Fundamentals of Web Development

Third Edition by Randy Connolly and Ricardo Hoar

Chapter 4

(In the book this is chapters 8,9 and 10)

JavaScript Part1:

Language Fundamentals

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 scripting language (interpreted)
- *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: 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.

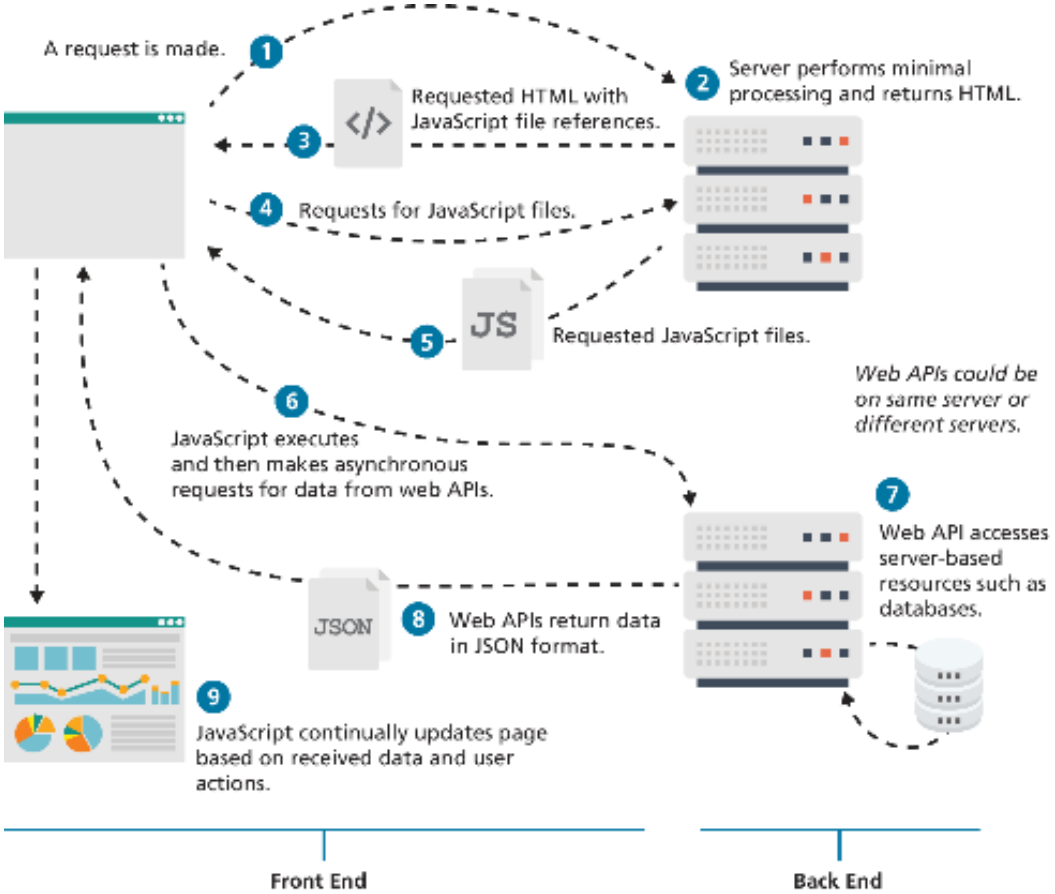
Client-Side Scripting: Disadvantages

- 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 the official specification 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 14th Edition (generally referred to as ES14 or ES2023)

JavaScript and Web 2.0



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 in the HTML document
- 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

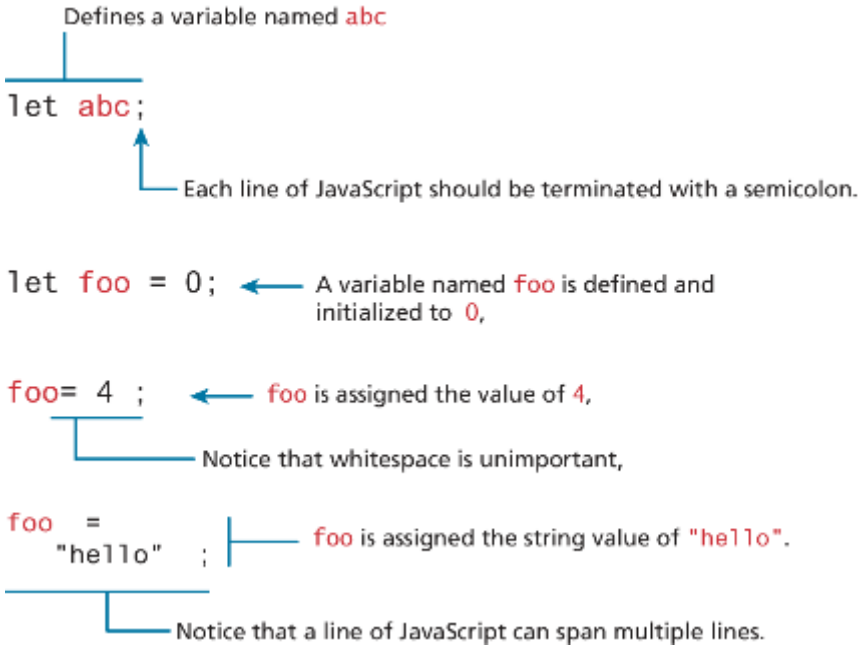
Inline JavaScript

Embedded JavaScript

Variables and Data Types

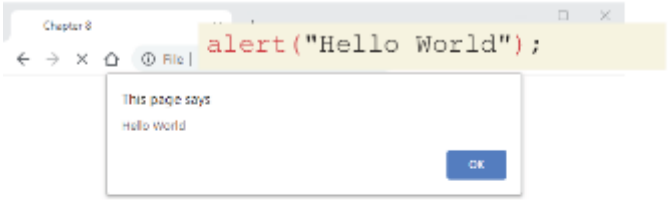
Variables in JavaScript are **dynamically typed**, meaning that you do not have to declare the type of a variable before you use it.

To declare a variable in JavaScript, use either the **var**, **const**, or **let** keywords.

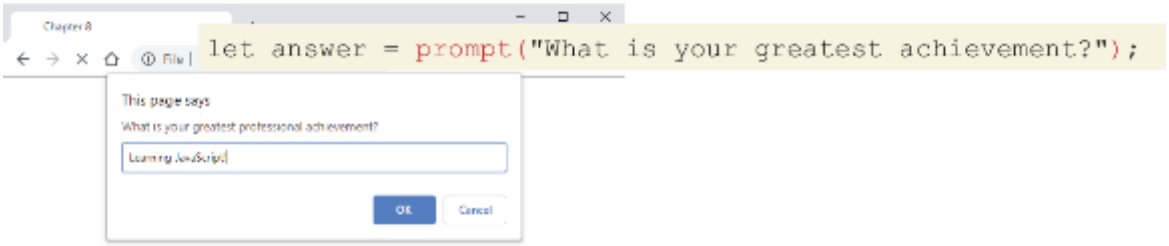


Note: When you copy/paste code in Javascript, the quotes can be altered to non-valid quotes, so always verify that you have the correct quotes (‘, “) after a copy/paste.

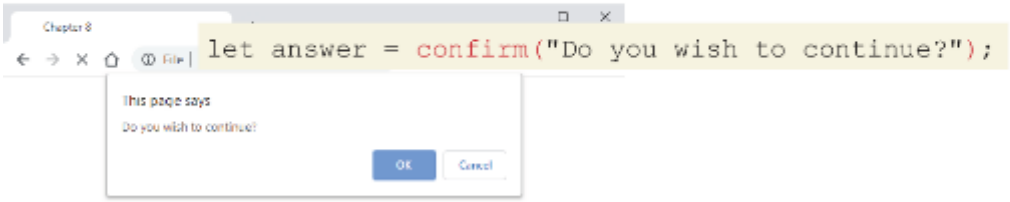
JavaScript Output



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.

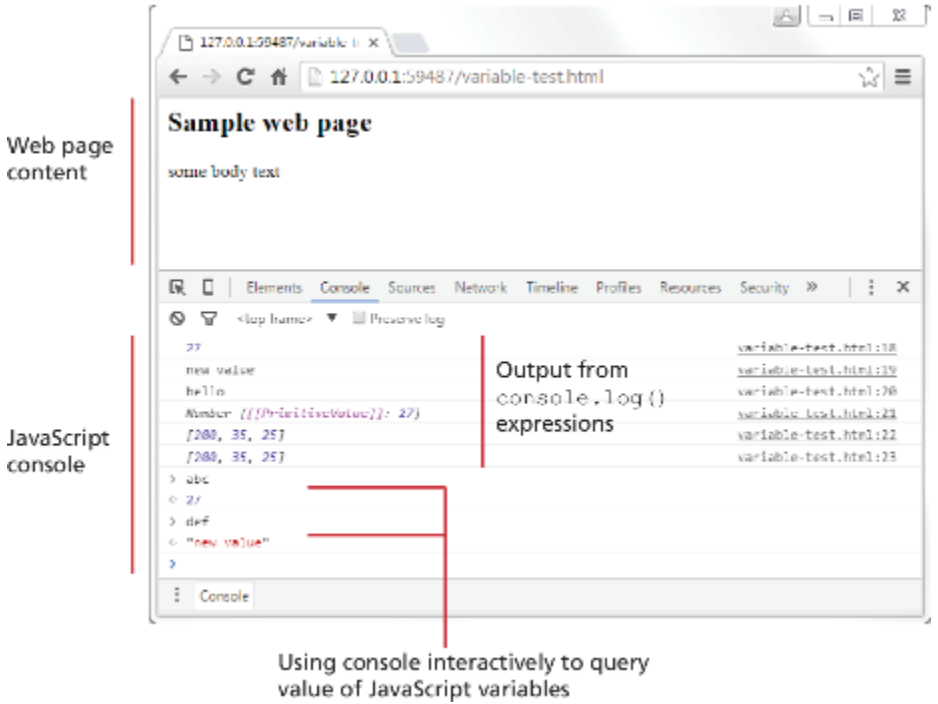
```
let answer = prompt("Please enter your name:");
alert('your name is ' + answer);
```

JavaScript Output (ii)

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

```
let answer = prompt("Please enter your name:");
document.write('<h1>your name is ' + answer + '</h1>');
```

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



Data Types

JavaScript has two basic data types:

- **reference types** (usually referred to as objects)
- **primitive types** (i.e., non-object, 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.
- **bigint** Represents an integer that can be very large (> 2^{53})
- **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** Is a key that is guaranteed to be unique created for a given value Symbol.for(“Selem”)

Primitive vs Reference Types

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

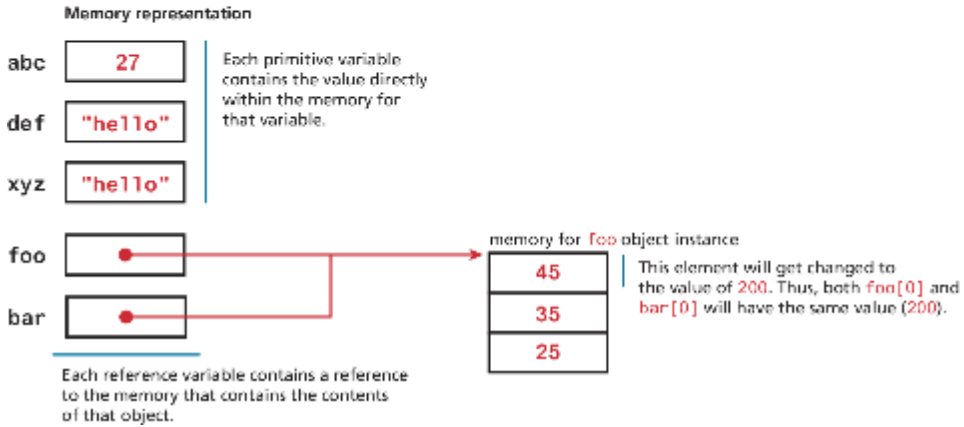
In contrast, object variables contain a reference or pointer to the block of memory associated with the content of the object.

```
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**.

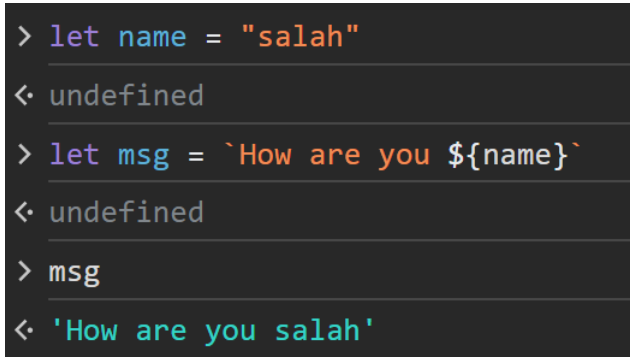
Later we will also frequently make use of several vital objects that are not part of the language but are part of the browser environment. These include the **document**, **console**, and **window** objects.

```
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 (+).

Or use template strings:




```
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 (<, >, ==, <=, >=, !=, !==, ===).

```
let answer = prompt("Please enter your name:");

if(!answer) console.log('the answer is empty');
  else console.log('Great: ' + answer);
```

```
> let y
< undefined
> y
< undefined
> let z
< undefined
> z = null
< null
> z
< null
> z == y
< true
> z === y
< false
```

Switch statement

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

Note: Better avoid switch syntax because it can easily lead to errors.

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";
}
```

```
let answer = prompt("Please enter your name:");
console.log( (!answer)? 'the answer is empty': 'Great: ' + answer);
```

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 (0/0, sqrt(-1)), and undefined

Try:

```
let a = 2;
let b;
!!a ;   // true -> truthy
!a ;   // false -> falsy
!!b;    // ??
```

While and do . . . while Loops

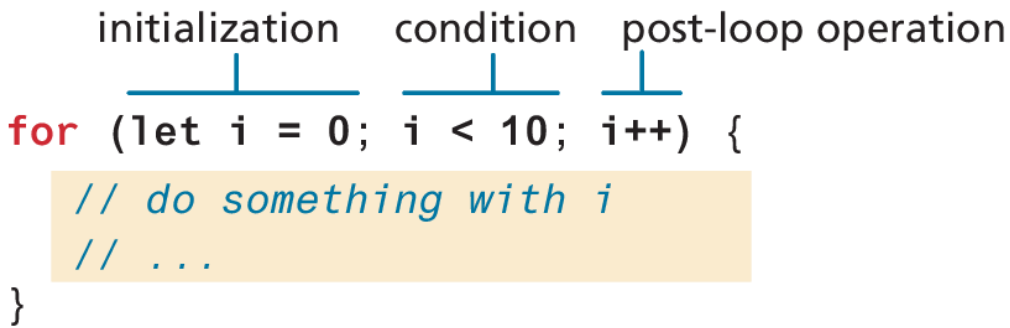
While and **do…while** loops execute nested statements repeatedly as long as the while expression evaluates to true.

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);
```

For Loops

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



Source code: chapter04/03_loops

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.

Source code: chapter04/01_exceptions

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 = [“KSA", “Italy",
"Germany", "Nigeria",
“Vietnam", “Mali"];

// 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];
```

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);
}
```

```
const countries = ["KSA", "Japan", "Oman"];

document.write('<table><tr><th>Country</th></tr>');
for ( let c of countries){
    document.write('<tr><td>' + c + '</td><tr>')
}
document.write('</table>');
```

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.

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"]
```

Source code: chapter04/02_objects/01_creation.js

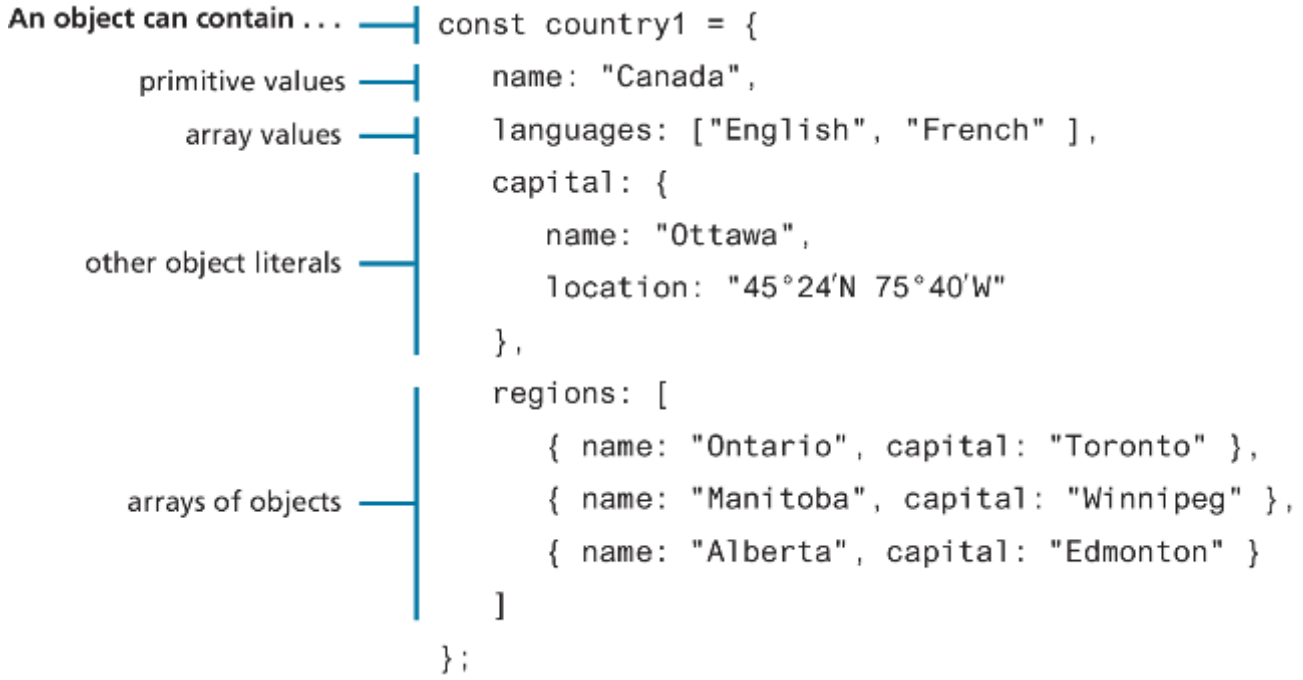
Object Creation Using Object Constructor

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



Exercise: (object creation)

- Create an object that represents KSA, you have to include an id (the phone country code), the name, and an object that represents its currency. The currency is characterized by a name, a value against the dollar, a list of available coins and a list of available banknotes.

```
let ksa = {
  id: ‘966',
  name: 'KSA',
  currency: {
    name: 'riyal',
    valueAgaintDollar: 0.2666,
    coins: [ 0.01, 0.05, 0.1, 0.2, 0.5 ],
    banknotes: [ 1,   5,  10,  20,
      50, 100, 200, 500]
   }
}
```

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)

You have to use the name of the property to access

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["city"];
```

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;
```

Source code: chapter04/02_objects/03_object_destructuring.js

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"
}';
```

Try to access: https://mocki.io/v1/689a2e6a-39b0-4a2d-9f64-5baf8cf36571

Source code: chapter04/04_json/index.html

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.

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

```
const countries = ['{"id": "01", "name": "KSA"}',
                   '{"id": "02", "name": "Japan"}',
                   '{"id": "03", "name": "Oman"}'];

document.write('<table><tr><th>ID</th><th>Country</th></tr>')
;
for ( let c of countries){
   let co = JSON.parse(c);
   document.write('<tr><td>' + co.id + '</td><td>’
                              + co.name + '</td><tr>')
}
document.write('</table>');
```

Source code: chapter04/04_json/index02.html

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

```
// 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");
```

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);     // 3 + undefined -> NaN
```

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","alia");
let boys = concatenate("jamal","nasir");

console.log(girls); // "fatima hema jane alia“
console.log(boys); // "jamal nasir“
```

Example:

```
let sum = function ( ...args ) { let s = 0; for (let e of args ) s += e; return s}
```

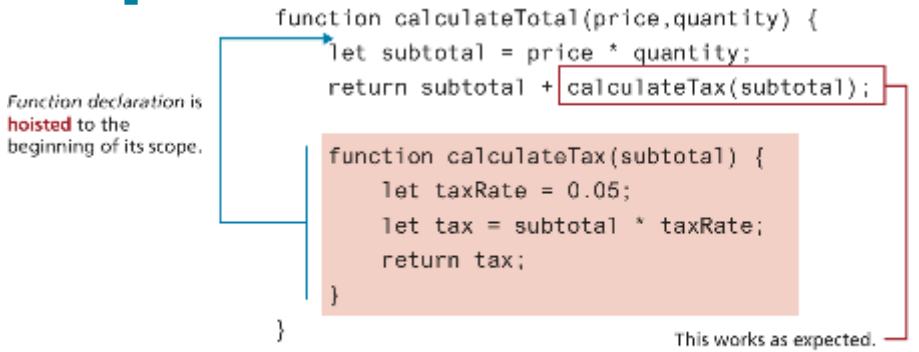
Source code: chapter04/04b_functions/01_rest_parameters.html

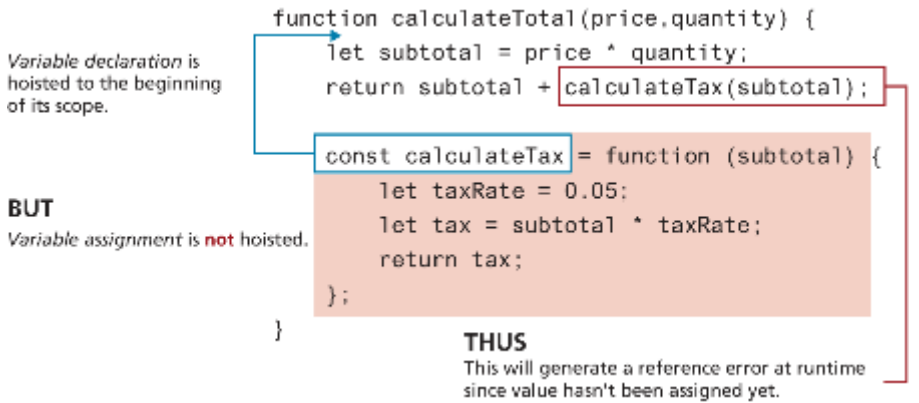
Hoisting in JavaScript

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

Hoisting is moving declarations to the top.

Note: the assignments are NOT hoisted.

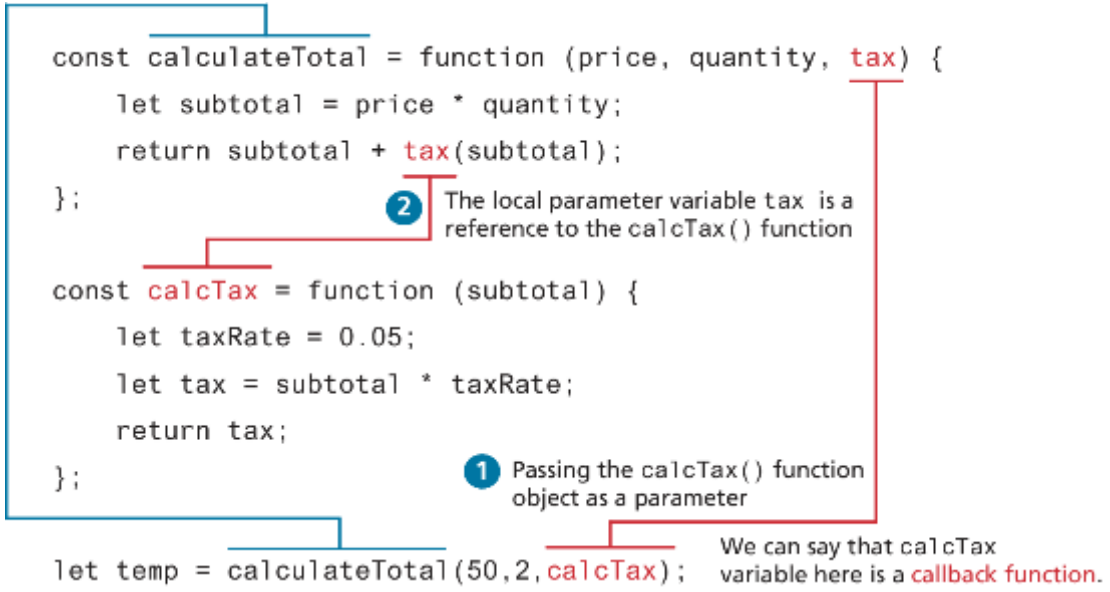




Callback Functions

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.



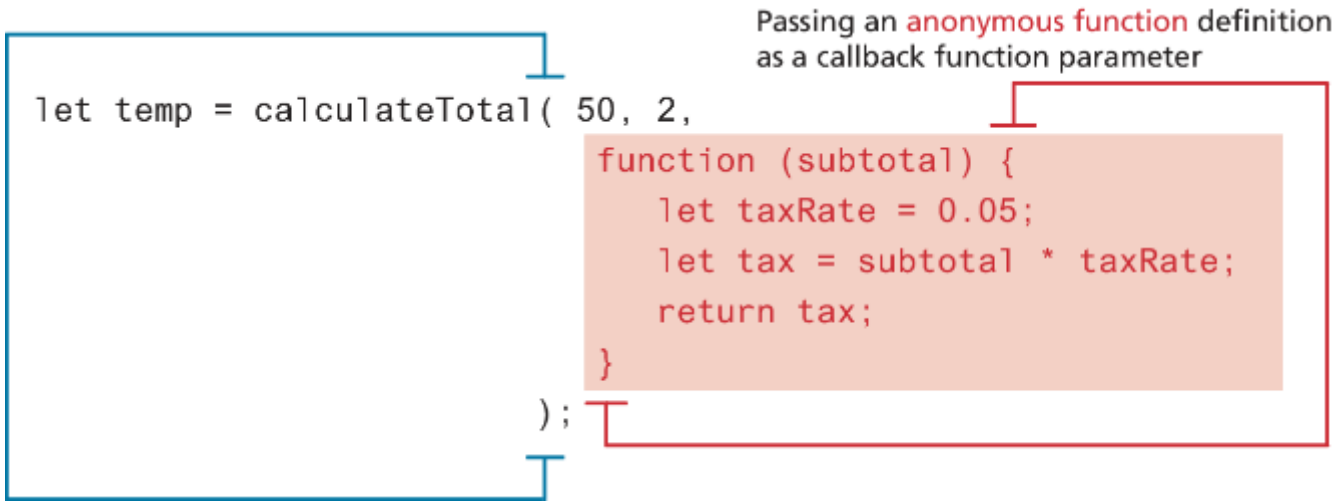
Example:

```
let sum = function( a ) { let s = 0; for (let e of a ) s += e; return s};
let map = function(f, ...args) { let s = [ ], i =0 ; for (let e of args ) s[i++] = f(e); return s};
map(sum, [3, 5, 6], [4, 7, 8], [8, 5])
```

Source code: chapter04/05_callback

Callback Functions (ii)

We can actually define a function directly within the invocation



Objects and Functions Together

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

Note: Without the "this" keyword inside the "output" function, "brand" and "price" are not defined.

```
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());
```

Constructors as functions

The following syntax creates a constructor (Customer).

```
// constructor as a function
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());
```

Then we can create an object of that type using the **new** keyword.

We can call the output function on the created object.

Source code: chapter04/06_constructor

Arrow Syntax (a, b) => {return a + b}

Arrow syntax provide a more concise syntax for the definition of anonymous functions.

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

The arrow function version would look like the following:

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

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

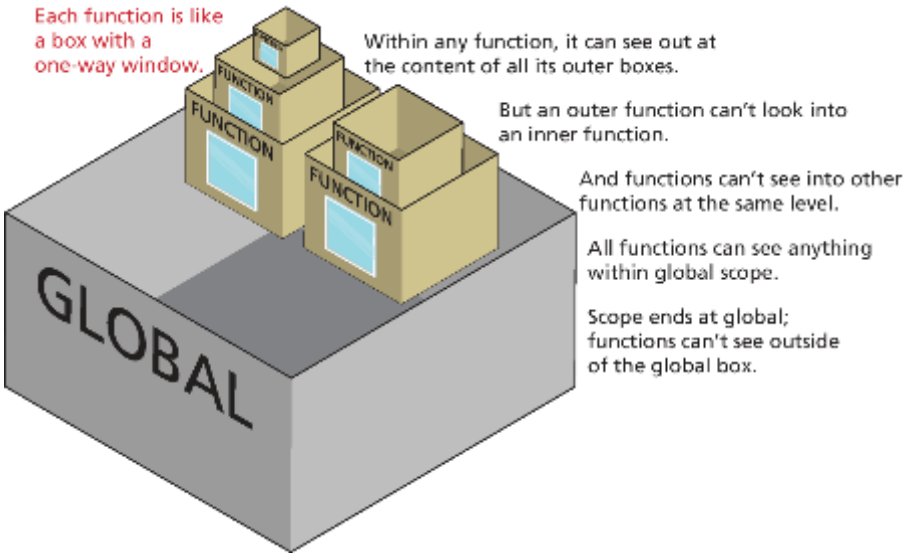
`function (a,b) {`
`  return {`
`  p1: a,`
`  p2: b`
`  }`
`}`
	`(a,b) => ({`
	`  p1: a,`
	`  p2: b`
	`  })`
		When arrow function returns an object literal, the object literal must be wrapped in parentheses.

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**,
- **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
```

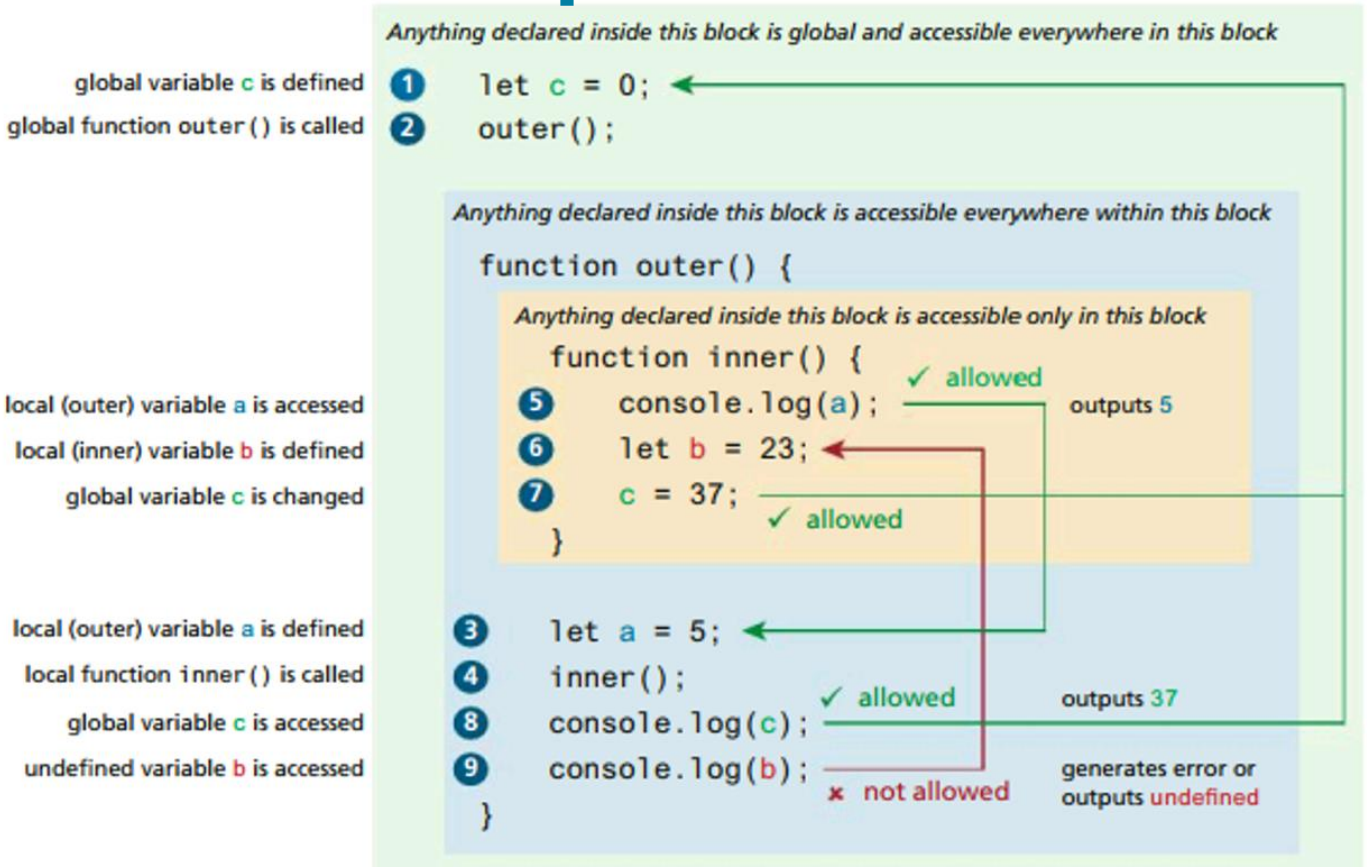
A variable will be in global scope if declared outside of a function *and* uses the `var` keyword.

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.

Function/Local Scope



Fundamentals of Web Development

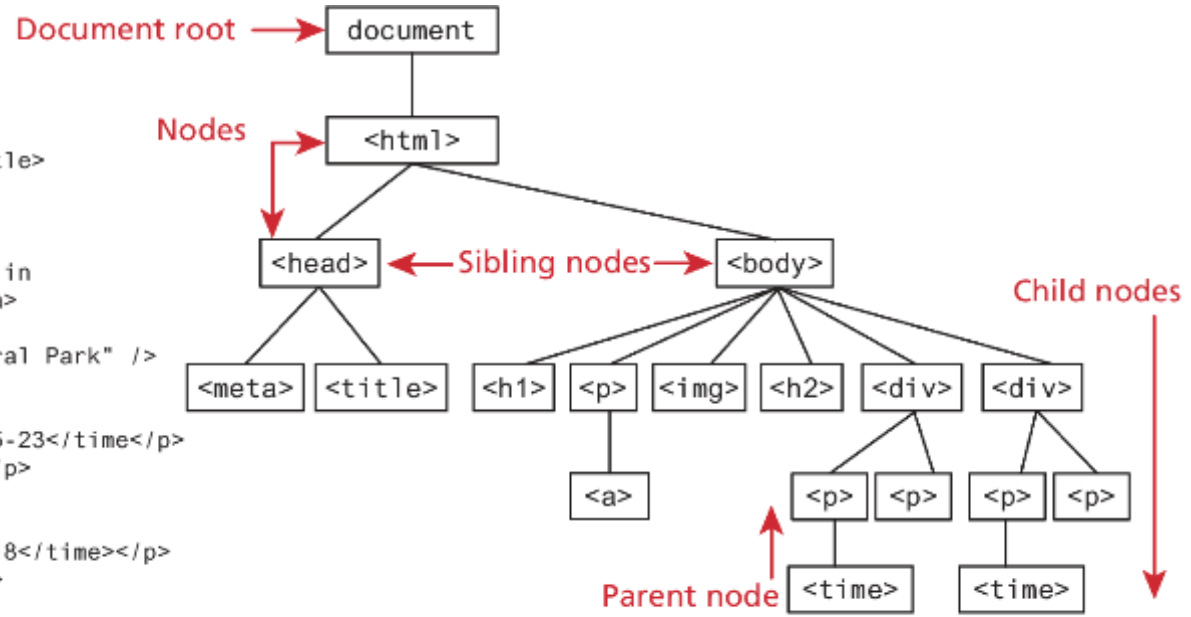
Third Edition by Randy Connolly and Ricardo Hoar

JavaScript Part 2:

Using JavaScript in the front-end

The Document Object Model (DOM)

```
<html>
<head>
   <meta charset="utf-8">
   <title>Share Your Travels</title>
</head>
<body>
   <h1>Share Your Travels</h1>
   <p>Photo of Conservatory Pond in
      <a href="#">Central Park</a>
   </p>
   <img src="park.jpg" alt="Central Park" />
   <h2>Reviews</h2>
   <div id="latestComment">
      <p>Ricardo on <time>2021-05-23</time></p>
      <p>Easy on the HDR buddy.</p>
   </div>
   <div>
      <p>Susan on <time>2021-11-18</time></p>
      <p>I love Central Park.</p>
   </div>
</body>
</html>
```



Document Object

The **DOM document object** is the root JavaScript object representing the entire HTML document. It is globally accessible via the **document** object reference.

The properties of a document cover information about the page. Some are read-only, but others are modifiable. Like any JavaScript object, you can access its properties using either dot notation or square bracket notation

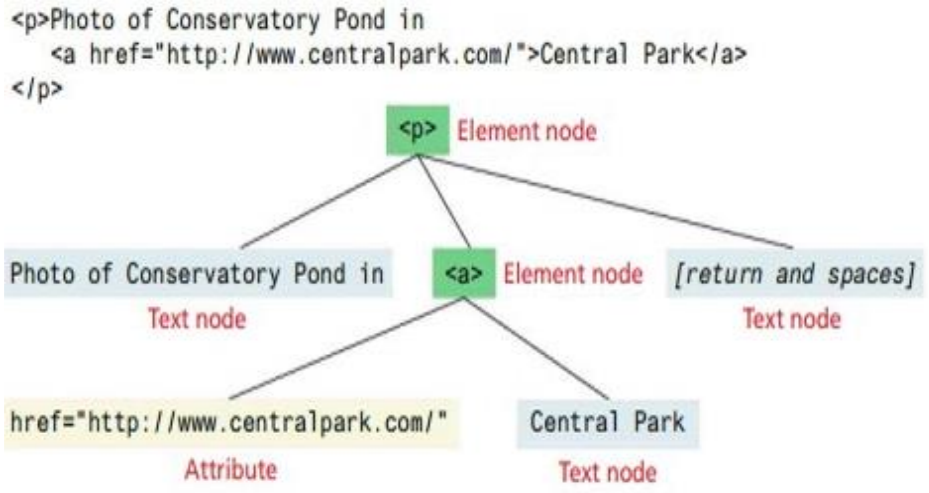
```
// retrieve the URL of the current page
let a = document.URL;
// retrieve the page encoding, for example ISO-8859-1
let b = document["inputEncoding"];          // equivalent to document.inputEncoding
```

DOM Nodes and NodeLists

In the DOM, each element within the HTML document is called a **node**.

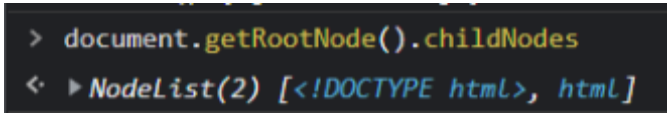
The DOM also defines a specialized object called a **NodeList** that represents a collection of nodes. It operates very similarly to an array.

Many programming tasks that we typically perform in JavaScript involve finding one or more nodes and then modifying them.



Some Essential Node Object Properties

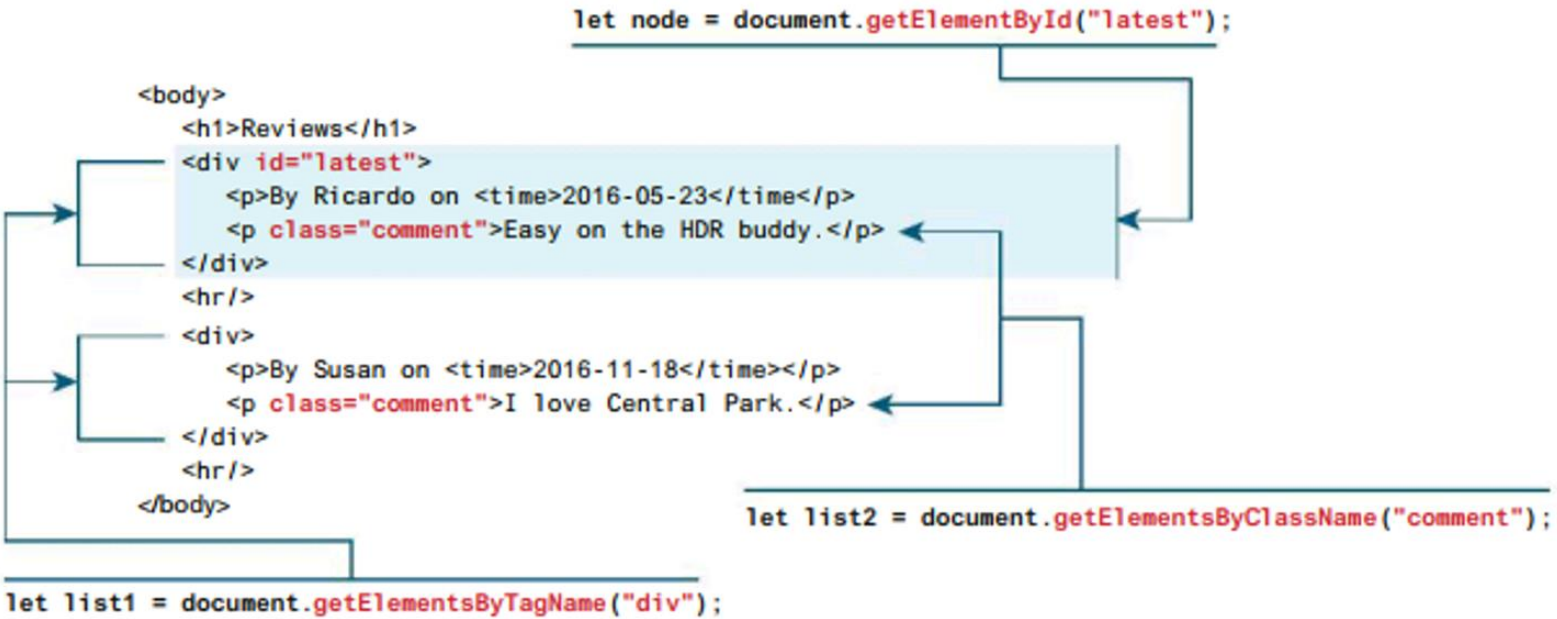
- **childNodes** A NodeList of child nodes for this node



- **firstChild** First child node of this node
- **lastChild** Last child of this node
- **nextSibling** Next sibling node for this node
- **nodeName** Name of the node
- **parentNode** Parent node for this node
- **previousSibling** Previous sibling node for this node
- **textContent** Represents the text content (stripped of any tags) of the node

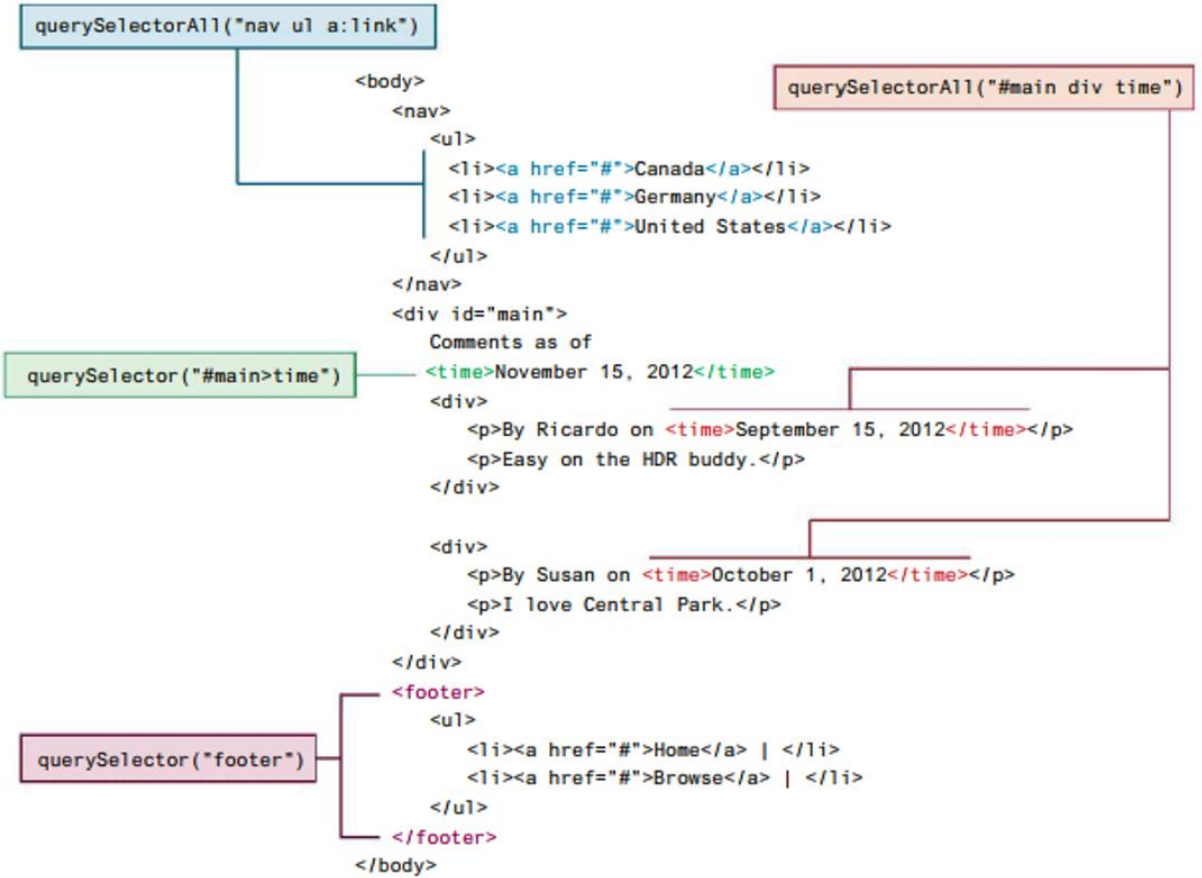
document object Methods: Selection

They allow you to select one or more document elements. The oldest 3 are: **getElementById("*id*"), getElementsByClassName("*name*")** and **getElementsByTagName("*name*")**



document object Methods: Query Selection

The newer **querySelector**() and **querySelectorAll**() methods allow you to query for DOM elements the same way you specify CSS styles



Element Node Object

Element Node object represents an HTML element in the hierarchy, contained between the opening <> and closing </>.

Every Element Node has also these properties:

- **classList** A read-only list of CSS classes assigned to this element. This list has a variety of helper methods for manipulating this list.
- **className** The current value for the class attribute of this HTML element.
- **id** The current value for the id of this element.
- **innerHTML** Represents all the content (text and tags) of the element.
- **style** The style attribute of an element. This returns a CSSStyleDeclaration object that contains sub-properties that correspond to the various CSS properties.
- **tagName** The tag name for the element.

Extra Properties for Certain Tag Types

Property	Description	Tags
href	Used in <a> tags to specify the linking URL.	a
name	Used to identify a tag. Unlike id which is available to all tags, name is limited to certain form-related tags.	a, input, textarea, form
src	Links to an external URL that should be loaded into the page (as opposed to href which is a link to follow when clicked).	img, input, iframe, script
value	Provides access to the value attribute of input tags. Typically used to access the user's input into a form field.	input, textarea, submit

Accessing elements and their properties

```
<p id="here">hello <span>there</span></p>
<ul>
    <li>France</li>
    <li>Spain</li>
    <li>Thailand</li>
</ul>
<div id="main">
    <a href="somewhere.html">
        <img src="whatever.gif" class="thumb">
    </a>
</div>

<script>

const node = document.getElementById("here");
console.log(node.innerHTML); // hello <span>there</span>
console.log(node.textContent); //“hello there”

const items = document.getElementsByTagName("li");
for (let i=0; i<items.length; i++) {
    // outputs: France, then Spain, then Thailand
    console.log(items[i].textContent);
}

const link = document.querySelector("#main a");

console.log(link.href); // outputs: somewhere.html

const img = document.querySelector("#main img");
console.log(img.src); // outputs: whatever.gif

console.log(img.className); // outputs: thumb

</script>
```

LISTING 9.1 Accessing elements and their properties

Pearson

Source code: chapter04/08_dom

Changing an Element's Style

To programmatically modify the styles associated with a particular element one must change the properties of the style property for that element

For instance, to change an element's background color and add a three pixel border, we could use the following code:

```
const node = document.getElementById("someId");
node.style.backgroundColor = "#FFFF00";
node.style.borderWidth = "3px";
```

How CSS styles can be programmatically manipulated in JavaScript

While you can directly change CSS style elements via this **style** property, it is preferable to change the appearance of an element using the **className** or **classList** properties

```
<style>
    .box {
        margin: 2em; padding: 0;
        border: solid 1pt black;
    }
   .yellowish { background-color: #EFE63F; }
   .hide { display: none; }
</style>
<main>
   <div class="box">
    ...
   </div>
</main>
```

```
var node = document.querySelector("main div");

1 node.className = "yellowish";
    This replaces the existing class specification with this one. Thus the <div> no longer has the box class

2 node.classList.remove("yellowish");
  node.classList.add("box");
    Removes the specified class specification and adds the box class

3 node.classList.add("yellowish");
    Adds a new class to the existing class specification

4 node.classList.toggle("hide");
    If it isn't in the class specification, then add it

5 node.classList.toggle("hide");
    If it is in the class specification, then remove it
```

Equivalent to:

```
1 <div class="yellowish">

2 <div class="">
  <div class="box">

3 <div class="box yellowish">

4 <div class="box yellowish hide">

5 <div class="box yellowish">
```

InnerHTML vs textContent vs DOM Manipulation

You can programmatically access the content of an element node through its **innerHTML** or **textContent** property. These properties can also be used to modify the content of any given element.

For instance, you could change the content of the using the following:

```
const div = document.querySelector("#main");
div.innerHTML = '<a href="#"><img src="ab.gif"></a>';
```

This replaces the existing content with the new content. **But, every time innerHTML is set, the HTML has to be parsed, a DOM constructed, and inserted into the document. This takes time.**

Exercise 2 (innerHTML + functions)

1- Write a function expression ***makeArticle*** that produces an HTML code that represents an article containing an h2 element and three p elements as follows:

Example:

makeArticle (“manager”, “Director”, “Salah”, “Abed”, salahabed@abc.com);

This call to the function should produce the HTML code displayed on the right inside into the node with the given id.

2- Rewrite this function as an arrow function

3- Create a constructor function for the Employee entity, add a function to its properties that returns the given HTML.

** **

```
<div id=“manger”>
  <article>
    <h2>Position: Director</h2>
    <p>Name: Salah</p>
    <p>Last Name: Abed</p>
    <p>Email: salahabed@abc.com</p>
  </article>
</div>
```

Source code: chapter04/09_ex_innerhtml

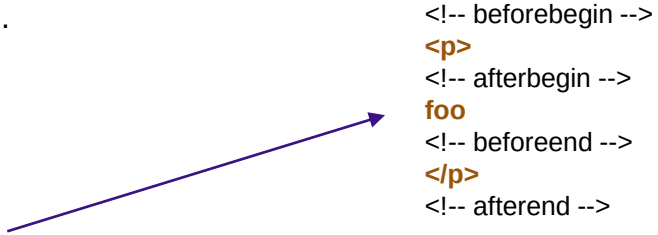
DOM family relations

Each node in the DOM has a variety of “family relations” properties and methods for navigating between elements and for adding or removing elements from the document hierarchy.

Child and sibling properties can be an unreliable mechanism for selecting nodes and thus, in general, you will instead use selector methods

DOM Manipulation Methods

- **appendChild** Adds a new child node to the end of the current node.
- **createElement** Creates an HTML element node.
- **createTextNode** Creates a text node.
- **insertAdjacentElement** Inserts a new child node at one of four positions relative to the current node.
- **insertAdjacentText** Inserts a new text node at one of four positions relative to the current node.
- **removeChild** Removes a child from the current node.
- **replaceChild** Replaces a child node with a different child.

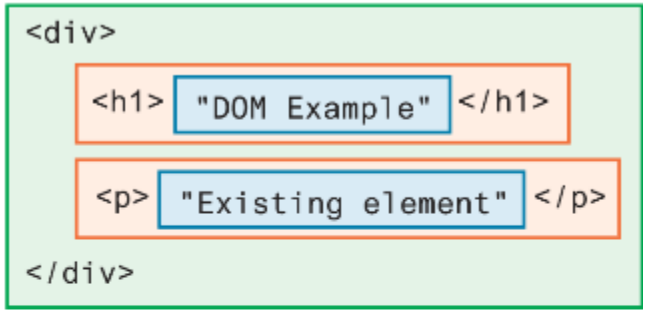


Visualizing the DOM modification

We want to add a new to this :

```
<div id="first">
    <h1>DOM Example</h1>
    <p>Existing element</p>
</div>
```

Visualizing the DOM elements



1 Create a new text node

"this is dynamic"

```
const text = document.createTextNode("this is dynamic");
```

2 Create a new empty element

```
const p = document.createElement("p");
```

Visualizing the DOM modification (ii)

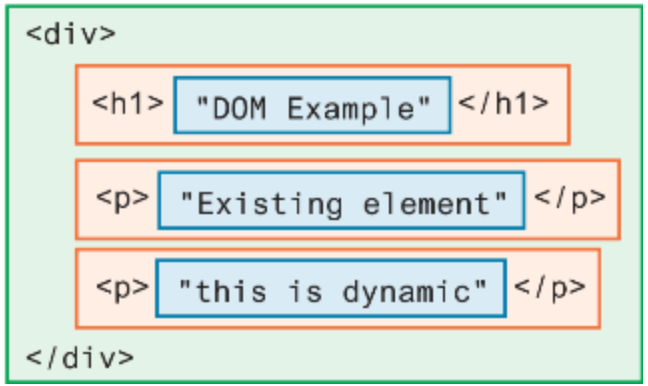
3 Add the text node to new <p> element

```
p.appendChild(text);
```

4 Add the <p> element to the

```
const first = document.getElementById("first");
first.appendChild(p);
```

```
<div id="first">
    <h1>DOM Example</h1>
    <p>Existing element</p>
    <p>this is dynamic</p>
</div>
```



Same Exercise p65 (DOM manipulation)

Modify makeArticle to use DOM manipulation functions (createElement, appendChild, etc).

```
<article>
  <h2>Position: Director</h2>
  <p>Name: Salah</p>
  <p>Last Name: Abed</p>
  <p>Email: salahabed@abc.com</p>
</article>
```

```
4  const makeArticle = function (displayElement, position, name, lastName, email) {
5    let p = document.getElementById(displayElement);
6    let art = document.createElement("article");
7    let h2 = document.createElement("h2");
8    h2.appendChild( document.createTextNode('Position: ' + position) );
9    let p1 = document.createElement("p");
10   p1.appendChild( document.createTextNode('Name: ' + name) );
11   let p2 = document.createElement("p");
12   p2.appendChild( document.createTextNode('Last Name: ' + lastName ) );
13   let p3 = document.createElement("p");
14   p3.appendChild( document.createTextNode('Email: ' + email) );
15
16   art.appendChild(h2);
17   art.appendChild(p1);
18   art.appendChild(p2);
19   art.appendChild(p3);
20   p.appendChild(art);
21 }
22
23 function getvalue(id) {
24   return document.getElementById(id).value;
25 }
26
27 function clearDisplay(displayElement){
28   let p = document.getElementById(displayElement);
29   p.removeChild(p.firstElementChild);
30 }
```

DOM Timing

Before finishing this section on using the DOM, it should be emphasized that the timing of any DOM code is very important.

You cannot access or modify the DOM until it has been loaded.

If the DOM programming is written *after* the markup that *should* ensure that the elements exist in the DOM before the code executes.

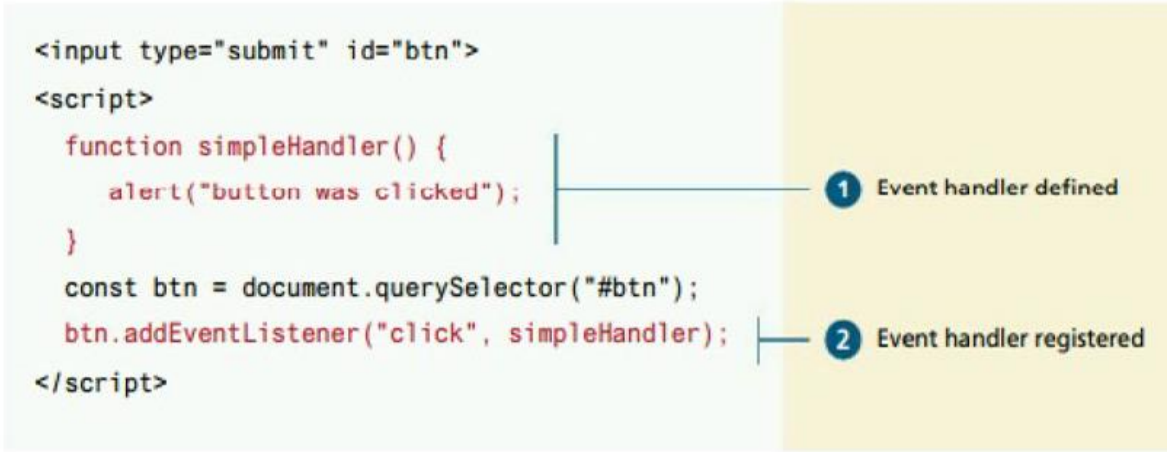
To wait until we know for sure that the DOM has been loaded requires knowledge from our next section on **event handling**.

JavaScript Event Handling

Implementing an Event Handler

An event handler is first defined, then registered to an element node object.

Registering an event handler requires passing a callback function to the **addEventListener()**



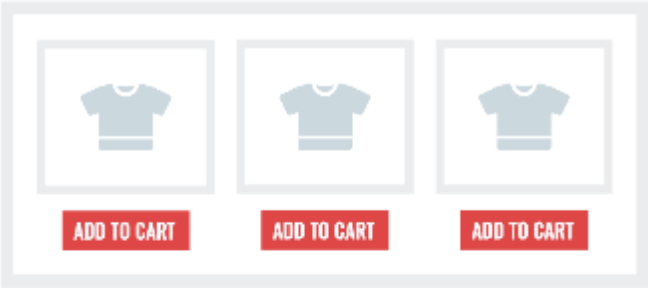
Handling events with anonymous functions

It is much more common to make use of an *anonymous function* passed to **addEventListener**()

```
const btn = document.getElementById("btn");
btn.addEventListener("click", function () {
    alert("used an anonymous function");
});

document.querySelector("#btn").addEventListener("click", function (){
    alert("a different approach but same result");
});

document.querySelector("#btn").addEventListener("click", () => {
    alert("arrow syntax but same result");
});
```

LISTING 9.3 Listening to an event with an anonymous function, three versions

Event handling with NodeList arrays




```
<ul id="list">
  <li>
    <img src="a.png" ... >
    <button>Add To Cart</button>
  </li>
  <li>
    <img src="b.png" ... >
    <button>Add To Cart</button>
  </li>
  <li>
    <img src="c.png" ... >
    <button>Add To Cart</button>
  </li>
</ul>
```

```
// select all the buttons
const btns = document.querySelectorAll("#list button");

// this won't work and will generate error
btns.addEventListener("click", function () { ... });
```

```
// instead must loop through node list ...
for (let bt of btns) {
   // ...and assign event listener to each node
   bt.addEventListener("click", function () { ... });
}
```

Remember that a node list (i.e., array of nodes) doesn't support event listeners. Only individual node objects have the addEventListener() method defined.

Source code: chapter04/12_events_window

Page Loading and the DOM

To ensure your DOM manipulation code executes *after* the page is loaded, use one of the following two different page load events.

- **window.load** Fires when the entire page is loaded. This includes images and stylesheets, so on a slow connection or a page with a lot of images, the load event can take a long time to fire.
- **document.DOMContentLoaded** Fires when the HTML document has been completely downloaded and parsed. Generally, this is the event you want to use.

Using one of these, your DOM coding can now appear anywhere, including within the **<head>** element as long as you do not try to access the DOM.

Source code: chapter04/13_events_DOM

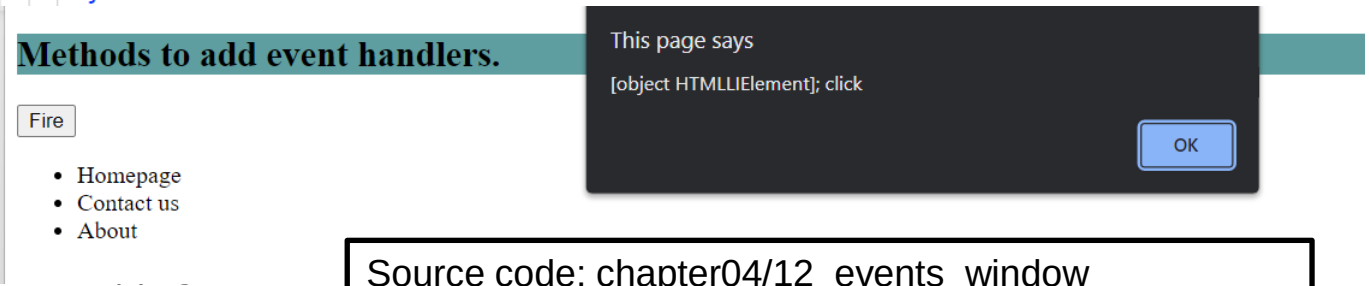
Wrapping DOM code within a DOMContentLoaded event handler

```
document.addEventListener('DOMContentLoaded', function() {
   const menu = document.querySelectorAll("#menu li");
   for (let item of menu) {
      item.addEventListener("click", function () {
         item.classList.toggle('shadow’);
      });
   }
   const heading = document.querySelector("h3");
   heading.addEventListener('click', function() {
      heading.classList.toggle('shadow’);
   });
});
```

Event Object

- When an event is triggered, the browser will construct an **event object** that contains information about the event.
- Your event handlers can access this event object simply by including it as an argument to the callback function (this event object parameter is often named *e*)

```
const heading = document.querySelector("h2");
heading.addEventListener('click', function(e) {
                                    heading.classList.toggle('shadow');
                                    alert(e.target + "; " + e.type);
                                  }
                        );
```



Source code: chapter04/12_events_window

Event Object Example


```
<ul id="menu">
  <li>Home</li>
  <li>About</li>
  <li>Products</li>
  <li>Contact</li>
</ul>
```

```
const menu = document.querySelectorAll("#menu li");
for (let item of menu) {
  item.addEventListener("click", menuHandler );
}
function menuHandler(e) {
    const x = e.clientX;
    const y = e.clientY;
    displayArrow(x,y);
    e.target.classList.toggle("selected");
    performMenuAction(e.target.innerHTML);
}
```

By receiving the event object as a parameter and using it to reference the clicked item, the `menuHandler()` function will work no matter where it is located.

Click events include the on-screen pixel location of the mouse cursor.

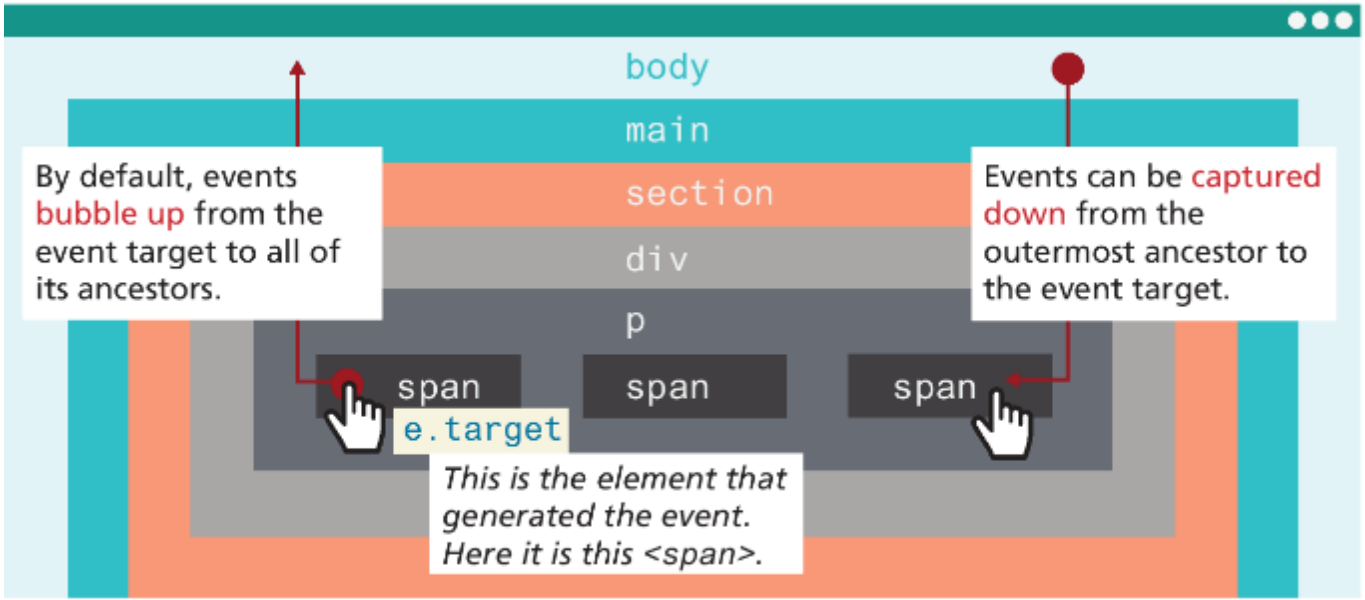
The `e.target` object in this case is referencing the clicked `<li>` item.

Event Propagation

When an event fires on an element that has ancestor elements, the event propagates to those ancestors. There are two distinct phases of propagation:

- In the **event capturing phase**, the browser checks the outermost ancestor (the <html> element) to see if that element has an event handler registered for the triggered event, and if so, it is executed (if configured to fire at this phase, *see code example_18_event_propagation*). It then proceeds to the next ancestor and performs the same steps; this continues until it reaches the element that triggered the event (that is, the **event target**).
- In the **event bubbling phase**, the opposite occurs. The browser checks if the element that triggered the event has an event handler registered for that event, and if so, it is executed.

Event capture and bubbling



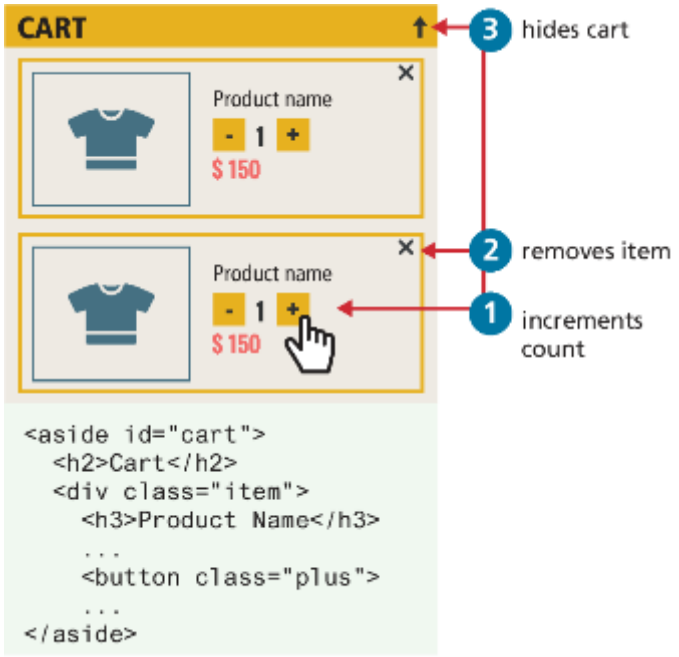
`e.currentTarget` *This is the element whose event handler is currently being executed. In this example, it could the or any of its ancestors.*

Problems with event propagation

Occasionally, the bubbling of events can cause problems. For instance consider elements nested within one another, each with its own on-click behaviors.

When the user clicks on the increment count button, the click handler for the increment <button> will trigger first. Unfortunately, it will then trigger the click event for the , and the <aside> element!

Thankfully, there is a solution to such problems. The **stopPropagation()** method of the event argument object will stop event propagation.



Stopping event propagation

```
const btns = document.querySelectorAll(".plus");
for (let b of btns) {
   b.addEventListener("click", function (e) {
      e.stopPropagation();
      incrementCount(e);
   });
}

const items = document.querySelectorAll(".item");
for (let it of items) {
   it.addEventListener("click", function (e) {
      e.stopPropagation();
      removeItemFromCart(e);
   });
}
```

```
const aside = document.querySelector("aside#cart");
aside.addEventListener("click", function () {
      minimizeCart();
});
```

LISTING 9.5 Stopping event propagation

Source code: chapter04/18_event_propagation

Event Delegation

To avoid creating duplicate event handlers for each element within a **NodeList**, an alternative is to use **event delegation** where we assign a single listener to the parent and make use of event bubbling

Suppose we have numerous image thumbnails within a parent element, similar to the following:

```
<body>
  <header>...</header>
  <main>
    <section id="list">
      <h2>Section Title</h2>
        <img ... />
        <img ... />
        ...
    </section>
  </main>
</body>
```

Event Delegation (ii)

Now what if you wanted to do something special when the user clicks the mouse on an <img>

You would probably write something like the following:

Notice that this solution adds an event listener to every <img> element.

```
const images =   document.querySelectorAll("#list img");
for (let img of images) {
            img.addEventListener("click", someHandler);
}
```

Event Delegation (iii)

Instead, we can add a single listener to the parent element, as shown in the following code

Since the user can click on any element within the <section> element, the click event handler needs to determine if the user has clicked on one of the <img> elements within it.

```
const parent = document.querySelector("#list");
parent.addEventListener("click", function (e) {
   // e.target is the object that generated the event.
   // to verify that e.target exists and that it is one of the
   // <img> elements. Note: NodeName always returns
   //upper case
   if (e.target && e.target.nodeName == "IMG") {
      doSomething(e.target);
    }
});
```

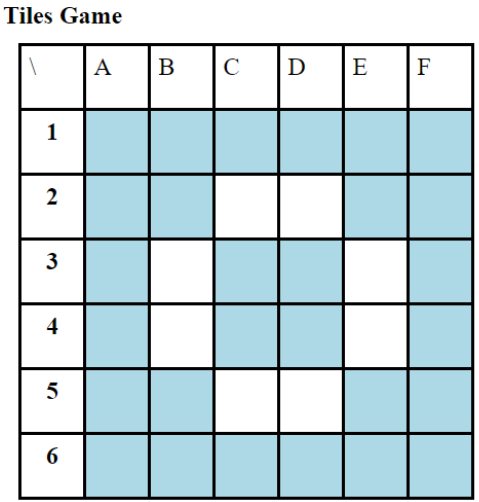
Event Types

There are many different types of events that can be triggered in the browser. Perhaps the most obvious event is the click event, but JavaScript and the DOM support several others.

- mouse events,
- keyboard events,
- touch events,
- form events, and
- frame events.

Mouse Events

- **click** The mouse was clicked on an element.
- **dblclick** The mouse was double clicked on an element.
- **mousedown** The mouse pressed down over an element.
- **mouseup** The mouse was released over an element.
- **mouseover** The mouse was moved (not clicked) over an element.
- **mouseout** The mouse was moved off of an element.
- **mousemove** The mouse was moved while over an element.



Practice: Try to build a board game where the tiles are colored when a mouse event occurs.

Source code: chapter04/15_tiles_game

Keyboard Events

- **keydown** The user is pressing a key (this happens first).
- **keyup** The user releases a key that was down (this happens last).

```
document.getElementById("pagebody").addEventListener("keydown", function (e) {
   // get the raw key code
   let keyPressed=e.key;
   // convert to string
   let character=String.fromCharCode(keyPressed);
   alert("Key " + character + " was pressed");
});
```

Form Events

- **blur** Triggered when a form element has lost focus (i.e., control has moved to a different element), perhaps due to a click or Tab key press.
- **change** Some <input>, <textarea>, or <select> field had their value changed. This could mean the user typed something or selected a new choice.
- **focus** Complementing the blur event, this is triggered when an element gets focus (the user clicks in the field or tabs to it).
- **reset** HTML forms have the ability to be reset. This event is triggered when that happens.
- **select** When the user selects some text. This is often used to try and prevent copy/paste.
- **submit** When the form is submitted this event is triggered. We can do some prevalidation of the form in JavaScript before sending the data on to the server.

Handling the submit event

```
document.querySelector("#loginForm").addEventListener("submit",
function(e) {
        let pass = document.querySelector("#pw").value;
        if (pass == "") {
                alert ("enter a password");
                e.preventDefault();     // prevents form submission
        }
});
```

LISTING 9.8 Handling the submit event

Source code: chapter04/16_form_submit_event

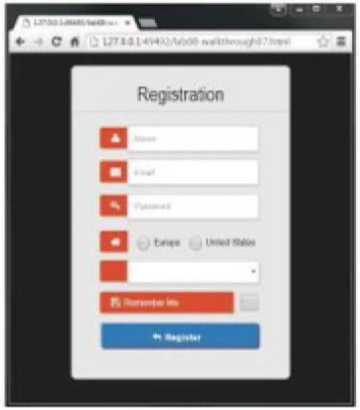
Forms in JavaScript

JavaScript within forms is more than just the client-side validation of form data; JavaScript is also used to improve the user experience of the typical browser-based form.

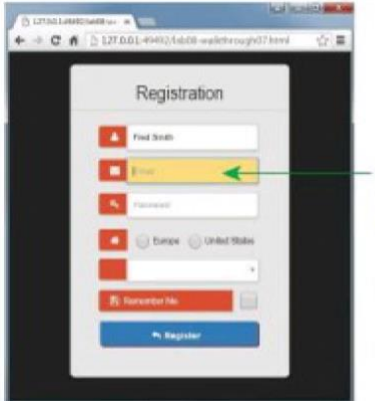
As a result, when working with forms in JavaScript, we are typically interested in three types of events:

- movement between elements,
- data being changed within a form element, and
- the final submission of the form.

Responding to form movement events



How form appears when no controls have the focus



When a control has the focus, then change its background color

```
// This function is going to get called every time the focus or blur events are
// triggered in one of our form's input elements.
function setBackground(e) {
   if (e.type == "focus") {
      e.target.style.backgroundColor = "#FFE393";
   }
   else if (e.type == "blur") {
      e.target.style.backgroundColor = "white";
   }
}

// set up the event listeners only after the DOM is loaded
window.addEventListener("load", function() {
   const cssSelector = "input[type=text],input[type=password]";
   const fields = document.querySelectorAll(cssSelector);
   for (let f of fields) {
      f.addEventListener("focus", setBackground);
      f.addEventListener("blur", setBackground);
   }
});
```

Here we use the `style` property instead of the `classList` property because of specificity conflicts (that is, attribute selectors override class selectors).

Selects the fields that will change.

Assigns the `setBackground()` function to change the background color of the control depending upon whether it has the focus.

Responding to Form Changes Events

We may want to change the options available within a form based on earlier user entry. For instance, we may want the payment options to be different based on the value of the region radio button.

We can add event listeners to the change event of the radio buttons; when one of these buttons changes its value, then the callback function will set the available payment options based on the selected region.

Validating a Submitted Form

Form validation continues to be one of the most common applications of JavaScript.

Checking user inputs to ensure that they follow expected rules must happen on the server side for security reasons (in case JavaScript was circumvented); checking those same inputs on the client side using JavaScript will reduce server load and increase the perceived speed and responsiveness of the form.

Some of the more common validation activities include email validation, number validation, and data validation.

In practice, regular expressions are used to concisely implement many of these validation checks.

Empty Field Validation

```
const form = document.querySelector("#loginForm");
form.addEventListener("submit", (e) => {
  const fieldValue = document.querySelector("#username").value;
  if (fieldValue == null || fieldValue == "") {
    // the field was empty. Stop form submission
    e.preventDefault();
    // Now tell the user something went wrong
    console.log("you must enter a username");
  }
});
```

LISTING 9.10 A simple validation script to check for empty fields

Determining which items in multiselect list are selected

```
const multi = document.querySelector("#listbox");
// using the options technique loops through each option and check if it is selected
for (let i=0; i < multi.options.length; i++) {
   if (multi.options[i].selected) {
      // this option was selected, do something with it ...
      console.log(multi.options[i].textContent);
   }
}

// the selectedOptions technique is simpler ... it only loops through the selected options
for (let i=0; i < multi.selectedOptions.length; i++) {
   console.log(multi.selectedOptions[i].textContent);
}
```

LISTING 9.11 Determining which items in multiselect list are selected

Source code: chapter04/20_multiselect_validation

Javascript validation examples

Some browsers may not support HTML5 validation, in addition to the fact that we want to have more control over how we react to bad user input, it is always better to use javascript validation:

```
<form ….. >
<input id="textA" type="number" max="100">
<button onclick="validateTextA()">Submit
</button>

<p id="output"></p>

</form>
```

```
<script>
  function validateTextA() {
    var value = "";
    if(document.getElementById("textA")
        .validity.rangeOverflow) {
      value = "The value must not be greater than 100";
    }
    document.getElementById("output")
        .innerHTML = value;
  }
</script>
```

Number Validation

Unfortunately, no simple functions exist for number validation. Using parseInt() or parseFloat(), isNAN(), and isFinite(), you can write your own number validation function.

Validating email, phone numbers, or social security numbers would include checking for blank fields and making use of isNumeric and regular expressions.

```
function isNumeric(n) {
        return !isNaN(parseFloat(n)) && isFinite(n);
}
// you have to use both because 1/0 is considered a number
```

Submitting Forms using JS

To submit a form using JavaScript simply call the **submit()** method:

```
const formExample = document.getElementById("loginForm");
formExample.submit();
```

This is often done in conjunction with calling preventDefault() on the submit event.

Javascript validation examples (2)

Validating that two entered emails are the same: We can use ***onchange*** *as it* is triggered when we type enter or leave the field, so this is the right event handler to use.

```
<form>
  <label>Preferred email address:
  <input type="email" id="email_addr" name="email1"
placeHolder="user@provider.domain" required></label>

  <label>Repeat email address:
  <input type="email" id="email_repeat" name="email2"
required onchange="check()"></label>

  <input type = 'submit' value = "Send">
</form>
```

```
<script>
  function check() {
    var email1 =
        document.getElementById('email_addr');
    var email2 =
        document.getElementById('email_repeat');
    if ( email1.value !== email2.value) {
                alert("The two emails have to match");
    }
  }
</script>
```

Javascript validation examples (3)

The problem with the previous code is that we still can submit the form. So, we need to add a second check that blocks the submission of the Http request:

```
<form>
  <label>Preferred email address:
  <input type="email" id="email_addr" name="email1"
placeHolder="user@provider.domain" required></label>

  <label>Repeat email address:
  <input type="email" id="email_repeat" name="email2"
required onchange="return check()"></label>

  <input type = 'submit' value ="Send"  onclick="return
check();">
</form>
```

```
<script type="text/javascript">
  function check() {
    var email1 =
         document.getElementById('email_addr');
    var email2 =
         document.getElementById('email_repeat');
    if ( email1.value != email2.value) {
       alert("The two emails have to match");
       return false;
    }
     return true;
  }
</script>
```

Regular Expressions

A **regular expression** is a set of special characters that define a pattern.

A regular expression consists of two types of characters: literals and metacharacters.

A **literal** is just a character you wish to match in the target (i.e., the text that you are searching within).

A **metacharacter** is a special symbol that acts as a command to the regular expression parser (there are 14, listed below).

. [ ] \ ( ) ^ $ | * ? { } +

Regular Expression Syntax (ii)

In JavaScript, regular expressions are case sensitive and contained within forward slashes. For instance

let pattern = /ran/;

will find matches in all three of the following strings:

'randy connolly'

'Sue ran to the store'

Note: The provided annex document gives more details about REGEX with javascript.

Examples: "crce".match(/[a-z]/g) , /[a-z]/i.test("vssdfs21321vsv")

Using the Dataset Property

- You can use the **dataset** property of DOM elements to store data, which provides read/write access to custom data attributes (data-*).

```
<div id="container" data-userid="2356" data-user-name="Salah">
  <p id="display">The user ID is:</p>
</div>

<script>
  const container = document.getElementById("container");
  const user_id = container.dataset.userid;
  const user_name = container.dataset.userName;

  container.appendChild(document.createTextNode(`User info: ${user_id}, ${user_name}`));
</script>
```

Source code: chapter04/14_datasets

Fundamentals of Web Development

Third Edition by Randy Connolly and Ricardo Hoar

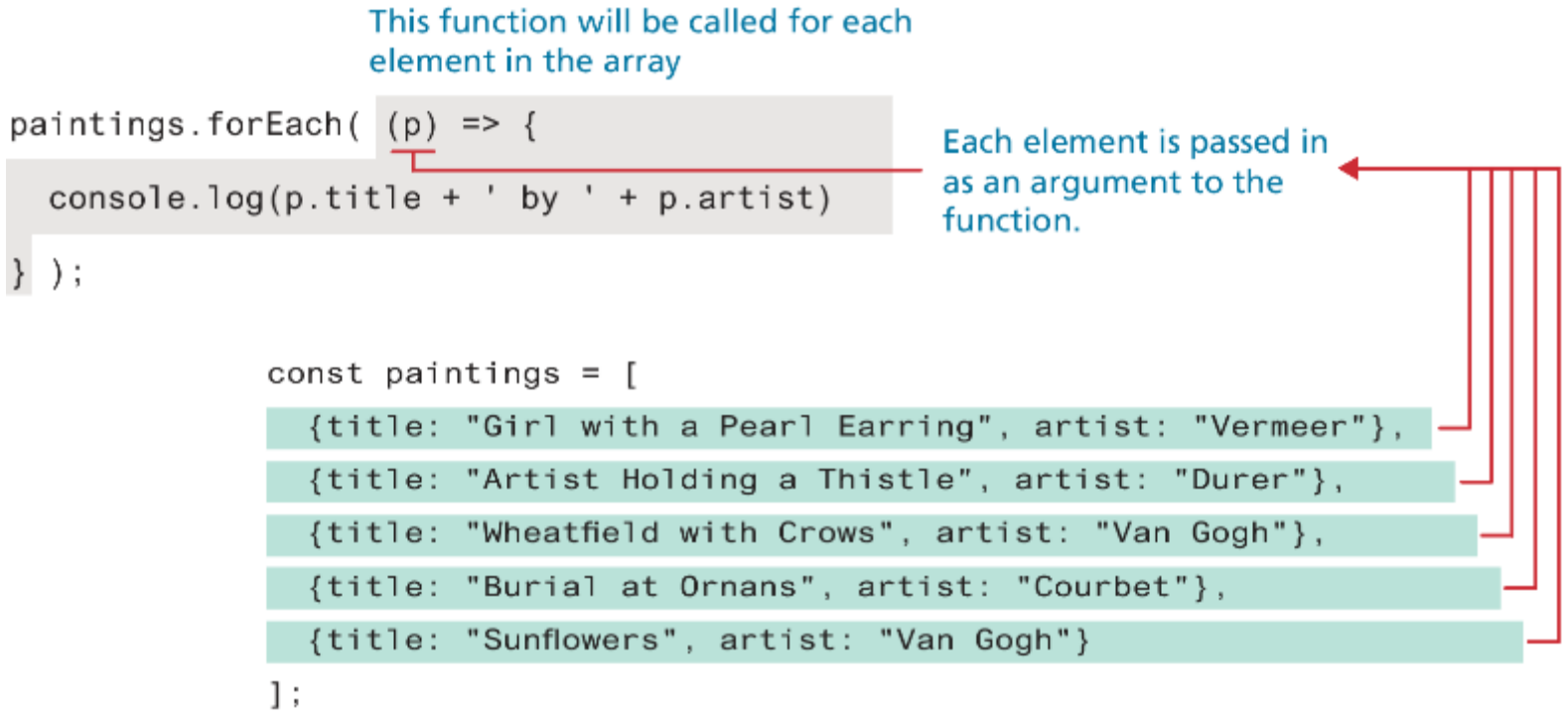
JavaScript Part 3:

Additional Features

Array Functions

- **forEach()** iterate through an array
- **find()** find the *first* object whose property matches some condition
- **filter()** find all matches whose property matches some condition
- **map()** is similar manner to filter except it creates a new array of the same size whose values have been transformed by the passed function
- **reduce()** reduces an array into a single value
- **sort()** sorts a one-dimensional array

Array forEach()



Array find()

One of the more common coding scenarios with an array of objects is to find the *first* object whose property matches some condition. This can be achieved via the **find**() method of the array object, as shown below.

```
const courbet = paintings.find( p => p.artist === 'Courbet' );
console.log(courbet.title); // Burial at Ornans
```

Like the **forEach**() method, the **find**() method is passed a function; this function must return either true (if condition matches) or false (if condition does not match). In the example code above, it returns the results of the conditional check on the artist name.

Array filter(), and map()

If you were interested in finding all use the **filter**() method.

The **map**() function creates a new array of the same size but whose values have been transformed by the passed function.

```
const arr = ["hello", "selem", "ciao", "hallo", "gutentag"];
const pat = /el/;
pat.test(arr[0])  // will return true
pat.test(arr[2])  // will return false


arr.map(o => pat.test(o))    // [true, true, false, false, false]
arr.filter(o => pat.test(o))   // ['hello', 'selem']
```

Reduce

The **reduce**() function is used to reduce an array into a single value. Like the other array functions in this section, the **reduce**() function is passed a function that is invoked for each element in the array.

For instance, the following example illustrates how this function can be used to sum the **value** property of each painting object in our sample paintings array:

```
let initial = 0;

const total = paintings.reduce( (prev, p) => prev + p.value, initial);
```

Example 2:

```
const all = arr.reduce((prev, p) => prev+ " " + p)      // 'hello selem ciao hallo gutentag'
```

Sort

sort() function sorts in ascending order (after converting to strings if necessary)

```
arr.sort()      //    ['ciao', 'gutentag', 'hallo', 'hello', 'selem’]
```

If you need to sort an array of objects based on one of the object properties, you will need to supply the sort() method with a compare function that returns either 0, 1, or −1, depending on whether two values are equal (0), the first value is greater than the second (1), or the first value is less than the second (−1).

Examples (reduce, sort array of objects)

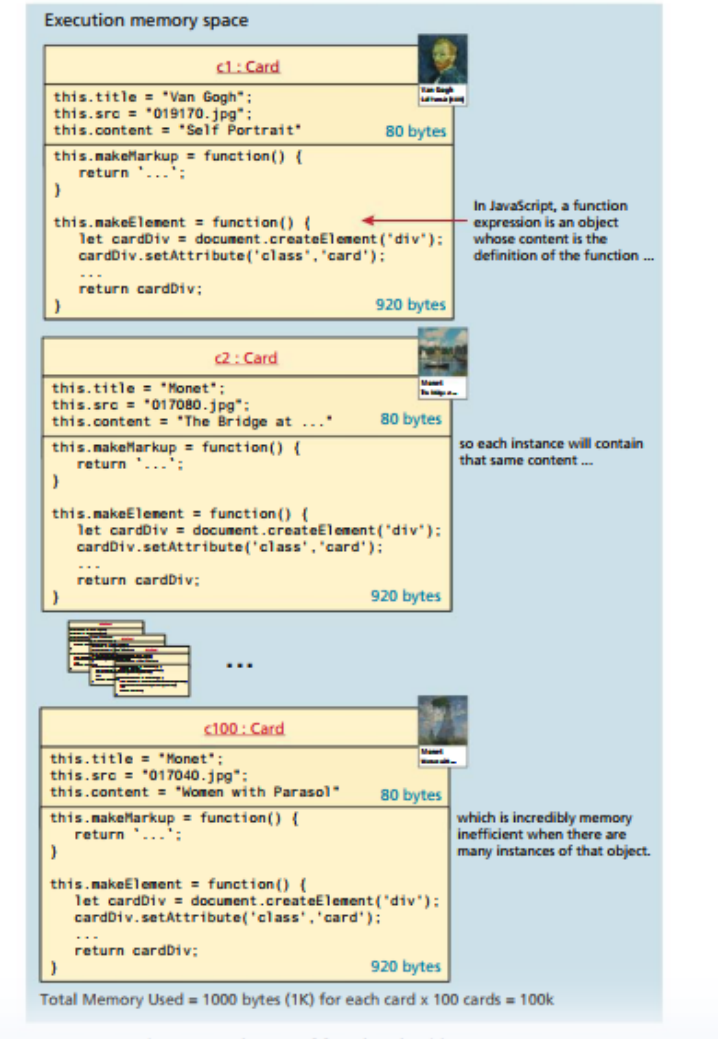
```
let initial = 0;

const paintings = [
  {title: "Girl with a pearl earring", artist: "Vermeer", value: 10},
  {title: "Artists Holding a Thristle", artist: "Durer", value: 7},
  {title: "Wheat field with Crows", artist: "Van Gogh", value: 16},
  {title: "Burial at Ornans", artist: "Courbet", value: 18},
  {title: "Wheat field with Crows", artist: "Van Gogh", value: 9}
];

//Compute sum of all these paintings values
const total = paintings.reduce( (prev, p) => prev + p.value, initial );
console.log( total );

// sort the array based on the value property of painting objects
const compareFn = (a, b) => a.value - b.value;
console.log( paintings.sort(compareFn));
```

Prototypes, Classes

The example on the right shows what happens when you use constructor functions for creating multiple instances of objects. Notice how the function properties are also duplicated.

Constructor functions are therefore inefficient since memory must be allocated for each (identical) method.

Prototypes help address this issue.



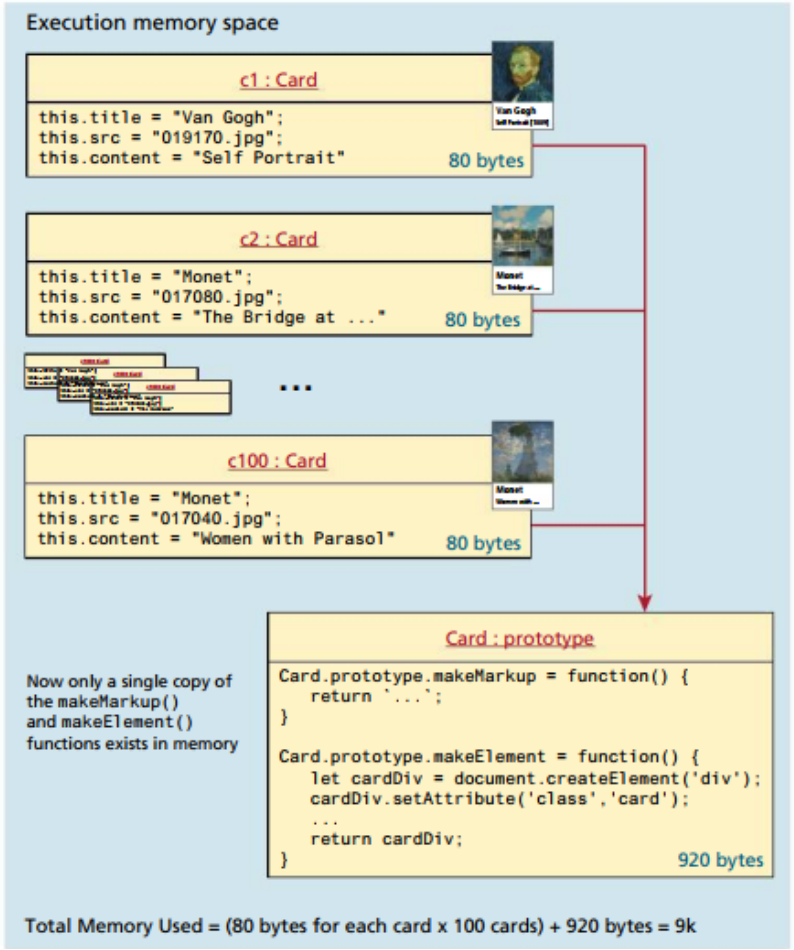
Prototypes

Prototypes make JavaScript behave more like an object-oriented language.

Every function object has a **prototype** property, which is initially an empty object.

The prototype properties are defined once for all instances of an object created with the new keyword from a constructor function.

This approach is far superior because it defines the method only once, no matter how many instances are created.



Using a prototype

```
function Card(title, src, content) {
    this.title = title;
    this.src = src;
    this.content = content;
}

Card.prototype.makeMarkup = function() {
    return `<div class="card">
            <img src="${this.src}" alt="${this.title}" >
            <div>
                <h4>${this.title}</h4>
                <p>${this.content}</p>
            </div>
            </div>`;
};
```

```
Card.prototype.makeElement = function() {
    let cardDiv = document.createElement('div');
     ….
    cardDiv.appendChild(div);
    return cardDiv;
};

// You use prototype functions as if they were declared
in the object
const container =
document.querySelector("#container");
const c1 = new Card("Van Gogh", "019170.jpg", "Self
Portrait");
container.appendChild( c1.makeElement() );
```

LISTING 10.5 Using a prototype

Using Prototypes to Extend Other Objects

Prototypes also enable you to extend existing objects (including built-in objects) by adding to their prototypes. Imagine a method added to the String object that allows you to count instances of a character.

```
String.prototype.countChars = function (c) {
        let count=0;
        for (let i=0;i<this.length;i++) {
                if (this.charAt(i) == c)
                        count++;
        }
        return count;
}
```

```
const msg = "HELLO WORLD";
console.log(msg + " has" +
msg.countChars("L") + " letter L's");
```

LISTING 10.6 Extending a built-in object using the prototype

Classes

- A **class** provides an alternate syntax for a function constructor and the extension of it via its prototype. In reality, they are merely “syntactical sugar” for JavaScript’s prototype approach
- While the class syntax provides a familiar alternate syntax for working with functions, the developer community has not universally adopted it
- Regardless of these concerns, the React framework, which has become one of the most widely adopted frameworks in the past several years uses JavaScript class syntax, so it is likely that as a JavaScript developer you will encounter this syntax more and more moving forward.

Using a class

```
class Card {
   // constructor replaces the function constructor
   constructor(title, src, content) {
   this.title = title;
   this.src = src;
   this.content = content;
   }
   // class methods replace prototypes
   makeMarkup() {
   return `<div class="card">
         <img src="${this.src}" alt="${this.title}" >
         <div>
            <h4>${this.title}</h4>
            <p>${this.content}</p>
         </div>
         </div>`;
   };
   // notice the function property shorthand syntax
   makeElement() {
      let cardDiv = document.createElement('div');
      …
      return cardDiv;
   }
}

// Use the class
const container =
document.querySelector("#container");
const c1 = new Card("Van Gogh", "images/019170.jpg",
"Self Portrait");
container.append( c1.makeElement() );
```

LISTING 10.7 Implementing Listing 10.5 (slide 13) using class syntax

Extending classes and more

There are additional syntactical features of classes in JavaScript, including getters/ setters and static functions that we are not covering.

Extending classes is one advanced feature worth noting. The **extends** keyword lets a class inherit the properties and methods of another class as with class-based programming languages such as Java or C#

```
class AnimatedCard extends Card {
    constructor(title, src, content, effect) {
    super(title, src, content)
    this.effect=effect;
}
```

Reading…..

- Read this book before graduation, when you finish it you'll know why….