> *"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

CHAPTER 3

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:

```
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.

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.