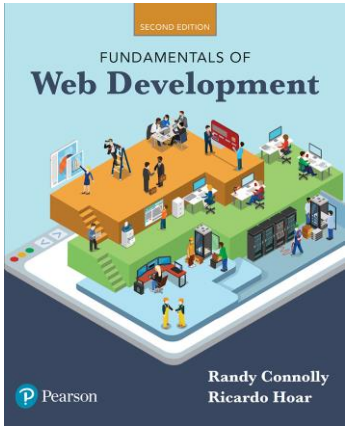
CSE 686 Internet Programming

Week 11: Introduction to JavaScript, Part 1: The Basics

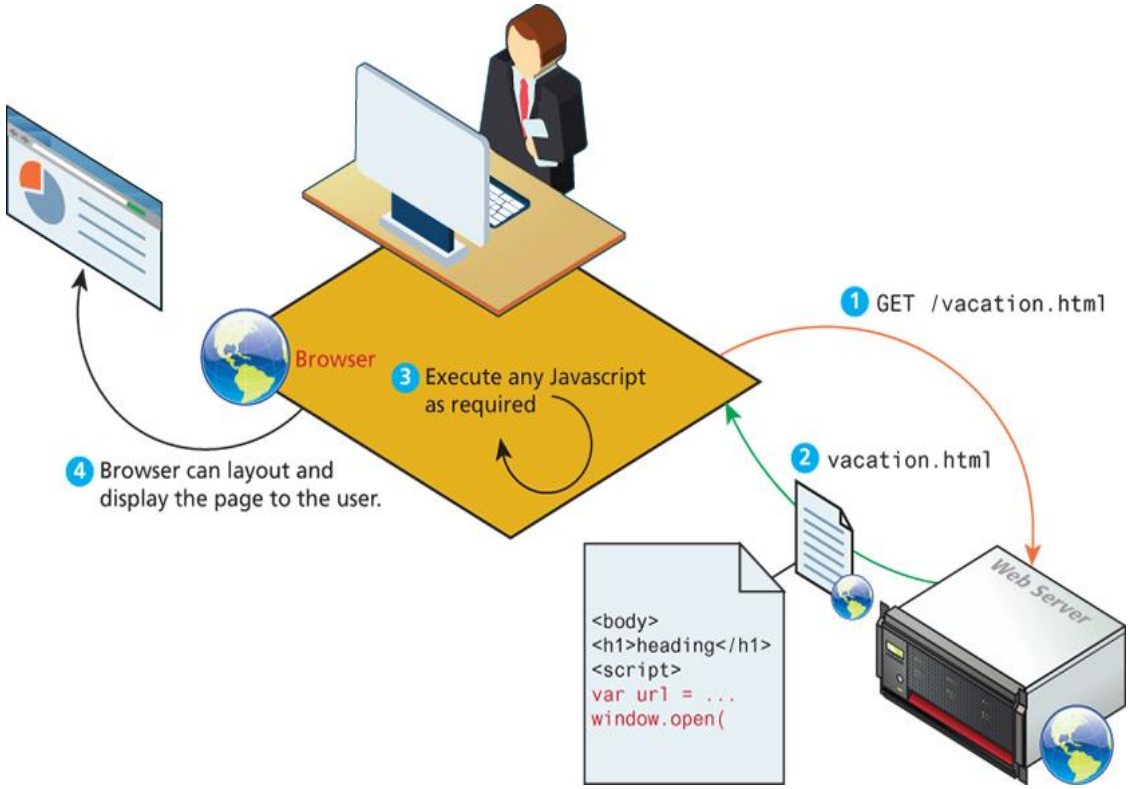
Edmund Yu, PhD
Associate Teaching Professor
esyu@syr.edu

March 26, 2018



JavaScript

- It is an object-oriented, dynamically typed scripting language. (Larry Ullman, *Modern JavaScript: Develop and Design*, Peachpit Press, 2012)
- In the context of this course, it is primarily used a client-side scripting language.
- Later in this course, we may discuss **node.js**, a popular server-side implementation of JavaScript.



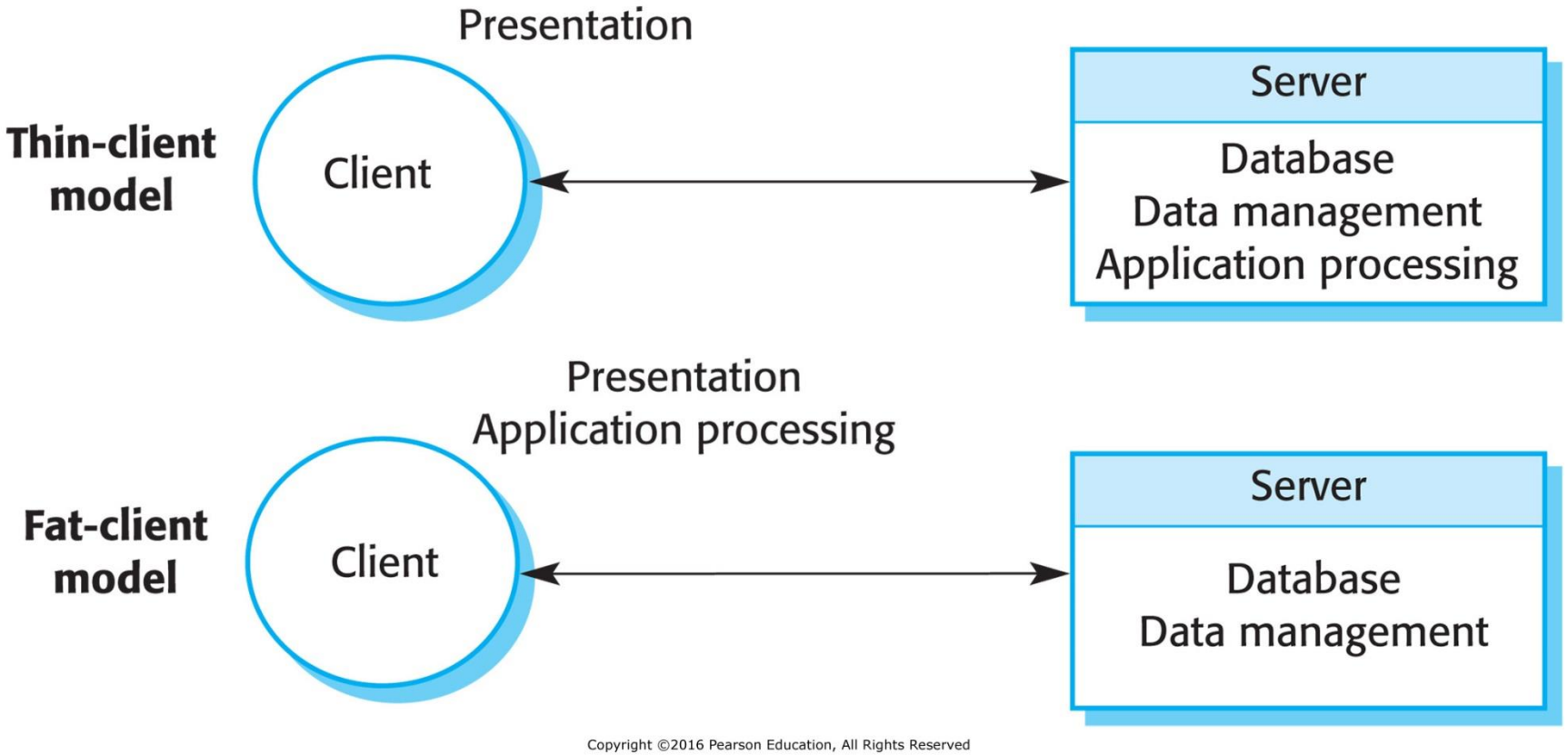
JS: Client-Side Scripting

- Client-side scripting refers to the client (i.e., the browser) running code locally rather than relying on the server to execute code and return the result.
- There are many client-side languages that have come into use over the past decades including Flash, VBScript, Java (Applets), and JavaScript.
- Some of these technologies only work in certain browsers, while others require plugins to function.
- JavaScript is the most popular, due to its browser interoperability (that is, its ability to work/operate on most browsers).

Advantages

- Processing can be off-loaded from the server to client machines, thereby reducing the load on the server. (See next slide)
- 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.

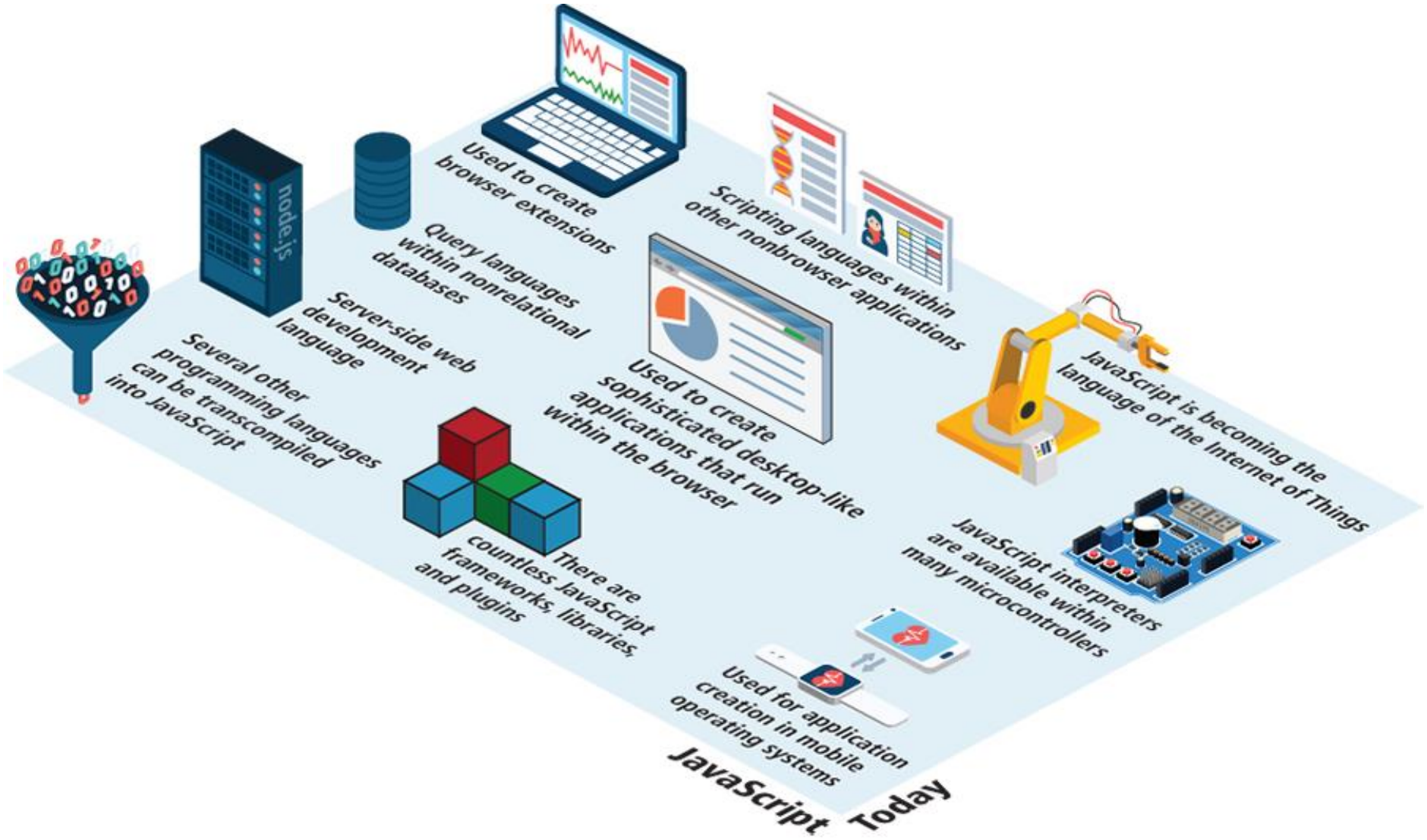
Thin- & Fat-client Architectural Models



Disadvantages

- There is no guarantee that the client has JavaScript enabled. Any required functionality must be implemented redundantly on the server.
- The idiosyncrasies of JavaScript implementation between various browsers and operating systems make it difficult to test for all potential client configurations. What works in one browser, may generate an error in another.
- 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.
- JavaScript-heavy web applications can be complicated to debug and maintain, especially with JavaScript embedded into the HTML of a web page. This decreases code readability.

What Can JavaScript Do?



Where Does JavaScript Go?

Route #1

Inline JavaScript refers to the practice of including JavaScript code directly within certain HTML attributes

more info

<input type="button" onClick="**alert('Are you sure?');**" />

Where Does JavaScript Go?

Route #2

Embedded JavaScript refers to the practice of placing JavaScript code within a <script> element

```
<script type="text/javascript">
    /* A JavaScript Comment */
    alert("Hello World!");
</script>
```

Where Does JavaScript Go?

Route #3

external JavaScript files typically contain function definitions, data definitions, and entire frameworks.

```
<head>
<script type="text/javascript“ src="greeting.js"></script>
</head>
```

Variables and Data Types

- Variables in JavaScript are **dynamically typed.**
- This simplifies variable declarations, since we do not require the familiar data-type identifiers.
- Instead we simply use the **var** keyword

Variables and Data Types

Defines a variable named abc

```
var abc;
```

Each line of JavaScript should be terminated with a semicolon

```
var def = 0;
```

A variable named def is defined and initialized to 0

```
def= 4 ;
```

def is assigned the value of 4

Notice that whitespace is unimportant

```
def  =
   "hello"  ;
```

def is assigned the value of "hello"

Notice that a line of JavaScript can span multiple lines

Variables and Data Types

- JavaScript is a case-sensitive language. Thus these two lines declare and initialize two different variables:

```
var count = 5;
var Count = 9;
```

Comments

- There are two styles of comment in JavaScript, the end-of-line comment which starts with two slashes //, and the block comment, which begins with /* and ends with */.

Run »

Result Size: 556 x 439

```
<!DOCTYPE html>
<html>
<body>

<h1 id="myH"></h1>
<p id="myP"></p>

<script>
// Change heading:
document.getElementById("myH").innerHTML = "My First Page";
// Change paragraph:
document.getElementById("myP").innerHTML = "My first paragraph.";
</script>

<p><strong>Note:</strong> The comments are not executed.</p>

</body>
</html>
```

My First Page

My first paragraph.

Note: The comments are not executed.

Run »

Result Size: 556 x 439

```
<!DOCTYPE html>
<html>
<body>

<h1 id="myH"></h1>
<p id="myP"></p>

<script>
/*
The code below will change
the heading with id = "myH"
and the paragraph with id = "myp"
in my web page:
*/
document.getElementById("myH").innerHTML = "My First Page";
document.getElementById("myP").innerHTML = "My first paragraph.";
</script>

<p><strong>Note:</strong> The comment block is not executed.</p>

</body>
</html>
```

My First Page

My first paragraph.

Note: The comment block is not executed.

Identifiers

- In JavaScript, just like other programming languages, identifies are used to name variables (and keywords, and functions, and labels).
- The rules for legal names are much the same in most programming languages: the first character must be a letter, or an underscore (_), or a dollar sign ($).
- Subsequent characters may be letters, digits, underscores, or dollar signs.
- However, Unicode characters can be used in JavaScript identifiers. (See next slide)

```
// uses Greek character
var π = 3.1415;
// uses Kannada character
var ಠ_ಠ = "disapproval";
// uses Katakana characters
var ロ = θ;
var ㄱㄱ = ロ;
```

Variables and Data Types

- There are two basic data types in JavaScript:
  - **Reference types** (usually referred to as **objects**): arrays, functions, and the built-in objects, such as Object, Function, Error, Math, Date, and Regexp.
  - **Primitive types**: see next slide

Table 8.1 Primitive Types

Data type	Description
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.

Variables and Data Types

```
var abc = 27;
var def = "hello";
```

variables with primitive types

```
var foo = [45, 35, 25];
```

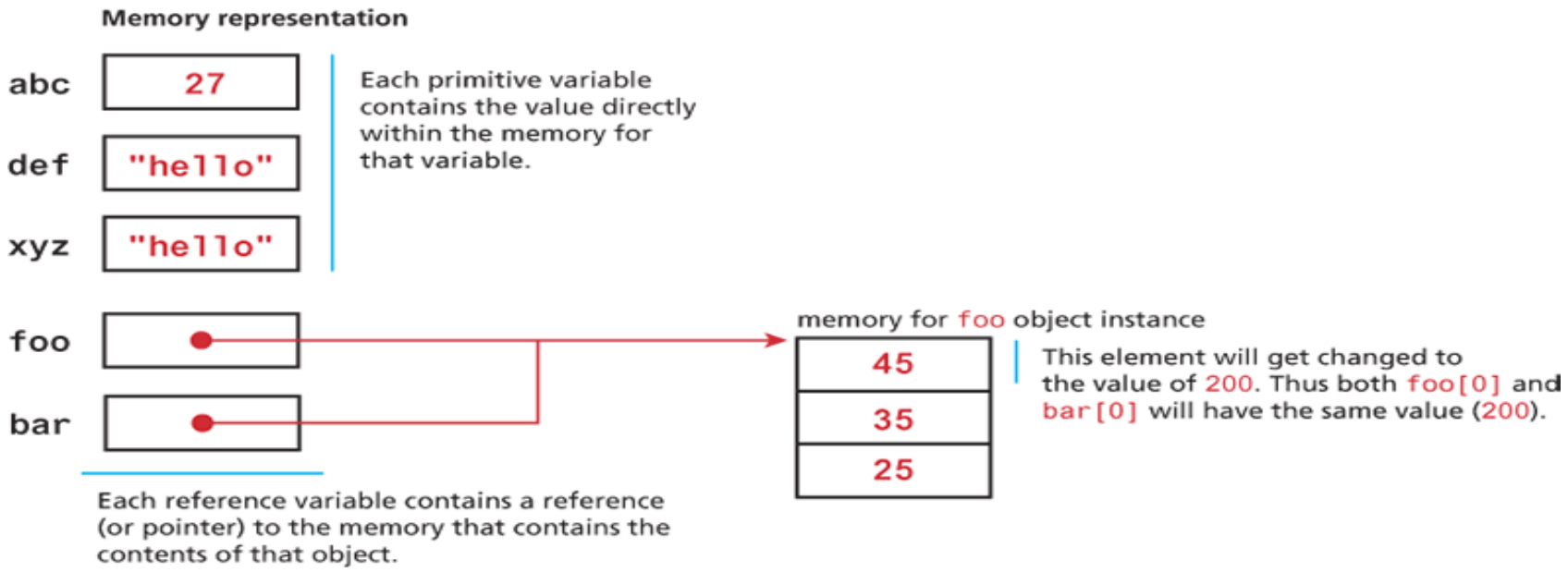
variable with reference type (i.e., array object)

```
var xyz = def;
var bar = foo;
```

these new variables differ in important ways (see below)

```
bar[0] = 200;
```

changes value of the first element of array



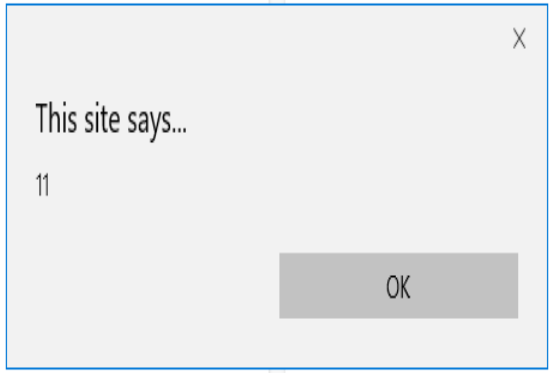
JavaScript Output (1)

- When first learning JavaScript, one often uses the **alert()** method.
- It makes the browser show a pop-up to the user, with whatever is passed being the message displayed.
- The following JavaScript code displays a simple hello world message in a pop-up:

```
alert("Hello world");
// or window.alert("Hello world");
```

Run »

```
<!DOCTYPE html>
<html>
<body>

<h2>My First Web Page</h2>
<p>My first paragraph.</p>

<script>
window.alert(5 + 6);
</script>

</body>
</html>
```



My First Web Page

JavaScript Output (2)

- Alerts are generally not used in production code, but are a useful tool for debugging and illustration purposes. However, using alerts can get tedious fast.
- As an alternative, people often use the **console.log()** method since console output doesn't interfere with the HTML content document.
- console.log() Displays content in the Browser's JavaScript console. (See next slide)

Run »

Result Size: 501 x 244

```
<!DOCTYPE html>
<html>
<body>

<h2>Activate debugging with F12</h2>

<script>
console.log(5 + 6);
</script>

</body>
</html>
```

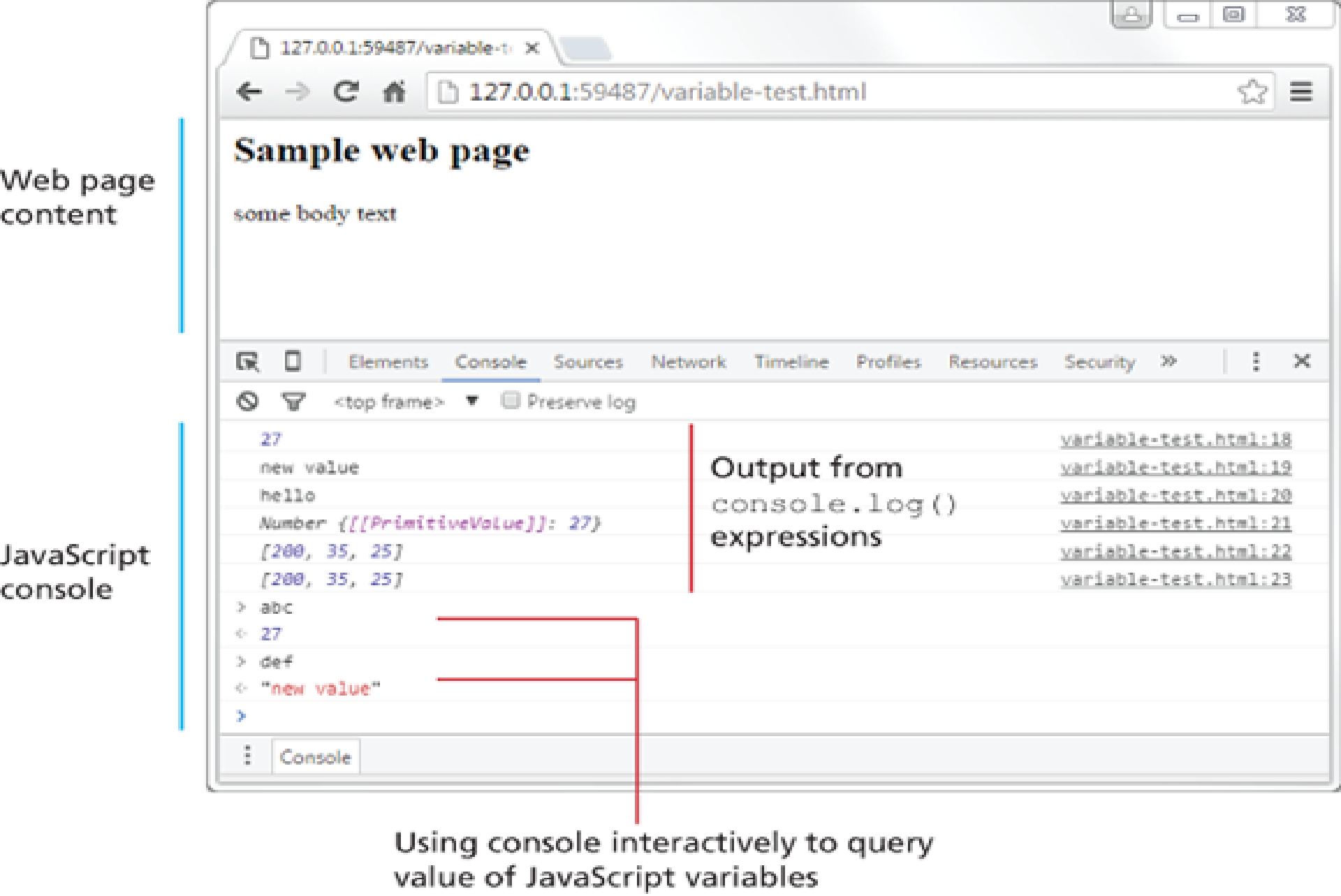
Activate debugging with F12

F12 DOM Explorer Console Debugger Network Performance Memory Emulation 11

0 0 0

Target _top: tryit.asp

11



Chrome JavaScript Console

JavaScript Output (3)

- **document.write**() Outputs the content directly to the HTML document.
  - This method can be a useful way to output markup content from within JavaScript.
  - This method is often used to output markup or to combine markup with JavaScript variables, as shown in the following example:

```
var name = "Randy";
document.write("<h1>Title</h1>");
// this uses the concatenate operator (+)
document.write("Hello " + name + " and welcome");
```

Run »

Result Size: 556 x 439

```
<html>
<body>

<script>
var name = "Randy";
document.write("<h1>Title</h1>");
// this uses the concatenate operator (+)
document.write("Hello " + name + " and welcome");
</script>
</body>
</html>
```

Title

Hello Randy and welcome

Conditionals

if else

```
if (hourOfDay > 4 && hourOfDay < 12) {
        greeting = "Good Morning";
}
else if (hourOfDay >= 12 && hourOfDay < 18) {
        greeting = "Good Afternoon";
}
else {
        greeting = "Good Evening";
}
```

switch

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

Conditional Assignment

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

Condition — Value if true — Value if false

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

Truthy and Falsy

- 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.
  - Almost all values in JavaScript are truthy
  - false, Null, "", '', 0, NaN, and **undefined** are falsy

Table 8.3 Comparator Operations

Operator	Description	Matches (assume x=9)
==	Equals	(x == 9) is true (x == "9") is true
===	Strict equality, including type	(x === "9") is false (x === 9) is true
< , >	Less than, greater than	(x < 5) is false
<= , >=	Less than or equal, greater than or equal	(x <= 9) is true
!=	Not equal	(x != 4) is true
!==	Not equal in either value or type	(x !== "9") is true (x !== 9) is false

Loops

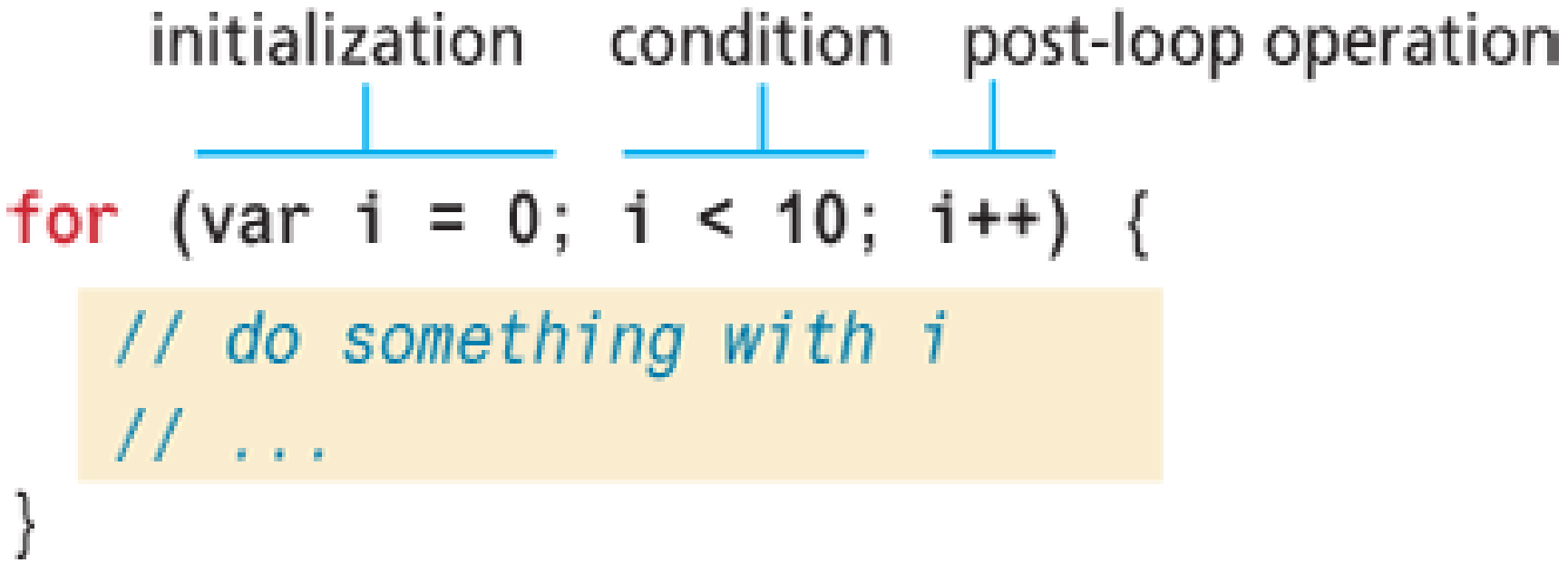
while & do-while

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

Are they equivalent?

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

for


initialization
condition
post-loop operation
for (var i = 0; i < 10; i++) {
// do something with i
// ...
}

Arrays

Arrays

- Arrays are one of the most commonly used data structures in programming.
- JavaScript provides two main ways to define an array.
    1. Use the **Array() constructor**
    2. **Object literal** notation (preferred)

Run »

Result Size: 470 x 350

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Arrays</h2>

<p id="demo"></p>

<script>
var cars = new Array("Saab", "Volvo", "BMW");
document.getElementById("demo").innerHTML = cars;
</script>

</body>
</html>
```

JavaScript Arrays

Saab,Volvo,BMW

Object Literal Notation

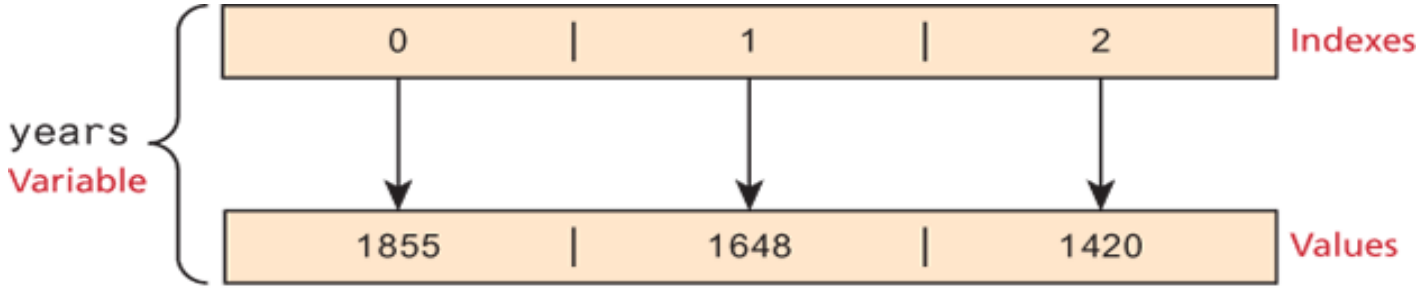
- The object literal notation approach is generally preferred since it involves less typing, is more readable, and executes a little bit quicker.

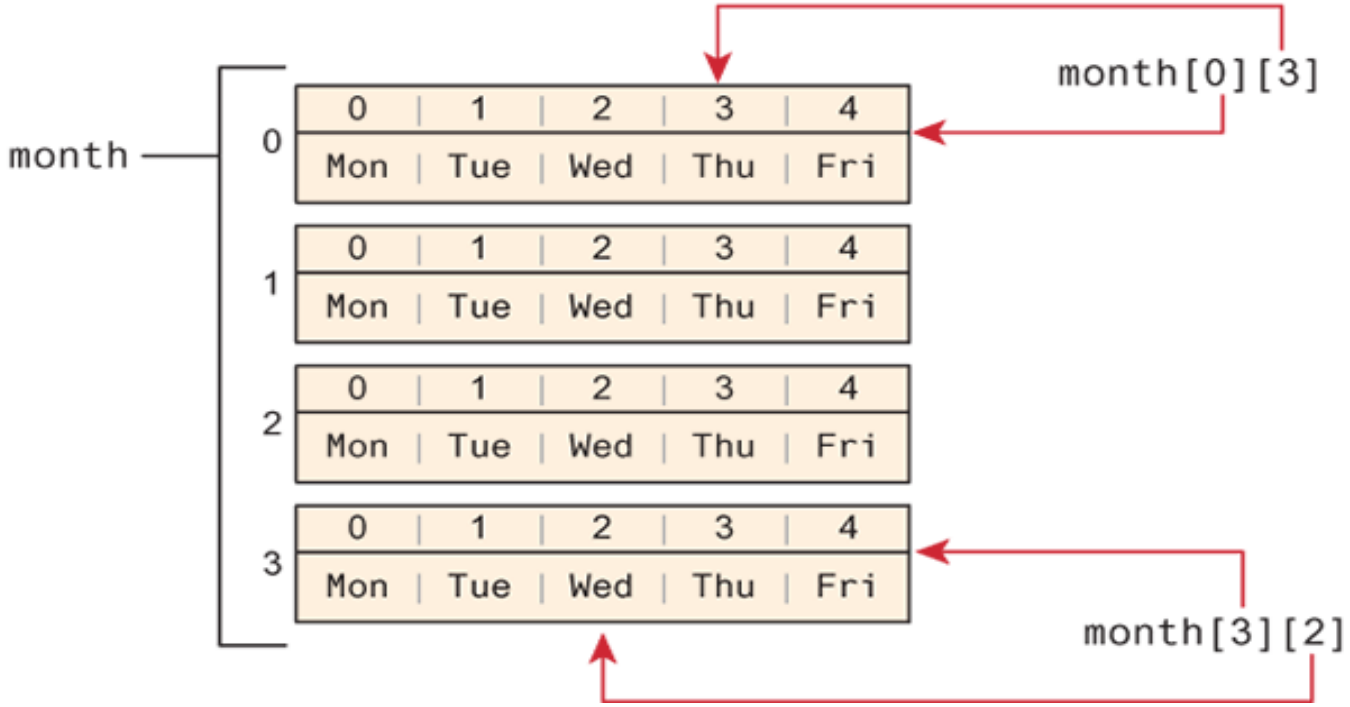
- See next slide for examples.

Listing 8.7 Creating arrays using object literal notation

```
var years =  [1855, 1648, 1420];
// remember that JavaScript statements can be
// spread across multiple lines for readability
var countries = ["Canada", "France",
                 "Germany", "Nigeria",
                 "Thailand", "United States"];
// arrays can also be multi-dimensional … notice the commas!
var month = [
        ["Mon","Tue","Wed","Thu","Fri"],
        ["Mon","Tue","Wed","Thu","Fri"],
        ["Mon","Tue","Wed","Thu","Fri"],
        ["Mon","Tue","Wed","Thu","Fri"]
    ];
// JavaScript arrays can contain different data types
var mess =  [53, "Canada", true, 1420];
```

Arrays Illustrated





JavaScript Array Methods

- All arrays inherit a variety of properties and methods that can be used to explore and manipulate an array:
  - **.length** gives the length of the array
  - **.push()** adds an item to the end of an existing array
  - **.pop()** removes the last element from an array
  - Additional methods that modify arrays include .shift(), .concat(), .slice(), .join(), .sort, .reverse(), etc. (see 'JavaScript Array Reference' at www.w3schools.com, next slide)

JavaScript Array Reference

‹ Previous
Next ›

Array Object

The Array object is used to store multiple values in a single variable:

```
var cars = ["Saab", "Volvo", "BMW"];
```

Array indexes are zero-based: The first element in the array is 0, the second is 1, and so on.

For a tutorial about Arrays, read our JavaScript Array Tutorial.

```
for (var i = 0;  i < years.length; i += 1) {
   // do something
}
```

A more efficient way of writing this loop is to cache the upper boundary.

```
var upper = years.length;
for (var i = 0;  i < upper; i += 1) {
   // do something
}
```

You can also combine these two lines into the initialization part of the `for` statement.

```
for (var i = 0, upper = years.length; i < upper; i += 1) {
```

Run »

Result Size: 625 x 520

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Array Methods</h2>

<h2>push()</h2>

<p>The push() method appends a new element to an array.</p>

<button onclick="myFunction()">Try it</button>

<p id="demo"></p>

<script>
var fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits;

function myFunction() {
    fruits.push("Kiwi");
    document.getElementById("demo").innerHTML = fruits;
}
</script>

</body>
</html>
```

JavaScript Array Methods

push()

The push() method appends a new element to an array.

Try it

Banana,Orange,Apple,Mango

Run »

Result Size: 625 x 520

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Array Methods</h2>

<h2>push()</h2>

<p>The push() method returns the new array length.</p>

<button onclick="myFunction()">Try it</button>

<p id="demo1"></p>
<p id="demo2"></p>

<script>
var fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo1").innerHTML = fruits;

function myFunction() {
    document.getElementById("demo2").innerHTML = fruits.push("Kiwi");
    document.getElementById("demo1").innerHTML = fruits;
}
</script>

</body>
</html>
```

JavaScript Array Methods

push()

The push() method returns the new array length.

Try it

Banana,Orange,Apple,Mango

Run »

Result Size: 625 x 520

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Array Methods</h2>

<h2>pop()</h2>

<p>The pop() method removes the last element from an array.</p>

<p id="demo1"></p>
<p id="demo2"></p>

<script>
var fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo1").innerHTML = fruits;
fruits.pop();
document.getElementById("demo2").innerHTML = fruits;
</script>

</body>
</html>
```

JavaScript Array Methods

pop()

The pop() method removes the last element from an array.

Banana,Orange,Apple,Mango

Banana,Orange,Apple

Run »

Result Size: 625 x 520

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Array Methods</h2>

<h2>pop()</h2>

<p>The pop() method removes the last element from an array.</p>

<p>The return value of the pop() method is the removed item.</p>

<p id="demo1"></p>
<p id="demo2"></p>
<p id="demo3"></p>

<script>
var fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo1").innerHTML = fruits;
document.getElementById("demo2").innerHTML = fruits.pop();
document.getElementById("demo3").innerHTML = fruits;
</script>

</body>
</html>
```

JavaScript Array Methods

pop()

The pop() method removes the last element from an array.

The return value of the pop() method is the removed item.

Banana,Orange,Apple,Mango

Mango

Banana,Orange,Apple

Objects

Objects

- Objects are essential in JavaScript. In fact, we have already used quite a few built-in objects: arrays, Math, Date, and document objects.
- In JavaScript, objects are a collection of named values (which are called **properties** in JavaScript).
- Almost everything within JavaScript is an object (or can be treated as an object). Unlike languages such as Java or C++, objects in JavaScript are not created from classes.
  - In fact, classes in JS are poorly supported by the major browsers.
- JavaScript is a **prototype-based** language in that new objects are created from already existing prototype objects.

Object Creation

```
var objName = {
        name1: value1,
        name2: value2,
        // …
        nameN: valueN
}
```

- Access using either of:
  1. objName.name1
  2. objName["name1"]

```
<!DOCTYPE html>
<html>
<body>

<p>Creating a JavaScript Object.</p>

<p id="demo"></p>

<script>
var person = {
    firstName : "John",
    lastName  : "Doe",
    age       : 50,
    eyeColor  : "blue"
};

document.getElementById("demo").innerHTML =
person.firstName + " is " + person.age + " years old.";
</script>

</body>
</html>
```

Creating a JavaScript Object.

John is 50 years old.

Object Creation: Constructed Form

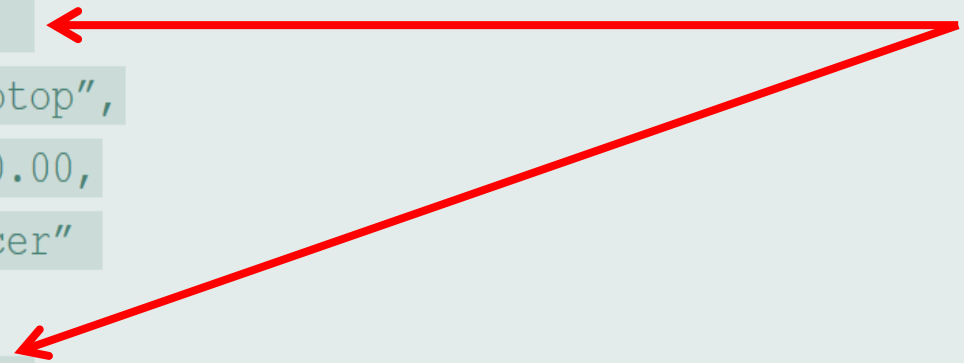
```
// first create an empty object
var objName = new Object();
// then define properties for this object
objName.name1 = value1;
objName.name2 = value2;
```

Or:

```
// create an empty object using literal notation
var objName = {};
```

Listing 8.10 Objects nested within other objects

```
var order = {
  salesDate : "May 5, 2016",
  product: {
  type: "laptop",
  price: 500.00,
  brand: "Acer"
  },
  customer: {
  name: "Sue Smith",
  address: "123 Somewhere St",
  city: "Calgary"
  }
};
alert(order.salesDate);
alert(order.product.type);
alert(order.customer.name);
```



JSON

```
// this is just a string though it looks like an object literal:

var text = '{ “name1” : “value1”, “name2” : “value2”,
                “name3” : “value3” }';

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

Run »

Result Size: 625 x 520

```
<!DOCTYPE html>
<html>
<body>

<h2>Create Object from JSON String</h2>

<p id="demo"></p>

<script>
var text = '{"employees":[' +
'{"firstName":"John","lastName":"Doe" },' +
'{"firstName":"Anna","lastName":"Smith" },' +
'{"firstName":"Peter","lastName":"Jones" }]}';

obj = JSON.parse(text);
document.getElementById("demo").innerHTML =
obj.employees[1].firstName + " " + obj.employees[1].lastName;
</script>

</body>
</html>
```

Create Object from JSON String

Anna Smith

Functions

Function Declaration

- **Functions** are the building block for modular code in JavaScript:

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

- This is formally called a **function declaration**, called or invoked by using the **()** operator:

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

Function Declaration Using New

```
// defines a function
var sub = new Function('price, quantity',
                       'return price * quantity');

// invokes the function
 var result = sub(10, 2);
```

Function Expressions

```
// defines a function using a function expression
var sub = function subtotal(price, quantity) {
        return price * quantity;
};

// invokes the function
var result = sub(10, 2);
```

Anonymous Function Expression

- It is conventional to leave out the function name in function expressions

```
// defines a function using an anonymous function expression
var calculateSubtotal = function (price, quantity)
                                    { return price * quantity; };
// invokes the function
var result = calculateSubtotal(10,2);
```

Quiz

```
// defines a function expression
var frenchHello = function () { return “bonjour”; };

// outputs bonjour
alert(frenchHello());

// what does this output? Notice the missing parentheses
alert(frenchHello);
```

Nested Functions

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

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

Hoisting in JavaScript

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

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

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

Variable declaration is hoisted to the beginning of its scope

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

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

BUT

Variable assignment is **not** hoisted

THUS

The value of the `calculateTax` variable here is **undefined**

Callback Functions

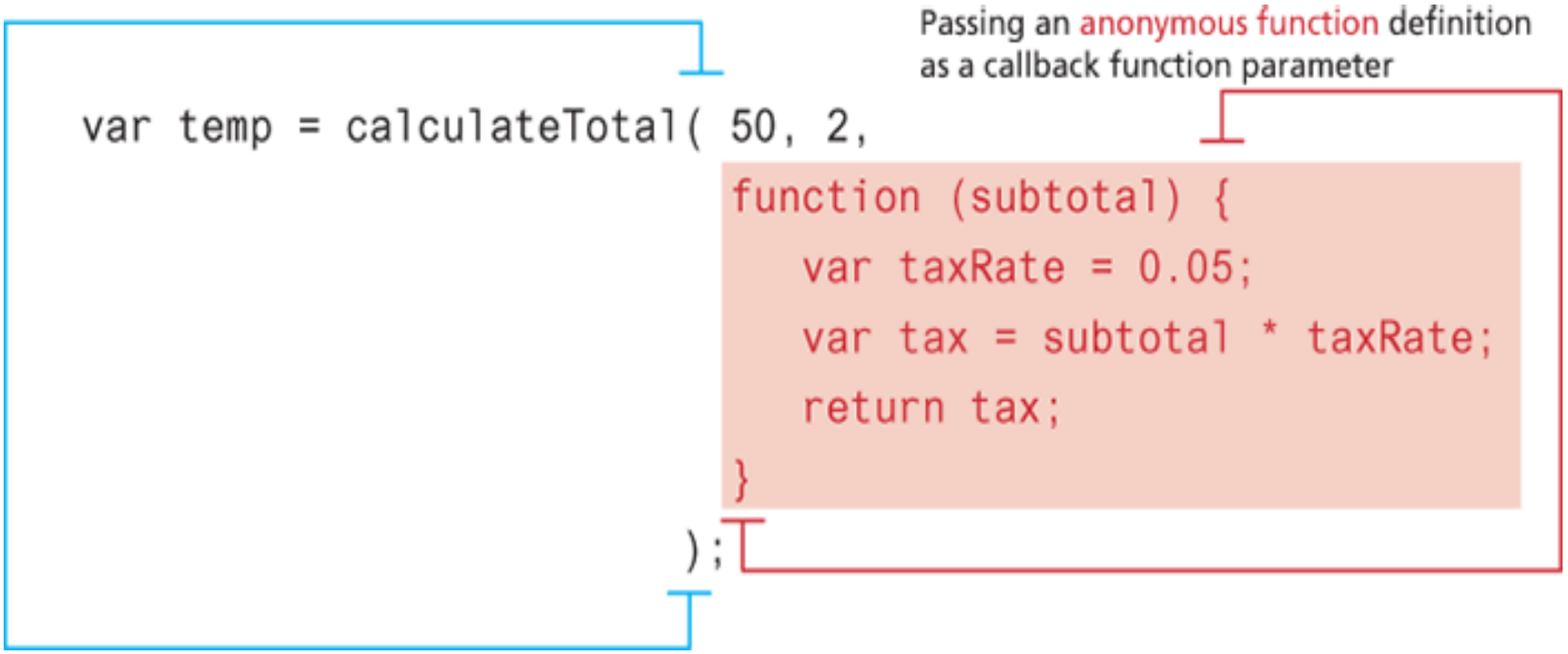
```
var calculateTotal = function (price, quantity, tax) {
    var subtotal = price * quantity;
    return subtotal + tax(subtotal);
};

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

var 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



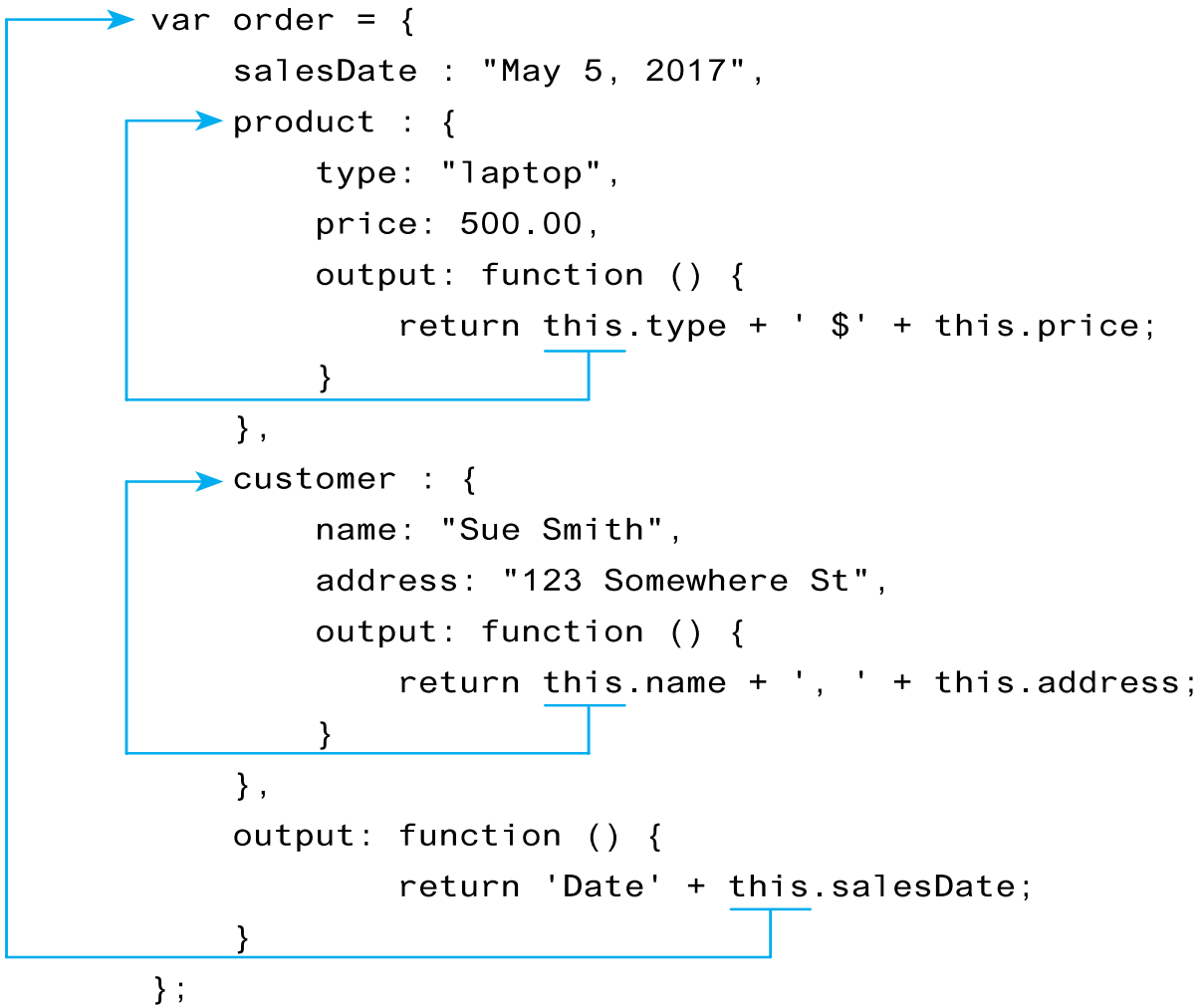
Arrow Functions

- One of the more interesting additions to the base JavaScript language in ES2015 is that of arrow functions.
- These arrow functions (also known as fat arrow function for reasons that will be obvious soon) provide a more concise syntax for the definition of anonymous functions and at the time of writing are supported in all modern browsers except Safari.
- They also provide a solution to problems encountered with the **this** keyword in callback functions, but that aspect of arrow functions is beyond our current level of understanding.

Arrow Functions

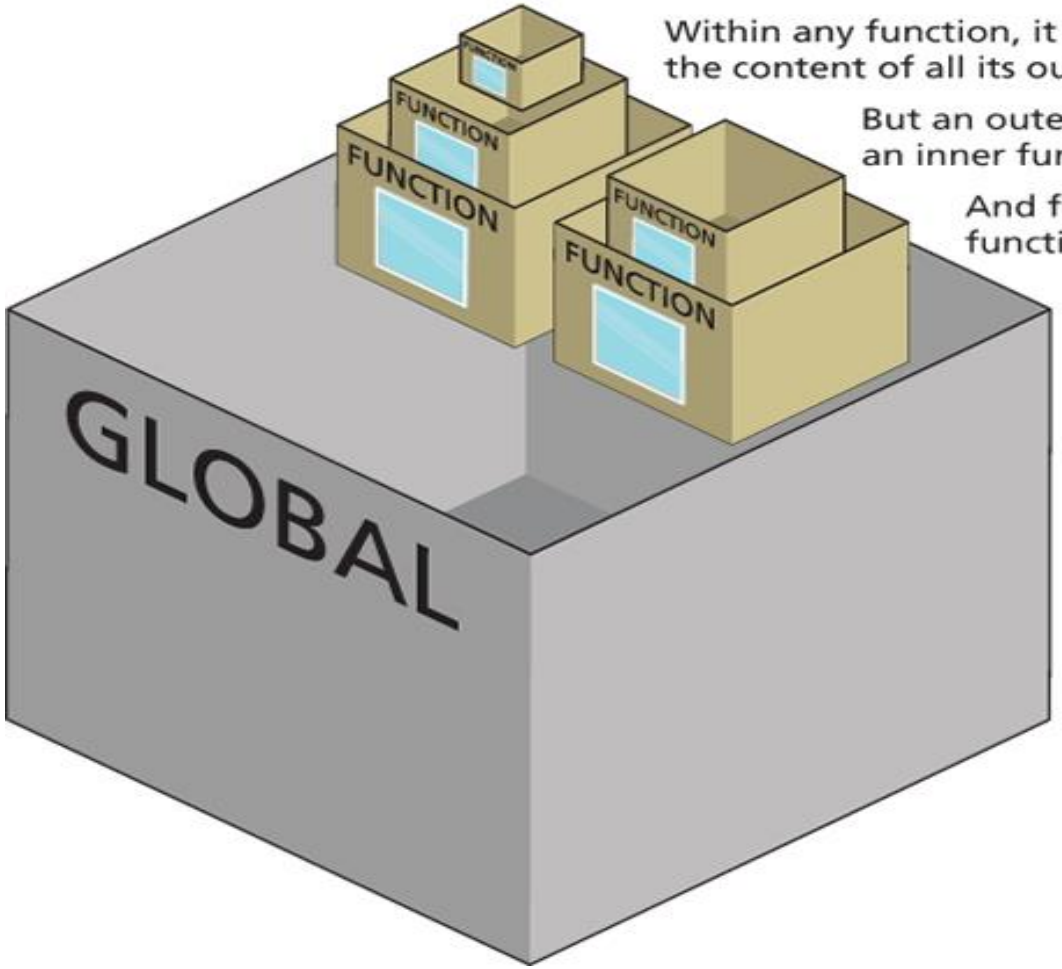
```
        var tax = function () { return 0.05; };
// becomes
        var tax = () => 0.05;


        var subtotal = function (price, quantity)
                        { var sub = price * quantity;
                          return sub;
                         };
// becomes:
        var subtotal = (price, quantity) =>
                        { var sub = price * quantity;
                          return sub;
                        };
```

Objects and Functions Together



Scope In JavaScript

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

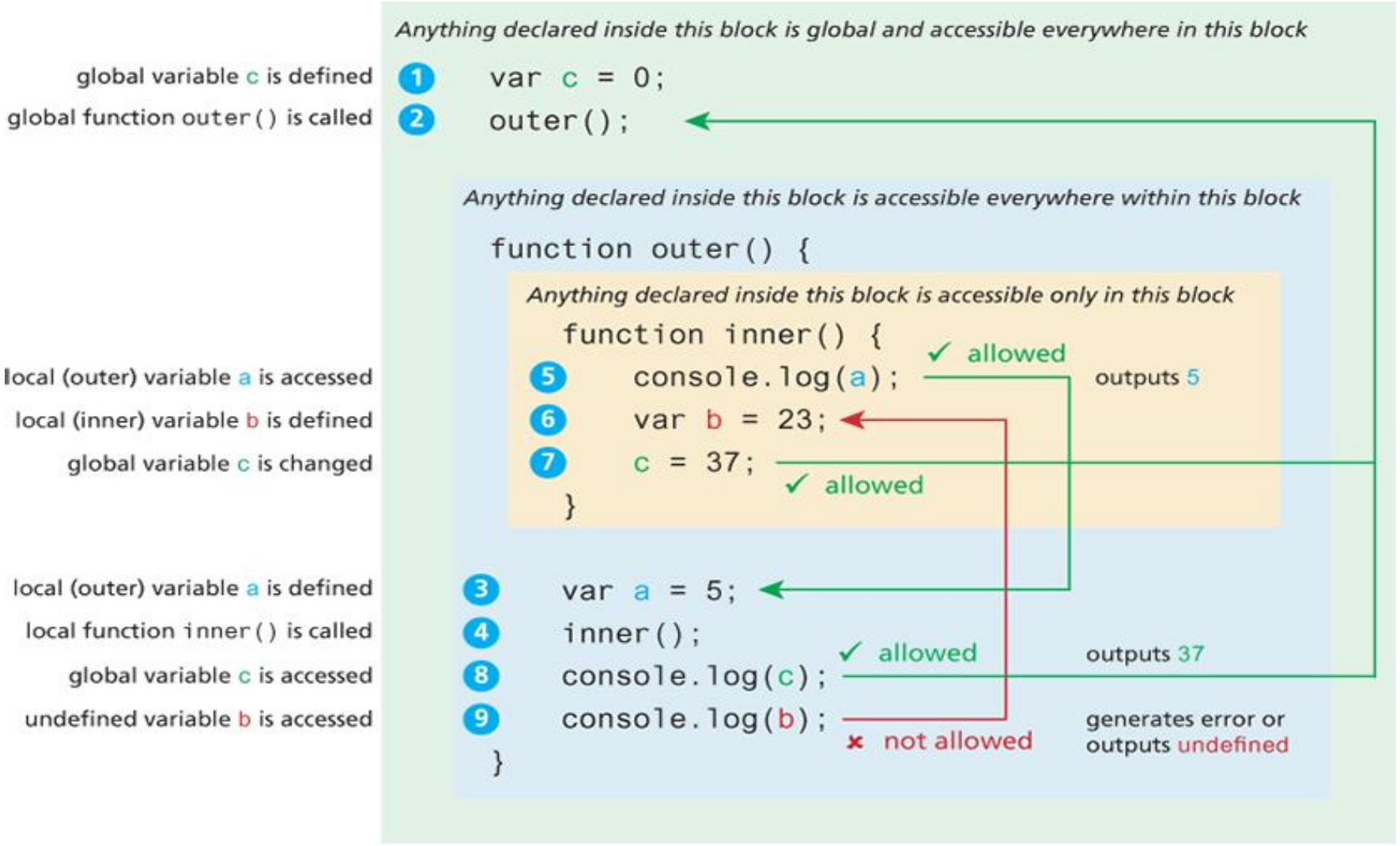
But an outer function can't look into an inner function

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

Scope In JavaScript



```
function parent() {
    var foo = "within parent";

    function child() {
        var bar = "within child";
        return foo + " " + bar;
    }

    return child();
}

var temp = parent();
alert("temp = " + temp);
```

Nothing surprising here ...
A nested function has access to variables in its parent

The temp variable is going to simply contain the value returned from the inner child() function

Closure example

```
function parent() {
    var foo = "within parent";

    function child() {
        var bar = "within child";
        return foo + " " + bar;
    }

    return child;
}

var temp = parent();
alert("temp = " + temp);
```

Notice that we are *not* invoking the inner function now

Instead, we are returning the inner function (and not its return value as in previous example)

After parent executes, we might expect that any local variables defined within the function to be gone (i.e., garbage collected).

Yet in this example, this is *not* what happens. The local variable foo sticks around even after it is finished executing. Why?

This happens because the parent function has become a closure.

A closure is a special object that contains a function and its scope environment. A closure thus lets a function continue to access its design-time lexical scope even if it is executed outside its original parent.

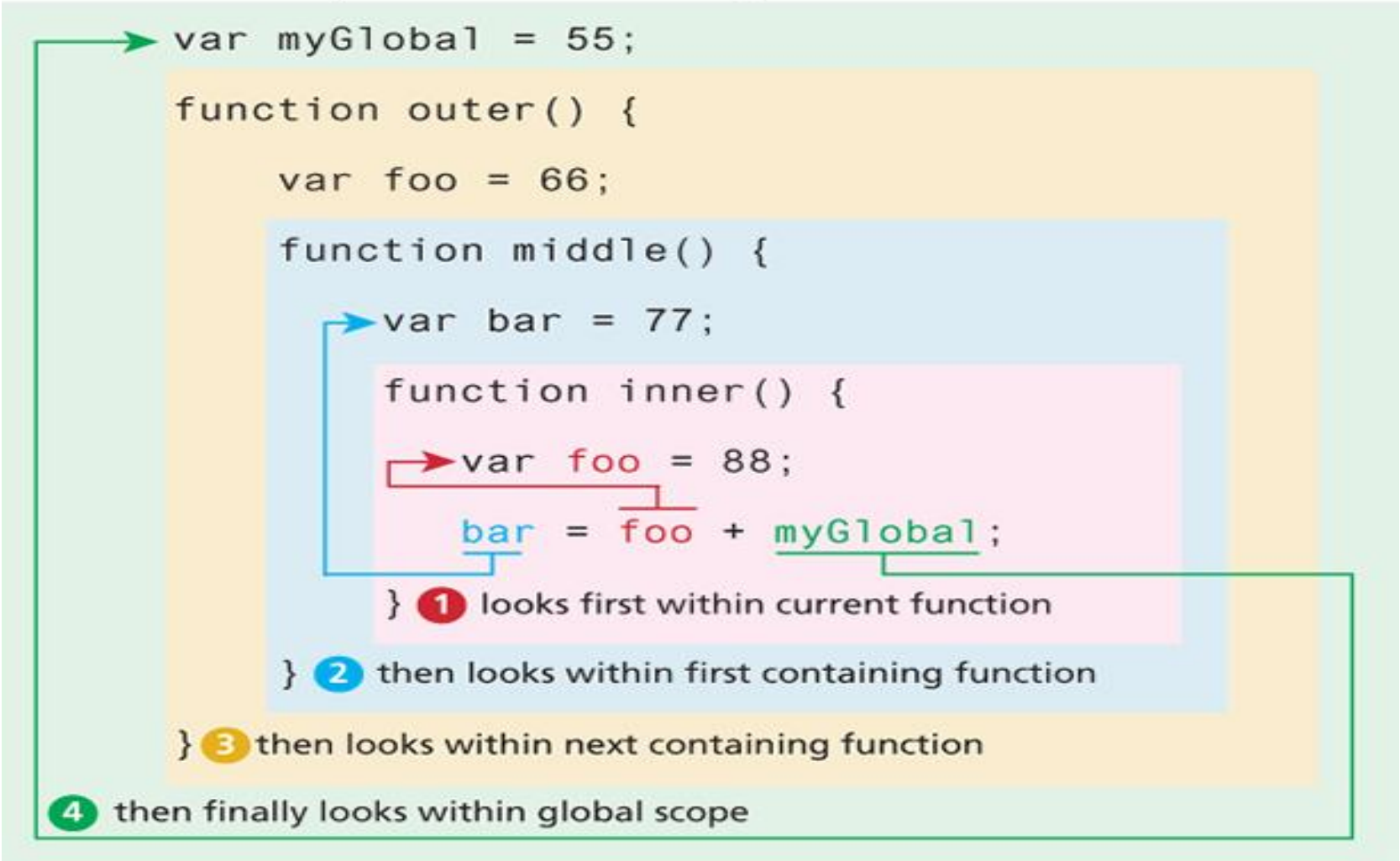
The temp variable is now going to contain the inner child() function


```
alert("temp() = " + temp());
```

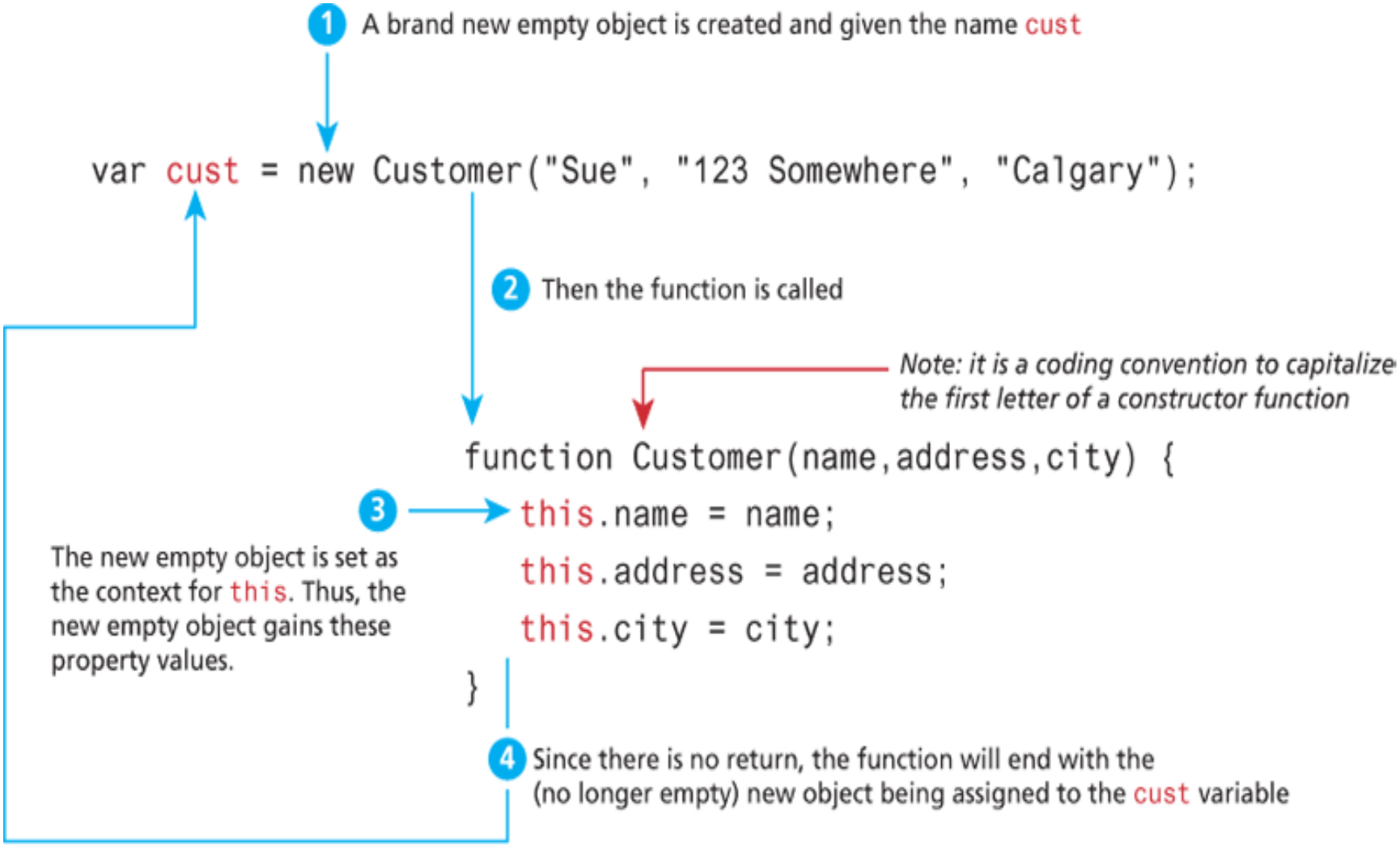
The temp function still has access to the foo variable within the parent function even though the temp function is now outside its declared lexical scope (i.e., the parent function)

Scope In JavaScript

Remember that scope is determined at design-time



Function Constructors



Object Prototype

- While the constructor function is simple to use, it can be an inefficient approach for objects that contain methods.

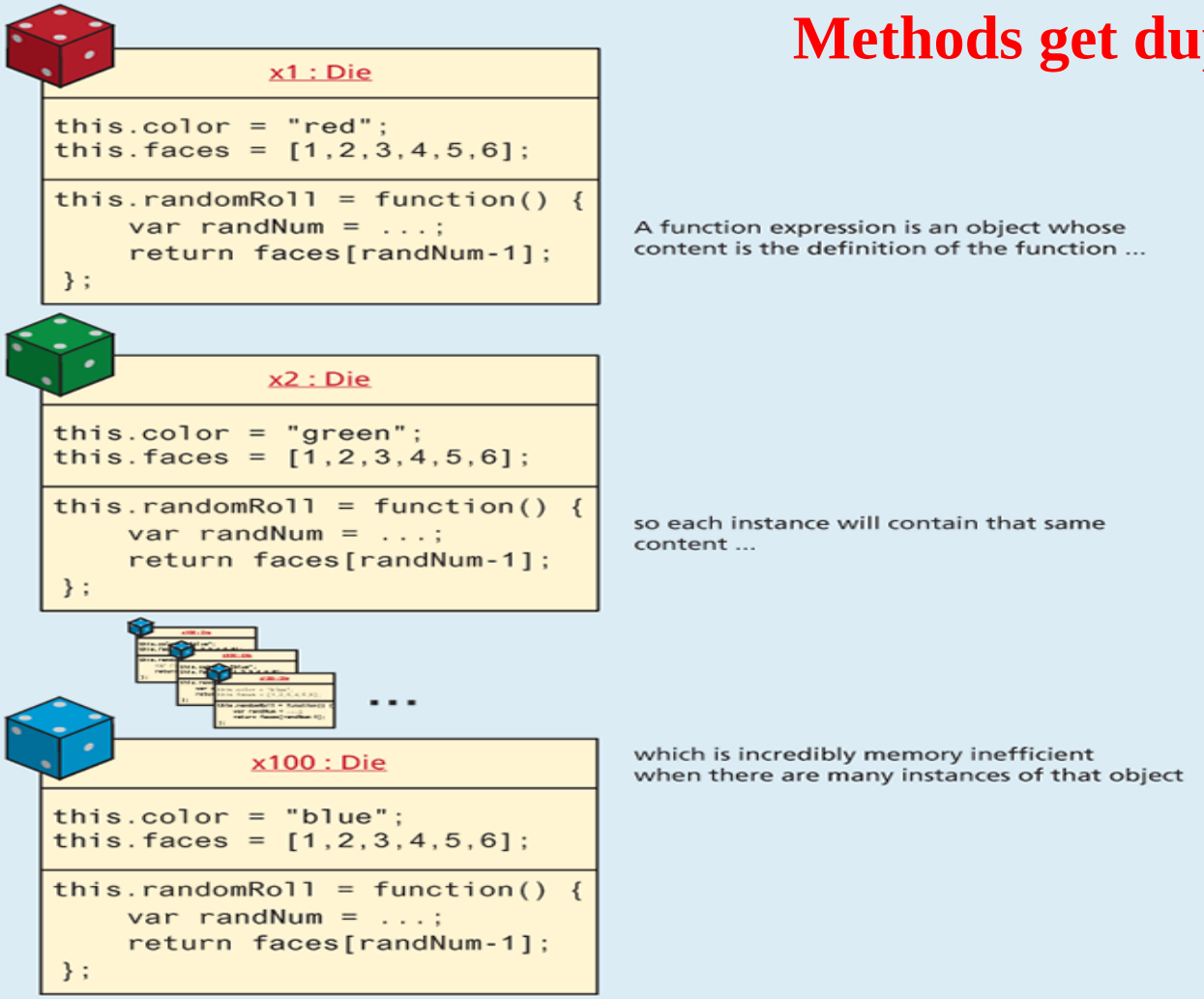
- **Prototypes** are an essential syntax mechanism in JavaScript, and are used to make JavaScript behave more like an object-oriented language.

Listing 8.18 Sample inefficient function constructor and some instances

```
function Die(col) {
   this.color=col;
   this.faces=[1,2,3,4,5,6];
   this.randomRoll = function() {
      var randNum = Math.floor((Math.random() * this.faces.length) +
1);
      return faces[randNum-1];
   };
}
// now create a whole bunch of Die objects and start rolling 'em!
var x1 = new Die("red");
alert(x1.randomRoll());
var x2 = new Die("green");
alert(x2.randomRoll());
// …
var x100 = new Die("blue");
```

Object Prototype

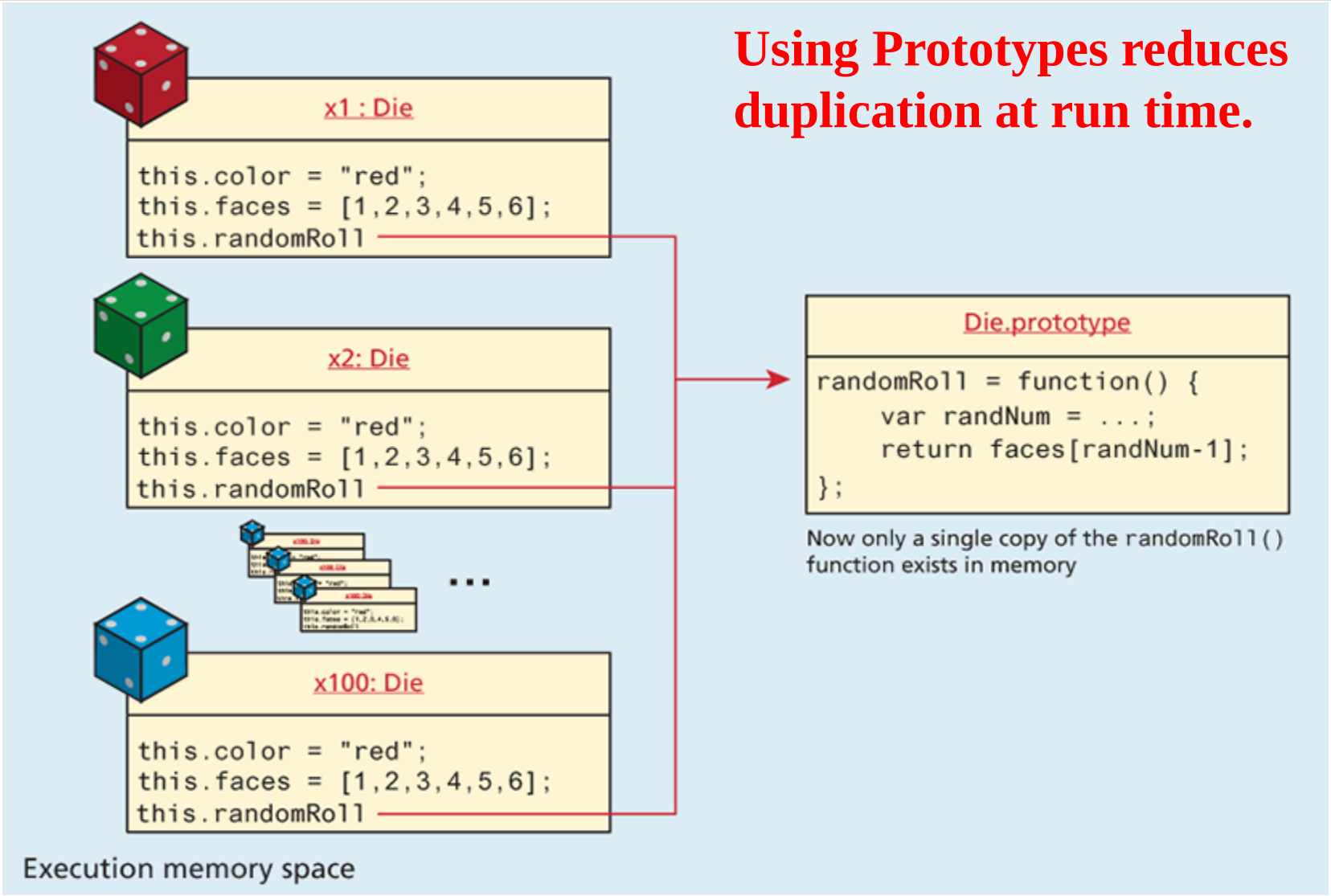
Methods get duplicated…



Listing 8.19 Using a prototype

```
function Die(col) {
   this.color=col;
   this.faces=[1,2,3,4,5,6];
}
Die.prototype.randomRoll = function() {
      var randNum = Math.floor((Math.random() * this.faces.length) +
1);
      return faces[randNum-1];
};
// now create a whole bunch of Die objects
var x1 = new Die("red");
alert(x1.randomRoll());
var x2 = new Die("green");
alert(x2.randomRoll());
…
```

Object Prototype



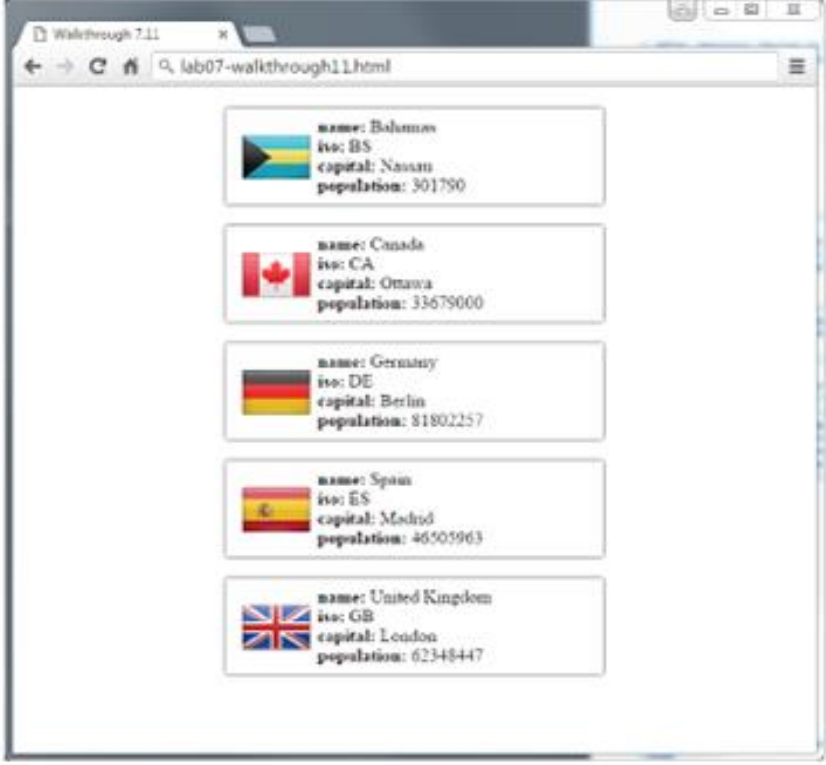
Object Prototype

- Using **Prototypes** to extend other objects:

```
String.prototype.countChars = function (c) {
        var count=0;
        for (var i=0;i<this.length;i++) {
                if (this.charAt(i) == c)
                        count++;
                }
        return count;
}
var msg = "Hello World";
console.log(msg + "has" + msg.countChars("l") + " letter l's");
```

An Extended Example

This is what the extended example will look like in the browser once the JavaScript is completed.



example.html

```
<!DOCTYPE html>
<html>
<head>
   <meta charset="utf-8">
   <title>Example</title>
   <link rel="stylesheet" type="text/css" href="css/styles.css" />
</head>
<body>
<script type="text/javascript" language="javascript" src="js/example.js"></script>
</body>
</html>
```

Notice that there is no markup within the <body> other than a <script> reference.

Here we are including the script where we want the code to insert the markup.

example.js ← Here the JavaScript is saved within an external file.

```
// define constructor function for Country objects
function Country(name, iso, capital, population) {
    "use strict";
    this.name = name;
    this.iso = iso;
    this.capital = capital;
    this.population = population;
}

/* wrap this into an IIFE  */
(function () {
    "use strict";
    // create an array of sample country objects
    var countries = [
        new Country("Bahamas", "BS", "Nassau", 301790),
        new Country("Canada", "CA", "Ottawa", 33679000),
        new Country("Germany", "DE", "Berlin", 81802257)
    ];

    // you can also push each new country object onto the end of the array
    countries.push(new Country("Spain", "ES", "Madrid", 46505963));
    countries.push(new Country("United Kingdom", "GB", "London", 62348447));

    // now loop through all this array of country objects
    for (var i = 0; i < countries.length; i++) {
        var c = countries[i];
        document.write("<div class='box'>");
        document.write("<img src='flags/" + c.iso + ".png' class='boxImg'>");

        // here is something we haven't seen: the in loop
        // which loops through properties in an object
        for (var propertyName in c) {
            document.write("<strong>");
            document.write(propertyName + ": ");
            document.write("</strong>");
            document.write(c[propertyName]);
            document.write("<br>");
        }
        document.write("</div>");
    }
})();
```

Remember that arrays are usually created using the [ ] literals.

Remember that when we use the function constructor we must use the new keyword.

Properties of an object are usually accessed using dot notation, but can also, as is the case here, by referencing the property name as a string within [ ] brackets.

IIFE combines the definition of an anonymous function with its execution.