Fundamentals of Web Development

Third Edition by Randy Connolly and Ricardo Hoar

Chapter 8

JavaScript 1:

Language Fundamentals

Pearson

In this chapter you will learn . . .

- About JavaScript's role in contemporary web development
- How to add JavaScript code to your web pages
- The main programming constructs of the language
- The importance of objects and arrays in JavaScript
- How to use functions in JavaScript

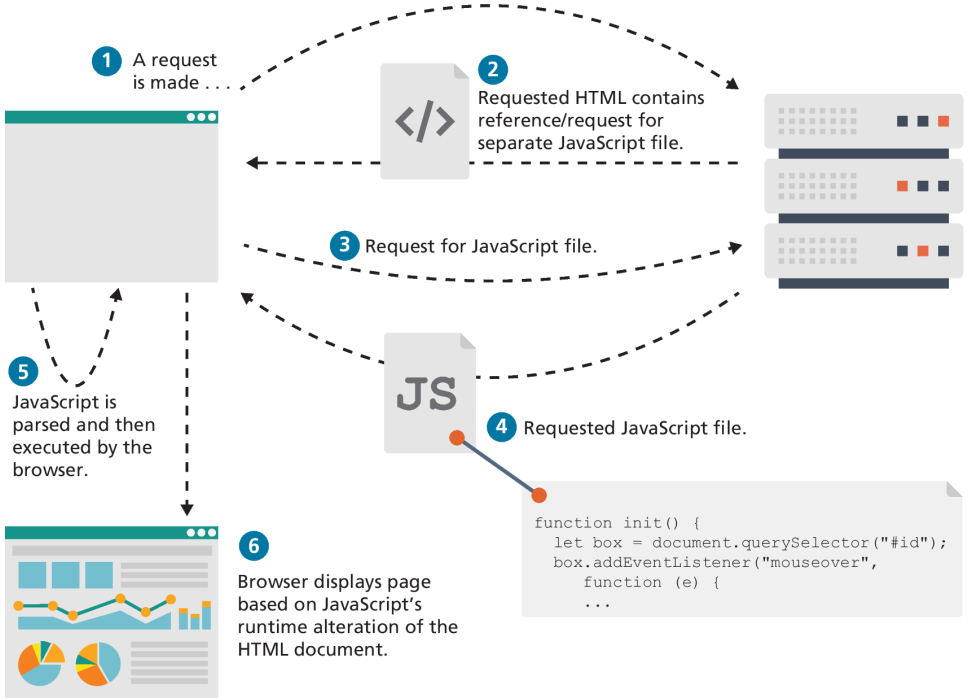
What Is JavaScript and What Can It Do?

- JavaScript: it is an object-oriented, dynamically typed scripting language
- *primarily* a client-side scripting language as well.
- variables are objects in that they have properties and methods
- Unlike more familiar object-oriented languages Such as Java, C#, and C++, functions in JavaScript are also objects.
- JavaScript is dynamically typed (also called weakly typed) in that variables can be easily (or implicitly) converted from one data type to another.

Client-Side Scripting

Client-side scripting refers to the client machine (i.e., the browser) running code locally rather than relying on the server to execute code and return the result.

A client machine downloads and executes JavaScript code



Client-Side Scripting: Advantages

- Processing can be off-loaded from the server to client machines, thereby reducing the load on the server.
- The browser can respond more rapidly to user events than a request to a remote server ever could, which improves the user experience.
- JavaScript can interact with the downloaded HTML in a way that the server cannot, creating a user experience more like desktop software than simple HTML ever could.

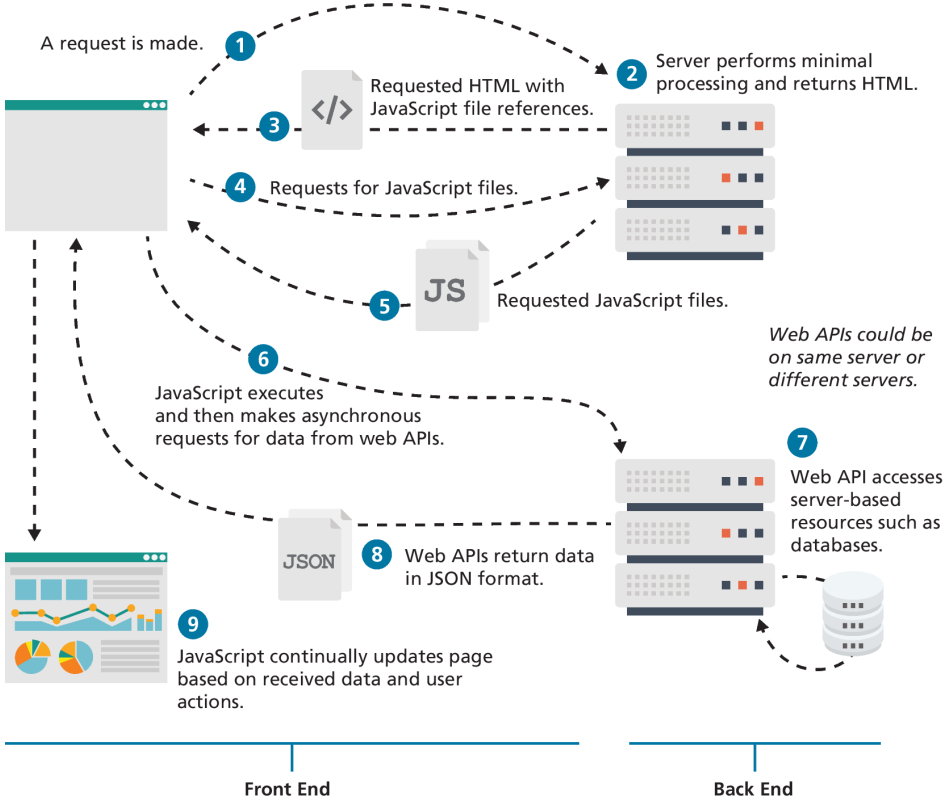
Client-Side Scripting: Disadvantages

- There is no guarantee that the client has JavaScript enabled, meaning any required functionality must be implemented redundantly on the server.
- JavaScript-heavy web applications can be complicated to debug and maintain.
- JavaScript is not fault tolerant. Browsers are able to handle invalid HTML or CSS. But if your page has invalid JavaScript, it will simply stop execution at the invalid line.
- While JavaScript is universally supported in all contemporary browsers, the language (and its APIs) is continually being expanded. As such, newer features of the language may not be supported in all browsers.

JavaScript's History

- JavaScript was introduced by Netscape in their Navigator browser back in 1996.
- Netscape submitted JavaScript to Ecma International in 1997, **ECMAScript** is simultaneously a superset and subset of the JavaScript programming language.
- The Sixth Edition (or **ES6**) was the one that introduced many notable new additions to the language (such as classes, iterators, arrow functions, and promises)
- The latest version of ECMAScript is the Tenth Edition (generally referred to as ES11 or ES2020)

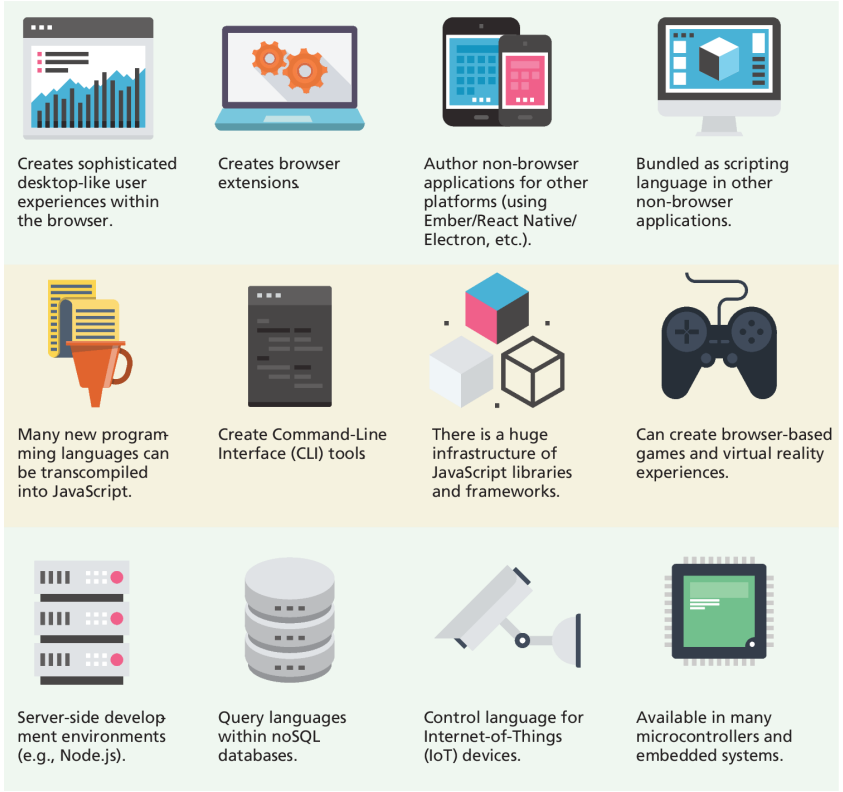
JavaScript and Web 2.0



JavaScript in Contemporary Software Development

JavaScript's role has expanded beyond the constraints of the browser

- It can be used as the language within server-side runtime environments such as Node.js.
- MongoDB use JavaScript as their query language
- Adobe Creative Suite and OpenOffice use JavaScript as their end-user scripting language
- ....



Where Does JavaScript Go?

Just as CSS styles can be inline, embedded, or external, JavaScript can be included in a number of ways.

- **Inline JavaScript** refers to the practice of including JavaScript code directly within some HTML element attributes.
- Embedded JavaScript refers to the practice of placing JavaScript code within a <script> element
- The recommended way to use JavaScript is to place it in an external file. You do this via the <script> tag

Adding JavaScript to a page

```
<html lang="en">
<head>
  <title>JavaScript placement possibilities</title>
  <script>
    /* A JavaScript Comment */
    alert("This will appear before any content");
  </script>

  <script src="greeting.js"></script>
</head>
<body>
<h1>Page Title</h1>

<a href="JavaScript:OpenWindow();">for more info</a>
<input type="button" onClick="alert('Are you sure?');" />

<script>
   alert("Hello World");
</script>
```

Embedded JavaScript

External JavaScript

Inline JavaScript

Embedded JavaScript

Users without JavaScript

Users have a myriad of reasons for not using JavaScript

- Search engines
- Browser extensions
- Text browser
- Accessible browser

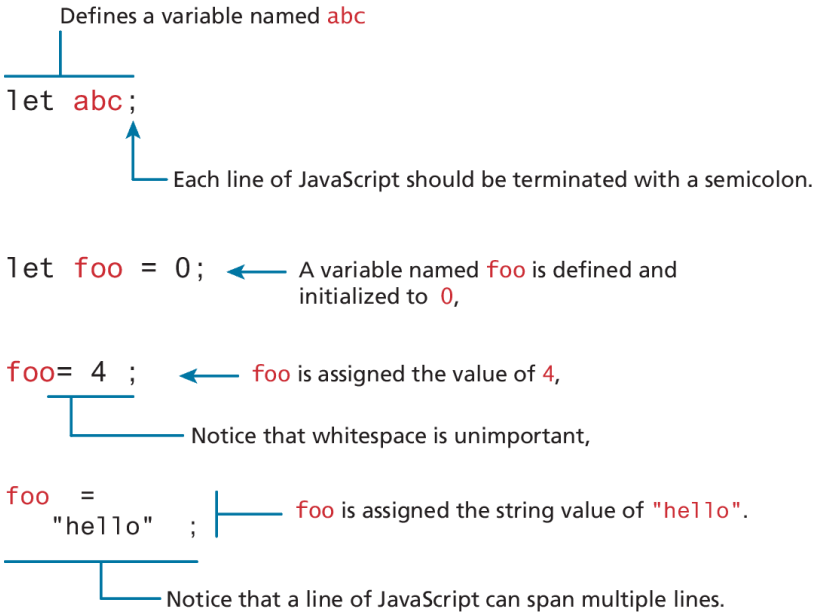
HTML provides an easy way to handle users who do not have JavaScript enabled: the <noscript> element. Any text between the opening and closing tags will only be displayed to users without the ability to load JavaScript.

Variables and Data Types

Variables in JavaScript are **dynamically typed**, meaning that you do not have to declare the type of a variable before you use it.

To declare a variable in JavaScript, use either the **var**, **const**, or **let** keywords (see Table 8.1)

Assignment can happen at declaration time by appending the value to the declaration, or at runtime with a simple right-to-left assignment

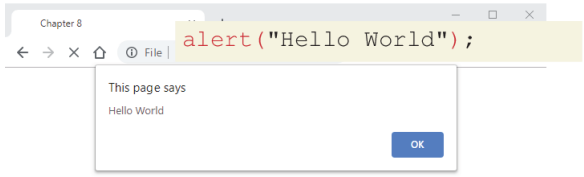


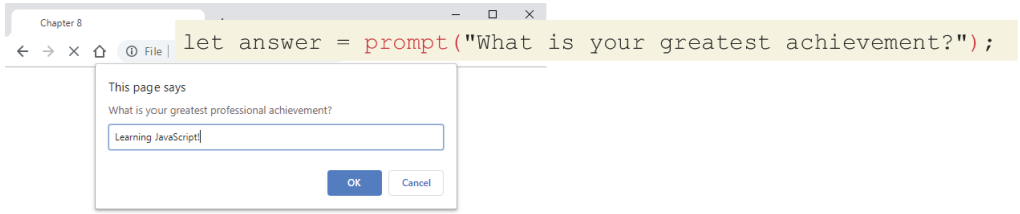
JavaScript Output

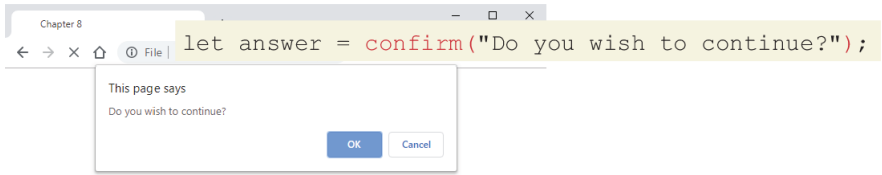
alert() Displays content within a browser-controlled pop-up/modal window.

prompt() Displays a message and an input field within a modal window.

confirm() Displays a question in a modal window with ok and cancel buttons.



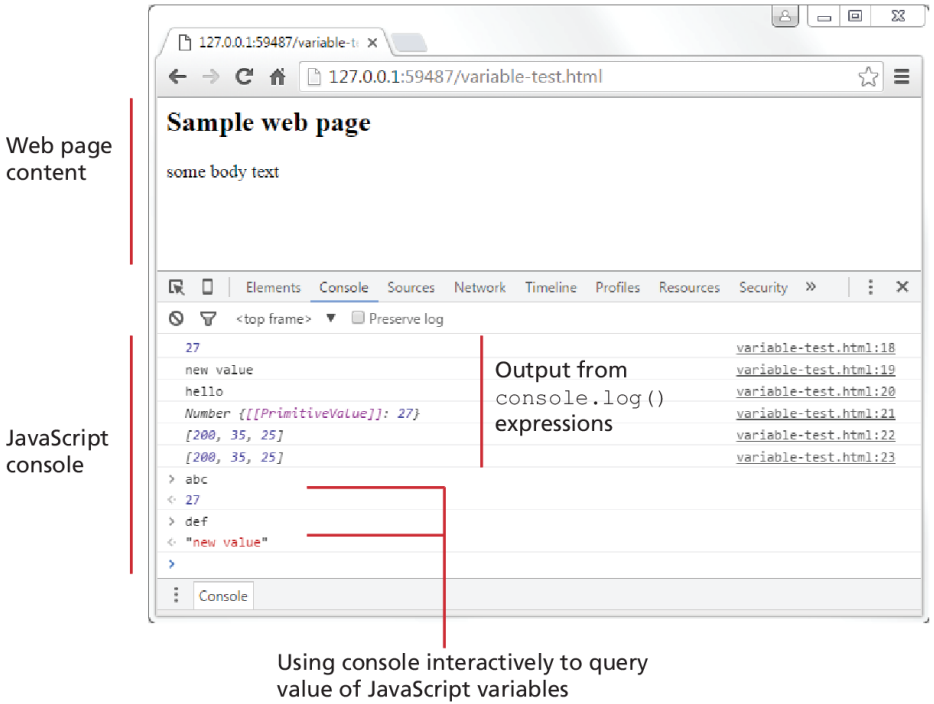




JavaScript Output (ii)

- **document.write()** Outputs the content (as markup) directly to the HTML document.

- **console.log()** Displays content in the browser's JavaScript console.



Document.write() note

NOTE

While several of the examples in this chapter make use of **document.write(),** the usual (and more trustworthy) way to generate content that you want to see in the browser window will be to use the appropriate JavaScript DOM (Document Model Object) method. You will learn how to do that in the next chapter.

Data Types

JavaScript has two basic data types:

- **reference types** (usually referred to as objects)
- **primitive types** (i.e., nonobject, simple types).
  - What makes things a bit confusing for new JavaScript developers is that the language lets you use primitive types as if they are objects.

Primitive Types

- **Boolean** True or false value.
- **Number** Represents some type of number. Its internal format is a double precision 64-bit floating point value.
- **String** Represents a sequence of characters delimited by either the single or double quote characters.
- **Null** Has only one value: **null**.
- **Undefined** Has only one value: **undefined**. This value is assigned to variables that are not initialized. Notice that undefined is different from null.
- **Symbol** New to ES2015, a symbol represents a unique value that can be used as a key value.

Primitive vs Reference Types

Primitive variables contain the value of the primitive directly within memory.

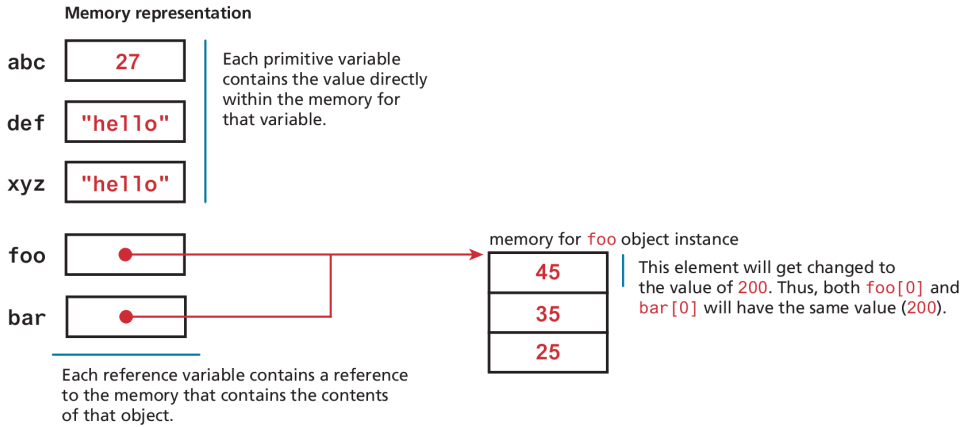
In contrast, object variables contain a reference or pointer to the block of memory associated with the content of the object.

```
let abc = 27;
let def = "hello";
```
variables with primitive types

```
let foo = [45, 35, 25];
```
variable with reference type (i.e., array object)

```
let xyz = def;
let bar = foo;
```
these new variables differ in important ways (see below)

```
bar[0] = 200;
```
changes value of the first element of array



Let vs const

All of these let examples work with no errors.

```
let abc = 27;
abc = 35;

let message = "hello";
message = "bye";

let msg = "hello";
msg = "hello";

let foo = [45, 35, 25];
foo[0] = 123;
foo[0] = "this is ok";

let person = {name: "Randy"};
person.name = "Ricardo";

person = {};
```

Some of these const examples work won't work, but some will work.

```
const abc = 27;
abc = 35;
```
Will generate runtime exception, since you cannot reassign a value defined with const.

```
const message = "hello";
message = "bye";
```
Will generate runtime exception.

```
const msg = "hello";
msg = "hello";
```
Will generate runtime exception.

```
const foo = [45, 35, 25];
foo[0] = 123;
foo[0] = "this is also ok";
```
You are allowed to change elements of an array, even if defined with a const keyword.

```
const person = {name: "Randy"};
person.name = "Ricardo";
```
Allowed to change properties of an object.

```
person = {};
```
Will generate runtime exception.

Built-In Objects

JavaScript has a variety of objects you can use at any time, such as arrays, functions, and the **built-in objects**.

Some of the most commonly used built-in objects include **Object**, **Function**, **Boolean**, **Error**, **Number**, **Math**, **Date**, **String**, and **Regexp**.

Later we will also frequently make use of several vital objects that are not part of the language but are part of the browser environment. These include the **document**, **console**, and **window** objects.

```
let def = new Date();
// sets the value of abc to a string containing the current date
let abc = def.toString();
```

Concatenation

To combine string literals together with other variables. Use the concatenate operator (+).

Alternative technique for concatenation is **template literals** (listing 8.2)

```
const country = "France";
const city = "Paris";
const population = 67;
const count = 2;

let msg = city + " is the capital of " + country;
msg += " Population of " + country + " is " + population;

let msg2 = population + count;

// what is displayed in the console?

console.log(msg);
//Paris is the capital of France Population of France is 67

console.log(msg2);
// 69
```

LISTING 8.1 Using the concatenate operator

Conditionals

JavaScript's syntax for conditional statements is almost identical to that of PHP, Java, or C++.

In this syntax the condition to test is contained within () brackets with the body contained in {} blocks. Optional **else if** statements can follow, with an **else** ending the branch.

JavaScript has all of the expected comparator operators (<, >, ==, <=, >=, !=, !==, ===) which are described in Table 8.4.

Switch statement

The **switch** statement is similar to a series of **if...else** statements.

There is another way to make use of conditionals: the **conditional operator** (also called the **ternary operator**).

```
switch (artType) {
   case "PT":
      output = "Painting";
      break;
   case "SC":
      output = "Sculpture";
      break;
   default:
      output = "Other";
}
// equivalent
if (artType == "PT") {
   output = "Painting";
} else if (artType == "SC") {
   output = "Sculpture";
} else {
   output = "Other";
}
```

LISTING 8.4 Conditional statement using switch and an equivalent if-else

The conditional (ternary) operator

```
/* conditional (ternary) assignment */
foo = (y==4) ? "y is 4" : "y is not 4";
```

Condition — Value if true — Value if false

```
/* equivalent to */
if (y==4) {
  foo = "y is 4";
}
else {
  foo = "y is not 4";
}
```

```
let tip  = isLargeGroup ? 0.25 : 0.15;
```

```
/* equivalent to */
let tip;
if (isLargeGroup) {
   tip = 0.25;
}
else {
   tip = 0.15;
}
```

```
let price  = isChild ? 5 : isSenior ? 7 : 9;
```

```
/* equivalent to */
let price;
if (isChild)
   price = 5;
else if (isSenior)
   price = 7;
else
   price = 9;
```

Pearson

Truthy and Falsy

Everything in JavaScript has an inherent Boolean value.

In JavaScript, a value is said to be **truthy** if it translates to true, while a value is said to be **falsy** if it translates to false.

All values in JavaScript are truthy except false, null, "", '', 0, NaN, and undefined

While and do . . . while Loops

While and **do…while** loops execute nested statements repeatedly as long as the while expression evaluates to true.

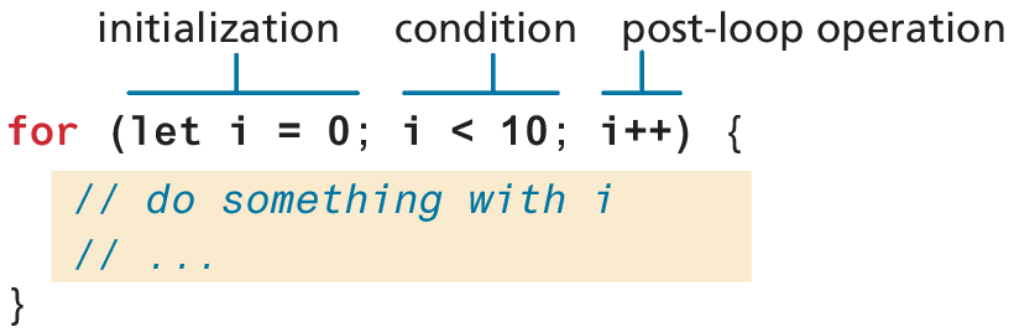
As you can see from this example, while loops normally initialize a **loop control variable** before the loop, use it in the condition, and modify it within the loop.

```
let count = 0;
while (count < 10) {
   // do something
   // ...
   count++;
}

count = 0;
do {
   // do something
   // ...
   count++;
} while (count < 10);
```

LISTING 8.5 While Loops

For Loops

For loops combine the common components of a loop—initialization, condition, and postloop operation—into one statement. This statement begins with the **for** keyword and has the components placed within () brackets, and separated by semicolons (;)



Infinite Loop Note

NOTE

Infinite loops can happen if you are not careful, and since the scripts are executing on the client computer, it can appear to them that the browser is “locked” while endlessly caught in a loop, processing.

Some browsers will even try to terminate scripts that execute for too long a time to mitigate this unpleasantness.

Try…catch

DIVE DEEPER

When the browser's JavaScript engine encounters a runtime error, it will throw an **exception**. These exceptions interrupt the regular, sequential execution of the program and can stop the JavaScript engine altogether. However, you can optionally catch these errors (and thus prevent the disruption) using the **try . . . catch block** as shown below.

```
try {
nonexistantfunction("hello");
}
catch(err) {
alert ("An exception was caught:" + err);
}
```

try...catch can also be used to your own error messages.

Arrays

Arrays are one of the most commonly used data structures in programming.

JavaScript provides two main ways to define an array.

First approach is **Array literal notation**, which has the following syntax:

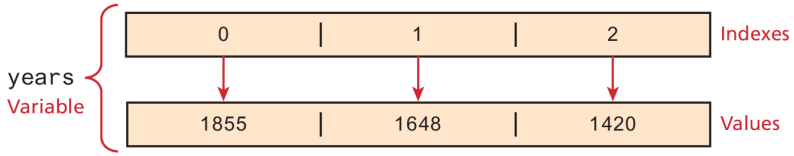
- const name = [*value1*, *value2*, ...];

The second approach is to use the Array() constructor:

- const name = new Array(*value1*, *value2*, ...);

Array example

```
const years = [1855, 1648, 1420];


const countries = ["Canada", "France",
"Germany", "Nigeria",
"Thailand", "United States"];

// arrays can also be multi-dimensional ... notice the commas!
const twoWeeks = [
["Mon","Tue","Wed","Thu","Fri"],
["Mon","Tue","Wed","Thu","Fri"]
];
// JavaScript arrays can contain different data types
const mess = [53, "Canada", true, 1420];
```



LISTING 8.6 Creating arrays using array literal notation

Iterating an array using for . . . of

ES6 introduced an alternate way to iterate through an array, known as the for...of loop, which looks as follows.

```
// iterating through an array
for (let yr of years) {
    console.log(yr);
}
```

```
//functionally equivalent to
for (let i = 0; i < years.length; i++) {
    let yr = years[i];
    console.log(yr);
}
```

Array Destructuring

Let's say you have the following array:

```
const league = ["Liverpool", "Man City", "Arsenal", "Chelsea"];
```

Now imagine that we want to extract the first three elements into their own variables. The "old-fashioned" way to do this would look like the following:

```
let first = league[0];
let second = league[1];
let third = league[2];
```

By using array destructuring, we can create the equivalent code in just a single line:

```
let [first,second,third] = league;
```

Objects

We have already encountered a few of the built-in objects in JavaScript, namely, arrays along with the Math, Date, and document objects.

In this section, we will learn how to create our own objects and examine some of the unique features of objects within JavaScript.

In JavaScript, **objects** are a collection of named values (which are called **properties** in JavaScript).

Unlike languages such as C++ or Java, objects in JavaScript are *not* created from classes. JavaScript is a prototype based language, in that new objects are created from already existing prototype objects, an idea that we will examine in Chapter 10.

Object Creation Using Object Literal Notation

The most common way is to use **object literal notation** (which we also saw earlier with arrays)

An object is represented by a list of key-value pairs with colons between the key and value, with commas separating key-value pairs.

To reference this object's properties, we can use either dot notation or square bracket notation.

```
const objName = {
    name1: value1,
    name2: value2,
    // ...
    nameN: valueN
};
```

```
objName.name1
objName["name1"]
```

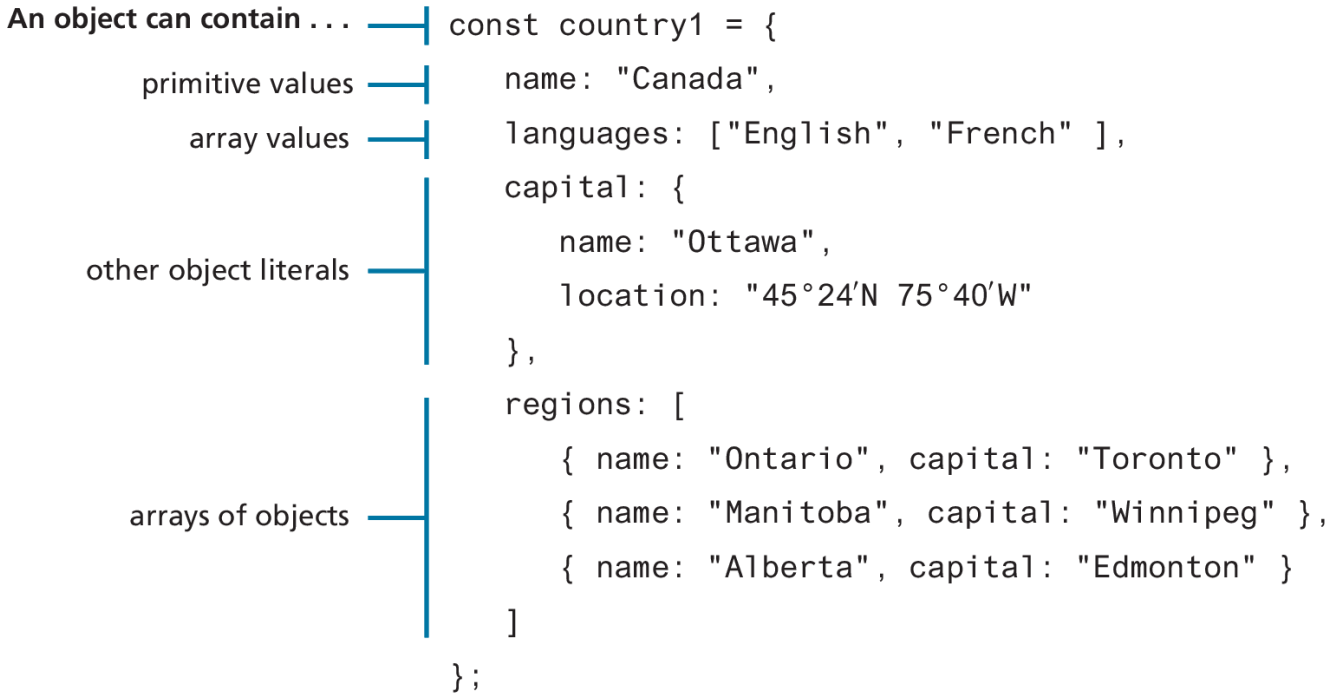
Object Creation Using Object Constructor

Another way to create an instance of an object is to use the Object constructor, as shown in the following:

```
// first create an empty object
const objName = new Object();

// then define properties for this object
objName.name1 = value1;
objName.name2 = value2;
```

Generally speaking, object literal notation is preferred in JavaScript over the constructed form.

Objects containing other content



Object Destructuring

Just as arrays can be destructured, so too can objects.

Let’s use the following object literal definition.

```
const photo = {
    id: 1,
    title: "Central Library",
    location: {
        country: "Canada",
        city: "Calgary"
    }
};
```

Object Destructuring (ii)

One can extract out a given property using dot or bracket notation as follows.

```
let id = photo.id;
let title = photo["title"];

let country = photo.location.country;
let city = photo.location["country"];
```

Equivalent assignments using object destructuring syntax would be:

```
let { id,title } = photo;
let { country,city } = photo.location;
```

These two statements could be combined into one:

```
let { id, title, location: {country,city} } = photo;
```

Spread

You can also make use of the spread syntax to copy contents from one array into another. Using the **photo** object from the previous section, we could copy some of its properties into another object using spread syntax:

```
const foo = { name:"Bob", .. .photo.location, iso:"CA" };
```

Is equivalent to:

```
const foo = { name:"Bob", country:"Canada", city:"Calgary", iso:"CA"};
```

It should be noted that this is a **shallow copy**, in that primitive values are copied,but for object references, only the references are copied.

JSON

JavaScript Object Notation or JSON is used as a language-independent data interchange format analogous in use to XML.

The main difference between JSON and object literal notation is that property names are enclosed in quotes, as shown in the following example:

```
// this is just a string though it looks like an object literal
const text = '{ "name1" : "value1",
   "name2" : "value2",
   "name3" : "value3"
}';
```

JSON object

The string literal on the last slide contains an object definition in JSON format (but is still just a string). To turn this string into an actual JavaScript object requires using the built-in JSON object.

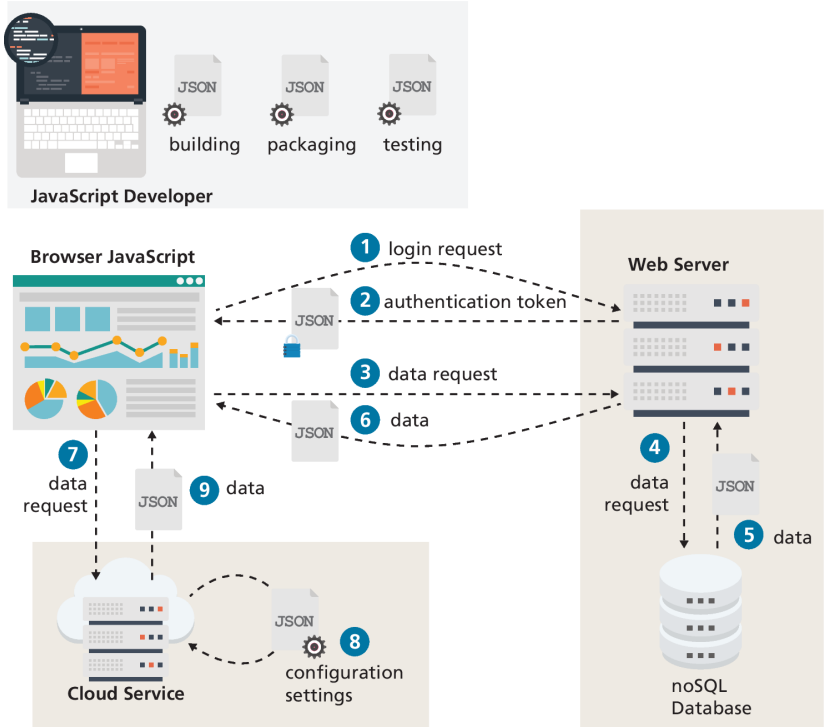
```
// this turns the JSON string into an object
const anObj = JSON.parse(text);
// displays "value1"
console.log(anObj.name1);
```

JSON in contemporary web development

JSON is encountered frequently in contemporary web development.

It is used by developers as part of their workflow, and most importantly, many web applications receive JSON from other sources, like other programs or websites.

This ability to interact with other web-based programs or sites will be covered in more detail in Chapter 10



Functions

Functions are defined by using the reserved word **function** and then the function name and (optional) parameters.

Functions do not require a return type, nor do the parameters require type specifications.

Declaring and calling functions

A function to calculate a subtotal as the price of a product multiplied by the quantity might be defined as follows:

```
function subtotal(price,quantity) {
    return price * quantity;
}
```

The above is formally called a **function declaration**. Such a declared function can be called or *invoked* by using the () operator.

```
let result = subtotal(10,2);
```

Function expressions

The object nature of functions can be further seen in the next example, which creates a function using a **function expression**.

When we invoked the function via the object variable name it is conventional to leave out the function name for so called **anonymous functions**

```
// defines a function using an anonymous function expression
const calculateSubtotal = function (price,quantity) {
                return price * quantity;
};

// invokes the function
let result = calculateSubtotal(10,2);

// define another function
const warn = function(msg) { alert(msg); };

// now invoke that function
warn("This doesn't return anything");
```

LISTING 8.11 Sample function expressions

Default Parameters

In the following code, what will happen (i.e., what will bar be equal to)?

```
function foo(a,b) {
    return a+b;
}
let bar = foo(3);
```

The answer is **NaN**. However, there is a way to specify **default parameters**

```
function foo(a=10,b=0) { return a+b; }
```

Now **bar** in the above example will be equal to 3.

Rest Parameters

How to write a function that can take a variable number of parameters?

The solution is to use the **rest operator** (...)

The concatenate method takes an indeterminate number of string parameters separated by spaces.

```
function concatenate(...args) {
        let s = "";
        for (let a of args)
                s += a + " ";
        return s;
}
let girls =
concatenate("fatima","hema","jane","alilah");
let boys = concatenate("jamal","nasir");

console.log(girls); //  "fatima hema jane alilah“

console.log(boys); // "jamal nasir“
```

Nested Functions

The object nature of functions can be further seen in the next example, which creates a function using a **function expression**.

When we invoked the function via the object variable name it is conventional to leave out the function name for so called **anonymous functions**

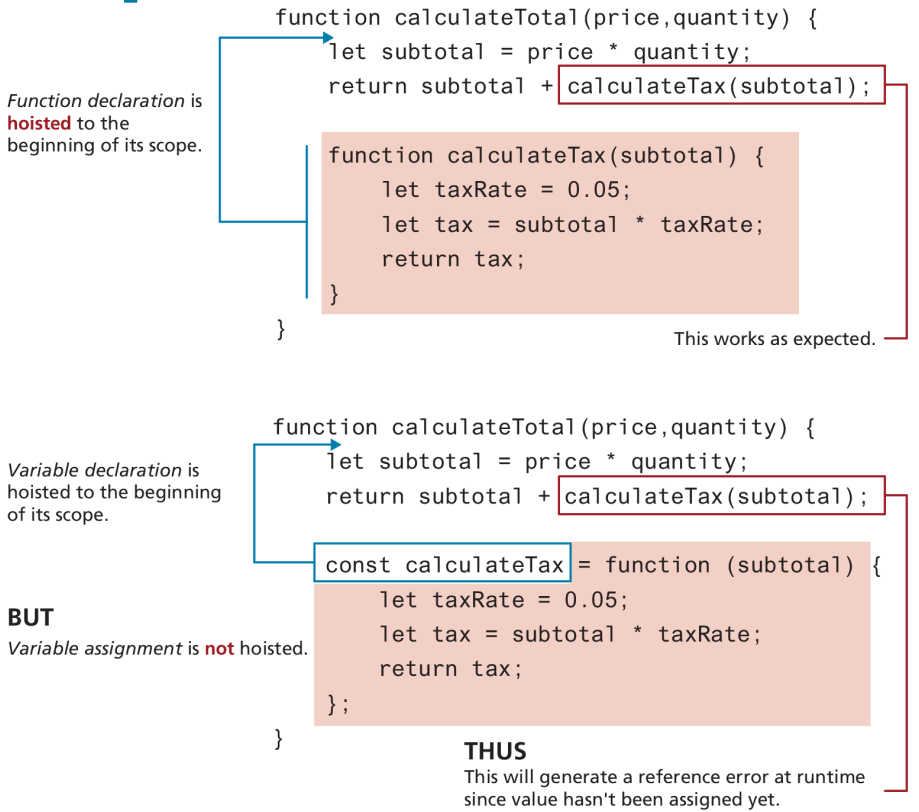
```
function calculateTotal(price,quantity) {
   let subtotal = price * quantity;
   return subtotal + calculateTax(subtotal);

   // this function is nested
   function calculateTax(subtotal) {
      let taxRate = 0.05;
      return subtotal * taxRate;
   }
}
```

LISTING 8.12 Nesting functions

Hoisting in JavaScript

JavaScript function declarations are *hoisted* to the beginning of their current level

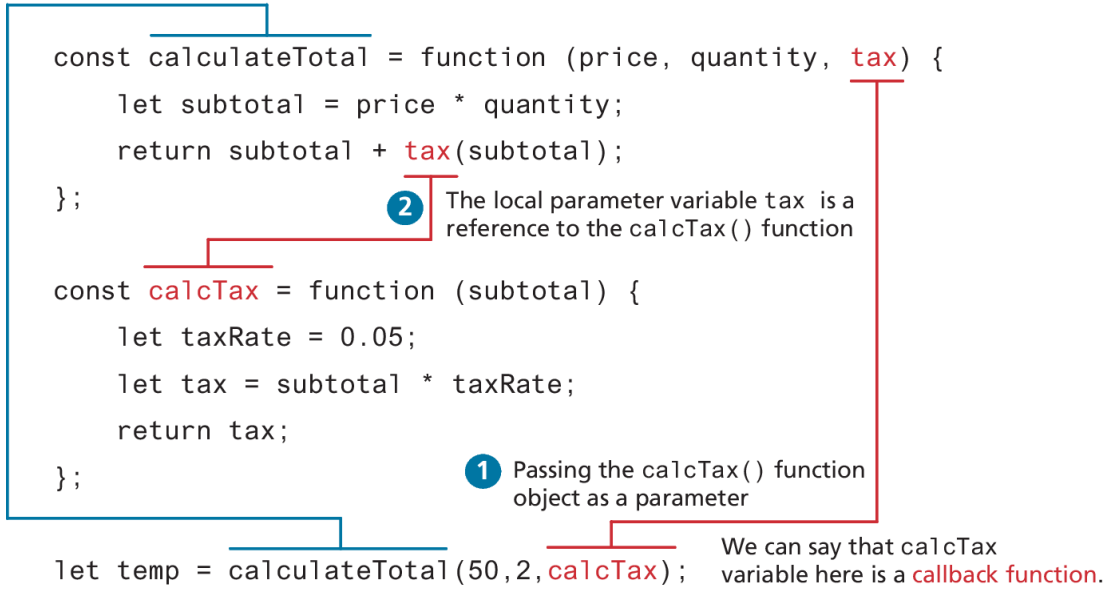
Note: the assignments are NOT hoisted.



Callback Functions

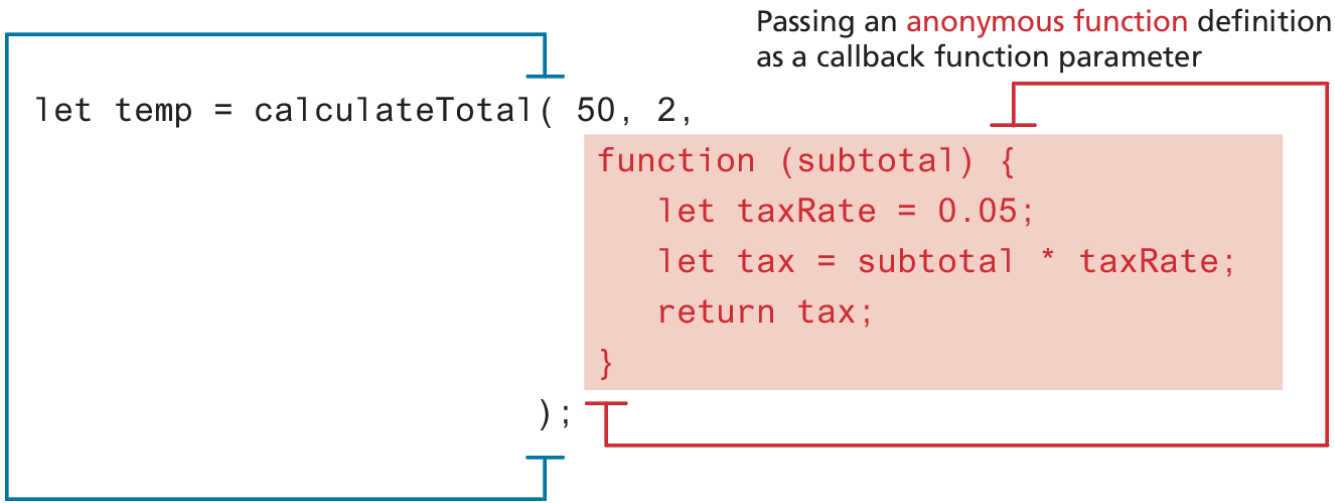
Since JavaScript functions are full-fledged objects, you can pass a function as an argument to another function.

Callback function is simply a function that is passed to another function.



Callback Functions (ii)

We can actually define a function definition directly within the invocation



Objects and Functions Together

In a class-oriented programming language like Java, we say classes define behavior via **methods**.

In a functional programming language like JavaScript, we say objects have properties that are **functions**.

Note the use of the keyword *this* in the two functions

```
const order ={
   salesDate : "May 5, 2016",
   product : {
      price: 500.00,
      brand: "Acer",
      output: function () {    return this.brand + ' $' + this.price; }
   },
   customer : {
      name: "Sue Smith",
      address: "123 Somewhere St",
      output: function () {return this.name + ', ' + this.address; }
   }
};

alert(order.product.output());
alert(order.customer.output());
```

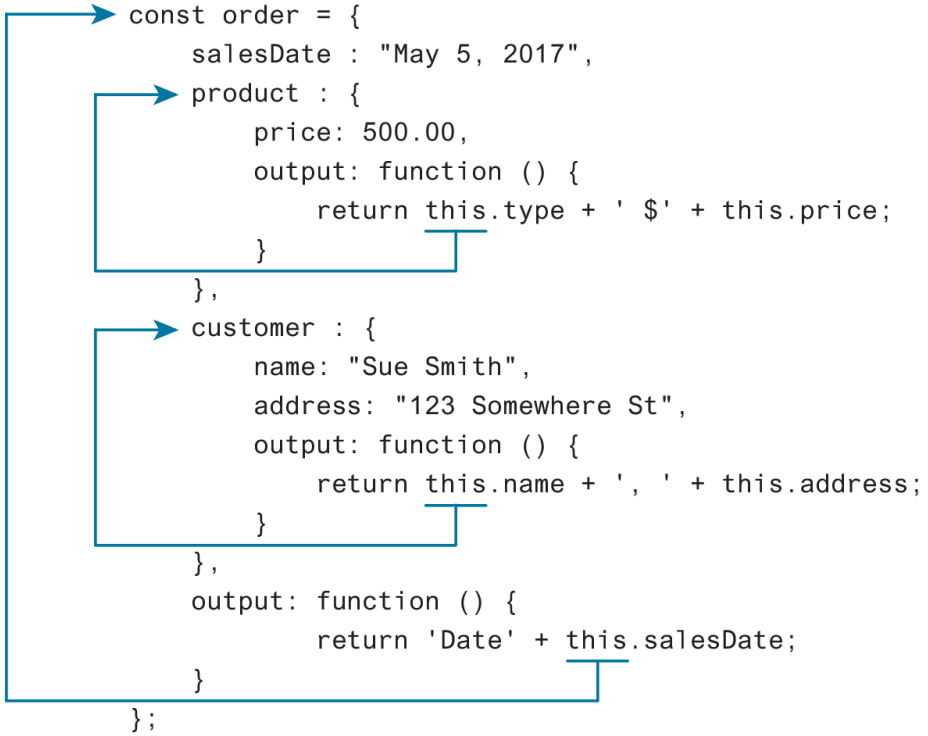
LISTING 8.13 Objects with functions

Contextual meaning of the this keyword

Note the use of the keyword *this* in the two functions

this is normally contextual and sometimes requires a full understanding of the current state

Luckily right now, we don't have to do anything so complex to understand the *this* in the example



Function constructors

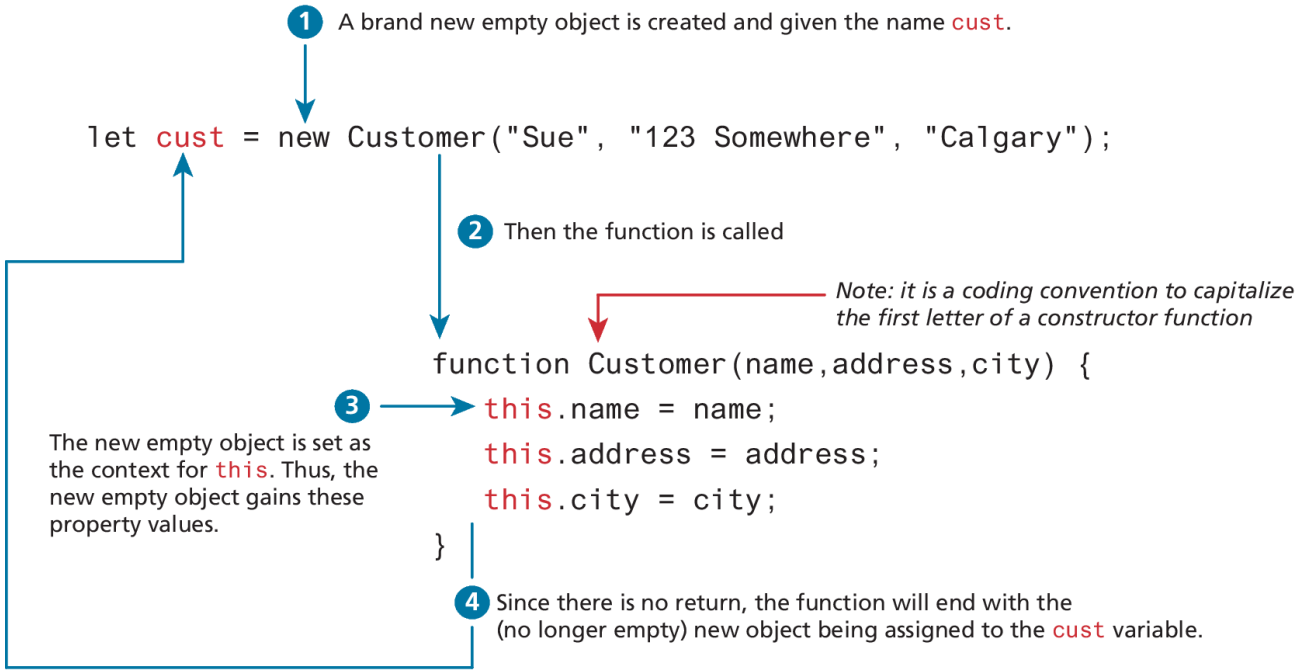
Looks similar to the approach used to create instances of objects in a class-based language like Java

The key difference between using a function constructor and using a regular function resides in the use of the **new** keyword before the function name.

```
// function constructor
function Customer(name,address,city) {
    this.name = name;
    this.address = address;
    this.city = city;
    this.output = function () {
        return this.name + " " + this.address + " " + this.city;
    };
}

// create instances of object using function constructor
const cust1 = new Customer("Sue", "123 Somewhere", "Calgary");
alert(cust1.output());
const cust2 = new Customer("Fred", "32 Nowhere St", "Seattle");
alert(cust2.output());
```

LISTING 8.14 Defining and using a function constructor

What happens with a constructor call of a function



Arrow Syntax

Arrow syntax provide a more concise syntax for the definition of anonymous functions. They also provide a solution to a potential scope problem encountered with the **this** keyword in callback functions.

To begin, consider an example of a simple function expression.

```
const taxRate = function () { return 0.05; };
```

The arrow function version would look like the following:

```
const taxRate = () => 0.05;
```

As you can see, this is a pretty concise (but perhaps confusing) way of writing code.

Array syntax overview

Traditional Syntax	Arrow Syntax	

`function () {`
`  statements`
`}`
	`() => {`
	`  statements`
	`}`
		Multi-line function,
		no parameters:
		{}, () required

`function (a,b) {`
`  statements`
`}`
	`(a,b) => {`
	`  statements`
	`}`
		Multi-line function,
		multiple parameters:
		() required

`function () {`
`  doSomething();`
`}`
	`() => {`
	`  doSomething();`
	`}`
		Single-line function,
		no return:
		{} required

`function (a) {`
`  return value;`
`}`
	`(a) => return value`	Single-line function,
		with return:
		{} optional

`function (a) {`
`  return value;`
`}`
	`a => value`	Single-line function,
		with return + one parameter:
		{}, (), return optional

Traditional Syntax	Arrow Syntax	

`function () {`
`  return value;`
`}`
	`() => value`	Single-line function,
		with return + no parameters:
		{}, return optional
		() required

`function (a,b) {`
`  return value;`
`}`
	`(a,b) => value`	Single-line function,
		with return + multiple parameters:
		{}, return optional
		() required

`const g = function(a) {`
`  return value;`
`}`
	`const g = a => value`	Function expression

`function (a,b) {`
`  return {`
`  p1: a,`
`  p2: b`
`  }`
`}`
	`(a,b) => ({`
	`  p1: a,`
	`  p2: b`
	`  })`
		When arrow function returns an object literal, the object literal must be wrapped in parentheses.

Changes to “this” in Arrow Functions

Arrow functions do not have their own **this** value (see Figure 8.22). Instead, the value of this within an arrow function is that of the enclosing lexical context (i.e., its enclosing parental scope at design time).

While this can occasionally be a limitation, it does allow the use of the **this** keyword in a way more familiar to object-oriented programming techniques in languages such as Java and C#.

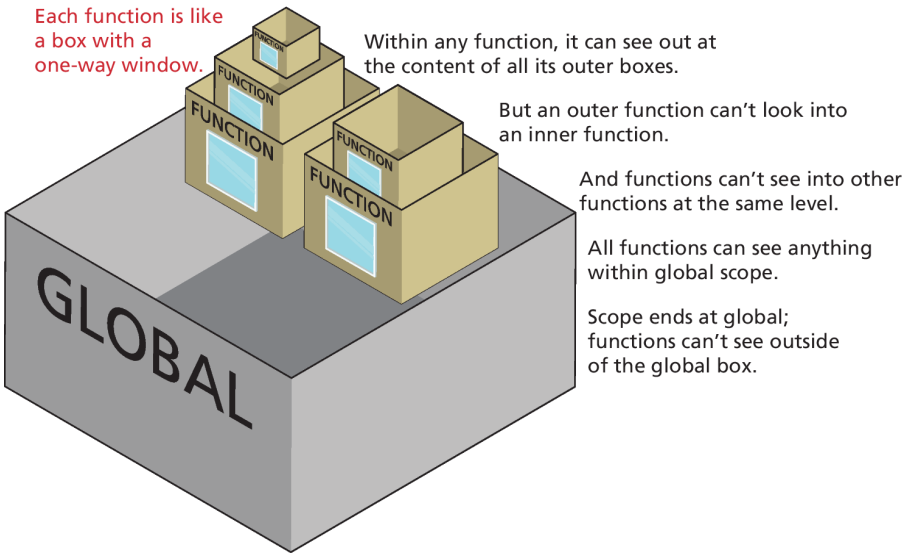
When we finally get to React development in Chapter 11, the use of arrow functions within ES6 classes will indeed make our code look a lot more like class code in a language like Java.

Scope in JavaScript

Scope generally refers to the context in which code is being executed.

JavaScript has four scopes:

- **function scope** (also called local scope),
- **block scope**,
- **module scope** (in Chapter 10)
- **global scope**.



Block scope

Block-level scope means variables defined within an **if {}** block or a **for {}** loop block using the **let** or **const** keywords are only available within the block in which they are defined. But if declared with **var** within a block, then it will be available outside the block.

3 Global Scope

```
for (var i=0; i<10;i++) {
   var tmp = "yes";
   console.log(tmp); outputs: yes
}
console.log(i);     outputs: 10
console.log(tmp);   outputs: yes
```

A variable will be in **global scope** if declared outside of a function *and* uses the `var` keyword.

4 Block Scope

```
for (let i=0; i<10;i++) {
   const tmp = "yes";
   console.log(tmp); outputs: yes
}
console.log(i);     error: i is not defined
console.log(tmp); error: tmp is not defined
```

A variable declared within a {} block using `let` or `const` will have **block scope** and *only* be available within the block it is defined.

Global Scope

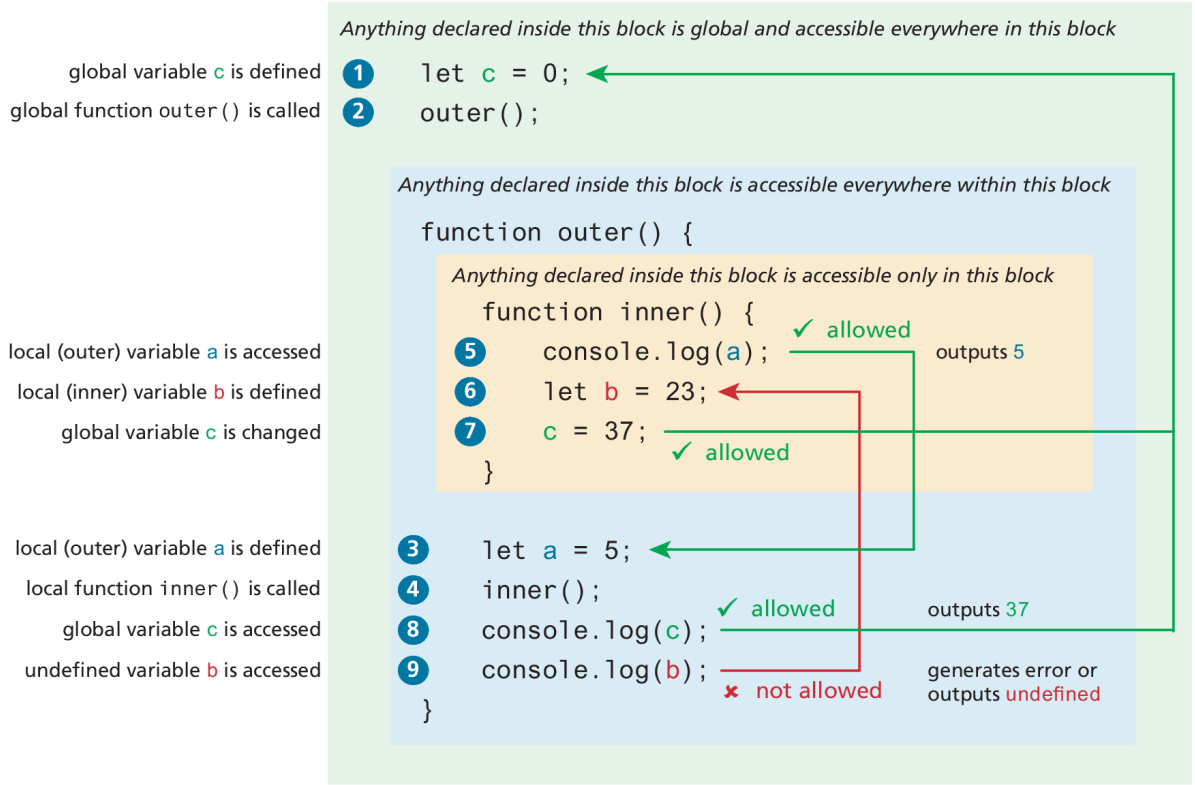
If an identifier has global scope, it is available everywhere.

Global scope can cause the **namespace conflict problem**.

- If the JavaScript compiler encounters another identifier with the same name at the same scope, you do not get an error. Instead, the new identifier *replaces* the old one!

In Chapter 10, you will learn of the new module feature in ES6 that helps address this problem.

Function/Local Scope

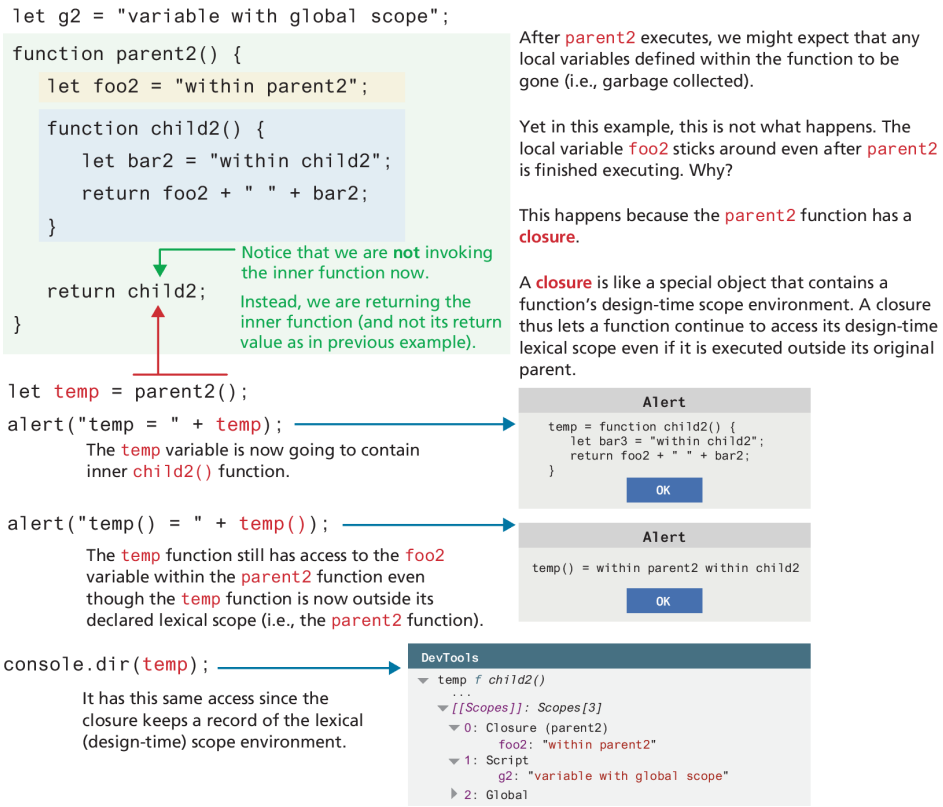


Closures in JavaScript

Scope in JavaScript is sometimes referred to as **lexical scope**

The ending bracket of a function is said to close the scope of that function. But closure refers to more than just this idea.

A **closure** is actually an object consisting of the scope environment in which the function is created; that is, a closure is a function that has an implicitly permanent link between itself and its scope chain.



Key Terms

AJAX
anonymous functions
array literal notation
assignment
arrays
arrow syntax
block scope
browser extension
browser plug-in
built-in objects
callback function
client-side scripting
closure
conditional operator
default parameters
dot notation
dynamically typed
ECMAScript
ES6
exception
falsy
for loops
functions
function constructor
function declaration
function expression
function scope
global scope
JavaScript frameworks
JavaScript Object Notation
JSON
keyword
lexical scope
loop control variable
method
module scope
namespace conflict
problem
objects
object literal notation
primitive types
property
reference types
rest operator
scope (local and global)
shallow copy
spread syntax
template literals
ternary operator
truthy
try. . . catch block
undefined
variables

Copyright

This work is protected by United States copyright laws and is provided solely for the use of instructors in teaching their courses and assessing student learning. Dissemination or sale of any part of this work (including on the World Wide Web) will destroy the integrity of the work and is not permitted. The work and materials from it should never be made available to students except by instructors using the accompanying text in their classes. All recipients of this work are expected to abide by these restrictions and to honor the intended pedagogical purposes and the needs of other instructors who rely on these materials.