

Introduction

Welcome Back!

In today's lesson, you're going to increase your programming prowess by learning about JavaScript arrays, loops, and new decision-making capabilities. At first blush, these new concepts probably won't seem to relate to anything from real life, and you may be wondering how in the heck you'd ever use such things in Web pages or apps. However, they're the kinds of programming constructs the pros use all the time, in all programming languages.

In fact, every time you open some app or program on your PC, laptop, tablet, or smartphone, there's a good chance that there are some arrays, loops, and advanced decision-making things going on in the background to make the app do what it does.

So please put on your thinking caps, and prepare to enter the strange new world of JavaScript arrays.

Fun with Arrays and Loops



Download

Below is the link to the ZIP file which contains the resources you'll need for this lesson:

- **Color Array:** This webpage displays an array of colors by name:
- **For Loop:** This webpage will be used to experiment with for loops:
- **Do While Loop:** This webpage will be used to explore do while loops:
- **Switch Demo:** This webpage will be used to explore switch statements:



[Download Lesson-08_ClassProject.zip](#)

1.5 KB.(Gigabytes) ZIP

You're welcome to type this page from scratch, if you're so inclined. Or, if you're not in the mood to practice typing right now, feel free to simply save it to your *Intro JavaScript* folder.

Chapter 1: Creating Arrays

Understanding Arrays

There are times in all forms of programming when you want the code to work with lists of items. Consider a slideshow, which consists of multiple images. Each image has a filename. With JavaScript, the easiest way to work with a group of filenames is to put them in a list. That way, you can work with them based on their position in the list (the first item in the list, the second item in the list, and so forth). Positions in a list can be expressed in numbers, and computers are great with numbers.



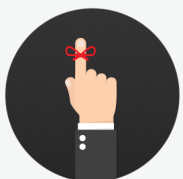
Array as a List

Those of you with HTML or publishing background might immediately envision a numbered or bulleted list that's visible on the page, but that's not what we're talking about here. Instead, we're talking about a list of items that's visible only to the programming language (not to people) and can be accessed and manipulated by the programming language. We typically refer to that kind of list as an *array* to help distinguish it from the more familiar visible lists on a page.

The Array Object

JavaScript includes an *array object*, which is basically a type of object with properties and methods to help you work with data stored in arrays (lists) in computer memory. To create a simple array, you use this syntax:

```
var name = new Array()
```



Remember

As always, replace *name* with a variable name of your own choosing. Like all names in JavaScript, it must start with a letter and cannot contain spaces or special characters. Also, notice that *Array()* starts with an uppercase *A*, which is typical of object names in JavaScript.

That one line of code doesn't actually put anything into the array. It just tells JavaScript that you intend to create an array. To put items into the array, you use the assignment operator(=) like with regular variables. Since the items in an array all have the same name, you distinguish between them by giving each a number in square brackets.

```
var color = new Array();

color[0]="Red";

color[1]="Green";

color[2]="Blue";
```

The number you assign to each item in the list is called a *subscript*. By default, JavaScript arrays are *zero-based*. This means the first item is number zero, not number one. There's no limit to the number of items you can put into an array.



Know the Lingo

When speaking to each other about arrays, programmers use the word *sub* to refer to subscript values. For example, `colors[0]` would be spoken as *color sub zero*, and `color[1]` would be spoken as *color sub one*.

Array Syntax

The syntax we just looked at for creating arrays is very common, but there are some shortcut methods you can use. For example, we could've created exactly the same array above using a *condensed syntax* like this:

```
var color=new Array("Red","Green","Blue");
```

This syntax saves a lot of coding by putting the values, separated by commas, right inside the parentheses after *Array*. Again, they're enclosed in quotation marks because they're literal strings (text). JavaScript assigns subscripts automatically, starting at 0 (zero) with the first item on the left.

However, there's an even shorter syntax you can use. This is sometimes called the *array initializer method* or *array literal method*. This syntax resembles that of creating a regular variable but puts the array elements, separated by commas, inside square brackets, like this:

```
var colors=["Red","Green","Blue"];
```

Aside from the number of characters you have to type to create the array, there's really no difference, advantages, or disadvantages among the three methods in terms of the end result. Each syntax we showed you produces an array containing the three elements where `color[0]` contains the word *Red*, `color[1]` contains the word *Green*, and `color[2]` contains the word *Blue*. Some people prefer the longer syntax because it makes the code very clear to read, from a human perspective. Some prefer the shorter syntax to keep the code tight and clean.

Array Elements

In the example above, we created an array named *colors* that contains three color names. We refer to each item in an array as an *element* of the array. Unlike *page elements* on a printed page or a webpage (like headings, paragraphs, pictures, and so forth), elements in an array have no visual appearance on the page. They're just data stored in memory, like any other JavaScript variable.

In this case, we're storing a literal string in each array element. There's nothing special about those strings. They could be anything—peoples' names, product names, filenames—because each item in the array is just a variable in which you can store information.

The difference is that rather than giving each variable a different name, we give each variable the same name and vary the subscript number. It might not seem like much of a difference, however giving each variable the same name and with a different subscript opens up whole worlds of programming possibilities.

Understanding Array Properties and Methods

Array Length Property

Every array you create in JavaScript is an array *object* that can be inspected using some properties, including `.length`. This simple property returns a number indicating how many elements are in the array. The syntax is:

```
arrayname.length
```

You simply replace *arrayname* with the name of the array. For example, you just saw three different methods of creating a simple array named *color* that consists of three elements (color names). For that array, this line of code returns *3* because there are three elements in that array:

```
color.length
```

If you add or remove elements from the array, the number returned will increase or decrease to reflect the number of items in the array.

Array Sort() and Reverse() Methods

The JavaScript array object also offers some methods for manipulating array elements. Two that can come in handy (depending on what your array contains and how you're using the data) are *.sort()* and *.reverse()*.

The *.sort()* method organizes the elements into *ascending order*. That means that if the array contains strings (text), it puts them in alphabetical order from A to Z.

The *.reverse()* method reverses whatever order elements happen to be in at the moment. If you apply a *.reverse()* right after a *.sort()*, you'll get a descending sort order, Z-A.

How Arrays Work

To give you a better idea of how this works, let's take a look at *colorarray.html* in your editor.

```
<!DOCTYPE html>

<html>

<head>

  <title>Color Array</title>

</head>

<body>

  <script>

    //Create an array with 3 elements

    var color= ["Red", "Green", "Blue"]

    //Show array length in alert box

    alert(color.length)

    //Show all array elements

    alert(color);

    //Sort array elements

    color.sort();

    //Show sorted array elements

    alert(color);

    //Reverse the sort order

    color.reverse();

    //Show reversed sort order

    alert(color);

  </script>

</body>

</html>
```

Let's step through the code and explain what it shows, when you open it in a browser.

Text equivalent start.

Topic	Information
<code>var color = ["Green", "Red", "Blue"]</code>	This line creates the array.
<code>alert(color.length)</code>	This creates the first alert box, which will show 3 because there are three items in the array.
<code>alert(color);</code>	When you click the OK button on the first alert box, a new alert box appears showing the entire array: <i>Green, Red, Blue</i> . This is simply the array elements in their natural order, separated by commas.
<code>color.sort();</code>	Since the array contains strings, this line of code puts them into alphabetical order. However, it does this in memory only. To see any proof that the sort actually happened, we need to look at the contents of the array again.

Topic	Information
<code>alert(color);</code>	This line of code is used to display the array, now sorted, in an alert box. The array is the same, but this time the elements are in alphabetical order: <i>Blue, Green, Red</i> .
<code>color.reverse();</code>	This line of code tells JavaScript to rearrange the array elements into the reverse of whatever order they're in right now. In this case, it reverses the alphabetical order. Again, this is in memory only.
<code>alert(color);</code>	So once again, this code displays the contents of the array in an alert box. Now they're in reverse alphabetical order: <i>Red, Green, Blue</i> . This is the opposite of the previous order.
Read the topic in the first column. Then read the second column for the information.	

Text equivalent stop.

So that, in a nutshell, is what arrays are all about. As you'll see in later lessons, they can be used in creative and powerful ways to do cool things in websites and other apps. But the coolest thing you can do is require *looping through* the array. We'll show you how that's done next.

Chapter 2: Creating Loops

Understanding Loops

Most programming languages, including JavaScript, offer *loops*. A loop is a block of code that repeats over and over again, any number of times, until some condition is satisfied. Loops can be used in a way that lets you avoid writing many repetitive code lines and are often used to access elements in an array. There are two different types of loops, commonly referred to as the *for loop* and the *while loop*. You'll learn about both in this chapter.

For Loops

A *for* loop provides a means of repeating a loop a predetermined number of times, while at the same time *incrementing* (increasing) or *decrementing* (decreasing) the value of some variable with each pass through the loop. In simpler terms a loop can:

- Count from up to any number.
- Count forward or backward.
- Start at any number and end at any number.

The syntax for a for loop is:

```
for (start; condition; increment) {  
  code to be executed  
}
```

To write your own for loop, you need to replace the following placeholders:

Text equivalent start.

Topic	Information
start;	A var statement to create a variable name and assign it an initial value for the loop counter.
condition;	An expression that defines the "as long as" condition that determines when the loop stops repeating.

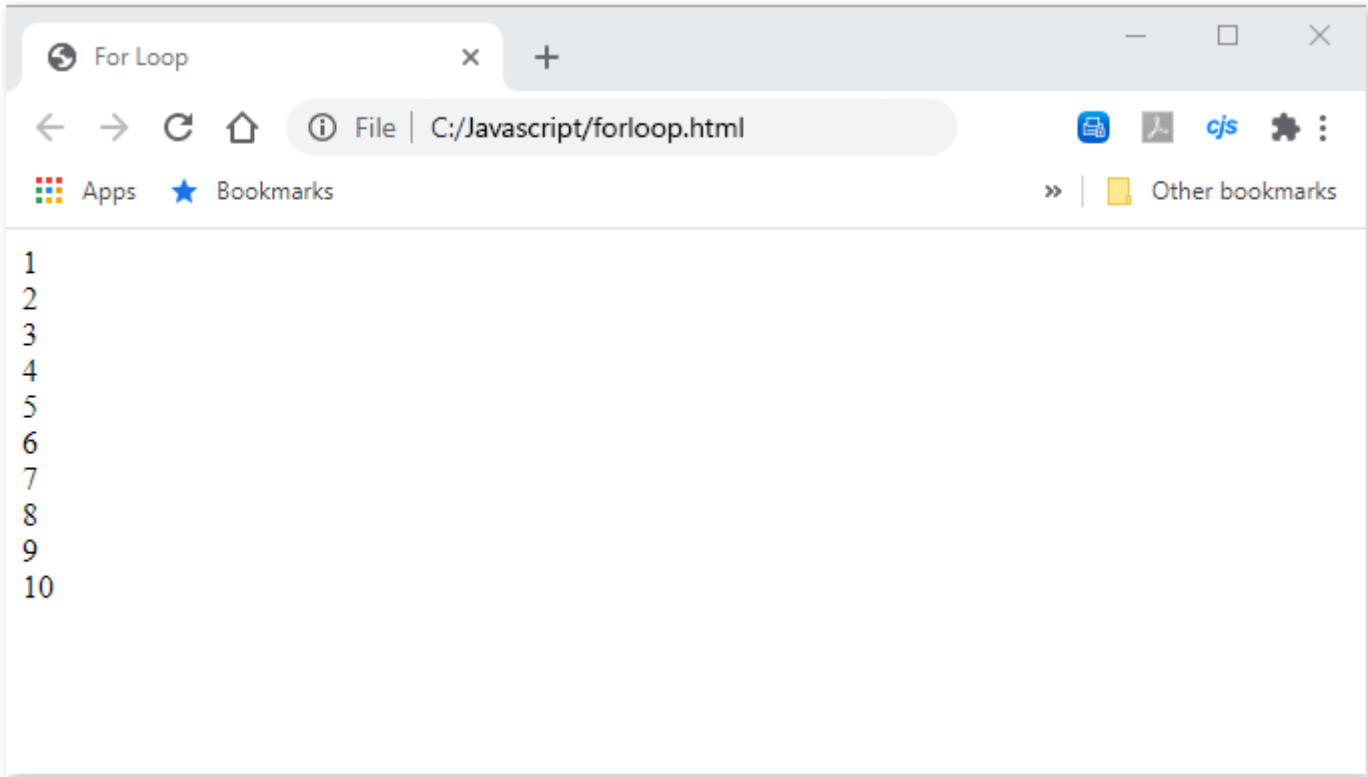
Topic	Information
increment	The amount that the variable increments (or decrements) with each pass through the loop. You can use ++ to increment the value by one, or -- to decrement the value by 1.
code to be executed	Code to be executed with each pass through the loop.
Read the topic in the first column. Then read the second column for the information.	

Text equivalent stop.

It might seem woefully complex and difficult to understand at first. But some hands-on practice can help.

Here are the Steps

1. Open *forloop.html* in a browser. You'll see the numbers 1-10 listed down the page.

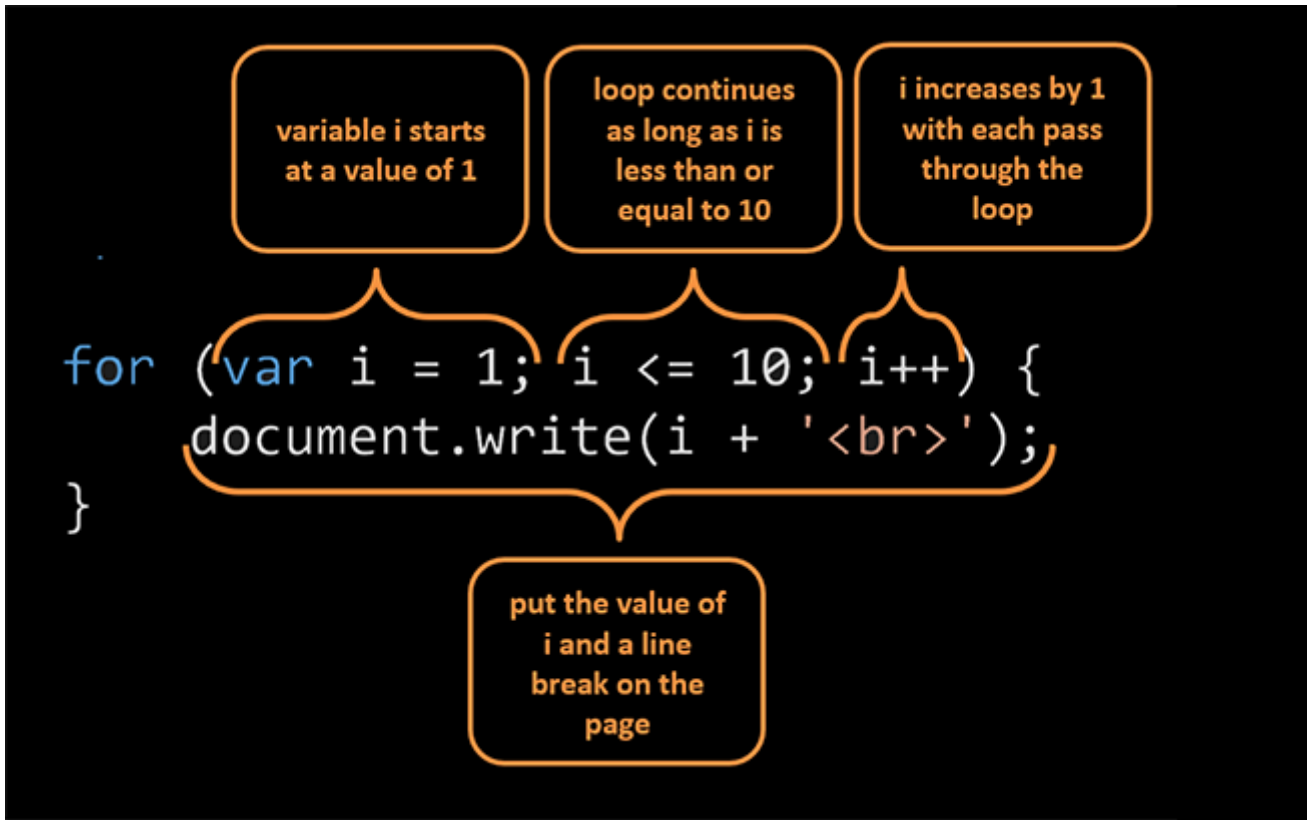


2. While this may not be all that impressive at first glance, if you look at the code, you'll see that there is a bit more going on under the hood. Go ahead and open *forloop.html* in your editor.

```
forloop.html
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <title>For Loop</title>
5 </head>
6 <body>
7 <script>
8   for (var i = 1; i <= 10; i++) {
9     document.write(i + '<br>');
10  }
11 </script>
12 </body>
13 </html>
```

3. If you're wondering how we got from an alien-looking chunk of JavaScript code to the numbers 1 to 10 listed down the page, it's all in understanding what the code *means*.

Let's break down the code:



Text equivalent start.

Topic	Information
<code>for () {}</code>	You'll notice that the for loop statement is similar to an if or else statement.
<code>var i = 1;</code>	This first expression in the parentheses of the for loop statement indicates that variable i starts with a value of 1.
<code>i <= 10;</code>	The second expression in the parentheses of the for loop statement indicates that the loop continues as long as i is less than or equal to 10.
<code>i++</code>	The last expression in the parentheses of the for loop statement indicates that i increases by 1 with each pass through the loop.
<code>document.write(i + '
');</code>	The expression in the curly braces of the for loop statement specifies that the code will place the value of i on the line and add a line break. Without this line break, all of the values would appear on the same line.

Read the topic in the first column. Then read the second column for the information.

Text equivalent stop.

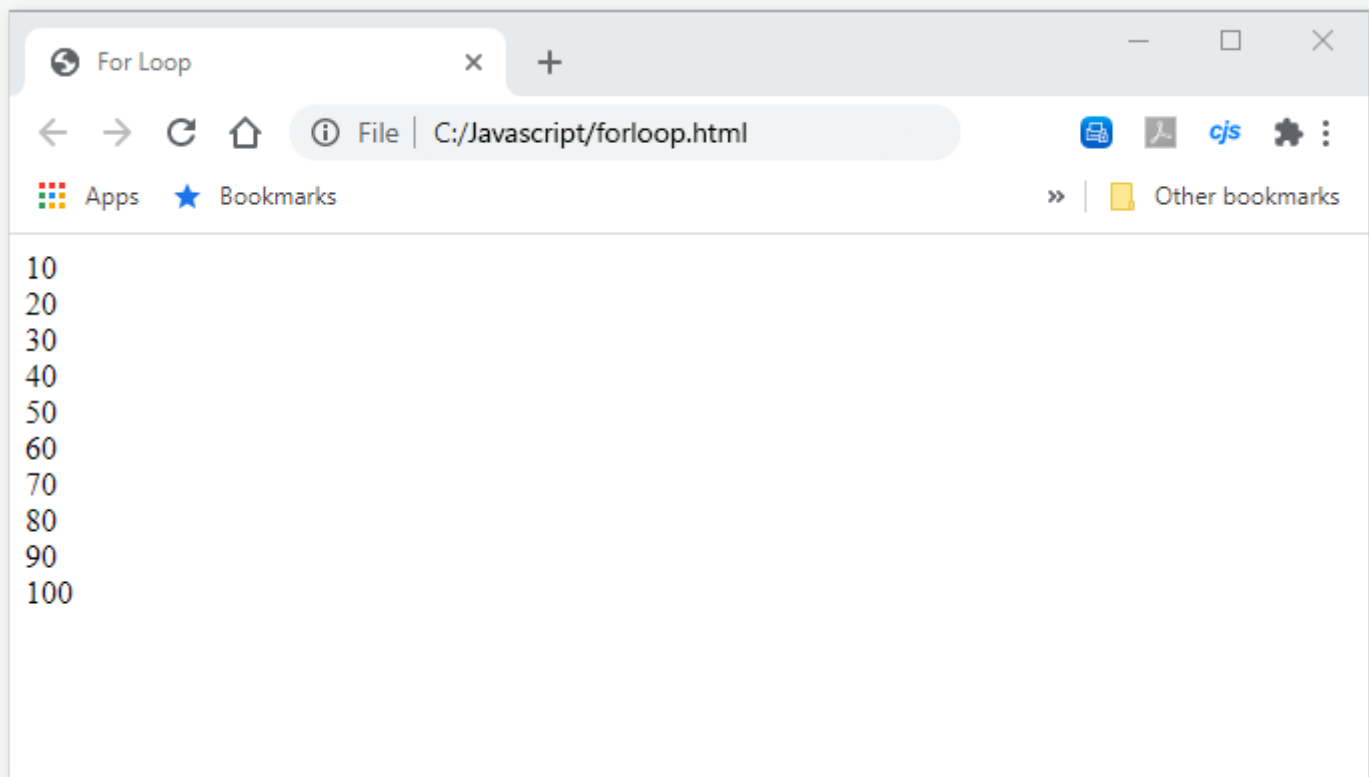
You Try It!

Experimentation can help you understand. Go ahead and make the following changes to the code:



- Change *i <= 10* to *i <= 100*
- Change *var i=1* to *var i=10*
- **Change *i++* to *i+=10***

Save your changes and reload or refresh the page in your browser. The numbers should now go up to 100 (rather than 10), the counting will start at 10 (rather than 1), and the value will increase in increments of 10 (rather than 1).



Customizing ForLoops

Increasing Integers

As you saw, you don't have to count by 1. You can use the += operator to increment by any value. The syntax is:

```
variable += value
```

Simply replace *variable* with the name of the variable to increment and *value* with the increment amount.

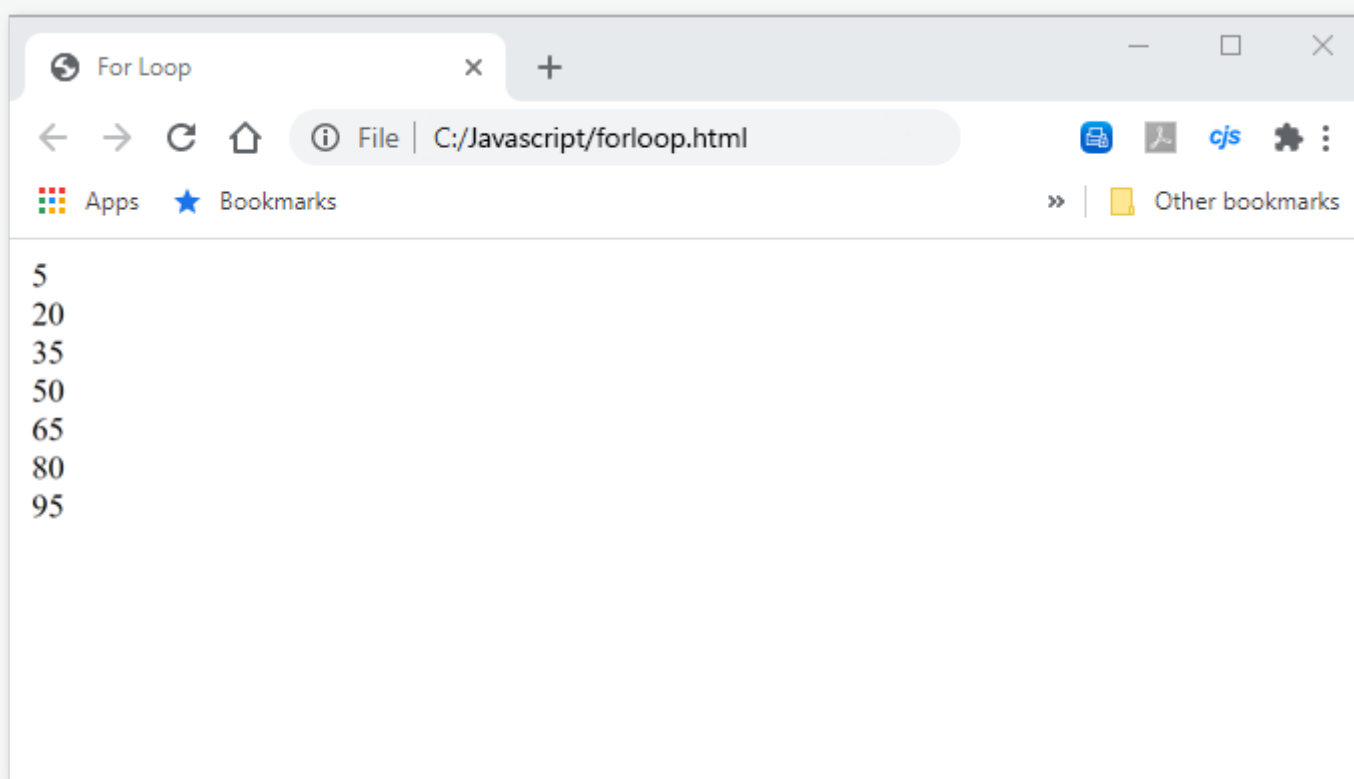
You Try It!

Try changing the loop to this:



```
for (var i = 5; i <= 100; i += 15) {  
    document.write(i + '<br>');  
}
```

In this new loop, *var i = 5* means "starting with *i* set to 5," and *i <= 100* means "for as long as *i* is less than or equal to 100", and *i += 15* means "incrementing *i* by 15 with each pass through the loop." We didn't change the code inside the loop, so it still shows the value of *i* and a line break with each pass through the loop.



Decreasing Integers

You can count backward, too, so long as the starting value is greater than the ending value.

- Use the -- operator to decrement by 1.
- Use the -= operator to decrement by some value other than 1.

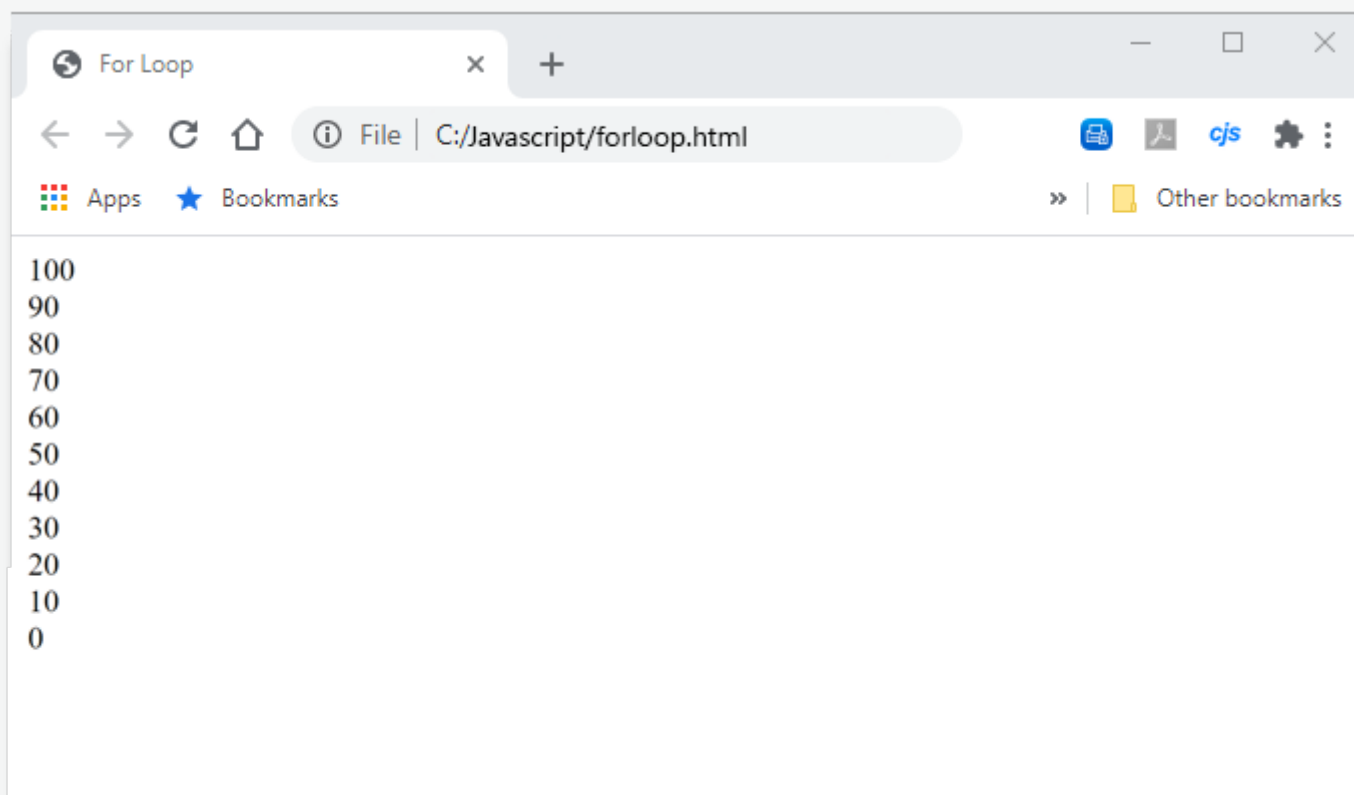
You Try It!

Change the loop so that it reads as follows:



```
for (var i = 100; i >= 0; i -= 10) {  
    document.write(i + '<br>');  
}
```

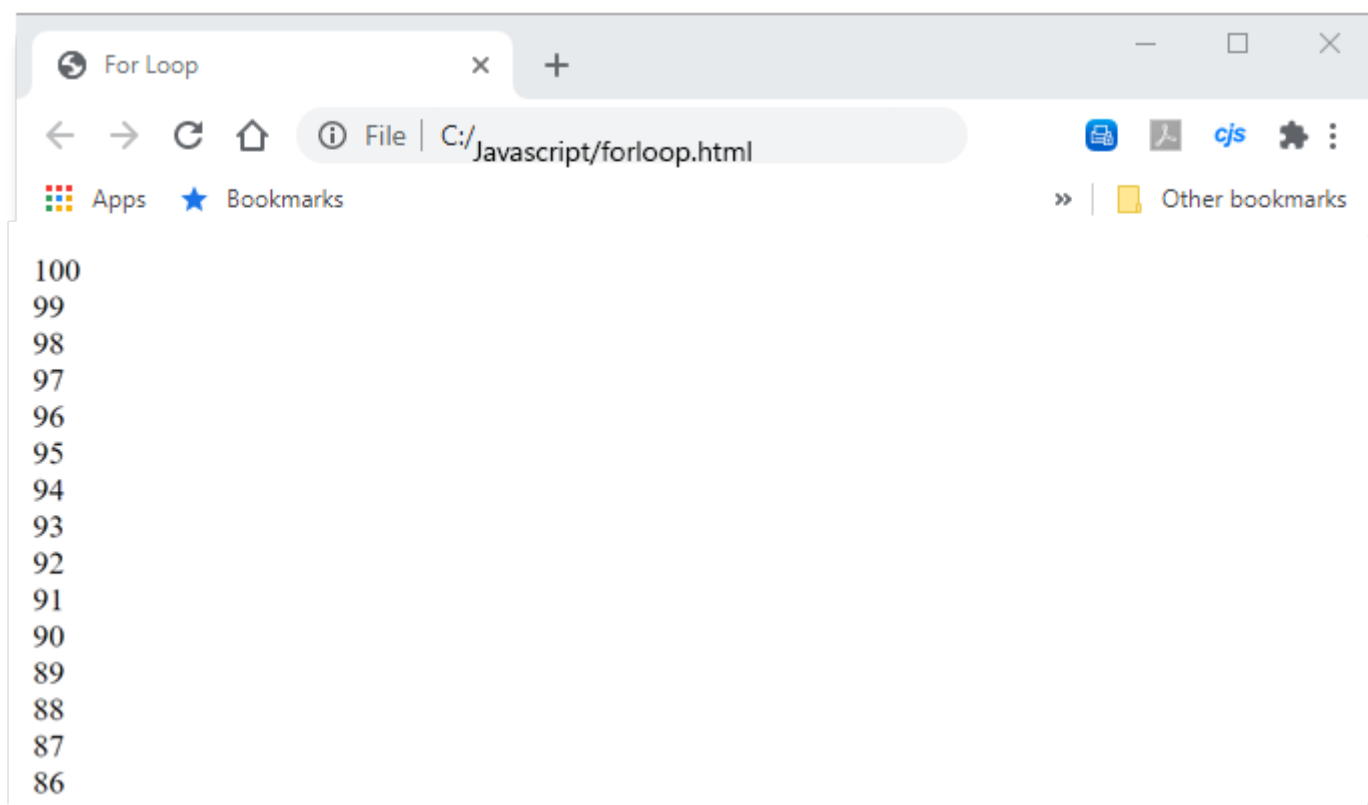
In this new loop, the variable *i* starts at 100, it loops as long as *i* is *greater than or equal to* zero, and with each loop it decrements by 10 each time. Once you save that change and view the results in a browser you'll see the loop counting from 100 to 0 by 10s.



If you take the loop we just created and change the decrementing value to `i--`, the loop will count by -1 from 100 down to 0.

```
for (var i = 100; i >= 0; i--) {  
    document.write(i + '<br>');  
}
```

You may need to scroll up and down the page to see the entire list:



Common Loop Operators

The most common two-character operators used in loops are as follows:

Operator	Meaning
<=	Less than or equal to
>=	Greater than or equal to
++	Increment by 1
--	Decrement by 1
+=	Increment by specified amount
-=	Decrement by specified amount

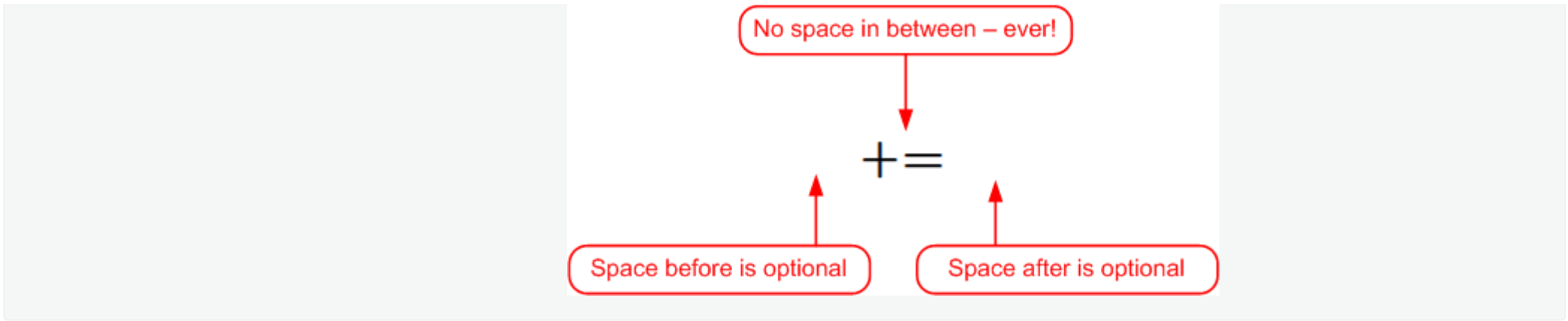
Common Two-Character Operators



Syntax Rules

If you've been typing code for a while, you've probably learned via the school of hard knocks that simply not knowing the syntax and typing rules alone can hang you up. So here are a few rules to keep in mind when typing two-character operators like the ones you've seen here:

- You can put a space *before* the operator.
- You can put a space *after* the operator.
- Never put a space *between* the two characters that form the operator.



Understanding *While* Loops

In addition to the for loop, JavaScript offers a couple of *while* loops. Like the *for* loops, these loops repeat one or more lines of code as long as some condition remains true. The condition does not need to be a counter with some incrementing number. There are two types: *while* loops and *do while* loops.

While and Do While Loop Syntax

The syntax for the *while* loop is:

```
while (condition) {  
  code to be executed  
}
```

The syntax for the *do while* loop is:

```
do {  
  code to be executed  
} while (condition)
```

As you can see, the *do while loop* is similar to the *while loop*. However, the decision whether to repeat the loop is at the top of the while loop and at the bottom of the do while loop. In both syntaxes, replace the placeholders with your own information as follows:

Text equivalent start.

Topic	Information
<code>condition;</code>	An expression that defines the "as long as" condition that determines when the loop stops repeating.

Topic	Information
code to be executed	Code to be executed with each pass through the loop
Read the topic in the first column. Then read the second column for the information.	

Text equivalent stop.

While vs. Do While Loops

The main difference between the two loops is that the while loop might not repeat at all, but the do while loop is guaranteed to repeat at least once. That's because with the while loop, the decision as to whether or not to do the statement inside the loop is made at the top.

Example



The code inside the following while loop would never execute, because the condition for looping is that the variable x must contain a number that's less than 10. However, x is already 100 before the condition is evaluated, so all the code inside the loop is skipped over, and execution resumes below the closing curly brace (}) at the bottom of the loop.

```
var x = 100
while (x < 10) {
  code to be executed
}
```

By contrast, code inside a do loop always executes at least once. Here's the same concept as above but with a do loop:

```
var x = 100
do {
  code to be executed
} while (x < 10)
```

Here the variable *x* gets a value of 100, so execution continues with the *do* and the code to be executed inside the loop. Thus, the code to be executed does execute at least that one time. The decision on whether or not to repeat the loop comes after the loop. In this case, the condition of *x* being less than 10 proves false, so the loop doesn't repeat. Instead, code execution would proceed below the loop.

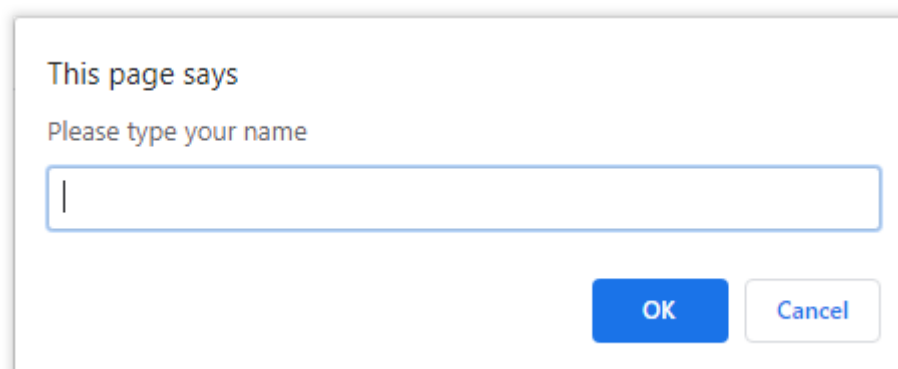
When creating *do* loops, it's important to ensure that the *condition* is something that eventually evaluates to false. If the condition never evaluates to false, then the loop never ends. We call that an *infinite loop*. It isn't truly infinite in that it would only run until the user closes the browser or browses to another page. Still, the goal is always to write the loop so that it only repeats as many times as necessary to perform some task. There would never be a good reason to intentionally create an infinite loop.

Putting It into Practice

Unlike a *for* loop, which always involves some kind of counting, the condition that keeps a *while* loop going, or not, can be any expression that evaluates to true or false. Let's try one out!

Here Are the Steps

1. Open the *dowhileloop.html* you saved in your *Intro JavaScript* folder in a browser.
2. A JavaScript prompt box appears. The exact appearance of the box depends on your browser, but it'll be some kind of box asking you to type your name.



This page says
Please type your name

OK Cancel

3. Don't type your name just yet. Try closing the box by clicking the **OK** button without typing your name first.



The box will come right back

In fact, it'll keep coming back until you actually type your name and click **OK**. That's because we put the code that displays the prompt inside a loop that repeats until the user actually types some text and clicks **OK**.

4. Now go ahead and type *your name* in the box and click **OK**.
5. The page should now say *Hello your name* because the parameters were finally met to move forward with the script.

Let's look at the code to see how that works.

Text equivalent start.

Topic	Information
<pre>do {</pre>	In the page, the script is right under the <body> tag, which means the script executes as soon as the page opens. This indicates a do loop is about to begin.
<pre>var fname = prompt("Please type your name", "");</pre>	That <i>var fname =</i> part should be familiar by now. It creates a variable named <i>fname</i> . The prompt () method tells JavaScript to prompt (ask) the user for a value by putting a prompt box on the page. That prompt box is a simple thing with a textbox, OK and Cancel buttons. The first value inside the parentheses of the prompt() method, <i>"Please type your name"</i> , appear as instructions inside the prompt box. The second value, "", appears as text already typed into the textbox that's in the prompt box. A pair of quotation marks with no space in between produces an empty string, which just leaves the textbox empty.
<pre>} while (fname == null fname.length < 1)</pre>	Here the do loop ends and the while loop begins. That says to repeat the loop if the <i>fname</i> variable equals null (fname == null), or () if the <i>fname</i> variable's length is less than one (fname.length < 1). The prompt box will keep coming back until the user types something into the box and clicks OK . So, basically, you have a loop that repeats until some condition is met. Unlike a for loop, where there's counting involved, this loop repeats as long as the user refuses to enter a name (or until the user closes the page or browser).
<pre>document.write("Hello " + fname);</pre>	Once the conditions are met to end the loop, this code executes. It grabs whatever the user entered in for their name (<i>fname</i> variable), and writes "Hello" plus their name on the page.

Read the topic in the first column. Then read the second column for the information.

Text equivalent stop.

Basically, the JavaScript code execution stops as soon as the prompt box shows on the screen, and the code just waits for the user to respond. Depending on the users response, here's what happens next:

- **If the user clicks Cancel**, the box closes and the variable receives a value of null.
- **If the user clicks OK without typing anything**, the variable retains its value of "", which is an empty string with a length of zero.
- **If the user types something in the box and clicks OK**, the variable receives as its value whatever the user typed into the box.

Clicking **Cancel** or **OK** without typing the name allows the box to close briefly, but because the conditions haven't been satisfied the loop continues. So, the box pops right back up again. Of course in real life, you might not want to be so insistent. But the purpose of this exercise is simply to try out a while loop and illustrate its syntax.

Decision-making is a big part of interactive apps and Web pages. In this course, you've seen *if* decisions. The loops you saw in the previous chapter also have to make decisions about whether or not to repeat the loop. In addition to all of that, JavaScript offers two more syntaxes to aid with programmatic decision-making, the *switch statement* and the *ternary operator*. Like most techniques, they have somewhat scary-sounding names. But they're just ways to write code to make decisions. You'll learn about both methods up next.

Chapter 3: More Decision-Making Options

Understanding Switch Statements

What is a Switch Statement?

The switch statement allows JavaScript to execute a single block of statements from a series of possibilities. Technically, it acts exactly like if . . . else but provides a shorter, easier-to-type syntax that's especially convenient when there are more than two possible values to consider for making the decision.

The switch statement syntax is:

```
switch(value)
{
  case x1:
    code to execute;
    break;
  case x2:
    code to execute;
    break;
  ...
  default:
    code to execute;
}
```

How it Works

In practice, *value* is typically the name of some variable that contains a value. That value is compared to the value expressed in each case (*x1*, *x2*, and so forth).

- **If the *x* value does not match the case *value*,** then execution jumps to the next case line.
- **If the *x* value does match the case *value*,** the code to execute for that case executes. Then, the break statement sends execution past the bottom of the switch statement, thereby ignoring all other possible cases.

The ellipsis (...) indicates that you can have any number of case statements, each inspecting for a value.



The *default:* block is optional.

It represents the code to execute if (and only if) none of the case statements proved true.

To see an example of the syntax in use, let's try out a relatively simple example. You'll need to use the *switchdemo.html* page you saved in your Intro JavaScript folder.

Here are the Steps

- 1. Open the *switchdemo.html* page in a browser.

Type a number from 1 to 4 then click Go:

- 2. Go ahead and type 0 into the box, and then click **Go**.

Type a number from 1 to 4 then click Go:

I asked for a number between 1 and 4.

- 3. Now type 3 into the box and click **Go**.

Type a number from 1 to 4 then click Go:

three

As you can see, after you click **Go**, a JavaScript switch statement decides what to put on the page. If you type a number between one and four, the name of that number shows on the page. If you type anything else, the page asks again for a number between 1 and 4. But, why does it do this? Let's step through the code to see how it makes this happen.

Topic	Information
<pre><p>Type a number from 1 to 4 then click Go: <input type="number" id="txtEntry" style="width: 60px;" /> <button onclick="switchdemo()"> Go</button> </p></pre>	<p>In the body of the page, you'll see some standard HTML and CSS were used to display text instructions, an input box, and a button. The input is named <i>txtEntry</i> and is set to allow only numbers to be entered. You'll also notice that the onclick event for the button calls a JavaScript function named <i>switchdemo()</i>.</p>
<pre><p id="result" style="color:red;"></p></pre>	<p>There's also a paragraph named <i>result</i> on the page, styled in red. However, it's a pair of <code><p> . . . </p></code> tags with no text in between them, which means you don't really see it on the page.</p>
<pre>function switchdemo() {</pre>	<p>The switchdemo() function is up in the head of the page between <code><script></code> and <code></script></code> tags.</p>
<pre>//Get the value from the textbox. var num = parseInt(document.getElementById("txtEntry").value)</pre>	<p>When called, the <i>switchdemo</i> function first executes this code. This line creates a variable named <i>num</i> and puts into it whatever value the user typed into the textbox named <i>txtEntry</i>. The <i>parseInt()</i> method is used to convert that entry to a number. If the user typed something other than a number into the textbox, parseInt will return NaN (not a number).</p>
<pre>//Declare msg variable as string. var msg = ""</pre>	<p>Next, the code creates a variable named <i>msg</i> and starts it off as an empty string.</p>
<pre>switch (num) {</pre>	<p>Next comes a switch statement that decides what to put into the <i>msg</i> variable based on what's in the <i>num</i> variable.</p>
<pre>case 1: msg = "one"; break;</pre>	<p>So if the variable named <i>num</i> contains 1, then the variable <i>msg</i> receives the word <i>one</i> as its value, and then the break statement skips over all other possible options to the first line under the closing <code>}</code> at the bottom of the switch.</p>
<pre>case 2: msg = "two" break;</pre>	<p>The second case puts <i>two</i> in the <i>msg</i> variable if <i>num</i> contains 2, and then skips over the rest.</p>
<pre>case 3: msg = "three" break;</pre>	<p>The third case puts <i>three</i> in the <i>msg</i> variable if <i>num</i> contains 3, and then skips over the rest.</p>

Topic	Information
<pre>case 4: msg = "four" break;</pre>	The fourth case puts <i>four</i> in the <i>msg</i> variable if <i>num</i> contains 4, and then skips over the rest.
<pre>default: msg = "I asked for a number between 1 and 4."; }</pre>	If none of the first four cases proves true (because <i>num</i> contains a value that's not 1, 2, 3, or 4), then the default case kicks in, which stores the words " <i>I asked for a number between 1 and 4.</i> " in the <i>msg</i> variable.
<pre>document.getElementById("result").innerHTML = msg; }</pre>	After the switch statement, this line of code puts whatever word or words are in the <i>msg</i> variable into that paragraph named <i>result</i> .
Read the topic in the first column. Then read the second column for the information.	

Text equivalent stop.



Remember

As always, *num* (short for *number*) and *msg* (short for *message*) are just made up variable names. They have no special, built-in significance in JavaScript. However, using variable names that make sense based on the code you are building makes it easier to remember.

Logically, the switch statement does exactly the same thing these if . . . else statements would do:

```
if (num == 1) {
  msg = "one";
} else if (num == 2) {
  msg = "two"
} else if (num == 3) {
  msg = "three"
} else if (num == 4) {
  msg = "four"
} else {
  msg = "I asked for a number between 1 and 4.";
}
```


It really wouldn't matter if you used the `if . . . else` statements rather than the case statement. The code would execute the same and in the same amount of time (a few milliseconds). But many people find it easier to type it as a switch statement, and so you're likely to see those often when viewing other peoples' JavaScript code.

Understanding Ternary Operators

What is a Ternary Operator?

While the switch statement is generally used for making a decision based on many options, the *ternary operator* (also called the *conditional operator*) is more of a shorthand notation for making small, simple *if* decisions, usually just to assign a value to a variable. The syntax is:

```
condition ? return if true : return if false;
```

To use it, replace these placeholders as follows:

- **condition** - This must be an expression that evaluates to true or false.
- **return if true** - The value returned if the *condition* is true.
- **return if false** - The value returned if the *condition* is false.

How it Works

As an example, consider the following block of code:

```
var age = 64  
  
var admission = "Your cost: $" + (age < 60 ? "7.50" : "5.00");
```

We start by creating a variable named *age*, and we put a number in it. (In real life, the user might enter that number, or it may be determined through some other means.) Next, we create a variable named *admission*, which starts off as the string "Your cost: \$". Then, added to that is either 7.50 or 5.00, depending on the value of *age*.

- If the *age* variable contains a value that's less than 60, then the admission variable receives the complete value "Your cost: \$7.50".

- If the *age* variable contains a value that isn't less than 60, the admission variable ends up with the value "Your cost: \$5.00".

So, basically, the ternary operator does exactly what the *if* statement below does. It just does it with less code typing (what programmers would call in a less verbose way):

```
var age = 64
var admission = "Your cost: $"
if (age < 60) {
  admission += "7.50"
} else {
  admission += "5.00"
}
```

We'll see that put to practical use, along with other new techniques you learned today, in some future code examples. For now, your head is likely spinning with new concepts. So let's call it a day and wrap up what you've learned with a quick review.

Review

Today's lesson was all about expanding your JavaScript skills to include new tools and concepts found in all programming languages. Let's review:

- **Creating Arrays:** You learned that an array is simply a list of variables you can access by number.
- **Creating Loops:** Loops provide a means of repeating one or more lines of code until some condition is met. We looked at *for* loops, *while* loops, and *do while* loops.
- **More Decision-Making Options:** You discovered that the switch statement and ternary operators provide alternative syntaxes for simplifying decision-making in code.

If you're feeling a bit of culture shock about these things that never crossed your mind before, don't panic. You use these things whenever you use your PC, laptop, tablet, or smartphone. It's just that they're in code that other people wrote. It's just that now, it's your turn to write such code.

In the next lesson, you'll learn about timers. When used in conjunction with loops and arrays, timers allow you to create slideshows, animations, and other visual effects. See you there!

Lesson 8 Assignment

Slideshow Array

For today's assignment we'll do another slideshow, but this time use an array to hold the image file names. That way, the file names can be anything you like, they don't have to follow a pattern like image01.png, image02.png, and so forth.

We'll also use JavaScript, rather than CSS, to pre-load the images. This isn't because the slide show is "better" or "faster" than the original. It's just because it illustrates how you can often achieve exactly the same result for something with a completely different approach.



Download

So you don't have to start from scratch, we've constructed a webpage that is already programmed to display a box that changes colors when you click it.

You'll need the following files:

- **Assignment 8.html and Assignment8.txt:** This webpage gives you the HTML and CSS code you'll need.
- **Slideshow:** This folder provides you with the images you'll be programming into the slideshow.



[Download L08-Assignment.zip](#)

1.3 MB.(Gigabytes) ZIP

You can retype or copy and paste if you'd like to run through the code for the page, or you can simply download and save the files to your *Intro to JavaScript* folder.

The code on the webpage we just gave you is a little bit different from the previous example because we won't be using background images this time. Instead, we'll use a single `` tag, and change it's source (src= attribute) with JavaScript.

Watch the following video explaining what's going on as the JavaScript code is being written, then follow the steps.

Here are the Steps

1. Open *Assignment8.html* in your code editor.
2. Go to the `<script> . . . </script>` tags below the `<style>` block, but still in the `<head>` block.
3. Add the following JavaScript comment and code to create a image number global variable:

```
//Global variable counter to keep track of current picture  
var imgnumber = 0;
```

4. Next, add a comment and code that creates a global variable for the location of the images:

```
//Name of the folder that contains the pictures  
var picfolder = "slideshow";
```

5. Now, create an array using the filenames for each image:

```
//Array of picture file names (can be any length)  
var picfiles = ["candle.jpg", "chip.jpg", "fire.png", "paint.png", "water.png"];
```

6. In order for the images to actually display, you need to create an empty array of image objects:

```
//Declare an empty array of image objects  
var images = [];
```

7. Then you need to tell the code, how to populate the image objects from a file:

```
// Populate each image object with a picture from a file
for (var i = 0; i < picfiles.length; i++) {
    images[i] = new Image();
    images[i].src = picfolder + "/" + picfiles[i];
}
```

8. Next, you'll establish a variable to define the number of images:

```
// Number of images in array is length minus one (because first is always zero)

var imgcount = images.length - 1;

function slide(num) {
    // Increment or decrement in range 1 to imgcount

    imgnumber = (imgnumber + num) % imgcount;

    // If number reaches zero, loop around to imgcount

    imgnumber = imgnumber < 0 ? imgcount : imgnumber;

    // Set the image source to the source of the array image

    document.getElementById("target").src = images[imgnumber].src;
}
```

9. This next bit of code, sets it so that the first image in the array is displayed once the page is done loading:

```
//Wait until the page is fully loaded before setting source of the image

window.addEventListener('load', (event) => {
    // Start by showing the first image in the array

    document.getElementById("target").src = images[0].src;
})
```

10. Save your progress, and then open *Assignment8.html* in your browser.

11. Click the left and right arrows and watch the images in the slideshow change. With each click it should change from candle.jpg, chip.jpg, fire.png, paint.png, to water.png in a continuous loop.



Download

To check your code, you can download the correct code here:



[Download L08-Answers.zip](#)

2.5 KB.(Gigabytes) ZIP

Lesson 8 Resources for Further Learning

JavaScript Array (<http://www.tizag.com/javascriptT/javascriptarray.php>)

<http://www.tizag.com/javascriptT/javascriptarray.php>

Here's another useful page offering a quick tutorial on JavaScript arrays.

Array Object (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Indexed_collections#Array_object)

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Indexed_collections#Array_object

Though a bit more technical than the previous two pages, this one offers good reference information on various array properties and methods.

JavaScript Loops (<http://www.landofcode.com/javascript-tutorials/javascript-loops.php>)

<http://www.landofcode.com/javascript-tutorials/javascript-loops.php>

Click this link for yet another good page about JavaScript loops.

JavaScript Switch Statement (https://www.w3schools.com/js/js_switch.asp)

https://www.w3schools.com/js/js_switch.asp

This page offers a good brief tutorial on switch statements.

JavaScript: Conditional operator (<https://www.w3resource.com/javascript/operators/conditional.php>)

<https://www.w3resource.com/javascript/operators/conditional.php>

Click this link for a good resource on the JavaScript ternary operator.

HTML DOM addEventListener() Method (https://www.w3schools.com/jsref/met_document_addeventlistener.asp)

https://www.w3schools.com/jsref/met_document_addeventlistener.asp

Here is a look at the code we used to delay JavaScript code execution until after the page is fully rendered (that is, after all the HTML and CSS code has been executed).

EventTarget.addEventListener() (<https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener>)

<https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener>

Here is some more information on that `.addEventListener`, which will also revisit in Lesson 10.

