3

JavaScript Terminology

In this chapter, you learn all the beginner-level JavaScript terminology you need to get the most out of this book and also feel comfortable in moving forward with the language. I've always said that the key to solving a problem is first knowing how to identify it, and a big secret of great coders is that we know how to find a solution. Sometimes it's in your head; often times it isn't, and you have to look it up somewhere. Knowing what a problem is called or a general solution that you need to expand upon will help guide you to the answer. This is why getting to know the terms you will be coming across and what they mean is vital to problem solving with JavaScript.

We go over some terms and look at general code examples, so whether you're creating JavaScript from scratch or jumping into someone else's code, you'll be able to identify what you're looking at. As you go through this book, knowing what these terms mean will help you flow through some of the examples.

Basics

You'll need to know some basic terms in JavaScript, and this section will briefly go over those topics and display some code examples where applicable. These are terms you should have a strong understanding of before moving on to the next chapter because you will be encountering them quite a bit in the course of reading this book.

JavaScript is a very flexible (some call it unstructured) language, so many of these terms may be called something else in different circumstances (like a function vs. a method—you'll get to that). But knowing the basics will help you a lot.

Document Object Model (DOM)

At its simplest level, the DOM is an outline of the HTML document you are accessing. It functions in much the same way that an outline for any document would function—a book, an article, a grocery list, anything with structure. There are top-level items, items nested under them, and items grouped in chunks (like pages in a chapter and chapters in a book). These items are called **nodes** and every node in the DOM has a relationship to its surrounding nodes. These relationships, just like in a real-life family, are

- Parent
- Child
- Sibling

Parents

A parent node is anything that contains other nodes. Is that a little too vague? For example, a `<body>` element is the parent of anything contained within it; div tags, table elements, headings, everything nested inside counts as a child. The `<body>` element can certainly contain other parents, but to the `<body>`, they will always be children.

Listing 3.1 shows a simple unordered list and keeping with the family theme, they are all members of *my* family. In this example, the `<UL>` is the parent to all the list items because they are all contained within it.

Listing 3.1 **Parent Element in HTML**

```
<!-- the <ul> is the parent element to all the list items -->
<ul>
    <li>Joan</li>
    <li>Charlie</li>
    <li>Peter</li>
    <li>Christine</li>
    <li>Anna</li>
    <li>Tim</li>
</ul>
```

Children

In the DOM, children are positioned inside parent nodes. In Listing 3.1.1 you can see that the parent `<ul>` is wrapping all children (child nodes). It's like a parent giving all the children a big hug.

Listing 3.1.1 **Children Element in HTML**

```
<!-- the <li>s are all children (child elements) of the <ul> -->
<ul>
    <li>Joan</li>
    <li>Charlie</li>
    <li>Peter</li>
    <li>Christine</li>
    <li>Anna</li>
    <li>Tim</li>
</ul>
```

Siblings

Siblings, like in real life, are on the same level (without the inherent rivalry). If a parent has multiple children, they are referred to as **siblings**. In Listing 3.1.2 you can see that all the list items underneath the parent `<ul>` are on the same level, making them children of the `<ul>`, but also siblings to each other. They can be a child, parent, and sibling all at the same time if they have children.

Listing 3.1.2 **Sibling Element in HTML**

```
<!-- each <LI> is a sibling of the other list items, because they're on the same level -->
<ul>
    <li>Joan</li>
    <li>Charlie</li>
    <li>Peter</li>
    <li>Christine</li>
    <li>Anna</li>
    <li>Tim</li>
</ul>
```

Although all the people in Listing 3.1.2 may not be siblings in real life, they are in the eyes of the document object model.

Variables

Variables in JavaScript can hold values, objects, or even functions. They can change, they can be static, and they can even be empty. I guess you could say that their function in JavaScript is variable (pun intended). The basic function of a variable is to store some information. There are a couple types of variables that you may encounter as you're learning JavaScript: **local variables** and **global variables**.

Local Variables

Local variables are defined within a function and can be used only within that function (you will learn about functions in a bit). That means that you can't declare a local variable inside function A and use it in function B because they are contained inside the function in which they were defined. Local variables are always prefaced using "`var`" when they're defined. Listing 3.2 shows a list of basic variables defined. These are empty variables right now, but we'll be filling them up pretty soon.

Listing 3.2 **A List of Empty Variables**

```
var familyMember1;
var familyMember2;
var familyMember3;
var familyMember4;
var familyMember5;
var familyMember6;
```

When variables are empty like this, it's called **initializing** a variable, and it is necessary from time to time, as you will see later in this chapter when you get into loops.

Global Variables

Another kind of variable you may come across is a global variable. Global variables are well named because they can be used globally throughout your JavaScript. They can be defined three different ways:

- Defining them with "`var`" outside a function
- Adding something directly to the `window` object
- Defining them anywhere without using "`var`"

Something to consider when using global variables is to keep track of the names so you don't have any duplicates. This is especially important when using a large amount of JavaScript to prevent naming collisions. Listing 3.2.1 shows you how to set up global variables. Use them sparingly.

Listing 3.2.1 **An Empty Global Variable**

```
/* with var outside a function */
var titleOfApplication;

/* attached to the window object */
window.titleOfApplication

/* global from inside a function */
titleOfApplication;
```

Global variables like this are generally used for things you know for a fact will never change or something that needs to be filtered through an entire JavaScript file, like a directory URL or a special prefix you may be using. They can be handy on smaller projects but quickly become difficult to work with as an application grows larger, so be sure to use them sparingly.

Strings

A string is a group of miscellaneous characters, basically just a *blob of whatever*, typically saved to a variable. Not everything saved to a variable is a string. In Listing 3.2.2 you can see that we have started adding some content (strings) to the variables defined in Listing 3.2. The variables are starting to make more sense now that they contain string values.

Listing 3.2.2 **Saving Data to Variables**

```
var familyMember1 = "joan";
var familyMember2 = "charlie";
var familyMember3 = "peter";
var familyMember4 = "christine";
var familyMember5 = "anna";
var familyMember6 = "tim"; /* this is me */
```

Strings are always quoted. So, if you have a number saved to a variable, JavaScript will treat it as a string rather than a number. This is important to note if you are planning to do any math with JavaScript operators.

Comments

JavaScript comments are like any comment in another language, but you can use two syntaxes. One is intended for blocks of code and the other for single-line comments, but it's really a matter of personal preference; no one will judge you if you choose to use one comment style over another. Listing 3.2.3 shows the two styles of commenting in JavaScript.

Listing 3.2.3 **JavaScript Comments**

```
// this is a single-line comment

/*
 this is a multiple
 line comment if you need to be more descriptive
 or disable a large chunk of code
*/
```

It's important to use helpful comments in your code, not only so you can go back in at a later date and easily make updates, but they can also serve as notes for other developers who may be working in the same codebase as you. It's always said that good commenting is a sign of a good developer, so comment away! Be sure to make your comments meaningful and detailed but still brief; it can serve as the front-line type of documentation for your site or application.

Operators

Operators are the symbols in JavaScript you may recognize from math class. You can add, subtract, multiply, compare, and set values. The equal sign (=) in Listing 3.2.2 is an example of an operator setting a value. Table 3.1 shows some various operators you can use and their meaning.

Table 3.1 **JavaScript Operators**

Operator	Description
+	Add or concatenate
-	Subtract
*	Multiply
/	Divide
++	Increment (count up)
--	Decrement (count down, two minus signs)
=	Set value

In Listing 3.3, you can see variables and strings being added and concatenated with the + operator.

Listing 3.3 **Adding Strings and Numbers Together with Operators**

```
/* save my name to a variable using the = operator */
var myName = "tim";

/* adding a variable to a string */
alert(myName + ", this is me");

/* adding numbers together */
alert(100 + 50); // should alert 150

/* concatenating strings together */
alert("100" + "50" + ", adding strings together"); // should alert 10050
```

Note

If you are using operators for any math, the values must be numbers and not strings. You can add, subtract, multiply, divide, or combine (concatenate) numbers, but the only actions of strings are combine and compare. If you have a number that JavaScript is treating as a string, it needs to be converted to an actual number to be treated as such by JavaScript operators. But it is best to start with a number if you can.

Use Strict

The “use strict” statement is something you insert into function definitions to make sure the parser uses stricter rules when executing your script; it’s like using a strict doctype in the older days of (X)HTML. It’s thought of as best practice now in JavaScript to prevent you from writing lazy or sloppy code (no offense). At times it can be frustrating trying to catch hard-to-find errors, but it will help you in the long run write cleaner, more scalable code. We use this method throughout the book. Listing 3.4 shows a function we use later in the chapter with the “use strict” statement at the top.

Listing 3.4 **Setting “use strict” Mode**

```
function getFamilyMemberNames() {

    "use strict";

    /* the rest of your function code goes here */
}
```

Storage

Data storage is one of the main functions of JavaScript; since the early days the community has been on a quest for the Holy Grail of client-side storage. Because of this, there are a lot of different ways to store data in JavaScript. Some are specific methods that you will get into later on in the book, but there are some terms that will help you along the way.

Cache

Cache, in regard to JavaScript, doesn’t necessarily mean browser cache, although browser cache is vitally important. Caching in a JavaScript file usually refers to variables. When you declare a variable, it is cached, and you can reference it at any point. This is where you start seeing some performance implications. Using a variable over and over will perform better than redeclaring the same string over and over. You can define it once and continuously reference it.

Variables are great for organizing your code, but if you are using a string only once, it may be better for performance to not save it to a variable. This is part of the constant balancing act you will have to perform in JavaScript—between making something as high performance as possible versus making it as maintainable as possible. As far as performance goes, it’s pretty minor, but still something to consider.

Arrays

In a nutshell, arrays are lists. They can get very complicated when you get deeply into them, but on the surface they are no more than a list. Lists can be simple and straightforward or complicated and nested (multidimensional arrays). Arrays are one of the most flexible data storage formats in JavaScript, and they are very common in Ajax calls because data is often stored in an array format of some kind for easy JavaScript parsing. Listing 3.5 shows an array in its most basic form.

Listing 3.5 **Saving Data to an Array**

```
/* store family member names in an array */
var family = [
   "joan", /* numbering starts at "0" */
   "charlie",
   "peter",
   "christine",
   "anna",
   "tim" /* this is me! */
];
```

Cookies

Cookies are used on both the server and the client (different kinds of cookies—think oatmeal vs. chocolate chip; sure, they're both technically cookies but they're very different monsters). They allow us to store data locally in the user's browser to be accessed at a later time. This is starting to become a dated way for storing data, but it does have full support in all major browsers. It's currently a setting that can be turned off by the user, so it's best to not rely too much on this type of storage for critical issues.

JavaScript Object Notation (JSON)

JSON is another data format that can be easily integrated with JavaScript; it's often used with external services you are consuming within your JavaScript. Generally speaking, JSON will live in its own file and often on another server completely. It is currently the most common format for API services, and it was chosen because the human eye very easily reads it. It was originally thought to be an alternative to XML in data exchanges and quickly took over.

JSON is favored heavily over XML because it's very lightweight and can be accessed across domains for easy remote Ajax calls. Although it is native to JavaScript, it is platform independent and can be used with any technology on the client or server side to transfer data.

Listing 3.5.1 shows how the array defined in Listing 3.5 would look if it were converted to JSON format.

Listing 3.5.1 **Saving Data to JSON**

```
{
    "family" : [
        "joan",
        "charlie",
        "peter",
        "christine",
        "anna",
        "tim"
    ]
}
```

Objects

Rather than saying an object is a "thing" that's made up of other things—because that isn't very helpful—let's right off the bat compare it to my grandmother. In JavaScript, my grandmother would be considered an object (not in real-life, but definitely in JavaScript) and she has traits or properties. She has a first name, a last name, and a nickname, and those are also all objects of the parent object "`grandmother`."

You can see my grandmother depicted in JavaScript object terms in Listing 3.5.2. Notice the parent declaration of "`grandmother`" and the nested objects of "`first-name`", "`last-name`", and "`nickname`." Everything in JavaScript is an object, and when you start aligning your code with that assumption, a lot of the concepts become more clear and easier to consume.

Listing 3.5.2 **Saving Data to an Object**

```
/* store extra information about my grandmother in an object */
var grandmother = {
    "first-name": " anna",
    "last-name": "carroll",
    "nickname": "ma"
};
```

Anything can be stored in an object; it could be a string, like "`anna`", or even an entire function, like `getTheMail()`. That is the first step in creating object-oriented JavaScript and developing a code organization pattern.

Creating Interaction

Creating an interaction layer is what JavaScript does, and this section contains the tools of the trade for doing just that. This is where all the action is.

Loops

Loops in JavaScript exist to allow you to execute a block of code a certain number of times. The block of code can be inside a function or it can be a function by itself. In the **family** array we build, there are six items; executing a loop on that code means we want to go into each item (called **looping through**) and do something. Listing 3.5.3 illustrates how to set up and use a basic `for` loop.

Listing 3.5.3 **A Basic `for` Loop Parsing Through Each Item in the Family Array**

```
/* save the array of family members to a variable, and save the length in
➥peopleCount */
var people = family,
    peopleCount = items.length,
    i;

/* checking to make sure there are people in the list */
if(peopleCount > 0){

    /* loop through each person, since i is the total number this code will loop six
➥times */
    for(i = 0; i < peopleCount; i = i + 1){

        /* this represents 1 person */
        var person = people[i];

    }
}
```

Conditionals

Conditionals are used when you want to execute different code based on certain conditions. The use of conditionals also opens up the opportunity to compare values with even more types of operators, like "===", which looks for an exact value match. This is how you write code that can make decisions.

if Statement

An `if` statement is the most common of all the conditionals. In plain English, it's as if you were saying, "If this is going on, do something." The `if` statement has three types:

- A normal `if` statement
- An `if/else` statement, which has two conditions
- An `if/ else if/ else`, which has an endless number of conditions

Listing 3.5.4 shows an example of a basic `if/else` statement, which checks to see if the variable "person" is an exact match for "tim."

Listing 3.5.4 **A Basic if/else Statement Checking to See if a Person Is "tim"**

```
/* looking for an exact value match for the string "tim" */
if(person === "tim") {

    alert(person + ", this is me");

} else {

    alert(person);

}
```

When comparing numbers, you can also use the following operators:

- < (less than)
- > (greater than)
- <= (less than or equal to)
- >= (greater than or equal to)
- `!=` (not equal to)
- == (equal to)
- === (exactly equal to)
- `!==` (exactly not equal to)

switch Statement

A `switch` statement checks for a certain condition just as the `if` statement does. The difference is that after a condition (`case`) is met, the `switch` statement will stop, whereas the `if` statement will continue checking all the other conditions before moving on. `switch` statements are good for long conditional statements for that reason. Listing 3.5.5 shows you how to set up a `switch` statement that might iterate over an array of data.

Listing 3.5.5 **Example of a `switch` Statement Looping Through the Family Array**

```
switch (person) {

case "tim":
    alert("this is me");
    break;
case "christine":
    alert("my sister");
    break;

default:
    alert(person);

}
```

Functions

A function is a block of code that is executed through an event. The event can be a page load or a use-initiated event like `click`. You can also pass variables into functions via arguments to extend their functionality and make them even more flexible. Functions can also create or return variables that can later be passed into other functions.

Listing 3.5.6 shows the code snippet you have been working with throughout this chapter and converts it into a function you can execute at any time. This function, called `getFamilyMemberNames`, contains the "use strict" statement, the `array` of family members, the loop to run a block of code over each name, and the `if` statement checking to see if the person's name is an exact match for "tim."

Listing 3.5.6 **A Function That Will Take the Family Array, Loop, and Alert the Results**

```
function getFamilyMemberNames() {

    "use strict";

/* store family member names in an array */
var family = [
   "joan", /* numbering starts at "0" */
   "charlie",
   "peter",
   "christine",
   "anna",
   "tim" /* this is me! */
];

var peopleCount = family.length;
var i;

/* checking to make sure there are people in the list */
if(peopleCount > 0){

    for(i = 0; i < peopleCount; i = i + 1){
        var person = family[i];

        /* if the person is "tim", do something special */
        if(person === "tim") {
            alert(person + ", this is me!");
        } else {
            alert(person);
        }
    }
}
}

/* call the function */
getFamilyMemberNames();
```

Just like before, this code block should `alert` each family member's name and add the string "this is me!" to the end of "tim."

Anonymous Functions

Anonymous functions are functions that are declared as they are run, and they have no name assigned to them. Rather than writing a detailed function, like the one in Listing 3.5.6, you can use an anonymous function and have it execute immediately when it is run, instead of having it reference a function elsewhere in your document.

Anonymous functions perform better than a normally defined function because there is nothing to reference; they just execute when needed. These functions are used only once; they can't be referenced over and over. Making something an anonymous function will prevent the variable being used from slipping into the global scope (the variables are kept local to the function). In Listing 3.6, you can see that the previously defined function of `getFamilyMemberNames` has been converted to an anonymous function that is immediately executed on page load.

Listing 3.6 **Converting `getFamilyMemberNames` to an Anonymous Function**

```
(function () {

"use strict";

/* store family member names in an array */
var family = [
   "joan", /* numbering starts at "0" */
   "charlie",
   "peter",
   "christine",
   "anna",
   "tim" /* this is me! */
];

var peopleCount = family.length;
var i;

/* checking to make sure there are people in the list */
if(peopleCount > 0){

    for(i = 0; i < peopleCount; i = i + 1){

        var person = family[i];

        /* if the person is "tim", do something special */
        if(person === "tim"){
            alert(person + ", this is me!");
        } else {
            alert(person);
        }
    }
}
}
})();
```

Callback Functions

When a function is passed into another function as an argument, it's called a callback function. The function can be something you define, or it can be something native to JavaScript already. Functions can be very helpful in separating your logic and keeping your codebase as reusable as possible. Anonymous functions can also be callback functions. They're a little difficult to explain, so let's just jump right into Listing 3.7, which shows an example of a callback function as an anonymous function.

Listing 3.7 **An Example of a Function Calling a Function (callback)**

```
window.addEventListener("load", function (){

    alert("call back function");

    }, false });
```

The preceding callback function is attached to the native JavaScript method `addEventListener` and executes on the event "load." This brings us right into talking about one of the more interesting topics of JavaScript events.

Methods

First off, a method is, for all intents and purposes, a function. The difference between labeling something a function versus a method is about code organization. Earlier in the chapter, there is a section about **objects** in JavaScript, and I mentioned that anything could be saved into an object, including functions. When this happens, when a function is saved inside an object, it is referred to as a **method**. That's why things like the JavaScript method `alert()` that we have been using throughout this chapter is called a method even though it looks exactly like a function call.

Although JavaScript created the `alert()` method, you can also create methods, and it's totally contingent on how you choose to organize your code. In Chapter 9, "Code Organization," you will get into code organization in more depth, but for now it is important to note that when something is native to JavaScript or built into an external library, it's usually called a method. Listing 3.8 illustrates how you would define your own method.

Listing 3.8 **An Example of a JavaScript Method**

```
/* defining your object aka naming this group of functions */
var getInformation = {

    /* first method (function inside an object) is called "names" */
    " names": function () {

        "use strict";

        alert("get the names");
    },

    /* second method is called "checkForTim" */
    "checkForTim": function () {

        "use strict";

        alert("checking for tim");

    }
};
/* get the names on load */
window.addEventListener("load", getInformation.names, false);

/* check for Tim on click */
document.addEventListener("click", getInformation.checkForTim, false);
```

As you progress and become a JavaScript expert, you will probably catch yourself creating more and more methods. It is a very object-oriented approach to writing in any language and great for maintainability, although there are performance implications you will get into later on by nesting functions deep into other objects. This is a very popular way to make well-organized and reusable code.

Events

Events are how you elicit feedback from the user. If you want to execute any JavaScript on a page, it must happen through an event of some kind. Loading of the page, clicking a link, and submitting a form are all considered events, and you can attach functions to them all. There are a lot of events you can attach functions to in JavaScript, including

- `click`
- `dblclick`
- `mousedown`
- `mousemove`
- `mouseout`
- `mouseover`
- `mouseup`
- `keydown`
- `keypress`
- `keyup`
- `blur`
- `focus`
- `submit`
- `load`
- `touchstart` *
- `touchmove` *
- `touchcancel` *
- `orientationchange` *
- `gesturestart` *
- `gestureend` *
- `gesturechange` *

* New event specialized for touch interactions.

Listing 3.9 shows how you would execute the `getFamilyMemberNames` function upon loading of a page and then again upon clicking anywhere in the document.

Listing 3.9 **Load and Click Events to Execute the `getFamilyMemberNames` Function**

```
/* execute the function on load of the window*/
window.addEventListener("load", getFamilyMemberNames, false);

/* execute the function on clicking the document */
document.addEventListener("click", getFamilyMemberNames, false);
```

Ajax

Ajax is the concept of refreshing a part of an HTML document without reloading the entire page. It is extremely misused, but I assure you that you will learn the proper ways to use this technology as you read this book (Chapter 8, “Communicating with the Server Through Ajax,” is all about Ajax and server communications).

Ajax is not an acronym; it does not stand for anything. Many think it stands for Asynchronous JavaScript and XML, but it doesn't; it's just a word. Ajax can be synchronous, and it doesn't have to be XML. It can be XML, JSON, or even HTML. It's all about the data you want to pass to and from the server, in almost any form you want.

Summary

In this chapter you learned the terminology that all JavaScript developers need to be familiar with to succeed. As you go through this book we will be expanding upon each of these topics in a lot more detail. We will be going over in-depth examples, real-life scenarios, and learning about situational JavaScript (when to use what thing).

We covered everything from the basics, such as the DOM, variables, and strings, to definitions of the various types of data storage options in JavaScript (there are a lot more, but that's for a later chapter), and we even delved into the more advanced topics like conditionals, loops, and functions.

If you found this chapter a little overwhelming, don't worry; we'll be breaking down each of the topics individually throughout the book and building on each one. Right now it is important to know the terms, be able to recognize them as code samples, and know their general meanings. This will aid in having a more fluid conversation about JavaScript. We can now talk about functions and loops because you know what they are.

Exercises

1. What is an anonymous function?
2. When is a function considered to be a callback function?
3. How do you elicit feedback from a user?