CHAPTER 3
FUNCTIONS

“People think that computer science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on each other, like a wall of mini stones.

—Donald Knuth

You’ve seen function values, such as `alert`, and how to call them. Functions are the bread and butter of JavaScript programming. The concept of wrapping a piece of program in a value has many uses. It is a tool 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 regular, human-language 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 add some of our own vocabulary to avoid repeating ourselves too much.

DEFINING A FUNCTION

A function definition is just a regular variable definition where the value given to the variable happens to be a function. For example, the following code defines the variable `square` to refer to a function that produces the square of a given number:

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

```
console.log(square(12));
// → 144
```

A function is created by 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 must always be wrapped in braces, even when it consists of only a single statement (as in the previous example).

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:

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

makeNoise();
// → Pling!

var power = function(base, exponent) {
  var result = 1;
  for (var 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`, which produces only 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. The `return` keyword without an expression after it will cause the function to return `undefined`.

Parameters and scopes

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

An important property of functions is that the variables created inside of them, including their parameters, are *local* to the function. This means, for example, that the `result` variable in the `power` example will be newly created every time the function is called, and these separate incarnations do not interfere with each other.

This "localness" of variables applies only to the parameters and variables declared with the `var` keyword inside the function body. Variables declared outside of any function are called *global*, because they are visible throughout the program. It is possible to access such variables from inside a function, as long as you haven't declared a local variable with the same name.

The following code demonstrates this. It defines and calls two functions that both assign a value to the variable `x`. The first one declares the variable as local and thus changes only the local variable. The second does not declare `x` locally, so references to `x` inside of it refer to the global variable `x` defined at the top of the example.

```
var x = "outside";

var f1 = function() {
  var x = "inside f1";
};
f1();
console.log(x);
// → outside

var f2 = function() {
  x = "inside f2";
};
f2();
```

```
console.log(x);
// → inside f2
```

This behavior helps prevent accidental interference between functions. If all variables were shared by the whole program, it'd take a lot of effort to make sure no name is ever used for two different purposes. And if you *did* reuse a variable name, you might see strange effects from unrelated code messing with the value of your variable. By treating function-local variables as existing only within the function, the language makes it possible to read and understand functions as small universes, without having to worry about all the code at once.

NESTED SCOPE

JavaScript distinguishes not just between *global* and *local* variables. Functions can be created inside other functions, producing several degrees of locality.

For example, this rather nonsensical function has two functions inside of it:

```
var landscape = function() {
  var result = "";
  var flat = function(size) {
    for (var count = 0; count < size; count++)
      result += "_";
  };
  var mountain = function(size) {
    result += "/";
    for (var count = 0; count < size; count++)
      result += "'";
    result += "\\";
  };

  flat(3);
  mountain(4);
  flat(6);
  mountain(1);
  flat(1);
```

```
    return result;
  };

  console.log(landscape());
  // → ___/''''\______/'\_
```

The `flat` and `mountain` functions can “see” the variable called `result`, since they are inside the function that defines it. But they cannot see each other’s `count` variables since they are outside each other’s scope. The environment outside of the `landscape` function doesn’t see any of the variables defined inside `landscape`.

In short, each local scope can also see all the local scopes that contain it. The set of variables visible inside a function is determined by the place of that function in the program text. All variables from blocks “around” a function’s definition are visible—meaning both those in function bodies that enclose it and those at the top level of the program. This approach to variable visibility is called *lexical scoping*.

People who have experience with other programming languages might expect that any block of code between braces produces a new local environment. But in JavaScript, functions are the only things that create a new scope. You are allowed to use free-standing blocks.

```
var something = 1;
{
  var something = 2;
  // Do stuff with variable something...
}
// Outside of the block again...
```

But the `something` inside the block refers to the same variable as the one outside the block. In fact, although blocks like this are allowed, they are useful only to group the body of an `if` statement or a loop.

If you find this odd, you’re not alone. The next version of JavaScript will introduce a `let` keyword, which works like `var` but creates a variable that is local to the enclosing *block*, not the enclosing *function*.

Functions as values

Usually, function variables simply act as names for a specific piece of the program. The variables are defined once and never change. This makes it easy to start confusing 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 all kinds of expressions, not just call it. It is possible to store a function value in a new place, pass it as an argument to a function, and so on. Similarly, a variable that holds a function is still just a regular variable and can be assigned a new value, like so:

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

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

Declaration notation

There is a slightly shorter way to say “`var square = function…`”. The `` `function `` keyword can also be used at the start of a statement, as in the following:

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

This is a function *declaration*. The statement defines the variable `square` and points it at the given function. So far so good. There is one subtlety with this form of function definition, however.

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

function future() {
  return "We STILL have no flying cars.";
}
```

This code works, even though the function is defined *below* the code that first uses it. This is because 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 gives us the freedom to order code in a way that seems meaningful, without worrying about having to define all functions above their first use.

What happens when you put such a function definition inside a conditional (`if`) block or a loop? Well, don't do that. Different JavaScript platforms in different browsers have traditionally done different things in that situation, and the latest standard actually forbids it. If you want your programs to behave consistently, only use this form of function-defining statements in the outermost block of a function or program.

```
function example() {
  function a() {} // Okay
  if (something) {
    function b() {} // Danger!
  }
}
```

The call stack

It will be helpful to take a closer look at the way control flows through functions. 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). It calls `console.log` (a built-in browser function), which takes control, does its job, and then returns control to line 2. Then it reaches the end of the `greet` function, so it returns to the place that called it, at line 4. The line after that calls `console.log` again.

We could show the flow of control schematically like this:

```
top
   greet
        console.log
   greet
top
   console.log
top
```

Because a function has to jump back to the place of the call when it returns, the computer must remember the context from which the function was called. In one case, `console.log` had to jump back to the `greet` function. In the other case, it jumps back 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 put on top of this “stack”. When the function returns, it takes the top context from the stack and uses it 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, which 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:

```
alert("Hello", "Good Evening", "How do you do?");
```

The function `alert` officially accepts only one argument. Yet when you call it like this, it doesn't complain. It simply ignores the other arguments and shows you "Hello".

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 simply 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 have a function take "optional" arguments. For example, the following version of `power` can be called either with two arguments or with a single argument, in which case the exponent is assumed to be two, and the function behaves like `square`.

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

console.log(power(4));
// → 16
console.log(power(4, 3));
// → 64
```

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

```
console.log("R", 2, "D", 2);
// → R 2 D 2
```

Closure

The ability to treat functions as values, combined with the fact that local variables are "re-created" every time a function is called, brings up an interesting question. What happens to local variables 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 variable. It then returns a function that accesses this local variable, returning its value when it is called.

```
function wrapValue(n) {
  var localVariable = n;
  return function(){ return localVariable; };
}

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

```
console.log(wrap2());
// → 2
```

This is allowed and works as you'd hope—the variable can still be accessed. In fact, multiple instances of the variable can be alive at the same time, which is another good illustration of the concept that local variables really are re-created for every call—different calls can't trample on one another's local variables.

This feature—being able to reference a specific instance of local variables in an enclosing function—is called *closure*. A function that "closes over" some local variables is called *a* closure. This behavior not only frees you from having to worry about lifetimes of variables but also allows for some creative use of function values.

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 function(number) {
    return number * factor;
  };
}

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

The explicit `localVariable` from the `wrapValue` example isn't needed since a parameter is itself a local variable.

Thinking about programs like this takes some exercise. A good mental model is to think of the `function` keyword as "freezing" the code in its body and wrapping it into a package (the function value). So when you read `return function(...) {...}`, think of it as returning a handle to a piece of computation being frozen for later use.

In the example, `multiplier` returns a frozen chunk of code that gets stored in the `twice` variable. The last line then calls the value in this variable, causing the frozen code (`return number * factor;`) to be activated. It still has access to the `factor` variable from the `multiplier` call that created it, and in addition it gets access to the argument passed when unfreezing it, 5, through its `number` parameter.

Recursion

It is perfectly okay for a function to call itself. A function that calls itself is called *recursive*. Recursion allows some functions to be written in a funny way. 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 in a more elegant way than the looping variant does. The function calls itself multiple times with different arguments to achieve the repeated multiplication.

But this implementation has one important problem: in typical JavaScript implementations, it's about 10 times slower than the looping version. Running through a simple loop is a lot cheaper than calling a function multiple times. On top of that, using a sufficiently large exponent to this function might cause the stack to overflow.

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 may decide on an appropriate balance.

In the case of the earlier `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 becomes an attractive choice.

The basic rule, which has been repeated by many programmers and with which I wholeheartedly agree, is to not worry about efficiency until you know for sure that the program is too slow. If it is, find out which parts are taking up the most time, and start exchanging elegance for efficiency in those parts.

Of course, this rule doesn't mean one should start ignoring performance altogether. In many cases, like the `power` function, not much simplicity is gained from the "elegant" approach. And sometimes an experienced programmer can see right away that a simple approach is never going to be fast enough.

The reason I'm stressing this is that surprisingly many beginning programmers focus fanatically on efficiency, even in the smallest details. The result is bigger, more complicated, and often less correct programs, that take longer to write than their more straightforward equivalents and that run only marginally faster.

But recursion is not always just a less-efficient alternative to looping. Some problems are much 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 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 produce 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(start, history) {
    if (start == target)
      return history;
    else if (start > target)
      return null;
    else
      return find(start + 5, "(" + history + " + 5)") ||
             find(start * 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.

I don't necessarily expect you to see how it works right away. But 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—and returns either a string that shows how to get to the target or `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 simply returned. If the current number is greater than the target, there's no sense in further exploring this history since both adding and multiplying will only make the number bigger. And finally, if we're still below the target, the function tries both possible paths that start from the current number, by calling itself twice, once for each of the allowed next steps. 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 suggests the depth of the call stack. The first call to `find` calls itself twice to explore the solutions that start with `(1 + 5)` and `(1 * 3)`. The first call tries to find a solution that starts with `(1 + 5)` and, using recursion, explores *every* solution that yields a number smaller than or equal to the target number. Since it doesn't find a solution 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 because its first recursive call, through yet *another* recursive call, hits upon the target number, 13. This innermost recursive call returns a string, and each of the `||` operators in the intermediate calls pass that string along, ultimately returning our 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 want to avoid doing that since 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 you'll then 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 zeroes padded before both numbers so that they are always three digits long.

```
007 Cows
011 Chickens
```

That clearly asks for a function of two arguments. Let's get coding.

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

Adding `.length` after a string value will give us the length of that string. Thus, the `while` loops keep adding zeroes 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, of course), he calls and tells us he'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) {
  var 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) {
  var 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);
```

`zeroPad` has a nice, obvious name, which makes it easier for someone who reads the code to figure out what it does. And it 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 simply pads a number so that it's three characters wide to a complicated generalized number-formatting system that handles fractional numbers, negative numbers, alignment of dots, padding with different characters, and so on.

A useful principle is not to add cleverness unless you are absolutely sure you're going to need it. It can be tempting to write general "frameworks" for every little bit of functionality you come across. Resist that urge. You won't get any real work done, and you'll end up writing a lot of code that no one will ever 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's definitely also possible to have both 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 variables that are occasionally changed by other code. 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). This makes it easy to reason about. A call to such a function can be mentally substituted by its result, 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 might return different values based on all kinds of factors and have side effects that might be hard to test and think about.

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 certainly useful. 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.

Summary

This chapter taught you how to write your own functions. The `function` keyword, when used as an expression, can create a function value. When used as a statement, it can be used to declare a variable and give it a function as its value.

```
// Create a function value f
```

```
var f = function(a) {
  console.log(a + 2);
};

// Declare g to be a function
function g(a, b) {
  return a * b * 3.5;
}
```

A key aspect in understanding functions is understanding local scopes. Parameters and variables declared inside a function are local to the function, re-created every time the function is called, and not visible from the outside. Functions declared inside another function have access to the outer function's local scope.

Separating the different tasks your program performs into different functions is helpful. You won't have to repeat yourself so much, and they can help someone trying to read your program by grouping the code into conceptual chunks, in the same way that chapters and sections help organize regular text.

Exercises

Minimum

The previous chapter introduced the standard function `Math.min` that returns its smallest argument. We can do that ourselves 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
```

» Display hints...

Recursion

We've seen that `%` (the remainder operator) can be used to test whether a number is even or odd by using `% 2` to check 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 `number` parameter 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));
// → ??
```

» Display hints...

Bean counting

You can get the Nth character, or letter, from a string by writing `"string".charAt(N)`, similar to how you get its length with `"s".length`. 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 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
```

» Display hints...