Lesson 4

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:

```
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 only be referenced 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 only see 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.

In short, each local scope can also see all the local scopes that contain it. 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 equals and greater-than characters (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 very good reason to have both arrow functions and functionexpressions 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 amounts 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 afterwards 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 amount of new 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. And

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.

Growing functions

There are two more or less natural ways for functions to be introduced into programs.

The first is that you find yourself writing very similar code multiple times. We'd prefer not to do that. Having more code means more space for mistakes to hide and more material to read for people trying to understand the program. So we take the repeated functionality, find a good name for it, and put it into a function.

The second way is that you find you need some functionality that you haven't written yet and that sounds like it deserves its own function. You'll start by naming the function, and then you'll write its body. You might even start writing code that uses the function before you actually define the function itself.

How difficult it is to find a good name for a function is a good indication of how clear a concept it is that you're trying to wrap. Let's go through an example.

We want to write a program that prints two numbers, the numbers of cows and chickens on a farm, with the words Cows and Chickens after them, and zeros padded before both numbers so that they are always three digits long.

```
007 Cows
011 Chickens
```

This asks for a function of two arguments—the number of cows and the number of chickens. Let's get coding.

```
function printFarmInventory(cows, chickens) {
  let cowString = String(cows);
  while (cowString.length < 3) {
```

```
    cowString = "0" + cowString;
  }
  console.log(`${cowString} Cows`);
  let chickenString = String(chickens);
  while (chickenString.length < 3) {
    chickenString = "0" + chickenString;
  }
  console.log(`${chickenString} Chickens`);
}
printFarmInventory(7, 11);
```

Writing .length after a string expression will give us the length of that string. Thus, the while loops keep adding zeros in front of the number strings until they are at least three characters long.

Mission accomplished! But just as we are about to send the farmer the code (along with a hefty invoice), she calls and tells us she's also started keeping pigs, and couldn't we please extend the software to also print pigs?

We sure can. But just as we're in the process of copying and pasting those four lines one more time, we stop and reconsider. There has to be a better way. Here's a first attempt:

```
function printZeroPaddedWithLabel(number, label) {
  let numberString = String(number);
  while (numberString.length < 3) {
    numberString = "0" + numberString;
  }
  console.log(`${numberString} ${label}`);
}

function printFarmInventory(cows, chickens, pigs) {
  printZeroPaddedWithLabel(cows, "Cows");
```

```
  printZeroPaddedWithLabel(chickens, "Chickens");
  printZeroPaddedWithLabel(pigs, "Pigs");
}

printFarmInventory(7, 11, 3);
```

It works! But that name, printZeroPaddedWithLabel, is a little awkward. It conflates three things—printing, zero-padding, and adding a label—into a single function.

Instead of lifting out the repeated part of our program wholesale, let's try to pick out a single *concept*.

```
function zeroPad(number, width) {
  let string = String(number);
  while (string.length < width) {
    string = "0" + string;
  }
  return string;
}

function printFarmInventory(cows, chickens, pigs) {
  console.log(`${zeroPad(cows, 3)} Cows`);
  console.log(`${zeroPad(chickens, 3)} Chickens`);
  console.log(`${zeroPad(pigs, 3)} Pigs`);
}

printFarmInventory(7, 16, 3);
```

A function with a nice, obvious name like zeroPad makes it easier for someone who reads the code to figure out what it does. And such a function is useful in more situations than just this specific program. For example, you could use it to help print nicely aligned tables of numbers.

How smart and versatile *should* our function be? We could write anything, from a terribly simple function that can only pad a number to be three characters wide, to a complicated generalized number-formatting system that handles fractional numbers, negative numbers, alignment of decimal dots, padding with different characters, and so on.

A useful principle is to not add cleverness unless you are absolutely sure you're going to need it. It can be tempting to write general "frameworks" for every bit of functionality you come across. Resist that urge. You won't get any real work done—you'll just be writing code that you never use.

Functions and side effects

Functions can be roughly divided into those that are called for their side effects and those that are called for their return value. (Though it is definitely also possible to both have side effects and return a value.)

The first helper function in the farm example, `printZeroPaddedWithLabel`, is called for its side effect: it prints a line. The second version, `zeroPad`, is called for its return value. It is no coincidence that the second is useful in more situations than the first. Functions that create values are easier to combine in new ways than functions that directly perform side effects.

A *pure* function is a specific kind of value-producing function that not only has no side effects but also doesn't rely on side effects from other code—for example, it doesn't read global bindings whose value might change. A pure function has the pleasant property that, when called with the same arguments, it always produces the same value (and doesn't do anything else). A call to such a function can be substituted by its return value without changing the meaning of the code. When you are not sure that a pure function is working correctly, you can test it by simply calling it, and know that if it works in that context, it will work in any context. Nonpure functions tend to require more scaffolding to test.

Still, there's no need to feel bad when writing functions that are not pure or to wage a holy war to purge them from your code. Side effects are often useful. There'd be no way to write a pure version of `console.log`, for example, and `console.log` is good to have. Some operations are also easier to express in an efficient way when we use side effects, so computing speed can be a reason to avoid purity.

Exercises

Minimum

The introduced the standard function `Math.min` that returns its smallest argument. We can build something like that now. Write a function `min` that takes two arguments and returns their minimum.

```
// Your code here.

console.log(min(0, 10));
// → 0
console.log(min(0, -10));
// → -10
```

Recursion

We've seen that `%` (the remainder operator) can be used to test whether a number is even or odd by using `% 2` to see whether it's divisible by two. Here's another way to define whether a positive whole number is even or odd:

- Zero is even.
- One is odd.
- For any other number *N*, its evenness is the same as *N* - 2.

Define a recursive function isEven corresponding to this description. The function should accept a single parameter (a positive, whole number) and return a Boolean.
Test it on 50 and 75. See how it behaves on -1. Why? Can you think of a way to fix this?

```
// Your code here.

console.log(isEven(50));
// → true
console.log(isEven(75));
// → false
console.log(isEven(-1));
// → ??
```

Bean counting

You can get the Nth character, or letter, from a string by writing "string"[N]. The returned value will be a string containing only one character (for example, "b"). The first character has position zero, which causes the last one to be found at position string.length - 1. In other words, a two-character string has length 2, and its characters have positions 0 and 1.
Write a function countBs that takes a string as its only argument and returns a number that indicates how many uppercase “B” characters there are in the string.
Next, write a function called countChar that behaves like countBs, except it takes a second argument that indicates the character that is to be counted (rather than counting only uppercase “B” characters). Rewrite countBs to make use of this new function.

```
// Your code here.
```

```
console.log(countBs("BBC"));
// → 2
console.log(countChar("kakkerlak", "k"));
// → 4
```