In Chapter 1 we covered some of the basics of JavaScript. We delved quite deep into a few of the concepts that people struggle with when learning the language. We didn’t really address the language as a whole, though, which is what we’ll do now. In this chapter we’ll dive into the details we glossed over in Chapter 1 and get to the nuts and bolts of the language. We’ll also discuss some of the things we touched on in Chapter 1 in more detail.

This chapter will provide you with a solid grounding in the JavaScript language, and will do so in a way that’s both accessible to novices to the language and still a valuable reference for the experienced JavaScript developer. Our hope is that, as you progress in your JavaScript development skills, you’ll refer to this chapter both to remind yourself of the basics and to dive into specific topics more deeply.

We will begin by reviewing some basic matters of formatting JavaScript code, especially as related to the examples in this book. Then we will cover expressions and statements, the two most basic building blocks of JavaScript from which all JavaScript programs are built. With that groundwork laid, we can then discuss creating more complex statements with operators. We will talk about variables and how to manage them in your JavaScript programs. Then we will discuss objects and arrays, which will give you the building blocks for everything else. Then we will have an in-depth discussion of functions: what they are, and how to make them, and we will gain some important insights into the dynamic nature of JavaScript. Finally, we will cover how to control our programs with conditionals and loops.

By the end of this chapter, you should have a solid understanding of JavaScript’s lexical structure and syntax, and should feel comfortable using its basic constructs for flow control and functionality.

---

**Note**  Throughout this chapter, we will be referring to, and even quoting directly, the ECMA-262 standard, the current version of which is ECMAScript Language Specification, 5.1 Edition. You are encouraged to explore the standard itself at www.ecma-international.org/ecma-262/5.1/ (which also provides a link for a downloadable PDF version), as this is an excellent way of expanding your understanding of JavaScript.

---

### Formatting JavaScript Code

Formatting code is one of the many subjects that will invariably result in a roomful of angry developers shouting at each other. (I once saw someone nearly throw a chair in the middle of an argument about indenting with spaces vs. tabs.) Even though it’s a touchy subject, this reference would be remiss without at least laying the groundwork for future arguments, as well as defining the conventions we will use throughout this book.

Broadly, JavaScript uses C-like formatting. Most notably, JavaScript uses curly brackets ({} ) to denote blocks of code, like loops or logical flow control.

JavaScript also uses two styles of comment delimiters. The double-slash (//) is the single-line delimiter, which indicates that everything from that point to the end of the line is a comment. JavaScript also uses /* to denote the beginning of a multiline comment and */ to indicate the end. Anything contained within those delimiters, regardless of new lines, is considered a comment.
Whitespace, including indentation, for the most part is unimportant. To quote Section 7.2 of the ECMA-262 standard: “White space characters are used to improve source text readability and to separate tokens (indivisible lexical units) from each other, but are otherwise insignificant.” JavaScript doesn’t care if you indent with tabs or spaces, or even if you indent at all. Similarly, JavaScript imposes no requirements for new lines. In fact, it’s common to “compress” JavaScript by removing all whitespace and running everything together on one line for the sake of reducing file size (see Chapter 4 for more information on compressing JavaScript).

JavaScript uses semicolons (;) to terminate statements. However, semicolons can be considered optional because JavaScript interpreters practice automatic semicolon insertion (ASI), which means they attempt to correct code that would be nonfunctional without semicolons by automatically inserting them as needed. As a result you can choose to write your JavaScript without using many (or even any) semicolons, and instead rely on ASI. Traditionally, it has been considered a best practice to explicitly use semicolons to terminate statements. However, with the advent of newer meta-scripting languages like CoffeeScript, many people now prefer to write terse code that employs a minimum of semicolons and instead relies on ASI as much as possible.

From a practical standpoint, either method is acceptable in that either method will help produce consistent, functional code. However, as with anything involving programming style, there have been many heated arguments recently about explicit semicolon use versus relying on ASI.

Relying on ASI

ASI follows a well-defined set of rules laid out in the ECMA-262 standard (Section 7.9 of Edition 5.1). If you would like to write JavaScript without semicolons, you are encouraged to review the standard so you know exactly what you are doing. We won’t cover the rules in detail here, but if you would like to rely on ASI, there are some important things to bear in mind.

Broadly, if the JavaScript engine encounters a new line (or a curly brace, though ASI is invoked mostly for new lines) that is used to break up tokens that otherwise don’t belong together, JavaScript will insert a semicolon—but only if it needs to do so in order to create syntactically valid code: code that the interpreter can successfully parse and execute. But the interpreter does not care if the code results in an error when it is executed. It only cares that the code can be executed.

To illustrate this, consider the two lines of JavaScript shown in Listing 2-1.

```javascript
myResult = argX - argY
myFunction()
```

If the interpreter were to encounter this code, it would determine that indeed a semicolon is needed to make this code functional, and it would insert one (Listing 2-2):

```javascript
myResult = argX - argY;
myFunction()
```

On the other hand, consider the two lines of code in Listing 2-3.

```javascript
myResult = argX - argY
[myResult].myProperty = "foo"
```

In this case, the interpreter would not insert a semicolon because, even though there is a new line, a semicolon isn’t needed to make the code functional. Instead, the interpreter would assume we meant what you see in Listing 2-4.