Contents

How to Use This Book

The examples in this book demonstrate JavaScript features that are supported by modern web browsers, and the screenshots illustrate the actual results produced by the listed code examples. Certain colorization conventions are used to clarify the code listed in the steps...

JavaScript code is colored blue, programmer-specified names are colored red, literal text is colored black, and code comments are colored green:

```
let sum = ( 9 + 12 ) / 3   // Equivalent to 21 / 3.
document.getElementById( 'info' ).innerHTML += 'Grouped sum: ' + sum
```

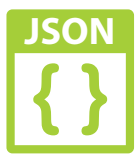
HTML tags are colored blue, literal text is colored black, and element attribute values are colored orange in both HTML and JavaScript code:

```
<p id="info">JavaScript in easy steps</p>
```

Additionally, in order to identify each source code file described in the steps, a file icon and file name appears in the margin alongside the steps:

page.html

external.js



data.json

data.xml

echo.pl

banner.svg

The source code of HTML documents used in the book's examples is not listed in full to avoid unnecessary repetition, but the listed HTML code is the entire fragment of the document to which the listed JavaScript code is applied. You can download a single ZIP archive file containing all the complete example files by following these easy steps:

1. Browse to **www.ineasysteps.com** then navigate to Free Resources and choose the Downloads section

2. Next, find JavaScript in easy steps, 6th edition in the list, then click on the hyperlink entitled All Code Examples to download the ZIP archive file

3. Now, extract the archive contents to any convenient location on your computer

If you don't achieve the result illustrated in any example, simply compare your code to that in the original example files you have downloaded to discover where you went wrong.

1 Get Started in JavaScript

This chapter is an introduction to the exciting world of JavaScript. It demonstrates how to add scripts to HTML documents that provide JavaScript variables and functions.

Meet JS

JavaScript ("JS") is an object-based scripting language whose interpreter is embedded inside web browser software such as Google Chrome, Microsoft Edge, Firefox, Opera, and Safari. This allows scripts contained in a web page to be interpreted when the page is loaded in the browser to provide functionality. For security reasons, JavaScript cannot read or write files, with the exception of "cookie" files that store minimal data.

Created by Brendan Eich at Netscape, JavaScript was first introduced in December 1995, and was initially named "LiveScript". It was soon renamed, however, to perhaps capitalize on the popularity of Sun Microsystem's Java programming language – although it bears little resemblance.

Before the introduction of JavaScript, web page functionality required the browser to call upon "server-side" scripts, resident on the web server, where slow response could impede performance. Calling upon "client-side" scripts resident on the user's system, overcame the latency problem and provided a superior experience.

Brendan Eich, creator of the JavaScript language, also co-founded the Mozilla project and helped launch the Firefox web browser.

JavaScript quickly became very popular but a disagreement arose between Netscape and Microsoft over its licensing – so Microsoft introduced its own version named "JScript". Although similar to JavaScript, the new JScript version had some extended features. Recognizing the danger of fragmentation, the JavaScript language was standardized by the Ecma International standards organization in June 1997 as "ECMAScript". This helped to stabilize core features but the name, sounding like some kind of skin disease, is not widely used and most people will always call the language "JavaScript".

The JavaScript examples in this book describe three key ingredients:

- **Language basics** – illustrating the mechanics of the language syntax, keywords, operators, structure, and built-in objects.
- **Web page functionality** – illustrating how to use the browser's Document Object Model (DOM) to provide user interaction.
- **Web applications** – illustrating responsive web-based apps and JavaScript Object Notation (JSON) techniques.

Include Scripts

To include JavaScript code directly in an HTML document it must be inserted between **<script>** and **</script>** tags, like this:

```
<script>
document.getElementById( 'message' ).innerText = 'Hello World!'
</script>
```

An HTML document can include multiple scripts, and these may be placed in the head or body section of the document. It is, however, recommended that you place scripts at the end of the body section (immediately before the **</body>** closing tag) so the browser can render the web page before interpreting the script.

JavaScript code can also be written in external plain text files that are given a **.js** file extension. This allows several different web pages to call upon the same script. In order to include an external script in the HTML document, the file name of the script must be assigned to a **src** attribute of the **<script>** tag, like this:

```
<script src="external_script.js"> </script>
```

Again, this can be placed in the head or body section of the document, and the browser will treat the script as though it was written directly at that position in the HTML document.

Assigning only the file name of an external script to the **src** attribute of a **<script>** tag requires the script file to be located in the same folder (directory) as the HTML document. If the script is located in an adjacent folder you can assign the relative path address of the file instead, like this:

```
<script src="js/external_script.js"> </script>
```

If the script is located elsewhere, you can assign the absolute path address of the file, like this:

```
<script src="https://www.example.com/js/external_script.js">
</script>
```

You can also specify content that will only appear in the web page if the user has disabled JavaScript in their web browser by including a **<noscript>** element in the body of the HTML document, like this:

```
<noscript>JavaScript is Not Enabled!</noscript>
```

You may see a **type="text/javascript"** attribute in a **<script>** tag but this is no longer required as JavaScript is now the default scripting language for HTML.

Do not include **<script>** and **</script>** tags in an external JavaScript file, only the script code.

External script files can make code maintenance easier but almost all examples in this book are standalone for clarity, so include the script code between tags directly in the HTML document.

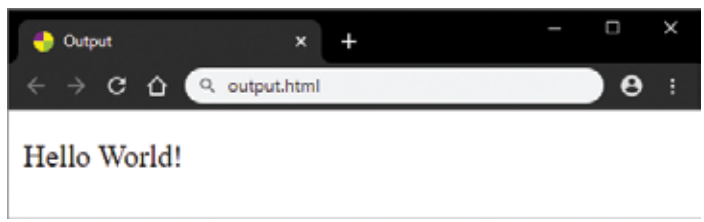
Console Output

Notice the use of the . period (full stop) operator to describe properties or methods of an object using "dot notation".

JavaScript can display output by dynamically writing content into an HTML element. For example, with this code:

```
document.getElementById( 'message' ).innerText = 'Hello World!'
```

The element is identified by the value assigned to its **id** attribute and the **innerText** property specifies text to be written there.



There is also a **document.write( )** method that replaces the entire header and body of the web page, but its use is generally considered bad practice.

Additionally, JavaScript can display output by writing content into a pop-up dialog box, like this:

```
window.alert( 'Hello World!' )
```

This calls the **alert( )** method of the **window** object to display the content specified within the ( ) parentheses in a dialog box.

The console provides helpful messages if an error occurs in your code – so is great for debugging the code.

When developing in JavaScript, and learning the language, it is initially better to display output in the browser's JavaScript console, like this:

```
console.log( 'Hello World!' )
```

This calls the **log( )** method of the **console** object to display the content specified within the ( ) parentheses in a console window. All leading browsers have a JavaScript console within their Developers Tools feature – typically accessed by pressing the F12 keyboard key. As the Google Chrome web browser is statistically the most popular browser at the time of writing it is used throughout this book to demonstrate JavaScript, and initially its console window is used to display output.

1 Create an HTML document that includes an empty paragraph and a script to display output in three ways

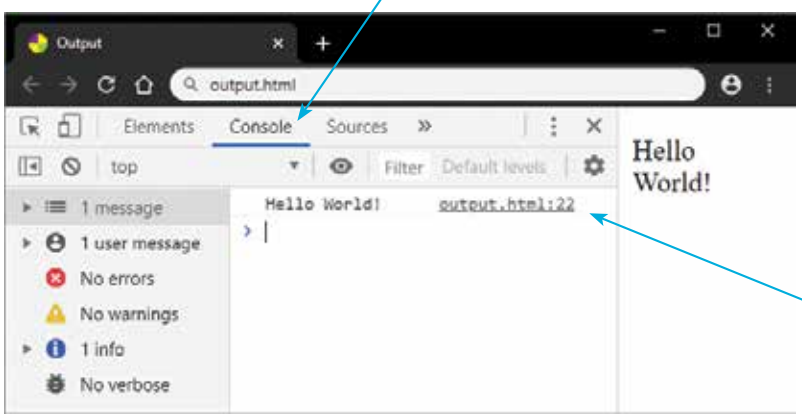
```
<p id="message"></p>
<script>
document.getElementById( 'message' ).innerText =
                                        'Hello World!'
window.alert( 'Hello World!' )
console.log( 'Hello World!' )
</script>
```

output.html

2 Save the HTML document then open it in your browser to see the output written in the paragraph and displayed in a dialog box – as illustrated opposite

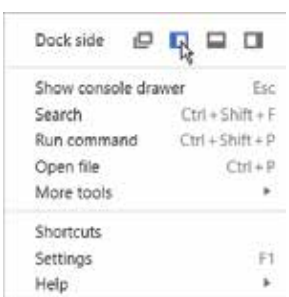
3 Next, hit the **F12** key, or use your browser's menu to open its Developers Tools feature

4 Now, select the **Console** tab to see the output written into the console window



See that the console displays the output plus the name of the HTML document and the line number upon which the JavaScript code appears that created the output.

5 Click the **Show/Hide** button to hide or show the sidebar, click the **Customize** button to choose how the console window docks in the browser window, then click the **Clear** button to clear all content from the console



Make Statements

The JavaScript keywords are described on pages 14-15 and you will learn about operators, values, and expressions later.

JavaScript code is composed of a series of instructions called "statements", which are generally executed in top-to-bottom order as the browser's JavaScript engine proceeds through the script.

Each statement may contain any of the following components:

- **Keywords** – words that have special significance in the JavaScript language.
- **Operators** – special characters that perform an operation on one or more operands.
- **Values** – text strings, numbers, Boolean **true** or **false**, **undefined**, and **null**.
- **Expressions** – units of code that produce a single value.

An "expression" produces a value, whereas a "statement" performs an action.

In earlier JavaScript code each statement had to be terminated by a **;** semicolon character – just as each sentence must end with a . period (full stop) character in the English language. This is now optional so may be omitted unless you wish to write multiple statements on a single line. In that case, the statements do need to be separated by a semicolon, like this:

statement* ; *statement* ; *statement

Some JavaScript programmers still prefer to end each statement with a semicolon. The examples in this book choose to omit them for the sake of concision but the choice is yours.

The JavaScript interpreter ignores tabs and spaces ("whitespace") so you should use spacing to make your code more readable. For example, when adding two numbers:

total = 100 + 200 rather than **total=100+200**

Use the space bar to indent statements, as tab spacing may be treated differently when viewing the code in text editors.

JavaScript statements are often grouped together within **{ }** curly brackets ("braces") in function blocks that can be repeatedly called to execute when required. It is good practice to indent those statements by two spaces to improve readability, like this:

```
{
  statement
  statement
  statement
}
```

The rules that govern the JavaScript language is called "syntax", and it recognizes two types of values – fixed and variable. Fixed numeric and text string values are called "literals":

- **Number literals** – whole number integers, such as **100**, or floating-point numbers such as **3.142**.
- **String literals** – text within either double quotes, such as **"JavaScript Fun"**, or single quotes such as **'JavaScript Fun'**.

Variable values are called, quite simply, "variables" and are used to store data within a script. They can be created using the JavaScript **let** keyword – for example, **let total** creates a variable named "total". The variable can be assigned a value to store using the JavaScript **=** assignment operator, like this:

```
let total = 300
```

Other JavaScript operators can be used to form expressions that will evaluate to a single value. Typically, an expression may be enclosed within **( )** parentheses like this expression that comprises numbers and the JavaScript **+** addition operator and evaluates to a single value of 100:

```
( 80 + 20 )
```

Expressions may also contain variable values too, like this expression that comprises the previous variable value, the JavaScript **-** subtraction operator, and a number, to also evaluate to a single value of 100:

```
( total - 200 )
```

JavaScript is a case-sensitive language so variables named **total** and **Total** are regarded as two entirely different variables.

It is good practice to add explanatory comments to your JavaScript code to make it more easily understood by others, and by yourself when revisiting the code later. Anything that appears on a single line following **//** double slashes or between **/*** and ***/** character sequences on one or more lines will be ignored.

```
let total = 100  // This code WILL be executed.

/* let total = 100
  This code will NOT be executed. */
```

Decide on one form of quotes to use in your code for string literals and stick with it for consistency. The examples in this book use single quotes.

It is often useful to "comment-out" lines of code to prevent their execution when debugging code.

Avoid Keywords

In JavaScript code you can choose your own names for variables and functions. The names should be meaningful and reflect the purpose of the variable or function. Your names may comprise letters, numbers, and underscore characters, but they may not contain spaces or begin with a number. You must also avoid these words of special significance in the JavaScript language:

JavaScript Keywords			
abstract	arguments	await	boolean
break	byte	case	catch
char	class	const	continue
debugger	default	delete	do
double	else	enum	eval
export	extends	false	final
finally	float	for	function
goto	if	implements	import
in	instanceof	int	interface
let	long	native	new
null	package	private	protected
public	return	short	static
super	switch	synchronized	this
throw	throws	transient	true
try	typeof	var	void
volatile	while	with	yield

JavaScript Objects, Properties, and Methods			
Array	Date	eval	function
hasOwnProperty	Infinity	isFinite	isNaN
isPrototypeOf	length	Math	NaN
name	Number	Object	prototype
String	toString	undefined	valueOf

HTML Names, Window Objects, and Properties			
alert	all	anchor	anchors
area	assign	blur	button
checkbox	clearInterval	clearTimeout	clientInformation
close	closed	confirm	constructor
crypto	decodeURI	decodeURIcomponent	defaultStatus
document	element	elements	embed
embeds	encodeURI	encodeURIcomponent	escape
event	fileUpload	focus	form
forms	frame	innerHeight	innerWidth
layer	layers	link	location
mimeTypes	navigate	navigator	frames
frameRate	hidden	history	image
images	offscreenBuffering	open	opener
option	outerHeight	outerWidth	packages
pageXOffset	pageYOffset	parent	parseFloat
parseInt	password	pkcs11	plugin
prompt	propertyIsEnum	radio	reset
screenX	screenY	scroll	secure
select	self	setInterval	setTimeout
status	submit	taint	text
textarea	top	unescape	untaint
window			

HTML Event Attributes For Example:			
onclick	ondblclick	onfocus	onfocusout
onkeydown	onkeypress	onkeyup	onload
onmousedown	onmouseup	onmouseover	onmouseout
onmousemove	onchange	onreset	onsubmit

Store Values

A "variable" is a container, common to every scripting and programming language, in which data can be stored and retrieved later. Unlike the "strongly typed" variables in most other languages, which must declare a particular data type they may contain, JavaScript variables are much easier to use because they are "loosely typed" – so they may contain any type of data:

A variable name is an alias for the value it contains – using the name in script references its stored value.

Data Type	Example	Description
String	**'Hello World!'**	A string of text characters
Number	**3.142**	An integer or floating-point number
Boolean	**true**	A true (1) or false (0) value
Object	**console**	A user-defined or built-in object
Function	**log( )**	A user-defined function, a built-in function, or an object method
Symbol	**Symbol( )**	A unique property identifier
null	**null**	Absolutely nothing (not even zero)
undefined	**undefined**	A non-configured property

Choose meaningful names for your variables to make the script easier to understand later.

A JavaScript variable can be declared using the **let**, **const**, or **var** keywords followed by a space and a name of your choosing. Variables declared with **let** can be reassigned new values as the script proceeds, whereas **const** (constant) does not allow this. The **var** keyword was used in JavaScript before the **let** keyword was introduced but is best avoided now as it does not prevent you declaring the same variable twice in the same context.

A **let** declaration of a variable in a script may simply create a variable to which a value can be assigned later, or may include an assignation to instantly "initialize" the variable with a value:

```
let myNumber                  // Declare a variable.
myNumber = 10                 // Initialize a variable.
let myString = 'Hello World!' // Declare and initialize a variable.
```

Multiple variables may be declared on a single line too:

```
let i , j , k                 // Declare 3 variables.
let num =10 , char = 'C'      // Declare and initialize 2 variables.
```

Constant variables must, however, be initialized when declared:

```
const myName = 'Mike'
```

Upon initialization, JavaScript automatically sets the variable type for the value assigned. Subsequent assignation of a different data type later in the script can be made to change the variable type. The current variable type can be revealed by the **typeof** keyword.

1 Create an HTML document with a script that declares several variables that are assigned different data types

```
const firstName = 'Mike'
const valueOfPi = 3.142
let isValid = true
let jsObject = console
let jsMethod = console.log
let jsSymbol = Symbol( )
let emptyVariable = null
let unusedVariable
```

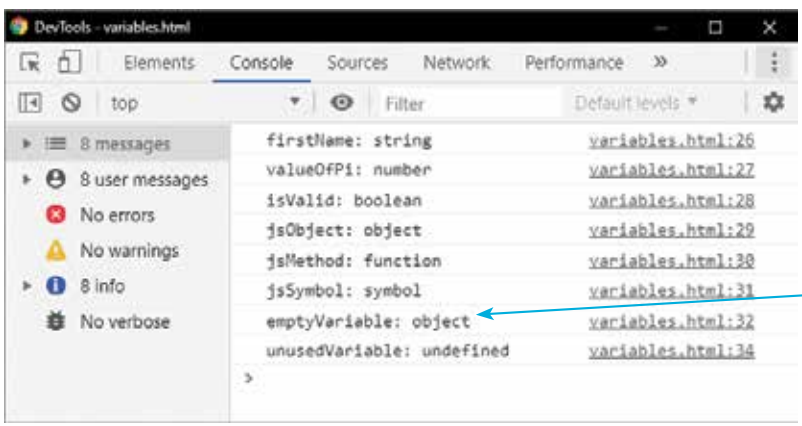
variables.html

2 Add statements to output the data type of each variable

```
console.log( 'firstName: ' + typeof firstName )
console.log( 'valueOfPi: ' + typeof valueOfPi )
console.log( 'isValid: ' + typeof isValid )
console.log( 'jsObject: ' + typeof jsObject )
console.log( 'jsMethod: ' + typeof jsMethod )
console.log( 'jsSymbol: ' + typeof jsSymbol )
console.log( 'emptyVariable: ' + typeof emptyVariable )
console.log( 'unusedVariable: ' + typeof unusedVariable )
```

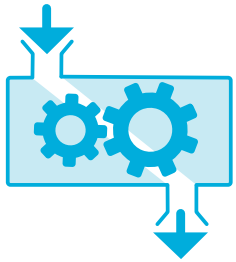
The concatenation + operator is used here to output a combined text string.

3 Save the HTML document then open it in your browser and launch the console to see the data types in output



You should be surprised to see that the variable assigned a **null** value is described as being an **object** type, rather than a **null** type. This is a known error in the JavaScript language.

Create Functions



A function expression is simply one, or more, statements that are grouped together in **{ }** curly brackets for execution, and it returns a final single value. Functions may be called as required by a script to execute their statements. Those functions that belong to an object, such as **console.log( )**, are known as "methods" – to differentiate them from built-in and user-defined functions. Both have trailing parentheses that may accept "argument" values to be passed to the function for manipulation – for example, an argument passed in the parentheses of the **console.log( )** method.

The number of arguments passed to a function must normally match the number of "parameters" specified within the parentheses of the function block declaration. For example, a user-defined function requiring exactly one argument looks like this:

```
function function-name ( parameter ) {
  // Statements to be executed go here.
}
```

Notice that the preferred format of a function declaration places the **{** opening curly bracket on the same line as the **function** keyword.

Multiple parameters can be specified as a comma-separated list and you can, optionally, specify a default value to be used when the function call does not pass an argument, like this:

```
function function-name ( parameter , parameter = value ) {
  // Statements to be executed go here.
}
```

You choose your own parameter names following the same naming conventions as for variable names. The parameter names can then be used within the function to reference the argument values passed from the parentheses of the function call.

You can omit the return statement, or use the **return** keyword without specifying a value, and the function will simply return an **undefined** value to the caller.

A function block can include a **return** statement so that script flow continues at the caller – no further statements in the function get executed. It is typical to finally return the result of manipulating passed argument values back to the caller:

```
function function-name ( parameter , parameter ) {
  // Statements to be executed go here.

  return result
}
```

It is common for statements within a function block to include calls to other functions – to modularize scripts into blocks.

1 Create an HTML document with a script that declares a function to return the squared value of a passed argument

```
function square ( arg ) {
  return arg * arg
}
```

functions.html

2 Next, add a function that returns the result of an addition

```
function add ( argOne, argTwo = 10 ) {
  return argOne + argTwo
}
```

3 Now, add a function that returns the result of squaring and an addition by calling each of the functions above

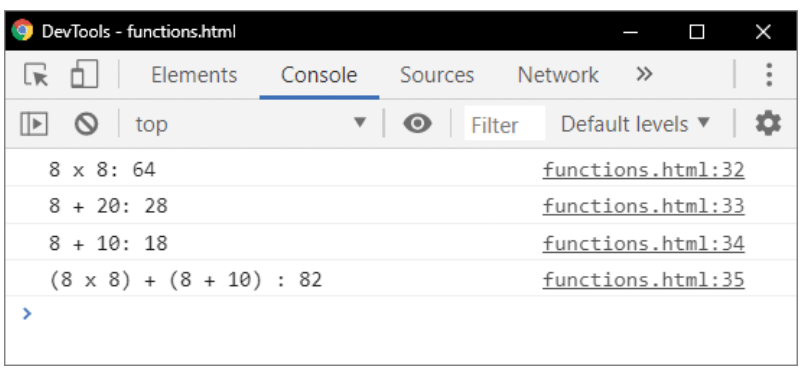
```
function squareAdd ( arg ) {
  let result = square( arg )
  return result + add( arg )
}
```

Notice that the default second parameter value (10) is used here when only one argument value is passed by the caller.

4 Finally, add statements that call the functions and print the returned values in output strings

```
console.log( '8 x 8: ' + square( 8 ) )
console.log( '8 + 20: ' + add( 8, 20 ) )
console.log( '8 + 10: ' + add( 8 ) )
console.log( '(8 x 8) + (8 + 10): ' + squareAdd( 8 ) )
```

5 Save the HTML document, then open it in your browser and launch the console to see values returned from functions



The * asterisk character is the arithmetical multiplication operator in JavaScript.

Assign Functions

When assigning a named function to a variable, only specify the function name in the statement.

Functions are really useful in JavaScript as they can be called ("invoked") to execute their statements whenever required, and the caller can pass different arguments to return different results.

It is important to recognize that the JavaScript **( )** parentheses operator is the component of the call statement that actually calls the function. This means a statement can assign a function to a variable by specifying just the function name. The variable can then be used to call the function in a statement that specifies the variable name followed by the **( )** operator. But beware, if you attempt to assign a function to a variable by specifying the function name followed by **( )** the function will be invoked and the value returned by that function will be assigned.

Function Hoisting

Variables that were declared using the older **var** keyword were also hoisted, but those declared with **let** or **const** are not hoisted.

Although scripts are read by the JavaScript interpreter in top-to-bottom order it actually makes two sweeps. The first sweep looks for function declarations and remembers any it finds in a process known as "hoisting". The second sweep is when the script is actually executed by the interpreter. Hoisting allows function calls to appear in the script before the function declaration, as the interpreter has already recognized the function on the first sweep. The first sweep does not, however, recognize functions that have been assigned to variables using the **let** or **const** keywords!

Anonymous Functions

When assigning a function to a variable, a function name can be omitted as the function can be called in a statement specifying the variable name and the **( )** operator. These are called anonymous function expressions, and their syntax looks like this:

let *variable* = function (*parameters*) { *statements* ; return *value* }

Self-invoking function expressions are also known as Immediately Invoked Function Expressions (IIFE – often pronounced "iffy").

Anonymous function expressions can also be made "self-invoking" by enclosing the entire function within **( )** parentheses and adding the **( )** parentheses operator at the end of the expression. This means that their statements are automatically executed one time when the script is first loaded by the browser. The syntax of a self-invoking function expression looks like this:

(function ( ) { *statements* ; return *value* }) ( )

Self-invoking functions are used widely throughout this book to execute example code when the script gets loaded.

1. Create an HTML document with a script that calls a function that has not yet been declared

```
console.log( 'Hoisted: ' + add( 100, 200 ) )
```

2. Next, add below the function that is called above

```
function add( numOne, numTwo ) {
  return numOne + numTwo
}
```

3. Now, add a function that assigns the function above to a variable, then calls the assigned function

```
let addition = add
console.log( 'Assigned: ' + addition( 32, 64 ) )
```

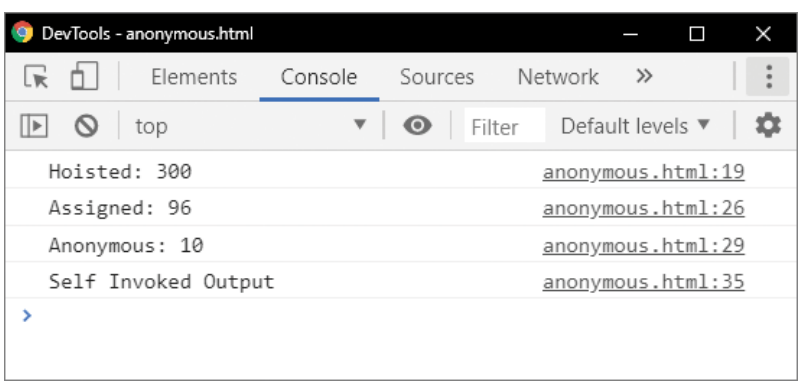
4. Then, assign a similar, but anonymous, function to a variable and call that assigned function

```
let anon = function ( numOne, numTwo ) {
  let result = numOne + numTwo ; return result
}
console.log( 'Anonymous: ' + anon( 9, 1 ) )
```

5. Finally, assign the value returned from a self-invoking function to a variable and display that value

```
let iffy = ( function ( ) {
  let str = 'Self Invoked Output' ; return str
} ) ( )
console.log( iffy )
```

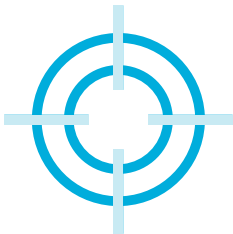
6. Save the HTML document, then open it in your browser and launch the console to see values returned from functions



anonymous.html

The significance of self-invoking functions may not be immediately obvious, but their importance should become clearer by the end of this chapter.

Recognize Scope



The extent to which variables are accessible in your scripts is determined by their "lexical scope" – the environment in which the variable was created. This can be either "global" or "local".

Global Scope

Variables created outside function blocks are accessible globally throughout the entire script. This means they exist continuously and are available to functions within the same script environment. At first glance this might seem very convenient, but it has a very serious drawback in that variables of the same name can conflict. For example, imagine that you have created a global **myName** variable that has been assigned your name, but then also include an external script in which another developer has created a global **myName** variable that has been assigned his or her name. Both like-named variables exist in the same script environment, so conflict. This is best avoided so you should not create global variables to store primitive values (all data types except Object and Function) within your scripts.

Local Scope

Variables created inside function blocks are accessible locally throughout the life of the function. They exist only while the function is executing, then they are destroyed. Their script environment is limited – from the point at which they are created, to the final **}** curly bracket, or the moment when the function returns. It is good practice to declare variables at the very beginning of the function block so their lexical scope is the duration of the function. This means that like-named variables can exist within separate functions without conflict. For example, a local **myName** variable can exist happily inside separate functions within your script and inside functions in included external scripts. It is recommended that you try to create only local variables to store values within your scripts.

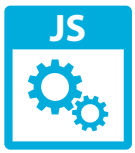
You will discover how to catch errors on pages 60-61.

Best Practice

Declaring global variables with the older **var** keyword allows like-named conflicting variables to overwrite their assigned values without warning. The more recent **let** and **const** keywords prohibit this and instead recognize the behavior as an "Uncaught SyntaxError". It is therefore recommended that you create variables declared using the **let** or **const** keywords to store values within your scripts.

1. Create an external script that calls a function to output the value of a global variable

```
let myName = 'External Script'
function readName( ) { console.log( myName ) }
readName( )
```



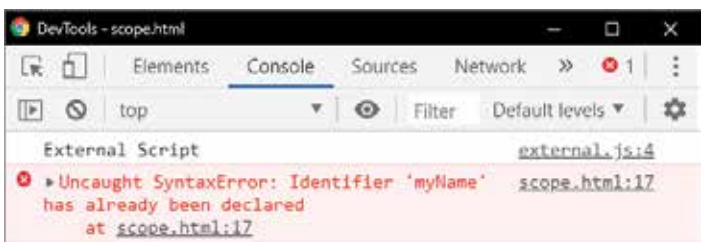
external.js

2. Create an HTML document that includes the external script and adds a similar script

```
<script src="external.js"></script>
<script>
let myName = 'Internal Script'
function getName( ) { console.log( myName ) }
getName( )
</script>
```

scope.html

3. Save both files in the same folder, then open the HTML document to see a conflict error reported in the console

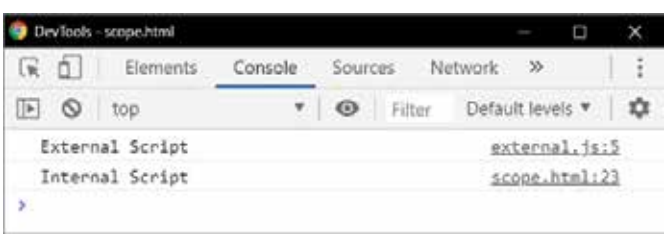


4. Edit both scripts to make the global variables into local variables then refresh the browser to see no conflict

```
function readName( ) {
  let myName = 'External Script' ; console.log( myName )
}

function getName( ) {
  let myName = 'Internal Script' ; console.log( myName )
}
```

The function calls **readName( )** and **getName( )** remain in the scripts without editing.



Use Closures

The previous example demonstrated the danger of creating global variables to store values in JavaScript, but sometimes you will want to store values that remain continuously accessible – for example, to remember an increasing score count as the script proceeds. How can you do this without using global variables to store primitive values? The answer lies with the use of "closures".

A closure is a function nested inside an outer function that retains access to variables declared in the outer function – because that is the lexical scope in which the nested function was created.

closure.html

1. Create an HTML document with a script that assigns a self-invoking anonymous function to a global variable

```
const add = ( function ( ) {
  // Statements to be inserted here.
} ) ( )
```

2. Next, insert statements to initialize a local variable and assign a function to a local variable in the same scope

```
let count = 0
const nested = function ( ) { return count = count + 1 }
```

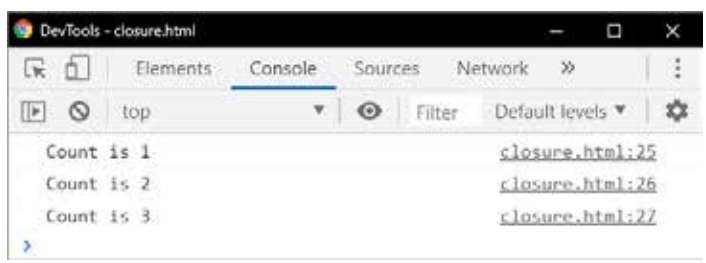
3. Now, insert a statement to return the inner function – assigning the inner function to the global variable

```
return nested
```

4. Finally, add three identical function calls to the inner function that is now assigned to the global variable

```
console.log( 'Count is ' + add( ) )
console.log( 'Count is ' + add( ) )
console.log( 'Count is ' + add( ) )
```

5. Save the HTML document, then open it in your browser and launch the console to see values returned from a closure



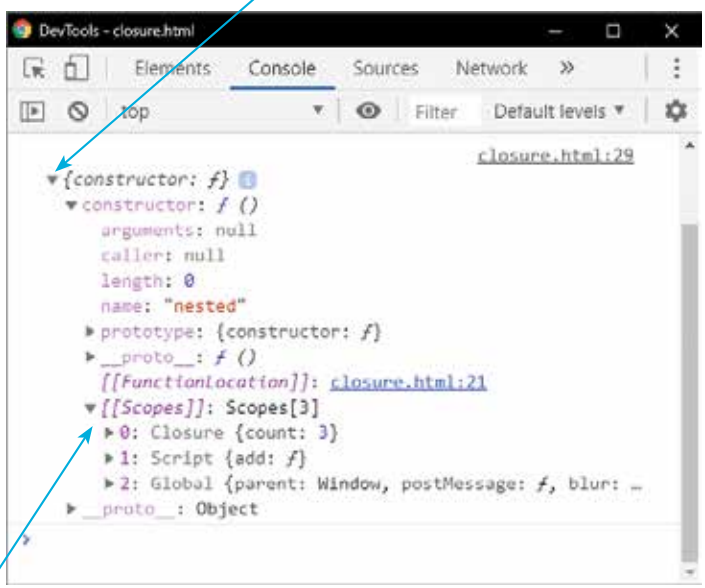
Self-invoking function expressions are described on page 20. They execute their statements one time only. Here, you can use **console.log(add)** to confirm that the function expression has been assigned to the outer variable.

It can be difficult to grasp the concept of closures, as it would seem that the **count** variable in this example should be destroyed when the self-invoking function has completed execution. In order to better understand how closures work, you can explore the **prototype** property of the assigned function.

6 Add a statement at the end of the script to reveal how the assigned function has been constructed internally

```
console.log( add.prototype )
```

7 Save the HTML document, then refresh the browser and expand the "constructor" drop-down to see the scopes



Closer inspection reveals that the assigned function has a special (Closure) scope in addition to the regular local (Script) scope and outer (Global) scope. This is how the **count** variable remains accessible via the assigned function yet, importantly, cannot be referenced in any other way.

The use of closures to hide persistent variables from other parts of your script is an important concept. It is similar to how "private" variables can be hidden in other programming languages and are only accessible via "getter" methods.

All JavaScript objects inherit properties and methods from a **prototype**. Standard JavaScript objects, such as functions, call an internal constructor function to create the object by defining its components.

Don't worry if you can't immediately understand how closures work. They can seem mystical at first, but will become clearer with experience. You can continue on and come back to this technique later.

Summary

- JavaScript code can be included in an HTML document directly or from an external file using **<script> </script>** tags.
- JavaScript can display output in an HTML element in an alert dialog box or in the browser's console window.
- JavaScript statements may contain keywords, operators, values, and expressions.
- The JavaScript interpreter ignores tabs and spaces.
- JavaScript statements can be grouped in **{ }** curly bracket function blocks that can be called to execute when required.
- Variable and function names may comprise letters, numbers, and underscore characters, but must avoid keywords.
- JavaScript variables may contain data types of String, Number, Boolean, Object, Function, Symbol, **null**, and **undefined**.
- Variables declared with the **let** keyword can be reassigned new values, but the **const** keyword does not allow this.
- A function expression has statements grouped in **{ }** curly brackets for execution, and it returns a final single value.
- The **( )** parentheses of a function expression may contain parameters for argument values to be passed from the caller.
- A function block can include a **return** statement to specify data to be passed back to the caller.
- The JavaScript **( )** parentheses operator calls the function.
- Hoisting allows function calls to appear in the script before the function declaration.
- Anonymous function expressions have no function name.
- Lexical scope is the environment in which the variable was created and can be global, local, or closure.
- Local variables should be used to store values, but global variables can be assigned functions to create closures.
- A closure is a function nested within an outer function that retains access to variables declared in the outer function.