**SUMMARY 1**

We looked at four types of JavaScript values in this chapter: numbers, strings, Booleans, and undefined values. Such values are created by typing in their name (true, null) or value (13,

"abc"). You can combine and transform values with operators. We saw binary operators for arithmetic (+, -, \*, /, and %), string concatenation (+), comparison (==, !=, ===, !==, <, >, <=, >=), and logic (&&, ||), as well as several unary operators (- to negate a number, ! to negate logically, and typeof to find a value’s type) and a ternary operator (?:) to pick one of two values based on a third value. This gives you enough information to use JavaScript as a pocket calcula-

tor but not much more. The next chapter will start tying these expressions

together into basic programs.

**SUMMARY 2**

You now know that a program is built out of statements, which themselves sometimes contain more statements. Statements tend to contain expressions, which themselves can be built out of smaller expressions. Putting statements after one another gives you a program that is exe-

cuted from top to bottom. You can introduce disturbances in the flow of control by using conditional (if, else, and switch) and looping (while, do, and for) statements. Bindings can be used to file pieces of data under a name, and they are

useful for tracking state in your program. The environment is the set of bindings that are defined. JavaScript systems always put a number of useful standard bindings into your environment. Functions are special values that encapsulate a piece of program. You

can invoke them by writing functionName(argument1, argument2). Such a func-

tion call is an expression and may produce a value.

**SUMMARY 3**

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

// Define f to hold a function value const f = function(a) { console.log(a + 2);

};

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

}

// A less verbose function value let h = a => a % 3;

A key aspect in understanding functions is understanding scopes. Each

block creates a new scope. Parameters and bindings declared in a given scope are local and not visible from the outside. Bindings declared with var behave differently—they end up in the nearest function scope or the global scope.

Separating the tasks your program performs into different functions is

helpful. You won’t have to repeat yourself as much, and functions can help

organize a program by grouping code into pieces that do specific things.

**SUMMARY 4**

Objects and arrays (which are a specific kind of object) provide ways to group several values into a single value. Conceptually, this allows us to put a bunch of related things in a bag and run around with the bag, instead of wrapping our arms around all of the individual things and trying to hold on to them separately. Most values in JavaScript have properties, the exceptions being null and

undefined. Properties are accessed using value.prop or value["prop"]. Objects tend to use names for their properties and store more or less a fixed set of them. Arrays, on the other hand, usually contain varying amounts of conceptually identical values and use numbers (starting from 0) as the names of their properties. There are some named properties in arrays, such as length and a number

of methods. Methods are functions that live in properties and (usually) act on the value they are a property of. You can iterate over arrays using a special kind of for loop—for (let

element of array).