Section 16. JavaScript Functions, Objects, and Arrays

Just like PHP, JavaScript offers access to functions and objects. In fact, JavaScript is actually based on objects, because—as you've seen—it has to access the DOM, which makes every element of an HTML document available to manipulate as an object.

The usage and syntax are also quite similar to those of PHP, so you should feel right at home as I take you through using functions and objects in JavaScript, as well as through an in-depth exploration of array handling.

JavaScript Functions

In addition to having access to dozens of built-in functions (or methods), such as `write`, which you have already seen being used in `document.write`, you can easily create your own functions. Whenever you have a relatively complex piece of code that is likely to be reused, you have a candidate for a function.

Defining a Function

The general syntax for a function is shown here:

```
function function_name([parameter [, ...]])
{
  statements
}
```

The first line of the syntax indicates the following:

- A definition starts with the word `function`.
- A name follows that must start with a letter or underscore, followed by any number of letters, digits, dollar signs, or underscores.
- The parentheses are required.
- One or more parameters, separated by commas, are optional (indicated by the square brackets, which are not part of the function syntax).

Function names are case-sensitive, so all of the following strings refer to different functions: `getInput`, `GETINPUT`, and `getinput`.

In JavaScript there is a general naming convention for functions: the first letter of each word in a name is capitalized, except for the very first letter, which is lowercase. Therefore, of the previous

examples, `getInput` would be the preferred name used by most programmers. This convention is commonly referred to as *bumpyCaps*, *bumpyCase*, or (most frequently) *camelCase*.

The opening curly brace starts the statements that will execute when you call the function; a matching curly brace must close it. These statements may include one or more `return` statements, which force the function to cease execution and return to the calling code. If a value is attached to the `return` statement, the calling code can retrieve it.

The `arguments` array

The `arguments` array is a member of every function. With it, you can determine the number of variables passed to a function and what they are. Take the example of a function called `displayItems`. Example 16-1 shows one way of writing it.

Example 16-1. Defining a function

```
<script>
  displayItems("Dog", "Cat", "Pony", "Hamster", "Tortoise")

  function displayItems(v1, v2, v3, v4, v5)
  {
    document.write(v1 + "<br>")
    document.write(v2 + "<br>")
    document.write(v3 + "<br>")
    document.write(v4 + "<br>")
    document.write(v5 + "<br>")
  }
</script>
```

When you call up this script in your browser, it will display the following:

```
Dog
Cat
Pony
Hamster
Tortoise
```

All of this is fine, but what if you wanted to pass more than five items to the function? Also, reusing the `document.write` call multiple times instead of employing a loop is wasteful programming. Luckily, the `arguments` array gives you the flexibility to handle a variable number of arguments. Example 16-2 shows how you can use it to rewrite the previous example in a much more efficient manner.

Example 16-2. Modifying the function to use the `arguments` array

```
<script>
  let c = "Car"

  displayItems("Bananas", 32.3, c)

  function displayItems()
  {
    for (j = 0 ; j < displayItems.arguments.length ; ++j)
      document.write(displayItems.arguments[j] + "<br>")
  }
</script>
```

Note the use of the `length` property, which you already encountered in the previous section, and also that I reference the array `displayItems.arguments` using the variable `j` as an offset into it. I also chose to keep the function short and sweet by not surrounding the contents of the `for` loop in curly braces, as it contains only a single statement. Remember that the loop must stop when `j` is one less than `length`, not equal to `length`.

Using this technique, you now have a function that can take as many (or as few) arguments as you like and act on each argument as you desire.

Returning a Value

Functions are not used just to display things. In fact, they are mostly used to perform calculations or data manipulations and then return a result. The function `fixNames` in Example 16-3 uses the `arguments` array (discussed in the previous section) to take a series of strings passed to it and return them as a single string. The "fix" it performs is to convert every character in the arguments to lowercase except for the first character of each argument, which is set to a capital letter.

Example 16-3. Cleaning up a full name

```
<script>
  document.write(fixNames("the", "DALLAS", "CowBoys"))

  function fixNames()
  {
    var s = ""

    for (j = 0 ; j < fixNames.arguments.length ; ++j)
      s += fixNames.arguments[j].charAt(0).toUpperCase() +
           fixNames.arguments[j].substr(1).toLowerCase() + " "

    return s.substr(0, s.length-1)
  }
</script>
```

When called with the parameters `the`, `DALLAS`, and `CowBoys`, for example, the function returns the string `The Dallas Cowboys`. Let's walk through the function.

It first initializes the temporary (and local) variable `s` to the empty string. Then a `for` loop iterates through each of the passed parameters, isolating the parameter's first character using the `charAt` method and converting it to uppercase with the `toUpperCase` method. The various methods shown in this example are all built into JavaScript and available by default.

Then the `substr` method is used to fetch the rest of each string, which is converted to lowercase via the `toLowerCase` method. A fuller version of the `substr` method here would specify how many characters are part of the substring as a second argument:

```
substr(1, (arguments[j].length) - 1 )
```

In other words, this `substr` method says, "Start with the character at position 1 (the second character) and return the rest of the string (the length minus one)." As a nice touch, though, the `substr` method assumes that you want the rest of the string if you omit the second argument.

After the whole argument is converted to our desired case, a space character is added to the end, and the result is appended to the temporary variable `s`.

Finally, the `substr` method is used again to return the contents of the variable `s`, except for the final space—which is unwanted. We remove this by using `substr` to return the string up to, but not including, the final character.

This example is particularly interesting in that it illustrates the use of multiple properties and methods in a single expression, for example:

```
fixNames.arguments[j].substr(1).toLowerCase()
```

You have to interpret the statement by mentally dividing it into parts at the periods. JavaScript evaluates these elements of the statement from left to right as follows:

1. Start with the name of the function itself: `fixNames`.

2. Extract element `j` from the array `arguments` representing `fixNames` arguments.

3. Invoke `substr` with a parameter of `1` to the extracted element. This passes all but the first character to the next section of the expression.

4. Apply the method `toLowerCase` to the string that has been passed thus far.

This practice is often referred to as *method chaining*. So, for example, if the string `mixedCASE` is passed to the example expression, it will go through the following transformations:

```
mixedCASE
ixedCASE
ixedcase
```

In other words, `fixNames.arguments[j]` produces "mixedCASE", then `substr(1)` takes "mixedCASE" and produces "ixedCASE", and finally `toLowerCase()` takes "ixedCASE" and produces "ixedcase".

One final reminder: the `s` variable created inside the function is local and therefore cannot be accessed outside the function. By returning `s` in the `return` statement, we made its value available to the caller, which could store or use it any way it wanted. But `s` itself disappears at the end of the

function. Although we could make a function operate on global variables (and sometimes that's necessary), it's much better to just return the values you want to preserve and let JavaScript clean up all the other variables used by the function.

Returning an Array

In Example 16-3, the function returned only one parameter—but what if you need to return multiple parameters? You can do this by returning an array, as in Example 16-4.

Example 16-4. Returning an array of values

```
<script>
  words = fixNames("the", "DALLAS", "CowBoys")

  for (j = 0 ; j < words.length ; ++j)
    document.write(words[j] + "<br>")

  function fixNames()
  {
    var s = new Array()

    for (j = 0 ; j < fixNames.arguments.length ; ++j)
      s[j] = fixNames.arguments[j].charAt(0).toUpperCase() +
             fixNames.arguments[j].substr(1).toLowerCase()

    return s
  }
</script>
```

Here the variable `words` is automatically defined as an array and populated with the returned result of a call to the function `fixNames`. Then a `for` loop iterates through the array and displays each member.

As for the `fixNames` function, it's almost identical to Example 16-3, except that the variable `s` is now an array; after each word has been processed, it is stored as an element of this array, which is returned by the `return` statement.

This function enables the extraction of individual parameters from its returned values, like the following (the output from which is simply `The Cowboys`):

```
words = fixNames("the", "DALLAS", "CowBoys")
document.write(words[0] + " " + words[2])
```

JavaScript Objects

A JavaScript object is a step up from a variable, which can contain only one value at a time. In contrast, objects can contain multiple values and even functions. An object groups data together with the functions needed to manipulate it.

Declaring a Class

When creating a script to use objects, you need to design a composite of data and code called a *class*. Each new object based on this class is called an *instance* (or *occurrence*) of that class. As you've already seen, the data associated with an object is called its *properties*, while the functions it uses are called *methods*.

Let's look at how to declare the class for an object called `User` that will contain details about the current user. To create the class, just write a function named after the class. This function can accept arguments (I'll show later how it's invoked) and can create properties and methods for objects in that class. The function is called a *constructor*.

Example 16-5 shows a constructor for the class `User` with three properties: `forename`, `username`, and `password`. The class also defines the method `showUser`.

Example 16-5. Declaring the `User` class and its method

```
<script>
  function User(forename, username, password)
  {
    this.forename = forename
    this.username = username
    this.password = password

    this.showUser = function()
    {
      document.write("Forename: " + this.forename + "<br>")
      document.write("Username: " + this.username + "<br>")
      document.write("Password: " + this.password + "<br>")
    }
  }
</script>
```

The function is different from other functions we've seen so far in several ways:

- Each time the function is called, it creates a new object. Thus, you can call the same function over and over with different arguments to create users with different forenames, for example.
- The function refers to an object named `this`, which refers to the instance being created. As the example shows, the object uses the name `this` to set its own properties, which will be different from one `User` to another.
- A new function named `showUser` is created within the function. The syntax shown here is new and rather complicated, but its purpose is to tie `showUser` to the `User` class. Thus, `showUser`

comes into being as a method of the `User` class.

The naming convention I have used is to keep all properties in lowercase and to use at least one uppercase character in method names, following the camelCase convention mentioned earlier in the section.

Example 16-5 follows the recommended way to write a class constructor, which is to include methods in the constructor function. However, you can also refer to functions defined outside the constructor, as in Example 16-6.

Example 16-6. Separately defining a class and method

```
<script>
  function User(forename, username, password)
  {
    this.forename = forename
    this.username = username
    this.password = password
    this.showUser = showUser
  }

  function showUser()
  {
    document.write("Forename: " + this.forename + "<br>")
    document.write("Username: " + this.username + "<br>")
    document.write("Password: " + this.password + "<br>")
  }
</script>
```

I show you this form because you are certain to encounter it when perusing other programmers' code.

Creating an Object

To create an instance of the class `User`, you can use a statement such as the following:

```
details = new User("Wolfgang", "w.a.mozart", "composer")
```

Or you can create an empty object, like this:

```
details = new User()
```

and then populate it later, like this:

```
details.forename = "Wolfgang"
details.username = "w.a.mozart"
details.password = "composer"
```

You can also add new properties to an object, like this:

```
details.greeting = "Hello"
```

You can verify that adding such new properties works with the following statement:

```
document.write(details.greeting)
```

Accessing Objects

To access an object, you can refer to its properties, as in the following two unrelated example statements:

```
name = details.forename
if (details.username == "Admin") loginAsAdmin()
```

So, to access the `showUser` method of an object of class `User`, you would use the following syntax, in which the object `details` has already been created and populated with data:

```
details.showUser()
```

Assuming the data supplied earlier, this code would display the following:

```
Forename: Wolfgang
Username: w.a.mozart
Password: composer
```

The prototype Keyword

The `prototype` keyword can save you a lot of memory. In the `User` class, every instance will contain the three properties and the method. Therefore, if you have one thousand of these objects in memory, the method `showUser` will also be replicated one thousand times. However, because the method is identical in every case, you can specify that new objects should refer to a single instance of the method instead of creating a copy of it. So, instead of using the following in a class constructor:

```
this.showUser = function()
```

you could replace it with this:

```
User.prototype.showUser = function()
```

Example 16-7 shows what the new constructor would look like.

Example 16-7. Declaring a class using the `prototype` keyword for a method

```
<script>
  function User(forename, username, password)
  {
    this.forename = forename
```

```
    this.username = username
    this.password = password

    User.prototype.showUser = function()
    {
      document.write("Forename: " + this.forename + "<br>")
      document.write("Username: " + this.username + "<br>")
      document.write("Password: " + this.password + "<br>")
    }
  }
</script>
```

This works because all functions have a `prototype` property, designed to hold properties and methods that are not replicated in any objects created from a class. Instead, they are passed to its objects by reference.

This means that you can add a `prototype` property or method at any time and all objects (even those already created) will inherit it, as the following statements illustrate:

```
User.prototype.greeting = "Hello"
document.write(details.greeting)
```

The first statement adds the `prototype` property of `greeting` with a value of `Hello` to the class `User`. In the second line, the object `details`, which has already been created, correctly displays this new property.

You can also add to or modify methods in a class, as the following statements illustrate:

```
User.prototype.showUser = function()
{
  document.write("Name "  + this.forename +
                 " User " + this.username +
                 " Pass " + this.password)
}

details.showUser()
```

You might add these lines to your script in a conditional statement (such as `if`), so they run if user activities cause you to decide you need a different `showUser` method. After these lines run, even if the object `details` has been created already, further calls to `details.showUser` will run the new function. The old definition of `showUser` has been erased.

Static methods and properties

When reading about PHP objects, you learned that classes can have static properties and methods as well as properties and methods associated with a particular instance of a class. JavaScript also supports static properties and methods, which you can conveniently store and retrieve from the class's `prototype`. Thus, the following statements set and read a static string from `User`:

```
User.prototype.greeting = "Hello"
document.write(User.prototype.greeting)
```

Extending JavaScript objects

The `prototype` keyword even lets you add functionality to a built-in object. For example, suppose that you would like to add the ability to replace all spaces in a string with nonbreaking spaces in order to prevent it from wrapping around. You can do this by adding a prototype method to JavaScript's default `String` object definition, like this:

```
String.prototype.nbsp = function()
{
  return this.replace(/ /g, ' ')
}
```

Here the `replace` method is used with a regular expression to find and replace all single spaces with the string ` `.

Note

If you are not already familiar with regular expressions, they are a handy means of extracting information from or manipulating strings and are fully explained in Section 17. Suffice it to say that for now, you can copy and paste the preceding examples and they will work as described, illustrating the power of extending JavaScript `String` *objects.*

If you then enter the following command:

```
document.write("The quick brown fox".nbsp())
```

it will output the string `The quick brown fox`. Or here's a method you can add that will trim leading and trailing spaces from a string (once again using a regular expression):

```
String.prototype.trim = function()
{
  return this.replace(/^\s+|\s+$/g, '')
}
```

If you issue the following statement, the output will be the string `Please trim me` (with the leading and trailing spaces removed):

```
document.write("  Please trim me    ".trim())
```

If we break down the expression into its component parts, the two `/` characters mark the start and end of the expression, and the final `g` specifies a global search. Inside the expression, the `^\s+` part searches for one or more whitespace characters appearing at the start of the search string, while the `\s+$` part searches for one or more whitespace characters at the end of the search string. The `|` character in the middle acts to separate the alternatives.

The result is that when either of these expressions matches, the match is replaced with the empty string, thus returning a trimmed version of the string without any leading or trailing whitespace.

Warning

There is debate about whether extending objects is good or bad practice. Some programmers say that should an object later be extended to officially offer the functionality you have added, it could be implemented another way, or do something quite different to your extension, which could then cause a conflict. However, other programmers, such as the inventor of JavaScript, Brendan Eich, say that this is a perfectly acceptable practice. My take is to agree with the latter but in production code to choose extension names that are most unlikely to ever be officially used. So, for example, the `trim` *extension could be renamed as* `mytrim`*, and the supporting code might more safely be written as the following:*

```
String.prototype.mytrim = function()
{
  return this.replace(/^\s+|\s+$/g, '')
}
```

JavaScript Arrays

Array handling in JavaScript is very similar to PHP, although the syntax is a little different. Nevertheless, given all you have already learned about arrays, this section should be relatively straightforward for you.

Numeric Arrays

To create a new array, use the following syntax:

```
arrayname = new Array()
```

Or you can use the shorthand form, as follows:

```
arrayname = []
```

Assigning element values

In PHP, you could add a new element to an array by simply assigning it without specifying the element offset, like this:

```
$arrayname[] = "Element 1";
$arrayname[] = "Element 2";
```

But in JavaScript you use the `push` method to achieve the same thing, like this:

```
arrayname.push("Element 1")
arrayname.push("Element 2")
```

This allows you to keep adding items to an array without having to keep track of the number of items. When you need to know how many elements are in an array, you can use the `length` property, like this:

```
document.write(arrayname.length)
```

Alternatively, if you wish to keep track of the element locations yourself and place them in specific locations, you can use syntax such as this:

```
arrayname[0] = "Element 1"
arrayname[1] = "Element 2"
```

Example 16-8 shows a simple script that creates an array, loads it with some values, and then displays them.

Example 16-8. Creating, building, and printing an array

```
<script>
  numbers = []
  numbers.push("One")
  numbers.push("Two")
  numbers.push("Three")

  for (j = 0 ; j < numbers.length ; ++j)
    document.write("Element " + j + " = " + numbers[j] + "<br>")
</script>
```

The output from this script is as follows:

```
Element 0 = One
Element 1 = Two
Element 2 = Three
```

Assignment using the Array keyword

You can also create an array together with some initial elements by using the `Array` keyword, like this:

```
numbers = Array("One", "Two", "Three")
```

There is nothing stopping you from adding more elements afterward as well.

You've now seen a couple of ways you can add items to an array, and one way of referencing them. JavaScript offers many more, which I'll get to shortly—but first, we'll look at another type of array.

Associative Arrays

An *associative array* is one in which the elements are referenced by name rather than by an integer offset. However, JavaScript doesn't support such things. Instead, we can achieve the same result by creating an object with properties that will act the same way.

So, to create an "associative array," define a block of elements within curly braces. For each element, place the key on the left and the contents on the right of a colon (`:`). Example 16-9 shows how you might create an associative array to hold the contents of the "balls" section of an online sports equipment retailer.

Example 16-9. Creating and displaying an associative array

```
<script>
  balls = {"golf":    "Golf balls, 6",
           "tennis":  "Tennis balls, 3",
           "soccer":  "Soccer ball, 1",
           "ping":    "Ping Pong balls, 1 doz"}

  for (ball in balls)
    document.write(ball + " = " + balls[ball] + "<br>")
</script>
```

To verify that the array has been correctly created and populated, I have used another kind of `for` loop using the `in` keyword. This creates a new variable to use only within the array (`ball`, in this example) and iterates through all elements of the array to the right of the `in` keyword (`balls`, in this example). The loop acts on each element of `balls`, placing the key value into `ball`.

Using this key value stored in `ball`, you can also get the value of the current element of `balls`. The result of calling up the example script in a browser is as follows:

```
golf = Golf balls, 6
tennis = Tennis balls, 3
soccer = Soccer ball, 1
ping = Ping Pong balls, 1 doz
```

To get a specific element of an associative array, you can specify a key explicitly, in the following manner (in this case, outputting the value `Soccer ball, 1`):

```
document.write(balls['soccer'])
```

Multidimensional Arrays

To create a multidimensional array in JavaScript, just place arrays inside other arrays. For example, to create an array to hold the details of a two-dimensional checkerboard (8 × 8 squares), you could use the code in Example 16-10.

Example 16-10. Creating a multidimensional numeric array

```
<script>
  checkerboard = Array(
    Array(' ', 'o', ' ', 'o', ' ', 'o', ' ', 'o'),
    Array('o', ' ', 'o', ' ', 'o', ' ', 'o', ' '),
    Array(' ', 'o', ' ', 'o', ' ', 'o', ' ', 'o'),
    Array(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '),
    Array(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '),
    Array('O', ' ', 'O', ' ', 'O', ' ', 'O', ' '),
    Array(' ', 'O', ' ', 'O', ' ', 'O', ' ', 'O'),
    Array('O', ' ', 'O', ' ', 'O', ' ', 'O', ' '))

  document.write("<pre>")

  for (j = 0 ; j < 8 ; ++j)
  {
    for (k = 0 ; k < 8 ; ++k)
      document.write(checkerboard[j][k] + " ")

    document.write("<br>")
  }

  document.write("</pre>")
</script>
```

In this example, the lowercase letters represent black pieces, and the uppercase white. A pair of nested `for` loops walks through the array and displays its contents.

The outer loop contains two statements, so curly braces enclose them. The inner loop then processes each square in a row, outputting the character at location `[j][k]`, followed by a space (to square up the printout). This loop contains a single statement, so curly braces are not required to enclose it. The `<pre>` and `</pre>` tags ensure that the output displays correctly, like this:

```
  o   o   o   o
o   o   o   o
  o   o   o   o


O   O   O   O
  O   O   O   O
O   O   O   O
```

You can also directly access any element within this array by using square brackets:

```
document.write(checkerboard[7][2])
```

This statement outputs the uppercase letter `O`, the eighth element down and the third along—remember that array indexes start at 0, not 1.

Using Array Methods

Given the power of arrays, JavaScript comes ready-made with a number of methods for manipulating them and their data. Here is a selection of the most useful ones.

some

When you need to know whether at least one array element matches a certain criterion, you can use the `some` function, which will test all the elements and automatically stop and return the required value as soon as one matches. This saves you from having to write your own code to perform such searches, like this:

```
function isBiggerThan10(element, index, array)
{
  return element > 10
}

result = [2, 5, 8, 1, 4].some(isBiggerThan10); // result will be false
result = [12, 5, 8, 1, 4].some(isBiggerThan10); // result will be true
```

indexOf

To find out where an element can be found in an array, you can call the `indexOf` function on the array, which will return the offset of the located element (starting from 0), or `-1` if it is not found. For example, the following gives `offset` the value `2`:

```
animals = ['cat', 'dog', 'cow', 'horse', 'elephant']
offset = animals.indexOf('cow')
```

concat

The `concat` method concatenates two arrays, or a series of values within an array. For example, the following code outputs `Banana,Grape,Carrot,Cabbage`:

```
fruit = ["Banana", "Grape"]
veg   = ["Carrot", "Cabbage"]

document.write(fruit.concat(veg))
```

You can specify multiple arrays as arguments, in which case `concat` adds all their elements in the order that the arrays are specified.

Here's another way to use `concat`. This time, plain values are concatenated with the array `pets`, which outputs `Cat,Dog,Fish,Rabbit,Hamster`:

```
pets      = ["Cat", "Dog", "Fish"]
more_pets = pets.concat("Rabbit", "Hamster")

document.write(more_pets)
```

forEach

The `forEach` method in JavaScript is another way of achieving functionality similar to the PHP `foreach` keyword. To use it, you pass it the name of a function, which will be called for each element within the array. Example 16-11 shows how.

Example 16-11. Using the `forEach` method

```
<script>
  pets = ["Cat", "Dog", "Rabbit", "Hamster"]
  pets.forEach(output)

  function output(element, index, array)
  {
    document.write("Element at index " + index + " has the value " +
      element + "<br>")
  }
</script>
```

In this case, the function passed to `forEach` is called `output`. It takes three parameters: the `element`, its `index`, and the `array`. These can be used as required by your function. This example displays just the `element` and `index` values using the function `document.write`.

Once an array has been populated, the method is called like this:

```
pets.forEach(output)
```

This is the output:

```
Element at index 0 has the value Cat
Element at index 1 has the value Dog
Element at index 2 has the value Rabbit
Element at index 3 has the value Hamster
```

join

With the `join` method, you can convert all the values in an array to strings and then join them together into one large string, placing an optional separator between them. Example 16-12 shows three ways of using this method.

Example 16-12. Using the `join` method

```
<script>
  pets = ["Cat", "Dog", "Rabbit", "Hamster"]

  document.write(pets.join()      + "<br>")
  document.write(pets.join(' ')   + "<br>")
  document.write(pets.join(' : ') + "<br>")
</script>
```

Without a parameter, `join` uses a comma to separate the elements; otherwise, the string passed to `join` is inserted between each element. The output of Example 16-12 looks like this:

```
Cat,Dog,Rabbit,Hamster
Cat Dog Rabbit Hamster
Cat : Dog : Rabbit : Hamster
```

push and pop

You already saw how the `push` method can be used to insert a value into an array. The inverse method is `pop`. It deletes the most recently inserted element from an array and returns it. Example 16-13 shows an example of its use.

Example 16-13. Using the `push` and `pop` methods

```
<script>
  sports = ["Football", "Tennis", "Baseball"]
  document.write("Start = "      + sports +  "<br>")

  sports.push("Hockey")
  document.write("After Push = " + sports +  "<br>")

  removed = sports.pop()
  document.write("After Pop = "  + sports +  "<br>")
  document.write("Removed = "    + removed + "<br>")
</script>
```

The three main statements of this script are shown in bold type. First, the script creates an array called `sports` with three elements and then `push`es a fourth element into the array. After that, it `pop`s that element back off. In the process, the various current values are displayed via `document.write`. The script outputs the following:

```
Start = Football,Tennis,Baseball
After Push = Football,Tennis,Baseball,Hockey
After Pop = Football,Tennis,Baseball
Removed = Hockey
```

The `push` and `pop` functions are useful in situations where you need to divert from some activity to do another, and then return. For example, let's suppose you want to put off some activities until later, while you get on with something more important now. This often happens in real life when we're going through "to-do" lists, so let's emulate that in code, with tasks number 2 and 5 in a list of six items being granted priority status, as in Example 16-14.

Example 16-14. Using `push` and `pop` inside and outside of a loop

```
<script>
  numbers = []

  for (j=1 ; j<6 ; ++j)
  {
    if (j == 2 || j == 5)
    {
      document.write("Processing 'todo' #" + j + "<br>")
    }
    else
    {
      document.write("Putting off 'todo' #" + j + " until later<br>")
      numbers.push(j)
    }
  }

  document.write("<br>Finished processing the priority tasks.")
  document.write("<br>Commencing stored tasks, most recent first.<br><br>")

  document.write("Now processing 'todo' #" + numbers.pop() + "<br>")
  document.write("Now processing 'todo' #" + numbers.pop() + "<br>")
  document.write("Now processing 'todo' #" + numbers.pop() + "<br>")
</script>
```

Of course, nothing is actually getting processed here, just text being output to the browser, but you get the idea. The output from this example is as follows:

```
Putting off 'todo' #1 until later
Processing 'todo' #2
Putting off 'todo' #3 until later
Putting off 'todo' #4 until later
Processing 'todo' #5

Finished processing the priority tasks.
Commencing stored tasks, most recent first.

Now processing 'todo' #4
Now processing 'todo' #3
Now processing 'todo' #1
```

Using reverse

The `reverse` method simply reverses the order of all elements in an array. Example 16-15 shows this in action.

Example 16-15. Using the `reverse` method

```
<script>
  sports = ["Football", "Tennis", "Baseball", "Hockey"]
  sports.reverse()
  document.write(sports)
</script>
```

The original array is modified, and the output from this script is as follows:

```
Hockey,Baseball,Tennis,Football
```

sort

With the `sort` method, you can place all the elements of an array in alphabetical order, depending on the parameters used. Example 16-16 shows four types of sort.

Example 16-16. Using the `sort` method

```
<script>
  // Alphabetical sort
  sports = ["Football", "Tennis", "Baseball", "Hockey"]
  sports.sort()
  document.write(sports + "<br>")

  // Reverse alphabetical sort
  sports = ["Football", "Tennis", "Baseball", "Hockey"]
  sports.sort().reverse()
  document.write(sports + "<br>")

  // Ascending numeric sort
  numbers = [7, 23, 6, 74]
  numbers.sort(function(a,b){return a - b})
  document.write(numbers + "<br>")

  // Descending numeric sort
  numbers = [7, 23, 6, 74]
  numbers.sort(function(a,b){return b - a})
```

```
  document.write(numbers + "<br>")
</script>
```

The first of the four example sections uses the default `sort` method to perform an *alphabetical sort*, while the second uses the default `sort` and then applies the `reverse` method to get a *reverse alphabetical sort*.

The third and fourth sections are a little more complicated; they use a function to compare the relationships between `a` and `b`. The function doesn't have a name, because it's used only in the sort. You have already seen the function named `function` used to create an anonymous function; we used it to define a method in a class (the `showUser` method).

Here, `function` creates an anonymous function meeting the needs of the `sort` method. If the function returns a value greater than zero, the sort assumes that `b` comes before `a`. If the function returns a value less than zero, the sort assumes that `a` comes before `b`. The sort runs this function across all the values in the array to determine their order. (Of course, if `a` and `b` have the same value, the function returns zero and it doesn't matter which value is first.)

By manipulating the value returned (`a - b` in contrast to `b - a`), the third and fourth sections of Example 16-16 choose between an *ascending numerical sort* and a *descending numerical sort*.

And, believe it or not, this represents the end of your introduction to JavaScript. You should now have a core knowledge of the three main technologies covered in this module. The next section will look at some advanced techniques used across these technologies, such as pattern matching and input validation.