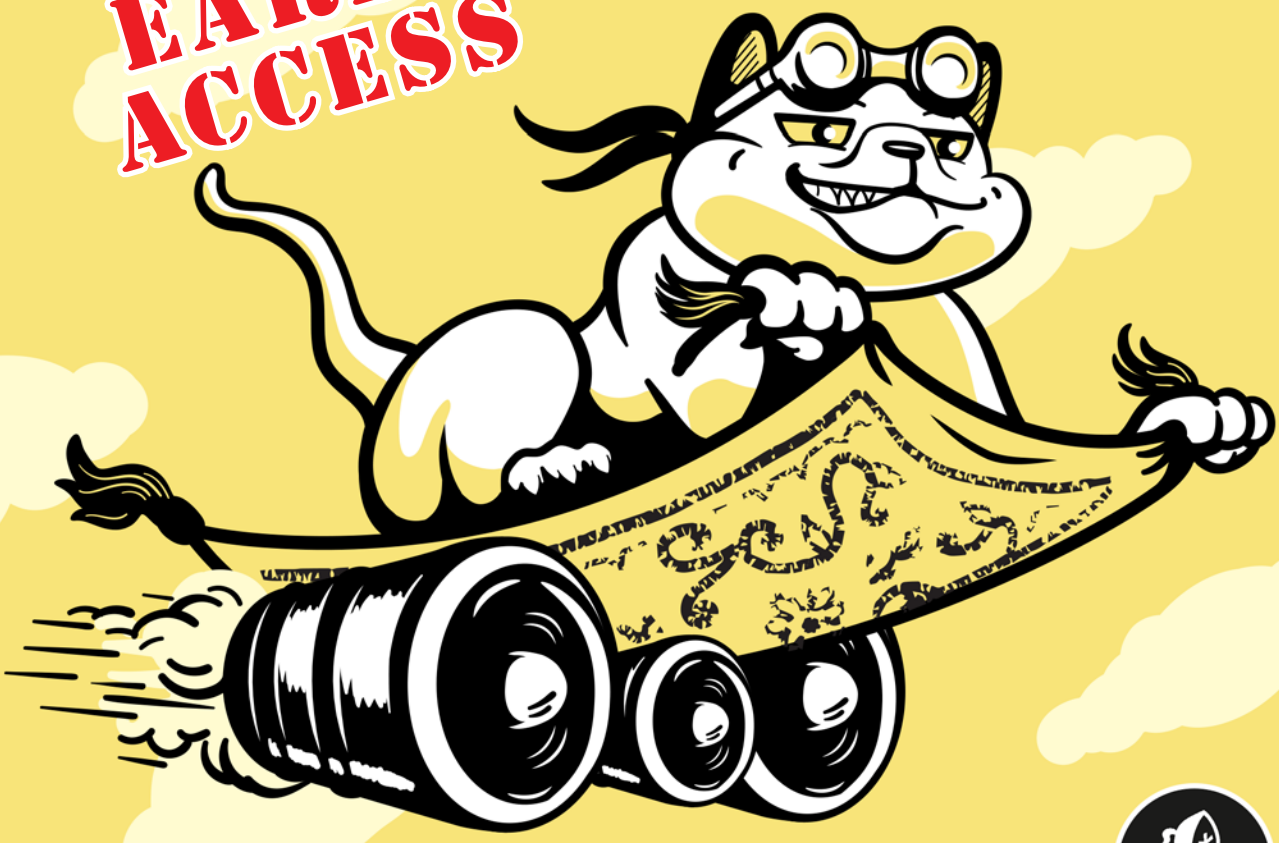
JAVASCRIPT CRASH COURSE

NICK MORGAN

EARLY ACCESS



no starch press

NO STARCH PRESS EARLY ACCESS PROGRAM: FEEDBACK WELCOME!

The Early Access program lets you read significant portions of an upcoming book while it's still in the editing and production phases, so you may come across errors or other issues you want to comment on. But while we sincerely appreciate your feedback during a book's EA phase, please use your best discretion when deciding what to report.

At the EA stage, we're most interested in feedback related to content—general comments to the writer, technical errors, versioning concerns, or other high-level issues and observations. As these titles are still in draft form, we already know there may be typos, grammatical mistakes, missing images or captions, layout issues, and instances of placeholder text. No need to report these—they will all be corrected later, during the copyediting, proofreading, and typesetting processes.

If you encounter any errors ("errata") you'd like to report, please fill out **this Google form** so we can review your comments.

JAVASCRIPT CRASH COURSE

NICK MORGAN

Early Access edition, 5/2/22

ISBN 13: 978-1-7185-0226-0 (print)
ISBN 13: 978-1-7185-0227-7 (ebook)

Publisher: William Pollock
Managing Editor: Jill Franklin
Production Manager: Rachel Monaghan
Developmental Editor: Nathan Heidelberger
Production Editor: Jenn Kepler
Cover Illustrator: Gina Redman
Interior Design: Octopod Studios
Compositor: Happenstance Type-O-Rama

CONTENTS

The chapters in **red** are included in this Early Access PDF.

4

OBJECTS

Objects are a data structure similar to arrays, but which use strings known as *keys* to access the values. Each key is associated with a value, which together is known as a *key-value pair*.

While we use arrays to store ordered lists of different elements, objects are usually used to store multiple pieces of information about a single entity, such as a person's name and age. In this chapter we'll learn how to create and manipulate objects.

Creating Objects

Objects can be created with *object literals*, which consist of a pair of curly braces, enclosing a series of key-value pairs, separated by commas. Each key-value pair must have a colon between the key and the value. 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"}
```

We create a new object with three keys: `"title"`, `"released"`, and `"director"`. Each key has a value associated with it. I use new lines to separate each key-value pair simply because it can be easier to read that way, but it's not strictly necessary.

All keys in objects are strings, but if your key is a valid identifier you can leave off the quotes, which is common practice among JavaScript programmers:

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

`key 3` contains a space, so we have to use quotes.

WHAT IS A VALID IDENTIFIER?

A valid identifier is a series of characters that can be used as a variable name, or as an un-quoted object key. JavaScript identifiers can contain letters, numbers, and the characters `_` and `$`. They cannot start with a number.

Invalid characters include symbols like `*` or `(` or `#`, as well as white space characters like space and new line. These characters are all allowed in object keys, but only if the key is enclosed in quotes.

Accessing Object Values

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

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

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

For keys that are also valid identifiers you can use dot notation:

```
obj.key_2;
2
```

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

You might notice that this dot notation looks like the syntax we used for

properties like `.length` in Chapters 2 and 3. That's because it's the same thing! A property is really just another name for a key. JavaScript treats strings like objects, and arrays are just a kind of object, so something like `[1, 2, 3].length;` is actually getting the value associated with the `length` key. You can even access these properties with bracket notation if you want:

```
[1, 2, 3]["length"];
3
```

Setting Object Values

You can also use bracket notation or dot notation to update or set new key-value pairs. Here we'll set up an empty dictionary, 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 an empty object `dictionary` set to a pair of empty curly braces. We then set two new properties: `"mouse"` and `"computer mouse"` and give them definitions as values. As before, we use bracket notation for `"computer mouse"` because it contains a space and therefore is not a valid identifier.

Nesting Arrays and Objects

As with arrays, we can nest objects in other objects. We can also nest objects in arrays, and arrays in objects! We create 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. I'll show both these techniques.

Nested Array and Object Literals

We'll create an array of objects that represents your favorite book trilogies:

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

```
];
```

The variable `favoriteTrilogies` contains an array of two elements, each of which is an object. These objects both contain a key called `books`, which itself contains an array of strings.

To get the name of the first book from the second trilogy you use a combination of array indexing and object dot notation, specifying the index of the second object in the outer array, the key in that object that gives the inner array, and then the index of the element within that array.

```
favoriteTrilogies[1].books[0];
"The Fifth Season"
```

Building Nested Structures with Variables

An alternative technique 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 you want to model some coins, and then create an object that represents just the coins in your pocket.

```
let cent = { name: "Cent", 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 };
let change = [quarter, quarter, dime, cent, cent, cent];
change[0].value;
25
```

Here we create four objects representing four different kinds of coins. We then create an array containing a combination of these coin objects.

You'll see that some of the coin objects appear in the array multiple times. This is one advantage of creating the elements of the array before we create the array—the same element can be repeated within the same array without having to repeat the object literal.

Another interesting consequence of building the array like this is that the repeated elements share a common identity. For example, `change[3]` and `change[4]` refer to the same cent object. If we decided to update the weight of a cent, that weight would be reflected in all the `cent` elements of the change array:

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

Any changes to `cent` will be shown in each occurrence in the `change` array.

TRY IT OUT

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 gives you the ability to explore nested objects, like we did in Chapter 3 with nested arrays. We'll create a deeply nested object and try to look inside.

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

As you can see, you can access the content of the inner-most object by chaining together `nested.content.content.content`.

When we ask for the value of `nested`, the console just gives an abbreviated version with the value of `content` shown as `{…}` to imply that there is an object here but there isn't room to display it. Click the arrow to the left 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, and then one more time to expand the object with `name: "Inner"`. You should now see the entire content of the object, as below:

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

      ▸ __proto__: Object
    ▸ __proto__: Object
  ▸ __proto__: Object
```

As mentioned in Chapter 3, the `__proto__` properties will be explained in Chapter 7, when we learn about object prototypes.

Printing Objects with JSON.stringify

You can also view objects as *JSON*, or *JavaScript Object Notation*, a textual data format based on JavaScript object and array literals that's heavily used across the Web and beyond. `JSON.stringify` converts a JavaScript object into a JSON string.

Let's take the `nested` object and convert it into a JSON string:

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

The returned string is an unformatted equivalent of the original literal we used to create `nested`. To format the literal, you pass an argument that represents the number of spaces to indent each new nested object:

```
JSON.stringify(nested, null, 2);
"{
  "name": "Outer",
  "content": {
    "name": "Middle",
    "content": {
      "name": "Inner",
      "content": "Whoa..."
    }
  }
}"
```

The second argument lets you define a replacer function which can be used to modify the output by optionally 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 give a new line after each opening brace, and then two extra characters of space for each additional level of nesting.

Calling `JSON.stringify` in this way is helpful for getting a quick visual representation of an object without having to click into each key in the console.

Working with Objects

Objects have plenty of methods, and we'll examine a few of the most common ones here. Unlike arrays, where the methods are called directly on the array you want to operate on, object methods are called as *static methods* on the `Object` constructor, passing the object you want to operate on as an argument. We'll learn more about static methods and constructors in Chapter 7, but briefly, a *constructor* is a type of function used to create objects, and static methods are methods defined directly on the constructor.

Getting an Object's Keys

To get an array of all the keys of an object you can use the static method `Object.keys`. 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.

Methods like `Object.keys` can't be called as methods on the object itself, because then you wouldn't be able to define a `keys` property on any object you created; the name would conflict with the `keys` method. There are a lot of these methods, so the designers of JavaScript decided to make them available as static methods, where they won't conflict with your keys.

`Object.keys` can be helpful in cases like this where the only information you need from an object is 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.

Getting an Object's Keys and Values

To get an array of the keys *and* values you use `Object.entries`. This static method returns an array of two-element arrays, where the first element of the inner array is the key and the second is the value.

```
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 view the full contents, click on the arrow to the left.

```
▾ (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
__proto__: Array(0)
```

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

`Object.entries` is useful when you want to convert an object into an array so you can loop over its key-value pairs. We'll see how loops work in Chapter 5.

Combining Objects

The `Object.assign` method lets you combine multiple objects into one. For example, say you had 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" };
let target = {};
Object.assign(target, physical, contents);
▸ {pages: 208, binding: "Hardcover", genre: "Fiction", subgenre:
"Mystery"}
```

The first argument to `Object.assign` is the *target* object, that is, the object that the keys from the other objects are *assigned* to. In this case, we use `{}` which is an empty object. The next two arguments are the objects whose key-value pairs you want copied into the target object. These two objects are untouched, but the target object is mutated. You can pass as many objects after the initial target argument as you want—we're just doing two here.

`Object.assign` mutates and returns the target object with the key-value pairs copied from the other objects.

The first argument of `Object.assign` is always mutated. You don't have to use an empty object as the target, but if you don't then you'll be modifying one of your input objects. For example, we could remove the first argument from the previous call and get the same return value:

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

The problem here is that `physical` is now the target object, and so gains all the key-values pairs from `contents`:

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

All of the key-value pairs from the `contents` object have been copied into the `physical` object, which now contains both sets of key-value pairs, which is usually not what you want. For this reason, it's common practice to use an empty object as the first argument to `Object.assign`.

Conclusion

In this chapter you learned the basics of JavaScript objects, and a little about JavaScript Object Notation, or JSON. You should have a good understanding of

the differences between objects and arrays, and have some idea of why you would choose one over the other. In the next chapter we'll look at conditionals and loops, which will allow us to write much more complex and interesting programs, by introducing logic.

5

CONDITIONALS AND LOOPS

Conditionals and *loops* are how you can add *logic* to your programs, allowing your code to make decisions based on conditions. Conditionals allow you to run a particular bit of code only if some condition is `true`. Loops will keep on running the same piece of code for as long as some condition is `true`. Together, conditionals and loops are known as *control structures* because they let you control when and how often your code runs.

In this chapter you'll learn how to conditionally run code with `if` statements and how to loop code with `while` and `for` statements. You'll also learn some techniques for looping over the elements of arrays and objects. This is helpful, for example, if you want to do something with every element in an array.

Making Decisions with Conditionals

Conditionals let you run particular code when some condition you set is `true`. There are two main kinds of conditional statement: `if` statements and `if...else` statements.

if Statements

An `if` statement runs code if some condition is `true`. For example, let's create a program that logs a message to the console if a value is greater than a certain threshold. Open VS Code, create a new file called *if.html*, and enter the following. Then open the file in Chrome (see page xxx for instructions).

```
<html><body><script>
let speed = 30;
console.log(`Your current speed is ${speed} mph.`);
1 if (speed > 25) {
2  console.log("Slow down!");
}
</script></body></html>
```

Listing 5-1 An `if` statement

This code is checking the value of `speed` and, if the value is above the threshold we set in the `if` statement, it runs a particular piece of code. If the value of `speed` is not above the threshold, it does nothing.

When you run this, you'll see the following output in the JavaScript console:

```
Your current speed is 30 mph.
Slow down!
```

The `if` statement here has two parts: the *condition* inside the parentheses 1 and the code to run if the condition is true, called the *body*, which is everything between the curly braces 2. Here, the condition is `speed > 25` and the code to run if that is true is `console.log("Slow down!");`. Because `speed` is greater than `25`, this code will print the string `Slow down!` to the console.

Now let's see what happens if the condition isn't met. Change `speed` to `20` and reload the page. This time you'll just see the following output:

```
Your current speed is 20 mph.
```

Because `speed > 25` is now `false`, the code inside the braces isn't run, but the code outside the `if` statement body does still run.

if...else statements

Sometimes you'll want to run one piece of code when a condition is true, and another when that condition is false. For this we use an `if...else` statement. Create a new file called *ifelse.html* and enter the following:

```
<html><body><script>
let speed = 20;
```

```
console.log(`Your current speed is ${speed} mph.`);
if (speed > 25) {
1 console.log("Slow down!");
} else {
2 console.log("You're obeying the speed limit.");
}
</script></body></html>
```

Listing 5-2 An `if...else` statement

Like Listing 5-1, this code will check if `speed` is over some threshold. If it is, we run the first block of code, and if it's not we run the second block.

Running this in Chrome will output the following:

```
Your current speed is 20 mph.
You're obeying the speed limit.
```

An `if...else` statement has two bodies. The first body 1 is run if the condition is `true`, and the second body 2 is run if the condition is `false`. In this case the condition is `false`, so the second body runs.

TRY IT OUT

Create a new file called *cointoss.html* and add the usual setup code. Generate a random number with `Math.random()`, then write an `if...else` statement that will log `"heads"` if the number is less than 0.5 and `"tails"` otherwise. Every time you reload this file you'll get a new coin value!

Chaining if...else statements

If you need to check multiple conditions you can chain together multiple `if...else` statements. Create a new file called *ifelseif.html* with the following code:

```
<html><body><script>
let speed = 20;
console.log(`Your current speed is ${speed} mph.`);
if (speed > 25) {
  console.log("Slow down!");
} else if (speed > 15) {
  console.log("You're driving at a good speed.");
} else {
  console.log("You're driving too slowly.");
}
</script></body></html>
```

Listing 5-3 A chained `if...else` statement with three bodies

This listing checks if `speed` is over a certain threshold. If so, it runs the first body. It then checks if it's over another threshold, in which case it runs the second body. If neither condition is true it runs the final body.

Running this will output:

```
Your current speed is 20 mph.
You're driving at a good speed.
```

This is very similar to Listing 5-2 except now there are three sections: `if`, `else if`, and `else`. If the first condition is true, the first body runs. Otherwise, if the second condition is true, the second body runs. If none of the conditions are true, the final `else` body runs. Just like a normal `if...else` statement, in a chained `if...else` statement only one of the bodies will run, that is, the first body that meets the condition.

You can include as many `else if` sections as you want:

```
if (speed > 25) {
  console.log("Slow down!");
} else if (speed > 20) {
  console.log("You're driving at a good speed.");
} else if (speed > 15) {
  console.log("You're driving a little bit too slowly.");
} else if (speed > 10) {
  console.log("You're driving too slowly.");
} else {
  console.log("You're driving far too slowly!");
}
```

Listing 5-4 A chained `if...else` statement with five bodies

It's important to note that falling through to a later condition means that the earlier conditions had to be false. For example, when we write `if (speed > 20)` in Listing 5-4, it's on the understanding that `if (speed > 25)` is false. Therefore, `if (speed > 20)` actually means `if (speed > 20 && speed <= 25)` in this context, but since we already know `speed` can't be greater than 25, we don't need to specify the `&& speed <= 25` part. Table 5-1 shows the full conditions for each section in Listing 5-4.

Table 5-2 Each condition and output for the if...else statement in Listing 5-4

Condition	Output
`speed > 25`	`Slow down!`
`speed > 20 && speed <= 25`	`You're driving at a good speed.`
`speed > 15 && speed <= 20`	`You're driving a little bit too slowly.`
`speed > 10 && speed <= 15`	`You're driving too slowly.`
`speed <= 10`	`You're driving far too slowly!`

DO YOU NEED BRACES?

It's possible to write `if` and `if...else` statements without braces, if each body is a single statement. For example, this is valid:

```
if (speed > 25)
  console.log("Slow down!");
else
  console.log("You're driving under the speed limit.");
```

and so is this:

```
if (speed > 25) console.log("Slow down!");
else console.log("You're driving under the speed limit.");
```

These examples works fine, but if you have more than one line in the body you need to include the braces. In this book we'll always use braces to surround the body, for consistency.

Repeating Code with Loops

Loops are another form of control structure in JavaScript that let you run the same code multiple times. There are four main kinds of loops you'll learn in this chapter: `while` loops, `for` loops, `for...in` loops and `for...of` loops. Let's start with `while` loops.

while loops

The `while` loop is the simplest kind of loop; in essence, the `while` loop says "while some condition is true, run this code". In that sense, it's similar to an `if` statement, the difference being that the `while` loop will keep running the code as long as the condition is `true`, whereas an `if` statement will only run it at most once. Often you'll need to write code that runs multiple times, instead of just once. This allows your program to keep running as long as it needs to, instead of just running through once and stopping.

Let's try it out. Create a new file called *while.html* and enter the following:

```
<html><body><script>
let speed = 30;
1 while (speed > 25) {
  console.log(`Your current speed is ${speed} mph.`);
  2 speed--;
  }
3 console.log(`Now your speed is ${speed} mph.`);
</script></body></html>
```

Listing 5-5 A `while` loop

This loop will keep running the same code, outputting a line of text and decrementing `speed` until the condition is `false`.

Running this will output the following:

```
Your current speed is 30 mph.
Your current speed is 29 mph.
Your current speed is 28 mph.
Your current speed is 27 mph.
Your current speed is 26 mph.
Now your speed is 25 mph.
```

The first time we hit the `while` loop at 1, `speed` is `30`, so `speed > 25` is `true`. This means the body of the while loop is run, which outputs some text. Then at 2, we decrement the `speed` variable by 1 so the next time around the loop

we have a different value for speed. At the end of the loop, we go back to the start 1 and check the condition again. If it's still `true`, we run the body again, then go back to the start and check the condition, and so on. This keeps happening until the condition is no longer met, so when `speed` gets to `25`. We again check the condition, which is now `false` (`25 > 25` is `false`). This causes JavaScript to jump to the first line of code following the while loop, at 3, and output a final line of text.

BEWARE OF INFINITE LOOPS!

It's very easy to accidentally end up in an *infinite loop* where your loop never stops looping until you close the browser tab. If you're unlucky you might even cause your browser to crash!

For example, if we changed Listing 5-5 to use `speed++` instead of `speed--` we'd end up in an infinite loop. The output would look something like this:

```
Your current speed is 30 mph.
Your current speed is 31 mph.
Your current speed is 32 mph.
Your current speed is 33 mph.
Your current speed is 34 mph.
Your current speed is 35 mph.
Your current speed is 36 mph.
...
```

The speed would keep increasing until you closed the browser tab, because the condition would never not be met. If you ever end up in a situation like this, read your code back carefully, and make sure that the condition will eventually become `false`.

for loops

A `for` loop is another kind of loop. `for` loops make it easier to keep track of your position in the loop by giving you the opportunity to set up some state, update it, and check it. Like a `while` loop, a `for` loop keeps looping as long as some condition is `true`.

Often loops have a particular *looping variable*, which we use for keeping track of the state of the loop, like the `speed` variable in Listing 5-5. A common pattern is to set the looping variable to a starting value, update it somehow, and check some condition based on the looping variable. A `for` loop is just a more convenient way to write this pattern.

With `for` loops we move the setup and updating of the looping variable into the first line of the loop. Let's rewrite the previous example to use a `for` loop instead of a `while` loop. Save the following in *for.html*.

```
<html><body><script>
for (let speed = 30; speed > 25; speed--) {
  console.log(`Your current speed is ${speed} mph.`);
}
</script></body></html>
```

Listing 5-6 A `for` loop

The first line of the `for` loop has three parts: we *set up* the initial variable value with `let speed = 30`, set the *condition* with `speed > 25`, and then *update* the variable in the same line with `speed--` to decrement the variable with each loop. You separate each part with a semicolon.

Running this will output mostly the same as the `while` loop from earlier:

```
Your current speed is 30 mph.
Your current speed is 29 mph.
Your current speed is 28 mph.
Your current speed is 27 mph.
Your current speed is 26 mph.
```

The only difference is that we don't have access to the `speed` variable outside of the `for` loop, so we can't log the final speed at the end. This is actually one of the advantages of `for` loops: the looping variable stays within the scope of the loop and can't be accidentally used outside of the loop.

There's nothing you can do with a `for` loop that you can't do with a `while` loop, but most programmers find `for` loops easier to read than the equivalent `while` loop, because all the looping logic is confined to one place.

Looping Over an Array With for...of

You can use the `for…of` loop to loop over the items in an array. Where `while` loops and `for` loops keep looping as long as some condition is `true`, `for...of` loops go over each item in an array, one at a time, and stop when they run out of items.

Let's have a look at a `for...of` loop in action. Create a new file called *forof.html*:

```
<html><body><script>
let colors = ["Red", "Green", "Blue"];

for (let color of colors) {
  console.log(`${color} is a color.`);
}
</script></body></html>
```

Listing 5-7 Looping over an array with a `for...of` loop

This code logs a sentence for each color in the array `colors`, then stops. This will output the following:

```
Red is a color.
Green is a color.
Blue is a color.
```

We create an array containing three strings. We then use the statement `for (let color of colors)` to set the variable `color` to each element in `colors`, one at a time. The first time around the loop, the looping variable

`color` will be set to `"Red"`. The second time it will be set to `"Green"`. Finally, the third time around it will be set to `"Blue"`. When it runs out of items, the loop ends.

It's also possible to use a regular `for` loop to loop over an array:

```
for (let index = 0; index < colors.length; index++) {
  console.log(`${colors[index]} is a color.`);
}
```

Listing 5-8 Using a regular `for` loop instead of a `for...of` loop

You'll see this style a lot in older code. For a long time this was the only way to loop over an array in JavaScript, so it's worth being able to recognize it. One benefit of this technique is that it gives you access to the array index. Sometimes it's important to know which element of the array you're dealing with. For example, you might want to do something different with even and odd indexes, or you might just want to print out the index to make a numbered list, like we do in Listing 5-9 below. To do this with a `for...of` loop you can use the `entries()` method on the array (you'll learn more about methods in Chapter XX).

```
for (let [index, color] of colors.entries()) {
  console.log(`${index}: ${color} is a color.`);
}
```

Listing 5-9 Using `.entries()` to get access to the indexes in an array

This outputs:

```
0: Red is a color.
1: Green is a color.
2: Blue is a color.
```

The `entries()` method gives you a list of elements where each original item is paired with the index of that item. In this case, if `colors` is `["Red", "Green", "Blue"]` then `colors.entries()` gives you something like `[[0, "Red"], [1, "Green"], [2, "Blue"]]`. This lets you loop over pairs of `[index, item]` instead of just looping over the items.

TRY IT OUT

It's also possible to loop over strings. Write a `for...of` loop that will loop over your name, printing each letter on a separate line, for example:

```
N
i
c
k
```

Now see if you can print out the following:

```
N 0
i 1
c 2
```

```
k 3
```

To use the `entries()` technique from earlier you'll first have to convert the string to an array. To do that, use the `split` method: `name.split("").entries()`.

Finally, rewrite this but with a standard `for` loop. Which version do you prefer? Think about efficiency and readability.

for...in loops

To loop over the keys in an object you can use a `for...in` loop. This works similarly to a `for...of` loop, but with objects instead of arrays, looping over the keys, not the values.

Save the following as *forin.html*:

```
<html><body><script>
let me = {
  firstName: "Nick",
  lastName: "Morgan",
  age: 35
};

for (let key in me) {
  console.log(`My ${key} is ${me[key]}.`);
}
</script></body></html>
```

Listing 5-10 Looping over the keys in an object with a `for...in` loop

This code loops over each key in the object, and outputs a string based on the key and the value. Running the code will output:

```
My firstName is Nick.
My lastName is Morgan.
My age is 35.
```

The first thing we do is to create an object with three keys, then loop over the keys. Like a `for...of` loop, each time round the loop, the looping variable will be set to a different item. In this case, `key` is set to each key from the object `me`. To get the value from the object we can use the subscript notation, `me[key]`, which you learned in Chapter XX.

Conclusion

This chapter introduced conditionals and loops, the control structures that let you control how your programs run. Conditionals are used to run code *conditionally*, if the condition you set is true. Loops let you run the same code multiple times.

6

FUNCTIONS

JavaScript lets you package up code for performing a certain task using *functions*. These functions can then be called, which will run the associated code. Functions make it so you don't have to repeat code every time you have to perform some task.

In this chapter you'll learn to write your own functions, and make those functions more flexible with *parameters*.

Creating and Calling Functions

We'll begin with a basic function that takes an argument and logs it out to the console with the text "Hello". Open the JavaScript console and enter the following:

```
1 function 2sayHello(3name) {
  4 console.log(`Hello, ${name}!`);
  }
```

Listing 6-1 Our first function

This code is called a *function declaration*, because we're declaring a function. Here we create a function called `sayHello` with one parameter, `name`. Parameters are pieces of information a function needs to do its job.

A function declaration has four parts: the `function` keyword to tell JavaScript we're creating a function 1, the name you give the function 2 (`sayHello` in this case), a comma-separated list of parameter names 3 surrounded by parentheses (`name` in this case), and a body 4 surrounded by braces (`console.log(`Hello, ${name}!`);` in this case).

Let's test our function out.

```
sayHello("Nick");
Hello, Nick!
undefined
```

Listing 6-2 Calling our first function

We call our `sayHello` function, passing the argument `"Nick"`. When you call a function, the values of the arguments are assigned to the parameters, and the function body is run with the parameters set to those values. You can imagine that `name` is a variable, and here you assign the value `"Nick"` to that variable.

The first line of output is the result of running the function body with `name` set to `"Nick"`. The second line is the return value of the function. Because we didn't explicitly return a value, `undefined` is returned. In the next section we'll see how to return a value from a function.

ARGUMENTS AND PARAMETERS

The distinction between arguments and parameters can be confusing. Put simply, the parameters are the names of a function's inputs. Each function has only one set of parameters. The arguments are the actual values passed to the function when you call it, and so every time a function is called it can have a new set of arguments.

The `sayHello` function has one parameter, `name`, but could be called with a different argument each time, for example, `sayHello("Mei")`, `sayHello("JavaScript")`, and so on.

Returning Values from Functions

To have your function return a value, you use the `return` keyword in the body of the function, followed by an expression.

```
function add(x, y) {
  return x + y;
}
```

Listing 6-3 A function that returns a value

The `add` function has two parameters, `x` and `y`. We use the `return` keyword to return a value from the function. Reload the page and run the following in the console:

```
add(1, 2);
3
```

Listing 6-4 Calling a function that returns a value

When you call `add` with two arguments, it sets `x` and `y` to those values and runs the body. The body sums those two arguments and returns the value.

It's important to understand the difference between printing a string to the console and returning a value. With the functions we've looked at so far, the difference isn't completely obvious, because JavaScript also prints the return value of a function to the console. The difference is this: if your function returns a value, then you can use that value later in your code, but if you just log a value, the only place that value exists is in the log.

With our `add` function, we can save the result in a variable, and then do something with that value:

```
let sum = add(500, 500);
`I walked ${sum} miles`;
"I walked 1000 miles"
```

Listing 6-5 Using the return value of a function

When you call `add(500, 500)` here nothing gets logged, because we're storing the return value in a variable called `sum`. The value is then used to create a string.

This wouldn't be possible with the `sayHello` function because it returns `undefined`. There's no way to access the string that was logged to the console.

Parameter Types

JavaScript is a *dynamically typed* programming language, which means that the types of variables and parameters can change while the program is running. This is opposed to *statically typed* languages, where the types of variables and parameters are fixed before the program is run.

Because of this, JavaScript parameters have no concept of type. For example, so far we've only passed numbers to the `add` function, but there's nothing stopping us from passing other types, like Booleans, strings, objects, or

arrays. You could even pass `null` or `undefined`.

```
add(1, "1");
"11"
add({}, {});
"[object Object][object Object]"
add(123, null);
123
```

Listing 6-6 Mixing argument types

As you learned in Chapter 2, JavaScript has some complicated rules around coercion, which is why these calls return somewhat confusing results. When we try to add `1` and `"1"` with the `+` operator, JavaScript converts both of the operands to strings before concatenating them.

Dynamic typing brings a lot of flexibility to JavaScript, but it can also open the way for some confusing bugs. It's essential to have a good idea of the types you're using, to make sure you're not passing a string to a function that expects a number, for example.

Side Effects

The *side effects* of a function are any changes to the *environment* caused by calling that function. These could include updating the value of a variable defined outside of the function, modifying an array or object, or outputting a string to the console (the console is part of the environment so this also counts as a side effect). In simple terms, a side effect is anything a function does that isn't returning a value, and that makes a difference outside of the function. Side effects can be intended or unintended.

As an example of some intended side effects, we could redefine our `add` function to log some information to the console with `console.log` and update a variable called `addCalls` in addition to returning the sum of its arguments:

```
let addCalls = 0;

function add(x, y) {
  addCalls++;
  console.log(`x was ${x} and y was ${y}`);
  return x + y;
}
```

Listing 6-7 A function that returns a value and has side effects

Now, when you call this function in the console, you'll see the value of the arguments printed:

```
add(Math.PI, Math.E);
x was 3.141592653589793 and y was 2.718281828459045
```

```
5.859874482048838
```

Listing 6-8 Calling the side-effecting function and observing logging

Also, every time you call the function, `addCalls` will be incremented:

```
addCalls;
1
add(2, 2);
x was 2 and y was 2
4
addCalls;
2
```

Listing 6-9 Calling the side-effecting function and observing changes in variables

Some functions are only called for their return value, and some are called only for the side effects. As you've just seen, it's also possible to write functions that return a value *and* have a side effect.

Methods are a special kind of function (you'll learn more about the distinction in Chapter XX) that are properties of an object. We've seen methods that are called for their return value, like the `join` method on arrays (see Chapter 3) that returns a value but has no side effects. We've also seen methods called for their side effects, like the `push` method on arrays that modifies the array as a side effect and also returns a value (the new length of the array).

Passing a Function as an Argument

In JavaScript, functions are first-class citizens, which means that they can be used like any other value. For example, you can pass functions to other functions, return them from functions, and store them in variables, arrays, and objects.

We'll illustrate this with the `setTimeout` function, which allows you to delay the calling of another function. It takes two arguments: a function to call, and a time in milliseconds to wait before calling that function.

```
function sayHi() {
    console.log("Hi!");
}
setTimeout(sayHi, 2000);

1
Hi!
```

Listing 6-10 Passing a function as an argument

Here we create a simple function with no arguments, `sayHi`, which just calls `console.log`. We then call `setTimeout`, passing the `sayHi` function and the number `2000`, indicating 2000 milliseconds or two seconds.

`setTimeout` returns a *timeout id*, which is a unique identifier we can use to cancel the timeout with the `clearTimeout` function. After two seconds, the `sayHi` function is called and the string `"Hi!"` is logged to the console.

You might notice that we pass `sayHi` and not `sayHi()`. The function name without parentheses is used to *refer* to the function, while `sayHi()` actually *calls* the function.

We can see this comparison with the JavaScript console:

```
1 sayHi;
  ƒ sayHi() {
      console.log("Hi!");
  }

2 sayHi();
  Hi!
  undefined
```

Listing 6-11 The difference between referring to a function and calling a function

Just executing the plain `sayHi;` 1 shows that `sayHi` refers to a function. However, executing `sayHi();` 2 calls the `sayHi` function, printing the string `"Hi!"` and returning `undefined`.

Function Expressions

There are multiple ways to create functions in JavaScript, and each one has its own strengths. In the previous section you learned about function declarations. An advantage of function declarations is that they're straightforward, and most like how functions are defined in many other languages, like C++ or Python.

In this section you'll learn about *function expressions*, also known as *function literals*—these are code literals that produce a function, just as `123` is a literal that produces the number 123. Function literals are useful when we want to use functions as values, for example, to pass to a function, to return from a function, or to add to an array.

Function Expression Syntax

A function expression looks very similar to a function declaration, with two main differences. First, a function expression doesn't have to have a name, although you can include a name. Function expressions without names are also called *anonymous functions*. Second, a function expression can't be at the start of a line of code, otherwise the JavaScript interpreter thinks it must be a function declaration. There has to be some code before the `function` keyword for JavaScript to understand that you're writing a function expression.

Here we create a function expression and assign it to a variable.

```
1 let sayHola = function () {
    console.log("Hola!");
  };

2 sayHola();
  Hola!
  undefined
```

Listing 6-12 Storing an anonymous function in a variable

Because the `function` keyword appears on the right side of an assignment statement 1, the JavaScript interpreter treats this as a function expression. Also, because this is an assignment statement, we put a semicolon at the end after the closing brace of the function expression.

The function expression defines an anonymous function that logs the string `"Hola!"` and has no parameters. We assign the function to the variable `sayHola`. Now, `sayHola` refers to the function, just like `sayHi` referred to the function declaration.

Because the name `sayHola` is bound to the function, we can call `sayHola` in the same way we would call any other function, by putting a pair of parentheses after the name 2.

In almost all respects, this code would be equivalent to the function declaration:

```
function sayHola() {
  console.log("Hola!");
}
```

Listing 6-13 The same function as 6-12 but defined using a function declaration

Choosing between a function expression and function declaration in mostly a matter of style, but sometimes one or the other is more appropriate. Next we'll see an example where a function declaration can't be used, so we must use a function expression.

Passing a Function Expression as an Argument

In Listing 6-10 we passed the name of a function to `setTimeout`. A more common way to achieve the same task is to pass a function expression as the first argument, instead of the name of an already-defined function. For variety, we'll use `setInterval` this time, which calls a function repeatedly with the given delay between each call. We have to use a function expression in this case because a function declaration can only appear at the start of a line of code.

```
setInterval(function () {
```

```
1 console.log("Beep.");
}, 21000);
2
Beep.
```

Listing 6-14 Passing an anonymous function to `setInterval`

When you execute this code, it'll wait a second before the first `"Beep."` is logged. After that, a number will appear on the left of the console output showing how many times `"Beep."` has been logged, which will increment every second. This is a trick used by the Chrome browser to avoid filling the console with duplicate lines of output. `setInterval` also returns an interval id that we can pass to the `clearInterval` function if we want to stop it.

In Listing 6-14, `setInterval` has two arguments: the first is the function expression 1, the second is the number `1000` 2. Compare this with the earlier call to `setTimeout`: `setTimeout(sayHi, 2000);`. Previously we passed the name of a function, but this time we're passing a function literal.

NOTE If you want this code to stop `"Beep."`-ing, you can either refresh the page, or call the `clearInterval` function, passing the interval id (in our example, this would be `clearInterval(2);`).

Arrow Functions

JavaScript has yet another syntax for defining functions, called *arrow function expressions*, or *arrow functions* for short. Arrow functions are a more compact version of a function expression, and in most cases it's a stylistic decision on whether you use function expressions or arrow functions (we'll discuss an important difference in Chapter XX). Arrow functions can be used anywhere a normal function expression can be used.

Here's how you could make an `add` function using an arrow function:

```
let addArrow = (x, y) => {
  return x + y;
};
```

Listing 6-15 Defining a function using an arrow function expression

An arrow function doesn't use the `function` keyword, and uses an arrow (`=>`) between the argument list and the function body.

Arrow functions can be called just like other functions:

```
addArrow(2, 2);
4
```

Listing 6-16 Calling an arrow function

Listing 6-15 uses *block body* syntax, where the body is placed between braces, but there's an even simpler syntax called *concise body*:

```
let addArrowConcise = (x, y) => x + y;
```

Listing 6-17 Defining an arrow function with concise body syntax

With concise body syntax, the body does not have surrounding braces, the body has to be a single expression, and the `return` keyword is implied (that is, the expression in the body is the return value of the function). In this case, the return value of the function is the expression `x + y`.

The concise syntax is great for simple functions, but as soon as you need multiple statements you'll have to use the block body syntax.

Let's set up our beep interval again using an arrow function:

```
setInterval(() => {
  console.log("Beep.");
}, 1000);
2
Beep.
```

Listing 6-18 Passing an arrow function to `setInterval`

We could also do this using the concise body syntax:

```
setInterval(() => console.log("Beep."), 1000);
```

Listing 6-19 Passing a concise body arrow function to `setInterval`

In both these cases, the function takes zero arguments, so we use an empty parameter list: `()`. For arrow functions with exactly one parameter, the parentheses around the parameter list are optional:

```
let sayBonjour = name => console.log(`Bonjour, ${name}!`);
sayBonjour("Nick");
Bonjour, Nick!
```

Listing 6-20 Defining an arrow function with one parameter

TRY IT OUT

You've now seen three different ways of creating functions. Write the following functions in all three styles:

- A function that takes a number from zero to five and returns the English word for that number, for example, `1` should return `"one"` (hint: you can use an array to define the mapping from number to string).
- A function with no parameters that prints how many times it's been called (hint: define a variable outside of the function to keep track of the number of calls, like we did in the section on side

effects).

- A function that prints the current date and time (tip: you can get the current date and time with `new Date()`).

Rest Parameters

Sometimes you want your function to take a variable number of arguments. We'll make a function that takes someone's name and their favorite colors, however many there are, and prints a string listing those colors. In JavaScript, you can do this with *rest parameters*. A rest parameter is a special parameter that collects the remainder of the arguments when the function is called with multiple arguments. Rest parameters work with any kind of function definition. Here we use them in an arrow function:

```
let myColors = (name, ...favoriteColors) => {
  let colorString = favoriteColors.join(", ");
  console.log(`My name is ${name} and my favorite colors are
${colorString}.`);
};
myColors("Nick", "Blue", "Green", "Orange");
My name is Nick and my favorite colors are Blue, Green,
Orange.
```

Listing 6-21 Rest parameters

A rest parameter looks like an ordinary parameter preceded by three periods, and it always has to be the last parameter. When the function is called, the rest parameter contains however many arguments come after the regular parameter arguments and bundles them into an array.

In Listing 6-21, `name` is a regular parameter, and `favoriteColors` is the rest parameter. When we call the function, the argument `"Nick"` is assigned to the `name` parameter. The remaining arguments `"Blue", "Green", "Orange"` are collected together into a single array and assigned to the `favoriteColors` parameter. Because `favoriteColors` is an array, we can use the `join` method to convert the array into a string, which we then combine into a larger string and print using `console.log`.

Here's another example of using a rest parameter, this time to sum all the numbers provided as arguments:

```
function sum(...numbers) {
  let total = 0;
  for (let number of numbers) {
    total += number;
  }
  return total;
}
sum(1, 2, 3, 4, 5);
```

```
15
```

Listing 6-22 Summing numbers with a rest parameter

We use a function declaration instead of an arrow function, and the only parameter is the rest parameter. Because there are no other parameters, all the arguments (`1, 2, 3, 4, 5`) are collected up into an array and assigned to the `numbers` rest parameter.

Higher-Order Functions

A *higher-order function* is a function that takes another function as an argument, or returns another function as its return value. In this chapter you've already seen two higher-order functions, `setTimeout` and `setInterval`, which both take a function as an argument to execute later.

Functions that are passed as arguments are often called *callbacks*, because they allow the other function to "call back" to the function that was passed in.

Array Methods That Take Callbacks

There are a number of methods defined on arrays that take a callback. Remember: a method is a function that operates on an object. In most cases, the callback is called once for each item in the array. We'll take a look at some examples next.

Finding an item with `find`

The `find` method on arrays finds the first element in the array that matches some predicate. You specify the predicate with a callback function. For example, if you wanted to find the first item in your shopping list with more than 6 characters, you could do the following:

```
let shoppingList = ["Milk", "Sugar", "Bananas", "Ice Cream"];
shoppingList.find(item => item.length > 6);
"Bananas"
```

Listing 6-23 Using the `find` method

The callback function is `item => item.length > 6`. This callback uses two useful syntactic features of arrow functions. First, because our function only has one parameter, we're leaving off the parentheses around the parameter list. Second, we're using the concise body syntax, so we leave off the `return` keyword and the braces around the body. These features let us define the logic for finding the element as compactly as possible, so arrow functions are ideal for callback arguments.

The `find` method runs the callback for each element in the array in turn.. If the callback returns `true`, that item is returned from the `find` method, and

no more elements are checked.

If no item is found that meets the predicate, the `find` method returns `undefined`:

```
shoppingList.find(item => item[0] === 'A');
undefined
```

Listing 6-24 Trying to find an item that doesn't exist

In this case, none of the items have `'A'` as their first character, so `find` returns `undefined`.

Filtering the Elements of an Array with `filter`

The `filter` method returns a new array containing all the elements from the original array that satisfy some predicate. We'll update the previous `find` example, just changing the method name to `filter` to get a list of all items with more than 6 characters.

```
let shoppingList = ["Milk", "Sugar", "Bananas", "Ice Cream"];
shoppingList.filter(item => item.length > 6);
▸ (2) ["Bananas", "Ice Cream"]
```

Listing 6-25 Listing 6-26: Using `filter`

Creating a New Array with `map`

Sometimes you'll want to create a new array that contains transformed versions of all the elements of an original array. For example, say you have an array of objects that represent items in a store, and you want to get an array of just the prices of those items. The `map` method lets you do this, by passing a callback function that defines how to convert each element in the array into a new element.

```
let stockList = [
  { name: "Cheese", price: 3 },
  { name: "Bread", price: 1 },
  { name: "Butter", price: 2 }
];
let prices = stockList.map(item => item.price);
prices;
▸ (3) [3, 1, 2]
```

Listing 6-26 Using `map`

Here, the callback function is `item => item.price`, and it takes an `item` and returns the value of that item's `price` property. The `map` function applies the callback to each item of the original array in turn, and creates a new array with the return value of each call. The original array is unchanged.

Creating Functions That Take Callbacks

There's nothing magic about functions that take callbacks. You define the callback just like any other parameter, and then when you want to call it within the function body, you just add parentheses like with any other function name. Let's illustrate this with a function that just calls its callback twice.

```
function doubler(callback) {
  callback();
  callback();
}
doubler(() => console.log("Hi there!"));
Hi there!
Hi there!
```

Listing 6-27 Creating a function that calls its callback

We pass a function as the `callback` parameter, so we can call it just like any other function, by appending parentheses to its name. Of course, there's nothing stopping you from passing a non-function to `doubler`. If you do, you'll get an error:

```
doubler("hello");
▸Uncaught TypeError: callback is not a function
```

Listing 6-28 Passing a string where a function was expected

You can also pass arguments into your callback. Here we create a function that calls another function some number of times, passing the current number of times into the callback.

```
function callMultipleTimes(times, callback) {
  for (let i = 0; i < times; i++) {
    callback(i);
  }
}
callMultipleTimes(3, time => console.log(`This was time:
${time}`));
This was time: 0
This was time: 1
This was time: 2
```

Listing 6-29 Passing arguments to a callback function

This function takes two arguments: a number of times to call the callback function, and a callback function. The callback is called that many times, passing the current index as an argument.

Functions That Return Functions

As I said at the start of this section, higher-order functions either take functions as arguments, or return functions as their return value. We've seen a

lot of examples of the former, so now let's look at functions that return other functions.

We'll create a function that always adds some suffix to the end of a string. We want to control what that suffix is, so our function will take a suffix and returns a function that takes a string and appends the suffix.

```
function makeAppender(suffix) {
  return function (text) {
    return text + suffix;
  };
}
```

Listing 6-30 Returning a function from a function

There are two `return` keywords here. The first is used by the `makeAppender` function to return the anonymous function. The second `return` keyword is inside the anonymous function, and returns a value when *that* function is called. To be able to call the inner function, we first have to call the outer function.

```
let exciting = makeAppender("!!!");
exciting("Hello");
"Hello!!!"
```

Listing 6-31 Calling a function returned from another function

Calling `makeAppender("!!!")` returns a new function, which is assigned to the `exciting` variable. The `exciting` variable now contains the function expression that was returned from `makeAppender`, which takes a string. When we call `exciting("Hello")` we get the string `"Hello!!!"`, which is the result of adding the two strings together.

One important thing to note is that the function we returned from `makeAppender` remembers the value of `suffix`. Even though the call to `makeAppender` has completed, the inner function it returned is able to hold onto a parameter that was in scope when it was defined. Functions that hold onto variables and parameters from their enclosing scopes are known as *closures*, because they "close over" their environments.

WHAT EXACTLY IS SCOPE?

All bindings in JavaScript have *scope*, which is the area of code in which they are accessible. For example, if you declare a variable with `let` inside a `while` loop, that variable can't be used outside of the `while` loop, but if you define the variable outside of the loop, it can be used inside the loop. Each nesting of control structure or function definition adds a new layer of scope. Bindings defined in the outer layers can be accessed by the inner layers, but not vice versa.

For example, in the following listing, the body of the `while` loop is able to access a variable defined outside of the loop, but code outside of the loop can't access a variable defined inside the loop.

```
let outerVariable = 10;

while (outerVariable > 0) {
  let innerVariable = "hello!";
  outerVariable--;
}

console.log(innerVariable);
▸Uncaught ReferenceError: innerVariable is not defined
```

Listing 6-32 Trying to access an out-of-scope variable

The body of the while loop can access `outerVariable` and increment it down to `0`, since it was declared outside of the loop. However, trying to access `innerVariable` outside the loop results in an error. `innerVariable` only has scope within the `while` loop.

In my definition of `makeAppender` I chose to return a function expression, because it shows more clearly that there are two distinct functions. Here's how it would look with a concise arrow function:

```
function makeAppenderConcise(suffix) {
  return text => text + suffix;
}
```

Listing 6-33 Returning an arrow function from a function

TRY IT OUT

Write a function called `makeWrapper` that takes a prefix and a suffix, and returns a new function that takes a string, and returns the string surrounded by the prefix and suffix. For example, you could enter `let bracketWrapper = makeWrapper("[", "]");` and then call `bracketWrapper("Bracket Me!");` to get the string `"[Bracket Me!]"`.

Conclusion

In this chapter you learned how to create and work with functions. You saw the four main ways to create functions: function declarations, function expressions, block body arrow functions, and concise body arrow functions.

In the next chapter we'll look at classes, which tie together objects and functions and give us new ways of grouping code and data.

9

EVENT-BASED PROGRAMMING

When a user clicks a button, scrolls, or simply moves the mouse within a web page, that action creates an *event*. An event is the browser's way of signaling that an action happened in the DOM. Events allow us to create interactive web applications that respond to the user's actions. We do this by writing *handlers* for specific events: functions that are called when an event occurs. Using event handlers, we can change the color of an element when the user clicks on it, move an element around the screen when the user presses a certain key, and much more.

In this chapter you'll learn how to write event handlers to respond to some common DOM events. In this way, you'll add interactivity to your web pages.

Event Handlers

Events are how the browser tells JavaScript that something has occurred in the DOM. It's almost as if every time the mouse moves over the window, or a key is pressed, the browser is shouting "Hey, the mouse moved! A key was pressed!" These shouts happen all the time, but your JavaScript code can only respond to them if you explicitly tell it to listen for them. You do this by writing a JavaScript *event handler* that will perform some action when a specific type of event occurs.

An event handler is a function triggered by a specific event type on a specific element. For example, you could attach a handler to a specific `h1` element that handles clicks on that element. Let's try that out! We'll create a simple web page with a heading and an event handler that logs a message to the console when the heading is clicked.

First, you'll need an HTML file. Create a new directory called *chapter9* and make a new file in that directory call *index.html*. Enter the content shown in Listing 9-1.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Event Handlers</title>
  </head>
  <body>
    <h1 id="main-heading">Hello <em>World!</em></h1>
    <script src="script.js"></script>
  </body>
</html>
```

Listing 9-1 *index.html*

As usual, our HTML file has a single `html` element containing a `head` with some metadata and a `body` with some content. Specifically, the `body` contains an `h1` element with the `id` of `"main-heading"`, and part of the heading text is wrapped in an `em` element (short for *emphasis*), which by default italicizes that portion of the text. The HTML file also has a `script` element with a link to the file *script.js*. In a moment, that's where we'll write the code for our event handler.

Overall, this file is very similar to the HTML we created in Chapter 8, with one important difference: the `script` element is inside the `body` element, not the `head` element. This is a bit of a cheat to get around a problem with how web browsers read web pages. As described in the previous chapter, the browser builds a model of the page called the DOM. It builds the DOM incrementally by reading through the HTML file from top to bottom. Any time the browser reaches a `script` element, it executes the whole script before continuing. That means that if we had our `script` element in the `head`, and looked up the `h1` element in that script, the `h1` element wouldn't be in the DOM yet! By placing the `script` element at the end of the body we can be sure that all the page content has been loaded into the DOM *before* we run our JavaScript.

Now create a file called *script.js* in the same directory as the HTML code, and

enter the script shown in Listing 9-2. This script adds an event handler for when the user clicks on the `h1` element.

```
let heading = document.querySelector("#main-heading");

heading.addEventListener("click", () => {
  console.log("You clicked the heading!");
});
```

Listing 9-2 *script.js*

Using the DOM API's `querySelector` method, we get the element with the `id` of `main-heading` and save it as the variable `heading`. You may recall from Chapter 8 that this method returns the first element to match the selector. In our case, there's only one `main-heading` element, so we know this method will select the element we want. We then use the `addEventListener` method on the `heading` element to attach an event handler to that element. `addEventListener` tells JavaScript to watch for a particular event to happen on the element, and to execute some function when it does.

NOTE Although the DOM API uses the term *listener*, the term *handler* is more commonly used to describe the function that reacts to an event.

The `addEventListener` method has two required arguments. The first is the event type. This is a string representing the type of event to respond to, for example `"click"` (for mouse clicks), `"keydown"` (for keyboard key presses), or `"scroll"` (for the window scrolling). We've specified `"click"`. The second argument is the function to execute when the specified event happens. This function is the event handler. It will be called any time the event happens on the element `addEventListener` was called on. In this case, the function, which logs a message to the console, will be called any time someone clicks on the `heading` element.

NOTE As explained in Chapter 6, when a function is passed as an argument to another function, it's known as a *callback function*. All event handlers are callback functions, since they're passed as an argument to the `addEventListener` method.

Open *index.html* in your browser and open the console. When you click the heading you should see the message `"You clicked the heading!"` printed to the console. Congratulations: you've made your first interactive web page!

TRY IT OUT

Add a `p` element to the page and attach a `click` event handler to it.

Event Bubbling

When an event is triggered on an element, it also gets triggered on all the element's ancestors (that is, the parent of the element, the parent's parent, and so on). For instance, when you clicked on the `h1` element in the previous example, you were also technically clicking on the `body` element that contains the `h1` element. Therefore, a separate handler attached to the `body` element would also receive the `click` event. This progression of events from children to ancestors is known as *event bubbling*. Bubbling matches how we might think about interacting with the document: if you click on some text in a box, you're also clicking on the box.

Let's harness bubbling by adding event handlers to the `em` and `body` elements in *index.html*. Like our `h1` event handler, these new handlers will log a message to the console when the element is clicked. Since `em` is a child of `h1` and `h1` is a child of `body`, a single click on `em` should trigger the handlers attached to all three elements.

Add the code in Listing 9-3 to the end of *script.js*:

```
document.querySelector('em').addEventListener("click", () => {
  console.log('You clicked the em element!');
});

document.querySelector('body').addEventListener("click", () => {
  console.log('You clicked the body element!');
});
```

Listing 9-3 Adding more handlers to *script.js*

This snippet adds two handlers, one to the `em` element and one to the `body` element, but we do it slightly differently from how we created the `main-heading` handler in Listing 9-2. Instead of saving each element to a variable, we just call `addEventListener` directly on the result of the `document.querySelector` method. This technique of calling a method directly on the return value of another method is known as *method chaining*: we chain multiple method calls together, so that the result of the first link in the chain is used as the object for the next method call. I used the longer form technique for Listing 9-2 because it makes it more explicit that `addEventListener` is being called on an element, but the chaining technique is often preferred because of its terseness.

Reload *index.html* and open the console. When you click on the word *World!* you should see the following output:

```
You clicked the em element!
You clicked the heading!
You clicked the body element!
```

Listing 9-4 The output from *index.html*

When you click on the `em` element, the handler function on the `em` element is the first to get called. After that, the handler function on the `h1` element is called, followed by the one on the `body` element. This is because the event "bubbles up"

through the DOM, from the innermost element to the outermost element. If you click the non-italic part of the heading, you'll see just the `main-heading` and `body` handlers triggered.

Event Delegation

One of the more common uses for event bubbling is *event delegation*, a technique where you use a single handler to respond to events on multiple child or other descendant elements. For example, imagine you have a list of words, where each list item is a separate HTML element, and you want to handle clicks on each item in the same way. By adding a single handler to the list items' parent element, you can catch events on each item with only a few lines of code.

To illustrate event delegation, we'll write a simple application that builds up and displays a sentence based on words that you click on from a list. First we'll update our HTML file to include a list of words and an empty `p` element that we'll populate dynamically with the words of your choice. Then we'll write the necessary event handler with JavaScript to take clicked words and add them to the `p` element for display. Finally, we'll add some CSS rules to make the application easier to interact with.

There are two types of list in HTML: ordered (numbered) lists and unordered (bulleted) lists. We'll use an unordered list, which is created with the `ul` (unordered list) element. Each individual item in the list is wrapped in an `li` (list item) element. Therefore, the event resulting from a click on any `li` element will bubble up to the parent `ul` element.

Update *index.html* as shown in Listing 9-5.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Event Handlers</title>
  </head>
  <body>
    <h1 id="main-heading">Hello <em>World!</em></h1>

    <ul id="word-list">
      <li>The</li>
      <li>Dog</li>
      <li>Cat</li>
      <li>Is</li>
      <li>Was</li>
      <li>And</li>
      <li>Hungry</li>
      <li>Green</li>
    </ul>

    <p id="sentence"></p>

    <script src="script.js"></script>
  </body>
```

```
</html>
```

Listing 9-5 Adding a list to *index.html*

This adds an unordered list of words to the document, as well as an empty `p` element, which we'll be modifying with JavaScript.

Next we'll write an event handler for the `ul` element to handle clicks on any of the list items. The handler will take the word that was clicked on and add it to the `p` element, allowing you to build up a sentence one word at a time. Delete all the code in *script.js* and replace it with Listing 9-6.

```
let wordList = document.querySelector("#word-list");
let sentence = document.querySelector("#sentence");

wordList.addEventListener("click", 1event => {
2 let word = event.target.textContent;
  sentence.textContent += word;
  sentence.textContent += " ";
});
```

Listing 9-6 Delegating events

First we look up the two elements we care about using `document.querySelector`: the `ul` with the word list, and the empty `p` element. Next, we add a `click` handler to the `ul` element. This example is a little different to our previous handlers, because the callback function has a parameter, which we're calling `event` 1. If we give a handler function a single parameter (in this case, `event`, but the name isn't important), the parameter represents an object through which the DOM API passes information about the event that just happened. That information, which includes the element that was clicked on, the element's text content, and so on, then becomes available for use within the callback function.

In our example, we use the `event` object to find out what word was clicked on and store it in the variable `word` 2. To do this, we find out which specific element was clicked on with the `event` object's `target` property. When you click on one of the `li` elements, `event.target` will be the `li` element you clicked on, not the `ul` (the `ul` element is available with the `currentTarget` property). Then we use the `textContent` property, which returns the text of that element. Putting it together, if you clicked on the first `li` element, then `event.target.textContent` would return the string `"The"`, and that string would become the value of the variable `word`.

Now that we have the word the user clicked on, we can add it to the sentence. We use the `+=` operator to append the word to the text content of the `sentence` element. You may recall that `sentence.textContent += word;` essentially converts to `sentence.textContent = sentence.textContent + word;`. In other words, we're taking the `sentence` element's existing text content, adding the string stored in `word` to the end, and then re-assigning that text to the element's text content. Then, after adding the word to the sentence, we use

the same `+=` trick to append a space to the end of the sentence in preparation for the next word that gets added.

Open *index.html* again in your browser. You should see the list of words. You won't see the empty `p` element because it doesn't have any content yet. As you click on words from the list, you should see them being added to the p element.

To finish off our application, we need to add a small amount of CSS. JavaScript and CSS often go hand-in-hand because the styling can give helpful tips to the user that certain elements are interactive. In this case we'll add two hints via CSS: we'll modify the list items to change the mouse pointer to a finger when it hovers over them, so they look more "clickable," and we'll give the element that's currently under the mouse pointer an underline so it's easier to tell which word you're about to click on (which isn't always obvious when the mouse is in the vertical space between words).

Create a new CSS file called *style.css* and add the two CSS declarations shown in Listing 9-7.

```
li {
  cursor: pointer;
}

li:hover {
  text-decoration: underline;
}
```

Listing 9-7 *style.css*

To change the cursor for `li` elements, we use `cursor: pointer`. This changes the cursor from the default arrow to a hand with a finger when it is over the `li` element, as happens when you hover over a link on a web page. The `li:hover` selector uses the `:hover` pseudo-class, which only applies when the element is hovered (that is, when the cursor is over the element). A *pseudo-class* is a kind of selector that only applies when an element is in a certain state. So `li:hover` matches any `li` that the mouse is currently hovering over. When that's the case, we use `text-decoration: underline` to underline the text of that `li` element.

To include this CSS on the page, add a `link` element to the `head` of the *index.html* file, as shown in Listing 9-8.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Event Handlers</title>
    <link rel="stylesheet" href="style.css">
  </head>
```

Listing 9-8 Including *style.css* in *index.html*

Now when you hover over one of the list items, the cursor will change, and the currently hovered word will be underlined, as shown in Figure 9-1.

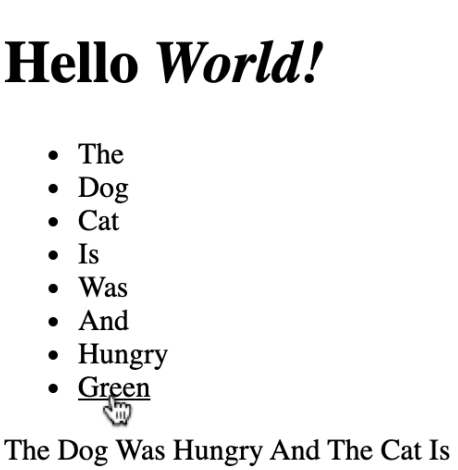


Figure 9-1 Using CSS to give hints to users

With that styling, our simple sentence-building application is complete! Event delegation simplified the JavaScript we wrote by letting us attach a single event handler, rather than using a separate handler for each list item.

Mouse Movement Events

The DOM produces events when the mouse moves, with the event name `mousemove`. These `mousemove` events are triggered on an element while the mouse moves over that element, and we can listen for them with the `addEventListener` method, just as we did for mouse clicks. Let's set up a simple `mousemove` handler to see it in action. The handler will log the mouse's position to the console as the mouse moves around the web page.

Still working within your *chapter9* project folder, add the code in Listing 9-9 to the end of *script.js*.

```
document.querySelector("html").addEventListener("mousemove", e =>
{
  console.log(`mousemove x: ${e.clientX}, y: ${e.clientY}`)
});
```

Listing 9-9 A `mousemove` event handler

In this listing we add a `mousemove` event handler to the `html` element. Since the `html` element encompasses the entire web page, this handler will respond to movements of the mouse anywhere in the browser window. The handler logs a message to the console, including the `clientX` and `clientY` properties of the event, which tell us the x and y coordinates of the mouse relative to the browser window. In this example I'm using the shorter name `e` for the event parameter, rather than `event`, as in Listing 9-6. Remember, the name of the parameter doesn't matter—if the event handler callback function has a single parameter, that parameter will carry information about the event.

Refresh *index.html* in your browser and you should see messages logged to the console, as in Listing 9-10.

```
mousemove x: 434, y: 47
mousemove x: 429, y: 47
mousemove x: 425, y: 48
mousemove x: 421, y: 51
mousemove x: 416, y: 51
mousemove x: 413, y: 54
mousemove x: 408, y: 55
```

Listing 9-10 Sample output from Listing 9-9

There are two important things to note as you watch the console. First, the co-ordinates start at 0 in the top left corner of the browser window, increasing as you go across and down. The x coordinate increases as you move right, and the y coordinate increases as you move down. This follows the standard convention for computer graphics.

Second, there are "gaps" in locations that the mouse appears to jump over. This is because `mousemove` events aren't triggered continuously, but some limited number of times per second. The exact number depends on the mouse, the browser, and the computer, but it tends to be in the low hundreds. Therefore, if you move the mouse fast enough, there are locations on the screen where the mouse seems to skip over, because the events weren't triggering fast enough.

Now that you've seen `mousemove` events in action, we can try to do something slightly more interesting with them. In this next example, we'll make a box move around the page, following your cursor. To do that we'll need to modify our HTML, CSS, and JavaScript files. The HTML change is simple. Add the highlighted line in Listing 9-11 to *index.html*.

```
    <p id="sentence"></p>

    <div id="box"></div>

    <script src="script.js"></script>
  </body>

</html>
```

Listing 9-11 Adding a `div` to *index.html*

Here we're using a new HTML element called `div`, short for *content division*. It will become the moveable box on our page. The `div` element is HTML's generic container element. This means that it is an element that can contain other elements, but by default has no appearance, and no specific *meaning* (unlike `ul` which means a list, or `h1` which means a heading). We'll use CSS to give the div element an appearance next. Add Listing 9-12 to the end of *style.css*.

```
#box {
  position: fixed;
  left: 0px;
  top: 0px;
```

```
  width: 10px;
  height: 10px;
  background-color: hotpink;
}
```

Listing 9-12 Styling the `div` with CSS

Here we're using `#box` to select the element with the id `box`. There are a number of declarations within this ruleset. The first, `position: fixed`, tells the browser to put this element at a position specified next by the `left` and `top` declarations. We indicate `0px` for both, which tells the browser to put the element at the very top-left corner of the browser *viewport*, the part of the browser that shows the content. We specify the `width` and `height` of the element to be 10 pixels each. Finally, we give our 10×10 box a background color that's sure to jump out: hot pink.

Refresh the page now, and you'll see a small pink square appear at the top-left corner. Now it's time to write an event handler so you can move the square with your mouse. Modify *script.js* as shown in Listing 9-13.

```
let wordList = document.querySelector("#word-list");
let sentence = document.querySelector("#sentence");

wordList.addEventListener("click", event => {
  let word = event.target.textContent;
  sentence.textContent += word;
  sentence.textContent += " ";
});

let box = document.querySelector("#box");

document.querySelector("html").addEventListener("mousemove", e =>
{
  box.style.left = e.clientX + "px";
  box.style.top = e.clientY + "px";
});
```

Listing 9-13 Moving the `div` with JavaScript

The first addition here is to find the box using `document.querySelector` and save a reference to the element in the variable `box`. Next, we modify the `mousemove` event handler we wrote earlier. In order to move the box around, we're modifying its `style` property, which is an object representing the CSS applied to the element. For example, setting a value for `box.style.left` has the same effect as updating the value of `left` in the CSS file. In our handler, we set both the `left` and `top` values using the current position of the mouse.

As mentioned in Chapter 8, numeric values in CSS require a unit. We can't just assign a number, like `box.style.left = 10`. Instead, we have to provide a string including the units, like `box.style.left = "10px"`. This is why we include `+ "px"` at the end of each statement in our event handler. If `e.clientX` is `50` then `e.clientX + "px"` will give the string `"50px"`,

which gets assigned to the `box.style.left` property, updating the left position of the box. As the mouse moves, this handler will be called with `e.clientX` and `e.clientY` set to the current position of the mouse, and so the pink box will move around as your mouse moves. Refresh the page and give it a try!

TRY IT OUT

What happens if you modify the values that set the box's position? For example, what would happen if you multiplied `e.clientX` by 2, or added 50 to `e.clientY`?

Keyboard Events

Keyboard events are triggered when keys are pressed on the keyboard. We'll focus on one keyboard event called `keydown`, which is triggered when a key is pressed down. (There's a corresponding event called `keyup` which is triggered when a key is released.)

We'll add a handler to our web page that simply logs `keydown` events to the console as they happen. Add Listing 9-14 to the end of *script.js*.

```
document.querySelector("html").addEventListener("keydown", e => {
  console.log(e);
});
```

Listing 9-14 Logging `keydown` events

As in Listing 9-13, we're adding an event handler to the `html` element, meaning it will apply to the entire web page, but this time we're handling the `keydown` event. This event is triggered whenever you press down a key on your keyboard. Our handler logs e to the console, meaning the entire `event` object will be logged when a key is pressed.

Try reloading the page to see the handler in action. You'll need to open the console, then click inside the document to give it *focus*. This just means that the key presses will get sent to your web page, instead of to the console text input. As long as everything's set up correctly, you should see events being logged to the console, as in Listing 9-15.

```
▸KeyboardEvent {isTrusted: true, key: "h", code: "KeyH",
location: 0, ctrlKey: false, …}
▸KeyboardEvent {isTrusted: true, key: "e", code: "KeyE",
location: 0, ctrlKey: false, …}
▸KeyboardEvent {isTrusted: true, key: "l", code: "KeyL",
location: 0, ctrlKey: false, …}
▸KeyboardEvent {isTrusted: true, key: "l", code: "KeyL",
location: 0, ctrlKey: false, …}
▸KeyboardEvent {isTrusted: true, key: "o", code: "KeyO",
location: 0, ctrlKey: false, …}
```

Listing 9-15 Some logged `keydown` events generated by typing the word *hello*

Click on the arrow next to one of the events to see the properties each event has. As you'll see, there are a lot, but we mostly care about which key was pressed, which we can find with the `key` property. We'll use this information next to move the pink box around using the keyboard.

In order to move the box around, we'll create two new variables to keep track of its x and y positions, and then update those variables with an event handler when specific keys are pressed. Update *script.js* as shown in Listing 9-16. (Note that the `mousemove` handler has been removed in this listing.)

```
let wordList = document.querySelector("#word-list");
let sentence = document.querySelector("#sentence");

wordList.addEventListener("click", event => {
  let word = event.target.textContent;
  sentence.textContent += word;
  sentence.textContent += " ";
});

let box = document.querySelector("#box");

let currentX = 0
let currentY = 0

document.querySelector("html").addEventListener("keydown", e => {
  if (e.key == "w") {
    currentY -= 5;
  } else if (e.key == "a") {
    currentX -= 5;
  } else if (e.key == "s") {
    currentY += 5;
  } else if (e.key == "d") {
    currentX += 5;
  }

  box.style.left = currentX + "px";
  box.style.top = currentY + "px";
});
```

Listing 9-16 Using `keydown` events to move the box

We create two variables called `currentX` and `currentY` to store the location of the box. Then we modify our `keydown` handler to include an `if...else` statement that checks to see if the event's `key` property matches any of `"w"`, `"a"`, `"s"`, or `"d"`. If so, that indicates one of those four keys has been pressed (I'm using these keys as they're typically used for movement in games). Depending on which key has been pressed, we add or subtract `5` to `currentX` or `currentY`, corresponding to the box moving 5 pixels up, down, left, or right. After we update the variables, we update the style of the box with `box.style.left` and `box.style.top` as we did in Listing 9-13. This time, however, we use the updated value of `currentX` and `currentY` to change the CSS.

When you reload the page, try holding down the s or d keys to make the box move down or right. You should notice that holding the keys down results in the box continuing to move, as the keyboard sends repeating `keydown` events. This is the normal behavior of a computer keyboard when you hold down a key. The exact repeat speed is controlled by your operating system.

TRY IT OUT

The event object for a `keydown` has a Boolean property called `repeat` that tells you if the current `keydown` was generated as a repeat from holding down the key. How could you modify the `keydown` handler to only respond to actual key presses, and not automatic repeats?

Hint: One approach is to use the `return` keyword to return early from the handler function.

Conclusion

In this chapter you learned the basics of DOM events and event handling. DOM events are how the browser tells your code that something happened on your page. You can respond to these events with event handlers, JavaScript functions that are executed when a certain event happens to a certain DOM element. Event handlers allow you to create web pages that respond interactively to the user's actions. In particular, you saw how to write event handlers triggered by clicks, mouse movements, and key presses. You'll learn about other events in subsequent chapters, such as `change`, for when a user changes the value of a form element, and `submit`, for when a user submits a form.

In the next chapter we'll learn about the `canvas` element, which gives us a way to draw pictures and animations using JavaScript.

10

THE CANVAS ELEMENT

One of the more interactive elements in HTML is the `canvas` element. This element acts like a painter's canvas: it provides space for you to draw images within the browser window using JavaScript. What's more, by repeatedly erasing old images and drawing new ones, you can create animations on the canvas. In this sense, the `canvas` element is more like the screen at a movie theater, where the image is updated many times every second to create the appearance of motion.

In this chapter you'll learn how to create `canvas` elements and how to use the Canvas API, which gives you a way to manipulate the canvas via JavaScript. You'll write JavaScript to draw static images to the canvas. Then

you'll build a simple interactive drawing application. Finally, you'll learn the basics of creating 2D animations on the canvas.

Creating a Canvas

To include a `canvas` element on a web page, you create it as part of the page's *index.html* file. All you need are opening and closing HTML tags, like this: `<canvas></canvas>`. The `canvas` element doesn't have any required attributes. However, it's a good idea to give the canvas an `id`, so you can easily access it using JavaScript. It's also common to set the element's `width` and `height` attributes so you can establish the size of the canvas.

Images that appear in the canvas are generated using JavaScript, not HTML. Any HTML between the opening and closing `canvas` tags will only appear if the browser doesn't support the `canvas` element, and so can be used as a fallback for older browsers or text-only browsers.

Let's create an HTML file that includes a `canvas` element and a `script` element linking to a JavaScript file, where we'll write code to generate images on the canvas. We'll use the same HTML file throughout the chapter to draw different kinds of images. Create a new directory called *chapter10*, and make a new file in that directory called *index.html*. Enter the content shown in Listing 10-1.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Canvas</title>
  </head>
  <body>
    <canvas id="canvas" width="300" height="300"></canvas>
    <script src="script.js"></script>
  </body>
</html>
```

Listing 10-1 *index.html*

This is our familiar HTML template, similar to the *index.html* files we've created in previous chapters, but with a `canvas` element instead of an `h1` element. The `width` and `height` attributes specify the size of the canvas in pixels. By default, the canvas is transparent, so you won't actually see anything just yet if you load the page.

Making Static Drawings

Now that we have a `canvas` element, we're ready to draw on it using JavaScript and the Canvas API. We'll start by drawing a solid rectangle. Then we'll look at how to create other static drawings. Create a new file called

script.js in the *chapter10* directory, and enter the code shown in Listing 10-2.

```
let canvas = document.querySelector("#canvas");
let ctx = canvas.getContext("2d");
ctx.fillStyle = "blue";
ctx.fillRect(10, 10, 200, 100);
```

Listing 10-2 *script.js*

First we get a reference to the `canvas` element using the `document.querySelector` method. The `canvas` element has a method called `getContext` which we use the get the canvas's *drawing context*. The drawing context is an object that provides the entire Canvas API as a set of methods and properties (like `fillRect` and `fillStyle`, respectively, both used in Listing 10-2). These methods and properties are what we'll use to draw images on the canvas. In this case, we pass the string `"2d"` to the `getContext` method to request the two-dimensional drawing context.

NOTE You can make 3D graphics with the canvas by passing the string `"webgl"` to the `getContext` method instead of `"2d"`, but that is much more complicated than 2D graphics and deserves its own book!

Next we tell the drawing context that we want the fill color for new elements to be blue, using the `fillStyle` property. Finally, we draw a filled rectangle using the current fill color with the `fillRect` method. This method takes four arguments: the x and y coordinates of the top-left corner of the rectangle, and the width and height of the rectangle in pixels. The coordinates work in the same way as coordinates for the whole browser window: x values increase moving to the right along the canvas, and y values increase moving downwards, with `(0, 0)` representing the top-left corner of the canvas.

Open *index.html* in your browser and you should see a solid blue rectangle, as shown in Figure 10-1.

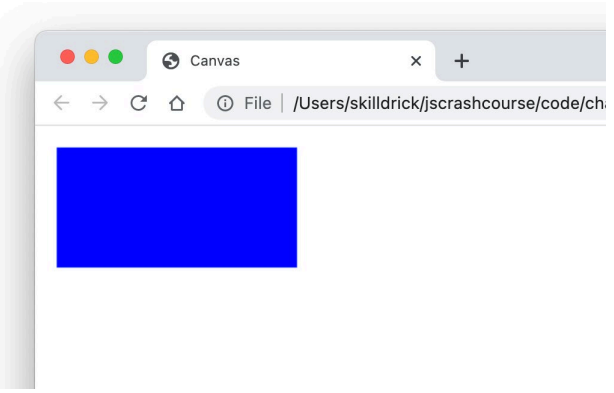


Figure 10-1 The blue rectangle

Any subsequent calls to `fillRect` will use the same `fillStyle` (until you set a new `fillStyle`, that is). You can confirm this by drawing some more rectangles to the canvas.

TRY IT OUT

1. Draw a 100 pixel square starting at (0, 0).
2. We set the canvas to be 300 pixels wide by 300 pixels tall. What happens if you draw a rectangle that's bigger than the canvas?

Drawing Outlined Rectangles

As well as `fillRect` for making a rectangle filled with a color, the Canvas API provides the `strokeRect` method for outlining (*stroking*) a rectangle. To try it out, modify *script.js* as shown in Listing 10-3.

```
let canvas = document.querySelector("#canvas");
let ctx = canvas.getContext("2d");
ctx.lineWidth = 2;
ctx.strokeStyle = "red";
ctx.strokeRect(10, 10, 200, 100);
```

Listing 10-3 Using strokeRect to outline a rectangle

First we specify the width of the outline with the `lineWidth` property, setting it to 2 pixels wide. Then we use `strokeStyle` and `strokeRect` rather than `fillStyle` and `fillRect` to create an outlined rectangle with no fill color. The `strokeRect` method takes the same arguments as `fillRect`: the x and y coordinates for the top left corner, and the width and height of the rectangle.

When you reload *index.html* you should see the rectangle is now outlined in red, with no fill, as show in Figure 10-2.

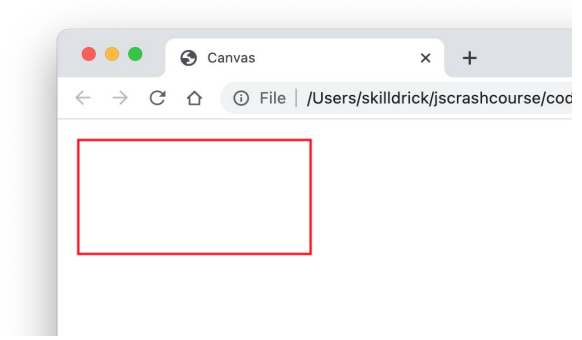


Figure 10-2 A red outlined rectangle

When you set styles on the drawing context, such as the line width or line color, those settings only apply to subsequent additions to the canvas. That is, they don't retroactively affect anything that's already been drawn. In this sense, the canvas really is very much like a physical canvas, where the current style is like the color of paint and type of brush you're currently using. To demonstrate, we'll draw several rectangles with different colors. Add the code in Listing 10-4 to the end of *script.js*, after the code for drawing the red rectangle.

```
ctx.strokeStyle = "orange";
ctx.strokeRect(20, 20, 180, 80);

ctx.strokeStyle = "yellow";
ctx.strokeRect(30, 30, 160, 60);

ctx.strokeStyle = "green";
ctx.strokeRect(40, 40, 140, 40);

ctx.strokeStyle = "blue";
ctx.strokeRect(50, 50, 120, 20);
```

Listing 10-4 Drawing more rectangles

This code draws a series of nested rectangles, each offset by 10 pixels from the previous one, and each 20 pixels smaller than the previous one. Before we draw each successive rectangle, we change the color of the outline by updating the `strokeStyle` property.

Refresh *index.html* and you should see something like the image in Figure 10-3.

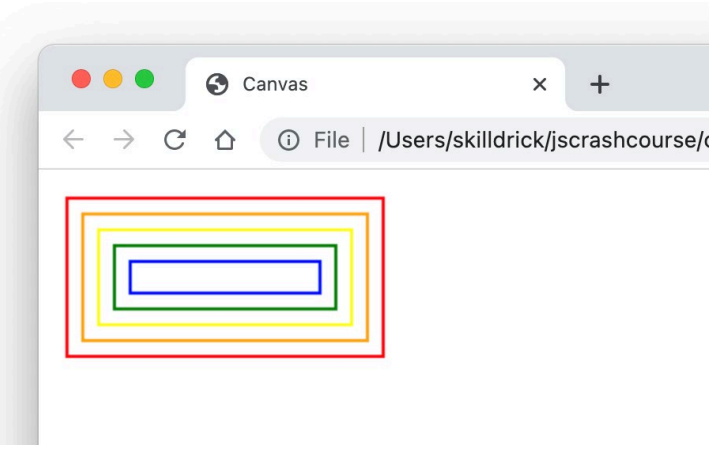


Figure 10-3 Concentric rectangles

Each rectangle is a different color, indicating that the style changes didn't impact anything that had already been drawn.

Drawing Other Shapes Using Paths

All other shapes besides rectangles are drawn on the canvas as *paths*. A path is a series of points connected by straight or curved lines, and then either stroked with an outline or filled in with a color. As an example, we'll draw a path between three different points and then fill it in to make a red triangle. Replace the contents of *script.js* with the code in Listing 10-5.

```
let canvas = document.querySelector("#canvas");
let ctx = canvas.getContext("2d");
ctx.fillStyle = "red";
ctx.beginPath();
ctx.moveTo(100, 100);
ctx.lineTo(150, 15);
ctx.lineTo(200, 100);
ctx.lineTo(100, 100);
ctx.fill();
```

Listing 10-5 Drawing a triangle with path methods

Drawing a path takes three steps. First, you declare that you want to start drawing a new path, with `beginPath()`. Then, you use various methods to define where the path will be. Finally, you use `fill()` or `stroke()` to fill or stroke the path.

In this case, we use two different methods to define the path: `moveTo` and `lineTo`. The `moveTo` method moves an imaginary pen to a particular point on the canvas defined by x and y coordinates, without drawing a line. We use this method to define the starting point of our path, `(100, 100)`, which will be the bottom left corner of the triangle. The `lineTo` method does the same as `moveTo`, but it draws a line as it moves . Thus, `lineTo(150, 15)` draws a line from `(100, 100)` to `(150, 15)`, and so on. Finally we fill the shape with the `fill()` method. When you refresh the page you should see a red triangle, as shown in Figure 10-4.

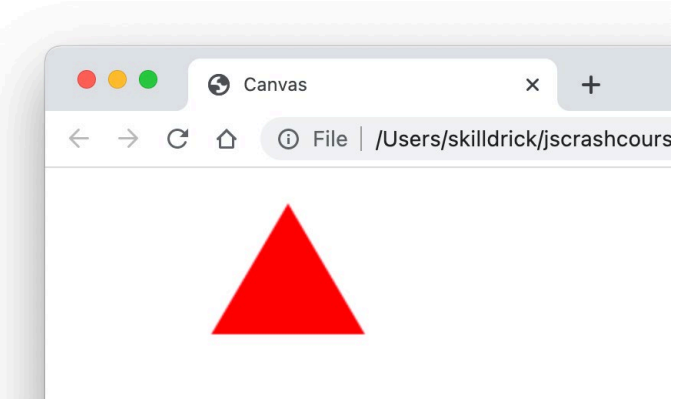


Figure 10-4 Drawing a filled triangle

Drawing circles follows a similar pattern, but uses a method called `arc` instead of `moveTo` and `lineTo`. The `arc` method draws an *arc*, a section of the circumference of a circle. You can produce any length of arc with the `arc` method, but here we'll use it to produce an entire circle.

Update *script.js* with the code in Listing 10-6. This code keeps the first and third steps of the path drawing code but replaces the second step with the code for drawing a circle rather than a triangle.

```
let canvas = document.querySelector("#canvas");
let ctx = canvas.getContext("2d");
ctx.fillStyle = "red";
ctx.beginPath();
ctx.arc(150, 100, 50, 0, Math.PI * 2, false);
ctx.fill();
```

Listing 10-6 Drawing a circle with path methods

The `arc` method takes a whopping six arguments! The first two are the x and y coordinates of the center of the circle. In this case we're centering the circle at the coordinates `(150, 100)`. The third argument is the circle's radius in pixels, which is set here to `50`. The next two arguments give the starting and ending angle of the arc in radians. Because we want a full circle, we provide `0` for the starting angle and `2π` for the ending angle. The final argument specifies whether the arc should be drawn clockwise (`false`) or counterclockwise (`true`) from the starting angle to the ending angle. In this case, we pick clockwise, but since we're drawing a full circle, the direction is irrelevant.

NOTE Radians are a way of measuring angles. In degrees, a full revolution of circle goes from 0 to 360. In radians, a revolution goes from 0 to 2π.

When you refresh the page now, you should see a red circle, as show in Figure 10-5.

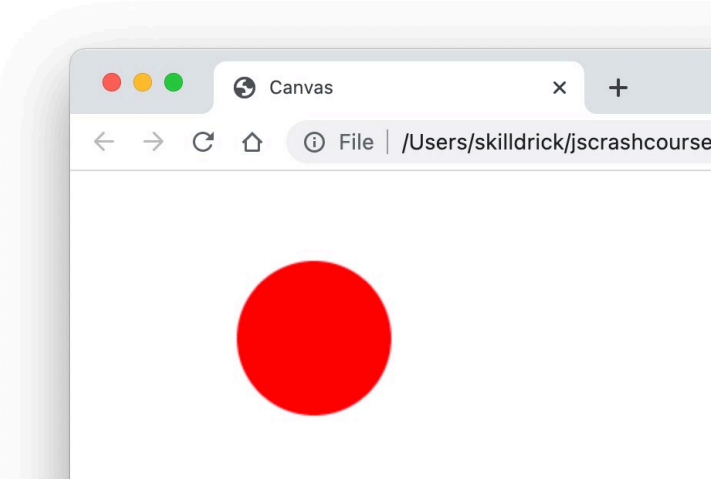


Figure 10-5 Drawing a filled circle

You can use the same technique to draw a stroked circle instead by using the `stroke` method rather than the `fill` method. What's more, you can make compound shapes like rounded rectangles by combining calls to the `line` and `arc` methods. The Canvas API also allows for drawing more complex curves with the `quadraticCurveTo` and `bezierCurveTo` methods. Search the Mozilla Developer Network (MDN) Web Docs for more details about these other methods.

Interacting with the Canvas

The canvas gets a lot more interesting when the user can interact with it. The `canvas` element itself doesn't have any notion of interactivity built in, but we can add that interactivity with JavaScript. To make the canvas interactive, we write event handlers that listen for certain user actions and trigger Canvas API methods that update the canvas in response.

In this section we'll build a very basic drawing application using a canvas with a `click` handler. The handler will listen for clicks on the canvas, and call a method that draws a circle at the position where the click happened. We'll also create a slider so the user can set the opacity of the circles, and a button to clear the canvas.

First, let's add the necessary HTML elements to create a slider and a button. Make the modifications shown in Listing 10-7 to *index.html*.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Canvas</title>
  </head>
```

```
  <body>
    <canvas id="canvas" width="300" height="300"></canvas>
    <div>
      <button id="clear">Clear</button>
      <input id="opacity" type="range" min="0" max="1"
value="1" step="0.1">
      <label for="opacity">Opacity</label>
    </div>
    <script src="script.js"></script>
  </body>
</html>
```

Listing 10-7 Adding some additional elements to index.html

Here we're adding a new `div` element containing three other HTML elements. The `div` element is there to group the elements inside it together and to put them below the canvas (without the `div` they'd appear to the right of the canvas).

The first element inside the div is a `button` element. It creates a clickable button. Any content between the opening and closing `button` tags will appear as text on the button. In this instance, the button will have the text `Clear`. Later, we'll write a JavaScript function that clears any circles on the canvas when the user clicks the button.

Next inside the `div` is an `input` element, which is used for taking values from the user. The `input` element doesn't allow any child elements, and so doesn't need a closing tag. In this case the `input` is of type `range`, which means it will display as a slider. This slider will be used to set the opacity of new circles drawn on the canvas. It has several attributes defining its functionality. `min` defines the minimum value the slider will produce, and `max` defines the maximum value. `value` defines the initial value the slider is set to, and `step` is how much the slider moves at a time. This slider is set to range from `0` to `1` in steps of `0.1`, and it starts at `1`, which corresponds to full opacity.

The last element in the `div` is a `label` element, which applies a label to another element. The `for` attribute of the label determines what element the label should be applied to; its value has to match the `id` of another element. In this case, we assign the label to the slider by specifying `opacity` as the target `id`. This way, the slider will be labeled *Opacity*, which is the text content of the `label` element. Thanks to the `label` element's `for` attribute, the browser understands that the `label` and `input` are related, and certain actions performed on the `label` will apply to the `input`. For example, if you hover over the `label`, the `input` will display as hovered, and if you click the `label`, the `input` will get keyboard focus (in this case, pressing left and right will decrease and increase the value of the slider).

Figure 10-6 shows how these elements should look, although their exact

appearance can vary depending on your browser and operating system. Load *index.html* in your browser and you should see something similar.

Figure 10-6 The new button and slider elements

Now that we have the HTML elements, we can write the JavaScript that will make this application interactive. First, we'll add some general declarations and the code for drawing circles when the user clicks on the `canvas` element. Update *script.js* with the code shown in Listing 10-8.

```
let canvas = document.querySelector("#canvas");
let ctx = canvas.getContext("2d");

let width = canvas.width;
let height = canvas.height;

let opacity = 1;

function drawCircle(x, y) {
  ctx.fillStyle = `rgba(0, 255, 0, ${opacity})`;
  ctx.beginPath();
  ctx.arc(x, y, 10, 0, Math.PI * 2, false);
  ctx.fill();
}

canvas.addEventListener("click", e => {
  drawCircle(e.offsetX, e.offsetY);
});
```

Listing 10-8 Drawing a circle on click

First we store the width and height of the `canvas` element in two variables, `width` and `height`. We'll need these variables later in our function for clearing the canvas. The `width` and `height` properties of the JavaScript `canvas` object come straight from the HTML `canvas` element's `width` and `height` properties (which are both `300` in *index.html*). We also initialize the variable `opacity` to 1.

Next, we create a helper function called `drawCircle`. This function takes an x and y coordinate and draws a filled circle at that location. We use the same path drawing methods demonstrated in Listing 10-6 to draw the circle. The `x` and `y` parameter becomes the circle's center, and we set its radius to `10` pixels.

One key difference from the previous drawing examples is that we're setting `fillStyle` to an *RGBA* color instead of a named color like `red` or

`blue` `1`. RGBA is a way of defining colors using four numbers: red, green, blue, and alpha. The first three correspond to the amount of each primary color of light. These three range from 0 to 255 and can be combined to produce any color you might want. Setting all three to 0 produces black and setting all three to 255 produces white. *Alpha* is another word for opacity, and it defines how opaque or transparent the color should be, ranging from 0 (completely transparent) to 1 (completely opaque).

In the Canvas API, you set RGBA colors using the string `"rgba(...)"` with the four values comma-separated. For example, setting `fillStyle` to the string `"rgba(0, 255, 0, 0.9)"` would make bright green circles that were slightly transparent. In our case, we wrap the RGBA string in backticks and use a placeholder for the alpha value to allow the user to change the opacity with the slider.

Lastly, we add a `click` event handler to the `canvas` element using `addEventListener`. The handler calls the `drawCircle` function we just created, passing the `offsetX` and `offsetY` properties of the click event as the function's parameters. The `offsetX` and `offsetY` properties give the distance of the click event from the top-left corner of the clicked element itself (rather than from the top-left corner of the whole browser window), and so are ideal for determining exactly where on the canvas the click happened.

Reload *index.html* in your browser and try clicking on the canvas. Wherever you click, a small green circle should appear, as shown in Figure 10-7!

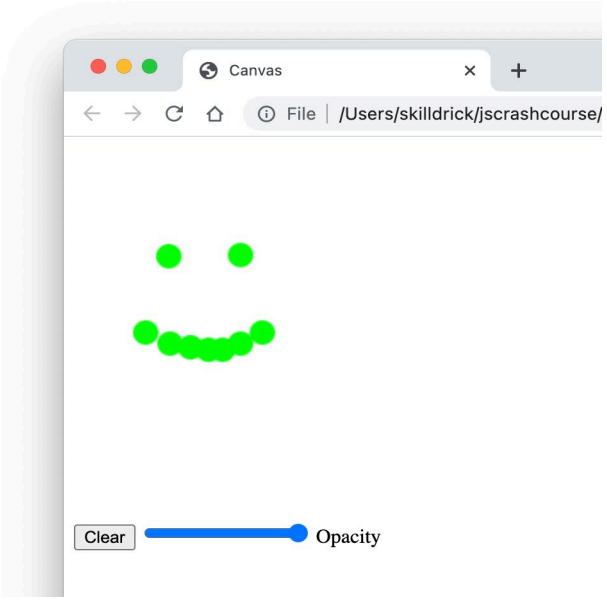


Figure 10-7 Drawing green circles with mouse clicks

To complete the drawing application, we need to wire up the Clear button and the Opacity slider. Add the code in Listing 10-9 to the end of *script.js*.

```
document.querySelector("#clear").addEventListener("click", ()
=> {
  ctx.clearRect(0, 0, width, height);
});

document.querySelector("#opacity").addEventListener("change",
e => {
  opacity = e.target.value;
});
```

Listing 10-9 Wiring up the Clear and Opacity controls

First we add a `click` event handler to the Clear button. This calls a Canvas API method called `clearRect`, which is used to clear a rectangular section of the canvas. Just like drawing a rectangle, you define the rectangle to be cleared using the x and y coordinates of its top left corner, followed by its width and height. Here we're clearing a rectangle that starts at the top-left corner of the canvas and is as wide and high as the canvas itself. Thus, `ctx.clearRect(0, 0, width, height);` clears the entire canvas.

Next we add a `change` event handler to the Opacity slider. The `change` event is triggered on `input` elements when their value changes, so this handler will be called whenever the slider is set to a new position. We get the `input` element with `e.target` and get the element's current value with `.value`. Then we update the `opacity` variable with this value. Because the `drawCircle` function uses the value of `opacity` as the alpha component of the RGBA color, any new circles will use the latest value set with the Opacity slider.

Now when you reload *index.html* in your browser you should have a fully functioning (if basic) drawing application! You can use the Opacity slider to change the opacity of new circles, and the Clear button to clear the canvas and start drawing again. Try drawing overlapping circles with the opacity slider set halfway to see how they overlay.

TRY IT OUT

3. Add sliders for controlling the R, G, and B components of the color. These will need to range from 0 to 255. You could also add a Radius slider that controls the radius of the circle drawn in the `drawCircle` function.
4. Make a new function called `drawSquare` that draws a square centered on a point, and call that function from the `click` handler instead of `drawCircle`.

Animating the Canvas

As noted at the beginning of the chapter, you can animate the canvas by updating the image multiple times per second. In this section we'll code a very simple animation to show the basics of how this works.

Animating the canvas generally follows this basic pattern:

1. Update state
2. Clear canvas
3. Draw image
4. Wait a short time
5. Repeat

State here means some variables storing information about the current frame of the animation. It could be the current location of an object in motion, a variable indicating what direction the object is moving, and so on. In our example, the state will be the x and y coordinates of a circle. When it's time to update the state, we'll increment the x and y coordinates by 1, meaning that the circle's position will gradually move diagonally down and to the right. Drawing the image will entail drawing a small circle at the x and y coordinates. We clear the canvas before drawing the circle to ensure that the image from the previous cycle is removed. We'll tackle the last two steps (waiting and repeating) by using the `setInterval` function to call our code every 100 milliseconds, or 10 times a second.

We can continue to work with the same HTML and JavaScript files. The only change to *index.html* is to remove the `div` and its nested elements that we added in Listing 10-7, as they're not needed any more. After removing those elements, update *script.js* based on the code in Listing 10-10.

```
let canvas = document.querySelector("#canvas");
let ctx = canvas.getContext("2d");
let width = canvas.width;
let height = canvas.height;

let x = 0;
let y = 0;

function drawCircle(x, y) {
  ctx.fillStyle = "rgb(0, 128, 255)";
  ctx.beginPath();
  ctx.arc(x, y, 10, 0, Math.PI * 2, false);
  ctx.fill();
}

function update() {
```

```
  x += 1;
  y += 1;
}

function draw() {
  ctx.clearRect(0, 0, width, height);
  drawCircle(x, y);
}

setInterval(() => {
  update();
  draw();
}, 100);
```

Listing 10-10 Creating an animation

We create two new variables, `x` and `y`, representing the location of the circle that we'll animate. These variables will store the current state of the animation and will be updated at regular intervals. The `drawCircle` function itself is mostly unchanged, although the `fillStyle` is now different. Now that we're not setting an opacity we can use the simpler `rgb()` format string for setting the R, G, and B values. With `rgb()`,the opacity of the color is always 100%.

After `drawCircle` we declare the `update` function, where we update the `x` and `y` variables, incrementing each by `1`. Next we declare the `draw` function, which clears the canvas and then calls `drawCircle` to draw a circle at the current x and y coordinates. Finally, we call `setInterval` to orchestrate the animation. You may recall from Chapter 6 that `setInterval` takes a function and a time interval in milliseconds, and repeatedly calls that function once every time interval. Here we're calling an anonymous function every 100 milliseconds. The function itself calls `update()` and `draw()` to create each frame of the animation.

Reload *index.html* in your browser, and you should see a small circle gradually move down the canvas from the top-left to the bottom right. Even after the circle leaves the canvas, the x and y coordinates will keep increasing, but the canvas ignores anything drawn outside of its bounds.

TRY IT OUT

5. Update the animation so the circle restarts at the top-left corner when it reaches the bottom-right corner. There are a few ways to do this. One option is to use the `%` remainder operator, which evenly divides the first operand by the second, and returns the remainder. For example, `325 % 100` gives `25`. By passing `x % width` and `y % height` to the `draw` function, you can ensure that the circle will always be drawn within the canvas. You can also use the `%=` operator to keep the `x` and `y` values within bounds in the `update` function using `x %= width` and `y %= height`

after incrementing their values. Try out both options.

6. How could you make the ball start out at the left of the canvas, move to the right, then move back, and so on? Hint: you'll need to declare another state variable to keep track of the direction, for example `let forwards = true`, and use that variable to decide whether to increment or decrement `x`. You'll then need to update the new variable to be `false` when the `x` value gets past a certain point.
7. Try changing the time interval in the `setInterval` function. For example, what does the animation look like with 1,000 ms, or 10 ms, or 1 ms? Note that at a certain point, the browser won't be able to update as fast as you're asking it to, so it's unlikely that a 1 ms interval will run 10 times faster than a 10 ms interval.

Conclusion

In this chapter you learned the basics of drawing on the `canvas` element, as well as some techniques for creating interactive applications and animations using the canvas. We'll build on some of these techniques later in this book as we learn how to make a canvas-based game.

Congratulations, you've just finished the second section of this book! You now know not only the basics of JavaScript, but some powerful techniques for using JavaScript within web pages to create interactive applications and games. In the rest of the book, we'll put these techniques to use over a series of projects, starting with a game.

11

PROJECT 1: MAKING A GAME

We've now reached the most exciting part of this book: it's time to start building real projects! In this first project, you'll use JavaScript to recreate one of the first arcade video games: the classic *Pong* from Atari. *Pong* is a simple game, but it will teach you some important aspects of game design: a game loop, player input, collision detection, and score keeping. There's even some basic artificial intelligence in there.

The Game

Pong was developed in 1972 and was released that year as a hugely

successful arcade machine (see Figure 11-1). It's a very basic game, consisting of a ball and two paddles, like table tennis. If the ball hits the top or bottom wall, it bounces off, but if it hits the left or right wall, the player on the opposite side scores a point. The ball bounces off the paddle normally, unless it hits near the top or bottom edge of the paddle, in which case the angle of return changes.

Figure 11-1 The original Pong arcade game (*https://www.flickr.com/photos/31988656@N00/33875579648*)

In this chapter, we'll make our own version of *Pong*, which we'll call *Tennjs* (like *Tennis* but with *JS*, get it?). In our game, the left paddle will be controlled by the computer and the right paddle will be controlled by a human player. In the original game, the paddles were controlled with rotating dial controllers, but in our version we'll use the mouse. The computer, rather than trying to anticipate where the ball will bounce, will just attempt to always match the vertical position of the ball. In order to give the human player a chance, we'll set an upper limit on how fast the computer can move the paddle.

Set Up

We'll begin by setting up the project's file structure and creating a canvas for displaying the game. As usual, the project will require an HTML file and a JavaScript file. We'll start with the HTML file. Create a directory called *tennjs* and a file in that directory called *index.html*. Then enter the content shown in Listing 11-1.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Tennjs</title>
  </head>
```

```
  <body>
    <canvas id="canvas" width="300" height="300"></canvas>
    <script src="script.js"></script>
  </body>
</html>
```

Listing 11-1 *index.html*

This is almost exactly the same as the HTML file created in Chapter 10, so there should be no surprises. The HTML creates a `canvas` element, where we'll draw the game, and a `script` element referencing the file *script.js*, where our game code will live.

Next we'll write some JavaScript to set up the canvas. Create the file *script.js*, and enter the code shown in Listing 11-2.

```
let canvas = document.querySelector("#canvas");
let ctx = canvas.getContext("2d");
let width = canvas.width;
let height = canvas.height;

ctx.fillStyle = "black";
ctx.fillRect(0, 0, width, height);
```

Listing 11-2 Setting up the canvas in *script.js*

The code here should also be familiar. We first get a reference to the canvas with `document.querySelector` and get the canvas's drawing context. Then we save the width and height of the canvas to variables called `width` and `height` for easy access within the code. Finally, we set the fill style to black and draw a black square the size of the canvas. This way the canvas appears to have a black background.

Open *index.html* in your browser and you should see something like in Figure 11-2.

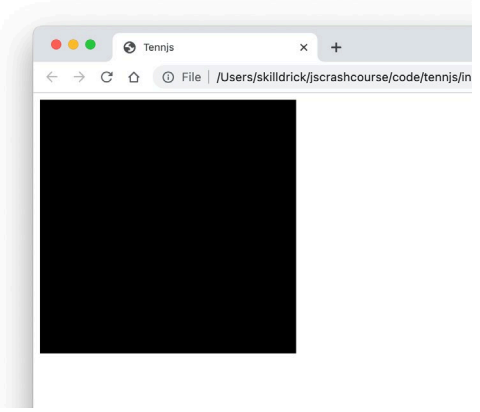


Figure 11-2 Our black square

We now have a blank, black canvas where we can create our game.

The Ball

Now it's time to draw the ball. Add the code in Listing 11-3 to the end of *script.js*.

```
1 const BALL_SIZE = 5;
2 let ballPosition = { x: 20, y: 30 };

  ctx.fillStyle = "white";
3 ctx.fillRect(ballPosition.x, ballPosition.y, BALL_SIZE,
  BALL_SIZE);
```

Listing 11-3 Drawing the ball

This code uses the `fillRect` method to draw the ball as a small white square near the top-left corner of the canvas 3. As in the original Pong game, the ball is a square rather than a circle. This gives the game a retro feel, and will also simplify the task of detecting when the ball has collided with the walls or with a paddle. The size of the ball is stored in a constant called `BALL_SIZE` 1. We use the "true constant" all-caps style for the identifier name because the ball size won't change during the course of the program. We could just use the value `5` instead of the constant `BALL_SIZE` when we call the `fillRect` method to draw the ball, but we're going to end up needing to refer to the ball's size a lot more throughout the program. Giving the size a name will make it much easier to understand code that needs to know the size of the ball. The other good thing about this approach is that if we change our mind later and decide the ball should be bigger or smaller, we only have to update the code in one place: at the declaration of the `BALL_SIZE` constant.

We keep track of the ball's position with an object containing its x and y coordinates, created at 2 using an object literal. In Chapter 10 we used separate variables for the x and y position of the circle that was being drawn, but it's a bit tidier to store the two variables together as on object, especially since this program is going to be longer and more complex.

Refresh *index.html* and you should see the white ball sitting in the top-left corner of the canvas, as shown in Figure 11-3.

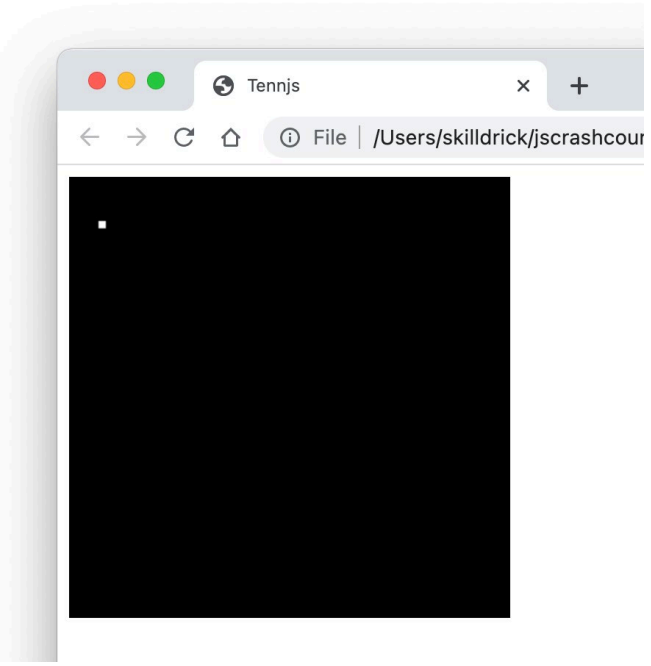


Figure 11-3 Figure 11-3: The ball

The ball is stationary for now, but soon enough we'll write code to make it move.

Refactoring

Next we're going to do a simple refactor. *Refactoring* is a software development term for modifying some code without changing its behavior, usually to make the code easier to understand or update. As the code for a project grows more complex, refactoring can help keep the code organized.

In this case, I know that we're going to want to draw to the canvas multiple times, not just once. In fact, we'll eventually want to redraw the canvas once every 30 milliseconds to give our game the appearance of motion. To make that easier to accomplish, we'll refactor so all the current drawing code becomes part of a function called `draw`. That way we can simply call the `draw` function any time we want to redraw the canvas.

Update *script.js* with the changes show in Listing 11-4.

```
let canvas = document.querySelector("#canvas");
let ctx = canvas.getContext("2d");
let width = canvas.width;
let height = canvas.height;
```

```
const BALL_SIZE = 5;
let ballPosition = { x: 20, y: 30 };

1 function draw() {
  ctx.fillStyle = "black";
  ctx.fillRect(0, 0, width, height);

  ctx.fillStyle = "white";
  ctx.fillRect(ballPosition.x, ballPosition.y, BALL_SIZE,
BALL_SIZE);
}

2 draw();
```

Listing 11-4 Refactoring the drawing code

The only change here is to group all the drawing code into a single function called `draw` 1, which we then immediately call at 2. Because it's a refactoring, nothing actually changes in the behavior of the program. You can refresh *index.html* to confirm that everything still looks as before.

The Game Loop

Almost all games contain a *game loop* that orchestrates everything that has to happen for each frame of the game. Game loops are similar to animation loops, like the one we looked in Chapter 10, but with some additional logic. Here's the general shape of the game loop in most games.

1. Clear canvas
2. Draw image
3. Get player input
4. Update state
5. Check collisions
6. Wait a short time
7. Repeat

Getting and acting on input from a player (or players) is the main thing that distinguishes a game from an animation. *Collision detection* is another important aspect of most games: checking for when two objects in the game meet and responding accordingly. Collision detection is what stops you from walking through walls or driving through another car—or in this case, it's what will make the ball bounce off the walls and paddles. Apart from these elements—player input and collision detection—the steps in the game loop are more or less the same as in an animation loop: we clear the canvas, draw the

image, update the state of the game to move objects to their new positions, pause, and repeat.

Rather than trying to write the whole game loop at once, we'll build it up gradually. Update *script.js* with the content in Listing 11-5, which will be the beginnings of the game loop in our game. This code moves the ball (that is, updates the ball's state), redraws the canvas, pauses, and repeats.

```
--snip--
const BALL_SIZE = 5;
let ballPosition = { x: 20, y: 30 };

1 let xSpeed = 4;
  let ySpeed = 2;

  function draw() {
    ctx.fillStyle = "black";
    ctx.fillRect(0, 0, width, height);

    ctx.fillStyle = "white";
    ctx.fillRect(ballPosition.x, ballPosition.y, BALL_SIZE,
BALL_SIZE);
  }

2 function update() {
    ballPosition.x += xSpeed;
    ballPosition.y += ySpeed;
  }

3 function gameLoop() {
    draw();
    update();

    // Call this function again after a timeout
  4 setTimeout(gameLoop, 30);
  }

5 gameLoop();
```

Listing 11-5 The game loop

The first change here is to initialize two new variables: `xSpeed` and `ySpeed` 1. We'll use these to control the horizontal and vertical speed of the ball. The new `update` function 2 uses these two variables to update the position of the ball. For every frame, the ball will move `xSpeed` pixels along the x axis and `ySpeed` pixels along the y axis. The two variables start out at `4` and `2`, so every frame the ball will move 4 pixels to the right and 2 pixels down.

The `gameLoop` function 3 calls the `draw` function followed by the `update` function. Finally it calls `setTimeout(gameLoop, 30);` 4.

which will call the `gameLoop` function again after 30 milliseconds. This is almost exactly the same as the `setInterval` technique we used in Chapter 10. You may recall that `setTimeout` only calls its function once after the timeout, while `setInterval` calls its function repeatedly. We're using `setTimeout` here so we have more control over whether or not to keep looping; later on we'll add some conditional logic to either call `setTimeout` or end the game.

At the end of the script, we call the `gameLoop` function to set the game in motion 5. Since `gameLoop` currently ends with `setInterval`, the result is that `gameLoop` will be repeatedly called once every 30 milliseconds. Reload your page and you should see the ball move across and down, much like the animation from Chapter 10.

Bouncing

In the previous section you got the ball moving, but it just flew off the edge of the canvas. Next you'll learn how to make it bounce off the edge of the canvas at the appropriate angle—our first collision detection code. Update *script.js* with the code in Listing 11-6, which adds a `checkCollision` function to our game.

```
--snip--
function update() {
  ballPosition.x += xSpeed;
  ballPosition.y += ySpeed;
}

function checkCollision() {
1 let ball = {
    left: ballPosition.x,
    right: ballPosition.x + BALL_SIZE,
    top: ballPosition.y,
    bottom: ballPosition.y + BALL_SIZE
  }

2 if (ball.left < 0 || ball.right > width) {
    xSpeed = -xSpeed;
  }
3 if (ball.top < 0 || ball.bottom > height) {
    ySpeed = -ySpeed;
  }
}

function gameLoop() {
  draw();
  update();
4 checkCollision();
```

```
  // Call this function again after a timeout
  setTimeout(gameLoop, 30);
}

gameLoop();
```

Listing 11-6 Wall collision detection

The new function `checkCollision` checks to see if the ball has collided with one of the four walls of the canvas. If it has, it updates `xSpeed` or `ySpeed` as appropriate to make the ball bounce off the wall. First, we calculate values for the edges of the ball. We need to know where the left, right, top, and bottom edges are to determine if these edges have exceeded the bounds of the playing area. We group the values in an object called `ball` 1 that has `left`, `right`, `top`, and `bottom` properties. Identifying the left and top ball edges is easy: they're `ballPosition.x` and `ballPosition.y`, respectively. To get the right and bottom edges, we add `BALL_SIZE` to `ballPosition.x` and `ballPosition.y`. This is one of those cases noted earlier where having access to the ball's size as a constant is helpful.

Next, we perform the actual collision detection. If the left edge of the ball is less than 0, or the right edge of the ball is greater than the width of the canvas 2, we know that the ball has hit the left or right wall. In both cases, the math is the same: the new value of `xSpeed` should be the negative of the current value (that is, the value is *negated*). For example, the first time the ball hits the right edge, `xSpeed` will go from `4` to `-4`. Meanwhile, `ySpeed` remains unchanged. As a result, the ball continues moving down the screen at the same rate, but now it's moving to the left instead of to the right.

The same kind of check happens for the top of the ball colliding with the top wall or the bottom of the ball colliding with the bottom wall 3. In either of these cases, we negate `ySpeed`, changing it from `2` to `-2` when the ball hits the top edge, or from `-2` to `2` when the ball hits the bottom edge.

The only other change to the code is to add a call to `checkCollision` to the list of things that happen in the `gameLoop` function 4. Now when you refresh *index.html* you should see the ball continuously bounce around the play area.

If you've been paying attention, you might have noticed that the ball isn't supposed to bounce off the left and right walls. Once we have moving paddles, we'll modify the collision detection code to only bounce off the paddles or the top and bottom walls, and to score a point for a side wall collision.

The Paddles

Our next task is to draw the two paddles. To do that we'll first introduce some new constants that establish the paddle dimensions and their horizontal

position relative to the sides of the canvas, as well as some variables defining their vertical positions. (The paddles can only move up and down, not from side to side, so only their vertical positions need to be variables.) Update *script.js* with the changes in Listing 11-7.

```
--snip--
let xSpeed = 4;
let ySpeed = 2;

const PADDLE_WIDTH = 5;
const PADDLE_HEIGHT = 20;
const PADDLE_OFFSET = 10;

let leftPaddleTop = 10;
let rightPaddleTop = 30;

function draw() {
--snip--
```

Listing 11-7 Defining the paddles

First we set up the constants that define the paddles. `PADDLE_WIDTH` and `PADDLE_HEIGHT` define both paddles to be 5 pixels wide and 20 pixels tall. `PADDLE_OFFSET` refers to the distance of the paddle from the left or right edge of the playing area.

The variables `leftPaddleTop` and `rightPaddleTop` define the current vertical position of the top of each paddle. Eventually, `leftPaddleTop` will be controlled by the computer through a function we'll write to follow the ball, and `rightPaddleTop` will be updated when the player moves the mouse. For now, we're simply setting these values to `10` and `30`, respectively.

Next we update the `draw` function to display the paddles using the information we just defined. I've also added comments to the code to clarify what's happening at each step of the `draw` function. Modify the code as shown in Listing 11-8.

```
--snip--
function draw() {
  // Fill the canvas with black
  ctx.fillStyle = "black";
  ctx.fillRect(0, 0, width, height);

  // Everything else will be white
  ctx.fillStyle = "white";

  // Draw the ball
  ctx.fillRect(ballPosition.x, ballPosition.y, BALL_SIZE,
BALL_SIZE);
```

```
  // Draw the paddles
1 ctx.fillRect(
    PADDLE_OFFSET,
    leftPaddleTop,
    PADDLE_WIDTH,
    PADDLE_HEIGHT
  );
2 ctx.fillRect(
    width - PADDLE_WIDTH - PADDLE_OFFSET,
    rightPaddleTop,
    PADDLE_WIDTH,
    PADDLE_HEIGHT
  );
}

function update() {
--snip--
```

Listing 11-8 Drawing the paddles

The new code features two calls of `fillRect`, one for drawing each paddle, at 1 and 2. I've split the arguments over multiple lines because the identifiers are so long! Remember that the parameters to `fillRect` are *`x`*, *`y`*, *`width`*, and *`height`*, where *`x`* and *`y`* are the coordinates of the top-left corner of the rectangle. The x coordinate of the left paddle is `PADDLE_OFFSET` because we're using that to mean the paddle's distance from the left edge of the canvas, while the y coordinate of the left paddle is just `leftPaddleTop`. The *`width`* and *`height`* arguments are the `PADDLE_WIDTH` and `PADDLE_HEIGHT` constants.

The right paddle is a little more complicated to draw: to get the x coordinate of the paddle's top-left corner, we need to take the width of the canvas and subtract the width of the paddle and the offset of the paddle from the right edge. Given that the width of the canvas is 500, and the paddle width and offset are both 10, that means the x coordinate of the right paddle is 480.

When you refresh *index.html* you should see the two paddles now, in addition to the bouncing ball, as shown in Figure 11-4.

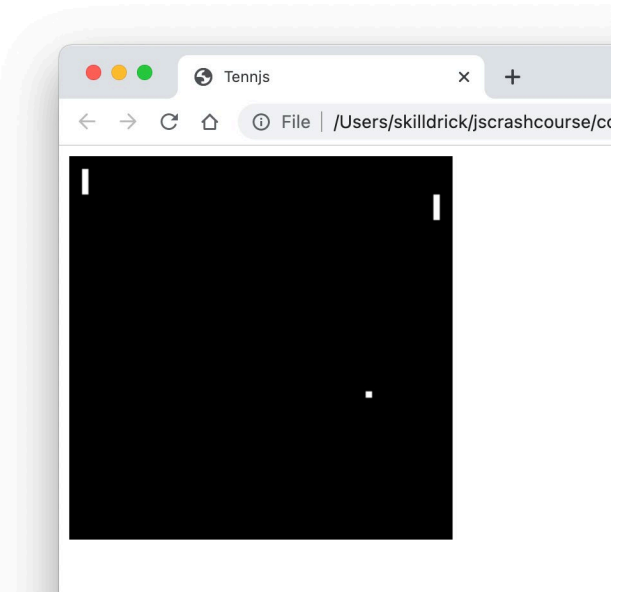


Figure 11-4 The paddles and ball

Note that the ball currently passes straight through the paddles, because we haven't set up collision detection for the paddles yet. We'll get to that later in this section.

Player Input to Move the Paddle

The paddles are drawn at the vertical positions given by the variables `leftPaddleTop` and `rightPaddleTop`, so to make the paddles move up and down, we just have to update the value of these variables. Right now we're just concerned with the right paddle, which will be controlled by the human player.

To let the player control the right paddle, we'll add an event handler to *script.js* that listens for `mousemove` events. Listing 11-9 shows how it's done.

```
--snip--
let leftPaddleTop = 10;
let rightPaddleTop = 30;

document.addEventListener("mousemove", e => {
  rightPaddleTop = e.y - canvas.offsetTop;
});

function draw() {
--snip--
```

Listing 11-9 Moving the right paddle

This code follows the same pattern for event handling you first saw in Chapter 9. We use `document.addEventListener` to check for mouse movements. When one is detected, the event handler function updates the value of `rightPaddleTop` based on the y coordinate of the `mousemove` event, accessed as `e.y`. The y coordinate is relative to the top of the page, not the top of the canvas, so we subtract `canvas.offsetTop` (the distance of the top of the canvas from the top of the page) from the y coordinate. This way the assigned `rightPaddleTop` value will be based on the distance of the mouse from the top of the canvas, and the paddle will follow the mouse accurately.

Refresh *index.html* and you should see the right paddle move vertically as the mouse moves up and down. Figure 11-5 shows how it should look.

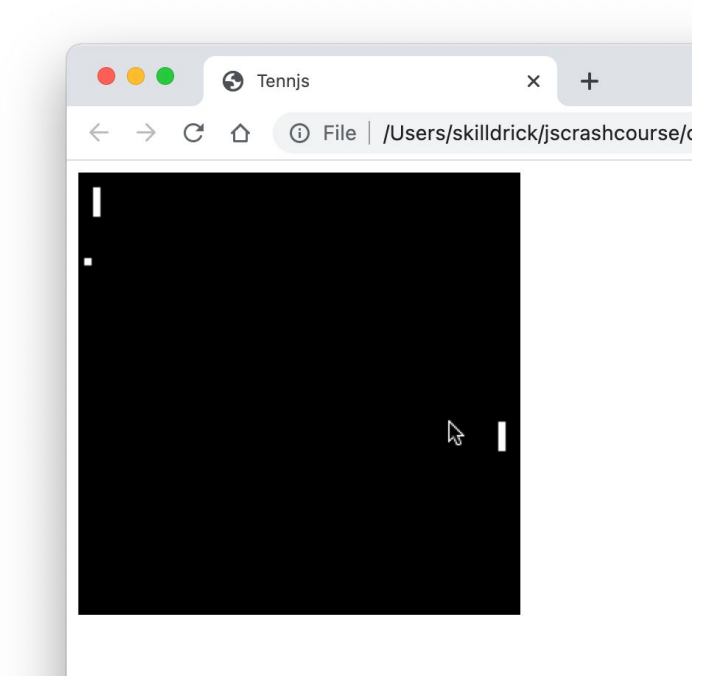


Figure 11-5 The right paddle moving with the mouse

Our game has now officially become interactive! The player has full control of the position of the right paddle.

Paddle Collision Detection

The next step is to add collision detection for the paddles. We need to

know if the ball has hit a paddle, and make the ball bounce off the paddle appropriately. This requires a lot of code, so I'll break it up over a few listings.

The first thing we have to do is to create objects defining the four edges of the two paddles, as we did earlier for the ball in Listing 11-6. These changes are shown in Listing 11-10.

```
--snip--
function checkCollision() {
  let ball = {
    left: ballPosition.x,
    right: ballPosition.x + BALL_SIZE,
    top: ballPosition.y,
    bottom: ballPosition.y + BALL_SIZE
  }

  let leftPaddle = {
    left: PADDLE_OFFSET,
    right: PADDLE_OFFSET + PADDLE_WIDTH,
    top: leftPaddleTop,
    bottom: leftPaddleTop + PADDLE_HEIGHT
  };

  let rightPaddle = {
    left: width - PADDLE_WIDTH - PADDLE_OFFSET,
    right: width - PADDLE_OFFSET,
    top: rightPaddleTop,
    bottom: rightPaddleTop + PADDLE_HEIGHT
  };

  if (ball.left < 0 || ball.right > width) {
--snip--
```

Listing 11-10 Calculating the edges of the paddles

The two objects, `leftPaddle` and `rightPaddle`, each contain the edges of the respective paddles as four properties, `left`, `right`, `top`, and `bottom`. As in Listing 11-8, the right paddle needs a bit more math because its position has to take into account the width of the canvas, the offset of the paddle, and the width of the paddle.

Next we need a function, which we'll call `checkPaddleCollision`, that takes the ball object and one of the paddle objects and returns `true` if the ball is intersecting with that paddle. See the definition in Listing 11-11.

```
--snip--
function update() {
  ballPosition.x += xSpeed;
  ballPosition.y += ySpeed;
}

function checkPaddleCollision(ball, paddle) {
```

```
  // check if the paddle and ball overlap vertically and
horizontally
  return (
    ball.left   < paddle.right &&
    ball.right  > paddle.left &&
    ball.top    < paddle.bottom &&
    ball.bottom > paddle.top
  );
}

function checkCollision() {
--snip--
```

Listing 11-11 The `checkPaddleCollision` function

This function will be called with the ball and each of the paddle objects defined earlier. `checkPaddleCollision` uses a long Boolean expression made up of four sub-expressions which are all `&&`'d together, so it only returns `true` if all four sub-expressions are `true`. (Note: I added spacing to each sub-expression so the operands line up vertically—this is just to make the code easier to read.) In English, the sub-expressions say:

1. The left edge of the ball must be to the left of the right edge of the paddle
2. The right edge of the ball must be to the right of the left edge of the paddle
3. The top edge of the ball must be above the bottom edge of the paddle
4. The bottom edge of the ball must be below the top edge of the paddle

If the first two conditions are true then the ball is intersecting horizontally, and if the last two conditions are true, the ball is intersecting vertically. The ball is only truly intersecting with the paddle if all four conditions are `true`. To illustrate this, see Figure 11-6.

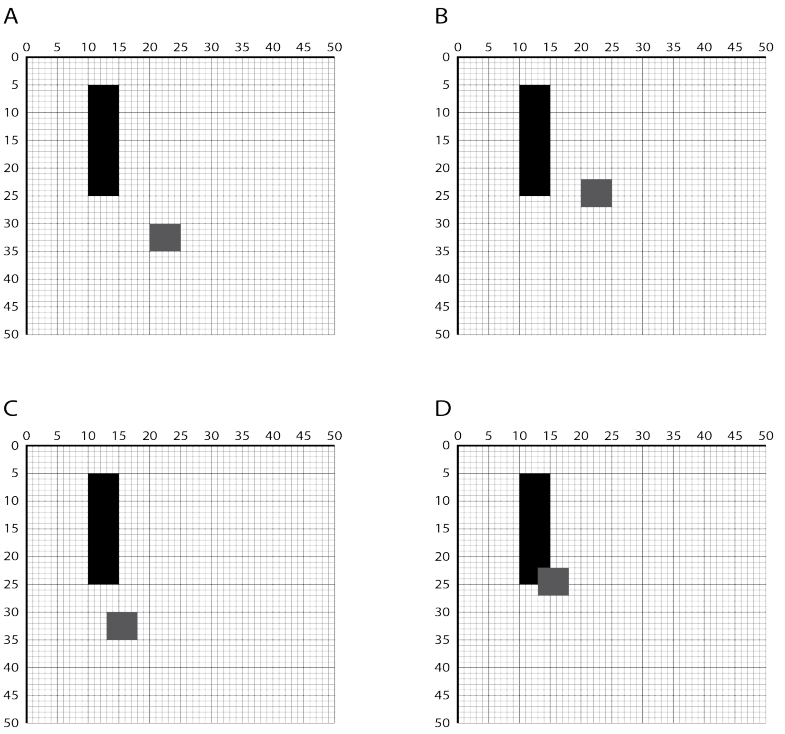


Figure 11-6 Collision detection illustrated

The figure shows four possible scenarios we might check. In all the scenarios, the paddle has the following bounds: `{ left: 10, right: 15, top: 5, bottom: 25 }`.

In A, `ball` has the bounds `{ left: 20, right: 25, top: 30, bottom: 35 }`. `ball.left < paddle.right` is `false` (the left side of the ball is not to the left of the right side of the paddle), but `ball.right > paddle.left` is `true`. Likewise, `ball.top < paddle.bottom` is `false` and `ball.bottom > paddle.top` is `true`.

In B, `ball` has the bounds `{ left: 20, right: 25, top: 22, bottom: 27 }`. This time, `ball.top < paddle.bottom` and `ball.bottom > paddle.top` are both `true`, which means that the ball is vertically intersecting with the paddle, but not horizontally intersecting.

In C, `ball` has the bounds `{ left: 13, right: 18, top: 30, bottom: 35 }`. In this case, the ball is horizontally intersecting with the paddle, but not vertically intersecting.

Finally, in D, `ball` has the bounds `{ left: 13, right: 18, top: 22, bottom: 27 }`. Now the ball is both horizontally and

vertically intersecting with the paddle, all four sub-expressions are `true`, and so `checkPaddleCollision` returns `true`.

Now it's time to actually call the `checkPaddleCollision` function for the two paddles, and handle the case where the function returns `true`. You can find this code in Listing 11-12.

```
--snip--
  let rightPaddle = {
    left: width - PADDLE_WIDTH - PADDLE_OFFSET,
    right: width - PADDLE_OFFSET,
    top: rightPaddleTop,
    bottom: rightPaddleTop + PADDLE_HEIGHT
  };

  if (checkPaddleCollision(ball, leftPaddle)) {
    // left paddle collision happened
  1 xSpeed = Math.abs(xSpeed);
  }

  if (checkPaddleCollision(ball, rightPaddle)) {
    // right paddle collision happened
  2 xSpeed = -Math.abs(xSpeed);
  }

  if (left < 0 || right > width) {
    xSpeed = -xSpeed;
  }
  if (top < 0 || bottom > height) {
    ySpeed = -ySpeed;
  }
}
--snip--
```

Listing 11-12 Checking for paddle collisions

Remember that `checkPaddleCollision` takes an object representing the ball and an object representing a paddle and returns `true` if the two are intersecting. If `checkPaddleCollision(ball, leftPaddle)` returns `true`, we set `xSpeed` to `Math.abs(xSpeed)` 1, which has the effect of setting it to 4 because in our game `xSpeed` is only ever 4 (when moving to the right) or -4 (when moving to the left).

You might be wondering why we didn't just negate `xSpeed`, as we did with the vertical wall collision code earlier. Using the absolute value is a little trick to avoid multiple collisions that could send the ball bouncing back and forth inside the paddle. It's possible if the ball hits at just the right point at the end of the paddle that it would get bounced back, but the next frame would also result in a collision with the same paddle. If we were negating the `xSpeed` then it would just keep bouncing "inside" the paddle. By forcing the updated `xSpeed` to be positive, we can ensure that a collision with the left

paddle will always result in the ball bouncing to the right.

Following this, we do the same thing with the right paddle. In this case, if there's a collision we update `xSpeed` to `-Math.abs(xSpeed)` ,which in effect is -4, meaning that the ball will bounce to the left 2.

Refresh *index.html* again and try to move the right paddle with your mouse so the ball hits it. You should now have ball/paddle bounces happening! At this point the ball can still safely bounce off the side walls, but we'll fix that soon.

Bouncing Near the Paddle Ends

I mentioned at the beginning of this chapter that in *Pong* you can change the angle of the ball's bounce by hitting near the top or bottom of the paddle. We'll implement that functionality now. First we'll add a new function called `adjustAngle` immediately before `checkCollision`. It checks if the ball is near the top or bottom of the paddle, and updates `ySpeed` if it is. See Listing 11-13 for the code.

```
--snip--
function adjustAngle(distanceFromTop, distanceFromBottom) {
1 if (distanceFromTop < 0) {
    // If ball hit near top of paddle, reduce ySpeed
    ySpeed -= 0.5;
  } 2 else if (distanceFromBottom < 0) {
    // If ball hit near bottom of paddle, increase ySpeed
    ySpeed += 0.5;
  }
}

function checkCollision() {
--snip--
```

Listing 11-13 Adjusting the bounce angle

The `adjustAngle` function has two parameters, `distanceFromTop` and `distanceFromBottom`. These represent the distance from the top of the ball to the top of the paddle, and from the bottom of the paddle to the bottom of the ball, respectively. At 1 we check if `distanceFromTop` is less that `0`. If so, that means the top edge of the ball is above the top edge of the paddle at collision time, which is how we'll define being near the top of the paddle. In this case, we subtract `0.5` from `ySpeed`. If the ball is moving down the screen when it hits near the top of the paddle, then `ySpeed` is positive, so subtracting 0.5 reduces the vertical speed. For example, at the start of the game, `ySpeed` is `2`. If you align the paddle so the ball hits the top, `ySpeed` will become 1.5 after the bounce, effectively reducing the angle of bounce. However, if the ball is moving up the screen, then `ySpeed` is negative. In this case, subtracting 0.5 after a hit near the top of the paddle will

increase the ball's vertical speed. For example, a `ySpeed` of `-2` will become `-2.5`.

The opposite happens if the ball hits near the bottom of the paddle, which we check at 2. In this case, we add `0.5` to `ySpeed`, increasing the vertical speed if the ball is moving down the screen or decreasing the speed if the ball is moving up the screen.

Next we update the `checkCollision` function to call the new `adjustAngle` function as part of the collision-detection logic for the two paddles. Listing 11-14 shows the changes.

```
--snip--
function checkCollision() {
--snip--
  if (checkPaddleCollision(ball, leftPaddle)) {
    // left paddle collision happened
    let distanceFromTop = ball.top - leftPaddle.top;
    let distanceFromBottom = leftPaddle.bottom - ball.bottom;
    adjustAngle(distanceFromTop, distanceFromBottom);
    xSpeed = Math.abs(xSpeed);
  }

  if (checkPaddleCollision(ball, rightPaddle)) {
    // right paddle collision happened
    let distanceFromTop = ball.top - rightPaddle.top;
    let distanceFromBottom = rightPaddle.bottom - ball.bottom;
    adjustAngle(distanceFromTop, distanceFromBottom);
    xSpeed = -Math.abs(xSpeed);
  }
--snip--
```

Listing 11-14 Listing 11-14: Calling `adjustAngle`

Within the `if` statement for each paddle, we declare `distanceFromTop` and `distanceFromBottom`, the arguments needed for the `adjustAngle` function. Then we call `adjustAngle` before updating `xSpeed` as before.

Now try out the game and see if you can hit the ball near the edge of the paddle!

TRY IT OUT

Hitting the edge of the paddle can be tricky. To make it easier, try reducing the speed of the game by increasing the `setTimeout` interval—for example, from 30 milliseconds to 60 milliseconds. Another option for making it easier is to expand what counts as "near the top" and "near the bottom" of the paddle. Instead of `distanceFromTop < 0` you could do `distanceFromTop < 5`, for example, which would check that the top of the ball is less than 5 pixels below the top of the paddle.

It also isn't always obvious when a top or bottom hit has occurred, since the change to ySpeed is pretty small. To get some more feedback as to what's actually happening when the ball hits the paddle, you can add logging to the `adjustAngle` function. For example, you could add the following line to the start of the function:

```
console.log(`top: ${distanceFromTop}, bottom: ${distanceFromBottom}`);
```

This way the console will show the ball's distance from the top and bottom of the paddle every time the ball hits the paddle. Another thing that might help is adding logging to the two conditionals within the `adjustAngle` function, like so:

```
  if (distanceFromTop < 0) {
    // If ball hit near top of paddle, reduce ySpeed
    console.log("Top hit!");
    ySpeed -= 0.5;
  } else if (distanceFromBottom < 0) {
    // If ball hit near bottom of paddle, increase ySpeed
    console.log("Bottom hit!");
    ySpeed += 0.5;
  }
```

Now you'll get additional feedback that you've hit the top or bottom of the paddle, and that `ySpeed` is being adjusted.

You should be careful about where you add logging in games. If you add logging in the `checkCollision` function, for example, then every frame of the game will produce a new log line, which gets very noisy and hard to read, and can also lead to performance problems. It's best to limit the logging to certain conditions that won't be true all the time, for example only logging when a collision occurs, as we did here.

Scoring Points

Games are usually more fun when you can win or lose. In *Pong*, you score a point if you hit the wall behind the opposing player's paddle. After the ball hits the wall, the ball is reset to its starting position and speed.

First we're going to need a way to keep track of the scores. To do that, we'll create some new variables. Update *script.js* with the code in Listing 11-15.

```
--snip--
let leftPaddleTop = 10;
let rightPaddleTop = 30;

let leftScore = 0;
let rightScore = 0;

document.addEventListener("mousemove", e => {
  rightPaddleTop = e.y - canvas.offsetTop;
});
--snip--
```

Listing 11-15 Variables to keep track of the scores

We declare two new variables, `leftScore` and `rightScore`, and set them both to zero. Later we'll add logic to increment these variables when points are scored.

Next we add code for displaying the scores to the end of the `draw` function. Update the function as shown in Listing 11-16.

```
function draw() {
  --snip--
  ctx.fillRect(
    width - PADDLE_WIDTH - PADDLE_OFFSET,
    rightPaddleTop,
    PADDLE_WIDTH,
    PADDLE_HEIGHT
  );

  // Draw scores
1 ctx.font = "30px monospace";
2 ctx.textAlign = "left";
3 ctx.fillText(leftScore.toString(), 50, 50);
  ctx.textAlign = "right";
  ctx.fillText(rightScore.toString(), width - 50, 50);
}
```

Listing 11-16 Drawing the scores

This added code uses some new canvas properties and methods we haven't seen yet. First, we set the font of the text we're about to draw 1. This is similar to a CSS font declaration. In this case, we're setting the font to be 30 pixels tall and monospace style. Monospace means that each character takes up the same width, and is usually used for code, as in this book's code listings. It looks `like this`. There are many monospace fonts, but because operating systems can come with different fonts installed, we only give a generic font style (monospace), meaning the operating system should use the default font for that font style. In most operating systems, Courier or Courier New is the default monospace font.

Next we use `ctx.textAlign` to set the alignment for the text 2. We choose left alignment, but this will only apply to the left score. Before drawing the right score, we change the alignment to `"right"`. This way if the scores get into double digits, the numbers will extend towards the middle of the screen, keeping things visually balanced.

At 3 we use the `fillText` method to display the left score. This method has three parameters: the text to be drawn, and the x and y coordinates at which to draw the text. The first parameter must be a string, so we call the `toString` method on `leftScore` to convert it from a number to a string. We use `50` for the x and y coordinates to place the text near the top-left corner of the canvas.

NOTE The meaning of the x coordinate parameter for `fillText` depends on the text's alignment. For left-aligned text, the x coordinate specifies the left edge of the text, whereas for right-aligned text it specifies the right edge.

The right score is handled similarly to the left score: we set the text alignment, then call `fillText` to display the score. This time we set the x coordinate to `width - 50`, so it appears as far from the right as the left score appears from the left.

When you refresh *index.html* you should see the initial scores rendered, as illustrated in Figure 11-7.

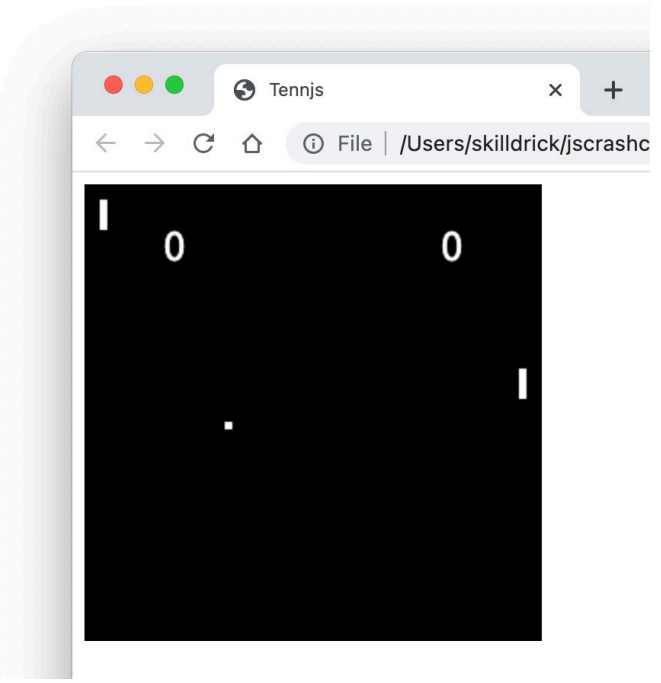


Figure 11-7 Displaying the scores

Next we have to handle the case where the ball hits the side walls. Instead of bouncing, the appropriate score should be incremented and the ball should be reset to its original speed and position. First we'll do another refactor and write a function that resets the ball. This also requires some changes to how the ball's speed and position variables are handled. Listing 11-17 shows the changes.

```
--snip--
const BALL_SIZE = 5;
let ballPosition;

let xSpeed;
```

```
let ySpeed;

function initBall() {
2 ballPosition = { x: 20, y: 30 };
  xSpeed = 4;
  ySpeed = 2;
}

const PADDLE_WIDTH = 5;
--snip--
```

Listing 11-17 The `initBall` function

Here we've separated the *declaration* of the ball state variables (`ballPosition`, `xSpeed`, and `ySpeed`) from the *initialization* of those variables. For example, `ballPosition` is declared at the top level of the program 1, but initialized in the new `initBall` function (short for "initialize ball") 2. The same goes for `xSpeed` and `ySpeed`. This is so we can reset the ball to its initial position and speed whenever we want simply by calling `initBall`, rather than by copy-pasting the values of the ball state variables all over the program. In particular, we can now call `initBall` at the start of the program to set up the ball for the first time, and we can also call it anytime the ball hits the left or right wall, to reset the ball to its original state.

Note that we can't both declare *and* initialize the ball state variables inside the `initBall` function—for example by placing `let ballPosition = { x: 20, y: 30 };` within the function—because the `let` keyword defines a new variable *in the current scope*, which in that case would be the body of `initBall`. Thus, the variables would only be available within `initBall`. In fact, we want the variables to be available throughout the program, so we declare them with `let` at the top level of the program, outside the body of any functions. However, because we want to initialize the variables multiple times, we assign them their value in the `initBall` function, which can be called repeatedly.

Next we have to modify the collision detection code to increment the score and reset the ball when the left or right wall is hit. Listing 11-18 shows how.

```
function checkCollision() {
--snip--
  if (checkPaddleCollision(ball, rightPaddle)) {
    // right paddle collision happened
    let distanceFromTop = ball.top - rightPaddle.top;
    let distanceFromBottom = rightPaddle.bottom - ball.bottom;
    adjustAngle(distanceFromTop, distanceFromBottom);
    xSpeed = -Math.abs(xSpeed);
  }

1 if (ball.left < 0) {
    rightScore++;
```

```
      initBall();
    }
  2 if (ball.right > width) {
      leftScore++;
      initBall();
    }
    if (ball.top < 0 || ball.bottom > height) {
      ySpeed = -ySpeed;
    }
  }
  --snip--
```

Listing 11-18 Scoring points on wall collisions

Previously, we checked for left and right wall collisions in a single `if` statement, but now we have to handle the left and right walls individually, since a different player scores depending on which wall is hit. Therefore, we've split the original `if` statement into two. If the ball hits the left wall 1, `rightScore` is incremented and the ball is reset with a call to our new `initBall` function. If the ball hits the right wall 2, `leftScore` is incremented and the ball is reset. The logic for collisions with the top and bottom walls remains the same.

Finally, since we've moved the initialization of the ball state variables to the `initBall` function, we need to call that function before the game loop starts in order to set the ball up for the first time. Scroll down to the bottom of script.js and update the code as shown in Listing 11-19.

```
--snip--
function gameLoop() {
  draw();
  update();
  checkCollision();

  // Call this function again after a timeout
  setTimeout(gameLoop, 30);
}

1 initBall();
gameLoop();
```

Listing 11-19 Calling `initBall` for the first time

We've added a call to `initBall` before the call to `gameLoop` 1. Now when you refresh *index.html* you should see the scores increment when the ball hits the side wall, and the ball should reset to its original speed and position after a side wall hit. Of course, it's pretty easy to beat the computer right now because it doesn't move its paddle yet!

Computer Control

Now let's add some challenge to this game! We want the computer-controlled opponent to move the paddle and try to hit the ball. There are various ways to do this, but in our simple approach, we'll have the computer always try to match the current position of the ball. The logic for the computer will be very simple:

- If the top of the ball is above the top of the paddle, move the paddle up.
- If the bottom of the ball is below the bottom of the paddle, move the paddle down.
- Otherwise, do nothing.

With this approach, if the computer could move at any speed, then it would never miss. Since this would be no fun for us humans, we'll set a speed limit for the computer. We'll do that first, declaring the computer's speed limit as a constant. Listing 11-20 shows how.

```
let canvas = document.querySelector("#canvas");
let ctx = canvas.getContext("2d");
let width = canvas.width;
let height = canvas.height;

const MAX_COMPUTER_SPEED = 2;

const BALL_SIZE = 5;
--snip--
```

Listing 11-20 Limiting the computer's speed

We declare the constant `MAX_COMPUTER_SPEED`. By setting it to `2`, we're saying that the computer isn't allowed to move the paddle more than two pixels per frame of the game.

Next we'll define a function called `followBall` that moves the computer's paddle. The new function is shown in Listing 11-21.

```
--snip--
function draw() {
--snip--

function followBall() {
1 let ball = {
    top: ballPosition.y,
    bottom: ballPosition.y + BALL_SIZE
  };
2 let leftPaddle = {
    top: leftPaddleTop,
    bottom: leftPaddleTop + PADDLE_HEIGHT
  };
```

```
3 if (ball.top < leftPaddle.top) {
    leftPaddleTop -= MAX_COMPUTER_SPEED;
  } 4 else if (ball.bottom > leftPaddle.bottom) {
    leftPaddleTop += MAX_COMPUTER_SPEED;
  }
}

function update() {
  ballPosition.x += xSpeed;
  ballPosition.y += ySpeed;
5 followBall();
}
```

Listing 11-21 Computer-controlled paddle

Within the `followBall` function, we define objects representing the ball 1 and the left paddle 2, each with `top` and `bottom` properties representing their upper and lower bounds. Then we implement the paddle movement logic with two `if` statements. If the top of the ball is above the top of the paddle 3, we move the paddle up by subtracting `MAX_COMPUTER_SPEED` from `leftPaddleTop`. Likewise, if the bottom of the ball is below the bottom of the paddle 4, we move the paddle down by adding `MAX_COMPUTER_SPEED` to `leftPaddleTop`.

At 5 we call the new `followBall` function within the `update` function. This way moving the left paddle becomes part of the process of updating the state of the game that happens with each iteration of the game loop.

Reload the page and see if you can score a point against the computer!

Game Over

The final step in making our game is to make it winnable! To do that, we have to add some kind of game-over condition, and stop the game loop at that point. In this case, we'll stop the game loop once one of the players reaches 10 points, then display the text "GAME OVER".

First we need to declare a variable for keeping track of whether or not the game is over. We'll use this variable to decide whether or not to continue repeating the gameLoop function. We add it in Listing 11-22.

```
--snip--
let leftScore = 0;
let rightScore = 0;
1 let gameOver = false;
--snip--

function checkCollision() {
```

```
    --snip--
    if (ball.right > width) {
      leftScore++;
      initBall();
    }
  2 if (leftScore > 9 || rightScore > 9) {
      gameOver = true;
    }
    if (ball.top < 0 || ball.bottom > height) {
      ySpeed = -ySpeed;
    }
  }
  --snip--
```

Listing 11-22 The `gameOver` variable

At 1, near the top of *script.js*, we declare a variable called `gameOver` for recording whether the game is over. We initialize it to `false` so the game doesn't end before it begins. Then, within the `checkCollision` function, we check to see if either of the scores has exceeded 9 2. If so, we set `gameOver` to `true`. This check could happen anywhere, but we do it in `checkCollision` to keep the logic that increments the scores and the logic that checks the scores together.

Next we add a function for writing the text "GAME OVER" and modify the game loop so it ends when `gameOver` is `true`. Listing 11-23 shows how.

```
function checkCollision() {
  --snip--

1 function drawGameOver() {
    ctx.fillStyle = "white";
    ctx.font = "30px monospace";
    ctx.textAlign = "center";
    ctx.fillText("GAME OVER", width / 2, height / 2);
  }

  function gameLoop() {
    draw();
    update();
    checkCollision();

  2 if (gameOver) {
      draw();
      drawGameOver();
    } else {
      // Call this function again after a timeout
      setTimeout(gameLoop, 30);
    }
  }
```

Listing 11-23 Ending the game

At 1 we define the `drawGameOver` function. It draws the text "GAME OVER" to the middle of the canvas in large, white text. To position the text in the middle of the canvas, we set the text alignment to `"center"` and use half the canvas width and height as the text's x and y coordinates. (With center alignment, the x coordinate refers to the horizontal midpoint of the text.)

Within the `gameLoop` function, we've wrapped the call to `setTimeout` within a conditional statement. At 2 we check the value of the `gameOver` variable. If it's `true`, the game is over, so we call the `draw` and `drawGameOver` functions. (The `draw` function is needed to display the final score—otherwise the winning player would still be stuck with 9 points.) If `gameOver` is `false`, the game can continue: we keep looping as before by using `setTimeout` to call `gameLoop` again after 30 milliseconds.

Once `gameOver` becomes `true` and the game loop ends, the game effectively stops. Nothing else will be drawn to the screen after the "GAME OVER" text—at least, not until the page is refreshed and the program starts again from the beginning. Go ahead and do that now: refresh *index.html* and see if you can beat the computer! Once one of you gets more than 9 points you should see the "GAME OVER" text, as shown in Figure 11-8.

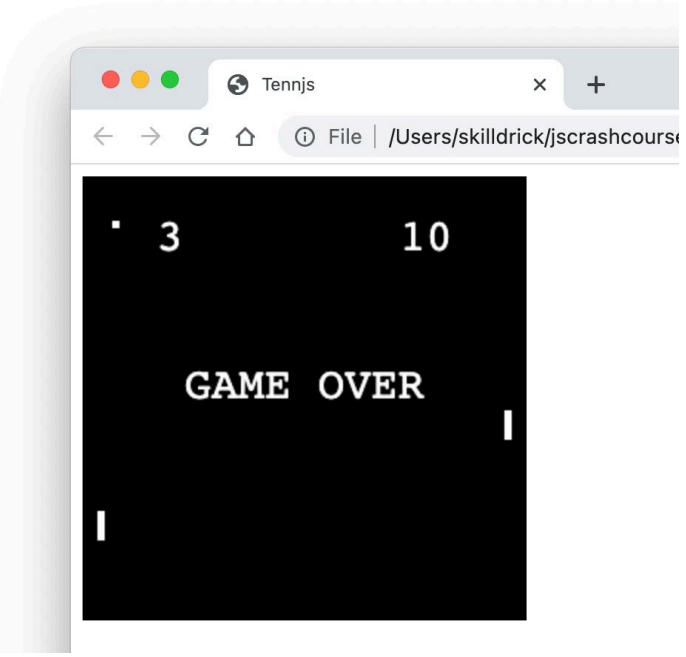


Figure 11-8 Game over

I hope you beat the computer but don't worry if you didn't—the game is pretty hard. Here are some things you can do to make it easier for yourself:

- Increase the time between frames in `gameLoop`
- Make the paddles taller
- Reduce the computer's max speed
- Make it easier to hit the edge of the paddle
- Increase the effect on `ySpeed` from hitting the edge of the paddle

Whatever you do, have fun! It's your game now.

TRY IT OUT

Now that you have a working game, you can make any changes you want. I gave some suggestions above for how to make the game easier. How about making the game harder? Here are some things to try:

- Increase the speed of the game as the scores increase (note that you could either do this by increasing the `xSpeed` and `ySpeed` of the ball, or by reducing the `setTimeout` time in `gameLoop`)
- Slow down the player's paddle—this will require something similar to the computer paddle movement, with the right paddle moving by some max amount each frame to try to reach the current mouse position
- Add a second, slower ball!

The Complete Code

For your convenience, the whole *script.js* file is shown in Listing 11-24.

```
let canvas = document.querySelector("#canvas");
let ctx = canvas.getContext("2d");
let width = canvas.width;
let height = canvas.height;

const MAX_COMPUTER_SPEED = 2;

const BALL_SIZE = 5;
let ballPosition;

let xSpeed;
let ySpeed;

function initBall() {
  ballPosition = { x: 20, y: 30 };
  xSpeed = 4;
  ySpeed = 2;
}
```

```
const PADDLE_WIDTH = 5;
const PADDLE_HEIGHT = 20;
const PADDLE_OFFSET = 10;

let leftPaddleTop = 10;
let rightPaddleTop = 30;

let leftScore = 0;
let rightScore = 0;
let gameOver = false;

document.addEventListener("mousemove", e => {
  rightPaddleTop = e.y - canvas.offsetTop;
});

function draw() {
  // Fill the canvas with black
  ctx.fillStyle = "black";
  ctx.fillRect(0, 0, width, height);

  // Everything else will be white
  ctx.fillStyle = "white";

  // Draw the ball
  ctx.fillRect(ballPosition.x, ballPosition.y, BALL_SIZE,
BALL_SIZE);

  // Draw the paddles
  ctx.fillRect(
    PADDLE_OFFSET,
    leftPaddleTop,
    PADDLE_WIDTH,
    PADDLE_HEIGHT
  );
  ctx.fillRect(
    width - PADDLE_WIDTH - PADDLE_OFFSET,
    rightPaddleTop,
    PADDLE_WIDTH,
    PADDLE_HEIGHT
  );

  // Draw scores
  ctx.font = "30px monospace";
  ctx.textAlign = "left";
  ctx.fillText(leftScore.toString(), 50, 50);
  ctx.textAlign = "right";
  ctx.fillText(rightScore.toString(), width - 50, 50);
}

function followBall() {
  let ball = {
```

```
    top: ballPosition.y,
    bottom: ballPosition.y + BALL_SIZE
  };
  let leftPaddle = {
    top: leftPaddleTop,
    bottom: leftPaddleTop + PADDLE_HEIGHT
  };

  if (ball.top < leftPaddle.top) {
    leftPaddleTop -= MAX_COMPUTER_SPEED;
  } else if (ball.bottom > leftPaddle.bottom) {
    leftPaddleTop += MAX_COMPUTER_SPEED;
  }
}

function update() {
  ballPosition.x += xSpeed;
  ballPosition.y += ySpeed;
  followBall();
}

function checkPaddleCollision(ball, paddle) {
  // check if the paddle and ball overlap vertically and
horizontal
  return (
    ball.left   < paddle.right &&
    ball.right  > paddle.left &&
    ball.top    < paddle.bottom &&
    ball.bottom > paddle.top
  );
}

function adjustAngle(distanceFromTop, distanceFromBottom) {
  if (distanceFromTop < 0) {
    // If ball hit near top of paddle, reduce ySpeed
    ySpeed -= 0.5;
  } else if (distanceFromBottom < 0) {
    // If ball hit near bottom of paddle, increase ySpeed
    ySpeed += 0.5;
  }
}

function checkCollision() {
  let ball = {
    left: ballPosition.x,
    right: ballPosition.x + BALL_SIZE,
    top: ballPosition.y,
    bottom: ballPosition.y + BALL_SIZE
  }

  let leftPaddle = {
    left: PADDLE_OFFSET,
```

```
    right: PADDLE_OFFSET + PADDLE_WIDTH,
    top: leftPaddleTop,
    bottom: leftPaddleTop + PADDLE_HEIGHT
  };

  let rightPaddle = {
    left: width - PADDLE_WIDTH - PADDLE_OFFSET,
    right: width - PADDLE_OFFSET,
    top: rightPaddleTop,
    bottom: rightPaddleTop + PADDLE_HEIGHT
  };

  if (checkPaddleCollision(ball, leftPaddle)) {
    // left paddle collision happened
    let distanceFromTop = ball.top - leftPaddle.top;
    let distanceFromBottom = leftPaddle.bottom - ball.bottom;
    adjustAngle(distanceFromTop, distanceFromBottom);
    xSpeed = Math.abs(xSpeed);
  }

  if (checkPaddleCollision(ball, rightPaddle)) {
    // right paddle collision happened
    let distanceFromTop = ball.top - rightPaddle.top;
    let distanceFromBottom = rightPaddle.bottom - ball.bottom;
    adjustAngle(distanceFromTop, distanceFromBottom);
    xSpeed = -Math.abs(xSpeed);
  }

  if (ball.left < 0) {
    rightScore++;
    initBall();
  }
  if (ball.right > width) {
    leftScore++;
    initBall();
  }
  if (leftScore > 9 || rightScore > 9) {
    gameOver = true;
  }
  if (ball.top < 0 || ball.bottom > height) {
    ySpeed = -ySpeed;
  }
}

function drawGameOver() {
  ctx.fillStyle = "white";
  ctx.font = "30px monospace";
  ctx.textAlign = "center";
  ctx.fillText("GAME OVER", width / 2, height / 2);
}

function gameLoop() {
```

```
    draw();
    update();
    checkCollision();

    if (gameOver) {
      draw();
      drawGameOver();
    } else {
      // Call this function again after a timeout
      setTimeout(gameLoop, 30);
    }
  }

  initBall();
  gameLoop();
```

Listing 11-24 The complete code

Conclusion

In this chapter you created a full game from scratch! With the knowledge you learned here, you can start creating all kinds of 2D games. The basics of game loops, collision detection, and rendering are applicable to many games. Some other games you might try implementing next are *Breakout* or *Snake*. If you need some help with the logic, there are lots of tutorials online you could follow. Have fun!