**Chapter 1**

**Values, Types, and Operators**

Below the surface of the machine, the program moves. Without effort, it expands and contracts. In great harmony, electrons scatter and regroup. The forms on the monitor are but ripples on the water. The essence stays invisibly below.

* Master Yuan-Ma, The Book of Programming

Inside the computer’s world, there is only data. You can read data, modify data, create new data—but anything that isn’t data simply does not exist. All this data is stored as long sequences of bits and is thus fundamentally alike.

Bits are any kind of two-valued things, usually described as zeros and ones. Inside the computer, they take forms such as a high or low electrical charge, a strong or weak signal, or a shiny or dull spot on the surface of a CD. Any piece of discrete information can be reduced to a sequence of zeros and ones and thus represented in bits.

For example, think about how you might show the number 13 in bits. It works the same way you write decimal numbers, but instead of 10 different digits, you have only 2, and the weight of each increases by a factor of 2 from right to left. Here are the bits that make up the number 13, with the weights of the digits shown below them:

0 0 0 0 1 1 0 1

128 64 32 16 8 4 2 1

So that’s the binary number 00001101, or 8 + 4 + 1, which equals 13.

**Values**

Imagine a sea of bits. An ocean of them. A typical modern computer has more than 30 billion bits in its volatile data storage. Nonvolatile storage (the hard disk or equivalent) tends to have yet a few orders of magnitude more.

To be able to work with such quantities of bits without getting lost, you can separate them into chunks that represent pieces of information. In a JavaScript environment, those chunks are called *values*. Though all values are made of bits, they play different roles. Every value has a type that determines its role. There are six basic types of values in JavaScript: numbers, strings, Booleans, objects, functions, and undefined values.

To create a value, you must merely invoke its name. This is convenient. You don’t have to gather building material for your values or pay for them. You just call for one, and *woosh*, you have it. They are not created from thin air, of course. Every value has to be stored somewhere, and if you want to use a gigantic amount of them at the same time, you might run out of bits. Fortunately, this is a problem only if you need them all simultaneously. As soon as you no longer use a value, it will dissipate, leaving behind its bits to be recycled as building material for the next generation of values.

This chapter introduces the atomic elements of JavaScript programs, that is, the simple value types and the operators that can act on such values.

**Numbers**

Values of the *number* type are, unsurprisingly, numeric values. In a JavaScript program, they are written as follows:

13

Use that in a program, and it will cause the bit pattern for the number 13 to come into existence inside the computer’s memory.

JavaScript uses a fixed number of bits, namely 64 of them, to store a single number value. There are only so many patterns you can make with 64 bits, which means that the amount of different numbers that can be represented is limited. For *N* decimal digits, the amount of numbers that can be represented is 10*N*. Similarly, given 64 binary digits, you can represent 264 different numbers, which is about 18 quintillion (an 18 with 18 zeros after it). This is a lot.

Computer memory used to be a lot smaller, and people tended to use groups of 8 or 16 bits to represent their numbers. It was easy to accidentally *overflow* such small numbers—to end up with a number that did not fit into the given amount of bits. Today, even personal computers have plenty of memory, so you are free to use 64-bit chunks, which means you need to worry about overflow only when dealing with truly astronomical numbers.

Not all whole numbers below 18 quintillion fit in a JavaScript number, though. Those bits also store negative numbers, so one bit indicates the sign of the number. A bigger issue is that nonwhole numbers must also be represented. To do this, some of the bits are used to store the position of the decimal point. The actual maximum whole number that can be stored is more in the range of 9 quadrillion (15 zeros), which is still pleasantly huge.

Fractional numbers are written by using a dot.

9.81

For very big or very small numbers, you can also use scientific notation by adding an “e” (for “exponent”), followed by the exponent of the number:

2.998e8

That is 2.998 × 108 = 299,800,000.

Calculations with whole numbers (also called *integers*) smaller than the aforementioned 9 quadrillion are guaranteed to always be precise. Unfortunately, calculations with fractional numbers are generally not. Just as π (pi) cannot be precisely expressed by a finite number of decimal digits, many numbers lose some precision when only 64 bits are available to store them. This is a shame, but it causes practical problems only in specific situations. The important thing is to be aware of it and treat fractional digital numbers as approximations, not as precise values.

**Arithmetic**

The main thing to do with numbers is arithmetic. Arithmetic operations such as addition or multiplication take two number values and produce a new number from them. Here is what they look like in JavaScript:

100 + 4 \* 11

The + and \* symbols are called *operators*. The first stands for addition, and the second stands for multiplication. Putting an operator between two values will apply it to those values and produce a new value.

Does the example mean “add 4 and 100, and multiply the result by 11”, or is the multiplication done before the adding? As you might have guessed, the multiplication happens first. But as in mathematics, you can change this by wrapping the addition in parentheses.

(100 + 4) \* 11

For subtraction, there is the - operator, and division can be done with the / operator.

When operators appear together without parentheses, the order in which they are applied is determined by the *precedence* of the operators. The example shows that multiplication comes before addition. The / operator has the same precedence as \*. Likewise for + and -. When multiple operators with the same precedence appear next to each other, as in 1 - 2 + 1, they are applied left to right: (1 - 2) + 1.

These rules of precedence are not something you should worry about. When in doubt, just add parentheses.

There is one more arithmetic operator, which you might not immediately recognize. The % symbol is used to represent the *remainder* operation. X % Y is the remainder of dividing X by Y. For example, 314 % 100 produces 14, and 144 % 12 gives 0. Remainder’s precedence is the same as that of multiplication and division. You’ll often see this operator referred to as *modulo*, though technically *remainder* is more accurate.

**Special numbers**

There are three special values in JavaScript that are considered numbers but don’t behave like normal numbers.

The first two are Infinity and -Infinity, which represent the positive and negative infinities. Infinity - 1 is still Infinity, and so on. Don’t put too much trust in infinity-based computation. It isn’t mathematically solid, and it will quickly lead to our next special number: NaN.

NaN stands for “not a number”, even though it is a value of the number type. You’ll get this result when you, for example, try to calculate 0 / 0 (zero divided by zero), Infinity - Infinity, or any number of other numeric operations that don’t yield a precise, meaningful result.

**Strings**

The next basic data type is the *string*. Strings are used to represent text. They are written by enclosing their content in quotes.

"Patch my boat with chewing gum"

'Monkeys wave goodbye'

Both single and double quotes can be used to mark strings as long as the quotes at the start and the end of the string match.

Almost anything can be put between quotes, and JavaScript will make a string value out of it. But a few characters are more difficult. You can imagine how putting quotes between quotes might be hard. *Newlines* (the characters you get when you press Enter) also can’t be put between quotes. The string has to stay on a single line.

To make it possible to include such characters in a string, the following notation is used: whenever a backslash (\) is found inside quoted text, it indicates that the character after it has a special meaning. This is called *escaping* the character. A quote that is preceded by a backslash will not end the string but be part of it. When an n character occurs after a backslash, it is interpreted as a newline. Similarly, a t after a backslash means a tab character. Take the following string:

"This is the first line\nAnd this is the second"

The actual text contained is this:

This is the first line

And this is the second

There are, of course, situations where you want a backslash in a string to be just a backslash, not a special code. If two backslashes follow each other, they will collapse together, and only one will be left in the resulting string value. This is how the string “A newline character is written like "\n".” can be expressed:

"A newline character is written like \"\\n\"."

Strings cannot be divided, multiplied, or subtracted, but the + operator *can* be used on them. It does not add, but it *concatenates*—it glues two strings together. The following line will produce the string "concatenate":

"con" + "cat" + "e" + "nate"

There are more ways of manipulating strings, which we will discuss when we get to methods in [Chapter 4](http://eloquentjavascript.net/04_data.html#methods).

**Unary operators**

Not all operators are symbols. Some are written as words. One example is the typeof operator, which produces a string value naming the type of the value you give it.

edit & run code by clicking it

console.log(typeof 4.5)

// → number

console.log(typeof "x")

// → string

We will use console.log in example code to indicate that we want to see the result of evaluating something. When you run such code, the value produced should be shown on the screen, though how it appears will depend on the JavaScript environment you use to run it.

The other operators we saw all operated on two values, but typeof takes only one. Operators that use two values are called *binary* operators, while those that take one are called *unary* operators. The minus operator can be used both as a binary operator and as a unary operator.

console.log(- (10 - 2))

// → -8

**Boolean values**

Often, you will need a value that simply distinguishes between two possibilities, like “yes” and “no” or “on” and “off”. For this, JavaScript has a *Boolean* type, which has just two values: true and false (which are written simply as those words).

**Comparisons**

Here is one way to produce Boolean values:

console.log(3 > 2)

// → true

console.log(3 < 2)

// → false

The > and < signs are the traditional symbols for “is greater than” and “is less than”, respectively. They are binary operators. Applying them results in a Boolean value that indicates whether they hold true in this case.

Strings can be compared in the same way.

console.log("Aardvark" < "Zoroaster")

// → true

The way strings are ordered is more or less alphabetic: uppercase letters are always “less” than lowercase ones, so "Z" < "a" is true, and non-alphabetic characters (!, -, and so on) are also included in the ordering. The actual comparison is based on the *Unicode* standard. This standard assigns a number to virtually every character you would ever need, including characters from Greek, Arabic, Japanese, Tamil, and so on. Having such numbers is useful for storing strings inside a computer because it makes it possible to represent them as a sequence of numbers. When comparing strings, JavaScript goes over them from left to right, comparing the numeric codes of the characters one by one.

Other similar operators are >= (greater than or equal to), <= (less than or equal to), == (equal to), and != (not equal to).

console.log("Itchy" != "Scratchy")

// → true

There is only one value in JavaScript that is not equal to itself, and that is NaN, which stands for “not a number”.

console.log(NaN == NaN)

// → false

NaN is supposed to denote the result of a nonsensical computation, and as such, it isn’t equal to the result of any *other* nonsensical computations.

**Logical operators**

There are also some operations that can be applied to Boolean values themselves. JavaScript supports three logical operators: *and*, *or*, and *not*. These can be used to “reason” about Booleans.

The && operator represents logical *and*. It is a binary operator, and its result is true only if both the values given to it are true.

console.log(true && false)

// → false

console.log(true && true)

// → true

The || operator denotes logical *or*. It produces true if either of the values given to it is true.

console.log(false || true)

// → true

console.log(false || false)

// → false

*Not* is written as an exclamation mark (!). It is a unary operator that flips the value given to it—!true produces false and !false gives true.

When mixing these Boolean operators with arithmetic and other operators, it is not always obvious when parentheses are needed. In practice, you can usually get by with knowing that of the operators we have seen so far, || has the lowest precedence, then comes &&, then the comparison operators (>, ==, and so on), and then the rest. This order has been chosen such that, in typical expressions like the following one, as few parentheses as possible are necessary:

1 + 1 == 2 && 10 \* 10 > 50

The last logical operator I will discuss is not unary, not binary, but *ternary*, operating on three values. It is written with a question mark and a colon, like this:

console.log(true ? 1 : 2);

// → 1

console.log(false ? 1 : 2);

// → 2

This one is called the *conditional* operator (or sometimes just *ternary* operator since it is the only such operator in the language). The value on the left of the question mark “picks” which of the other two values will come out. When it is true, the middle value is chosen, and when it is false, the value on the right comes out.

**Undefined values**

There are two special values, written null and undefined, that are used to denote the absence of a meaningful value. They are themselves values, but they carry no information.

Many operations in the language that don’t produce a meaningful value (you’ll see some later) yield undefined simply because they have to yield *some* value.

The difference in meaning between undefined and null is an accident of JavaScript’s design, and it doesn’t matter most of the time. In the cases where you actually have to concern yourself with these values, I recommend treating them as interchangeable (more on that in a moment).

**Automatic type conversion**

In the introduction, I mentioned that JavaScript goes out of its way to accept almost any program you give it, even programs that do odd things. This is nicely demonstrated by the following expressions:

console.log(8 \* null)

// → 0

console.log("5" - 1)

// → 4

console.log("5" + 1)

// → 51

console.log("five" \* 2)

// → NaN

console.log(false == 0)

// → true

When an operator is applied to the “wrong” type of value, JavaScript will quietly convert that value to the type it wants, using a set of rules that often aren’t what you want or expect. This is called *type coercion*. So the null in the first expression becomes 0, and the "5" in the second expression becomes 5 (from string to number). Yet in the third expression, + tries string concatenation before numeric addition, so the 1 is converted to "1" (from number to string).

When something that doesn’t map to a number in an obvious way (such as "five" or undefined) is converted to a number, the value NaN is produced. Further arithmetic operations on NaN keep producing NaN, so if you find yourself getting one of those in an unexpected place, look for accidental type conversions.

When comparing values of the same type using ==, the outcome is easy to predict: you should get true when both values are the same, except in the case of NaN. But when the types differ, JavaScript uses a complicated and confusing set of rules to determine what to do. In most cases, it just tries to convert one of the values to the other value’s type. However, when null or undefined occurs on either side of the operator, it produces true only if both sides are one of null or undefined.

console.log(null == undefined);

// → true

console.log(null == 0);

// → false

That last piece of behavior is often useful. When you want to test whether a value has a real value instead of null or undefined, you can simply compare it to null with the == (or !=) operator.

But what if you want to test whether something refers to the precise value false? The rules for converting strings and numbers to Boolean values state that 0, NaN, and the empty string ("") count as false, while all the other values count as true. Because of this, expressions like 0 == false and "" == false are also true. For cases like this, where you do *not* want any automatic type conversions to happen, there are two extra operators: === and !==. The first tests whether a value is precisely equal to the other, and the second tests whether it is not precisely equal. So "" === false is false as expected.

I recommend using the three-character comparison operators defensively to prevent unexpected type conversions from tripping you up. But when you’re certain the types on both sides will be the same, there is no problem with using the shorter operators.

**Short-circuiting of logical operators**

The logical operators && and || handle values of different types in a peculiar way. They will convert the value on their left side to Boolean type in order to decide what to do, but depending on the operator and the result of that conversion, they return either the *original* left-hand value or the right-hand value.

The || operator, for example, will return the value to its left when that can be converted to true and will return the value on its right otherwise. This conversion works as you’d expect for Boolean values and should do something analogous for values of other types.

console.log(null || "user")

// → user

console.log("Karl" || "user")

// → Karl

This functionality allows the || operator to be used as a way to fall back on a default value. If you give it an expression that might produce an empty value on the left, the value on the right will be used as a replacement in that case.

The && operator works similarly, but the other way around. When the value to its left is something that converts to false, it returns that value, and otherwise it returns the value on its right.

Another important property of these two operators is that the expression to their right is evaluated only when necessary. In the case of true || X, no matter what X is—even if it’s an expression that does something *terrible*—the result will be true, and X is never evaluated. The same goes for false && X, which is false and will ignore X. This is called *short-circuit evaluation*.

The conditional operator works in a similar way. The first expression is always evaluated, but the second or third value, the one that is not picked, is not.

**Summary**

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 calculator, but not much more. The [next chapter](http://eloquentjavascript.net/02_program_structure.html#program_structure) will start tying these expressions together into basic programs.

**Chapter 2**

**Program Structure**

And my heart glows bright red under my filmy, translucent skin and they have to administer 10cc of JavaScript to get me to come back. (I respond well to toxins in the blood.) Man, that stuff will kick the peaches right out your gills!

\_why, Why's (Poignant) Guide to Ruby

In this chapter, we will start to do things that can actually be called *programming*. We will expand our command of the JavaScript language beyond the nouns and sentence fragments we’ve seen so far, to the point where we can express some meaningful prose.

**Expressions and statements**

In [Chapter 1](http://eloquentjavascript.net/01_values.html#values), we made some values and then applied operators to them to get new values. Creating values like this is an essential part of every JavaScript program, but it is only a part.

A fragment of code that produces a value is called an *expression*. Every value that is written literally (such as 22 or "psychoanalysis") is an expression. An expression between parentheses is also an expression, as is a binary operator applied to two expressions or a unary operator applied to one.

This shows part of the beauty of a language-based interface. Expressions can nest in a way very similar to the way subsentences in human languages are nested—a subsentence can contain its own subsentences, and so on. This allows us to combine expressions to express arbitrarily complex computations.

If an expression corresponds to a sentence fragment, a JavaScript *statement* corresponds to a full sentence in a human language. A program is simply a list of statements.

The simplest kind of statement is an expression with a semicolon after it. This is a program:

edit & run code by clicking it

1;

!false;

It is a useless program, though. An expression can be content to just produce a value, which can then be used by the enclosing expression. A statement stands on its own and amounts to something only if it affects the world. It could display something on the screen—that counts as changing the world—or it could change the internal state of the machine in a way that will affect the statements that come after it. These changes are called *side effects*. The statements in the previous example just produce the values 1 and true and then immediately throw them away. This leaves no impression on the world at all. When executing the program, nothing observable happens.

In some cases, JavaScript allows you to omit the semicolon at the end of a statement. In other cases, it has to be there, or the next line will be treated as part of the same statement. The rules for when it can be safely omitted are somewhat complex and error-prone. In this book, every statement that needs a semicolon will always be terminated by one. I recommend you do the same in your own programs, at least until you’ve learned more about subtleties involved in leaving out semicolons.

**Variables**

How does a program keep an internal state? How does it remember things? We have seen how to produce new values from old values, but this does not change the old values, and the new value has to be immediately used or it will dissipate again. To catch and hold values, JavaScript provides a thing called a *variable*.

var caught = 5 \* 5;

And that gives us our second kind of statement. The special word (*keyword*) var indicates that this sentence is going to define a variable. It is followed by the name of the variable and, if we want to immediately give it a value, by an = operator and an expression.

The previous statement creates a variable called caught and uses it to grab hold of the number that is produced by multiplying 5 by 5.

After a variable has been defined, its name can be used as an expression. The value of such an expression is the value the variable currently holds. Here’s an example:

var ten = 10;

console.log(ten \* ten);

// → 100

Variable names can be any word that isn’t a reserved word (such as var). They may not include spaces. Digits can also be part of variable names—catch22 is a valid name, for example—but the name must not start with a digit. A variable name cannot include punctuation, except for the characters $ and \_.

When a variable points at a value, that does not mean it is tied to that value forever. The = operator can be used at any time on existing variables to disconnect them from their current value and have them point to a new one.

var mood = "light";

console.log(mood);

// → light

mood = "dark";

console.log(mood);

// → dark

You should imagine variables as tentacles, rather than boxes. They do not *contain* values; they *grasp* them—two variables can refer to the same value. A program can access only the values that it still has a hold on. When you need to remember something, you grow a tentacle to hold on to it or you reattach one of your existing tentacles to it.

Let’s look at an example. To remember the number of dollars that Luigi still owes you, you create a variable. And then when he pays back $35, you give this variable a new value.

var luigisDebt = 140;

luigisDebt = luigisDebt - 35;

console.log(luigisDebt);

// → 105

When you define a variable without giving it a value, the tentacle has nothing to grasp, so it ends in thin air. If you ask for the value of an empty variable, you’ll get the value undefined.

A single var statement may define multiple variables. The definitions must be separated by commas.

var one = 1, two = 2;

console.log(one + two);

// → 3

**Keywords and reserved words**

Words with a special meaning, such as var, are *keywords*, and they may not be used as variable names. There are also a number of words that are “reserved for use” in future versions of JavaScript. These are also officially not allowed to be used as variable names, though some JavaScript environments do allow them. The full list of keywords and reserved words is rather long.

break case catch class const continue debugger

default delete do else enum export extends false

finally for function if implements import in

instanceof interface let new null package private

protected public return static super switch this

throw true try typeof var void while with yield

Don’t worry about memorizing these, but remember that this might be the problem when a variable definition does not work as expected.

**The environment**

The collection of variables and their values that exist at a given time is called the *environment*. When a program starts up, this environment is not empty. It always contains variables that are part of the language standard, and most of the time, it has variables that provide ways to interact with the surrounding system. For example, in a browser, there are variables and functions to inspect and influence the currently loaded website and to read mouse and keyboard input.

**Functions**

A lot of the values provided in the default environment have the type *function*. A function is a piece of program wrapped in a value. Such values can be *applied* in order to run the wrapped program. For example, in a browser environment, the variable alert holds a function that shows a little dialog box with a message. It is used like this:

alert("Good morning!");



Executing a function is called *invoking*, *calling*, or *applying* it. You can call a function by putting parentheses after an expression that produces a function value. Usually you’ll directly use the name of the variable that holds the function. The values between the parentheses are given to the program inside the function. In the example, the alert function uses the string that we give it as the text to show in the dialog box. Values given to functions are called *arguments*. The alert function needs only one of them, but other functions might need a different number or different types of arguments.

**The console.log function**

The alert function can be useful as an output device when experimenting, but clicking away all those little windows will get on your nerves. In past examples, we’ve used console.log to output values. Most JavaScript systems (including all modern web browsers and Node.js) provide a console.log function that writes out its arguments to *some* text output device. In browsers, the output lands in the JavaScript console. This part of the browser interface is hidden by default, but most browsers open it when you press F12 or, on Mac, when you press Command-Option-I. If that does not work, search through the menus for an item named “web console” or “developer tools”.

When running the examples, or your own code, on the pages of this book, console.log output will be shown after the example, instead of in the browser’s JavaScript console.

var x = 30;

console.log("the value of x is", x);

// → the value of x is 30

Though variable names cannot contain period characters, console.log clearly has one. This is because console.log isn’t a simple variable. It is actually an expression that retrieves the log property from the value held by the console variable. We will find out exactly what this means in [Chapter 4](http://eloquentjavascript.net/04_data.html#properties).

**Return values**

Showing a dialog box or writing text to the screen is a *side effect*. A lot of functions are useful because of the side effects they produce. Functions may also produce values, and in that case, they don’t need to have a side effect to be useful. For example, the function Math.max takes any number of number values and gives back the greatest.

console.log(Math.max(2, 4));

// → 4

When a function produces a value, it is said to *return* that value. Anything that produces a value is an expression in JavaScript, which means function calls can be used within larger expressions. Here a call to Math.min, which is the opposite of Math.max, is used as an input to the plus operator:

console.log(Math.min(2, 4) + 100);

// → 102

The [next chapter](http://eloquentjavascript.net/03_functions.html#functions) explains how to write your own functions.

**prompt and confirm**

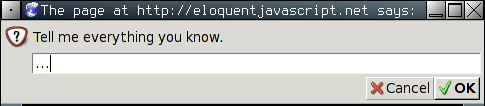
Browser environments contain other functions besides alert for popping up windows. You can ask the user an OK/Cancel question using confirm. This returns a Boolean: true if the user clicks OK and false if the user clicks Cancel.

confirm("Shall we, then?");



The prompt function can be used to ask an “open” question. The first argument is the question, the second one is the text that the user starts with. A line of text can be typed into the dialog window, and the function will return this text as a string.

prompt("Tell me everything you know.", "...");



These two functions aren’t used much in modern web programming, mostly because you have no control over the way the resulting windows look, but they are useful for toy programs and experiments.

**Control flow**

When your program contains more than one statement, the statements are executed, predictably, from top to bottom. As a basic example, this program has two statements. The first one asks the user for a number, and the second, which is executed afterward, shows the square of that number.

var theNumber = Number(prompt("Pick a number", ""));

alert("Your number is the square root of " +

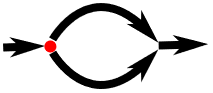
theNumber \* theNumber);

The function Number converts a value to a number. We need that conversion because the result of prompt is a string value, and we want a number. There are similar functions called String and Boolean that convert values to those types.

Here is the rather trivial schematic representation of straight control flow: -🡪

**Conditional execution**

Executing statements in straight-line order isn’t the only option we have. An alternative is *conditional execution*, where we choose between two different routes based on a Boolean value, like this:

Conditional execution is written with the if keyword in JavaScript. In the simple case, we just want some code to be executed if, and only if, a certain condition holds. For example, in the previous program, we might want to show the square of the input only if the input is actually a number.

var theNumber = Number(prompt("Pick a number", ""));

if (!isNaN(theNumber))

alert("Your number is the square root of " +

theNumber \* theNumber);

With this modification, if you enter “cheese”, no output will be shown.

The keyword if executes or skips a statement depending on the value of a Boolean expression. The deciding expression is written after the keyword, between parentheses, followed by the statement to execute.

The isNaN function is a standard JavaScript function that returns true only if the argument it is given is NaN. The Number function happens to return NaN when you give it a string that doesn’t represent a valid number. Thus, the condition translates to “unless theNumber is not-a-number, do this”.

You often won’t just have code that executes when a condition holds true, but also code that handles the other case. This alternate path is represented by the second arrow in the diagram. The else keyword can be used, together with if, to create two separate, alternative execution paths.

var theNumber = Number(prompt("Pick a number", ""));

if (!isNaN(theNumber))

alert("Your number is the square root of " +

theNumber \* theNumber);

else

alert("Hey. Why didn't you give me a number?");

If we have more than two paths to choose from, multiple if/else pairs can be “chained” together. Here’s an example:

var num = Number(prompt("Pick a number", "0"));

if (num < 10)

alert("Small");

else if (num < 100)

alert("Medium");

else

alert("Large");

The program will first check whether num is less than 10. If it is, it chooses that branch, shows "Small", and is done. If it isn’t, it takes the else branch, which itself contains a second if. If the second condition (< 100) holds, that means the number is between 10 and 100, and "Medium" is shown. If it doesn’t, the second, and last, else branch is chosen.

The flow chart for this program looks something like this:

**while and do loops**

Consider a program that prints all even numbers from 0 to 12. One way to write this is as follows:

console.log(0);

console.log(2);

console.log(4);

console.log(6);

console.log(8);

console.log(10);

console.log(12);

That works, but the idea of writing a program is to make something *less* work, not more. If we needed all even numbers less than 1,000, the previous would be unworkable. What we need is a way to repeat some code. This form of control flow is called a *loop*:

Looping control flow allows us to go back to some point in the program where we were before and repeat it with our current program state. If we combine this with a variable that counts, we can do something like this:

var number = 0;

while (number <= 12) {

console.log(number);

number = number + 2;

}

// → 0

// → 2

// … etcetera

A statement starting with the keyword while creates a loop. The word while is followed by an expression in parentheses and then a statement, much like if. The loop executes that statement as long as the expression produces a value that is true when converted to Boolean type.

In this loop, we want to both print the current number and add two to our variable. Whenever we need to execute multiple statements inside a loop, we wrap them in curly braces ({ and }). Braces do for statements what parentheses do for expressions: they group them together, making them count as a single statement. A sequence of statements wrapped in braces is called a *block*.

Many JavaScript programmers wrap every single loop or if body in braces. They do this both for the sake of consistency and to avoid having to add or remove braces when changing the number of statements in the body later. In this book, I will write most single-statement bodies without braces, since I value brevity. You are free to go with whichever style you prefer.

The variable number demonstrates the way a variable can track the progress of a program. Every time the loop repeats, number is incremented by 2. Then, at the beginning of every repetition, it is compared with the number 12 to decide whether the program has done all the work it intended to do.

As an example that actually does something useful, we can now write a program that calculates and shows the value of 210 (2 to the 10th power). We use two variables: one to keep track of our result and one to count how often we have multiplied this result by 2. The loop tests whether the second variable has reached 10 yet and then updates both variables.

var result = 1;

var counter = 0;

while (counter < 10) {

result = result \* 2;

counter = counter + 1;

}

console.log(result);

// → 1024

The counter could also start at 1 and check for <= 10, but, for reasons that will become apparent in [Chapter 4](http://eloquentjavascript.net/04_data.html#array_indexing), it is a good idea to get used to counting from 0.

The do loop is a control structure similar to the while loop. It differs only on one point: a do loop always executes its body at least once, and it starts testing whether it should stop only after that first execution. To reflect this, the test appears after the body of the loop:

do {

var yourName = prompt("Who are you?");

} while (!yourName);

console.log(yourName);

This program will force you to enter a name. It will ask again and again until it gets something that is not an empty string. Applying the ! operator will convert a value to Boolean type before negating it, and all strings except "" convert to true. This means the loop continues going round until you provide a name that is not the empty string.

**Indenting Code**

You’ve probably noticed the spaces I put in front of some statements. In JavaScript, these are not required—the computer will accept the program just fine without them. In fact, even the line breaks in programs are optional. You could write a program as a single long line if you felt like it. The role of the indentation inside blocks is to make the structure of the code stand out. In complex code, where new blocks are opened inside other blocks, it can become hard to see where one block ends and another begins. With proper indentation, the visual shape of a program corresponds to the shape of the blocks inside it. I like to use two spaces for every open block, but tastes differ—some people use four spaces, and some people use tab characters.

**for loops**

Many loops follow the pattern seen in the previous while examples. First, a “counter” variable is created to track the progress of the loop. Then comes a while loop, whose test expression usually checks whether the counter has reached some boundary yet. At the end of the loop body, the counter is updated to track progress.

Because this pattern is so common, JavaScript and similar languages provide a slightly shorter and more comprehensive form, the for loop.

for (var number = 0; number <= 12; number = number + 2)

console.log(number);

// → 0

// → 2

// … etcetera

This program is exactly equivalent to the [earlier](http://eloquentjavascript.net/02_program_structure.html#loops) even-number-printing example. The only change is that all the statements that are related to the “state” of the loop are now grouped together.

The parentheses after a for keyword must contain two semicolons. The part before the first semicolon *initializes* the loop, usually by defining a variable. The second part is the expression that *checks* whether the loop must continue. The final part *updates* the state of the loop after every iteration. In most cases, this is shorter and clearer than a while construct.

Here is the code that computes 210, using for instead of while:

var result = 1;

for (var counter = 0; counter < 10; counter = counter + 1)

result = result \* 2;

console.log(result);

// → 1024

Note that even though no block is opened with a {, the statement in the loop is still indented two spaces to make it clear that it “belongs” to the line before it.

**Breaking Out of a Loop**

Having the loop’s condition produce false is not the only way a loop can finish. There is a special statement called break that has the effect of immediately jumping out of the enclosing loop.

This program illustrates the break statement. It finds the first number that is both greater than or equal to 20 and divisible by 7.

for (var current = 20; ; current++) {

if (current % 7 == 0)

break;

}

console.log(current);

// → 21

Using the remainder (%) operator is an easy way to test whether a number is divisible by another number. If it is, the remainder of their division is zero.

The for construct in the example does not have a part that checks for the end of the loop. This means that the loop will never stop unless the break statement inside is executed.

If you were to leave out that break statement or accidentally write a condition that always produces true, your program would get stuck in an *infinite loop*. A program stuck in an infinite loop will never finish running, which is usually a bad thing.

If you create an infinite loop in one of the examples on these pages, you’ll usually be asked whether you want to stop the script after a few seconds. If that fails, you will have to close the tab that you’re working in, or on some browsers close your whole browser, in order to recover.

The continue keyword is similar to break, in that it influences the progress of a loop. When continue is encountered in a loop body, control jumps out of the body and continues with the loop’s next iteration.

**Updating variables succinctly**

Especially when looping, a program often needs to “update” a variable to hold a value based on that variable’s previous value.

counter = counter + 1;

JavaScript provides a shortcut for this:

counter += 1;

Similar shortcuts work for many other operators, such as result \*= 2 to double result or counter -= 1 to count downward.

This allows us to shorten our counting example a little more.

for (var number = 0; number <= 12; number += 2)

console.log(number);

For counter += 1 and counter -= 1, there are even shorter equivalents: counter++ and counter--.

**Dispatching on a value with switch**

It is common for code to look like this:

if (variable == "value1") action1();

else if (variable == "value2") action2();

else if (variable == "value3") action3();

else defaultAction();

There is a construct called switch that is intended to solve such a “dispatch” in a more direct way. Unfortunately, the syntax JavaScript uses for this (which it inherited from the C/Java line of programming languages) is somewhat awkward—a chain of if statements often looks better. Here is an example:

switch (prompt("What is the weather like?")) {

case "rainy":

console.log("Remember to bring an umbrella.");

break;

case "sunny":

console.log("Dress lightly.");

case "cloudy":

console.log("Go outside.");

break;

default:

console.log("Unknown weather type!");

break;

}

You may put any number of case labels inside the block opened by switch. The program will jump to the label that corresponds to the value that switch was given or to default if no matching value is found. It starts executing statements there, even if they’re under another label, until it reaches a break statement. In some cases, such as the "sunny" case in the example, this can be used to share some code between cases (it recommends going outside for both sunny and cloudy weather). But beware: it is easy to forget such a break, which will cause the program to execute code you do not want executed.

**Capitalization**

Variable names may not contain spaces, yet it is often helpful to use multiple words to clearly describe what the variable represents. These are pretty much your choices for writing a variable name with several words in it:

fuzzylittleturtle

fuzzy\_little\_turtle

FuzzyLittleTurtle

fuzzyLittleTurtle

The first style can be hard to read. Personally, I like the look of the underscores, though that style is a little painful to type. The standard JavaScript functions, and most JavaScript programmers, follow the bottom style—they capitalize every word except the first. It is not hard to get used to little things like that, and code with mixed naming styles can be jarring to read, so we will just follow this convention.

In a few cases, such as the Number function, the first letter of a variable is also capitalized. This was done to mark this function as a constructor. What a constructor is will become clear in [Chapter 6](http://eloquentjavascript.net/06_object.html#constructors). For now, the important thing is not to be bothered by this apparent lack of consistency.

**Comments**

Often, raw code does not convey all the information you want a program to convey to human readers, or it conveys it in such a cryptic way that people might not understand it. At other times, you might just feel poetic or want to include some thoughts as part of your program. This is what *comments* are for.

A comment is a piece of text that is part of a program but is completely ignored by the computer. JavaScript has two ways of writing comments. To write a single-line comment, you can use two slash characters (//) and then the comment text after it.

var accountBalance = calculateBalance(account);

// It's a green hollow where a river sings

accountBalance.adjust();

// Madly catching white tatters in the grass.

var report = new Report();

// Where the sun on the proud mountain rings:

addToReport(accountBalance, report);

// It's a little valley, foaming like light in a glass.

A // comment goes only to the end of the line. A section of text between /\* and \*/ will be ignored, regardless of whether it contains line breaks. This is often useful for adding blocks of information about a file or a chunk of program.

/\*

I first found this number scrawled on the back of one of

my notebooks a few years ago. Since then, it has often

dropped by, showing up in phone numbers and the serial

numbers of products that I've bought. It obviously likes

me, so I've decided to keep it.

\*/

var myNumber = 11213;

**Summary**

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 executed 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.

Variables 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 variables that are defined. JavaScript systems always put a number of useful standard variables into your environment.

Functions are special values that encapsulate a piece of program. You can invoke them by writing functionName(argument1, argument2). Such a function call is an expression, and may produce a value.

# Chapter 3

# Functions

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

Donald Knuth

You’ve seen function values, such as alert, and how to call them. Functions are the bread and butter of JavaScript programming. The concept of wrapping a piece of program in a value has many uses. It is a tool to structure larger programs, to reduce repetition, to associate names with subprograms, and to isolate these subprograms from each other.

The most obvious application of functions is defining new vocabulary. Creating new words in regular, human-language prose is usually bad style. But in programming, it is indispensable.

Typical adult English speakers have some 20,000 words in their vocabulary. Few programming languages come with 20,000 commands built in. And the vocabulary that is available tends to be more precisely defined, and thus less flexible, than in human language. Therefore, we usually have to add some of our own vocabulary to avoid repeating ourselves too much.

## Defining a function

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

edit & run code by clicking it

var square = function(x) {

return x \* x;

};

console.log(square(12));

// → 144

A function is created by an expression that starts with the keyword function. Functions have a set of parameters (in this case, only x) and a body, which contains the statements that are to be executed when the function is called. The function body must always be wrapped in braces, even when it consists of only a single statement (as in the previous example).

A function can have multiple parameters or no parameters at all. In the following example, makeNoise does not list any parameter names, whereas power lists two:

var makeNoise = function() {

console.log("Pling!");

};

makeNoise();

// → Pling!

var power = function(base, exponent) {

var result = 1;

for (var count = 0; count < exponent; count++)

result \*= base;

return result;

};

console.log(power(2, 10));

// → 1024

Some functions produce a value, such as power and square, and some don’t, such as makeNoise, which produces only a side effect. A return statement determines the value the function returns. When control comes across such a statement, it immediately jumps out of the current function and gives the returned value to the code that called the function. The return keyword without an expression after it will cause the function to return undefined.

## Parameters and scopes

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

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

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

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

var x = "outside";

var f1 = function() {

var x = "inside f1";

};

f1();

console.log(x);

// → outside

var f2 = function() {

x = "inside f2";

};

f2();

console.log(x);

// → inside f2

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

## Nested scope

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

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

var landscape = function() {

var result = "";

var flat = function(size) {

for (var count = 0; count < size; count++)

result += "\_";

};

var mountain = function(size) {

result += "/";

for (var count = 0; count < size; count++)

result += "'";

result += "\\";

};

flat(3);

mountain(4);

flat(6);

mountain(1);

flat(1);

return result;

};

console.log(landscape());

// → \_\_\_/''''\\_\_\_\_\_\_/'\\_

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

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

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

var something = 1;

{

var something = 2;

// Do stuff with variable something...

}

// Outside of the block again...

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

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

## Functions as values

Function variables usually simply act as names for a specific piece of the program. Such a variable is defined once and never changed. This makes it easy to start confusing the function and its name.

But the two are different. A function value can do all the things that other values can do—you can use it in arbitrary expressions, not just call it. It is possible to store a function value in a new place, pass it as an argument to a function, and so on. Similarly, a variable that holds a function is still just a regular variable and can be assigned a new value, like so:

var launchMissiles = function(value) {

missileSystem.launch("now");

};

if (safeMode)

launchMissiles = function(value) {/\* do nothing \*/};

In [Chapter 5](http://eloquentjavascript.net/05_higher_order.html#higher_order), we will discuss the wonderful things that can be done by passing around function values to other functions.

## Declaration notation

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

function square(x) {

return x \* x;

}

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

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

function future() {

return "We STILL have no flying cars.";

}

This code works, even though the function is defined below the code that uses it. This is because function declarations are not part of the regular top-to-bottom flow of control. They are conceptually moved to the top of their scope and can be used by all the code in that scope. This is sometimes useful because it gives us the freedom to order code in a way that seems meaningful, without worrying about having to define all functions above their first use.

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

function example() {

function a() {} // Okay

if (something) {

function b() {} // Danger!

}

}

## The call stack

It will be helpful to take a closer look at the way control flows through functions. Here is a simple program that makes a few function calls:

function greet(who) {

console.log("Hello " + who);

}

greet("Harry");

console.log("Bye");

A run through this program goes roughly like this: the call to greet causes control to jump to the start of that function (line 2). It calls console.log (a built-in browser function), which takes control, does its job, and then returns control to line 2. Then it reaches the end of the greet function, so it returns to the place that called it, at line 4. The line after that calls console.log again.

We could show the flow of control schematically like this:

top

greet

console.log

greet

top

console.log

top

Because a function has to jump back to the place of the call when it returns, the computer must remember the context from which the function was called. In one case, console.log has to jump back to the greet function. In the other case, it jumps back to the end of the program.

The place where the computer stores this context is the call stack. Every time a function is called, the current context is put on top of this “stack”. When the function returns, it removes the top context from the stack and uses it to continue execution.

Storing this stack requires space in the computer’s memory. When the stack grows too big, the computer will fail with a message like “out of stack space” or “too much recursion”. The following code illustrates this by asking the computer a really hard question, which causes an infinite back-and-forth between two functions. Rather, it would be infinite, if the computer had an infinite stack. As it is, we will run out of space, or “blow the stack”.

function chicken() {

return egg();

}

function egg() {

return chicken();

}

console.log(chicken() + " came first.");

// → ??

## Optional Arguments

The following code is allowed and executes without any problem:

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

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

JavaScript is extremely broad-minded about the number of arguments you pass to a function. If you pass too many, the extra ones are ignored. If you pass too few, the missing parameters simply get assigned the value undefined.

The downside of this is that it is possible—likely, even—that you’ll accidentally pass the wrong number of arguments to functions and no one will tell you about it.

The upside is that this behavior can be used to have a function take “optional” arguments. For example, the following version of power can be called either with two arguments or with a single argument, in which case the exponent is assumed to be two, and the function behaves like square.

function power(base, exponent) {

if (exponent == undefined)

exponent = 2;

var result = 1;

for (var count = 0; count < exponent; count++)

result \*= base;

return result;

}

console.log(power(4));

// → 16

console.log(power(4, 3));

// → 64

In the [next chapter](http://eloquentjavascript.net/04_data.html#arguments_object), we will see a way in which a function body can get at the exact list of arguments that were passed. This is helpful because it makes it possible for a function to accept any number of arguments. For example, console.log makes use of this—it outputs all of the values it is given.

console.log("R", 2, "D", 2);

// → R 2 D 2

## Closure

The ability to treat functions as values, combined with the fact that local variables are “re-created” every time a function is called, brings up an interesting question. What happens to local variables when the function call that created them is no longer active?

The following code shows an example of this. It defines a function, wrapValue, which creates a local variable. It then returns a function that accesses and returns this local variable.

function wrapValue(n) {

var localVariable = n;

return function() { return localVariable; };

}

var wrap1 = wrapValue(1);

var wrap2 = wrapValue(2);

console.log(wrap1());

// → 1

console.log(wrap2());

// → 2

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

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

With a slight change, we can turn the previous example into a way to create functions that multiply by an arbitrary amount.

function multiplier(factor) {

return function(number) {

return number \* factor;

};

}

var twice = multiplier(2);

console.log(twice(5));

// → 10

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

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

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

## Recursion

It is perfectly okay for a function to call itself, as long as it takes care not to overflow the stack. A function that calls itself is called recursive. Recursion allows some functions to be written in a different style. Take, for example, this alternative implementation of power:

function power(base, exponent) {

if (exponent == 0)

return 1;

else

return base \* power(base, exponent - 1);

}

console.log(power(2, 3));

// → 8

This is rather close to the way mathematicians define exponentiation and arguably describes the concept in a more elegant way than the looping variant does. The function calls itself multiple times with different arguments to achieve the repeated multiplication.

But this implementation has one important problem: in typical JavaScript implementations, it’s about 10 times slower than the looping version. Running through a simple loop is a lot cheaper than calling a function multiple times.

The dilemma of speed versus elegance is an interesting one. You can see it as a kind of continuum between human-friendliness and machine-friendliness. Almost any program can be made faster by making it bigger and more convoluted. The programmer must decide on an appropriate balance.

In the case of the [earlier](http://eloquentjavascript.net/03_functions.html#power) power function, the inelegant (looping) version is still fairly simple and easy to read. It doesn’t make much sense to replace it with the recursive version. Often, though, a program deals with such complex concepts that giving up some efficiency in order to make the program more straightforward becomes an attractive choice.

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

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

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

But recursion is not always just a less-efficient alternative to looping. Some problems are much easier to solve with recursion than with loops. Most often these are problems that require exploring or processing several “branches”, each of which might branch out again into more branches.

Consider this puzzle: by starting from the number 1 and repeatedly either adding 5 or multiplying by 3, an infinite amount of new numbers can be produced. How would you write a function that, given a number, tries to find a sequence of such additions and multiplications that produce that number? For example, the number 13 could be reached by first multiplying by 3 and then adding 5 twice, whereas the number 15 cannot be reached at all.

Here is a recursive solution:

function findSolution(target) {

function find(current, history) {

if (current == target)

return history;

else if (current > target)

return null;

else

return find(current + 5, "(" + history + " + 5)") ||

find(current \* 3, "(" + history + " \* 3)");

}

return find(1, "1");

}

console.log(findSolution(24));

// → (((1 \* 3) + 5) \* 3)

Note that this program doesn’t necessarily find the shortest sequence of operations. It is satisfied when it finds any sequence at all.

I don’t necessarily expect you to see how it works right away. But let’s work through it, since it makes for a great exercise in recursive thinking.

The inner function find does the actual recursing. It takes two arguments—the current number and a string that records how we reached this number—and returns either a string that shows how to get to the target or null.

To do this, the function performs one of three actions. If the current number is the target number, the current history is a way to reach that target, so it is simply returned. If the current number is greater than the target, there’s no sense in further exploring this history since both adding and multiplying will only make the number bigger. And finally, if we’re still below the target, the function tries both possible paths that start from the current number, by calling itself twice, once for each of the allowed next steps. If the first call returns something that is not null, it is returned. Otherwise, the second call is returned—regardless of whether it produces a string or null.

To better understand how this function produces the effect we’re looking for, let’s look at all the calls to find that are made when searching for a solution for the number 13.

find(1, "1")

find(6, "(1 + 5)")

find(11, "((1 + 5) + 5)")

find(16, "(((1 + 5) + 5) + 5)")

too big

find(33, "(((1 + 5) + 5) \* 3)")

too big

find(18, "((1 + 5) \* 3)")

too big

find(3, "(1 \* 3)")

find(8, "((1 \* 3) + 5)")

find(13, "(((1 \* 3) + 5) + 5)")

found!

The indentation suggests the depth of the call stack. The first time find is called it calls itself twice to explore the solutions that start with (1 + 5) and (1 \* 3). The first call tries to find a solution that starts with (1 + 5) and, using recursion, explores every solution that yields a number less than or equal to the target number. Since it doesn’t find a solution that hits the target, it returns null back to the first call. There the || operator causes the call that explores (1 \* 3) to happen. This search has more luck because its first recursive call, through yet another recursive call, hits upon the target number, 13. This innermost recursive call returns a string, and each of the || operators in the intermediate calls pass that string along, ultimately returning our solution.

## Growing functions

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

The first is that you find yourself writing very similar code multiple times. We want to avoid doing that since having more code means more space for mistakes to hide and more material to read for people trying to understand the program. So we take the repeated functionality, find a good name for it, and put it into a function.

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

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

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

007 Cows

011 Chickens

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

function printFarmInventory(cows, chickens) {

var cowString = String(cows);

while (cowString.length < 3)

cowString = "0" + cowString;

console.log(cowString + " Cows");

var chickenString = String(chickens);

while (chickenString.length < 3)

chickenString = "0" + chickenString;

console.log(chickenString + " Chickens");

}

printFarmInventory(7, 11);

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

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

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

function printZeroPaddedWithLabel(number, label) {

var numberString = String(number);

while (numberString.length < 3)

numberString = "0" + numberString;

console.log(numberString + " " + label);

}

function printFarmInventory(cows, chickens, pigs) {

printZeroPaddedWithLabel(cows, "Cows");

printZeroPaddedWithLabel(chickens, "Chickens");

printZeroPaddedWithLabel(pigs, "Pigs");

}

printFarmInventory(7, 11, 3);

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

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

function zeroPad(number, width) {

var string = String(number);

while (string.length < width)

string = "0" + string;

return string;

}

function printFarmInventory(cows, chickens, pigs) {

console.log(zeroPad(cows, 3) + " Cows");

console.log(zeroPad(chickens, 3) + " Chickens");

console.log(zeroPad(pigs, 3) + " Pigs");

}

printFarmInventory(7, 16, 3);

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

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

A useful principle is not to add cleverness unless you are absolutely sure you’re going to need it. It can be tempting to write general “frameworks” for every little bit of functionality you come across. Resist that urge. You won’t get any real work done, and you’ll end up writing a lot of code that no one will ever use.

## Functions and side effects

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

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

A pure function is a specific kind of value-producing function that not only has no side effects but also doesn’t rely on side effects from other code—for example, it doesn’t read global variables that are occasionally changed by other code. A pure function has the pleasant property that, when called with the same arguments, it always produces the same value (and doesn’t do anything else). This makes it easy to reason about. A call to such a function can be mentally substituted by its result, without changing the meaning of the code. When you are not sure that a pure function is working correctly, you can test it by simply calling it, and know that if it works in that context, it will work in any context. Nonpure functions might return different values based on all kinds of factors and have side effects that might be hard to test and think about.

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

## Summary

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

// Create a function value f

var f = function(a) {

console.log(a + 2);

};

// Declare g to be a function

function g(a, b) {

return a \* b \* 3.5;

}

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

Separating the tasks your program performs into different functions is helpful. You won’t have to repeat yourself as much, and functions can make a program more readable by grouping code into conceptual chunks, in the same way that chapters and sections help organize regular text.

# Chapter 4

# Data Structures: Objects and Arrays

On two occasions I have been asked, ‘Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?’ [...] I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.

Charles Babbage, Passages from the Life of a Philosopher (1864)

Numbers, Booleans, and strings are the bricks that data structures are built from. But you can’t make much of a house out of a single brick. Objects allow us to group values—including other objects—together and thus build more complex structures.

The programs we have built so far have been seriously hampered by the fact that they were operating only on simple data types. This chapter will add a basic understanding of data structures to your toolkit. By the end of it, you’ll know enough to start writing some useful programs.

The chapter will work through a more or less realistic programming example, introducing concepts as they apply to the problem at hand. The example code will often build on functions and variables that were introduced earlier in the text.

## The weresquirrel

Every now and then, usually between eight and ten in the evening, Jacques finds himself transforming into a small furry rodent with a bushy tail.

On one hand, Jacques is quite glad that he doesn’t have classic lycanthropy. Turning into a squirrel tends to cause fewer problems than turning into a wolf. Instead of having to worry about accidentally eating the neighbor (that would be awkward), he worries about being eaten by the neighbor’s cat. After two occasions where he woke up on a precariously thin branch in the crown of an oak, naked and disoriented, he has taken to locking the doors and windows of his room at night and putting a few walnuts on the floor to keep himself busy.

That takes care of the cat and oak problems. But Jacques still suffers from his condition. The irregular occurrences of the transformation make him suspect that they might be triggered by something. For a while, he believed that it happened only on days when he had touched trees. So he stopped touching trees entirely and even avoided going near them. But the problem persisted.

Switching to a more scientific approach, Jacques intends to start keeping a daily log of everything he did that day and whether he changed form. With this data he hopes to narrow down the conditions that trigger the transformations.

The first thing he does is design a data structure to store this information.

## Data sets

To work with a chunk of digital data, we’ll first have to find a way to represent it in our machine’s memory. Say, as a simple example, that we want to represent a collection of numbers: 2, 3, 5, 7, and 11.

We could get creative with strings—after all, strings can be any length, so we can put a lot of data into them—and use "2 3 5 7 11" as our representation. But this is awkward. You’d have to somehow extract the digits and convert them back to numbers to access them.

Fortunately, JavaScript provides a data type specifically for storing sequences of values. It is called an array and is written as a list of values between square brackets, separated by commas.

edit & run code by clicking it

var listOfNumbers = [2, 3, 5, 7, 11];

console.log(listOfNumbers[2]);

// → 5

console.log(listOfNumbers[2 - 1]);

// → 3

The notation for getting at the elements inside an array also uses square brackets. A pair of square brackets immediately after an expression, with another expression inside of them, will look up the element in the left-hand expression that corresponds to the index given by the expression in the brackets.

The first index of an array is zero, not one. So the first element can be read with listOfNumbers[0]. If you don’t have a programming background, this convention might take some getting used to. But zero-based counting has a long tradition in technology, and as long as this convention is followed consistently (which it is, in JavaScript), it works well.

## Properties

We’ve seen a few suspicious-looking expressions like myString.length (to get the length of a string) and Math.max (the maximum function) in past examples. These are expressions that access a property of some value. In the first case, we access the length property of the value in myString. In the second, we access the property named max in the Math object (which is a collection of mathematics-related values and functions).

Almost all JavaScript values have properties. The exceptions are null and undefined. If you try to access a property on one of these nonvalues, you get an error.

null.length;

// → TypeError: Cannot read property 'length' of null

The two most common ways to access properties in JavaScript are with a dot and with square brackets. Both value.x and value[x] access a property on value—but not necessarily the same property. The difference is in how x is interpreted. When using a dot, the part after the dot must be a valid variable name, and it directly names the property. When using square brackets, the expression between the brackets is evaluated to get the property name. Whereas value.x fetches the property of value named “x”, value[x] tries to evaluate the expression x and uses the result as the property name.

So if you know that the property you are interested in is called “length”, you say value.length. If you want to extract the property named by the value held in the variable i, you say value[i]. And because property names can be any string, if you want to access a property named “2” or “John Doe”, you must use square brackets: value[2] or value["John Doe"]. This is the case even though you know the precise name of the property in advance, because neither “2” nor “John Doe” is a valid variable name and so cannot be accessed through dot notation.

The elements in an array are stored in properties. Because the names of these properties are numbers and we often need to get their name from a variable, we have to use the bracket syntax to access them. The length property of an array tells us how many elements it contains. This property name is a valid variable name, and we know its name in advance, so to find the length of an array, you typically write array.length because that is easier to write than array["length"].

## Methods

Both string and array objects contain, in addition to the length property, a number of properties that refer to function values.

var doh = "Doh";

console.log(typeof doh.toUpperCase);

// → function

console.log(doh.toUpperCase());

// → DOH

Every string has a toUpperCase property. When called, it will return a copy of the string, in which all letters have been converted to uppercase. There is also toLowerCase. You can guess what that does.

Interestingly, even though the call to toUpperCase does not pass any arguments, the function somehow has access to the string "Doh", the value whose property we called. How this works is described in [Chapter 6](http://eloquentjavascript.net/06_object.html#obj_methods).

Properties that contain functions are generally called methods of the value they belong to. As in, “toUpperCase is a method of a string”.

This example demonstrates some methods that array objects have:

var mack = [];

mack.push("Mack");

mack.push("the", "Knife");

console.log(mack);

// → ["Mack", "the", "Knife"]

console.log(mack.join(" "));

// → Mack the Knife

console.log(mack.pop());

// → Knife

console.log(mack);

// → ["Mack", "the"]

The push method can be used to add values to the end of an array. The pop method does the opposite: it removes the value at the end of the array and returns it. An array of strings can be flattened to a single string with the join method. The argument given to join determines the text that is glued between the array’s elements.

## Objects

Back to the weresquirrel. A set of daily log entries can be represented as an array. But the entries do not consist of just a number or a string—each entry needs to store a list of activities and a Boolean value that indicates whether Jacques turned into a squirrel. Ideally, we would like to group these values together into a single value and then put these grouped values into an array of log entries.

Values of the type object are arbitrary collections of properties, and we can add or remove these properties as we please. One way to create an object is by using a curly brace notation.

var day1 = {

squirrel: false,

events: ["work", "touched tree", "pizza", "running",

"television"]

};

console.log(day1.squirrel);

// → false

console.log(day1.wolf);

// → undefined

day1.wolf = false;

console.log(day1.wolf);

// → false

Inside the curly braces, we can give a list of properties separated by commas. Each property is written as a name, followed by a colon, followed by an expression that provides a value for the property. Spaces and line breaks are not significant. When an object spans multiple lines, indenting it like in the previous example improves readability. Properties whose names are not valid variable names or valid numbers have to be quoted.

var descriptions = {

work: "Went to work",

"touched tree": "Touched a tree"

};

This means that curly braces have two meanings in JavaScript. At the start of a statement, they start a block of statements. In any other position, they describe an object. Fortunately, it is almost never useful to start a statement with a curly-brace object, and in typical programs, there is no ambiguity between these two uses.

Reading a property that doesn’t exist will produce the value undefined, which happens the first time we try to read the wolf property in the previous example.

It is possible to assign a value to a property expression with the = operator. This will replace the property’s value if it already existed or create a new property on the object if it didn’t.

To briefly return to our tentacle model of variable bindings—property bindings are similar. They grasp values, but other variables and properties might be holding onto those same values. You may think of objects as octopuses with any number of tentacles, each of which has a name inscribed on it.

The delete operator cuts off a tentacle from such an octopus. It is a unary operator that, when applied to a property access expression, will remove the named property from the object. This is not a common thing to do, but it is possible.

var anObject = {left: 1, right: 2};

console.log(anObject.left);

// → 1

delete anObject.left;

console.log(anObject.left);

// → undefined

console.log("left" in anObject);

// → false

console.log("right" in anObject);

// → true

The binary in operator, when applied to a string and an object, returns a Boolean value that indicates whether that object has that property. The difference between setting a property to undefined and actually deleting it is that, in the first case, the object still has the property (it just doesn’t have a very interesting value), whereas in the second case the property is no longer present and in will return false.

Arrays, then, are just a kind of object specialized for storing sequences of things. If you evaluate typeof [1, 2], this produces "object". You can see them as long, flat octopuses with all their arms in a neat row, labeled with numbers.

So we can represent Jacques’ journal as an array of objects.

var journal = [

{events: ["work", "touched tree", "pizza",

"running", "television"],

squirrel: false},

{events: ["work", "ice cream", "cauliflower",

"lasagna", "touched tree", "brushed teeth"],

squirrel: false},

{events: ["weekend", "cycling", "break",

"peanuts", "beer"],

squirrel: true},

/\* and so on... \*/

];

## Mutability

We will get to actual programming real soon now. But first, there’s one last piece of theory to understand.

We’ve seen that object values can be modified. The types of values discussed in earlier chapters, such as numbers, strings, and Booleans, are all immutable—it is impossible to change an existing value of those types. You can combine them and derive new values from them, but when you take a specific string value, that value will always remain the same. The text inside it cannot be changed. If you have reference to a string that contains "cat", it is not possible for other code to change a character in that string to make it spell "rat".

With objects, on the other hand, the content of a value can be modified by changing its properties.

When we have two numbers, 120 and 120, we can consider them precisely the same number, whether or not they refer to the same physical bits. But with objects, there is a difference between having two references to the same object and having two different objects that contain the same properties. Consider the following code:

var object1 = {value: 10};

var object2 = object1;

var object3 = {value: 10};

console.log(object1 == object2);

// → true

console.log(object1 == object3);

// → false

object1.value = 15;

console.log(object2.value);

// → 15

console.log(object3.value);

// → 10

The object1 and object2 variables grasp the same object, which is why changing object1 also changes the value of object2. The variable object3 points to a different object, which initially contains the same properties as object1 but lives a separate life.

JavaScript’s == operator, when comparing objects, will return true only if both objects are precisely the same value. Comparing different objects will return false, even if they have identical contents. There is no “deep” comparison operation built into JavaScript, which looks at object’s contents, but it is possible to write it yourself (which will be one of the [exercises](http://eloquentjavascript.net/04_data.html#exercise_deep_compare) at the end of this chapter).

## The lycanthrope’s log

So Jacques starts up his JavaScript interpreter and sets up the environment he needs to keep his journal.

var journal = [];

function addEntry(events, didITurnIntoASquirrel) {

journal.push({

events: events,

squirrel: didITurnIntoASquirrel

});

}

And then, every evening at ten—or sometimes the next morning, after climbing down from the top shelf of his bookcase—he records the day.

addEntry(["work", "touched tree", "pizza", "running",

"television"], false);

addEntry(["work", "ice cream", "cauliflower", "lasagna",

"touched tree", "brushed teeth"], false);

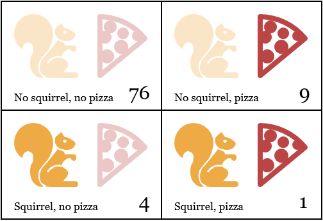
addEntry(["weekend", "cycling", "break", "peanuts",

"beer"], true);

Once he has enough data points, he intends to compute the correlation between his squirrelification and each of the day’s events and ideally learn something useful from those correlations.

Correlation is a measure of dependence between variables (“variables” in the statistical sense, not the JavaScript sense). It is usually expressed as a coefficient that ranges from -1 to 1. Zero correlation means the variables are not related, whereas a correlation of one indicates that the two are perfectly related—if you know one, you also know the other. Negative one also means that the variables are perfectly related but that they are opposites—when one is true, the other is false.

For binary (Boolean) variables, the phi coefficient (ϕ) provides a good measure of correlation and is relatively easy to compute. To compute ϕ, we need a table n that contains the number of times the various combinations of the two variables were observed. For example, we could take the event of eating pizza and put that in a table like this:

ϕ can be computed using the following formula, where n refers to the table:

|  |  |
| --- | --- |
| ϕ = | n11n00 - n10n01  √ n1•n0•n•1n•0 |

The notation n01 indicates the number of measurements where the first variable (squirrelness) is false (0) and the second variable (pizza) is true (1). In this example, n01 is 9.

The value n1• refers to the sum of all measurements where the first variable is true, which is 5 in the example table. Likewise, n•0 refers to the sum of the measurements where the second variable is false.

So for the pizza table, the part above the division line (the dividend) would be 1×76 - 4×9 = 40, and the part below it (the divisor) would be the square root of 5×85×10×80, or √340000. This comes out to ϕ ≈ 0.069, which is tiny. Eating pizza does not appear to have influence on the transformations.

## Computing correlation

We can represent a two-by-two table in JavaScript with a four-element array ([76, 9, 4, 1]). We could also use other representations, such as an array containing two two-element arrays ([[76, 9], [4, 1]]) or an object with property names like "11" and "01", but the flat array is simple and makes the expressions that access the table pleasantly short. We’ll interpret the indices to the array as two-bit binary number, where the leftmost (most significant) digit refers to the squirrel variable and the rightmost (least significant) digit refers to the event variable. For example, the binary number 10 refers to the case where Jacques did turn into a squirrel, but the event (say, "pizza") didn’t occur. This happened four times. And since binary 10 is 2 in decimal notation, we will store this number at index 2 of the array.

This is the function that computes the ϕ coefficient from such an array:

function phi(table) {

return (table[3] \* table[0] - table[2] \* table[1]) /

Math.sqrt((table[2] + table[3]) \*

(table[0] + table[1]) \*

(table[1] + table[3]) \*

(table[0] + table[2]));

}

console.log(phi([76, 9, 4, 1]));

// → 0.068599434

This is simply a direct translation of the ϕ formula into JavaScript. Math.sqrt is the square root function, as provided by the Math object in a standard JavaScript environment. We have to sum two fields from the table to get fields like n1• because the sums of rows or columns are not stored directly in our data structure.

Jacques kept his journal for three months. The resulting data set is available in the coding sandbox for this chapter, where it is stored in the JOURNAL variable, and in a downloadable [file](http://eloquentjavascript.net/code/jacques_journal.js).

To extract a two-by-two table for a specific event from this journal, we must loop over all the entries and tally up how many times the event occurs in relation to squirrel transformations.

function hasEvent(event, entry) {

return entry.events.indexOf(event) != -1;

}

function tableFor(event, journal) {

var table = [0, 0, 0, 0];

for (var i = 0; i < journal.length; i++) {

var entry = journal[i], index = 0;

if (hasEvent(event, entry)) index += 1;

if (entry.squirrel) index += 2;

table[index] += 1;

}

return table;

}

console.log(tableFor("pizza", JOURNAL));

// → [76, 9, 4, 1]

The hasEvent function tests whether an entry contains a given event. Arrays have an indexOf method that tries to find a given value (in this case, the event name) in the array and returns the index at which it was found or -1 if it wasn’t found. So if the call to indexOf doesn’t return -1, then we know the event was found in the entry.

The body of the loop in tableFor figures out which box in the table each journal entry falls into by checking whether the entry contains the specific event it’s interested in and whether the event happens alongside a squirrel incident. The loop then adds one to the number in the array that corresponds to this box on the table.

We now have the tools we need to compute individual correlations. The only step remaining is to find a correlation for every type of event that was recorded and see whether anything stands out. But how should we store these correlations once we compute them?

## Objects as maps

One possible way is to store all the correlations in an array, using objects with name and value properties. But that makes looking up the correlation for a given event somewhat cumbersome: you’d have to loop over the whole array to find the object with the right name. We could wrap this lookup process in a function, but we would still be writing more code, and the computer would be doing more work than necessary.

A better way is to use object properties named after the event types. We can use the square bracket access notation to create and read the properties and can use the in operator to test whether a given property exists.

var map = {};

function storePhi(event, phi) {

map[event] = phi;

}

storePhi("pizza", 0.069);

storePhi("touched tree", -0.081);

console.log("pizza" in map);

// → true

console.log(map["touched tree"]);

// → -0.081

A map is a way to go from values in one domain (in this case, event names) to corresponding values in another domain (in this case, ϕ coefficients).

There are a few potential problems with using objects like this, which we will discuss in [Chapter 6](http://eloquentjavascript.net/06_object.html#prototypes), but for the time being, we won’t worry about those.

What if we want to find all the events for which we have stored a coefficient? The properties don’t form a predictable series, like they would in an array, so we cannot use a normal for loop. JavaScript provides a loop construct specifically for going over the properties of an object. It looks a little like a normal for loop but distinguishes itself by the use of the word in.

for (var event in map)

console.log("The correlation for '" + event +

"' is " + map[event]);

// → The correlation for 'pizza' is 0.069

// → The correlation for 'touched tree' is -0.081

## The final analysis

To find all the types of events that are present in the data set, we simply process each entry in turn and then loop over the events in that entry. We keep an object phis that has correlation coefficients for all the event types we have seen so far. Whenever we run across a type that isn’t in the phis object yet, we compute its correlation and add it to the object.

function gatherCorrelations(journal) {

var phis = {};

for (var entry = 0; entry < journal.length; entry++) {

var events = journal[entry].events;

for (var i = 0; i < events.length; i++) {

var event = events[i];

if (!(event in phis))

phis[event] = phi(tableFor(event, journal));

}

}

return phis;

}

var correlations = gatherCorrelations(JOURNAL);

console.log(correlations.pizza);

// → 0.068599434

Let’s see what came out.

for (var event in correlations)

console.log(event + ": " + correlations[event]);

// → carrot: 0.0140970969

// → exercise: 0.0685994341

// → weekend: 0.1371988681

// → bread: -0.0757554019

// → pudding: -0.0648203724

// and so on...

Most correlations seem to lie close to zero. Eating carrots, bread, or pudding apparently does not trigger squirrel-lycanthropy. It does seem to occur somewhat more often on weekends, however. Let’s filter the results to show only correlations greater than 0.1 or less than -0.1.

for (var event in correlations) {

var correlation = correlations[event];

if (correlation > 0.1 || correlation < -0.1)

console.log(event + ": " + correlation);

}

// → weekend: 0.1371988681

// → brushed teeth: -0.3805211953

// → candy: 0.1296407447

// → work: -0.1371988681

// → spaghetti: 0.2425356250

// → reading: 0.1106828054

// → peanuts: 0.5902679812

A-ha! There are two factors whose correlation is clearly stronger than the others. Eating peanuts has a strong positive effect on the chance of turning into a squirrel, whereas brushing his teeth has a significant negative effect.

Interesting. Let’s try something.

for (var i = 0; i < JOURNAL.length; i++) {

var entry = JOURNAL[i];

if (hasEvent("peanuts", entry) &&

!hasEvent("brushed teeth", entry))

entry.events.push("peanut teeth");

}

console.log(phi(tableFor("peanut teeth", JOURNAL)));

// → 1

Well, that’s unmistakable! The phenomenon occurs precisely when Jacques eats peanuts and fails to brush his teeth. If only he weren’t such a slob about dental hygiene, he’d have never even noticed his affliction.

Knowing this, Jacques simply stops eating peanuts altogether and finds that this completely puts an end to his transformations.

All is well with Jacques for a while. But a few years later, he loses his job and is eventually forced to take employment with a circus, where he performs as The Incredible Squirrelman by stuffing his mouth with peanut butter before every show. One day, fed up with this pitiful existence, Jacques fails to change back into his human form, hops through a crack in the circus tent, and vanishes into the forest. He is never seen again.

## Further arrayology

Before finishing up this chapter, I want to introduce you to a few more object-related concepts. We’ll start by introducing some generally useful array methods.

We saw push and pop, which add and remove elements at the end of an array, [earlier](http://eloquentjavascript.net/04_data.html#array_methods) in this chapter. The corresponding methods for adding and removing things at the start of an array are called unshift and shift.

var todoList = [];

function rememberTo(task) {

todoList.push(task);

}

function whatIsNext() {

return todoList.shift();

}

function urgentlyRememberTo(task) {

todoList.unshift(task);

}

The previous program manages lists of tasks. You add tasks to the end of the list by calling rememberTo("eat"), and when you’re ready to do something, you call whatIsNext() to get (and remove) the front item from the list. The urgentlyRememberTo function also adds a task but adds it to the front instead of the back of the list.

The indexOf method has a sibling called lastIndexOf, which starts searching for the given element at the end of the array instead of the front.

console.log([1, 2, 3, 2, 1].indexOf(2));

// → 1

console.log([1, 2, 3, 2, 1].lastIndexOf(2));

// → 3

Both indexOf and lastIndexOf take an optional second argument that indicates where to start searching from.

Another fundamental method is slice, which takes a start index and an end index and returns an array that has only the elements between those indices. The start index is inclusive, the end index exclusive.

console.log([0, 1, 2, 3, 4].slice(2, 4));

// → [2, 3]

console.log([0, 1, 2, 3, 4].slice(2));

// → [2, 3, 4]

When the end index is not given, slice will take all of the elements after the start index. Strings also have a slice method, which has a similar effect.

The concat method can be used to glue arrays together, similar to what the + operator does for strings. The following example shows both concat and slice in action. It takes an array and an index, and it returns a new array that is a copy of the original array with the element at the given index removed.

function remove(array, index) {

return array.slice(0, index)

.concat(array.slice(index + 1));

}

console.log(remove(["a", "b", "c", "d", "e"], 2));

// → ["a", "b", "d", "e"]

## Strings and their properties

We can read properties like length and toUpperCase from string values. But if you try to add a new property, it doesn’t stick.

var myString = "Fido";

myString.myProperty = "value";

console.log(myString.myProperty);

// → undefined

Values of type string, number, and Boolean are not objects, and though the language doesn’t complain if you try to set new properties on them, it doesn’t actually store those properties. The values are immutable and cannot be changed.

But these types do have some built-in properties. Every string value has a number of methods. The most useful ones are probably slice and indexOf, which resemble the array methods of the same name.

console.log("coconuts".slice(4, 7));

// → nut

console.log("coconut".indexOf("u"));

// → 5

One difference is that a string’s indexOf can take a string containing more than one character, whereas the corresponding array method looks only for a single element.

console.log("one two three".indexOf("ee"));

// → 11

The trim method removes whitespace (spaces, newlines, tabs, and similar characters) from the start and end of a string.

console.log(" okay \n ".trim());

// → okay

We have already seen the string type’s length property. Accessing the individual characters in a string can be done with the charAt method but also by simply reading numeric properties, like you’d do for an array.

var string = "abc";

console.log(string.length);

// → 3

console.log(string.charAt(0));

// → a

console.log(string[1]);

// → b

## The arguments object

Whenever a function is called, a special variable named arguments is added to the environment in which the function body runs. This variable refers to an object that holds all of the arguments passed to the function. Remember that in JavaScript you are allowed to pass more (or fewer) arguments to a function than the number of parameters the function itself declares.

function noArguments() {}

noArguments(1, 2, 3); // This is okay

function threeArguments(a, b, c) {}

threeArguments(); // And so is this

The arguments object has a length property that tells us the number of arguments that were really passed to the function. It also has a property for each argument, named 0, 1, 2, and so on.

If that sounds a lot like an array to you, you’re right, it is a lot like an array. But this object, unfortunately, does not have any array methods (like slice or indexOf), so it is a little harder to use than a real array.

function argumentCounter() {

console.log("You gave me", arguments.length, "arguments.");

}

argumentCounter("Straw man", "Tautology", "Ad hominem");

// → You gave me 3 arguments.

Some functions can take any number of arguments, like console.log. These typically loop over the values in their arguments object. They can be used to create very pleasant interfaces. For example, remember how we created the entries to Jacques’ journal.

addEntry(["work", "touched tree", "pizza", "running",

"television"], false);

Since he is going to be calling this function a lot, we could create an alternative that is easier to call.

function addEntry(squirrel) {

var entry = {events: [], squirrel: squirrel};

for (var i = 1; i < arguments.length; i++)

entry.events.push(arguments[i]);

journal.push(entry);

}

addEntry(true, "work", "touched tree", "pizza",

"running", "television");

This version reads its first argument (squirrel) in the normal way and then goes over the rest of the arguments (the loop starts at index 1, skipping the first) to gather them into an array.

## The Math object

As we’ve seen, Math is a grab-bag of number-related utility functions, such as Math.max (maximum), Math.min (minimum), and Math.sqrt (square root).

The Math object is used simply as a container to group a bunch of related functionality. There is only one Math object, and it is almost never useful as a value. Rather, it provides a namespace so that all these functions and values do not have to be global variables.

Having too many global variables “pollutes” the namespace. The more names that have been taken, the more likely you are to accidentally overwrite the value of some variable. For example, it’s not unlikely that you’ll want to name something max in one of your programs. Since JavaScript’s built-in max function is tucked safely inside the Math object, we don’t have to worry about overwriting it.

Many languages will stop you, or at least warn you, when you are defining a variable with a name that is already taken. JavaScript does neither, so be careful.

Back to the Math object. If you need to do trigonometry, Math can help. It contains cos (cosine), sin (sine), and tan (tangent), as well as their inverse functions, acos, asin, and atan, respectively. The number π (pi)—or at least the closest approximation that fits in a JavaScript number—is available as Math.PI. (There is an old programming tradition of writing the names of constant values in all caps.)

function randomPointOnCircle(radius) {

var angle = Math.random() \* 2 \* Math.PI;

return {x: radius \* Math.cos(angle),

y: radius \* Math.sin(angle)};

}

console.log(randomPointOnCircle(2));

// → {x: 0.3667, y: 1.966}

If sines and cosines are not something you are very familiar with, don’t worry. When they are used in this book, in [Chapter 13](http://eloquentjavascript.net/13_dom.html#sin_cos), I’ll explain them.

The previous example uses Math.random. This is a function that returns a new pseudorandom number between zero (inclusive) and one (exclusive) every time you call it.

console.log(Math.random());

// → 0.36993729369714856

console.log(Math.random());

// → 0.727367032552138

console.log(Math.random());

// → 0.40180766698904335

Though computers are deterministic machines—they always react the same way if given the same input—it is possible to have them produce numbers that appear random. To do this, the machine keeps a number (or a bunch of numbers) in its internal state. Then, every time a random number is requested, it performs some complicated deterministic computations on this internal state and returns part of the result of those computations. The machine also uses the outcome to change its own internal state so that the next “random” number produced will be different.

If we want a whole random number instead of a fractional one, we can use Math.floor (which rounds down to the nearest whole number) on the result of Math.random.

console.log(Math.floor(Math.random() \* 10));

// → 2

Multiplying the random number by 10 gives us a number greater than or equal to zero, and below 10. Since Math.floor rounds down, this expression will produce, with equal chance, any number from 0 through 9.

There are also the functions Math.ceil (for “ceiling”, which rounds up to a whole number) and Math.round (to the nearest whole number).

## The global object

The global scope, the space in which global variables live, can also be approached as an object in JavaScript. Each global variable is present as a property of this object. In browsers, the global scope object is stored in the window variable.

var myVar = 10;

console.log("myVar" in window);

// → true

console.log(window.myVar);

// → 10

## Summary

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 trying to wrap 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.propName or value["propName"]. 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 numbers 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.

Objects can also serve as maps, associating values with names. The in operator can be used to find out whether an object contains a property with a given name. The same keyword can also be used in a for loop (for (var name in object)) to loop over an object’s properties.

# Chapter 5

# Higher-Order Functions

Tzu-li and Tzu-ssu were boasting about the size of their latest programs. ‘Two-hundred thousand lines,’ said Tzu-li, ‘not counting comments!’ Tzu-ssu responded, ‘Pssh, mine is almost a **million** lines already.’ Master Yuan-Ma said, ‘My best program has five hundred lines.’ Hearing this, Tzu-li and Tzu-ssu were enlightened.

Master Yuan-Ma, The Book of Programming

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.

C.A.R. Hoare, 1980 ACM Turing Award Lecture

A large program is a costly program, and not just because of the time it takes to build. Size almost always involves complexity, and complexity confuses programmers. Confused programmers, in turn, tend to introduce mistakes (bugs) into programs. A large program also provides a lot of space for these bugs to hide, making them hard to find.

Let’s briefly go back to the final two example programs in the introduction. The first is self-contained and six lines long.

edit & run code by clicking it

var total = 0, count = 1;

while (count <= 10) {

total += count;

count += 1;

}

console.log(total);

The second relies on two external functions and is one line long.

console.log(sum(range(1, 10)));

Which one is more likely to contain a bug?

If we count the size of the definitions of sum and range, the second program is also big—even bigger than the first. But still, I’d argue that it is more likely to be correct.

It is more likely to be correct because the solution is expressed in a vocabulary that corresponds to the problem being solved. Summing a range of numbers isn’t about loops and counters. It is about ranges and sums.

The definitions of this vocabulary (the functions sum and range) will still involve loops, counters, and other incidental details. But because they are expressing simpler concepts than the program as a whole, they are easier to get right.

## Abstraction

In the context of programming, these kinds of vocabularies are usually called abstractions. Abstractions hide details and give us the ability to talk about problems at a higher (or more abstract) level.

As an analogy, compare these two recipes for pea soup:

Put 1 cup of dried peas per person into a container. Add water until the peas are well covered. Leave the peas in water for at least 12 hours. Take the peas out of the water and put them in a cooking pan. Add 4 cups of water per person. Cover the pan and keep the peas simmering for two hours. Take half an onion per person. Cut it into pieces with a knife. Add it to the peas. Take a stalk of celery per person. Cut it into pieces with a knife. Add it to the peas. Take a carrot per person. Cut it into pieces. With a knife! Add it to the peas. Cook for 10 more minutes.

And the second recipe:

Per person: 1 cup dried split peas, half a chopped onion, a stalk of celery, and a carrot.

Soak peas for 12 hours. Simmer for 2 hours in 4 cups of water (per person). Chop and add vegetables. Cook for 10 more minutes.

The second is shorter and easier to interpret. But you do need to understand a few more cooking-related words—soak, simmer, chop, and, I guess, vegetable.

When programming, we can’t rely on all the words we need to be waiting for us in the dictionary. Thus, you might fall into the pattern of the first recipe—work out the precise steps the computer has to perform, one by one, blind to the higher-level concepts that they express.

It has to become second nature, for a programmer, to notice when a concept is begging to be abstracted into a new word.

## Abstracting array traversal

Plain functions, as we’ve seen them so far, are a good way to build abstractions. But sometimes they fall short.

In the [previous chapter](http://eloquentjavascript.net/04_data.html#data), this type of for loop made several appearances:

var array = [1, 2, 3];

for (var i = 0; i < array.length; i++) {

var current = array[i];

console.log(current);

}

It’s trying to say, “For each element in the array, log it to the console”. But it uses a roundabout way that involves a counter variable i, a check against the array’s length, and an extra variable declaration to pick out the current element. Apart from being a bit of an eyesore, this provides a lot of space for potential mistakes. We might accidentally reuse the i variable, misspell length as lenght, confuse the i and current variables, and so on.

So let’s try to abstract this into a function. Can you think of a way?

Well, it’s easy to write a function that goes over an array and calls console.log on every element.

function logEach(array) {

for (var i = 0; i < array.length; i++)

console.log(array[i]);

}

But what if we want to do something other than logging the elements? Since “doing something” can be represented as a function and functions are just values, we can pass our action as a function value.

function forEach(array, action) {

for (var i = 0; i < array.length; i++)

action(array[i]);

}

forEach(["Wampeter", "Foma", "Granfalloon"], console.log);

// → Wampeter

// → Foma

// → Granfalloon

(In some browsers, calling console.log in this way does not work. You can use alert instead of console.log if this example fails to work.)

Often, you don’t pass a predefined function to forEach but create a function value on the spot instead.

var numbers = [1, 2, 3, 4, 5], sum = 0;

forEach(numbers, function(number) {

sum += number;

});

console.log(sum);

// → 15

This looks quite a lot like the classical for loop, with its body written as a block below it. However, now the body is inside the function value, as well as inside the parentheses of the call to forEach. This is why it has to be closed with the closing brace and closing parenthesis.

Using this pattern, we can specify a variable name for the current element (number), rather than having to pick it out of the array manually.

In fact, we don’t need to write forEach ourselves. It is available as a standard method on arrays. Since the array is already provided as the thing the method acts on, forEach takes only one required argument: the function to be executed for each element.

To illustrate how helpful this is, let’s look back at a function from [the previous chapter](http://eloquentjavascript.net/04_data.html#analysis). It contains two array-traversing loops.

function gatherCorrelations(journal) {

var phis = {};

for (var entry = 0; entry < journal.length; entry++) {

var events = journal[entry].events;

for (var i = 0; i < events.length; i++) {

var event = events[i];

if (!(event in phis))

phis[event] = phi(tableFor(event, journal));

}

}

return phis;

}

Working with forEach makes it slightly shorter and quite a bit cleaner.

function gatherCorrelations(journal) {

var phis = {};

journal.forEach(function(entry) {

entry.events.forEach(function(event) {

if (!(event in phis))

phis[event] = phi(tableFor(event, journal));

});

});

return phis;

}

## Higher-order functions

Functions that operate on other functions, either by taking them as arguments or by returning them, are called higher-order functions. If you have already accepted the fact that functions are regular values, there is nothing particularly remarkable about the fact that such functions exist. The term comes from mathematics, where the distinction between functions and other values is taken more seriously.

Higher-order functions allow us to abstract over actions, not just values. They come in several forms. For example, you can have functions that create new functions.

function greaterThan(n) {

return function(m) { return m > n; };

}

var greaterThan10 = greaterThan(10);

console.log(greaterThan10(11));

// → true

And you can have functions that change other functions.

function noisy(f) {

return function(arg) {

console.log("calling with", arg);

var val = f(arg);

console.log("called with", arg, "- got", val);

return val;

};

}

noisy(Boolean)(0);

// → calling with 0

// → called with 0 - got false

You can even write functions that provide new types of control flow.

function unless(test, then) {

if (!test) then();

}

function repeat(times, body) {

for (var i = 0; i < times; i++) body(i);

}

repeat(3, function(n) {

unless(n % 2, function() {

console.log(n, "is even");

});

});

// → 0 is even

// → 2 is even

The lexical scoping rules that we discussed in [Chapter 3](http://eloquentjavascript.net/03_functions.html#scoping) work to our advantage when using functions in this way. In the previous example, the n variable is a parameter to the outer function. Because the inner function lives inside the environment of the outer one, it can use n. The bodies of such inner functions can access the variables around them. They can play a role similar to the {} blocks used in regular loops and conditional statements. An important difference is that variables declared inside inner functions do not end up in the environment of the outer function. And that is usually a good thing.

## Passing along arguments

The noisy function defined earlier, which wraps its argument in another function, has a rather serious deficit.

function noisy(f) {

return function(arg) {

console.log("calling with", arg);

var val = f(arg);

console.log("called with", arg, "- got", val);

return val;

};

}

If f takes more than one parameter, it gets only the first one. We could add a bunch of arguments to the inner function (arg1, arg2, and so on) and pass them all to f, but it is not clear how many would be enough. This solution would also deprive f of the information in arguments.length. Since we’d always pass the same number of arguments, it wouldn’t know how many arguments were originally given.

For these kinds of situations, JavaScript functions have an apply method. You pass it an array (or array-like object) of arguments, and it will call the function with those arguments.

function transparentWrapping(f) {

return function() {

return f.apply(null, arguments);

};

}

That’s a useless function, but it shows the pattern we are interested in—the function it returns passes all of the given arguments, and only those arguments, to f. It does this by passing its own arguments object to apply. The first argument to apply, for which we are passing null here, can be used to simulate a method call. We will come back to that in the [next chapter](http://eloquentjavascript.net/06_object.html#call_method).

## JSON

Higher-order functions that somehow apply a function to the elements of an array are widely used in JavaScript. The forEach method is the most primitive such function. There are a number of other variants available as methods on arrays. To familiarize ourselves with them, let’s play around with another data set.

A few years ago, someone crawled through a lot of archives and put together a book on the history of my family name (Haverbeke—meaning Oatbrook). I opened it hoping to find knights, pirates, and alchemists ... but the book turns out to be mostly full of Flemish farmers. For my amusement, I extracted the information on my direct ancestors and put it into a computer-readable format.

The file I created looks something like this:

[

{"name": "Emma de Milliano", "sex": "f",

"born": 1876, "died": 1956,

"father": "Petrus de Milliano",

"mother": "Sophia van Damme"},

{"name": "Carolus Haverbeke", "sex": "m",

"born": 1832, "died": 1905,

"father": "Carel Haverbeke",

"mother": "Maria van Brussel"},

… and so on

]

This format is called JSON (pronounced “Jason”), which stands for JavaScript Object Notation. It is widely used as a data storage and communication format on the Web.

JSON is similar to JavaScript’s way of writing arrays and objects, with a few restrictions. All property names have to be surrounded by double quotes, and only simple data expressions are allowed—no function calls, variables, or anything that involves actual computation. Comments are not allowed in JSON.

JavaScript gives us functions, JSON.stringify and JSON.parse, that convert data to and from this format. The first takes a JavaScript value and returns a JSON-encoded string. The second takes such a string and converts it to the value it encodes.

var string = JSON.stringify({name: "X", born: 1980});

console.log(string);

// → {"name":"X","born":1980}

console.log(JSON.parse(string).born);

// → 1980

The variable ANCESTRY\_FILE, available in the sandbox for this chapter and in [a downloadable file](http://eloquentjavascript.net/code/ancestry.js) on the website, contains the content of my JSON file as a string. Let’s decode it and see how many people it contains.

var ancestry = JSON.parse(ANCESTRY\_FILE);

console.log(ancestry.length);

// → 39

## Filtering an array

To find the people in the ancestry data set who were young in 1924, the following function might be helpful. It filters out the elements in an array that don’t pass a test.

function filter(array, test) {

var passed = [];

for (var i = 0; i < array.length; i++) {

if (test(array[i]))

passed.push(array[i]);

}

return passed;

}

console.log(filter(ancestry, function(person) {

return person.born > 1900 && person.born < 1925;

}));

// → [{name: "Philibert Haverbeke", …}, …]

This uses the argument named test, a function value, to fill in a “gap” in the computation. The test function is called for each element, and its return value determines whether an element is included in the returned array.

Three people in the file were alive and young in 1924: my grandfather, grandmother, and great-aunt.

Note how the filter function, rather than deleting elements from the existing array, builds up a new array with only the elements that pass the test. This function is pure. It does not modify the array it is given.

Like forEach, filter is also a standard method on arrays. The example defined the function only in order to show what it does internally. From now on, we’ll use it like this instead:

console.log(ancestry.filter(function(person) {

return person.father == "Carel Haverbeke";

}));

// → [{name: "Carolus Haverbeke", …}]

## Transforming with map

Say we have an array of objects representing people, produced by filtering the ancestry array somehow. But we want an array of names, which is easier to read.

The map method transforms an array by applying a function to all of its elements and building a new array from the returned values. The new array will have the same length as the input array, but its content will have been “mapped” to a new form by the function.

function map(array, transform) {

var mapped = [];

for (var i = 0; i < array.length; i++)

mapped.push(transform(array[i]));

return mapped;

}

var overNinety = ancestry.filter(function(person) {

return person.died - person.born > 90;

});

console.log(map(overNinety, function(person) {

return person.name;

}));

// → ["Clara Aernoudts", "Emile Haverbeke",

// "Maria Haverbeke"]

Interestingly, the people who lived to at least 90 years of age are the same three people who we saw before—the people who were young in the 1920s, which happens to be the most recent generation in my data set. I guess medicine has come a long way.

Like forEach and filter, map is also a standard method on arrays.

## Summarizing with reduce

Another common pattern of computation on arrays is computing a single value from them. Our recurring example, summing a collection of numbers, is an instance of this. Another example would be finding the person with the earliest year of birth in the data set.

The higher-order operation that represents this pattern is called reduce (or sometimes fold). You can think of it as folding up the array, one element at a time. When summing numbers, you’d start with the number zero and, for each element, combine it with the current sum by adding the two.

The parameters to the reduce function are, apart from the array, a combining function and a start value. This function is a little less straightforward than filter and map, so pay close attention.

function reduce(array, combine, start) {

var current = start;

for (var i = 0; i < array.length; i++)

current = combine(current, array[i]);

return current;

}

console.log(reduce([1, 2, 3, 4], function(a, b) {

return a + b;

}, 0));

// → 10

The standard array method reduce, which of course corresponds to this function, has an added convenience. If your array contains at least one element, you are allowed to leave off the start argument. The method will take the first element of the array as its start value and start reducing at the second element.

To use reduce to find my most ancient known ancestor, we can write something like this:

console.log(ancestry.reduce(function(min, cur) {

if (cur.born < min.born) return cur;

else return min;

}));

// → {name: "Pauwels van Haverbeke", born: 1535, …}

## Composability

Consider how we would have written the previous example (finding the person with the earliest year of birth) without higher-order functions. The code is not that much worse.

var min = ancestry[0];

for (var i = 1; i < ancestry.length; i++) {

var cur = ancestry[i];

if (cur.born < min.born)

min = cur;

}

console.log(min);

// → {name: "Pauwels van Haverbeke", born: 1535, …}

There are a few more variables, and the program is two lines longer but still quite easy to understand.

Higher-order functions start to shine when you need to compose functions. As an example, let’s write code that finds the average age for men and for women in the data set.

function average(array) {

function plus(a, b) { return a + b; }

return array.reduce(plus) / array.length;

}

function age(p) { return p.died - p.born; }

function male(p) { return p.sex == "m"; }

function female(p) { return p.sex == "f"; }

console.log(average(ancestry.filter(male).map(age)));

// → 61.67

console.log(average(ancestry.filter(female).map(age)));

// → 54.56

(It’s a bit silly that we have to define plus as a function, but operators in JavaScript, unlike functions, are not values, so you can’t pass them as arguments.)

Instead of tangling the logic into a big loop, it is neatly composed into the concepts we are interested in—determining sex, computing age, and averaging numbers. We can apply these one by one to get the result we are looking for.

This is fabulous for writing clear code. Unfortunately, this clarity comes at a cost.

## The cost

In the happy land of elegant code and pretty rainbows, there lives a spoil-sport monster called inefficiency.

A program that processes an array is most elegantly expressed as a sequence of cleanly separated steps that each do something with the array and produce a new array. But building up all those intermediate arrays is somewhat expensive.

Likewise, passing a function to forEach and letting that method handle the array iteration for us is convenient and easy to read. But function calls in JavaScript are costly compared to simple loop bodies.

And so it goes with a lot of techniques that help improve the clarity of a program. Abstractions add layers between the raw things the computer is doing and the concepts we are working with and thus cause the machine to perform more work. This is not an iron law—there are programming languages that have better support for building abstractions without adding inefficiencies, and even in JavaScript, an experienced programmer can find ways to write abstract code that is still fast. But it is a problem that comes up a lot.

Fortunately, most computers are insanely fast. If you are processing a modest set of data or doing something that has to happen only on a human time scale (say, every time the user clicks a button), then it does not matter whether you wrote a pretty solution that takes half a millisecond or a super-optimized solution that takes a tenth of a millisecond.

It is helpful to roughly keep track of how often a piece of your program is going to run. If you have a loop inside a loop (either directly or through the outer loop calling a function that ends up performing the inner loop), the code inside the inner loop will end up running N×M times, where N is the number of times the outer loop repeats and M is the number of times the inner loop repeats within each iteration of the outer loop. If that inner loop contains another loop that makes P rounds, its body will run M×N×P times, and so on. This can add up to large numbers, and when a program is slow, the problem can often be traced to only a small part of the code, which sits inside an inner loop.

## Great-great-great-great-...

My grandfather, Philibert Haverbeke, is included in the data file. By starting with him, I can trace my lineage to find out whether the most ancient person in the data, Pauwels van Haverbeke, is my direct ancestor. And if he is, I would like to know how much DNA I theoretically share with him.

To be able to go from a parent’s name to the actual object that represents this person, we first build up an object that associates names with people.

var byName = {};

ancestry.forEach(function(person) {

byName[person.name] = person;

});

console.log(byName["Philibert Haverbeke"]);

// → {name: "Philibert Haverbeke", …}

Now, the problem is not entirely as simple as following the father properties and counting how many we need to reach Pauwels. There are several cases in the family tree where people married their second cousins (tiny villages and all that). This causes the branches of the family tree to rejoin in a few places, which means I share more than 1/2G of my genes with this person, where G for the number of generations between Pauwels and me. This formula comes from the idea that each generation splits the gene pool in two.

A reasonable way to think about this problem is to look at it as being analogous to reduce, which condenses an array to a single value by repeatedly combining values, left to right. In this case, we also want to condense our data structure to a single value but in a way that follows family lines. The shape of the data is that of a family tree, rather than a flat list.

The way we want to reduce this shape is by computing a value for a given person by combining values from their ancestors. This can be done recursively: if we are interested in person A, we have to compute the values for A’s parents, which in turn requires us to compute the value for A’s grandparents, and so on. In principle, that’d require us to look at an infinite number of people, but since our data set is finite, we have to stop somewhere. We’ll allow a default value to be given to our reduction function, which will be used for people who are not in the data. In our case, that value is simply zero, on the assumption that people not in the list don’t share DNA with the ancestor we are looking at.

Given a person, a function to combine values from the two parents of a given person, and a default value, reduceAncestors condenses a value from a family tree.

function reduceAncestors(person, f, defaultValue) {

function valueFor(person) {

if (person == null)

return defaultValue;

else

return f(person, valueFor(byName[person.mother]),

valueFor(byName[person.father]));

}

return valueFor(person);

}

The inner function (valueFor) handles a single person. Through the magic of recursion, it can simply call itself to handle the father and the mother of this person. The results, along with the person object itself, are passed to f, which returns the actual value for this person.

We can then use this to compute the amount of DNA my grandfather shared with Pauwels van Haverbeke and divide that by four.

function sharedDNA(person, fromMother, fromFather) {

if (person.name == "Pauwels van Haverbeke")

return 1;

else

return (fromMother + fromFather) / 2;

}

var ph = byName["Philibert Haverbeke"];

console.log(reduceAncestors(ph, sharedDNA, 0) / 4);

// → 0.00049

The person with the name Pauwels van Haverbeke obviously shared 100 percent of his DNA with Pauwels van Haverbeke (there are no people who share names in the data set), so the function returns 1 for him. All other people share the average of the amounts that their parents share.

So, statistically speaking, I share about 0.05 percent of my DNA with this 16th-century person. It should be noted that this is only a statistical approximation, not an exact amount. It is a rather small number, but given how much genetic material we carry (about 3 billion base pairs), there’s still probably some aspect in the biological machine that is me that originates with Pauwels.

We could also have computed this number without relying on reduceAncestors. But separating the general approach (condensing a family tree) from the specific case (computing shared DNA) can improve the clarity of the code and allows us to reuse the abstract part of the program for other cases. For example, the following code finds the percentage of a person’s known ancestors who lived past 70 (by lineage, so people may be counted multiple times):

function countAncestors(person, test) {

function combine(current, fromMother, fromFather) {

var thisOneCounts = current != person && test(current);

return fromMother + fromFather + (thisOneCounts ? 1 : 0);

}

return reduceAncestors(person, combine, 0);

}

function longLivingPercentage(person) {

var all = countAncestors(person, function(person) {

return true;

});

var longLiving = countAncestors(person, function(person) {

return (person.died - person.born) >= 70;

});

return longLiving / all;

}

console.log(longLivingPercentage(byName["Emile Haverbeke"]));

// → 0.129

Such numbers are not to be taken too seriously, given that our data set contains a rather arbitrary collection of people. But the code illustrates the fact that reduceAncestors gives us a useful piece of vocabulary for working with the family tree data structure.

## Binding

The bind method, which all functions have, creates a new function that will call the original function but with some of the arguments already fixed.

The following code shows an example of bind in use. It defines a function isInSet that tells us whether a person is in a given set of strings. To call filter in order to collect those person objects whose names are in a specific set, we can either write a function expression that makes a call to isInSet with our set as its first argument or partially apply the isInSet function.

var theSet = ["Carel Haverbeke", "Maria van Brussel",

"Donald Duck"];

function isInSet(set, person) {

return set.indexOf(person.name) > -1;

}

console.log(ancestry.filter(function(person) {

return isInSet(theSet, person);

}));

// → [{name: "Maria van Brussel", …},

// {name: "Carel Haverbeke", …}]

console.log(ancestry.filter(isInSet.bind(null, theSet)));

// → … same result

The call to bind returns a function that will call isInSet with theSet as first argument, followed by any remaining arguments given to the bound function.

The first argument, where the example passes null, is used for method calls, similar to the first argument to apply. I’ll describe this in more detail in the [next chapter](http://eloquentjavascript.net/06_object.html#call_method).

## Summary

Being able to pass function values to other functions is not just a gimmick but a deeply useful aspect of JavaScript. It allows us to write computations with “gaps” in them as functions and have the code that calls these functions fill in those gaps by providing function values that describe the missing computations.

Arrays provide a number of useful higher-order methods—forEach to do something with each element in an array, filter to build a new array with some elements filtered out, map to build a new array where each element has been put through a function, and reduce to combine all an array’s elements into a single value.

Functions have an apply method that can be used to call them with an array specifying their arguments. They also have a bind method, which is used to create a partially applied version of the function.

# Chapter 12

# JavaScript and the Browser

The browser is a really hostile programming environment.

Douglas Crockford, The JavaScript Programming Language (video lecture)

The next part of this book will talk about web browsers. Without web browsers, there would be no JavaScript. And even if there were, no one would ever have paid any attention to it.

Web technology has, from the start, been decentralized, not just technically but also in the way it has evolved. Various browser vendors have added new functionality in ad hoc and sometimes poorly thought out ways, which then sometimes ended up being adopted by others and finally set down as a standard.

This is both a blessing and a curse. On the one hand, it is empowering to not have a central party control a system but have it be improved by various parties working in loose collaboration (or, occasionally, open hostility). On the other hand, the haphazard way in which the Web was developed means that the resulting system is not exactly a shining example of internal consistency. In fact, some parts of it are downright messy and confusing.

## Networks and the Internet

Computer networks have been around since the 1950s. If you put cables between two or more computers and allow them to send data back and forth through these cables, you can do all kinds of wonderful things.

If connecting two machines in the same building allows us to do wonderful things, connecting machines all over the planet should be even better. The technology to start implementing this vision was developed in the 1980s, and the resulting network is called the Internet. It has lived up to its promise.

A computer can use this network to spew bits at another computer. For any effective communication to arise out of this bit-spewing, the computers at both ends must know what the bits are supposed to represent. The meaning of any given sequence of bits depends entirely on the kind of thing that it is trying to express and on the encoding mechanism used.

A network protocol describes a style of communication over a network. There are protocols for sending email, for fetching email, for sharing files, or even for controlling computers that happen to be infected by malicious software.

For example, a simple chat protocol might consist of one computer sending the bits that represent the text “CHAT?” to another machine and the other responding with “OK!” to confirm that it understands the protocol. They can then proceed to send each other strings of text, read the text sent by the other from the network, and display whatever they receive on their screens.

Most protocols are built on top of other protocols. Our example chat protocol treats the network as a streamlike device into which you can put bits and have them arrive at the correct destination in the correct order. Ensuring those things is already a rather difficult technical problem.

The Transmission Control Protocol (TCP) is a protocol that solves this problem. All Internet-connected devices “speak” it, and most communication on the Internet is built on top of it.

A TCP connection works as follows: one computer must be waiting, or listening, for other computers to start talking to it. To be able to listen for different kinds of communication at the same time on a single machine, each listener has a number (called a port) associated with it. Most protocols specify which port should be used by default. For example, when we want to send an email using the SMTP protocol, the machine through which we send it is expected to be listening on port 25.

Another computer can then establish a connection by connecting to the target machine using the correct port number. If the target machine can be reached and is listening on that port, the connection is successfully created. The listening computer is called the server, and the connecting computer is called the client.

Such a connection acts as a two-way pipe through which bits can flow—the machines on both ends can put data into it. Once the bits are successfully transmitted, they can be read out again by the machine on the other side. This is a convenient model. You could say that TCP provides an abstraction of the network.

## The Web

The World Wide Web (not to be confused with the Internet as a whole) is a set of protocols and formats that allow us to visit web pages in a browser. The “Web” part in the name refers to the fact that such pages can easily link to each other, thus connecting into a huge mesh that users can move through.

To add content to the Web, all you need to do is connect a machine to the Internet, and have it listen on port 80, using the Hypertext Transfer Protocol (HTTP). This protocol allows other computers to request documents over the network.

Each document on the Web is named by a Uniform Resource Locator (URL), which looks something like this:

http://eloquentjavascript.net/12\_browser.html

| | | |

protocol server path

The first part tells us that this URL uses the HTTP protocol (as opposed to, for example, encrypted HTTP, which would be https://). Then comes the part that identifies which server we are requesting the document from. Last is a path string that identifies the specific document (or resource) we are interested in.

Each machine connected to the Internet gets a unique IP address, which looks something like 37.187.37.82. You can use these directly as the server part of a URL. But lists of more or less random numbers are hard to remember and awkward to type, so you can instead register a domain name to point toward a specific machine or set of machines. I registered eloquentjavascript.net to point at the IP address of a machine I control and can thus use that domain name to serve web pages.

If you type the previous URL into your browser’s address bar, it will try to retrieve and display the document at that URL. First, your browser has to find out what address eloquentjavascript.net refers to. Then, using the HTTP protocol, it makes a connection to the server at that address and asks for the resource /12\_browser.html. We will take a closer look at the HTTP protocol in [Chapter 17](http://eloquentjavascript.net/17_http.html#http).

## HTML

HTML, which stands for Hypertext Markup Language, is the document format used for web pages. An HTML document contains text, as well as tags that give structure to the text, describing things such as links, paragraphs, and headings.

A simple HTML document looks like this:

edit & run code by clicking it

<!doctype html>

<html>

<head>

<title>My home page</title>

</head>

<body>

<h1>My home page</h1>

<p>Hello, I am Marijn and this is my home page.</p>

<p>I also wrote a book! Read it

<a href="http://eloquentjavascript.net">here</a>.</p>

</body>

</html>

The tags, wrapped in angle brackets (< and >), provide information about the structure of the document. The other text is just plain text.

The document starts with <!doctype html>, which tells the browser to interpret it as modern HTML, as opposed to various dialects that were in use in the past.

HTML documents have a head and a body. The head contains information about the document, and the body contains the document itself. In this case, we first declared that the title of this document is “My home page” and then gave a document containing a heading (<h1>, meaning “heading 1”—<h2> to <h6> produce more minor headings) and two paragraphs (<p>).

Tags come in several forms. An element, such as the body, a paragraph, or a link, is started by an opening tag like <p> and ended by a closing tag like </p>. Some opening tags, such as the one for the link (<a>), contain extra information in the form of name="value" pairs. These are called attributes. In this case, the destination of the link is indicated with href="http://eloquentjavascript.net", where href stands for “hypertext reference”.

Some kinds of tags do not enclose anything and thus do not need to be closed. An example of this would be <img src="http://example.com/image.jpg">, which will display the image found at the given source URL.

To be able to include angle brackets in the text of a document, even though they have a special meaning in HTML, yet another form of special notation has to be introduced. A plain opening angle bracket is written as &lt; (“less than”), and a closing bracket is written as &gt; (“greater than”). In HTML, an ampersand (&) character followed by a word and a semicolon (;) is called an entity, and will be replaced by the character it encodes.

This is analogous to the way backslashes are used in JavaScript strings. Since this mechanism gives ampersand characters a special meaning, too, those need to be escaped as &amp;. Inside an attribute, which is wrapped in double quotes, &quot; can be used to insert an actual quote character.

HTML is parsed in a remarkably error-tolerant way. When tags that should be there are missing, the browser reconstructs them. The way in which this is done has been standardized, and you can rely on all modern browsers to do it in the same way.

The following document will be treated just like the one shown previously:

<!doctype html>

<title>My home page</title>

<h1>My home page</h1>

<p>Hello, I am Marijn and this is my home page.

<p>I also wrote a book! Read it

<a href=http://eloquentjavascript.net>here</a>.

The <html>, <head>, and <body> tags are gone completely. The browser knows that <title> belongs in a head, and that <h1> in a body. Furthermore, I am no longer explicitly closing the paragraphs since opening a new paragraph or ending the document will close them implicitly. The quotes around the link target are also gone.

This book will usually omit the <html>, <head>, and <body> tags from examples to keep them short and free of clutter. But I will close tags and include quotes around attributes.

I will also usually omit the doctype. This is not to be taken as an encouragement to omit doctype declarations. Browsers will often do ridiculous things when you forget them. You should consider doctypes implicitly present in examples, even when they are not actually shown in the text.

## HTML and JavaScript

In the context of this book, the most important HTML tag is <script>. This tag allows us to include a piece of JavaScript in a document.

<h1>Testing alert</h1>

<script>alert("hello!");</script>

Such a script will run as soon as its <script> tag is encountered as the browser reads the HTML. The page shown earlier will pop up an alert dialog when opened.

Including large programs directly in HTML documents is often impractical. The <script> tag can be given an src attribute in order to fetch a script file (a text file containing a JavaScript program) from a URL.

<h1>Testing alert</h1>

<script src="code/hello.js"></script>

The code/hello.js file included here contains the same simple program, alert("hello!"). When an HTML page references other URLs as part of itself, for example an image file or a script—web browsers will retrieve them immediately and include them in the page.

A script tag must always be closed with </script>, even if it refers to a script file and doesn’t contain any code. If you forget this, the rest of the page will be interpreted as part of the script.

Some attributes can also contain a JavaScript program. The <button> tag shown next (which shows up as a button) has an onclick attribute, whose content will be run whenever the button is clicked.

<button onclick="alert('Boom!');">DO NOT PRESS</button>

Note that I had to use single quotes for the string in the onclick attribute because double quotes are already used to quote the whole attribute. I could also have used &quot;, but that’d make the program harder to read.

## In the sandbox

Running programs downloaded from the Internet is potentially dangerous. You do not know much about the people behind most sites you visit, and they do not necessarily mean well. Running programs by people who do not mean well is how you get your computer infected by viruses, your data stolen, and your accounts hacked.

Yet the attraction of the Web is that you can surf it without necessarily trusting all the pages you visit. This is why browsers severely limit the things a JavaScript program may do: it can’t look at the files on your computer or modify anything not related to the web page it was embedded in.

Isolating a programming environment in this way is called sandboxing, the idea being that the program is harmlessly playing in a sandbox. But you should imagine this particular kind of sandbox as having a cage of thick steel bars over it, which makes it somewhat different from your typical playground sandbox.

The hard part of sandboxing is allowing the programs enough room to be useful yet at the same time restricting them from doing anything dangerous. Lots of useful functionality, such as communicating with other servers or reading the content of the copy-paste clipboard, can also be used to do problematic, privacy-invading things.

Every now and then, someone comes up with a new way to circumvent the limitations of a browser and do something harmful, ranging from leaking minor private information to taking over the whole machine that the browser runs on. The browser developers respond by fixing the hole, and all is well again—that is, until the next problem is discovered, and hopefully publicized, rather than secretly exploited by some government or mafia.

## Compatibility and the browser wars

In the early stages of the Web, a browser called Mosaic dominated the market. After a few years, the balance had shifted to Netscape, which was then, in turn, largely supplanted by Microsoft’s Internet Explorer. At any point where a single browser was dominant, that browser’s vendor would feel entitled to unilaterally invent new features for the Web. Since most users used the same browser, websites would simply start using those features—never mind the other browsers.

This was the dark age of compatibility, often called the browser wars. Web developers were left with not one unified Web but two or three incompatible platforms. To make things worse, the browsers in use around 2003 were all full of bugs, and of course the bugs were different for each browser. Life was hard for people writing web pages.

Mozilla Firefox, a not-for-profit offshoot of Netscape, challenged Internet Explorer’s hegemony in the late 2000s. Because Microsoft was not particularly interested in staying competitive at the time, Firefox took quite a chunk of market share away from it. Around the same time, Google introduced its Chrome browser, and Apple’s Safari browser gained popularity, leading to a situation where there were four major players, rather than one.

The new players had a more serious attitude toward standards and better engineering practices, leading to less incompatibility and fewer bugs. Microsoft, seeing its market share crumble, came around and adopted these attitudes. If you are starting to learn web development today, consider yourself lucky. The latest versions of the major browsers behave quite uniformly and have relatively few bugs.

That is not to say that the situation is perfect just yet. Some of the people using the Web are, for reasons of inertia or corporate policy, stuck with very old browsers. Until those browsers die out entirely, writing websites that work for them will require a lot of arcane knowledge about their shortcomings and quirks. This book is not about those quirks. Rather, it aims to present the modern, sane style of web programming.

# Chapter 13

# The Document Object Model

When you open a web page in your browser, the browser retrieves the page’s HTML text and parses it, much like the way our parser from [Chapter 11](http://eloquentjavascript.net/11_language.html#parsing) parsed programs. The browser builds up a model of the document’s structure and then uses this model to draw the page on the screen.

This representation of the document is one of the toys that a JavaScript program has available in its sandbox. You can read from the model and also change it. It acts as a live data structure: when it is modified, the page on the screen is updated to reflect the changes.

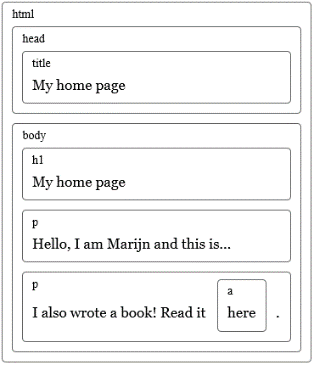
## Document structure

You can imagine an HTML document as a nested set of boxes. Tags such as <body> and </body> enclose other tags, which in turn contain other tags or text. Here’s the example document from the [previous chapter](http://eloquentjavascript.net/12_browser.html#browser):

edit & run code by clicking it

<!doctype html>

<html>

 <head>

<title>My home page</title>

</head>

<body>

<h1>My home page</h1>

<p>Hello, I am Marijn and this is my home page.</p>

<p>I also wrote a book! Read it

<a href="http://eloquentjavascript.net">here</a>.</p>

</body>

</html>

This page has the following structure:

The data structure the browser uses to represent the document follows this shape. For each box, there is an object, which we can interact with to find out things such as what HTML tag it represents and which boxes and text it contains. This representation is called the Document Object Model, or DOM for short.

The global variable document gives us access to these objects. Its documentElement property refers to the object representing the <html> tag. It also provides the properties head and body, which hold the objects for those elements.

## Trees

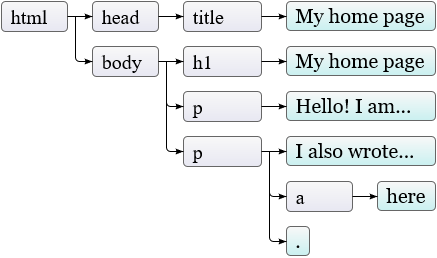
Think back to the syntax trees from [Chapter 11](http://eloquentjavascript.net/11_language.html#parsing) for a moment. Their structures are strikingly similar to the structure of a browser’s document. Each node may refer to other nodes, children, which in turn may have their own children. This shape is typical of nested structures where elements can contain sub-elements that are similar to themselves.

We call a data structure a tree when it has a branching structure, has no cycles (a node may not contain itself, directly or indirectly), and has a single, well-defined “root”. In the case of the DOM, document.documentElement serves as the root.

Trees come up a lot in computer science. In addition to representing recursive structures such as HTML documents or programs, they are often used to maintain sorted sets of data because elements can usually be found or inserted more efficiently in a sorted tree than in a sorted flat array.

A typical tree has different kinds of nodes. The syntax tree for [the Egg language](http://eloquentjavascript.net/11_language.html#language) had variables, values, and application nodes. Application nodes always have children, whereas variables and values are leaves, or nodes without children.

The same goes for the DOM. Nodes for regular elements, which represent HTML tags, determine the structure of the document. These can have child nodes. An example of such a node is document.body. Some of these children can be leaf nodes, such as pieces of text or comments (comments are written between <!-- and --> in HTML).

Each DOM node object has a nodeType property, which contains a numeric code that identifies the type of node. Regular elements have the value 1, which is also defined as the constant property document.ELEMENT\_NODE. Text nodes, representing a section of text in the document, have the value 3 (document.TEXT\_NODE). Comments have the value 8 (document.COMMENT\_NODE).

So another way to visualize our document tree is as follows:

The leaves are text nodes, and the arrows indicate parent-child relationships between nodes.

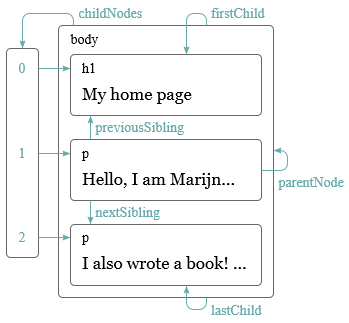
## The standard

Using cryptic numeric codes to represent node types is not a very JavaScript-like thing to do. Later in this chapter, we’ll see that other parts of the DOM interface also feel cumbersome and alien. The reason for this is that the DOM wasn’t designed for just JavaScript. Rather, it tries to define a language-neutral interface that can be used in other systems as well—not just HTML but also XML, which is a generic data format with an HTML-like syntax.

This is unfortunate. Standards are often useful. But in this case, the advantage (cross-language consistency) isn’t all that compelling. Having an interface that is properly integrated with the language you are using will save you more time than having a familiar interface across languages.

As an example of such poor integration, consider the childNodes property that element nodes in the DOM have. This property holds an array-like object, with a length property and properties labeled by numbers to access the child nodes. But it is an instance of the NodeList type, not a real array, so it does not have methods such as slice and forEach.

Then there are issues that are simply poor design. For example, there is no way to create a new node and immediately add children or attributes to it. Instead, you have to first create it, then add the children one by one, and finally set the attributes one by one, using side effects. Code that interacts heavily with the DOM tends to get long, repetitive, and ugly.

But these flaws aren’t fatal. Since JavaScript allows us to create our own abstractions, it is easy to write some helper functions that allow you to express the operations you are performing in a clearer and shorter way. In fact, many libraries intended for browser programming come with such tools.

## Moving through the tree

DOM nodes contain a wealth of links to other nearby nodes. The following diagram illustrates these:

Although the diagram shows only one link of each type, every node has a parentNode property that points to its containing node. Likewise, every element node (node type 1) has a childNodes property that points to an array-like object holding its children.

In theory, you could move anywhere in the tree using just these parent and child links. But JavaScript also gives you access to a number of additional convenience links. The firstChild and lastChild properties point to the first and last child elements or have the value null for nodes without children. Similarly, previousSibling and nextSibling point to adjacent nodes, which are nodes with the same parent that appear immediately before or after the node itself. For a first child, previousSibling will be null, and for a last child, nextSibling will be null.

When dealing with a nested data structure like this one, recursive functions are often useful. The following recursive function scans a document for text nodes containing a given string and returns true when it has found one:

function talksAbout(node, string) {

if (node.nodeType == document.ELEMENT\_NODE) {

for (var i = 0; i < node.childNodes.length; i++) {

if (talksAbout(node.childNodes[i], string))

return true;

}

return false;

} else if (node.nodeType == document.TEXT\_NODE) {

return node.nodeValue.indexOf(string) > -1;

}

}

console.log(talksAbout(document.body, "book"));

// → true

The nodeValue property of a text node refers to the string of text that it represents.

## Finding elements

Navigating these links among parents, children, and siblings is often useful, as in the previous function, which runs through the whole document. But if we want to find a specific node in the document, reaching it by starting at document.body and blindly following a hard-coded path of links is a bad idea. Doing so bakes assumptions into our program about the precise structure of the document—a structure we might want to change later. Another complicating factor is that text nodes are created even for the whitespace between nodes. The example document’s body tag does not have just three children (<h1> and two <p> elements) but actually has seven: those three, plus the spaces before, after, and between them.

So if we want to get the href attribute of the link in that document, we don’t want to say something like “Get the second child of the sixth child of the document body”. It’d be better if we could say “Get the first link in the document”. And we can.

var link = document.body.getElementsByTagName("a")[0];

console.log(link.href);

All element nodes have a getElementsByTagName method, which collects all elements with the given tag name that are descendants (direct or indirect children) of the given node and returns them as an array-like object.

To find a specific single node, you can give it an id attribute and use document.getElementById instead.

<p>My ostrich Gertrude:</p>

<p><img id="gertrude" src="img/ostrich.png"></p>

<script>

var ostrich = document.getElementById("gertrude");

console.log(ostrich.src);

</script>

A third, similar method is getElementsByClassName, which, like getElementsByTagName, searches through the contents of an element node and retrieves all elements that have the given string in their class attribute.

## Changing the document

Almost everything about the DOM data structure can be changed. Element nodes have a number of methods that can be used to change their content. The removeChild method removes the given child node from the document. To add a child, we can use appendChild, which puts it at the end of the list of children, or insertBefore, which inserts the node given as the first argument before the node given as the second argument.

<p>One</p>

<p>Two</p>

<p>Three</p>

<script>

var paragraphs = document.body.getElementsByTagName("p");

document.body.insertBefore(paragraphs[2], paragraphs[0]);

</script>

A node can exist in the document in only one place. Thus, inserting paragraph “Three” in front of paragraph “One” will first remove it from the end of the document and then insert it at the front, resulting in “Three/One/Two”. All operations that insert a node somewhere will, as a side effect, cause it to be removed from its current position (if it has one).

The replaceChild method is used to replace a child node with another one. It takes as arguments two nodes: a new node and the node to be replaced. The replaced node must be a child of the element the method is called on. Note that both replaceChild and insertBefore expect the new node as their first argument.

## Creating nodes

In the following example, we want to write a script that replaces all images (<img> tags) in the document with the text held in their alt attributes, which specifies an alternative textual representation of the image.

This involves not only removing the images but adding a new text node to replace them. For this, we use the document.createTextNode method.

<p>The <img src="img/cat.png" alt="Cat"> in the

<img src="img/hat.png" alt="Hat">.</p>

<p><button onclick="replaceImages()">Replace</button></p>

<script>

function replaceImages() {

var images = document.body.getElementsByTagName("img");

for (var i = images.length - 1; i >= 0; i--) {

var image = images[i];

if (image.alt) {

var text = document.createTextNode(image.alt);

image.parentNode.replaceChild(text, image);

}

}

}

</script>

Given a string, createTextNode gives us a type 3 DOM node (a text node), which we can insert into the document to make it show up on the screen.

The loop that goes over the images starts at the end of the list of nodes. This is necessary because the node list returned by a method like getElementsByTagName (or a property like childNodes) is live. That is, it is updated as the document changes. If we started from the front, removing the first image would cause the list to lose its first element so that the second time the loop repeats, where i is 1, it would stop because the length of the collection is now also 1.

If you want a solid collection of nodes, as opposed to a live one, you can convert the collection to a real array by calling the array slice method on it.

var arrayish = {0: "one", 1: "two", length: 2};

var real = Array.prototype.slice.call(arrayish, 0);

real.forEach(function(elt) { console.log(elt); });

// → one

// two

To create regular element nodes (type 1), you can use the document.createElement method. This method takes a tag name and returns a new empty node of the given type.

The following example defines a utility elt, which creates an element node and treats the rest of its arguments as children to that node. This function is then used to add a simple attribution to a quote.

<blockquote id="quote">

No book can ever be finished. While working on it we learn

just enough to find it immature the moment we turn away

from it.

</blockquote>

<script>

function elt(type) {

var node = document.createElement(type);

for (var i = 1; i < arguments.length; i++) {

var child = arguments[i];

if (typeof child == "string")

child = document.createTextNode(child);

node.appendChild(child);

}

return node;

}

document.getElementById("quote").appendChild(

elt("footer", "—",

elt("strong", "Karl Popper"),

", preface to the second editon of ",

elt("em", "The Open Society and Its Enemies"),

", 1950"));

</script>

## Attributes

Some element attributes, such as href for links, can be accessed through a property of the same name on the element’s DOM object. This is the case for a limited set of commonly used standard attributes.

But HTML allows you to set any attribute you want on nodes. This can be useful because it allows you to store extra information in a document. If you make up your own attribute names, though, such attributes will not be present as a property on the element’s node. Instead, you’ll have to use the getAttribute and setAttribute methods to work with them.

<p data-classified="secret">The launch code is 00000000.</p>

<p data-classified="unclassified">I have two feet.</p>

<script>

var paras = document.body.getElementsByTagName("p");

Array.prototype.forEach.call(paras, function(para) {

if (para.getAttribute("data-classified") == "secret")

para.parentNode.removeChild(para);

});

</script>

I recommended prefixing the names of such made-up attributes with data- to ensure they do not conflict with any other attributes.

As a simple example, we’ll write a “syntax highlighter” that looks for <pre> tags (“preformatted”, used for code and similar plaintext) with a data-language attribute and crudely tries to highlight the keywords for that language.

function highlightCode(node, keywords) {

var text = node.textContent;

node.textContent = ""; // Clear the node

var match, pos = 0;

while (match = keywords.exec(text)) {

var before = text.slice(pos, match.index);

node.appendChild(document.createTextNode(before));

var strong = document.createElement("strong");

strong.appendChild(document.createTextNode(match[0]));

node.appendChild(strong);

pos = keywords.lastIndex;

}

var after = text.slice(pos);

node.appendChild(document.createTextNode(after));

}

The function highlightCode takes a <pre> node and a regular expression (with the “global” option turned on) that matches the keywords of the programming language that the element contains.

The textContent property is used to get all the text in the node and is then set to an empty string, which has the effect of emptying the node. We loop over all matches of the keyword expression, appending the text between them as regular text nodes, and the text matched (the keywords) as text nodes wrapped in <strong> (bold) elements.

We can automatically highlight all programs on the page by looping over all the <pre> elements that have a data-language attribute and calling highlightCode on each one with the correct regular expression for the language.

var languages = {

javascript: /\b(function|return|var)\b/g /\* … etc \*/

};

function highlightAllCode() {

var pres = document.body.getElementsByTagName("pre");

for (var i = 0; i < pres.length; i++) {

var pre = pres[i];

var lang = pre.getAttribute("data-language");

if (languages.hasOwnProperty(lang))

highlightCode(pre, languages[lang]);

}

}

Here is an example:

<p>Here it is, the identity function:</p>

<pre data-language="javascript">

function id(x) { return x; }

</pre>

<script>highlightAllCode();</script>

There is one commonly used attribute, class, which is a reserved word in the JavaScript language. For historical reasons—some old JavaScript implementations could not handle property names that matched keywords or reserved words—the property used to access this attribute is called className. You can also access it under its real name, "class", by using the getAttribute and setAttribute methods.

## Layout

You might have noticed that different types of elements are laid out differently. Some, such as paragraphs (<p>) or headings (<h1>), take up the whole width of the document and are rendered on separate lines. These are called block elements. Others, such as links (<a>) or the <strong> element used in the previous example, are rendered on the same line with their surrounding text. Such elements are called inline elements.

For any given document, browsers are able to compute a layout, which gives each element a size and position based on its type and content. This layout is then used to actually draw the document.

The size and position of an element can be accessed from JavaScript. The offsetWidth and offsetHeight properties give you the space the element takes up in pixels. A pixel is the basic unit of measurement in the browser and typically corresponds to the smallest dot that your screen can display. Similarly, clientWidth and clientHeight give you the size of the space inside the element, ignoring border width.

<p style="border: 3px solid red">

I'm boxed in

</p>

<script>

var para = document.body.getElementsByTagName("p")[0];

console.log("clientHeight:", para.clientHeight);

console.log("offsetHeight:", para.offsetHeight);

</script>

The most effective way to find the precise position of an element on the screen is the getBoundingClientRect method. It returns an object with top, bottom, left, and right properties, indicating the pixel positions of the sides of the element relative to the top left of the screen. If you want them relative to the whole document, you must add the current scroll position, found under the global pageXOffset and pageYOffset variables.

Laying out a document can be quite a lot of work. In the interest of speed, browser engines do not immediately re-layout a document every time it is changed but rather wait as long as they can. When a JavaScript program that changed the document finishes running, the browser will have to compute a new layout in order to display the changed document on the screen. When a program asks for the position or size of something by reading properties such as offsetHeight or calling getBoundingClientRect, providing correct information also requires computing a layout.

A program that repeatedly alternates between reading DOM layout information and changing the DOM forces a lot of layouts to happen and will consequently run really slowly. The following code shows an example of this. It contains two different programs that build up a line of X characters 2,000 pixels wide and measures the time each one takes.

<p><span id="one"></span></p>

<p><span id="two"></span></p>

<script>

function time(name, action) {

var start = Date.now(); // Current time in milliseconds

action();

console.log(name, "took", Date.now() - start, "ms");

}

time("naive", function() {

var target = document.getElementById("one");

while (target.offsetWidth < 2000)

target.appendChild(document.createTextNode("X"));

});

// → naive took 32 ms

time("clever", function() {

var target = document.getElementById("two");

target.appendChild(document.createTextNode("XXXXX"));

var total = Math.ceil(2000 / (target.offsetWidth / 5));

for (var i = 5; i < total; i++)

target.appendChild(document.createTextNode("X"));

});

// → clever took 1 ms

</script>

## Styling

We have seen that different HTML elements display different behavior. Some are displayed as blocks, others inline. Some add styling, such as <strong> making its content bold and <a> making it blue and underlining it.

The way an <img> tag shows an image or an <a> tag causes a link to be followed when it is clicked is strongly tied to the element type. But the default styling associated with an element, such as the text color or underline, can be changed by us. Here is an example using the style property:

<p><a href=".">Normal link</a></p>

<p><a href="." style="color: green">Green link</a></p>

A style attribute may contain one or more declarations, which are a property (such as color) followed by a colon and a value (such as green). When there is more than one declaration, they must be separated by semicolons, as in "color: red; border: none".

There are a lot of aspects that can be influenced by styling. For example, the display property controls whether an element is displayed as a block or an inline element.

This text is displayed <strong>inline</strong>,

<strong style="display: block">as a block</strong>, and

<strong style="display: none">not at all</strong>.

The block tag will end up on its own line since block elements are not displayed inline with the text around them. The last tag is not displayed at all—display: none prevents an element from showing up on the screen. This is a way to hide elements. It is often preferable to removing them from the document entirely because it makes it easy to reveal them again at a later time.

JavaScript code can directly manipulate the style of an element through the node’s style property. This property holds an object that has properties for all possible style properties. The values of these properties are strings, which we can write to in order to change a particular aspect of the element’s style.

<p id="para" style="color: purple">

Pretty text

</p>

<script>

var para = document.getElementById("para");

console.log(para.style.color);

para.style.color = "magenta";

</script>

Some style property names contain dashes, such as font-family. Because such property names are awkward to work with in JavaScript (you’d have to say style["font-family"]), the property names in the style object for such properties have their dashes removed and the letters that follow them capitalized (style.fontFamily).

## Cascading styles

The styling system for HTML is called CSS for Cascading Style Sheets. A style sheet is a set of rules for how to style elements in a document. It can be given inside a <style> tag.

<style>

strong {

font-style: italic;

color: gray;

}

</style>

<p>Now <strong>strong text</strong> is italic and gray.</p>

The cascading in the name refers to the fact that multiple such rules are combined to produce the final style for an element. In the previous example, the default styling for <strong> tags, which gives them font-weight: bold, is overlaid by the rule in the <style> tag, which adds font-style and color.

When multiple rules define a value for the same property, the most recently read rule gets a higher precedence and wins. So if the rule in the <style> tag included font-weight: normal, conflicting with the default font-weight rule, the text would be normal, not bold. Styles in a style attribute applied directly to the node have the highest precedence and always win.

It is possible to target things other than tag names in CSS rules. A rule for .abc applies to all elements with "abc" in their class attributes. A rule for #xyz applies to the element with an id attribute of "xyz" (which should be unique within the document).

.subtle {

color: gray;

font-size: 80%;

}

#header {

background: blue;

color: white;

}

/\* p elements, with classes a and b, and id main \*/

p.a.b#main {

margin-bottom: 20px;

}

The precedence rule favoring the most recently defined rule holds true only when the rules have the same specificity. A rule’s specificity is a measure of how precisely it describes matching elements, determined by the number and kind (tag, class, or ID) of element aspects it requires. For example, a rule that targets p.a is more specific than rules that target p or just .a, and would thus take precedence over them.

The notation p > a {…} applies the given styles to all <a> tags that are direct children of <p> tags. Similarly, p a {…} applies to all <a> tags inside <p> tags, whether they are direct or indirect children.

## Query selectors

We won’t be using style sheets all that much in this book. Although understanding them is crucial to programming in the browser, properly explaining all the properties they support and the interaction among those properties would take two or three books.

The main reason I introduced selector syntax—the notation used in style sheets to determine which elements a set of styles apply to—is that we can use this same mini-language as an effective way to find DOM elements.

The querySelectorAll method, which is defined both on the document object and on element nodes, takes a selector string and returns an array-like object containing all the elements that it matches.

<p>And if you go chasing

<span class="animal">rabbits</span></p>

<p>And you know you're going to fall</p>

<p>Tell 'em a <span class="character">hookah smoking

<span class="animal">caterpillar</span></span></p>

<p>Has given you the call</p>

<script>

function count(selector) {

return document.querySelectorAll(selector).length;

}

console.log(count("p")); // All <p> elements

// → 4

console.log(count(".animal")); // Class animal

// → 2

console.log(count("p .animal")); // Animal inside of <p>

// → 2

console.log(count("p > .animal")); // Direct child of <p>

// → 1

</script>

Unlike methods such as getElementsByTagName, the object returned by querySelectorAll is not live. It won’t change when you change the document.

The querySelector method (without the All part) works in a similar way. This one is useful if you want a specific, single element. It will return only the first matching element or null if no elements match.

## Positioning and animating

The position style property influences layout in a powerful way. By default it has a value of static, meaning the element sits in its normal place in the document. When it is set to relative, the element still takes up space in the document, but now the top and left style properties can be used to move it relative to its normal place. When position is set to absolute, the element is removed from the normal document flow—that is, it no longer takes up space and may overlap with other elements. Also, its top and left properties can be used to absolutely position it relative to the top-left corner of the nearest enclosing element whose position property isn’t static, or relative to the document if no such enclosing element exists.

We can use this to create an animation. The following document displays a picture of a cat that floats around in an ellipse:

<p style="text-align: center">

<img src="img/cat.png" style="position: relative">

</p>

<script>

var cat = document.querySelector("img");

var angle = 0, lastTime = null;

function animate(time) {

if (lastTime != null)

angle += (time - lastTime) \* 0.001;

lastTime = time;

cat.style.top = (Math.sin(angle) \* 20) + "px";

cat.style.left = (Math.cos(angle) \* 200) + "px";

requestAnimationFrame(animate);

}

requestAnimationFrame(animate);

</script>

The picture is centered on the page and given a position of relative. We’ll repeatedly update that picture’s top and left styles in order to move it.

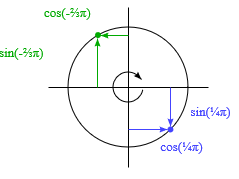
The script uses requestAnimationFrame to schedule the animate function to run whenever the browser is ready to repaint the screen. The animate function itself again calls requestAnimationFrame to schedule the next update. When the browser window (or tab) is active, this will cause updates to happen at a rate of about 60 per second, which tends to produce a good-looking animation.

If we just updated the DOM in a loop, the page would freeze and nothing would show up on the screen. Browsers do not update their display while a JavaScript program is running, nor do they allow any interaction with the page. This is why we need requestAnimationFrame—it lets the browser know that we are done for now, and it can go ahead and do the things that browsers do, such as updating the screen and responding to user actions.

Our animation function is passed the current time as an argument, which it compares to the time it saw before (the lastTime variable) to ensure the motion of the cat per millisecond is stable, and the animation moves smoothly. If it just moved a fixed amount per step, the motion would stutter if, for example, another heavy task running on the same computer were to prevent the function from running for a fraction of a second.

Moving in circles is done using the trigonometry functions Math.cos and Math.sin. For those of you who aren’t familiar with these, I’ll briefly introduce them since we will occasionally need them in this book.

Math.cos and Math.sin are useful for finding points that lie on a circle around point (0,0) with a radius of one unit. Both functions interpret their argument as the position on this circle, with zero denoting the point on the far right of the circle, going clockwise until 2π (about 6.28) has taken us around the whole circle. Math.cos tells you the x-coordinate of the point that corresponds to the given position around the circle, while Math.sin yields the y-coordinate. Positions (or angles) greater than 2π or less than 0 are valid—the rotation repeats so that a+2π refers to the same angle as a.

The cat animation code keeps a counter, angle, for the current angle of the animation and increments it in proportion to the elapsed time every time the animate function is called. It can then use this angle to compute the current position of the image element. The top style is computed with Math.sin and multiplied by 20, which is the vertical radius of our circle. The left style is based on Math.cos and multiplied by 200 so that the circle is much wider than it is high, resulting in an elliptic motion.

Note that styles usually need units. In this case, we have to append "px" to the number to tell the browser we are counting in pixels (as opposed to centimeters, “ems”, or other units). This is easy to forget. Using numbers without units will result in your style being ignored—unless the number is 0, which always means the same thing, regardless of its unit.

## Summary

JavaScript programs may inspect and interfere with the current document that a browser is displaying through a data structure called the DOM. This data structure represents the browser’s model of the document, and a JavaScript program can modify it to change the visible document.

The DOM is organized like a tree, in which elements are arranged hierarchically according to the structure of the document. The objects representing elements have properties such as parentNode and childNodes, which can be used to navigate through this tree.

The way a document is displayed can be influenced by styling, both by attaching styles to nodes directly and by defining rules that match certain nodes. There are many different style properties, such as color or display. JavaScript can manipulate an element’s style directly through its style property.