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 actually express some meaningful prose.

EXPRESSIONS AND STATEMENTS

In Chapter 1, 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 still 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:

```
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.

Semicolons

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 strange things will happen. 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
```

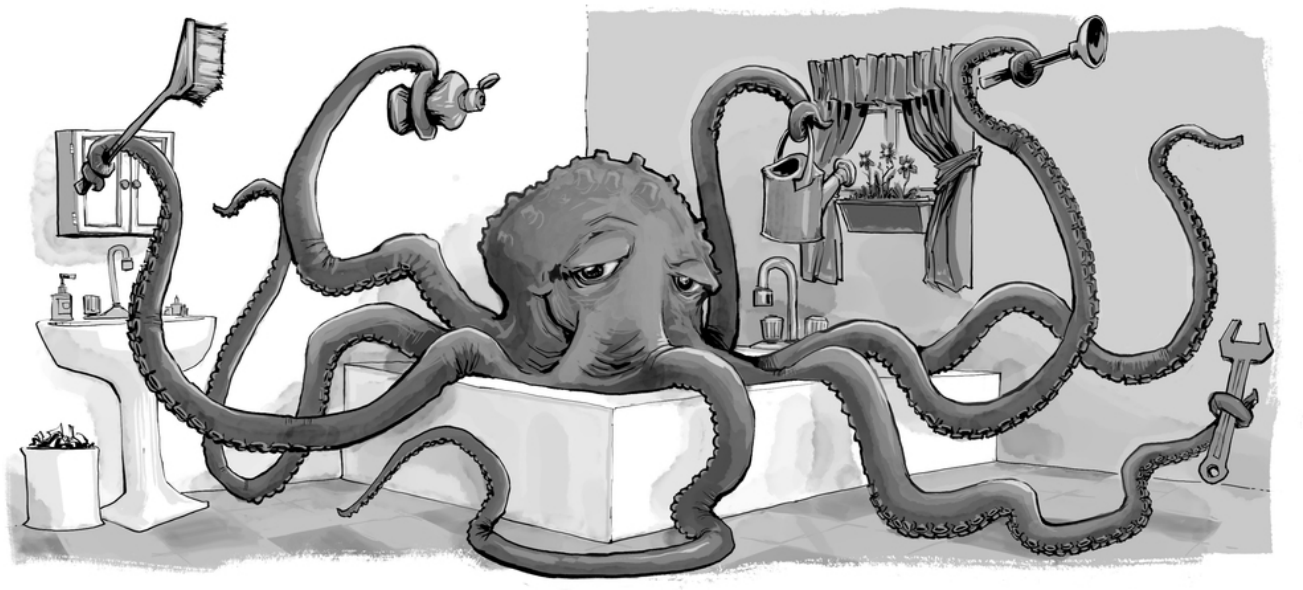
Variable names can be any word that isn't reserved as a keyword (such as `var`). They may not include spaces. Digits can also be part of variable names—`catch22` is a valid name, for example—but the name must not start with a digit. A variable name cannot include punctuation, except for the characters "$" 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 continue debugger default delete
do else false finally for function if implements
in instanceof interface let new null package private
protected public return static switch throw true
try typeof var void while with yield this
```

Don't worry about memorizing these, but remember that this might be the problem when something does not work as expected.

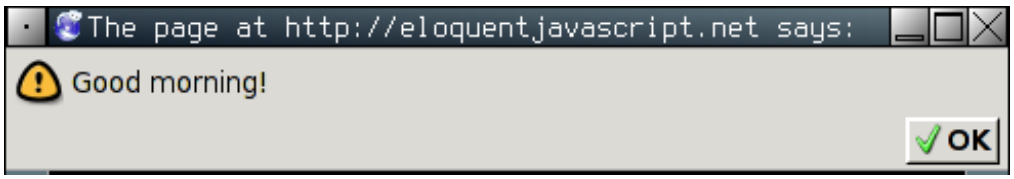
The environment

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 refer to a variable that holds a 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 dots, `console.log` clearly has one. This is because `console.log` isn't a simple variable. It is actually an expression that retrieves the `log` field from the value held by the `console` variable. We will find out exactly what this means in Chapter 4.

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 can 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:

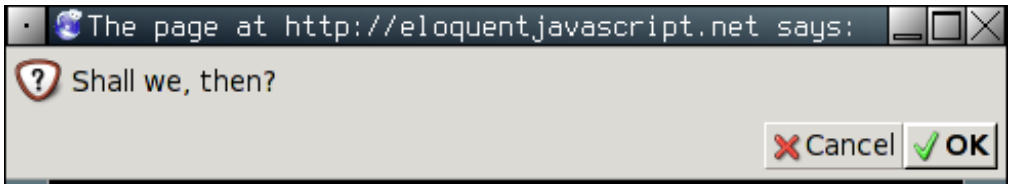
```
console.log(Math.min(2, 4) + 100);
// → 102
```

The next chapter explains how we can write our own functions.

Prompt and confirm

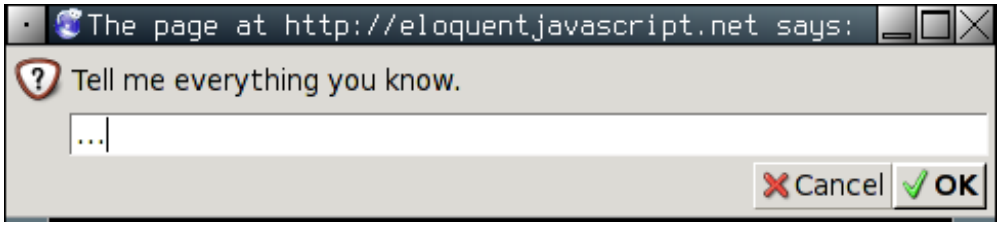
Browser environments contain other functions besides `alert` for popping up windows. You can ask the user an "OK"/"Cancel" question using `confirm`. This returns a boolean: `true` if the user clicks OK and `false` if the user clicks Cancel.

```
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 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.

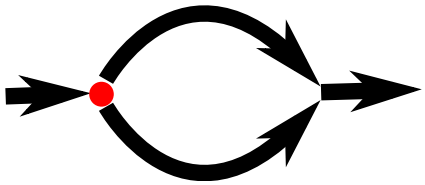
```
var theNumber = Number(prompt("Pick a number", ""));
alert("Your number is the square root of " +
      theNumber * theNumber);
```

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.



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 keywords, between parentheses, followed by the statement to execute.

The `isNaN` function is a standard JavaScript function that returns `true` 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 won't always just have code that executes when a condition holds true.

Often, you'll also need code that handles the other case, when the condition doesn't hold. This alternate path is represented by the second arrow in the previous diagram. The `else` keyword can be used, together with `if`, to create two separate, parallel 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?");
```

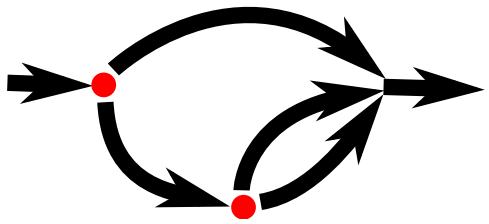
If we have more than two paths to choose from, multiple `if`/`else` pairs can be "chained" together. Here's an example:

```
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:

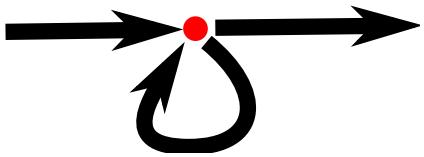


WHILE AND DO LOOPS

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—a *loop*.



Looping control flow allows us to go back to some point in the program where we were before and repeat it in the context of our current program state. If we combine this with a variable that counts, we can do 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 want to execute multiple statements inside a loop, we wrap them in 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 2^{10} (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 later, 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 name = prompt("Who are you?");
} while (!name);
console.log(name);
```

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`.)

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 even-number-printing example. The only change is that all the statements that are related to the "state" of the loop are now on one line.

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 2^{10}, 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 its 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
```

The trick with 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.

Updating variables succinctly

Especially when looping, a program often needs to "update" a variable with 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 using underscores, though that style is a little painful to type. However, 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. 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
 occasionally dropped by, showing up in phone numbers and
 the serial numbers of products that I bought. It
 obviously likes me, so I've decided to keep it.
*/
var theNumber = 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 in a program. 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)`, which is an expression that may produce a value.

Exercises

If you are unsure how to try your solutions to exercises, refer to the end of the introduction.

Each exercise starts with a problem description. Read that and try to solve the exercise. If you run into problems, consider reading the hints after the exercise. Full solutions to the exercises are not included in this book, but you can find them online at *http://eloquentjavascript.net/exercises.html*. If you want to learn, I recommend looking at these only after you've solved the exercise, or at least after you've attacked it long and hard enough to have a slight headache.

Looping a triangle

Write a program that makes seven calls to `console.log` to output the following triangle:

```
#
##
###
####
#####
######
#######

// Your code here.
// (Click code listings to edit them.)
```

» Display hints...

FizzBuzz

Write a program that uses `console.log` to print all the numbers from 1 to 100, with two exceptions. For numbers divisible by 3, print `"Fizz"` instead of the number, and for numbers divisible by 5 (and not 3), print `"Buzz"` instead.

When you have that working, modify your program to print `"FizzBuzz"` for numbers that are divisible by both 3 and 5.

(This is actually an interview question that has been claimed to weed out a significant percentage of programmer candidates. So if you solved it, you're now allowed to feel good about yourself.)

```
// Your code here.
```

» Display hints...

Chess board

Write a program that creates a string that represents an 8×8 grid, using newline characters to separate lines. At each position of the grid there is either a space or a "`#`" character. The characters should form a chess board.

Passing this string to `console.log` should look like this:

```
# # # #
 # # # #
# # # #
 # # # #
# # # #
 # # # #
# # # #
 # # # #
```

When you have a program that generates this pattern, define a variable `size = 8` and change the program so that it works for any `size`, outputting a grid of the given width and height.

```
// Your code here.
```

» Display hints...