Chapter 8. Functions

This chapter covers JavaScript functions. Functions are a fundamental building block for JavaScript programs and a common feature in almost all programming languages. You may already be familiar with the concept of a function under a name such as *subroutine* or *procedure*.

A *function* is a block of JavaScript code that is defined once but may be executed, or *invoked*, any number of times. JavaScript functions are *parameterized*: a function definition may include a list of identifiers, known as *parameters*, that work as local variables for the body of the function. Function invocations provide values, or *arguments*, for the function's parameters. Functions often use their argument values to compute a *return value* that becomes the value of the function-invocation expression. In addition to the arguments, each invocation has another value—the *invocation context*—that is the value of the `this` keyword.

If a function is assigned to a property of an object, it is known as a *method* of that object. When a function is invoked *on* or *through* an object, that object is the invocation context or `this` value for the function. Functions designed to initialize a newly created object are called *constructors*. Constructors were described in §6.2 and will be covered again in Chapter 9.

In JavaScript, functions are objects, and they can be manipulated by programs. JavaScript can assign functions to variables and pass them to other functions, for example. Since functions are objects, you can set properties on them and even invoke methods on them.

JavaScript function definitions can be nested within other functions, and they have access to any variables that are in scope where they are defined. This means that JavaScript functions are *closures*, and it enables important and powerful programming techniques.

8.1 Defining Functions

The most straightforward way to define a JavaScript function is with the `function` keyword, which can be used as a declaration or as an expres-

sion. ES6 defines an important new way to define functions without the `function` keyword: "arrow functions" have a particularly compact syntax and are useful when passing one function as an argument to another function. The subsections that follow cover these three ways of defining functions. Note that some details of function definition syntax involving function parameters are deferred to §8.3.

In object literals and class definitions, there is a convenient shorthand syntax for defining methods. This shorthand syntax was covered in §6.10.5 and is equivalent to using a function definition expression and assigning it to an object property using the basic `name:value` object literal syntax. In another special case, you can use keywords `get` and `set` in object literals to define special property getter and setter methods. This function definition syntax was covered in §6.10.6.

Note that functions can also be defined with the `Function()` constructor, which is the subject of §8.7.7. Also, JavaScript defines some specialized kinds of functions. `function*` defines generator functions (see Chapter 12) and `async function` defines asynchronous functions (see Chapter 13).

8.1.1 Function Declarations

Function declarations consist of the `function` keyword, followed by these components:

- An identifier that names the function. The name is a required part of function declarations: it is used as the name of a variable, and the newly defined function object is assigned to the variable.
- A pair of parentheses around a comma-separated list of zero or more identifiers. These identifiers are the parameter names for the function, and they behave like local variables within the body of the function.
- A pair of curly braces with zero or more JavaScript statements inside. These statements are the body of the function: they are executed whenever the function is invoked.

Here are some example function declarations:

```
// Print the name and value of each property of o.  Return undefined.
function printprops(o) {
    for(let p in o) {
```

```
            console.log(`${p}: ${o[p]}\n`);
        }
    }

    // Compute the distance between Cartesian points (x1,y1) and (x2,y2).
    function distance(x1, y1, x2, y2) {
        let dx = x2 - x1;
        let dy = y2 - y1;
        return Math.sqrt(dx*dx + dy*dy);
    }

    // A recursive function (one that calls itself) that computes factorials
    // Recall that x! is the product of x and all positive integers less than i
    function factorial(x) {
        if (x <= 1) return 1;
        return x * factorial(x-1);
    }
```

One of the important things to understand about function declarations is that the name of the function becomes a variable whose value is the function itself. Function declaration statements are "hoisted" to the top of the enclosing script, function, or block so that functions defined in this way may be invoked from code that appears before the definition. Another way to say this is that all of the functions declared in a block of JavaScript code will be defined throughout that block, and they will be defined before the JavaScript interpreter begins to execute any of the code in that block.

The `distance()` and `factorial()` functions we've described are designed to compute a value, and they use `return` to return that value to their caller. The `return` statement causes the function to stop executing and to return the value of its expression (if any) to the caller. If the `return` statement does not have an associated expression, the return value of the function is `undefined`.

The `printprops()` function is different: its job is to output the names and values of an object's properties. No return value is necessary, and the function does not include a `return` statement. The value of an invocation of the `printprops()` function is always `undefined`. If a function does not contain a `return` statement, it simply executes each statement in the function body until it reaches the end, and returns the `undefined` value to the caller.

Prior to ES6, function declarations were only allowed at the top level within a JavaScript file or within another function. While some implementations bent the rule, it was not technically legal to define functions inside the body of loops, conditionals, or other blocks. In the strict mode of ES6, however, function declarations are allowed within blocks. A function defined within a block only exists within that block, however, and is not visible outside the block.

8.1.2 Function Expressions

Function expressions look a lot like function declarations, but they appear within the context of a larger expression or statement, and the name is optional. Here are some example function expressions:

```
// This function expression defines a function that squares its argument.
// Note that we assign it to a variable
const square = function(x) { return x*x; };

// Function expressions can include names, which is useful for recursion.
const f = function fact(x) { if (x <= 1) return 1; else return x*fact(x-1);

// Function expressions can also be used as arguments to other functions:
[3,2,1].sort(function(a,b) { return a-b; });

// Function expressions are sometimes defined and immediately invoked:
let tensquared = (function(x) {return x*x;}(10));
```

Note that the function name is optional for functions defined as expressions, and most of the preceding function expressions we've shown omit it. A function declaration actually *declares* a variable and assigns a function object to it. A function expression, on the other hand, does not declare a variable: it is up to you to assign the newly defined function object to a constant or variable if you are going to need to refer to it multiple times. It is a good practice to use `const` with function expressions so you don't accidentally overwrite your functions by assigning new values.

A name is allowed for functions, like the factorial function, that need to refer to themselves. If a function expression includes a name, the local function scope for that function will include a binding of that name to the function object. In effect, the function name becomes a local variable within the function. Most functions defined as expressions do not need

names, which makes their definition more compact (though not nearly as compact as arrow functions, described below).

There is an important difference between defining a function `f()` with a function declaration and assigning a function to the variable `f` after creating it as an expression. When you use the declaration form, the function objects are created before the code that contains them starts to run, and the definitions are hoisted so that you can call these functions from code that appears above the definition statement. This is not true for functions defined as expressions, however: these functions do not exist until the expression that defines them are actually evaluated. Furthermore, in order to invoke a function, you must be able to refer to it, and you can't refer to a function defined as an expression until it is assigned to a variable, so functions defined with expressions cannot be invoked before they are defined.

8.1.3 Arrow Functions

In ES6, you can define functions using a particularly compact syntax known as "arrow functions." This syntax is reminiscent of mathematical notation and uses an `=>` "arrow" to separate the function parameters from the function body. The `function` keyword is not used, and, since arrow functions are expressions instead of statements, there is no need for a function name, either. The general form of an arrow function is a comma-separated list of parameters in parentheses, followed by the `=>` arrow, followed by the function body in curly braces:

```
const sum = (x, y) => { return x + y; };
```

But arrow functions support an even more compact syntax. If the body of the function is a single `return` statement, you can omit the `return` keyword, the semicolon that goes with it, and the curly braces, and write the body of the function as the expression whose value is to be returned:

```
const sum = (x, y) => x + y;
```

Furthermore, if an arrow function has exactly one parameter, you can omit the parentheses around the parameter list:

```
const polynomial = x => x*x + 2*x + 3;
```

Note, however, that an arrow function with no arguments at all must be written with an empty pair of parentheses:

```
const constantFunc = () => 42;
```

Note that, when writing an arrow function, you must not put a new line between the function parameters and the `=>` arrow. Otherwise, you could end up with a line like `const polynomial = x`, which is a syntactically valid assignment statement on its own.

Also, if the body of your arrow function is a single `return` statement but the expression to be returned is an object literal, then you have to put the object literal inside parentheses to avoid syntactic ambiguity between the curly braces of a function body and the curly braces of an object literal:

```
const f = x => { return { value: x }; };  // Good: f() returns an object
const g = x => ({ value: x });            // Good: g() returns an object
const h = x => { value: x };              // Bad: h() returns nothing
const i = x => { v: x, w: x };            // Bad: Syntax Error
```

In the third line of this code, the function `h()` is truly ambiguous: the code you intended as an object literal can be parsed as a labeled statement, so a function that returns undefined is created. On the fourth line, however, the more complicated object literal is not a valid statement, and this illegal code causes a syntax error.

The concise syntax of arrow functions makes them ideal when you need to pass one function to another function, which is a common thing to do with array methods like `map()`, `filter()`, and `reduce()` (see §7.8.1), for example:

```
// Make a copy of an array with null elements removed.
let filtered = [1,null,2,3].filter(x => x !== null); // filtered == [1,2,3]
// Square some numbers:
let squares = [1,2,3,4].map(x => x*x);               // squares == [1,4,9,1
```

Arrow functions differ from functions defined in other ways in one critical way: they inherit the value of the `this` keyword from the environ-

ment in which they are defined rather than defining their own invocation context as functions defined in other ways do. This is an important and very useful feature of arrow functions, and we'll return to it again later in this chapter. Arrow functions also differ from other functions in that they do not have a `prototype` property, which means that they cannot be used as constructor functions for new classes (see §9.2).

8.1.4 Nested Functions

In JavaScript, functions may be nested within other functions. For example:

```
function hypotenuse(a, b) {
    function square(x) { return x*x; }
    return Math.sqrt(square(a) + square(b));
}
```

The interesting thing about nested functions is their variable scoping rules: they can access the parameters and variables of the function (or functions) they are nested within. In the code shown here, for example, the inner function `square()` can read and write the parameters `a` and `b` defined by the outer function `hypotenuse()`. These scope rules for nested functions are very important, and we will consider them again in §8.6.

8.2 Invoking Functions

The JavaScript code that makes up the body of a function is not executed when the function is defined, but rather when it is invoked. JavaScript functions can be invoked in five ways:

- As functions
- As methods
- As constructors
- Indirectly through their `call()` and `apply()` methods
- Implicitly, via JavaScript language features that do not appear like normal function invocations

8.2.1 Function Invocation

Functions are invoked as functions or as methods with an invocation expression (§4.5). An invocation expression consists of a function expression that evaluates to a function object followed by an open parenthesis, a comma-separated list of zero or more argument expressions, and a close parenthesis. If the function expression is a property-access expression—if the function is the property of an object or an element of an array—then it is a method invocation expression. That case will be explained in the following example. The following code includes a number of regular function invocation expressions:

```
printprops({x: 1});
let total = distance(0,0,2,1) + distance(2,1,3,5);
let probability = factorial(5)/factorial(13);
```

In an invocation, each argument expression (the ones between the parentheses) is evaluated, and the resulting values become the arguments to the function. These values are assigned to the parameters named in the function definition. In the body of the function, a reference to a parameter evaluates to the corresponding argument value.

For regular function invocation, the return value of the function becomes the value of the invocation expression. If the function returns because the interpreter reaches the end, the return value is `undefined`. If the function returns because the interpreter executes a `return` statement, then the return value is the value of the expression that follows the `return` or is `undefined` if the `return` statement has no value.

CONDITIONAL INVOCATION

In ES2020 you can insert `?.` after the function expression and before the open parenthesis in a function invocation in order to invoke the function only if it is not `null` or `undefined`. That is, the expression `f?.(x)` is equivalent (assuming no side effects) to:

```
(f !== null && f !== undefined) ? f(x) : undefined
```

Full details on this conditional invocation syntax are in §4.5.1.

For function invocation in non-strict mode, the invocation context (the `this` value) is the global object. In strict mode, however, the invocation context is `undefined`. Note that functions defined using the arrow syntax behave differently: they always inherit the `this` value that is in effect where they are defined.

Functions written to be invoked as functions (and not as methods) do not typically use the `this` keyword at all. The keyword can be used, however, to determine whether strict mode is in effect:

```
// Define and invoke a function to determine if we're in strict mode.
const strict = (function() { return !this; }());
```

RECURSIVE FUNCTIONS AND THE STACK

A *recursive* function is one, like the `factorial()` function at the start of this chapter, that calls itself. Some algorithms, such as those involving tree-based data structures, can be implemented particularly elegantly with recursive functions. When writing a recursive function, however, it is important to think about memory constraints. When a function A calls function B, and then function B calls function C, the JavaScript interpreter needs to keep track of the execution contexts for all three functions. When function C completes, the interpreter needs to know where to resume executing function B, and when function B completes, it needs to know where to resume executing function A. You can imagine these execution contexts as a stack. When a function calls another function, a new execution context is pushed onto the stack. When that function returns, its execution context object is popped off the stack. If a function calls itself recursively 100 times, the stack will have 100 objects pushed onto it, and then have those 100 objects popped off. This call stack takes memory. On modern hardware, it is typically fine to write recursive functions that call themselves hundreds of times. But if a function calls itself ten thousand times, it is likely to fail with an error such as "Maximum call-stack size exceeded."

8.2.2 Method Invocation

A *method* is nothing more than a JavaScript function that is stored in a property of an object. If you have a function `f` and an object `o`, you can

define a method named `m` of `o` with the following line:

```
o.m = f;
```

Having defined the method `m()` of the object `o`, invoke it like this:

```
o.m();
```

Or, if `m()` expects two arguments, you might invoke it like this:

```
o.m(x, y);
```

The code in this example is an invocation expression: it includes a function expression `o.m` and two argument expressions, `x` and `y`. The function expression is itself a property access expression, and this means that the function is invoked as a method rather than as a regular function.

The arguments and return value of a method invocation are handled exactly as described for regular function invocation. Method invocations differ from function invocations in one important way, however: the invocation context. Property access expressions consist of two parts: an object (in this case `o`) and a property name (`m`). In a method-invocation expression like this, the object `o` becomes the invocation context, and the function body can refer to that object by using the keyword `this`. Here is a concrete example:

```
let calculator = { // An object literal
    operand1: 1,
    operand2: 1,
    add() {         // We're using method shorthand syntax for this function
        // Note the use of the this keyword to refer to the containing obje
        this.result = this.operand1 + this.operand2;
    }
};
calculator.add();  // A method invocation to compute 1+1.
calculator.result  // => 2
```

Most method invocations use the dot notation for property access, but property access expressions that use square brackets also cause method invocation. The following are both method invocations, for example:

```
o["m"](x,y);    // Another way to write o.m(x,y).
a[0](z)         // Also a method invocation (assuming a[0] is a function).
```

Method invocations may also involve more complex property access expressions:

```
customer.surname.toUpperCase(); // Invoke method on customer.surname
f().m();                        // Invoke method m() on return value of f()
```

Methods and the `this` keyword are central to the object-oriented programming paradigm. Any function that is used as a method is effectively passed an implicit argument—the object through which it is invoked. Typically, a method performs some sort of operation on that object, and the method-invocation syntax is an elegant way to express the fact that a function is operating on an object. Compare the following two lines:

```
rect.setSize(width, height);
setRectSize(rect, width, height);
```

The hypothetical functions invoked in these two lines of code may perform exactly the same operation on the (hypothetical) object `rect`, but the method-invocation syntax in the first line more clearly indicates the idea that it is the object `rect` that is the primary focus of the operation.

METHOD CHAINING

When methods return objects, you can use the return value of one method invocation as part of a subsequent invocation. This results in a series (or "chain") of method invocations as a single expression. When working with Promise-based asynchronous operations (see Chapter 13), for example, it is common to write code structured like this:

```
// Run three asynchronous operations in sequence, handling errors.
doStepOne().then(doStepTwo).then(doStepThree).catch(handleErrors);
```

When you write a method that does not have a return value of its own, consider having the method return `this`. If you do this consistently throughout your API, you will enable a style of programming known as *method chaining*[1] in which an object can be named once and then multiple methods can be invoked on it:

```
new Square().x(100).y(100).size(50).outline("red").fill("blue").draw();
```

Note that `this` is a keyword, not a variable or property name. JavaScript syntax does not allow you to assign a value to `this`.

The `this` keyword is not scoped the way variables are, and, except for arrow functions, nested functions do not inherit the `this` value of the containing function. If a nested function is invoked as a method, its `this` value is the object it was invoked on. If a nested function (that is not an arrow function) is invoked as a function, then its `this` value will be either the global object (non-strict mode) or `undefined` (strict mode). It is a common mistake to assume that a nested function defined within a method and invoked as a function can use `this` to obtain the invocation context of the method. The following code demonstrates the problem:

```
let o = {                   // An object o.
    m: function() {         // Method m of the object.
        let self = this;    // Save the "this" value in a variable.
        this === o          // => true: "this" is the object o.
        f();                // Now call the helper function f().

        function f() {      // A nested function f
            this === o      // => false: "this" is global or undefined
```

```
            self === o    // => true: self is the outer "this" value.
        }
    }
};
o.m();                    // Invoke the method m on the object o.
```

Inside the nested function `f()`, the `this` keyword is not equal to the object `o`. This is widely considered to be a flaw in the JavaScript language, and it is important to be aware of it. The code above demonstrates one common workaround. Within the method `m`, we assign the `this` value to a variable `self`, and within the nested function `f`, we can use `self` instead of `this` to refer to the containing object.

In ES6 and later, another workaround to this issue is to convert the nested function `f` into an arrow function, which will properly inherit the `this` value:

```
const f = () => {
    this === o  // true, since arrow functions inherit this
};
```

Functions defined as expressions instead of statements are not hoisted, so in order to make this code work, the function definition for `f` will need to be moved within the method `m` so that it appears before it is invoked.

Another workaround is to invoke the `bind()` method of the nested function to define a new function that is implicitly invoked on a specified object:

```
const f = (function() {
    this === o  // true, since we bound this function to the outer this
}).bind(this);
```

We'll talk more about `bind()` in §8.7.5.

8.2.3 Constructor Invocation

If a function or method invocation is preceded by the keyword `new`, then it is a constructor invocation. (Constructor invocations were introduced in §4.6 and §6.2.2, and constructors will be covered in more detail in Chapter 9.) Constructor invocations differ from regular function and

method invocations in their handling of arguments, invocation context, and return value.

If a constructor invocation includes an argument list in parentheses, those argument expressions are evaluated and passed to the function in the same way they would be for function and method invocations. It is not common practice, but you can omit a pair of empty parentheses in a constructor invocation. The following two lines, for example, are equivalent:

```
o = new Object();
o = new Object;
```

A constructor invocation creates a new, empty object that inherits from the object specified by the `prototype` property of the constructor. Constructor functions are intended to initialize objects, and this newly created object is used as the invocation context, so the constructor function can refer to it with the `this` keyword. Note that the new object is used as the invocation context even if the constructor invocation looks like a method invocation. That is, in the expression `new o.m()`, `o` is not used as the invocation context.

Constructor functions do not normally use the `return` keyword. They typically initialize the new object and then return implicitly when they reach the end of their body. In this case, the new object is the value of the constructor invocation expression. If, however, a constructor explicitly uses the `return` statement to return an object, then that object becomes the value of the invocation expression. If the constructor uses `return` with no value, or if it returns a primitive value, that return value is ignored and the new object is used as the value of the invocation.

8.2.4 Indirect Invocation

JavaScript functions are objects, and like all JavaScript objects, they have methods. Two of these methods, `call()` and `apply()`, invoke the function indirectly. Both methods allow you to explicitly specify the `this` value for the invocation, which means you can invoke any function as a method of any object, even if it is not actually a method of that object. Both methods also allow you to specify the arguments for the invocation. The `call()` method uses its own argument list as arguments to the func-

tion, and the `apply()` method expects an array of values to be used as arguments. The `call()` and `apply()` methods are described in detail in §8.7.4.

8.2.5 Implicit Function Invocation

There are various JavaScript language features that do not look like function invocations but that cause functions to be invoked. Be extra careful when writing functions that may be implicitly invoked, because bugs, side effects, and performance issues in these functions are harder to diagnose and fix than in regular functions for the simple reason that it may not be obvious from a simple inspection of your code when they are being called.

The language features that can cause implicit function invocation include:

- If an object has getters or setters defined, then querying or setting the value of its properties may invoke those methods. See §6.10.6 for more information.
- When an object is used in a string context (such as when it is concatenated with a string), its `toString()` method is called. Similarly, when an object is used in a numeric context, its `valueOf()` method is invoked. See §3.9.3 for details.
- When you loop over the elements of an iterable object, there are a number of method calls that occur. Chapter 12 explains how iterators work at the function call level and demonstrates how to write these methods so that you can define your own iterable types.
- A tagged template literal is a function invocation in disguise. §14.5 demonstrates how to write functions that can be used in conjunction with template literal strings.
- Proxy objects (described in §14.7) have their behavior completely controlled by functions. Just about any operation on one of these objects will cause a function to be invoked.

8.3 Function Arguments and Parameters

JavaScript function definitions do not specify an expected type for the function parameters, and function invocations do not do any type checking on the argument values you pass. In fact, JavaScript function invocations do not even check the number of arguments being passed. The subsections that follow describe what happens when a function is invoked with fewer arguments than declared parameters or with more arguments than declared parameters. They also demonstrate how you can explicitly test the type of function arguments if you need to ensure that a function is not invoked with inappropriate arguments.

8.3.1 Optional Parameters and Defaults

When a function is invoked with fewer arguments than declared parameters, the additional parameters are set to their default value, which is normally `undefined`. It is often useful to write functions so that some arguments are optional. Following is an example:

```
// Append the names of the enumerable properties of object o to the
// array a, and return a.  If a is omitted, create and return a new array.
function getPropertyNames(o, a) {
    if (a === undefined) a = [];  // If undefined, use a new array
    for(let property in o) a.push(property);
    return a;
}

// getPropertyNames() can be invoked with one or two arguments:
let o = {x: 1}, p = {y: 2, z: 3};  // Two objects for testing
let a = getPropertyNames(o); // a == ["x"]; get o's properties in a new arr
getPropertyNames(p, a);      // a == ["x","y","z"]; add p's properties to i
```

Instead of using an `if` statement in the first line of this function, you can use the `||` operator in this idiomatic way:

```
a = a || [];
```

Recall from §4.10.2 that the `||` operator returns its first argument if that argument is truthy and otherwise returns its second argument. In this case, if any object is passed as the second argument, the function will use that object. But if the second argument is omitted (or `null` or another falsy value is passed), a newly created empty array will be used instead.

Note that when designing functions with optional arguments, you should be sure to put the optional ones at the end of the argument list so that they can be omitted. The programmer who calls your function cannot omit the first argument and pass the second: they would have to explicitly pass `undefined` as the first argument.

In ES6 and later, you can define a default value for each of your function parameters directly in the parameter list of your function. Simply follow the parameter name with an equals sign and the default value to use when no argument is supplied for that parameter:

```
// Append the names of the enumerable properties of object o to the
// array a, and return a.  If a is omitted, create and return a new array.
function getPropertyNames(o, a = []) {
    for(let property in o) a.push(property);
    return a;
}
```

Parameter default expressions are evaluated when your function is called, not when it is defined, so each time this `getPropertyNames()` function is invoked with one argument, a new empty array is created and passed.[2] It is probably easiest to reason about functions if the parameter defaults are constants (or literal expressions like `[]` and `{}`). But this is not required: you can use variables, or function invocations, for example, to compute the default value of a parameter. One interesting case is that, for functions with multiple parameters, you can use the value of a previous parameter to define the default value of the parameters that follow it:

```
// This function returns an object representing a rectangle's dimensions.
// If only width is supplied, make it twice as high as it is wide.
const rectangle = (width, height=width*2) => ({width, height});
rectangle(1)  // => { width: 1, height: 2 }
```

This code demonstrates that parameter defaults work with arrow functions. The same is true for method shorthand functions and all other forms of function definitions.

8.3.2 Rest Parameters and Variable-Length Argument Lists

Parameter defaults enable us to write functions that can be invoked with fewer arguments than parameters. *Rest parameters* enable the opposite case: they allow us to write functions that can be invoked with arbitrarily more arguments than parameters. Here is an example function that expects one or more numeric arguments and returns the largest one:

```
function max(first=-Infinity, ...rest) {
    let maxValue = first; // Start by assuming the first arg is biggest
    // Then loop through the rest of the arguments, looking for bigger
    for(let n of rest) {
        if (n > maxValue) {
            maxValue = n;
        }
    }
    // Return the biggest
    return maxValue;
}

max(1, 10, 100, 2, 3, 1000, 4, 5, 6)  // => 1000
```

A rest parameter is preceded by three periods, and it must be the last parameter in a function declaration. When you invoke a function with a rest parameter, the arguments you pass are first assigned to the non-rest parameters, and then any remaining arguments (i.e., the "rest" of the arguments) are stored in an array that becomes the value of the rest parameter. This last point is important: within the body of a function, the value of a rest parameter will always be an array. The array may be empty, but a rest parameter will never be `undefined`. (It follows from this that it is never useful—and not legal—to define a parameter default for a rest parameter.)

Functions like the previous example that can accept any number of arguments are called *variadic functions*, *variable arity functions*, or *vararg functions*. This book uses the most colloquial term, *varargs*, which dates to the early days of the C programming language.

Don't confuse the `...` that defines a rest parameter in a function definition with the `...` spread operator, described in §8.3.4, which can be used in function invocations.

8.3.3 The Arguments Object

Rest parameters were introduced into JavaScript in ES6. Before that version of the language, varargs functions were written using the Arguments object: within the body of any function, the identifier `arguments` refers to the Arguments object for that invocation. The Arguments object is an array-like object (see §7.9) that allows the argument values passed to the function to be retrieved by number, rather than by name. Here is the `max()` function from earlier, rewritten to use the Arguments object instead of a rest parameter:

```
function max(x) {
    let maxValue = -Infinity;
    // Loop through the arguments, looking for, and remembering, the bigges
    for(let i = 0; i < arguments.length; i++) {
        if (arguments[i] > maxValue) maxValue = arguments[i];
    }
    // Return the biggest
    return maxValue;
}

max(1, 10, 100, 2, 3, 1000, 4, 5, 6)  // => 1000
```

The Arguments object dates back to the earliest days of JavaScript and carries with it some strange historical baggage that makes it inefficient and hard to optimize, especially outside of strict mode. You may still encounter code that uses the Arguments object, but you should avoid using it in any new code you write. When refactoring old code, if you encounter a function that uses `arguments`, you can often replace it with a `...args` rest parameter. Part of the unfortunate legacy of the Arguments object is that, in strict mode, `arguments` is treated as a reserved word, and you cannot declare a function parameter or a local variable with that name.

8.3.4 The Spread Operator for Function Calls

The spread operator `...` is used to unpack, or "spread out," the elements of an array (or any other iterable object, such as strings) in a context where individual values are expected. We've seen the spread operator used with array literals in §7.1.2. The operator can be used, in the same way, in function invocations:

```
let numbers = [5, 2, 10, -1, 9, 100, 1];
Math.min(...numbers)  // => -1
```

Note that `...` is not a true operator in the sense that it cannot be evaluated to produce a value. Instead, it is a special JavaScript syntax that can be used in array literals and function invocations.

When we use the same `...` syntax in a function definition rather than a function invocation, it has the opposite effect to the spread operator. As we saw in §8.3.2, using `...` in a function definition gathers multiple function arguments into an array. Rest parameters and the spread operator are often useful together, as in the following function, which takes a function argument and returns an instrumented version of the function for testing:

```
// This function takes a function and returns a wrapped version
function timed(f) {
    return function(...args) {  // Collect args into a rest parameter array
        console.log(`Entering function ${f.name}`);
        let startTime = Date.now();
        try {
            // Pass all of our arguments to the wrapped function
            return f(...args);  // Spread the args back out again
        }
        finally {
            // Before we return the wrapped return value, print elapsed tim
            console.log(`Exiting ${f.name} after ${Date.now()-startTime}ms`
        }
    };
}

// Compute the sum of the numbers between 1 and n by brute force
function benchmark(n) {
    let sum = 0;
    for(let i = 1; i <= n; i++) sum += i;
    return sum;
}

// Now invoke the timed version of that test function
timed(benchmark)(1000000) // => 500000500000; this is the sum of the number
```

8.3.5 Destructuring Function Arguments into Parameters

When you invoke a function with a list of argument values, those values end up being assigned to the parameters declared in the function definition. This initial phase of function invocation is a lot like variable assignment. So it should not be surprising that we can use the techniques of destructuring assignment (see §3.10.3) with functions.

If you define a function that has parameter names within square brackets, you are telling the function to expect an array value to be passed for each pair of square brackets. As part of the invocation process, the array arguments will be unpacked into the individually named parameters. As an example, suppose we are representing 2D vectors as arrays of two numbers, where the first element is the X coordinate and the second element is the Y coordinate. With this simple data structure, we could write the following function to add two vectors:

```
function vectorAdd(v1, v2) {
    return [v1[0] + v2[0], v1[1] + v2[1]];
}
vectorAdd([1,2], [3,4])  // => [4,6]
```

The code would be easier to understand if we destructured the two vector arguments into more clearly named parameters:

```
function vectorAdd([x1,y1], [x2,y2]) { // Unpack 2 arguments into 4 paramet
    return [x1 + x2, y1 + y2];
}
vectorAdd([1,2], [3,4])  // => [4,6]
```

Similarly, if you are defining a function that expects an object argument, you can destructure parameters of that object. Let's use a vector example again, except this time, let's suppose that we represent vectors as objects with `x` and `y` parameters:

```
// Multiply the vector {x,y} by a scalar value
function vectorMultiply({x, y}, scalar) {
    return { x: x*scalar, y: y*scalar };
}
vectorMultiply({x: 1, y: 2}, 2)  // => {x: 2, y: 4}
```

This example of destructuring a single object argument into two parameters is a fairly clear one because the parameter names we use match the

property names of the incoming object. The syntax is more verbose and more confusing when you need to destructure properties with one name into parameters with different names. Here's the vector addition example, implemented for object-based vectors:

```
function vectorAdd(
    {x: x1, y: y1}, // Unpack 1st object into x1 and y1 params
    {x: x2, y: y2}  // Unpack 2nd object into x2 and y2 params
)
{
    return { x: x1 + x2, y: y1 + y2 };
}
vectorAdd({x: 1, y: 2}, {x: 3, y: 4})  // => {x: 4, y: 6}
```

The tricky thing about destructuring syntax like `{x:x1, y:y1}` is remembering which are the property names and which are the parameter names. The rule to keep in mind for destructuring assignment and destructuring function calls is that the variables or parameters being declared go in the spots where you'd expect values to go in an object literal. So property names are always on the lefthand side of the colon, and the parameter (or variable) names are on the right.

You can define parameter defaults with destructured parameters. Here's vector multiplication that works with 2D or 3D vectors:

```
// Multiply the vector {x,y} or {x,y,z} by a scalar value
function vectorMultiply({x, y, z=0}, scalar) {
    return { x: x*scalar, y: y*scalar, z: z*scalar };
}
vectorMultiply({x: 1, y: 2}, 2)  // => {x: 2, y: 4, z: 0}
```

Some languages (like Python) allow the caller of a function to invoke a function with arguments specified in `name=value` form, which is convenient when there are many optional arguments or when the parameter list is long enough that it is hard to remember the correct order. JavaScript does not allow this directly, but you can approximate it by destructuring an object argument into your function parameters. Consider a function that copies a specified number of elements from one array into another array with optionally specified starting offsets for each array. Since there are five possible parameters, some of which have defaults, and it would be hard for a caller to remember which order to pass the ar-

guments in, we can define and invoke the `arraycopy()` function like this:

```
function arraycopy({from, to=from, n=from.length, fromIndex=0, toIndex=0})
    let valuesToCopy = from.slice(fromIndex, fromIndex + n);
    to.splice(toIndex, 0, ...valuesToCopy);
    return to;
}
let a = [1,2,3,4,5], b = [9,8,7,6,5];
arraycopy({from: a, n: 3, to: b, toIndex: 4}) // => [9,8,7,6,1,2,3,5]
```

When you destructure an array, you can define a rest parameter for extra values within the array that is being unpacked. That rest parameter within the square brackets is completely different than the true rest parameter for the function:

```
// This function expects an array argument. The first two elements of that
// array are unpacked into the x and y parameters. Any remaining elements
// are stored in the coords array. And any arguments after the first array
// are packed into the rest array.
function f([x, y, ...coords], ...rest) {
    return [x+y, ...rest, ...coords];  // Note: spread operator here
}
f([1, 2, 3, 4], 5, 6)   // => [3, 5, 6, 3, 4]
```

In ES2018, you can also use a rest parameter when you destructure an object. The value of that rest parameter will be an object that has any properties that did not get destructured. Object rest parameters are often useful with the object spread operator, which is also a new feature of ES2018:

```
// Multiply the vector {x,y} or {x,y,z} by a scalar value, retain other pro
function vectorMultiply({x, y, z=0, ...props}, scalar) {
    return { x: x*scalar, y: y*scalar, z: z*scalar, ...props };
}
vectorMultiply({x: 1, y: 2, w: -1}, 2)  // => {x: 2, y: 4, z: 0, w: -1}
```

Finally, keep in mind that, in addition to destructuring argument objects and arrays, you can also destructure arrays of objects, objects that have array properties, and objects that have object properties, to essentially any depth. Consider graphics code that represents circles as objects with `x`, `y`, `radius`, and `color` properties, where the `color` property is an array of red, green, and blue color components. You might define a func-

tion that expects a single circle object to be passed to it but destructures that circle object into six separate parameters:

```
function drawCircle({x, y, radius, color: [r, g, b]}) {
    // Not yet implemented
}
```

If function argument destructuring is any more complicated than this, I find that the code becomes harder to read, rather than simpler. Sometimes, it is clearer to be explicit about your object property access and array indexing.

8.3.6 Argument Types

JavaScript method parameters have no declared types, and no type checking is performed on the values you pass to a function. You can help make your code self-documenting by choosing descriptive names for function arguments and by documenting them carefully in the comments for each function. (Alternatively, see §17.8 for a language extension that allows you to layer type checking on top of regular JavaScript.)

As described in §3.9, JavaScript performs liberal type conversion as needed. So if you write a function that expects a string argument and then call that function with a value of some other type, the value you passed will simply be converted to a string when the function tries to use it as a string. All primitive types can be converted to strings, and all objects have `toString()` methods (if not necessarily useful ones), so an error never occurs in this case.

This is not always true, however. Consider again the `arraycopy()` method shown earlier. It expects one or two array arguments and will fail if these arguments are of the wrong type. Unless you are writing a private function that will only be called from nearby parts of your code, it may be worth adding code to check the types of arguments like this. It is better for a function to fail immediately and predictably when passed bad values than to begin executing and fail later with an error message that is likely to be unclear. Here is an example function that performs type-checking:

```
// Return the sum of the elements an iterable object a.
// The elements of a must all be numbers.
function sum(a) {
    let total = 0;
    for(let element of a) { // Throws TypeError if a is not iterable
        if (typeof element !== "number") {
            throw new TypeError("sum(): elements must be numbers");
        }
        total += element;
    }
    return total;
}
sum([1,2,3])     // => 6
sum(1, 2, 3);    // !TypeError: 1 is not iterable
sum([1,2,"3"]); // !TypeError: element 2 is not a number
```

8.4 Functions as Values

The most important features of functions are that they can be defined and invoked. Function definition and invocation are syntactic features of JavaScript and of most other programming languages. In JavaScript, however, functions are not only syntax but also values, which means they can be assigned to variables, stored in the properties of objects or the elements of arrays, passed as arguments to functions, and so on.[3]

To understand how functions can be JavaScript data as well as JavaScript syntax, consider this function definition:

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

This definition creates a new function object and assigns it to the variable `square`. The name of a function is really immaterial; it is simply the name of a variable that refers to the function object. The function can be assigned to another variable and still work the same way:

```
let s = square;  // Now s refers to the same function that square does
square(4)        // => 16
s(4)             // => 16
```

Functions can also be assigned to object properties rather than variables. As we've already discussed, we call the functions "methods" when we do this:

```
let o = {square: function(x) { return x*x; }}; // An object literal
let y = o.square(16);                          // y == 256
```

Functions don't even require names at all, as when they're assigned to array elements:

```
let a = [x => x*x, 20]; // An array literal
a[0](a[1])              // => 400
```

The syntax of this last example looks strange, but it is still a legal function invocation expression!

As an example of how useful it is to treat functions as values, consider the `Array.sort()` method. This method sorts the elements of an array. Because there are many possible orders to sort by (numerical order, alphabetical order, date order, ascending, descending, and so on), the `sort()` method optionally takes a function as an argument to tell it how to perform the sort. This function has a simple job: for any two values it is passed, it returns a value that specifies which element would come first in a sorted array. This function argument makes `Array.sort()` perfectly general and infinitely flexible; it can sort any type of data into any conceivable order. Examples are shown in §7.8.6.

Example 8-1 demonstrates the kinds of things that can be done when functions are used as values. This example may be a little tricky, but the comments explain what is going on.

Example 8-1. Using functions as data

```
// We define some simple functions here
function add(x,y) { return x + y; }
function subtract(x,y) { return x - y; }
function multiply(x,y) { return x * y; }
function divide(x,y) { return x / y; }

// Here's a function that takes one of the preceding functions
// as an argument and invokes it on two operands
```

```
function operate(operator, operand1, operand2) {
    return operator(operand1, operand2);
}

// We could invoke this function like this to compute the value (2+3) + (4*
let i = operate(add, operate(add, 2, 3), operate(multiply, 4, 5));

// For the sake of the example, we implement the simple functions again,
// this time within an object literal;
const operators = {
    add:      (x,y) => x+y,
    subtract: (x,y) => x-y,
    multiply: (x,y) => x*y,
    divide:   (x,y) => x/y,
    pow:      Math.pow  // This works for predefined functions too
};

// This function takes the name of an operator, looks up that operator
// in the object, and then invokes it on the supplied operands. Note
// the syntax used to invoke the operator function.
function operate2(operation, operand1, operand2) {
    if (typeof operators[operation] === "function") {
        return operators[operation](operand1, operand2);
    }
    else throw "unknown operator";
}

operate2("add", "hello", operate2("add", " ", "world")) // => "hello world"
operate2("pow", 10, 2)  // => 100
```

8.4.1 Defining Your Own Function Properties

Functions are not primitive values in JavaScript, but a specialized kind of object, which means that functions can have properties. When a function needs a “static” variable whose value persists across invocations, it is often convenient to use a property of the function itself. For example, suppose you want to write a function that returns a unique integer whenever it is invoked. The function must never return the same value twice. In order to manage this, the function needs to keep track of the values it has already returned, and this information must persist across function invocations. You could store this information in a global variable, but that is unnecessary, because the information is used only by the function itself. It is better to store the information in a property of the Function object. Here is an example that returns a unique integer whenever it is called:

```
// Initialize the counter property of the function object.
// Function declarations are hoisted so we really can
// do this assignment before the function declaration.
uniqueInteger.counter = 0;

// This function returns a different integer each time it is called.
// It uses a property of itself to remember the next value to be returned.
function uniqueInteger() {
    return uniqueInteger.counter++;  // Return and increment counter proper
}
uniqueInteger()  // => 0
uniqueInteger()  // => 1
```

As another example, consider the following `factorial()` function that uses properties of itself (treating itself as an array) to cache previously computed results:

```
// Compute factorials and cache results as properties of the function itsel
function factorial(n) {
    if (Number.isInteger(n) && n > 0) {           // Positive integers only
        if (!(n in factorial)) {                  // If no cached result
            factorial[n] = n * factorial(n-1);    // Compute and cache it
        }
        return factorial[n];                      // Return the cached resu
    } else {
        return NaN;                               // If input was bad
    }
}
factorial[1] = 1;  // Initialize the cache to hold this base case.
factorial(6)  // => 720
factorial[5]  // => 120; the call above caches this value
```

8.5 Functions as Namespaces

Variables declared within a function are not visible outside of the function. For this reason, it is sometimes useful to define a function simply to act as a temporary namespace in which you can define variables without cluttering the global namespace.

Suppose, for example, you have a chunk of JavaScript code that you want to use in a number of different JavaScript programs (or, for client-side JavaScript, on a number of different web pages). Assume that this code,

like most code, defines variables to store the intermediate results of its computation. The problem is that since this chunk of code will be used in many different programs, you don't know whether the variables it creates will conflict with variables created by the programs that use it. The solution is to put the chunk of code into a function and then invoke the function. This way, variables that would have been global become local to the function:

```
function chunkNamespace() {
    // Chunk of code goes here
    // Any variables defined in the chunk are local to this function
    // instead of cluttering up the global namespace.
}
chunkNamespace();  // But don't forget to invoke the function!
```

This code defines only a single global variable: the function name `chunkNamespace`. If defining even a single property is too much, you can define and invoke an anonymous function in a single expression:

```
(function() {  // chunkNamespace() function rewritten as an unnamed express
    // Chunk of code goes here
}());          // End the function literal and invoke it now.
```

This technique of defining and invoking a function in a single expression is used frequently enough that it has become idiomatic and has been given the name "immediately invoked function expression." Note the use of parentheses in the previous code example. The open parenthesis before `function` is required because without it, the JavaScript interpreter tries to parse the `function` keyword as a function declaration statement. With the parenthesis, the interpreter correctly recognizes this as a function definition expression. The leading parenthesis also helps human readers recognize when a function is being defined to be immediately invoked instead of defined for later use.

This use of functions as namespaces becomes really useful when we define one or more functions inside the namespace function using variables within that namesapce, but then pass them back out as the return value of the namespace function. Functions like this are known as *closures*, and they're the topic of the next section.

8.6 Closures

Like most modern programming languages, JavaScript uses *lexical scoping*. This means that functions are executed using the variable scope that was in effect when they were defined, not the variable scope that is in effect when they are invoked. In order to implement lexical scoping, the internal state of a JavaScript function object must include not only the code of the function but also a reference to the scope in which the function definition appears. This combination of a function object and a scope (a set of variable bindings) in which the function's variables are resolved is called a *closure* in the computer science literature.

Technically, all JavaScript functions are closures, but because most functions are invoked from the same scope that they were defined in, it normally doesn't really matter that there is a closure involved. Closures become interesting when they are invoked from a different scope than the one they were defined in. This happens most commonly when a nested function object is returned from the function within which it was defined. There are a number of powerful programming techniques that involve this kind of nested function closures, and their use has become relatively common in JavaScript programming. Closures may seem confusing when you first encounter them, but it is important that you understand them well enough to use them comfortably.

The first step to understanding closures is to review the lexical scoping rules for nested functions. Consider the following code:

```
let scope = "global scope";           // A global variable
function checkscope() {
    let scope = "local scope";        // A local variable
    function f() { return scope; }    // Return the value in scope here
    return f();
}
checkscope()                          // => "local scope"
```

The `checkscope()` function declares a local variable and then defines and invokes a function that returns the value of that variable. It should be clear to you why the call to `checkscope()` returns "local scope". Now, let's change the code just slightly. Can you tell what this code will return?

```
let scope = "global scope";           // A global variable
function checkscope() {
    let scope = "local scope";        // A local variable
    function f() { return scope; }    // Return the value in scope here
    return f;
}
let s = checkscope()();               // What does this return?
```

In this code, a pair of parentheses has moved from inside `checkscope()` to outside of it. Instead of invoking the nested function and returning its result, `checkscope()` now just returns the nested function object itself. What happens when we invoke that nested function (with the second pair of parentheses in the last line of code) outside of the function in which it was defined?

Remember the fundamental rule of lexical scoping: JavaScript functions are executed using the scope they were defined in. The nested function `f()` was defined in a scope where the variable `scope` was bound to the value "local scope". That binding is still in effect when `f` is executed, no matter where it is executed from. So the last line of the preceding code example returns "local scope", not "global scope". This, in a nutshell, is the surprising and powerful nature of closures: they capture the local variable (and parameter) bindings of the outer function within which they are defined.

In §8.4.1, we defined a `uniqueInteger()` function that used a property of the function itself to keep track of the next value to be returned. A shortcoming of that approach is that buggy or malicious code could reset the counter or set it to a noninteger, causing the `uniqueInteger()` function to violate the "unique" or the "integer" part of its contract. Closures capture the local variables of a single function invocation and can use those variables as private state. Here is how we could rewrite the `uniqueInteger()` using an immediately invoked function expression to define a namespace and a closure that uses that namespace to keep its state private:

```
let uniqueInteger = (function() {   // Define and invoke
    let counter = 0;                // Private state of function below
    return function() { return counter++; };
}());
uniqueInteger()  // => 0
uniqueInteger()  // => 1
```

In order to understand this code, you have to read it carefully. At first glance, the first line of code looks like it is assigning a function to the variable `uniqueInteger`. In fact, the code is defining and invoking (as hinted by the open parenthesis on the first line) a function, so it is the return value of the function that is being assigned to `uniqueInteger`. Now, if we study the body of the function, we see that its return value is another function. It is this nested function object that gets assigned to `uniqueInteger`. The nested function has access to the variables in its scope and can use the `counter` variable defined in the outer function. Once that outer function returns, no other code can see the `counter` variable: the inner function has exclusive access to it.

Private variables like `counter` need not be exclusive to a single closure: it is perfectly possible for two or more nested functions to be defined within the same outer function and share the same scope. Consider the following code:

```
function counter() {
    let n = 0;
    return {
        count: function() { return n++; },
        reset: function() { n = 0; }
    };
}

let c = counter(), d = counter();   // Create two counters
c.count()                           // => 0
d.count()                           // => 0: they count independently
c.reset();                          // reset() and count() methods share st
c.count()                           // => 0: because we reset c
d.count()                           // => 1: d was not reset
```

The `counter()` function returns a “counter” object. This object has two methods: `count()` returns the next integer, and `reset()` resets the internal state. The first thing to understand is that the two methods share access to the private variable `n`. The second thing to understand is that each invocation of `counter()` creates a new scope—independent of the scopes used by previous invocations—and a new private variable within that scope. So if you call `counter()` twice, you get two counter objects with different private variables. Calling `count()` or `reset()` on one counter object has no effect on the other.

It is worth noting here that you can combine this closure technique with property getters and setters. The following version of the `counter()` function is a variation on code that appeared in §6.10.6, but it uses closures for private state rather than relying on a regular object property:

```
function counter(n) {  // Function argument n is the private variable
    return {
        // Property getter method returns and increments private counter va
        get count() { return n++; },
        // Property setter doesn't allow the value of n to decrease
        set count(m) {
            if (m > n) n = m;
            else throw Error("count can only be set to a larger value");
        }
    };
}

let c = counter(1000);
c.count            // => 1000
c.count            // => 1001
c.count = 2000;
c.count            // => 2000
c.count = 2000;    // !Error: count can only be set to a larger value
```

Note that this version of the `counter()` function does not declare a local variable but just uses its parameter `n` to hold the private state shared by the property accessor methods. This allows the caller of `counter()` to specify the initial value of the private variable.

Example 8-2 is a generalization of the shared private state through the closures technique we've been demonstrating here. This example defines an `addPrivateProperty()` function that defines a private variable and two nested functions to get and set the value of that variable. It adds these nested functions as methods of the object you specify.

Example 8-2. Private property accessor methods using closures

```
// This function adds property accessor methods for a property with
// the specified name to the object o. The methods are named get<name>
// and set<name>. If a predicate function is supplied, the setter
// method uses it to test its argument for validity before storing it.
// If the predicate returns false, the setter method throws an exception.
//
// The unusual thing about this function is that the property value
```

```
// that is manipulated by the getter and setter methods is not stored in
// the object o. Instead, the value is stored only in a local variable
// in this function. The getter and setter methods are also defined
// locally to this function and therefore have access to this local variabl
// This means that the value is private to the two accessor methods, and it
// cannot be set or modified except through the setter method.
function addPrivateProperty(o, name, predicate) {
    let value;  // This is the property value

    // The getter method simply returns the value.
    o[`get${name}`] = function() { return value; };

    // The setter method stores the value or throws an exception if
    // the predicate rejects the value.
    o[`set${name}`] = function(v) {
        if (predicate && !predicate(v)) {
            throw new TypeError(`set${name}: invalid value ${v}`);
        } else {
            value = v;
        }
    };
}

// The following code demonstrates the addPrivateProperty() method.
let o = {};  // Here is an empty object

// Add property accessor methods getName and setName()
// Ensure that only string values are allowed
addPrivateProperty(o, "Name", x => typeof x === "string");

o.setName("Frank");        // Set the property value
o.getName()                // => "Frank"
o.setName(0);              // !TypeError: try to set a value of the wrong ty
```

We've now seen a number of examples in which two closures are defined in the same scope and share access to the same private variable or variables. This is an important technique, but it is just as important to recognize when closures inadvertently share access to a variable that they should not share. Consider the following code:

```
// This function returns a function that always returns v
function constfunc(v) { return () => v; }

// Create an array of constant functions:
let funcs = [];
```

```
for(var i = 0; i < 10; i++) funcs[i] = constfunc(i);

// The function at array element 5 returns the value 5.
funcs[5]()     // => 5
```

When working with code like this that creates multiple closures using a loop, it is a common error to try to move the loop within the function that defines the closures. Think about the following code, for example:

```
// Return an array of functions that return the values 0-9
function constfuncs() {
    let funcs = [];
    for(var i = 0; i < 10; i++) {
        funcs[i] = () => i;
    }
    return funcs;
}

let funcs = constfuncs();
funcs[5]()     // => 10; Why doesn't this return 5?
```

This code creates 10 closures and stores them in an array. The closures are all defined within the same invocation of the function, so they share access to the variable `i`. When `constfuncs()` returns, the value of the variable `i` is 10, and all 10 closures share this value. Therefore, all the functions in the returned array of functions return the same value, which is not what we wanted at all. It is important to remember that the scope associated with a closure is "live." Nested functions do not make private copies of the scope or make static snapshots of the variable bindings. Fundamentally, the problem here is that variables declared with `var` are defined throughout the function. Our `for` loop declares the loop variable with `var i`, so the variable `i` is defined throughout the function rather than being more narrowly scoped to the body of the loop. The code demonstrates a common category of bugs in ES5 and before, but the introduction of block-scoped variables in ES6 addresses the issue. If we just replace the `var` with a `let` or a `const`, then the problem goes away. Because `let` and `const` are block scoped, each iteration of the loop defines a scope that is independent of the scopes for all other iterations, and each of these scopes has its own independent binding of `i`.

Another thing to remember when writing closures is that `this` is a JavaScript keyword, not a variable. As discussed earlier, arrow functions

inherit the `this` value of the function that contains them, but functions defined with the `function` keyword do not. So if you're writing a closure that needs to use the `this` value of its containing function, you should use an arrow function, or call `bind()`, on the closure before returning it, or assign the outer `this` value to a variable that your closure will inherit:

```
const self = this;  // Make the this value available to nested functions
```

8.7 Function Properties, Methods, and Constructor

We've seen that functions are values in JavaScript programs. The `typeof` operator returns the string "function" when applied to a function, but functions are really a specialized kind of JavaScript object. Since functions are objects, they can have properties and methods, just like any other object. There is even a `Function()` constructor to create new function objects. The subsections that follow document the `length`, `name`, and `prototype` properties; the `call()`, `apply()`, `bind()`, and `toString()` methods; and the `Function()` constructor.

8.7.1 The length Property

The read-only `length` property of a function specifies the *arity* of the function—the number of parameters it declares in its parameter list, which is usually the number of arguments that the function expects. If a function has a rest parameter, that parameter is not counted for the purposes of this `length` property.

8.7.2 The name Property

The read-only `name` property of a function specifies the name that was used when the function was defined, if it was defined with a name, or the name of the variable or property that an unnamed function expression was assigned to when it was first created. This property is primarily useful when writing debugging or error messages.

8.7.3 The prototype Property

All functions, except arrow functions, have a `prototype` property that refers to an object known as the *prototype object*. Every function has a different prototype object. When a function is used as a constructor, the newly created object inherits properties from the prototype object. Prototypes and the `prototype` property were discussed in §6.2.3 and will be covered again in Chapter 9.

8.7.4 The call() and apply() Methods

`call()` and `apply()` allow you to indirectly invoke (§8.2.4) a function as if it were a method of some other object. The first argument to both `call()` and `apply()` is the object on which the function is to be invoked; this argument is the invocation context and becomes the value of the `this` keyword within the body of the function. To invoke the function `f()` as a method of the object `o` (passing no arguments), you could use either `call()` or `apply()`:

```
f.call(o);
f.apply(o);
```

Either of these lines of code are similar to the following (which assume that `o` does not already have a property named `m`):

```
o.m = f;      // Make f a temporary method of o.
o.m();        // Invoke it, passing no arguments.
delete o.m;   // Remove the temporary method.
```

Remember that arrow functions inherit the `this` value of the context where they are defined. This cannot be overridden with the `call()` and `apply()` methods. If you call either of those methods on an arrow function, the first argument is effectively ignored.

Any arguments to `call()` after the first invocation context argument are the values that are passed to the function that is invoked (and these arguments are not ignored for arrow functions). For example, to pass two numbers to the function `f()` and invoke it as if it were a method of the object `o`, you could use code like this:

```
f.call(o, 1, 2);
```

The `apply()` method is like the `call()` method, except that the arguments to be passed to the function are specified as an array:

```
f.apply(o, [1,2]);
```

If a function is defined to accept an arbitrary number of arguments, the `apply()` method allows you to invoke that function on the contents of an array of arbitrary length. In ES6 and later, we can just use the spread operator, but you may see ES5 code that uses `apply()` instead. For example, to find the largest number in an array of numbers without using the spread operator, you could use the `apply()` method to pass the elements of the array to the `Math.max()` function:

```
let biggest = Math.max.apply(Math, arrayOfNumbers);
```

The `trace()` function defined in the following is similar to the `timed()` function defined in §8.3.4, but it works for methods instead of functions. It uses the `apply()` method instead of a spread operator, and by doing that, it is able to invoke the wrapped method with the same arguments and the same `this` value as the wrapper method:

```
// Replace the method named m of the object o with a version that logs
// messages before and after invoking the original method.
function trace(o, m) {
    let original = o[m];          // Remember original method in the closure
    o[m] = function(...args) {    // Now define the new method.
        console.log(new Date(), "Entering:", m);      // Log message.
        let result = original.apply(this, args);      // Invoke original.
        console.log(new Date(), "Exiting:", m);       // Log message.
        return result;                                // Return result.
    };
}
```

8.7.5 The bind() Method

The primary purpose of `bind()` is to *bind* a function to an object. When you invoke the `bind()` method on a function `f` and pass an object `o`, the method returns a new function. Invoking the new function (as a function) invokes the original function `f` as a method of `o`. Any arguments

you pass to the new function are passed to the original function. For example:

```
function f(y) { return this.x + y; } // This function needs to be bound
let o = { x: 1 };                    // An object we'll bind to
let g = f.bind(o);                   // Calling g(x) invokes f() on o
g(2)                                 // => 3
let p = { x: 10, g };                // Invoke g() as a method of this obje
p.g(2)                               // => 3: g is still bound to o, not p.
```

Arrow functions inherit their `this` value from the environment in which they are defined, and that value cannot be overridden with `bind()`, so if the function `f()` in the preceding code was defined as an arrow function, the binding would not work. The most common use case for calling `bind()` is to make non-arrow functions behave like arrow functions, however, so this limitation on binding arrow functions is not a problem in practice.

The `bind()` method does more than just bind a function to an object, however. It can also perform partial application: any arguments you pass to `bind()` after the first are bound along with the `this` value. This partial application feature of `bind()` does work with arrow functions. Partial application is a common technique in functional programming and is sometimes called *currying*. Here are some examples of the `bind()` method used for partial application:

```
let sum = (x,y) => x + y;       // Return the sum of 2 args
let succ = sum.bind(null, 1);  // Bind the first argument to 1
succ(2)  // => 3: x is bound to 1, and we pass 2 for the y argument

function f(y,z) { return this.x + y + z; }
let g = f.bind({x: 1}, 2);     // Bind this and y
g(3)     // => 6: this.x is bound to 1, y is bound to 2 and z is 3
```

The `name` property of the function returned by `bind()` is the name property of the function that `bind()` was called on, prefixed with the word “bound”.

8.7.6 The toString() Method

Like all JavaScript objects, functions have a `toString()` method. The ECMAScript spec requires this method to return a string that follows the syntax of the function declaration statement. In practice, most (but not all) implementations of this `toString()` method return the complete source code for the function. Built-in functions typically return a string that includes something like "[native code]" as the function body.

8.7.7 The Function() Constructor

Because functions are objects, there is a `Function()` constructor that can be used to create new functions:

```
const f = new Function("x", "y", "return x*y;");
```

This line of code creates a new function that is more or less equivalent to a function defined with the familiar syntax:

```
const f = function(x, y) { return x*y; };
```

The `Function()` constructor expects any number of string arguments. The last argument is the text of the function body; it can contain arbitrary JavaScript statements, separated from each other by semicolons. All other arguments to the constructor are strings that specify the parameter names for the function. If you are defining a function that takes no arguments, you would simply pass a single string—the function body—to the constructor.

Notice that the `Function()` constructor is not passed any argument that specifies a name for the function it creates. Like function literals, the `Function()` constructor creates anonymous functions.

There are a few points that are important to understand about the `Function()` constructor:

- The `Function()` constructor allows JavaScript functions to be dynamically created and compiled at runtime.
- The `Function()` constructor parses the function body and creates a new function object each time it is called. If the call to the constructor appears within a loop or within a frequently called function, this process can be inefficient. By contrast, nested functions and function

expressions that appear within loops are not recompiled each time they are encountered.

- A last, very important point about the `Function()` constructor is that the functions it creates do not use lexical scoping; instead, they are always compiled as if they were top-level functions, as the following code demonstrates:

```
let scope = "global";
function constructFunction() {
    let scope = "local";
    return new Function("return scope");  // Doesn't capture local scope!
}
// This line returns "global" because the function returned by the
// Function() constructor does not use the local scope.
constructFunction()()  // => "global"
```

The `Function()` constructor is best thought of as a globally scoped version of `eval()` (see §4.12.2) that defines new variables and functions in its own private scope. You will probably never need to use this constructor in your code.

8.8 Functional Programming

JavaScript is not a functional programming language like Lisp or Haskell, but the fact that JavaScript can manipulate functions as objects means that we can use functional programming techniques in JavaScript. Array methods such as `map()` and `reduce()` lend themselves particularly well to a functional programming style. The sections that follow demonstrate techniques for functional programming in JavaScript. They are intended as a mind-expanding exploration of the power of JavaScript's functions, not as a prescription for good programming style.

8.8.1 Processing Arrays with Functions

Suppose we have an array of numbers and we want to compute the mean and standard deviation of those values. We might do that in nonfunctional style like this:

```
let data = [1,1,3,5,5];  // This is our array of numbers
```

```
// The mean is the sum of the elements divided by the number of elements
let total = 0;
for(let i = 0; i < data.length; i++) total += data[i];
let mean = total/data.length;  // mean == 3; The mean of our data is 3

// To compute the standard deviation, we first sum the squares of
// the deviation of each element from the mean.
total = 0;
for(let i = 0; i < data.length; i++) {
    let deviation = data[i] - mean;
    total += deviation * deviation;
}
let stddev = Math.sqrt(total/(data.length-1));  // stddev == 2
```

We can perform these same computations in concise functional style using the array methods `map()` and `reduce()` like this (see §7.8.1 to review these methods):

```
// First, define two simple functions
const sum = (x,y) => x+y;
const square = x => x*x;

// Then use those functions with Array methods to compute mean and stddev
let data = [1,1,3,5,5];
let mean = data.reduce(sum)/data.length;  // mean == 3
let deviations = data.map(x => x-mean);
let stddev = Math.sqrt(deviations.map(square).reduce(sum)/(data.length-1));
stddev  // => 2
```

This new version of the code looks quite different than the first one, but it is still invoking methods on objects, so it has some object-oriented conventions remaining. Let's write functional versions of the `map()` and `reduce()` methods:

```
const map = function(a, ...args) { return a.map(...args); };
const reduce = function(a, ...args) { return a.reduce(...args); };
```

With these `map()` and `reduce()` functions defined, our code to compute the mean and standard deviation now looks like this:

```
const sum = (x,y) => x+y;
const square = x => x*x;
```

```
let data = [1,1,3,5,5];
let mean = reduce(data, sum)/data.length;
let deviations = map(data, x => x-mean);
let stddev = Math.sqrt(reduce(map(deviations, square), sum)/(data.length-1)
stddev  // => 2
```

8.8.2 Higher-Order Functions

A *higher-order function* is a function that operates on functions, taking one or more functions as arguments and returning a new function. Here is an example:

```
// This higher-order function returns a new function that passes its
// arguments to f and returns the logical negation of f's return value;
function not(f) {
    return function(...args) {              // Return a new function
        let result = f.apply(this, args);   // that calls f
        return !result;                     // and negates its result.
    };
}

const even = x => x % 2 === 0; // A function to determine if a number is ev
const odd = not(even);         // A new function that does the opposite
[1,1,3,5,5].every(odd)         // => true: every element of the array is od
```

This `not()` function is a higher-order function because it takes a function argument and returns a new function. As another example, consider the `mapper()` function that follows. It takes a function argument and returns a new function that maps one array to another using that function. This function uses the `map()` function defined earlier, and it is important that you understand how the two functions are different:

```
// Return a function that expects an array argument and applies f to
// each element, returning the array of return values.
// Contrast this with the map() function from earlier.
function mapper(f) {
    return a => map(a, f);
}

const increment = x => x+1;
const incrementAll = mapper(increment);
incrementAll([1,2,3])  // => [2,3,4]
```

Here is another, more general, example that takes two functions, `f` and `g`, and returns a new function that computes `f(g())`:

```
// Return a new function that computes f(g(...)).
// The returned function h passes all of its arguments to g, then passes
// the return value of g to f, then returns the return value of f.
// Both f and g are invoked with the same this value as h was invoked with.
function compose(f, g) {
    return function(...args) {
        // We use call for f because we're passing a single value and
        // apply for g because we're passing an array of values.
        return f.call(this, g.apply(this, args));
    };
}

const sum = (x,y) => x+y;
const square = x => x*x;
compose(square, sum)(2,3)  // => 25; the square of the sum
```

The `partial()` and `memoize()` functions defined in the sections that follow are two more important higher-order functions.

8.8.3 Partial Application of Functions

The `bind()` method of a function `f` (see §8.7.5) returns a new function that invokes `f` in a specified context and with a specified set of arguments. We say that it binds the function to an object and partially applies the arguments. The `bind()` method partially applies arguments on the left—that is, the arguments you pass to `bind()` are placed at the start of the argument list that is passed to the original function. But it is also possible to partially apply arguments on the right:

```
// The arguments to this function are passed on the left
function partialLeft(f, ...outerArgs) {
    return function(...innerArgs) { // Return this function
        let args = [...outerArgs, ...innerArgs]; // Build the argument list
        return f.apply(this, args);              // Then invoke f with it
    };
}

// The arguments to this function are passed on the right
function partialRight(f, ...outerArgs) {
    return function(...innerArgs) {  // Return this function
```

```
            let args = [...innerArgs, ...outerArgs]; // Build the argument list
            return f.apply(this, args);              // Then invoke f with it
        };
    }

    // The arguments to this function serve as a template. Undefined values
    // in the argument list are filled in with values from the inner set.
    function partial(f, ...outerArgs) {
        return function(...innerArgs) {
            let args = [...outerArgs]; // local copy of outer args template
            let innerIndex=0;          // which inner arg is next
            // Loop through the args, filling in undefined values from inner ar
            for(let i = 0; i < args.length; i++) {
                if (args[i] === undefined) args[i] = innerArgs[innerIndex++];
            }
            // Now append any remaining inner arguments
            args.push(...innerArgs.slice(innerIndex));
            return f.apply(this, args);
        };
    }

    // Here is a function with three arguments
    const f = function(x,y,z) { return x * (y - z); };
    // Notice how these three partial applications differ
    partialLeft(f, 2)(3,4)         // => -2: Bind first argument: 2 * (3 - 4)
    partialRight(f, 2)(3,4)        // =>  6: Bind last argument: 3 * (4 - 2)
    partial(f, undefined, 2)(3,4)  // => -6: Bind middle argument: 3 * (2 - 4)
```

These partial application functions allow us to easily define interesting functions out of functions we already have defined. Here are some examples:

```
    const increment = partialLeft(sum, 1);
    const cuberoot = partialRight(Math.pow, 1/3);
    cuberoot(increment(26))  // => 3
```

Partial application becomes even more interesting when we combine it with other higher-order functions. Here, for example, is a way to define the preceding `not()` function just shown using composition and partial application:

```
    const not = partialLeft(compose, x => !x);
    const even = x => x % 2 === 0;
    const odd = not(even);
```

```
const isNumber = not(isNaN);
odd(3) && isNumber(2)  // => true
```

We can also use composition and partial application to redo our mean and standard deviation calculations in extreme functional style:

```
// sum() and square() functions are defined above. Here are some more:
const product = (x,y) => x*y;
const neg = partial(product, -1);
const sqrt = partial(Math.pow, undefined, .5);
const reciprocal = partial(Math.pow, undefined, neg(1));

// Now compute the mean and standard deviation.
let data = [1,1,3,5,5];   // Our data
let mean = product(reduce(data, sum), reciprocal(data.length));
let stddev = sqrt(product(reduce(map(data,
                                     compose(square,
                                             partial(sum, neg(mean)))),
                                 sum),
                          reciprocal(sum(data.length,neg(1)))));
[mean, stddev]  // => [3, 2]
```

Notice that this code to compute mean and standard deviation is entirely function invocations; there are no operators involved, and the number of parentheses has grown so large that this JavaScript is beginning to look like Lisp code. Again, this is not a style that I advocate for JavaScript programming, but it is an interesting exercise to see how deeply functional JavaScript code can be.

8.8.4 Memoization

In §8.4.1, we defined a factorial function that cached its previously computed results. In functional programming, this kind of caching is called *memoization*. The code that follows shows a higher-order function, `memoize()`, that accepts a function as its argument and returns a memoized version of the function:

```
// Return a memoized version of f.
// It only works if arguments to f all have distinct string representations
function memoize(f) {
    const cache = new Map();  // Value cache stored in the closure.
```

```
    return function(...args) {
        // Create a string version of the arguments to use as a cache key.
        let key = args.length + args.join("+");
        if (cache.has(key)) {
            return cache.get(key);
        } else {
            let result = f.apply(this, args);
            cache.set(key, result);
            return result;
        }
    };
}
```

The `memoize()` function creates a new object to use as the cache and assigns this object to a local variable so that it is private to (in the closure of) the returned function. The returned function converts its arguments array to a string and uses that string as a property name for the cache object. If a value exists in the cache, it returns it directly. Otherwise, it calls the specified function to compute the value for these arguments, caches that value, and returns it. Here is how we might use `memoize()`:

```
// Return the Greatest Common Divisor of two integers using the Euclidian
// algorithm: http://en.wikipedia.org/wiki/Euclidean_algorithm
function gcd(a,b) {  // Type checking for a and b has been omitted
    if (a < b) {           // Ensure that a >= b when we start
        [a, b] = [b, a];   // Destructuring assignment to swap variables
    }
    while(b !== 0) {       // This is Euclid's algorithm for GCD
        [a, b] = [b, a%b];
    }
    return a;
}

const gcdmemo = memoize(gcd);
gcdmemo(85, 187)  // => 17

// Note that when we write a recursive function that we will be memoizing,
// we typically want to recurse to the memoized version, not the original.
const factorial = memoize(function(n) {
    return (n <= 1) ? 1 : n * factorial(n-1);
});
factorial(5)      // => 120: also caches values for 4, 3, 2 and 1.
```

8.9 Summary

Some key points to remember about this chapter are as follows:

- You can define functions with the `function` keyword and with the ES6 `=>` arrow syntax.
- You can invoke functions, which can be used as methods and constructors.
- Some ES6 features allow you to define default values for optional function parameters, to gather multiple arguments into an array using a rest parameter, and to destructure object and array arguments into function parameters.
- You can use the `...` spread operator to pass the elements of an array or other iterable object as arguments in a function invocation.
- A function defined inside of and returned by an enclosing function retains access to its lexical scope and can therefore read and write the variables defined inside the outer function. Functions used in this way are called *closures*, and this is a technique that is worth understanding.
- Functions are objects that can be manipulated by JavaScript, and this enables a functional style of programming.

[1] The term was coined by Martin Fowler. See *http://martinfowler.com/dslCatalog/methodChaining.html*.

[2] If you are familiar with Python, note that this is different than Python, in which every invocation shares the same default value.

[3] This may not seem like a particularly interesting point unless you are familiar with more static languages, in which functions are part of a program but cannot be manipulated by the program.