Fundamentals of Web Development

Third Edition by Randy Connolly and Ricardo Hoar

Chapter 8

JavaScript 1:

Language Fundamentals

Pearson

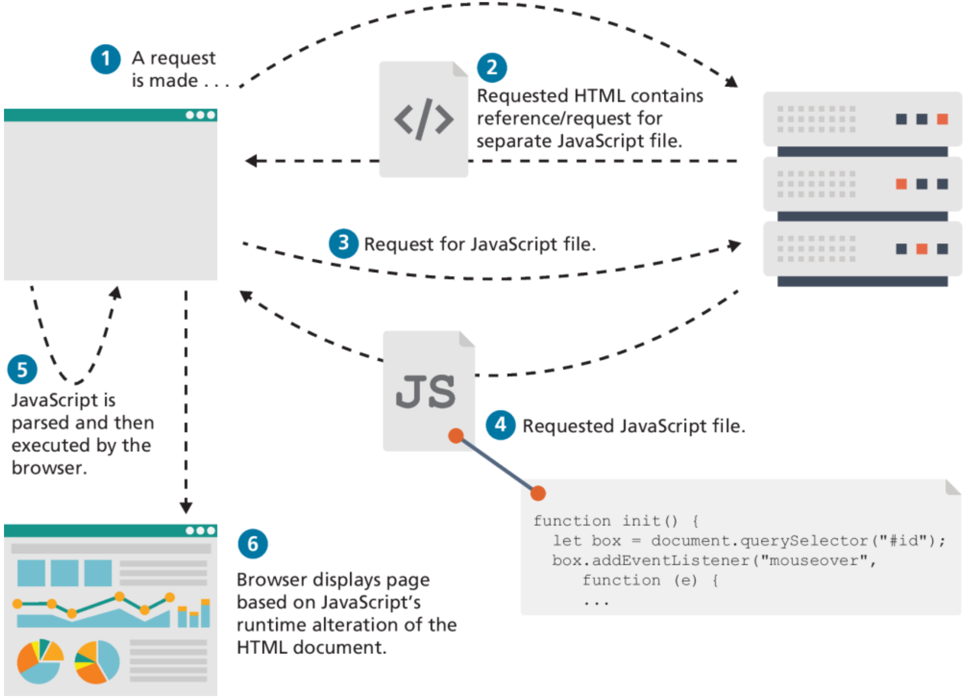
What Is JavaScript and What Can It Do?

- JavaScript: it is an object-oriented, dynamically typed scripting language
- *Primarily* a client-side scripting language.
- 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


1 A request is made . . .
2 Requested HTML contains reference/request for separate JavaScript file.
</>
3 Request for JavaScript file.
JS
4 Requested JavaScript file.
5 JavaScript is parsed and then executed by the browser.
6 Browser displays page based on JavaScript's runtime alteration of the HTML document.

```
function init() {
  let box = document.querySelector("#id");
  box.addEventListener("mouseover",
     function (e) {
      ...
```

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.

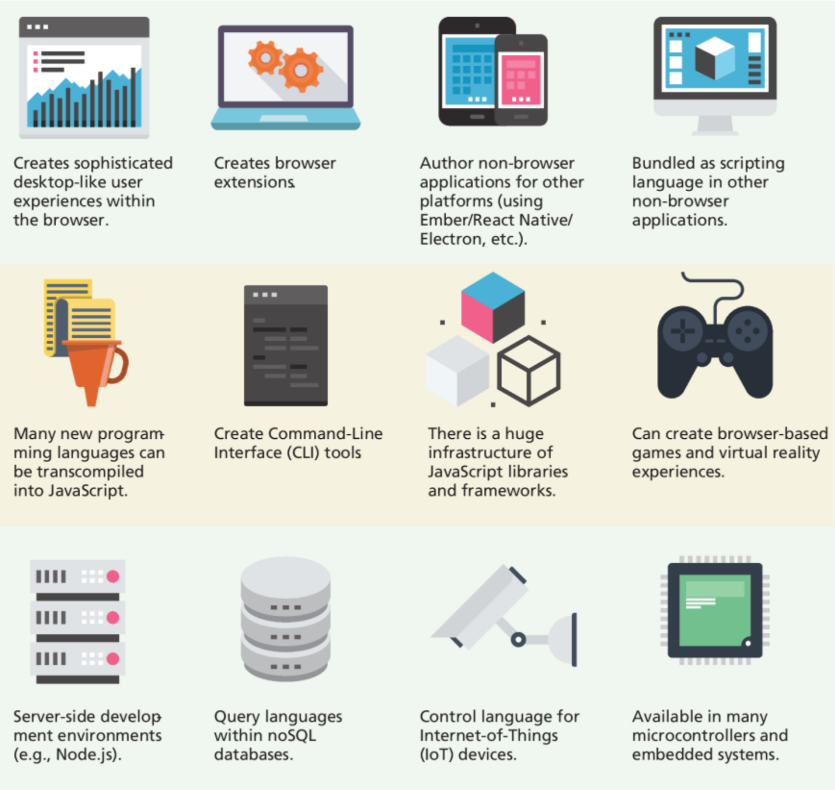
Client-Side Scripting: Disadvantages

- 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 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
- ....



Creates sophisticated desktop-like user experiences within the browser.

Creates browser extensions.

Author non-browser applications for other platforms (using Ember/React Native/ Electron, etc.).

Bundled as scripting language in other non-browser applications.

Many new programming languages can be transcompiled into JavaScript.

Create Command-Line Interface (CLI) tools

There is a huge infrastructure of JavaScript libraries and frameworks.

Can create browser-based games and virtual reality experiences.

Server-side development environments (e.g., Node.js).

Query languages within noSQL databases.

Control language for Internet-of-Things (IoT) devices.

Available in many microcontrollers and embedded systems.

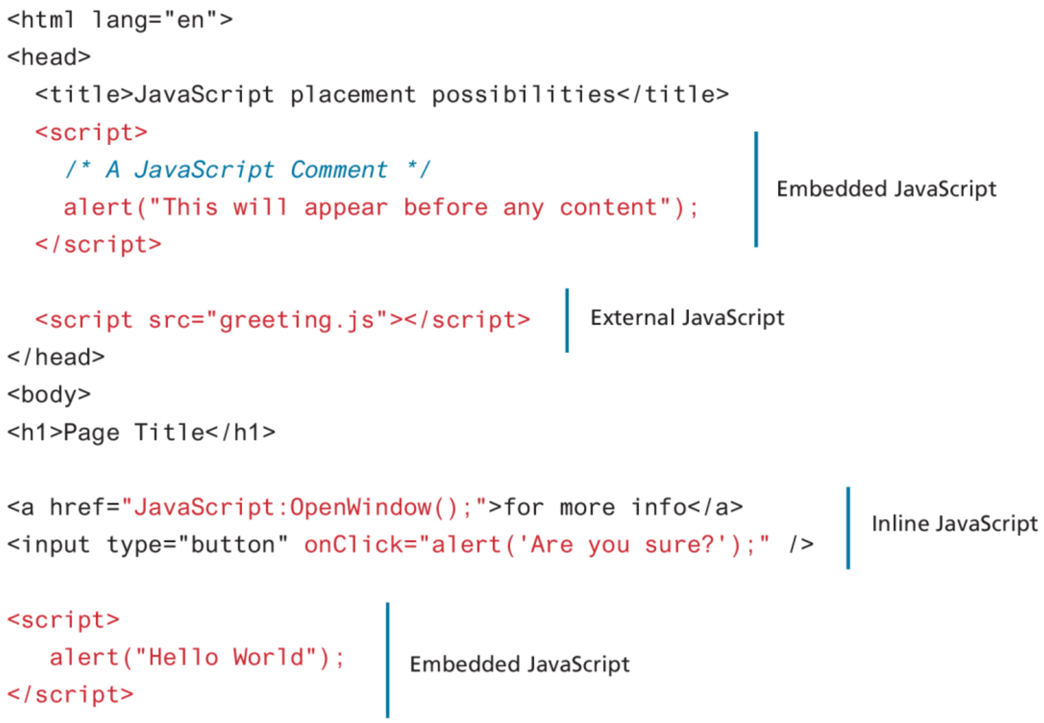
Pearson

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>
```

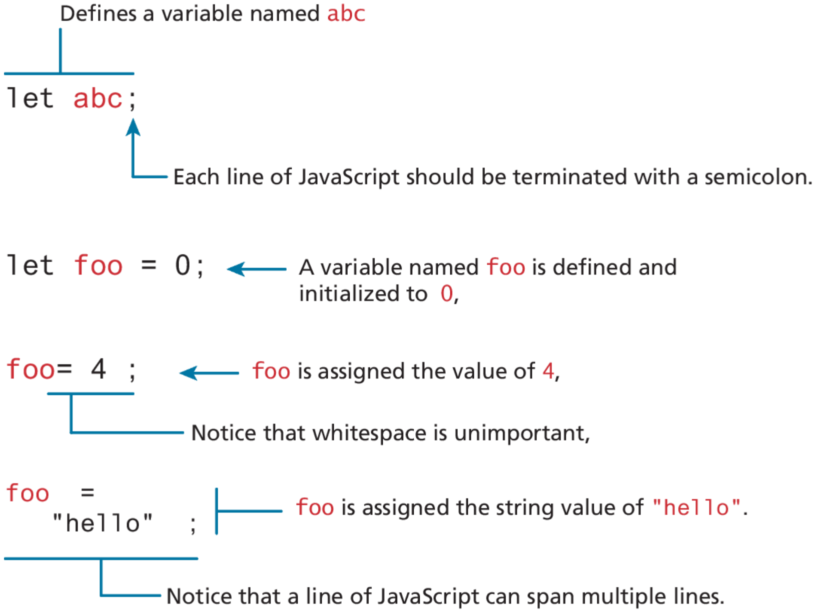
Embedded JavaScript
External JavaScript
Inline JavaScript
Embedded JavaScript

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.

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



Defines a variable named abc

```
let abc;
```

Each line of JavaScript should be terminated with a semicolon.

```
let foo = 0;
```

A variable named foo is defined and initialized to 0,

```
foo= 4 ;
```

foo is assigned the value of 4,

Notice that whitespace is unimportant,

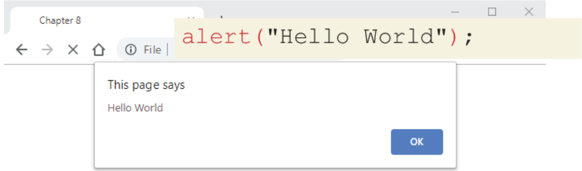
```
foo   =
   "hello"   ;
```

foo is assigned the string value of "hello".

Notice that a line of JavaScript can span multiple lines.

JavaScript Output

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

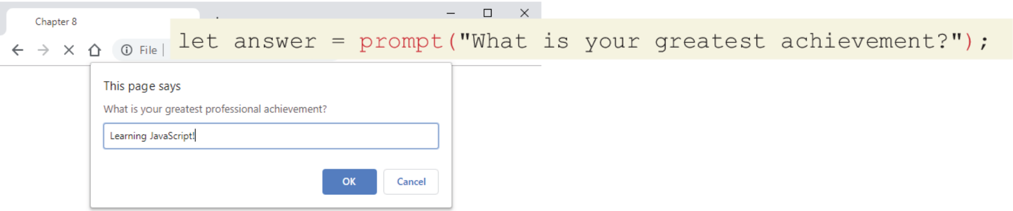


```
alert("Hello World");
```

This page says

Hello World

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



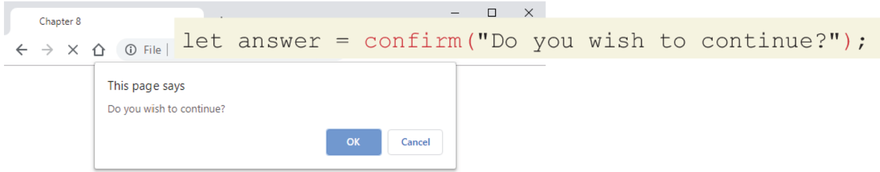
```
let answer = prompt("What is your greatest achievement?");
```

This page says

What is your greatest professional achievement?

Learning JavaScript!

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



```
let answer = confirm("Do you wish to continue?");
```

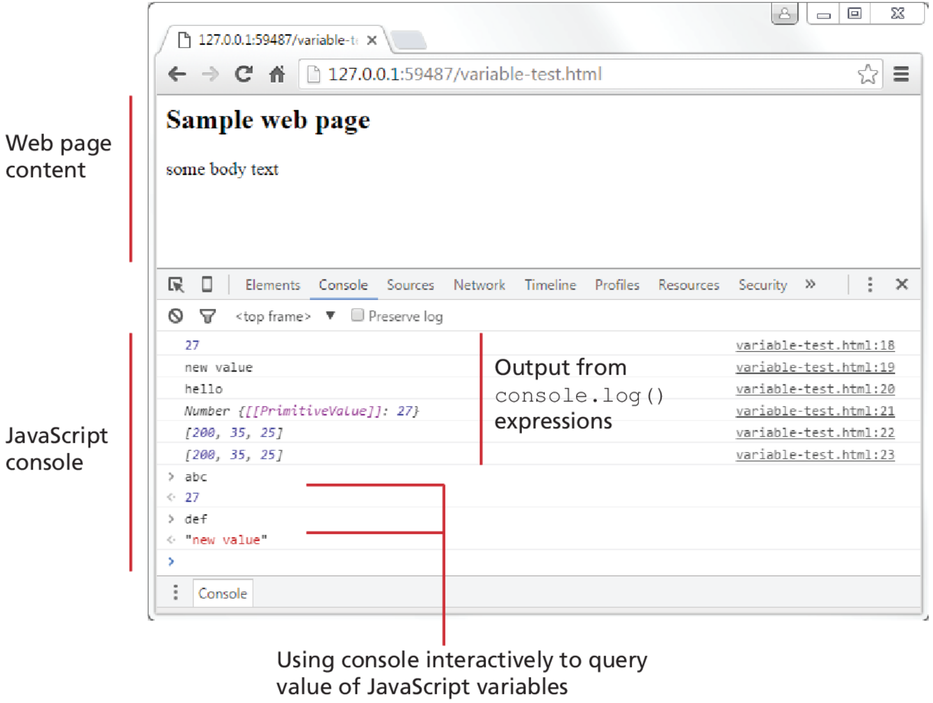
This page says

Do you wish to continue?

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.


127.0.0.1:59487/variable-t
127.0.0.1:59487/variable-test.html
Elements
Console
Sources
Network
Timeline
Profiles
Resources
Security
<top frame>
Preserve log
Console
Web page content
JavaScript console
Output from console.log() expressions
Using console interactively to query value of JavaScript variables

Sample web page

some body text

```
27                                       variable-test.html:18
new value                                variable-test.html:19
hello                                    variable-test.html:20
Number {[[PrimitiveValue]]: 27}          variable-test.html:21
[200, 35, 25]                            variable-test.html:22
[200, 35, 25]                            variable-test.html:23
> abc
< 27
> def
< "new value"
>
```

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.

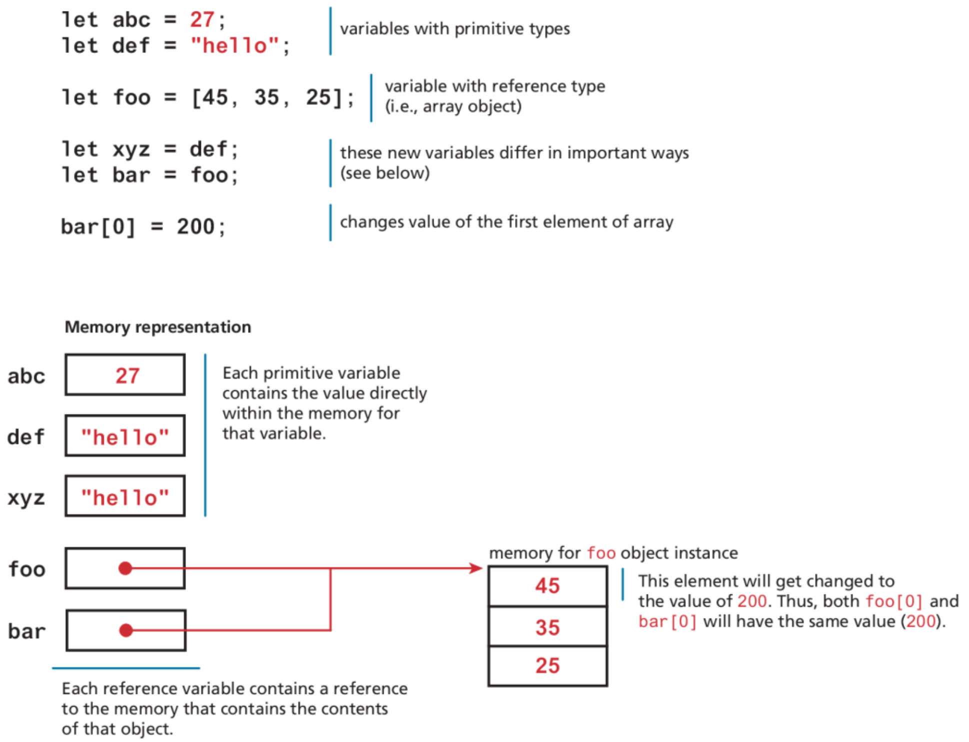
Primitive Types

- **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.

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

Memory representation

Each primitive variable contains the value directly within the memory for that variable.

This element will get changed to the value of 200. Thus, both foo[0] and bar[0] will have the same value (200).

Each reference variable contains a reference to the memory that contains the contents of that object.

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.

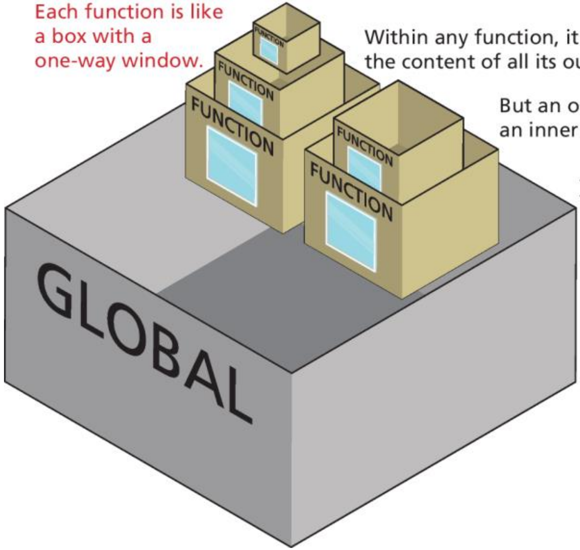
```
let def = new Date();
// sets the value of abc to a string containing the current date
let abc = def.toString();
```

Scope in JavaScript

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

JavaScript scopes:

- **function scope** (also called local scope),
- **block scope**,
- **global scope**.



Each function is like a box with a one-way window.

Within any function, it can see out at the content of all its outer boxes.

But an outer function can't look into an inner function.

And functions can't see into other functions at the same level.

All functions can see anything within global scope.

Scope ends at global; functions can't see outside of the global box.

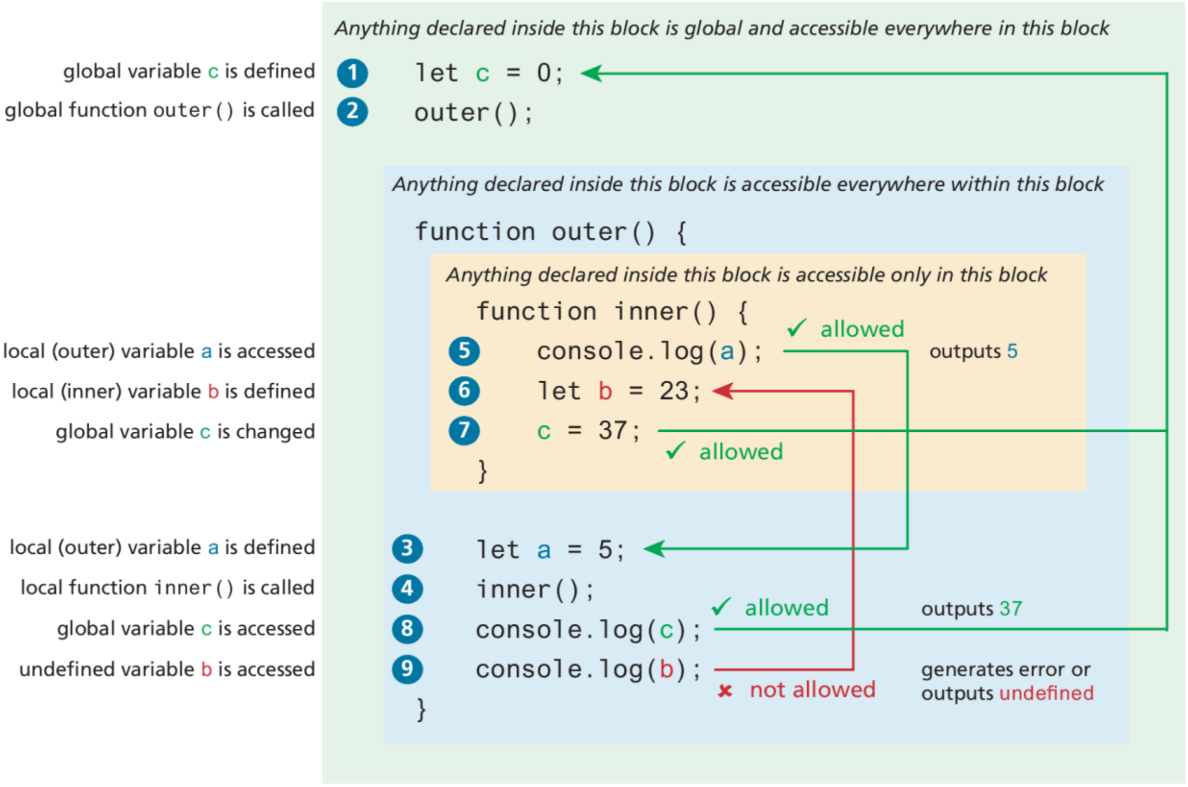
Global Scope

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!

Function/Local Scope



Anything declared inside this block is global and accessible everywhere in this block

global variable c is defined

global function outer() is called

Anything declared inside this block is accessible everywhere within this block

Anything declared inside this block is accessible only in this block

local (outer) variable a is accessed

local (inner) variable b is defined

global variable c is changed

local (outer) variable a is defined

local function inner() is called

global variable c is accessed

undefined variable b is accessed

```
1  let c = 0;
2  outer();

   function outer() {
      function inner() {
5        console.log(a);
6        let b = 23;
7        c = 37;
      }

3     let a = 5;
4     inner();
8     console.log(c);
9     console.log(b);
   }
```

✓ allowed — outputs 5

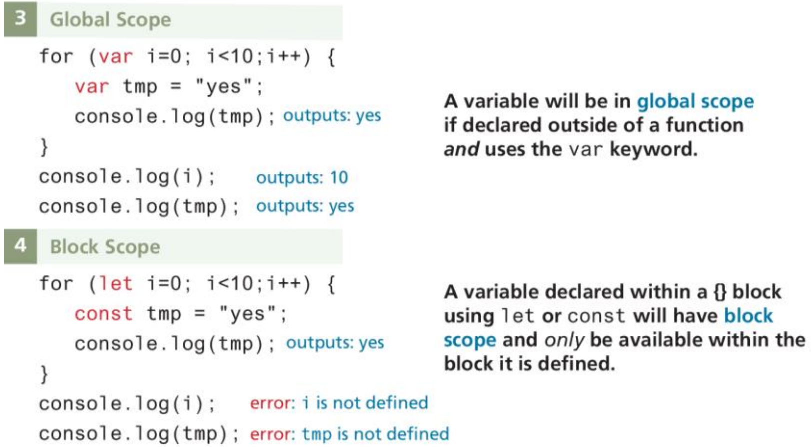
✓ allowed

✓ allowed — outputs 37

✘ not allowed — generates error or outputs undefined

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.

Concatenation

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

```
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.

While and do . . . while Loops

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

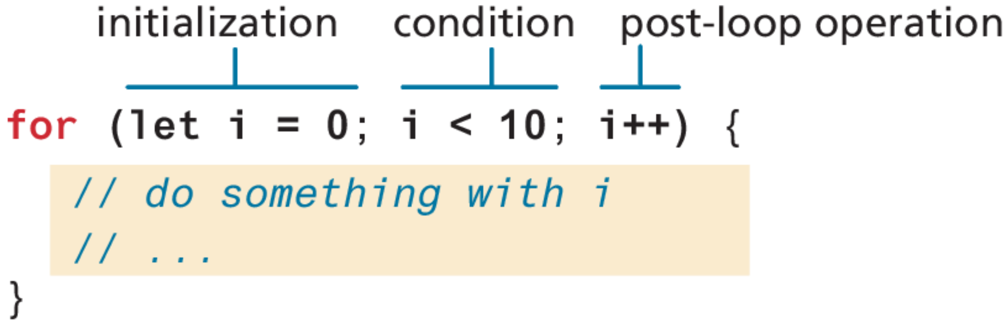
As you can see from this example, while loops normally initialize a **loop control variable** before the loop, use it in the condition, and modify it within the loop.

```
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 (;)



```
for (let i = 0; i < 10; i++) {
   // do something with i
   // ...
}
```

Use for (var i = 0; i < 10; i++) if you want to use i outside of the for loop.

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:

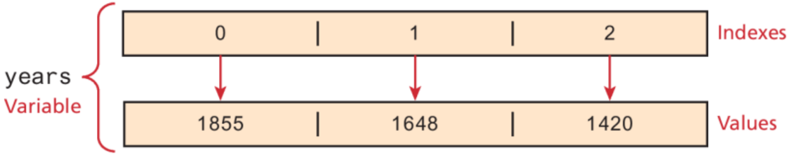
- const name = new Array(*value1*, *value2*, ...);

Pearson

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;
```

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.

There are many different ways to define and use functions in JavaScript.

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);
```

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.

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

Nested Functions

JavaScript allows for functions to be nested inside of each other.

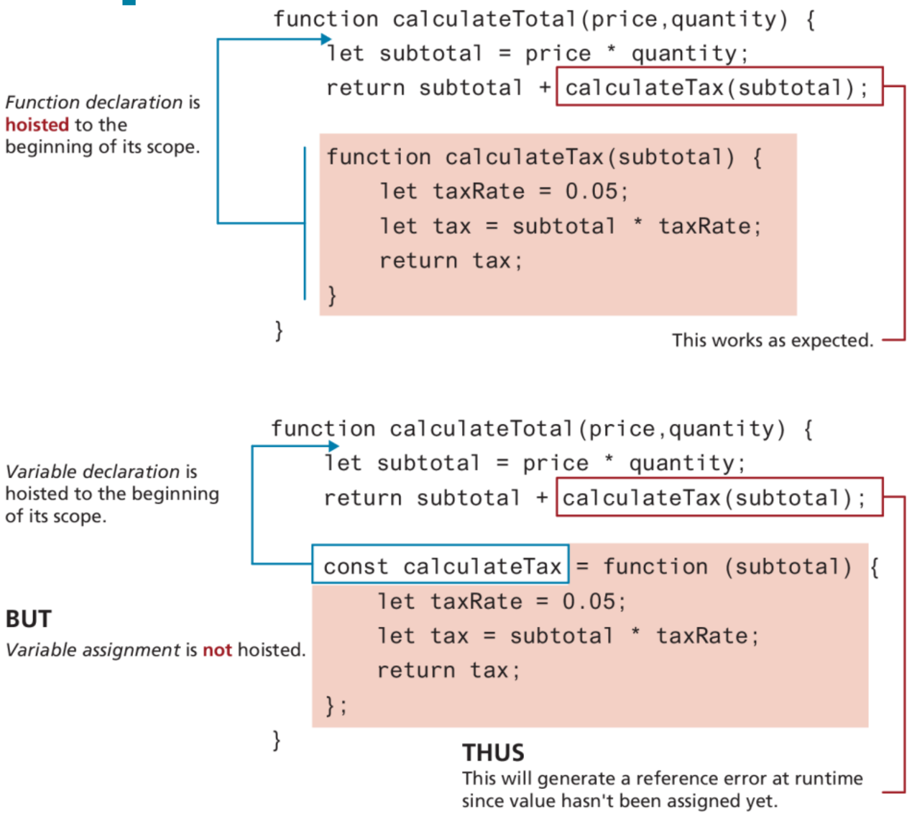
```
function calculateTotal(price,quantity) {
   let subtotal = price * quantity;
   return subtotal + calculateTax(subtotal);

   // 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.



Function declaration is **hoisted** to the beginning of its scope.

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

  function calculateTax(subtotal) {
     let taxRate = 0.05;
     let tax = subtotal * taxRate;
     return tax;
  }
}
```

This works as expected.

Variable declaration is hoisted to the beginning of its scope.

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

  const calculateTax = function (subtotal) {
     let taxRate = 0.05;
     let tax = subtotal * taxRate;
     return tax;
  };
}
```

BUT

Variable assignment is **not** hoisted.

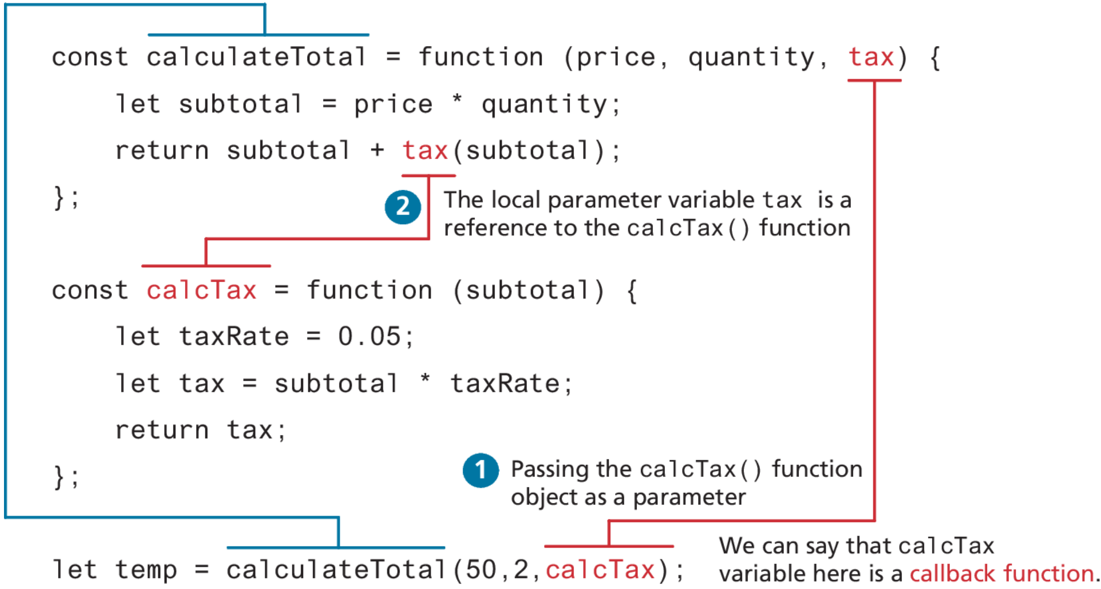
THUS

This will generate a reference error at runtime since value hasn't been assigned yet.

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.



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

const calcTax = function (subtotal) {
    let taxRate = 0.05;
    let tax = subtotal * taxRate;
    return tax;
};

let temp = calculateTotal(50,2,calcTax);
```

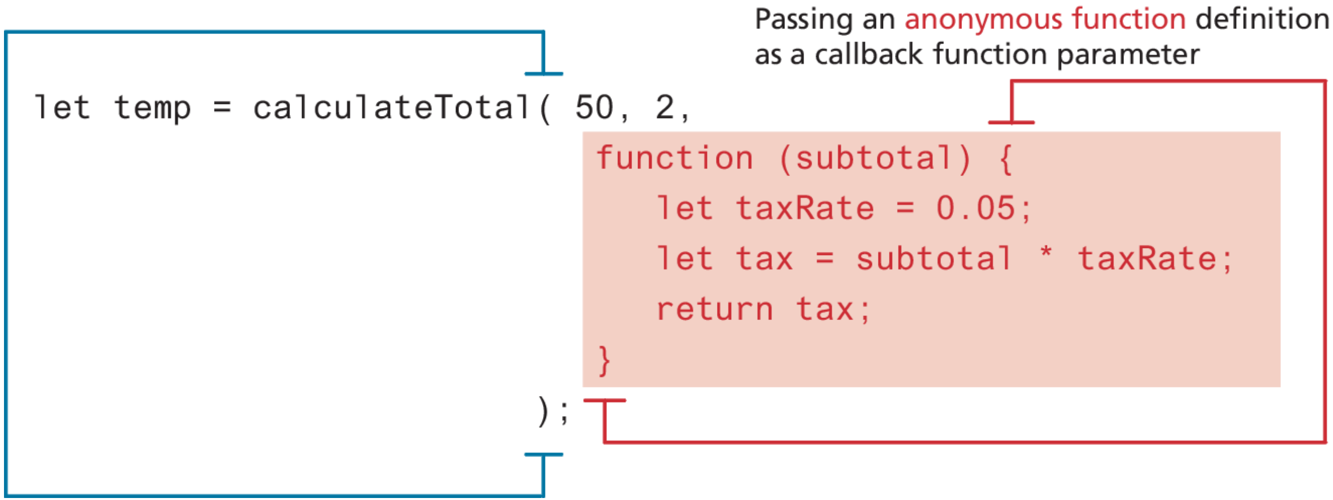
2 The local parameter variable tax is a reference to the calcTax() function

1 Passing the calcTax() function object as a parameter

We can say that calcTax variable here is a callback function.

Callback Functions (ii)

We can actually define a function definition directly within the invocation



```
let temp = calculateTotal( 50, 2,
                          function (subtotal) {
                             let taxRate = 0.05;
                             let tax = subtotal * taxRate;
                             return tax;
                          }
                        );
```

Passing an anonymous function definition
as a callback function parameter

Arrow Functions

An **arrow function expression** is a compact alternative to a traditional function expression, but is limited and can't be used in all situations.

```
// Traditional Anonymous Function

(function (a) {

  return a + 100;

});
```

```
// Arrow Function Break Down

// 1. Remove the word "function" and place arrow between the argument and opening body bracket

(a) => {

  return a + 100;

};
```

```
// 2. Remove the body braces and word "return" — the return is implied.

(a) => a + 100;
```

```
// 3. Remove the argument parentheses

a => a + 100;
```

Arrow Functions

More Examples:

```
// Traditional Anonymous Function
(function (a, b) {
  return a + b + 100;
});
// Arrow Function
(a, b) => a + b + 100;
```

```
// Traditional Anonymous Function (no arguments)
(function() {
  return a + b + 100;
});

// Arrow Function (no arguments)
() => a + b + 100;
```

Arrow Functions

More Examples:

```
// Traditional Anonymous Function with more than one line of code.

(function (a, b) {

  const chuck = 42;

  return a + b + chuck;

});
```

```
// Arrow Function must include return statement

(a, b) => {

  const chuck = 42;

  return a + b + chuck;

};
```

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.

Object Creation Using Object Literal Notation

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

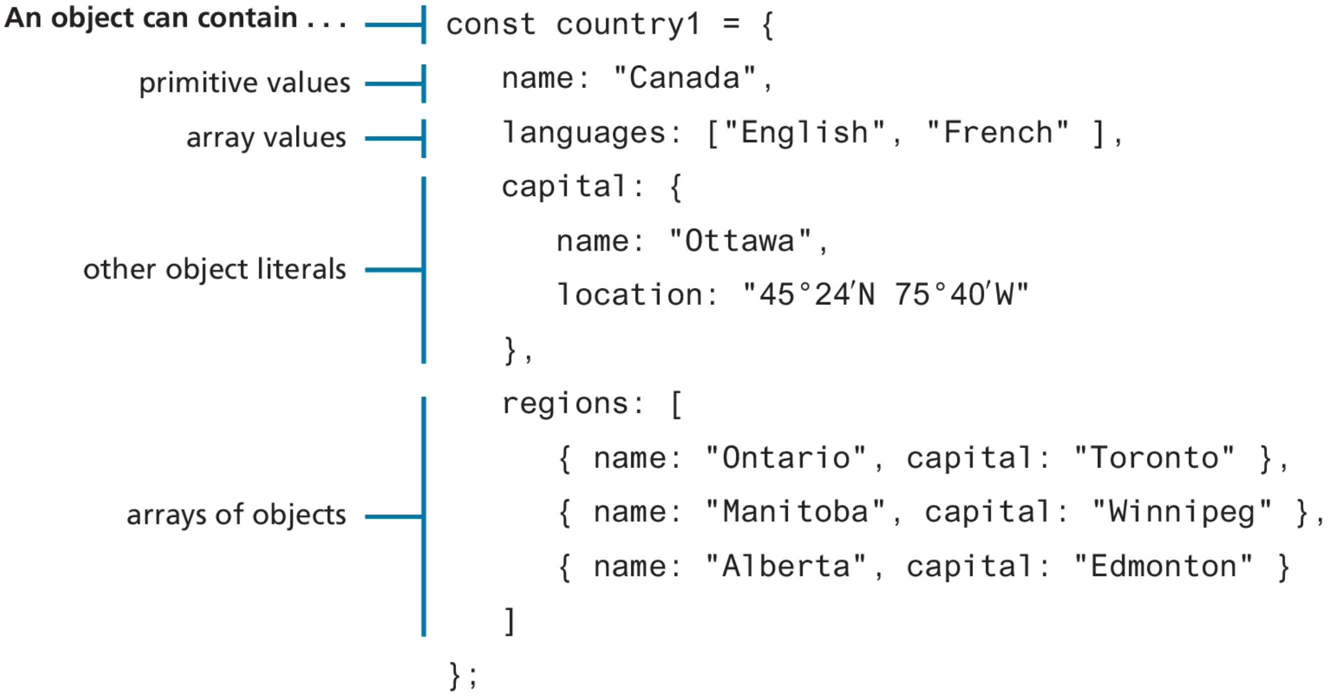
An object is represented by a list of key-value pairs with colons between the key and value, with commas separating key-value pairs.

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

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

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

Objects containing other content



```
const country1 = {
    name: "Canada",
    languages: ["English", "French" ],
    capital: {
        name: "Ottawa",
        location: "45°24′N 75°40′W"
    },
    regions: [
        { name: "Ontario", capital: "Toronto" },
        { name: "Manitoba", capital: "Winnipeg" },
        { name: "Alberta", capital: "Edmonton" }
    ]
};
```

Objects can contain functions

Remember functions in javascript are objects. We can include functions as an object within an object.

```
const person = {
  firstName: "John",
  lastName: "Doe",
  id: 5566,
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
};
```

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;
```

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

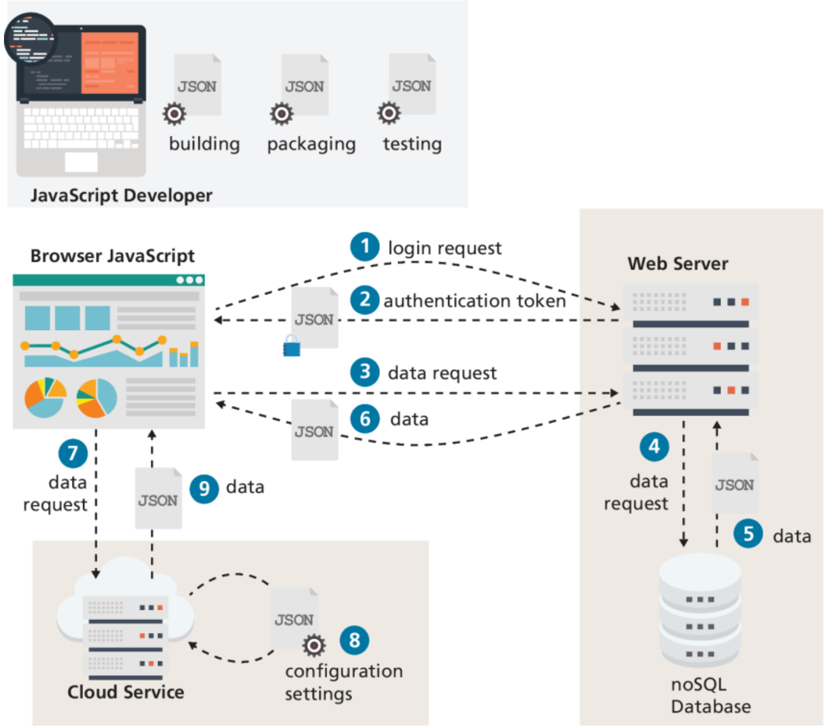
The string literal on the last slide contains an object definition in JSON format (but is still just a string). To turn this string into an actual JavaScript object requires using the built-in JSON object.

```
// 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.



Copyright