2

PROGRAM STRUCTURE

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 meaningful prose.

Expressions and Statements

In Chapter 1, we made values and applied operators to them to get new values. Creating values like this is the main substance of any JavaScript program. But that substance has to be framed in a larger structure to be useful. That's what we'll cover in this chapter.

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 contain other expressions in a way similar to how subsentences in human languages are nested—a subsentence can contain its own subsentences, and so on. This allows us to build expressions that describe arbitrarily complex computations.

If an expression corresponds to a sentence fragment, a JavaScript *statement* corresponds to a full sentence. A program is 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 code. However, a statement stands on its own, so if it doesn't affect the world, it's useless. It may display something on the screen, as with `console.log`, or change the 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 you run this 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. So in this book, every statement that needs a semicolon will always get one. I recommend you do the same, at least until you've learned more about the subtleties of missing semicolons.

Bindings

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 must be used immediately or it will dissipate again. To catch and hold values, JavaScript provides a thing called a *binding*, or *variable*.

```
let caught = 5 * 5;
```

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

The example creates a binding called `caught` and uses it to grab hold of the number that is produced by multiplying 5 by 5.

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

```
let ten = 10;
console.log(ten * ten);
// → 100
```

When a binding 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 bindings to disconnect them from their current value and have them point to a new one.

```
let mood = "light";
console.log(mood);
// → light
mood = "dark";
console.log(mood);
// → dark
```

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

Let's look at another example. To remember the number of dollars that Luigi still owes you, you create a binding. When he pays back $35, you give this binding a new value.

```
let luigisDebt = 140;
luigisDebt = luigisDebt - 35;
console.log(luigisDebt);
// → 105
```

When you define a binding 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 binding, you'll get the value `undefined`.

A single `let` statement may define multiple bindings. The definitions must be separated by commas.

```
let one = 1, two = 2;
console.log(one + two);
// → 3
```

The words `var` and `const` can also be used to create bindings, in a similar fashion to `let`.

```
var name = "Ayda";
const greeting = "Hello ";
console.log(greeting + name);
// → Hello Ayda
```

The first of these, `var` (short for "variable"), is the way bindings were declared in pre-2015 JavaScript, when `let` didn't exist yet. I'll get back to the precise way it differs from `let` in the next chapter. For now, remember that it mostly does the same thing, but we'll rarely use it in this book because it behaves oddly in some situations.

The word `const` stands for "constant." It defines a constant binding, which points at the same value for as long as it lives. This is useful for bindings that just give a name to a value so that you can easily refer to it later.

Binding Names

Binding names can be any sequence of one or more letters. Digits can be part of binding names—`catch22` is a valid name, for example—but the name must not start with a digit. A binding name may include dollar signs (`$`) or underscores (_) but no other punctuation or special characters.

Words with a special meaning, such as `let`, are *keywords*, and may not be used as binding names. There are also a number of words that are "reserved for use" in future versions of JavaScript, which also can't be used as binding names. 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 interface in instanceof l
new package private protected public return static supe
switch this throw true try typeof var void while with y
```

Don’t worry about memorizing this list. When creating a binding produces an unexpected syntax error, check whether you’re trying to define a reserved word.

The Environment

The collection of bindings 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 bindings that are part of the language standard, and most of the time, it also has bindings that provide ways to interact with the surrounding system. For example, in a browser, there are functions to interact with 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 binding `prompt` holds a function that shows a little dialog asking for user input. It is used like this:

```
prompt("Enter passcode");
```

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 binding that holds the function. The values between the parentheses are given to the program inside the function. In the example, the `prompt` function uses the string that we give it

as the text to show in the dialog. Values given to functions are called *arguments*. Different functions might need a different number or different types of arguments.

The `prompt` function isn't used much in modern web programming, mostly because you have no control over the way the resulting dialog looks, but it can be helpful in toy programs and experiments.

The console.log Function

In the examples, I 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 a Mac, COMMAND-OPTION-I. If that does not work, search through the menus for an item named Developer Tools or similar.

Though binding names cannot contain period characters, `console.log` does have one. This is because `console.log` isn't a simple binding, but an expression that retrieves the `log` property from the value held by the `console` binding. We'll find out exactly what this means in Chapter 4.

Return Values

Showing a dialog or writing text to the screen is a *side effect*. Many functions are useful because of the side effects they produce. Functions may also produce values, in which case they don't need to have a side effect to be useful. For example, the function `Math.max` takes any amount of number arguments 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 that function calls can be used within larger expressions. In the following code, a call to `Math.min`, which is the opposite of `Math.max`, is used as part of a plus expression:

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

Chapter 3 will explain how to write your own functions.

Control Flow

When your program contains more than one statement, the statements are executed as though they were a story, from top to bottom. For example, the following program has two statements. The first asks the user for a number, and the second, which is executed after the first, shows the square of that number.

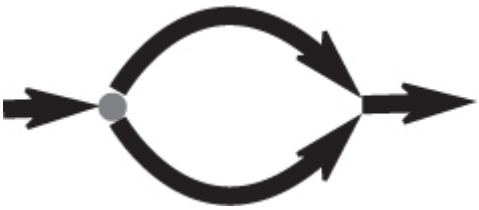
```
let theNumber = Number(prompt("Pick a number"));
console.log("Your number is the square root of " +
            theNumber * theNumber);
```

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-line control flow:

Conditional Execution

Not all programs are straight roads. We may, for example, want to create a branching road where the program takes the proper branch based on the situation at hand. This is called *conditional execution*.



Conditional execution is created with the `if` keyword in JavaScript. In the simple case, we want some code to be executed if, and only if, a certain condition holds. We might, for example, want to show the square of the input only if the input is actually a number.

```
let theNumber = Number(prompt("Pick a number"));
if (!Number.isNaN(theNumber)) {
  console.log("Your number is the square root of " +
              theNumber * theNumber);
}
```

With this modification, if you enter **parrot**, no output is shown.

The `if` keyword 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 `Number.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."

The statement after the `if` is wrapped in braces (`{` and `}`) in this example. The braces can be used to group any number of statements into a single statement, called a *block*. You could also have omitted them in this case, since they hold only a single statement, but to avoid having to think about whether they are needed, most JavaScript programmers use them in every wrapped statement like this. We'll mostly follow that convention in this book, except for the occasional one-liner.

```
if (1 + 1 == 2) console.log("It's true");
// → It's true
```

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. You can use the `else` keyword, together with `if`, to create two separate, alternative execution paths.

```
let theNumber = Number(prompt("Pick a number"));
if (!Number.isNaN(theNumber)) {
  console.log("Your number is the square root of " +
              theNumber * theNumber);
} else {
```

```
  console.log("Hey. Why didn't you give me a number?");
}
```

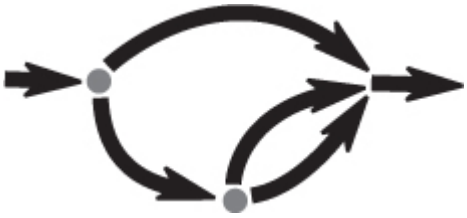
If you have more than two paths to choose from, you can "chain" multiple `if`/`else` pairs together. Here's an example:

```
let num = Number(prompt("Pick a number"));

if (num < 10) {
  console.log("Small");
} else if (num < 100) {
  console.log("Medium");
} else {
  console.log("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 at least 10 but below 100, and "`Medium`" is shown. If it doesn't, the second and last `else` branch is chosen.

The schema for this program looks something like this:



while and do Loops

Consider a program that outputs 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, this approach would be unworkable. What we need is a way to run a piece of code multiple times. 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 binding that counts, we can do something like this:

```
let 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 keeps entering that statement as long as the expression produces a value that gives `true` when converted to Boolean.

The `number` binding demonstrates the way a binding can track the progress of a program. Every time the loop repeats, `number` gets a value that is 2 more than its previous value. At the beginning of every repetition, it is compared with the number 12 to decide whether the program's work is finished.

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 bindings: 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 binding has reached 10 yet and, if not, updates both bindings.

```
let result = 1;
let counter = 0;
```

```
while (counter < 10) {
  result = result * 2;
  counter = counter + 1;
}
console.log(result);
// → 1024
```

The counter could also have started at `1` and checked for `<= 10`, but for reasons that will become apparent in Chapter 4, it is a good idea to get used to counting from 0.

Note that JavaScript also has an operator for exponentiation (`2 ** 10`), which you would use to compute this in real code—but that would have ruined the example.

A `do` loop is a control structure similar to a `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.

```
let yourName;
do {
  yourName = prompt("Who are you?");
} while (!yourName);
console.log("Hello " + 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 non-empty name.

Indenting Code

In the examples, I've been adding spaces in front of statements that are part of some larger statement. These spaces 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 this indentation inside blocks is to make the structure of the code stand out to human readers. In 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. The important thing is that each new block adds the same amount of space.

```
if (false != true) {
  console.log("That makes sense.");
  if (1 < 2) {
    console.log("No surprise there.");
  }
}
```

Most code editor programs will help by automatically indenting new lines the proper amount.

for Loops

Many loops follow the pattern shown in the `while` examples. First a "counter" binding is created to track the progress of the loop. Then comes a `while` loop, usually with a test expression that checks whether the counter has reached its end value. 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 (let 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 (on page 30). The only change is that all the statements that are

related to the “state” of the loop are grouped together after `for`.

The parentheses after a `for` keyword must contain two semicolons. The part before the first semicolon *initializes* the loop, usually by defining a binding. 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.

This is the code that computes 2^{10} using `for` instead of `while`:

```
let result = 1;
for (let counter = 0; counter < 10; counter = counter +
  result = result * 2;
}
console.log(result);
// → 1024
```

Breaking Out of a Loop

Having the looping condition produce `false` is not the only way a loop can finish. The `break` statement has the effect of immediately jumping out of the enclosing loop. Its use is demonstrated in the following program, which finds the first number that is both greater than or equal to 20 and divisible by 7.

```
for (let current = 20; ; current = current + 1) {
  if (current % 7 == 0) {
    console.log(current);
    break;
  }
}
// → 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 remove that `break` statement or you accidentally write an end 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.

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 Bindings Succinctly

Especially when looping, a program often needs to "update" a binding to hold a value based on that binding'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 further shorten our counting example.

```
for (let 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 not uncommon for code to look like this:

```
if (x == "value1") action1();
else if (x == "value2") action2();
```

```
else if (x == "value3") action3();
else defaultAction();
```

There is a construct called `switch` that is intended to express 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 may look 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 start executing at the label that corresponds to the value that `switch` was given, or at `default` if no matching value is found. It will continue executing, even across other labels, 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). Be careful, though—it is easy to forget such a `break`, which will cause the program to execute code you do not want executed.

Capitalization

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

```
fuzzylittleturtle
fuzzy_little_turtle
```

```
FuzzyLittleTurtle
fuzzyLittleTurtle
```

The first style can be hard to read. I rather 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 final 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 follow this convention.

In a few cases, such as the `Number` function, the first letter of a binding is also capitalized. This was done to mark this function as a constructor. It will become clear what a constructor is in Chapter 6. For now, the important thing is to not 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 want to include some related 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.

```
let accountBalance = calculateBalance(account);
// It's a green hollow where a river sings
accountBalance.adjust();
// Madly catching white tatters in the grass.
let 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 in its entirety, regardless of whether it contains line breaks. This is useful for adding blocks of information about a file or a chunk of program.

```
/*
  I first found this number scrawled on the back of an
  notebook. Since then, it has often dropped by, showir
  phone numbers and the serial numbers of products that
  bought. It obviously likes me, so I've decided to kee
*/
const 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.

Bindings can be used to file pieces of data under a name, and they are useful for tracking state in your program. The environment is the set of bindings that are defined. JavaScript systems always put a number of useful standard bindings into your environment.

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

Exercises

If you are unsure how to test your solutions to the exercises, refer to the introduction.

Each exercise starts with a problem description. Read this description and try to solve the exercise. If you run into problems, consider reading the hints at the end of the book. You can find full solutions to the exercises online at *https://eloquentjavascript.net/code#2*. If you want to learn something from the exercises, I recommend looking at the solutions 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 loop that makes seven calls to `console.log` to output the following triangle:

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

It may be useful to know that you can find the length of a string by writing `.length` after it.

```
let abc = "abc";
console.log(abc.length);
// → 3
```

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 (and still print "`Fizz`" or "`Buzz`" for numbers divisible by only one of those).

(This is actually an interview question that has been claimed to weed out a significant percentage of programmer candidates. So if you solved it, your labor market value just went up.)

Chessboard

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

Passing this string to `console.log` should show something like this:

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

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