JAVASCRIPT FOR WEB DEVELOPERS

UNIT – I

WHAT IS JAVASCRIPT?

- JavaScript was introduced in 1995 as a way to add programs to web pages in the Netscape Navigator browser. The language has since been adopted by all other major graphical web browsers. It has made modern web applications possible—applications with which you can interact directly without doing a page reload for every action.
- JavaScript is also used in more traditional websites to provide various forms of interactivity and cleverness.
- It is important to note that JavaScript has almost nothing to do with the programming language named Java. The similar name was inspired by marketing considerations rather than good judgment. When JavaScript was being introduced, the Java language was being heavily marketed and was gaining popularity. Someone thought it was a good idea to try to ride along on this success. Now we are stuck with the name.

There are those who will say *terrible* things about JavaScript. Many of these things are true. When I was required to write something in JavaScript for the first time, I quickly came to despise it. It would accept almost anything I typed but interpret it in a way that was completely different from what I meant. This had a lot to do with the fact that I did not have a clue what I was doing, of course, but there is a real issue here: JavaScript is ridiculously liberal in what it allows. The idea behind this design was that it would make programming in JavaScript easier for beginners. In actuality, it mostly makes finding problems in your programs harder because the system will not point them out to you.

There have been several versions of JavaScript. ECMAScript version 3 was the widely supported version in the time of JavaScript's ascent to dominance, roughly between 2000 and 2010. During this time, work was underway on an ambitious version 4, which planned a number of radical improvements and extensions to the language. Changing a living, widely used language in such a radical way turned out to be politically difficult, and work on the version 4 was abandoned in 2008, leading to a much less ambitious version 5, which made only some uncontroversial improvements, coming out in 2009. Then in 2015 version 6 came out, a major update that included some of the ideas planned for version 4. Since then we've had new, small updates every year.

The fact that the language is evolving means that browsers have to constantly keep up, and if you're using an older browser, it may not support every feature. The language designers are careful to not make any changes that could break existing programs, so new browsers can still run old programs.

PROGRAM STRUCTURE

Expressions and statements

In Chapter 1, we made values and applied operators to them to get new values. Creating values like this is the main substance of any JavaScript program. But that substance has to be framed in a larger structure to be useful. So that's what we'll cover next.

A fragment of code that produces a value is called an *expression*. Every value that is written literally (such as 22 or "psychoanalysis") is an expression. An expression between parentheses is also an expression, as is a binary operator applied to two expressions or a unary operator applied to one.

This shows part of the beauty of a language-based interface. Expressions can contain other expressions in a way similar to how subsentences in human languages are nested—a subsentence can contain its own subsentences, and so on. This allows us to build expressions that describe arbitrarily complex computations.

If an expression corresponds to a sentence fragment, a JavaScript *statement* corresponds to a full sentence. A program is a list of statements.

The simplest kind of statement is an expression with a semicolon after it. This is a program:

edit & run code by clicking it

```
1;
!false;
```

It is a useless program, though. An expression can be content to just produce a value, which can then be used by the enclosing code. A statement stands on its own, so it amounts to something only if it affects the world. It could display something on the screen—that counts as changing the world—or it could change the internal state of the machine in a way that will affect the statements that come after it. These changes are called *side effects*. The statements in the previous example just produce the values 1 and true and then immediately throw them away. This leaves no impression on the world at all. When you run this program, nothing observable happens.

In some cases, JavaScript allows you to omit the semicolon at the end of a statement. In other cases, it has to be there, or the next line will be treated as part of the same statement. The rules for when it can be safely omitted are somewhat complex and error-prone. So in this book, every statement that needs a semicolon will always get

one. I recommend you do the same, at least until you've learned more about the subtleties of missing semicolons.

Bindings

How does a program keep an internal state? How does it remember things? We have seen how to produce new values from old values, but this does not change the old values, and the new value has to be immediately used or it will dissipate again. To catch and hold values, JavaScript provides a thing called a *binding*, or *variable*:

```
let caught = 5 * 5;
```

That's a second kind of statement. The special word (*keyword*) let indicates that this sentence is going to define a binding. It is followed by the name of the binding and, if we want to immediately give it a value, by an = operator and an expression.

The previous statement creates a binding called caught and uses it to grab hold of the number that is produced by multiplying 5 by 5.

After a binding has been defined, its name can be used as an expression. The value of such an expression is the value the binding currently holds. Here's an example:

```
let ten = 10;
console.log(ten * ten);
// → 100
```

When a binding points at a value, that does not mean it is tied to that value forever. The = operator can be used at any time on existing bindings to disconnect them from their current value and have them point to a new one.

```
let mood = "light";
console.log(mood);
```

```
// → light
mood = "dark";
console.log(mood);
// → dark
```

You should imagine bindings as tentacles, rather than boxes. They do not *contain* values; they *grasp* them—two bindings can refer to the same value. A program can access only the values that it still has a reference to. When you need to remember something, you grow a tentacle to hold on to it or you reattach one of your existing tentacles to it.

Let's look at another example. To remember the number of dollars that Luigi still owes you, you create a binding. And then when he pays back $35, you give this binding a new value.

```
let luigisDebt = 140;
luigisDebt = luigisDebt - 35;
console.log(luigisDebt);
// → 105
```

When you define a binding without giving it a value, the tentacle has nothing to grasp, so it ends in thin air. If you ask for the value of an empty binding, you'll get the value undefined.

A single let statement may define multiple bindings. The definitions must be separated by commas.

```
let one = 1, two = 2;
console.log(one + two);
// → 3
```

The words var and const can also be used to create bindings, in a way similar to let.

```
var name = "Ayda";
const greeting = "Hello ";
console.log(greeting + name);
// → Hello Ayda
```

The first, var (short for "variable"), is the way bindings were declared in pre-2015 JavaScript. I'll get back to the precise way it differs from let in the next chapter. For now, remember that it mostly does the same thing, but we'll rarely use it in this book because it has some confusing properties.

The word const stands for *constant*. It defines a constant binding, which points at the same value for as long as it lives. This is useful for bindings that give a name to a value so that you can easily refer to it later.

Binding names

Binding names can be any word. Digits can be part of binding names—catch22 is a valid name, for example—but the name must not start with a digit. A binding name may include dollar signs ($) or underscores (_) but no other punctuation or special characters.

Words with a special meaning, such as let, are *keywords*, and they may not be used as binding names. There are also a number of words that are "reserved for use" in future versions of JavaScript, which also can't be used as binding names. The full list of keywords and reserved words is rather long.

```
break case catch class const continue debugger default
delete do else enum export extends false finally for
function if implements import interface in instanceof let
new package private protected public return static super
switch this throw true try typeof var void while with yield
```

Don't worry about memorizing this list. When creating a binding produces an unexpected syntax error, see whether you're trying to define a reserved word.

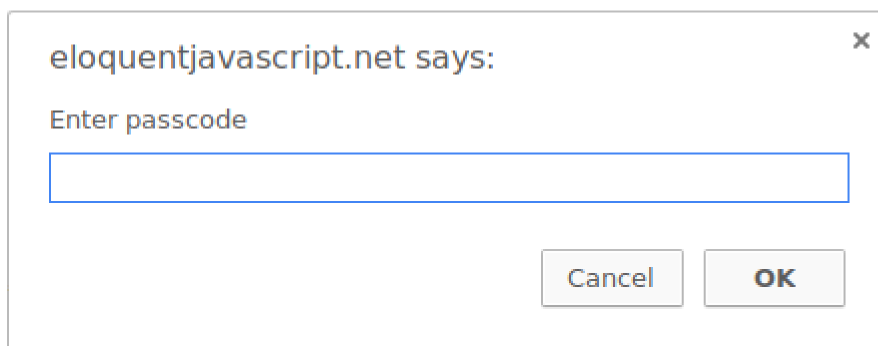
The environment

The collection of bindings and their values that exist at a given time is called the *environment*. When a program starts up, this environment is not empty. It always contains bindings that are part of the language standard, and most of the time, it also has bindings that provide ways to interact with the surrounding system. For example, in a browser, there are functions to interact with the currently loaded website and to read mouse and keyboard input.

Functions

A lot of the values provided in the default environment have the type *function*. A function is a piece of program wrapped in a value. Such values can be *applied* in order to run the wrapped program. For example, in a browser environment, the binding prompt holds a function that shows a little dialog box asking for user input. It is used like this:

```
prompt("Enter passcode");
```



Executing a function is called *invoking*, *calling*, or *applying* it. You can call a function by putting parentheses after an expression that produces a function value. Usually you'll directly use the name of the binding that holds the function. The values between the parentheses are given to the program inside the function. In the

example, the prompt function uses the string that we give it as the text to show in the dialog box. Values given to functions are called *arguments*. Different functions might need a different number or different types of arguments.

The prompt function isn't used much in modern web programming, mostly because you have no control over the way the resulting dialog looks, but can be helpful in toy programs and experiments.

The console.log function

In the examples, I used console.log to output values. Most JavaScript systems (including all modern web browsers and Node.js) provide a console.log function that writes out its arguments to *some* text output device. In browsers, the output lands in the JavaScript console. This part of the browser interface is hidden by default, but most browsers open it when you press F12 or, on a Mac, COMMAND-OPTION-I. If that does not work, search through the menus for an item named Developer Tools or similar.

When running the examples (or your own code) on the pages of this book, console.log output will be shown after the example, instead of in the browser's JavaScript console.

```
let x = 30;
console.log("the value of x is", x);
// → the value of x is 30
```

Though binding names cannot contain period characters, console.log does have one. This is because console.log isn't a simple binding. It is actually an expression that retrieves the log property from the value held by the console binding. We'll find out exactly what this means in Chapter 4.

Return values

Showing a dialog box or writing text to the screen is a *side effect*. A lot of functions are useful because of the side effects they produce. Functions may also produce values, in which case they don't need to have a side effect to be useful. For example, the function Math.max takes any amount of number arguments and gives back the greatest.

```
console.log(Math.max(2, 4));
// → 4
```

When a function produces a value, it is said to *return* that value. Anything that produces a value is an expression in JavaScript, which means function calls can be used within larger expressions. Here a call to Math.min, which is the opposite of Math.max, is used as part of a plus expression:

```
console.log(Math.min(2, 4) + 100);
// → 102
```

The next chapter explains how to write your own functions.

Control flow

When your program contains more than one statement, the statements are executed as if they are a story, from top to bottom. This example program has two statements. The first one asks the user for a number, and the second, which is executed after the first, shows the square of that number.

```
let theNumber = Number(prompt("Pick a number"));
console.log("Your number is the square root of " +
            theNumber * theNumber);
```

The function Number converts a value to a number. We need that conversion because the result of prompt is a string value, and we want a number. There are similar functions called String and Boolean that convert values to those types.

Here is the rather trivial schematic representation of straight-line control flow:

Conditional execution

Not all programs are straight roads. We may, for example, want to create a branching road, where the program takes the proper branch based on the situation at hand. This is called *conditional execution*.

Conditional execution is created with the if keyword in JavaScript. In the simple case, we want some code to be executed if, and only if, a certain condition holds. We might, for example, want to show the square of the input only if the input is actually a number.

```
let theNumber = Number(prompt("Pick a number"));
if (!Number.isNaN(theNumber)) {
  console.log("Your number is the square root of " +
              theNumber * theNumber);
}
```

With this modification, if you enter “parrot”, no output is shown.

The if keyword executes or skips a statement depending on the value of a Boolean expression. The deciding expression is written after the keyword, between parentheses, followed by the statement to execute.

The Number.isNaN function is a standard JavaScript function that returns true only if the argument it is given is NaN. The Number function happens to return NaN when you give it a string that doesn't represent a valid number. Thus, the condition translates to "unless theNumber is not-a-number, do this".

The statement after the if is wrapped in braces ({ and }) in this example. The braces can be used to group any number of statements into a single statement, called a *block*. You could also have omitted them in this case, since they hold only a single statement, but to avoid having to think about whether they are needed, most JavaScript programmers use them in every wrapped statement like this. We'll mostly follow that convention in this book, except for the occasional one-liner.

```
if (1 + 1 == 2) console.log("It's true");
// → It's true
```

You often won't just have code that executes when a condition holds true, but also code that handles the other case. This alternate path is represented by the second arrow in the diagram. You can use the else keyword, together with if, to create two separate, alternative execution paths.

```
let theNumber = Number(prompt("Pick a number"));
if (!Number.isNaN(theNumber)) {
  console.log("Your number is the square root of " +
              theNumber * theNumber);
} else {
  console.log("Hey. Why didn't you give me a number?");
}
```

If you have more than two paths to choose from, you can "chain" multiple if/else pairs together. Here's an example:

```
let num = Number(prompt("Pick a number"));

if (num < 10) {
  console.log("Small");
} else if (num < 100) {
  console.log("Medium");
} else {
  console.log("Large");
}
```

The program will first check whether num is less than 10. If it is, it chooses that branch, shows "Small", and is done. If it isn't, it takes the else branch, which itself contains a second if. If the second condition (< 100) holds, that means the number is at least 10 but below 100, and "Medium" is shown. If it doesn't, the second and last else branch is chosen.

The schema for this program looks something like this:

while and do loops

Consider a program that outputs all even numbers from 0 to 12. One way to write this is as follows:

```
console.log(0);
console.log(2);
console.log(4);
console.log(6);
console.log(8);
console.log(10);
```

```
console.log(12);
```

That works, but the idea of writing a program is to make something *less* work, not more. If we needed all even numbers less than 1,000, this approach would be unworkable. What we need is a way to run a piece of code multiple times. This form of control flow is called a *loop*.

Looping control flow allows us to go back to some point in the program where we were before and repeat it with our current program state. If we combine this with a binding that counts, we can do something like this:

```
let number = 0;
while (number <= 12) {
  console.log(number);
  number = number + 2;
}
// → 0
// → 2
//   … etcetera
```

A statement starting with the keyword while creates a loop. The word while is followed by an expression in parentheses and then a statement, much like if. The loop keeps entering that statement as long as the expression produces a value that gives true when converted to Boolean.

The number binding demonstrates the way a binding can track the progress of a program. Every time the loop repeats, number gets a value that is 2 more than its previous value. At the beginning of every repetition, it is compared with the number 12 to decide whether the program's work is finished.

As an example that actually does something useful, we can now write a program that calculates and shows the value of 2^{10} (2 to the 10th power). We use two bindings: one to keep track of our result and one to count how often we have multiplied this result by 2. The loop tests whether the second binding has reached 10 yet and, if not, updates both bindings.

```
let result = 1;
let counter = 0;
while (counter < 10) {
  result = result * 2;
  counter = counter + 1;
}
console.log(result);
// → 1024
```

The counter could also have started at 1 and checked for <= 10, but for reasons that will become apparent in Chapter 4, it is a good idea to get used to counting from 0.

A do loop is a control structure similar to a while loop. It differs only on one point: a do loop always executes its body at least once, and it starts testing whether it should stop only after that first execution. To reflect this, the test appears after the body of the loop.

```
let yourName;
do {
  yourName = prompt("Who are you?");
} while (!yourName);
console.log(yourName);
```

This program will force you to enter a name. It will ask again and again until it gets something that is not an empty string. Applying the ! operator will convert a value

to Boolean type before negating it, and all strings except "" convert to true. This means the loop continues going round until you provide a non-empty name.

Indenting Code

In the examples, I've been adding spaces in front of statements that are part of some larger statement. These spaces are not required—the computer will accept the program just fine without them. In fact, even the line breaks in programs are optional. You could write a program as a single long line if you felt like it.

The role of this indentation inside blocks is to make the structure of the code stand out. In code where new blocks are opened inside other blocks, it can become hard to see where one block ends and another begins. With proper indentation, the visual shape of a program corresponds to the shape of the blocks inside it. I like to use two spaces for every open block, but tastes differ—some people use four spaces, and some people use tab characters. The important thing is that each new block adds the same amount of space.

```
if (false != true) {
  console.log("That makes sense.");
  if (1 < 2) {
    console.log("No surprise there.");
  }
}
```

Most code editor programs (including the one in this book) will help by automatically indenting new lines the proper amount.

for loops

Many loops follow the pattern shown in the while examples. First a "counter" binding is created to track the progress of the loop. Then comes a while loop,

usually with a test expression that checks whether the counter has reached its end value. At the end of the loop body, the counter is updated to track progress.

Because this pattern is so common, JavaScript and similar languages provide a slightly shorter and more comprehensive form, the for loop.

```
for (let number = 0; number <= 12; number = number + 2) {
  console.log(number);
}
// → 0
// → 2
//   … etcetera
```

This program is exactly equivalent to the earlier even-number-printing example. The only change is that all the statements that are related to the "state" of the loop are grouped together after for.

The parentheses after a for keyword must contain two semicolons. The part before the first semicolon *initializes* the loop, usually by defining a binding. The second part is the expression that *checks* whether the loop must continue. The final part *updates* the state of the loop after every iteration. In most cases, this is shorter and clearer than a while construct.

This is the code that computes 2^{10} using for instead of while:

```
let result = 1;
for (let counter = 0; counter < 10; counter = counter + 1) {
  result = result * 2;
}
console.log(result);
// → 1024
```

Breaking Out of a Loop

Having the looping condition produce false is not the only way a loop can finish. There is a special statement called break that has the effect of immediately jumping out of the enclosing loop.

This program illustrates the break statement. It finds the first number that is both greater than or equal to 20 and divisible by 7.

```
for (let current = 20; ; current = current + 1) {
  if (current % 7 == 0) {
    console.log(current);
    break;
  }
}
// → 21
```

Using the remainder (%) operator is an easy way to test whether a number is divisible by another number. If it is, the remainder of their division is zero.

The for construct in the example does not have a part that checks for the end of the loop. This means that the loop will never stop unless the break statement inside is executed.

If you were to remove that break statement or you accidentally write an end condition that always produces true, your program would get stuck in an *infinite loop*. A program stuck in an infinite loop will never finish running, which is usually a bad thing.

If you create an infinite loop in one of the examples on these pages, you'll usually be asked whether you want to stop the script after a few seconds. If that fails, you

will have to close the tab that you're working in, or on some browsers close your whole browser, to recover.

The continue keyword is similar to break, in that it influences the progress of a loop. When continue is encountered in a loop body, control jumps out of the body and continues with the loop's next iteration.

Updating bindings succinctly

Especially when looping, a program often needs to "update" a binding to hold a value based on that binding's previous value.

```
counter = counter + 1;
```

JavaScript provides a shortcut for this.

```
counter += 1;
```

Similar shortcuts work for many other operators, such as result *= 2 to double result or counter -= 1 to count downward.

This allows us to shorten our counting example a little more.

```
for (let number = 0; number <= 12; number += 2) {
  console.log(number);
}
```

For counter += 1 and counter -= 1, there are even shorter equivalents: counter++ and counter--.

Dispatching on a value with switch

It is not uncommon for code to look like this:

```
if (x == "value1") action1();
else if (x == "value2") action2();
else if (x == "value3") action3();
else defaultAction();
```

There is a construct called switch that is intended to express such a "dispatch" in a more direct way. Unfortunately, the syntax JavaScript uses for this (which it inherited from the C/Java line of programming languages) is somewhat awkward—a chain of if statements may look better. Here is an example:

```
switch (prompt("What is the weather like?")) {
  case "rainy":
    console.log("Remember to bring an umbrella.");
    break;
  case "sunny":
    console.log("Dress lightly.");
  case "cloudy":
    console.log("Go outside.");
    break;
  default:
    console.log("Unknown weather type!");
    break;
}
```

You may put any number of case labels inside the block opened by switch. The program will start executing at the label that corresponds to the value that switch was given, or at default if no matching value is found. It will continue executing, even across other labels, until it reaches a break statement. In some cases, such as the "sunny" case in the example, this can be used to share some code between cases (it recommends going outside for both sunny and cloudy weather).

But be careful—it is easy to forget such a break, which will cause the program to execute code you do not want executed.

Capitalization

Binding names may not contain spaces, yet it is often helpful to use multiple words to clearly describe what the binding represents. These are pretty much your choices for writing a binding name with several words in it:

```
fuzzylittleturtle
fuzzy_little_turtle
FuzzyLittleTurtle
fuzzyLittleTurtle
```

The first style can be hard to read. I rather like the look of the underscores, though that style is a little painful to type. The standard JavaScript functions, and most JavaScript programmers, follow the bottom style—they capitalize every word except the first. It is not hard to get used to little things like that, and code with mixed naming styles can be jarring to read, so we follow this convention.

In a few cases, such as the Number function, the first letter of a binding is also capitalized. This was done to mark this function as a constructor. What a constructor is will become clear in Chapter 6. For now, the important thing is not to be bothered by this apparent lack of consistency.

Comments

Often, raw code does not convey all the information you want a program to convey to human readers, or it conveys it in such a cryptic way that people might not understand it. At other times, you might just want to include some related thoughts as part of your program. This is what *comments* are for.

A comment is a piece of text that is part of a program but is completely ignored by the computer. JavaScript has two ways of writing comments. To write a single-line comment, you can use two slash characters (//) and then the comment text after it.

```
let accountBalance = calculateBalance(account);
// It's a green hollow where a river sings
accountBalance.adjust();
// Madly catching white tatters in the grass.
let report = new Report();
// Where the sun on the proud mountain rings:
addToReport(accountBalance, report);
// It's a little valley, foaming like light in a glass.
```

A // comment goes only to the end of the line. A section of text between /* and */ will be ignored in its entirety, regardless of whether it contains line breaks. This is useful for adding blocks of information about a file or a chunk of program.

```
/*
  I first found this number scrawled on the back of an old
  notebook. Since then, it has often dropped by, showing up in
  phone numbers and the serial numbers of products that I've
  bought. It obviously likes me, so I've decided to keep it.
*/
const myNumber = 11213;
```

FUNCTIONS

Functions are the bread and butter of JavaScript programming. The concept of wrapping a piece of program in a value has many uses. It gives us a way to structure larger programs, to reduce repetition, to associate names with subprograms, and to isolate these subprograms from each other.

The most obvious application of functions is defining new vocabulary. Creating new words in prose is usually bad style. But in programming, it is indispensable.

Typical adult English speakers have some 20,000 words in their vocabulary. Few programming languages come with 20,000 commands built in. And the vocabulary that *is* available tends to be more precisely defined, and thus less flexible, than in human language. Therefore, we usually *have* to introduce new concepts to avoid repeating ourselves too much.

Defining a function

A function definition is a regular binding where the value of the binding is a function. For example, this code defines square to refer to a function that produces the square of a given number:

edit & run code by clicking it

```
const square = function(x) {
  return x * x;
};

console.log(square(12));
// → 144
```

A function is created with an expression that starts with the keyword function. Functions have a set of *parameters* (in this case, only x) and a *body*, which contains the statements that are to be executed when the function is called. The function

body of a function created this way must always be wrapped in braces, even when it consists of only a single statement.

A function can have multiple parameters or no parameters at all. In the following example, makeNoise does not list any parameter names, whereas power lists two:

```
const makeNoise = function() {
  console.log("Pling!");
};

makeNoise();
// → Pling!

const power = function(base, exponent) {
  let result = 1;
  for (let count = 0; count < exponent; count++) {
    result *= base;
  }
  return result;
};

console.log(power(2, 10));
// → 1024
```

Some functions produce a value, such as power and square, and some don't, such as makeNoise, whose only result is a side effect. A return statement determines the value the function returns. When control comes across such a statement, it immediately jumps out of the current function and gives the returned value to the code that called the function. A return keyword without an expression after it will

cause the function to return undefined. Functions that don't have a return statement at all, such as makeNoise, similarly return undefined.

Parameters to a function behave like regular bindings, but their initial values are given by the *caller* of the function, not the code in the function itself.

Bindings and scopes

Each binding has a *scope*, which is the part of the program in which the binding is visible. For bindings defined outside of any function or block, the scope is the whole program—you can refer to such bindings wherever you want. These are called *global*.

But bindings created for function parameters or declared inside a function can be referenced only in that function, so they are known as *local* bindings. Every time the function is called, new instances of these bindings are created. This provides some isolation between functions—each function call acts in its own little world (its local environment) and can often be understood without knowing a lot about what's going on in the global environment.

Bindings declared with let and const are in fact local to the *block* that they are declared in, so if you create one of those inside of a loop, the code before and after the loop cannot "see" it. In pre-2015 JavaScript, only functions created new scopes, so old-style bindings, created with the var keyword, are visible throughout the whole function that they appear in—or throughout the global scope, if they are not in a function.

```
let x = 10;
if (true) {
 let y = 20;
 var z = 30;
```

```
  console.log(x + y + z);
  // → 60
}
// y is not visible here
console.log(x + z);
// → 40
```

Each scope can "look out" into the scope around it, so x is visible inside the block in the example. The exception is when multiple bindings have the same name—in that case, code can see only the innermost one. For example, when the code inside the halve function refers to n, it is seeing its *own* n, not the global n.

```
const halve = function(n) {
  return n / 2;
};

let n = 10;
console.log(halve(100));
// → 50
console.log(n);
// → 10
```

Nested scope

JavaScript distinguishes not just *global* and *local* bindings. Blocks and functions can be created inside other blocks and functions, producing multiple degrees of locality.

For example, this function—which outputs the ingredients needed to make a batch of hummus—has another function inside it:

```
const hummus = function(factor) {
  const ingredient = function(amount, unit, name) {
```

```
    let ingredientAmount = amount * factor;
    if (ingredientAmount > 1) {
      unit += "s";
    }
    console.log(`${ingredientAmount} ${unit} ${name}`);
  };
  ingredient(1, "can", "chickpeas");
  ingredient(0.25, "cup", "tahini");
  ingredient(0.25, "cup", "lemon juice");
  ingredient(1, "clove", "garlic");
  ingredient(2, "tablespoon", "olive oil");
  ingredient(0.5, "teaspoon", "cumin");
};
```

The code inside the ingredient function can see the factor binding from the outer function. But its local bindings, such as unit or ingredientAmount, are not visible in the outer function.

The set of bindings visible inside a block is determined by the place of that block in the program text. Each local scope can also see all the local scopes that contain it, and all scopes can see the global scope. This approach to binding visibility is called *lexical scoping*.

Functions as values

A function binding usually simply acts as a name for a specific piece of the program. Such a binding is defined once and never changed. This makes it easy to confuse the function and its name.

But the two are different. A function value can do all the things that other values can do—you can use it in arbitrary expressions, not just call it. It is possible to store

a function value in a new binding, pass it as an argument to a function, and so on. Similarly, a binding that holds a function is still just a regular binding and can, if not constant, be assigned a new value, like so:

```
let launchMissiles = function() {
  missileSystem.launch("now");
};
if (safeMode) {
  launchMissiles = function() {/* do nothing */};
}
```

In Chapter 5, we will discuss the interesting things that can be done by passing around function values to other functions.

Declaration notation

There is a slightly shorter way to create a function binding. When the function keyword is used at the start of a statement, it works differently.

```
function square(x) {
  return x * x;
}
```

This is a function *declaration*. The statement defines the binding square and points it at the given function. It is slightly easier to write and doesn't require a semicolon after the function.

There is one subtlety with this form of function definition.

```
console.log("The future says:", future());

function future() {
```

```
  return "You'll never have flying cars";
}
```

The preceding code works, even though the function is defined *below* the code that uses it. Function declarations are not part of the regular top-to-bottom flow of control. They are conceptually moved to the top of their scope and can be used by all the code in that scope. This is sometimes useful because it offers the freedom to order code in a way that seems meaningful, without worrying about having to define all functions before they are used.

Arrow functions

There’s a third notation for functions, which looks very different from the others. Instead of the function keyword, it uses an arrow (=>) made up of an equal sign and a greater-than character (not to be confused with the greater-than-or-equal operator, which is written >=).

```
const power = (base, exponent) => {
  let result = 1;
  for (let count = 0; count < exponent; count++) {
    result *= base;
  }
  return result;
};
```

The arrow comes *after* the list of parameters and is followed by the function’s body. It expresses something like “this input (the parameters) produces this result (the body)”.

When there is only one parameter name, you can omit the parentheses around the parameter list. If the body is a single expression, rather than a block in braces, that

expression will be returned from the function. So, these two definitions of square do the same thing:

```
const square1 = (x) => { return x * x; };
const square2 = x => x * x;
```

When an arrow function has no parameters at all, its parameter list is just an empty set of parentheses.

```
const horn = () => {
  console.log("Toot");
};
```

There's no deep reason to have both arrow functions and function expressions in the language. Apart from a minor detail, which we'll discuss in Chapter 6, they do the same thing. Arrow functions were added in 2015, mostly to make it possible to write small function expressions in a less verbose way. We'll be using them a lot in Chapter 5.

The call stack

The way control flows through functions is somewhat involved. Let's take a closer look at it. Here is a simple program that makes a few function calls:

```
function greet(who) {
  console.log("Hello " + who);
}
greet("Harry");
console.log("Bye");
```

A run through this program goes roughly like this: the call to greet causes control to jump to the start of that function (line 2). The function calls console.log, which

takes control, does its job, and then returns control to line 2. There it reaches the end of the greet function, so it returns to the place that called it, which is line 4. The line after that calls console.log again. After that returns, the program reaches its end.

We could show the flow of control schematically like this:

```
not in function
  in greet
       in console.log
  in greet
not in function
  in console.log
not in function
```

Because a function has to jump back to the place that called it when it returns, the computer must remember the context from which the call happened. In one case, console.log has to return to the greet function when it is done. In the other case, it returns to the end of the program.

The place where the computer stores this context is the *call stack*. Every time a function is called, the current context is stored on top of this stack. When a function returns, it removes the top context from the stack and uses that context to continue execution.

Storing this stack requires space in the computer's memory. When the stack grows too big, the computer will fail with a message like "out of stack space" or "too much recursion". The following code illustrates this by asking the computer a really hard question that causes an infinite back-and-forth between two functions. Rather, it *would* be infinite, if the computer had an infinite stack. As it is, we will run out of space, or "blow the stack".

```
function chicken() {
  return egg();
}
function egg() {
  return chicken();
}
console.log(chicken() + " came first.");
// → ??
```

Optional Arguments

The following code is allowed and executes without any problem:

```
function square(x) { return x * x; }
console.log(square(4, true, "hedgehog"));
// → 16
```

We defined square with only one parameter. Yet when we call it with three, the language doesn't complain. It ignores the extra arguments and computes the square of the first one.

JavaScript is extremely broad-minded about the number of arguments you pass to a function. If you pass too many, the extra ones are ignored. If you pass too few, the missing parameters get assigned the value undefined.

The downside of this is that it is possible—likely, even—that you'll accidentally pass the wrong number of arguments to functions. And no one will tell you about it.

The upside is that this behavior can be used to allow a function to be called with different numbers of arguments. For example, this minus function tries to imitate the - operator by acting on either one or two arguments:

```
function minus(a, b) {
  if (b === undefined) return -a;
  else return a - b;
}

console.log(minus(10));
// → -10
console.log(minus(10, 5));
// → 5
```

If you write an = operator after a parameter, followed by an expression, the value of that expression will replace the argument when it is not given.

For example, this version of power makes its second argument optional. If you don't provide it or pass the value undefined, it will default to two, and the function will behave like square.

```
function power(base, exponent = 2) {
  let result = 1;
  for (let count = 0; count < exponent; count++) {
    result *= base;
  }
  return result;
}

console.log(power(4));
// → 16
console.log(power(2, 6));
// → 64
```

In the next chapter, we will see a way in which a function body can get at the whole list of arguments it was passed. This is helpful because it makes it possible for a function to accept any number of arguments. For example, console.log does this—it outputs all of the values it is given.

```
console.log("C", "O", 2);
// → C O 2
```

Closure

The ability to treat functions as values, combined with the fact that local bindings are re-created every time a function is called, brings up an interesting question. What happens to local bindings when the function call that created them is no longer active?

The following code shows an example of this. It defines a function, wrapValue, that creates a local binding. It then returns a function that accesses and returns this local binding.

```
function wrapValue(n) {
  let local = n;
  return () => local;
}

let wrap1 = wrapValue(1);
let wrap2 = wrapValue(2);
console.log(wrap1());
// → 1
console.log(wrap2());
// → 2
```

This is allowed and works as you'd hope—both instances of the binding can still be accessed. This situation is a good demonstration of the fact that local bindings are created anew for every call, and different calls can't trample on one another's local bindings.

This feature—being able to reference a specific instance of a local binding in an enclosing scope—is called *closure*. A function that references bindings from local scopes around it is called *a* closure. This behavior not only frees you from having to worry about lifetimes of bindings but also makes it possible to use function values in some creative ways.

With a slight change, we can turn the previous example into a way to create functions that multiply by an arbitrary amount.

```
function multiplier(factor) {
  return number => number * factor;
}

let twice = multiplier(2);
console.log(twice(5));
// → 10
```

The explicit local binding from the wrapValue example isn't really needed since a parameter is itself a local binding.

Thinking about programs like this takes some practice. A good mental model is to think of function values as containing both the code in their body and the environment in which they are created. When called, the function body sees the environment in which it was created, not the environment in which it is called.

In the example, multiplier is called and creates an environment in which its factor parameter is bound to 2. The function value it returns, which is stored in twice, remembers this environment. So when that is called, it multiplies its argument by 2.

Recursion

It is perfectly okay for a function to call itself, as long as it doesn't do it so often that it overflows the stack. A function that calls itself is called *recursive*. Recursion allows some functions to be written in a different style. Take, for example, this alternative implementation of power:

```
function power(base, exponent) {
  if (exponent == 0) {
    return 1;
  } else {
    return base * power(base, exponent - 1);
  }
}

console.log(power(2, 3));
// → 8
```

This is rather close to the way mathematicians define exponentiation and arguably describes the concept more clearly than the looping variant. The function calls itself multiple times with ever smaller exponents to achieve the repeated multiplication.

But this implementation has one problem: in typical JavaScript implementations, it's about three times slower than the looping version. Running through a simple loop is generally cheaper than calling a function multiple times.

The dilemma of speed versus elegance is an interesting one. You can see it as a kind of continuum between human-friendliness and machine-friendliness. Almost any program can be made faster by making it bigger and more convoluted. The programmer has to decide on an appropriate balance.

In the case of the power function, the inelegant (looping) version is still fairly simple and easy to read. It doesn't make much sense to replace it with the recursive version. Often, though, a program deals with such complex concepts that giving up some efficiency in order to make the program more straightforward is helpful.

Worrying about efficiency can be a distraction. It's yet another factor that complicates program design, and when you're doing something that's already difficult, that extra thing to worry about can be paralyzing.

Therefore, always start by writing something that's correct and easy to understand. If you're worried that it's too slow—which it usually isn't since most code simply isn't executed often enough to take any significant amount of time—you can measure afterward and improve it if necessary.

Recursion is not always just an inefficient alternative to looping. Some problems really are easier to solve with recursion than with loops. Most often these are problems that require exploring or processing several "branches", each of which might branch out again into even more branches.

Consider this puzzle: by starting from the number 1 and repeatedly either adding 5 or multiplying by 3, an infinite set of numbers can be produced. How would you write a function that, given a number, tries to find a sequence of such additions and multiplications that produces that number?

For example, the number 13 could be reached by first multiplying by 3 and then adding 5 twice, whereas the number 15 cannot be reached at all.

Here is a recursive solution:

```
function findSolution(target) {
  function find(current, history) {
    if (current == target) {
      return history;
    } else if (current > target) {
      return null;
    } else {
      return find(current + 5, `(${history} + 5)`) ||
             find(current * 3, `(${history} * 3)`);
    }
  }
  return find(1, "1");
}

console.log(findSolution(24));
// → (((1 * 3) + 5) * 3)
```

Note that this program doesn't necessarily find the *shortest* sequence of operations. It is satisfied when it finds any sequence at all.

It is okay if you don't see how it works right away. Let's work through it, since it makes for a great exercise in recursive thinking.

The inner function find does the actual recursing. It takes two arguments: the current number and a string that records how we reached this number. If it finds a solution, it returns a string that shows how to get to the target. If no solution can be found starting from this number, it returns null.

To do this, the function performs one of three actions. If the current number is the target number, the current history is a way to reach that target, so it is returned. If the current number is greater than the target, there's no sense in further exploring this branch because both adding and multiplying will only make the number bigger, so it returns null. Finally, if we're still below the target number, the function tries both possible paths that start from the current number by calling itself twice, once for addition and once for multiplication. If the first call returns something that is not null, it is returned. Otherwise, the second call is returned, regardless of whether it produces a string or null.

To better understand how this function produces the effect we're looking for, let's look at all the calls to find that are made when searching for a solution for the number 13.

```
find(1, "1")
  find(6, "(1 + 5)")
    find(11, "((1 + 5) + 5)")
      find(16, "(((1 + 5) + 5) + 5)")
        too big
      find(33, "(((1 + 5) + 5) * 3)")
        too big
    find(18, "((1 + 5) * 3)")
      too big
  find(3, "(1 * 3)")
    find(8, "((1 * 3) + 5)")
      find(13, "(((1 * 3) + 5) + 5)")
        found!
```

The indentation indicates the depth of the call stack. The first time find is called, it starts by calling itself to explore the solution that starts with (1 + 5). That call will

further recurse to explore *every* continued solution that yields a number less than or equal to the target number. Since it doesn't find one that hits the target, it returns null back to the first call. There the || operator causes the call that explores (1 * 3) to happen. This search has more luck—its first recursive call, through yet *another* recursive call, hits upon the target number. That innermost call returns a string, and each of the || operators in the intermediate calls passes that string along, ultimately returning the solution.