without quotes), the value for that key can be any kind of value, or even a variable containing a value.



You can also put the whole object on one line, but it can be harder to read like that:

```
var cat = { legs: 3, name: "Harmony", color: "Tortoiseshell" };
```

ACCESSING VALUES IN OBJECTS

You can access values in objects using square brackets, just like with arrays. The only difference is that instead of the index (a number), you use the key (a string).

```
cat["name"];
"Harmony"
```

Just as the quotes around keys are optional when you create an object literal, the quotes are also optional when you are accessing keys in objects. If you're not going to use quotes, however, the code looks a bit different:

```
cat.name;
"Harmony"
```

This style is called *dot notation*. Instead of typing the key name in quotes inside square brackets after the object name, we just use a period, followed by the key, without any quotes. As with unquoted keys in object literals, this will work only if the key doesn't contain any special characters, such as spaces.

Instead of looking up a value by typing its key, say you wanted to get a list of all the keys in an object. JavaScript gives you an easy way to do that, using `Object.keys()`:

```
var dog = { name: "Pancake", age: 6, color: "white", bark: "Yip yap ↵
yip!" };
var cat = { name: "Harmony", age: 8, color: "tortoiseshell" };
Object.keys(dog);
["name", "age", "color", "bark"]
Object.keys(cat);
["name", "age", "color"]
```

`Object.keys(`*anyObject*`)` returns an array containing all the keys of *anyObject*.

ADDING VALUES TO OBJECTS

An empty object is just like an empty array, but it uses curly brackets, { }, instead of square brackets:

```
var object = {};
```

You can add items to an object just as you'd add items to an array, but you use strings instead of numbers:

```
var cat = {};
cat["legs"] = 3;
cat["name"] = "Harmony";
cat["color"] = "Tortoiseshell";
cat;
{ color: "Tortoiseshell", legs: 3, name: "Harmony" }
```

Here, we started with an empty object named `cat`. Then we added three key-value pairs, one by one. Then, we type `cat;`, and the browser shows the contents of the object. Different browsers may output objects differently, though. For example, Chrome (at the time I'm writing this) outputs the `cat` object like this:

```
Object {legs: 3, name: "Harmony", color: "Tortoiseshell"}
```

While Chrome prints out the keys in that order (`legs`, `name`, `color`), other browsers may print them out differently. This is

because JavaScript doesn't store objects with their keys in any particular order.

Arrays obviously have a certain order: index 0 is before index 1, and index 3 is after index 2. But with objects, there's no obvious way to order each item. Should `color` go before `legs` or after? There's no "correct" answer to this question, so objects simply store keys without assigning them any particular order, and as a result different browsers will print the keys in different orders. For this reason, you should never write a program that relies on object keys being in a precise order.

ADDING KEYS WITH DOT NOTATION

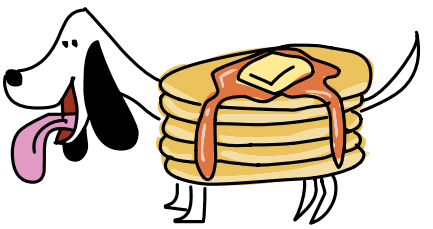
You can also use dot notation when adding new keys. Let's try the previous example, where we started with an empty object and added keys to it, but this time we'll use dot notation:

```
var cat = {};
cat.legs = 3;
cat.name = "Harmony";
cat.color = "Tortoiseshell";
```

If you ask for a property that JavaScript doesn't know about, it returns the special value `undefined`. `undefined` just means "There's nothing here!" For example:

```
var dog = {
  name: "Pancake",
  legs: 4,
  isAwesome: true
};
dog.isBrown;
undefined
```

Here we define three properties for `dog`: `name`, `legs`, and `isAwesome`. We didn't define `isBrown`, so `dog.isBrown` returns `undefined`.



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`.

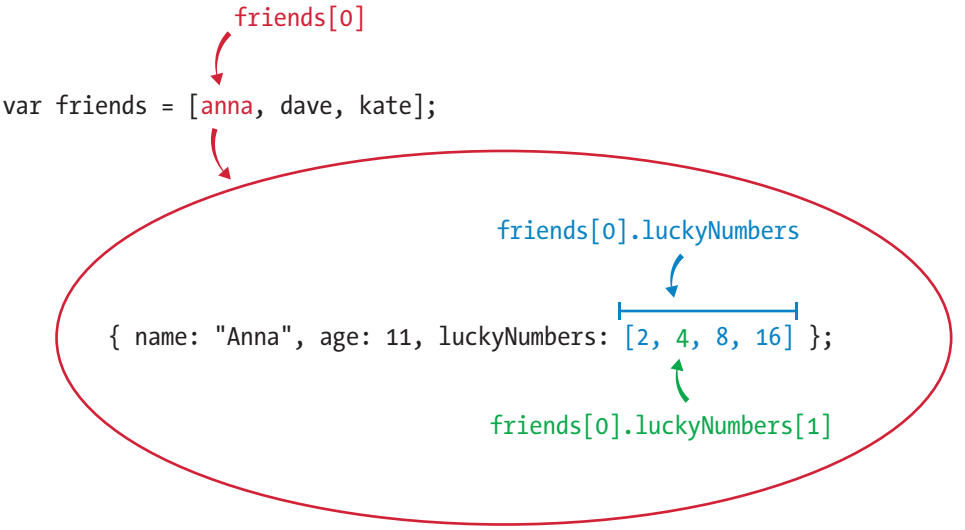


Figure 4-2: Accessing nested values

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                    age: 5                    age: 9
  ▶ luckyNumbers: Array[4]   ▶ luckyNumbers: Array[3]  ▶ luckyNumbers: Array[3]
    name: "Anna"               name: "Dave"              name: "Kate"
  ▶ __proto__: Object        ▶ __proto__: Object       ▶ __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.

PROGRAMMING CHALLENGES

Try out these challenges to practice working with objects.

#1: SCOREKEEPER

Imagine you're playing a game with some friends and you want to keep track of the score. Create an object called `scores`. The keys will be the names of your friends, and the values will be the scores (which will all start at 0). As the players earn points, you must increase their scores. How would you increase a player's score in the `scores` object?

#2: DIGGING INTO OBJECTS AND ARRAYS

Say you had the following object:

```
var myCrazyObject = {
  "name": "A ridiculous object",
  "some array": [7, 9, { purpose: "confusion", number: 123 }, 3.3],
  "random animal": "Banana Shark"
};
```

How would you get the number 123 out of this object using one line of JavaScript? Try it out in the console to see if you're right.