Lezione 6

JavaScript

A study of the client-side scripting language from its grammar to its capability for generating dynamic content in Web pages. Event-driven-programming concepts are introduced, along with common design patterns for cross-browser compatibility.

Michael Serritella

Summer 2010

© Michael Serritella, 2010 • CIS4930: Website Architecture • The Florida State University

Intro to JavaScript

JavaScript is our first scripting language. It runs within the browser in order to monitor user-generated events and to modify page content programmatically, among other things. Linguistically, it is a strange language, since it is object-oriented but also prototype-based, and it has several curious features found in functional programming, such as anonymous functions and closures.

Basic powers

You will recognize many of JavaScript's powers. JavaScript is resposnible for:

- Pop-up windows; message boxes; status-bar changes
- Cursor- and click-based events (except for CSS's `:hover`, which is on loan from JavaScript)
- Keyboard-based events
- Manipulation of form elements and their values before the loading of a new page, including preliminary form validation
- Virtually any change in the page content which occurs without reloading the page

JavaScript has earned its reputation as a tacky and filthy language, especially in its glory days of the 1990s. It is filth and it begets filth. However, in recent years, JavaScript has been used more responsibly to provide useful functionality and actually improve the efficiency of websites. After all, there are some things that only JavaScript can do.

Basic delivery methods

JavaScript can be inserted into an (X)HTML document very much like CSS. It can appear in its own tag, placed anywhere, or it can be referenced from an external file. There is a third option, akin to the `style` attribute in HTML, but it is used for event-driven programming, and we will see it later.

To load an external JavaScript file, place this in the head:

```html
<script type="text/javascript" src="myJavaScriptFile.js"></script>
<!-- Note: You must close the empty tag. -->
```

To insert JavaScript into your markup directly, you can do this, anywhere in the head or the body, any number of times:

```html
<div>
  Text text text..
  <script type="text/javascript">AFunction();</script>

  <script type="text/javascript">
    someGlobalVariable++;
  </script>
</div>
```

Basic language features

JavaScript has a C++-style syntax, but it operates under different paradigms than C++ or most any other language. Here we learn its most basic differences from more familiar languages.

Quick & dirty debugging

Throughout this section, you will want to print out the values of your variables, just for some basic "echo checking". Here are two methods:

JavaScript

```
// Actually writes HTML in-place, wherever this JavaScript occurs
// within the document (or perhaps within some control flow, like
// in a function).
document.write("<div class='Hello'>Hello, World!</div>");

// Opens a familiar message window. This is somewhat variably sized,
// depending on the length of string you pass in.
window.alert("Hey!");
```

Variable scope

Variables have scope very similar to C++ or other scripting languages[1]. However, it may seem strange that all .js files and all inline code share the same global scope. You can create a global variable in one file and then use it in another file.

Simple data types

JavaScript variables are untyped, in that they may contain anything, including the special values `null` and `undefined`. However, there are some built-in data types (really just classes (or objects)) that come with their own methods, so you will want to set your variables to contain these object references.

[1]This is true, although scope rules may be complicated by some of JavaScript's more advanced functional-programming features.

Strings

JavaScript strings work fairly intuitively. You can set them easily and not worry about memory management, as you would have to in C/C++, and you can access some built-in methods.

JavaScript

```
var myString = "Hello";

// Access the built-in length.
var theLength = myString.length;

// Transform the string.
var upperString = myString.ToUpperCase();

// Edit the string with search & replace (FIRST occurrence ONLY).
myString = myString.replace("ello", "3110");

// Add to the string; use concatenation operators.
myString += ", Sir." + " How are you?";

// Technically, all of these are String objects.
var overlyFormalNotation = new String("Good evening.");

// A built-in function in global scope, but useful with strings.
var anInteger = parseInt("  1337 people");

// Quotes have a flexible and/or sloppy escape syntax in JavaScript.
// You may use single or double quotes to delimit a string, which
// have different escape-character consequences.
// These equate to the same value:
var aQuote = 'He said, "I don\'t understand."';
var aQuote = "He said, \"I don't understand.\"";
```

And there are a few more friends - mostly what you would expect. You can see any of a million online references, like **W3Schools**, for more.

Arrays

JavaScript arrays work a little less intuitively, but they work. You can (re)initialize them via a special notation, and they also have built-in methods.

JavaScript

```
// This is equivalent to a 'new Array(..., ..., ...)' call.
var myArray = [1, "aString", 3.33];

// Access the built-in length.
var theLength = myArray.length;

// Access the second element (arrays start counting at 0).
var secondElement = myArray[1];

// Add to the array; then subtract from the end (highest index).
myArray.push("New value");
myArray.pop();                  // "New value" gone

// Subtract from the front.
myArray.shift();
```

In addition, you may want to manipulate Arrays like so:

```
JavaScript
// Create a string in the form of a comma-separated list.
var aString = myArray.join(", ");

// Built-in sort function; sorts everything as strings (lame).
// See documentation on how to provide your own sorting function.
myArray.sort();
```

More information on the members can be found at **W3Schools**.

Regular expressions

Sometimes, you want to test if a string matches a pattern. **Regular expresssion** notation is a standardized way to specify a text pattern. Regular expressions can be notoriously complicated - simply arcane - but they are at least as powerful as you need them to be. Just in case you need them, JavaScript has built-in support for regular expressions in the form of the RegExp object.

```
JavaScript
// This object is not a string object, though it may be used in
// combination with string objects via some methods.
var myRegExp = new RegExp("P(\\-P)*owerBook");

var aString = "PowerBook";
var anotherString = "P-P-PowerBook";
var thirdString = "powerbook";

myRegExp.test(aString); // Returns true
myRegExp.test(anotherString); // Returns true
myRegExp.test(thirdString); // Returns false
```

You can also retrieve an Array of all matches within a string, and more; see **W3Schools**.

Functions

Functions have a cheesy and simple syntax, but they can be abused in complex ways. Here is an example function:

```
JavaScript
function SumFunction(x, y, z)
{
    return x + y + z;
}
```

There is no declaration of a return type, and the function may actually choose to not return anything; in this case, the caller

might unexpectedly get an `undefined` value. The function may also take any number of arguments, more or fewer than what is declared. If the caller gives fewer than the declared number, then the uninitialized ones are `undefined`. In order to circumvent this problem, each function has an **`arguments`** Array. Functions in JavaScript are actually quite complicated, but we will see more of that later.

The Object

The Object in JavaScript is nearly the universal type. All interesting data types - Strings, Arrays, regular expressions, and more - are descendants of the Object type. And any types which you create are just extensions of the Object type.

You can create an instance of an Object and then add members to the *instance* as you go. For example:

JavaScript

```
var myPen = {}; // alternative to myPen = new Object();

// Just set them to a value, and they will poof into existence.
myPen.reservoir = "50ml";
myPen.fineness = "medium";

// Valid access to member.
window.alert(myPen.reservoir);

// Accessing a nonexistent value before initialization results in the
// 'undefined' value.
window.alert(myPen.inkColor);
```

Literal creation of Objects

Similar to the Array initialization syntax, there is an Object initialization sytnax which lets you literally specify its starting properties.

JavaScript

```
var myPen = {reservoir: "50ml", fineness: "medium"};
```

Property access via expressions

Sometimes, you want to refer to a property name which is expressed in terms of JavaScript code. You can do this with square-bracket syntax, as follows:

```javascript
myPen["refills_" + colorName]--;
```

That example is probably more naturally expressed as a nested Object, like this:

```javascript
// Earlier..
// (Make sure that this is instantaited first.)
myPen.refills = {};

// Later that same evening..
// (Assume that colorName is defined here, perhaps inside of a loop.)
myPen.refills[colorName]--;
```

References to Objects

A somewhat interesting thing happens when one Object variable is set to another Object variable. They both refer to the same data. They are "references", in programming-language

parlance. They are like pointers, except that you don't have to worry about managing their memory. Quick trivia about references in JavaScript:

- All of the built-in string functions (members of String) return new copies of the string and do not modify the original; this is ostensibly to preserve the semantics of a string as a non-reference type.
- The Array member functions which transform the Array will operate directly on the data and thus affect all who refer to it.
- After no variables hold references to an Object's data, the data dies.
- To explicitly remove a reference to an object, set the variable to `null`.

Control flow and idioms

There are a few new control-flow constructs in JavaScript, only one of which is probably useful to you. And there are some new operators and idioms.

For .. in

Similar to a "foreach" loop in other languages, you may use a "for .. in" loop in JavaScript. This is used for iterating through all of the properties of an Object, although you could also use it (somewhat periliously) with an Array.

```javascript
var myPencil = {type: "mechanical", size: "5mm"};

for (var propertyName in myPencil)
{
    window.alert(propertyName + ": " + myPencil[propertyName]);
    // (Or myPencil.propertyName)
}
```

The new variable, **`propertyName`**, becomes the string value of the name of each property in the Object.

This would technically work for simple Arrays, since property-

Name would become "0", "1", "2", etc., but if any other properties were added to the Array, like **`myArray.highestElementValue`**, then the loop would iterate over those, as well. You may want to add your own properties to Arrays, and some may even be added by external code without your knowledge, as we will see in this lesson.

The "default operator"

In JavaScript, the "||" operator doesn't just resolve to a boolean value; it resolves to one of its actual operand values - the first one which is "truthy" (can be evaluated to **`true`**, such as a non-empty string or a nonzero integer). So, in taking advantage of this, JavaScript programmers will sometimes use it to make shorthand initializations.

JavaScript

```
function FormalizedName(Forename, Surname, Suffix)
{
    // ...
    var suffix = Suffix || "Esq."; // Incorrect but humorous
    // Note that you can chain many of these if it makes sense.
}
```

Since each function parameter may or may not be given by the

caller, this operator helps us assign a default value. Hence, it is sometimes called the default operator.

Test for strict equality

Since JavaScript variables are untyped, equality tests become slightly complicated. Type coercion is a type conversion that happens on the fly, inside an expression which expects a certain type. For instance, boolean operators and tests for truthiness will expect a boolean value, so a variable containing the number 5 or the string "Hey" may be converted to a boolean.

Sometimes, it's important to distinguish between the value 0 and the value `false`, `false` and `undefined`, etc. In order to do this, use the "===" operator. This tests for both value equality and type equality. Its boolean opposite is the "!==" operator.

Test for undefined

We've seen that the value `undefined` may come up frequently in JavaScript; it is certainly useful to be able to test if a variable is undefined.

At first, this is tricky business. You can't do this:

JavaScript

```
// Referencing an undefined variable, just by checking!
// (That is an error.)
if (someUndefinedVariable)
{
    // ...
}
```

So there's this thing, where you have to do this:

JavaScript

```
if (typeof(someUndefinedVariable) == "undefined")
{
    // ...
}
```

Just do it.

Paradigm stunts

JavaScript doesn't adhere to the same paradigms as most other languages - even C/C++, to which it bears a resemblance. There are no classes, templates, etc. It really has its own flavor of paradigms, which, quite strangely enough, are closer to functional languages like Lisp.

Anonymous functions

Like Objects and Arrays, functions actually have a literal notation. Since you would have to put this function somewhere, for starters, we'll stick it in a variable:

```javascript
var aFunc = function(x, y, z) { return x + y + z; };
```

And you could then pass this around to other variables and call it, like so:

```javascript
// ...
var aValue = aFunc(3, 4, 5);
```

You could also capture a reference to any normally-declared function, simply by using its name as if it were a variable.

There is the more exotic and dangerous "new Function" syntax, which you should know but probably shouldn't use:

JavaScript

```
// ...
var anotherFunction
        = new Function("paramName",
                       "anotherParamName",
                       "asManyAsYouWant",
                       "return paramName * anotherParamName;");
```

This invokes the JavaScript compiler directly, which is slow and potentially dangerous, if some untrusted user input gets mixed in with the code. It is also inconvenient, since you have to maintain correct sytnax entirely within a string. We will see alternatives to this.

Anonymous Objects and Arrays

Less impressively, you can also use the literal notation for Objects and Arrays without needing to assign them to a variable.

You may want to use this to pass the Object or Array to a function, or in some other intermediate part of an overly complicated expression.

Nested functions

Functions may be defined entirely within other functions, which is often useful, especially in parsing algorithms. Here is a toy example:

```javascript
// ...
function Outer(A, B, C)
{
    var localVariable;

    function Inner(X, Y)
    {
        return X.toString() + Y.toString();
    }

    return Inner(A, C); // or whatever
}
```

In the inner function, you can refer to everything which is in scope for the outer function, including its local variables. If the outer function returns a reference to the inner function, the in-

ner function retains the scope that it had, so the `localVariable` and whatever else may still be modified if you call `Inner()`. When a function has an out-of-body experience like this, it is called a closure.

Closures are an advanced programming tool. They have many uses, but they may be particularly useful when you write a library and interface with external code; you may use these techniques to provide a portable access to your local state or to hide it. Keep them in mind.

Anonymous functions

JavaScript functions can also *operate upon* other functions. In particular, they can create new functions and then return them, like a function factory. This is probably more often done without any ties to local state, so they aren't closures. Here:

JavaScript

```
function AdderFactory(HowMuchToAdd)
{
  return function(X) { return HowMuchToAdd + X };
}

var specificAdder = AdderFactory(3);

var seven = specificAdder(4);
```

Now, you can even customize the anonymous functions that you create, building them according to a pattern.

Functions as constructors

It turns out that in JavaScript, even *functions* are Objects. You can use the `this` variable within a function, and it will refer to a set of static variables that the function keeps. For example:

JavaScript

```
function StorageLocker(ThingToStore)
{
    // Initialize the static variable for the first time.
    if (typeof(this.Things) == "undefined")
    {
        this.Things = []; // equivalent to this.Things = new Array();
    }

    // Add the argument to the storage locker.
    this.Things.push(ThingToStore);
}
```

But really, it's not most useful to think of them as static variables. When you add members to the function with **`this`**, you can use the function as a constructor (!):

JavaScript

```
function Pen(InkColor, Fineness)
{
    this.reservoirML = 0;
    this.fineness = Fineness || "medium";
    this.inkColor = InkColor;
}

var myPen = new Pen("black");
```

Now, **`myPen`** has all of the properties you would expect - **`reservoirML`**, **`fineness`**, etc., and they've even been nicely ini-

tialized. You can even add methods, akin to object-oriented methods in C++ and other languages:

```
JavaScript
function Pen(InkColor, Fineness)
{
    // ...
    this.Write = function()
    {
        // ... Perhaps some fancy output
    }
}
```

And within the **Write()** function, you can use **this** to refer to the state of the specific Pen object.

Prototype-based programming

So far, so good. What if you want to extend the capabilities of a "class"? What if you want to add a property to all Pens in existence, like a **nibStyle** property? Enter prototype-based programming.

In our previous example, we would say that **Pen()** is the *prototype* of **myPen**. We can add properties to Objects on the fly, but we can add properties to all Objects of a certain prototype

if we modify the prototype on the fly. However, this requires one bit of special syntax:

```
JavaScript
Pen.prototype.nibStyle = "oblique";
// Most pens are not oblique. Impish!
```

Each function has a **`prototype`** property. If we modify that, we modify all Objects of the prototype. All Objects have a hidden[2] link to their prototype.

If you refer to a property of an Object and the JavaScript engine can't find it, it will look in the Object's prototype. This has some implications. It means that you can hide (shadow) a property from your prototype if you make an object-specific property of the same name.

Using prototypes, we may approximate the classical flavor of OOP. Using the prototype model and other linguistic peculiarities, we can add to classes dynamically and in strange ways. It's still a little dirty. But it is powerful for extremely late-bound and late-developing applications. Dirty power.

[2]Mozilla lets you read and modify this by the Object's `__proto__` property, but they probably shouldn't, anyway, so don't get too comfortable.

The DOM

The DOM ("The Dom") is not a French nobleman. It is the bridge between JavaScript and HTML. In this section, we learn how the HTML document is abstracted into JavaScript Objects and how we may modify the document through those Objects. In addition, we learn how to read information about the browser and window, as well as sense user events.

The DOM uses Objects and properties of Objects in order to form a tree-based data structure.

In this section, we will be learning the DOM as it is implemented by Firefox, which is reasonably standards-compliant and conventions-compliant. We will see strategies later for cross-browser compatibility. We will learn simple methods of reading & writing data, as well as introduce the practice of event-driven programming in JavaScript.

Basic structure

Basically, the document and browser properties are accessible via a few global variables in JavaScript. The DOM is a heirarchy with a root node, but the heirarchy is initially hidden from you in a sense, because the top-level elements are all in global scope.

The root of the DOM is a **window** object, and it has properties (like children) which have distinct jobs. Let's see a summary of its most useful ones:

window Gives information about the browser window, including its sizing and scrolling. Allows some interaction with the opening and closing events of windows (or tabs), as well as the capacity to open new browser windows (pop-ups) and display dialog-box messages.

document Provides access to all HTML elements.

history Allows interaction with the browsing history, including the ability to go forward or back a number of steps.

location Allows interaction with the current URL, including some properties which tell you slight information about the URL from having parsed it.

screen Gives information on the user's resolution.

Technically, all of these are children of **window**, but they are also in global scope, so you can simply refer to **document**, **history**, etc., without qualifying it as a child of **window**.

Supporting actors

In this section, we will see basic use of all Objects except for **document**. We'll skip over event-driven programming for this section and come back to it later.

window

With the **window** Object, we can read simple information about the browsing window (or tab, if applicable) and launch other windows or message boxes.

Properties:

scrollX *(read-only)* The number of pixels in the X direction (right) that the window is scrolled.

scrollY *(read-only)* The number of pixels in the Y direction (down) that the window is scrolled.

innerWidth *(read-only)* The width of the document's area in the browsing window (not related to content size).

innerHeight *(read-only)* The height of the document's area in the browsing window (not related to content size).

scrollMaxY *(read-only)* A maximum for **scrollY**.

scrollMaxX *(read-only)* A maximum for **scrollX**.

scrollMaxY *(read-only)* A maximum for **scrollY**.

screenX *(read-only)* The displacement of the browser window in the entire screen in the X direction (right).

screenY *(read-only)* The displacement of the browser window in the entire screen in the Y direction (down).

Methods:

alert() Display a quick message in a dialog box with an "OK" button.

prompt() Display a quick message in a dialog box asking the user to input text (which is returned).

confirm() Display a quick message in a dialog box asking the user to confirm or cancel ("OK"/"Cancel").

openDialog() Display a quick message in a dialog box with more custom options.

print() Display the print dialog.

scroll() *(and friends)* Scroll the window.

resizeBy() Resize the window by an (width, height) amount.

resizeTo() Resize the window to a (width, height) pair.

moveBy() Move the window to a different coordinate on the screen by nudging it by an X,Y vector.

moveTo() Move the window to a coordinate on the screen.

open() Open a window (pop-up), optionally with size and toolbar-existence information.

close() Close the window (tab).

atob() Base64 encode.

btoa() Base64 decode.

See the Mozilla Developer Center's **page** for more on **window**.

history

With the **history** Object, we can send the user forward or backward in time.

Methods:

back() Go back.

forward() Go forward.

go() Go a number of pages forward or backward (positive/negative).

See the Mozilla Developer Center's **page** for more on **history**.

location

With the **location** Object, we can change the URL Of the page. This is most useful for redirects.

You may set **window.location** directly to a string, but you can also access some of its properties and methods for more information. It has a decent number of properties that we don't know enough to care about yet.

Methods:

reload() Reload the page, optionally by forcing a reload from the server.

See the Mozilla Developer Center's **page** for more on **location**.

With **location**, you will often want to use the `setTimeOut()` function. It has strange semantics. See the MDC **page** for reference.

screen

With the **screen** Object, we can get information about the user's screen, outside the context of the browser.

Properties:

width The width of the screen - i.e. from the user's monitor resolution.

height The height of the screen - i.e. from the user's monitor resolution.

See the Mozilla Developer Center's **page** for more on **screen**.

The document and its elements

The **document** property is our star of the show. It contains a set of **element** Objects that represent HTML elements.

Each HTML element has a JavaScript Object (commonly called a node) associated with it. The name of the game is how to get these nodes. Once we get one, we can use easy-enough properties within each Object to navigate around the tree.

Before you begin, you may want to know of a few good reference pages so that you can look up more members and (perhaps prematurely for now) check compatibility across browsers. You might like the Mozilla Developer Center's page on **document** (**here**) and **element** (**here**), as well as the **compatibility charts** on quirksmode.org.

Querying the document

To get yourself started, you will want to get your hands on a node. There are technically a few ways to start, but the most direct way to do this is to give the element an ID, using the HTML `id` attribute. Then, use the **document**.`getElementById()`

function. You pass in a string, and you get back a node. This is the most direct way of Crossing Over.

NOTE: The "ID" in that function is not fully capitalized; it is "Id", like you are getting the element by its Freudian "id".

Alternatively, you can start at the document's body element and start looking for children, using Arrays of these nodes and other accessors. There is a shortcut to this node - you can use **document**.`body`.

Also alternatively, you can fetch an Array of all nodes which have a certain HTML tag. Use the function **document**.`getElementsByTagName()` and pass a lowercase tag name (to keep it simple), such as "p" or "div". We will see this later, since you can also call this method off of a node in order to fetch its children.

Manipulating element nodes

Once you have an element node, you can traverse to other element nodes, but let's not focus on that yet. Let's start with

the reason we're here: we want to edit the HTML document dynamically. There are a number of things we can edit, like CSS styles, HTML/text, and node/document structure. For now, as we limit our scope to one node, we will start with styles and text.

Styles

Element nodes typically have Object properties that are named similar to the HTML attributes of the element. Here is a good example: there is a **`style`** property of the node which corresponds to the `style` attribute in the HTML. It represents the same information but has slightly unique syntax.

You may do this:

```javascript
var aNode = document.getElementById("anID");
aNode.style.color = "red";

aNode.style.backgroundColor = "blue";
```

Note, however, that CSS properties which have a hyphen (e.g. `background-color`) are represented here as camel-cased names.

That's not all, though. This **`style`** property corresponds quite exactly to the `style` attribute in that it represents *inline* styles. So if you want to read the currently active style (e.g. color) of a node, and it has no inline style for color, you can't read it from here. You need a way to retrieve the aggregate/ultimate style of an element.

The W3C gives a standard for the DOM, and Mozilla mostly follows the recommendation. To retrieve the aggregate style, use the global-scope[3] function **`getComputedStyle()`**, like so:

JavaScript

```
// ...
window.alert(getComputedStyle(aNode, null).color);
```

The **`null`** basically has to be there unless you want to select pseudo-elements (advanced topic from CSS).

If you want to write a style, you can still use the **`style`** property, and this will work well, since inline styles trump all others.

[3]This is actually a member of `window`, but we can treat it as global; it's a matter of style.

Classes

In addition to the inline style, you can change the element's `class` attribute, which will change its style if there is relevant CSS in place. You cannot use the `class` Object property, though, due to historical problems with reserved words. You must use `className` to access the attribute. You can read and write `className` like any string. The String functions `split()` and `replace()` are very handy here.

Text

Using the DOM, you can read the text of an element fairly easily, but changing the text is more complicated. To read the text contained in a node, for Firefox, simply read the `textContent` property.

If you set `textContent`, it will "work", but you will clear out any HTML elements within the node; the node will now contain only the plain text you gave.

If you want to add text, you have to step into the world of node creation. Or you can dive in; here is an example:

JavaScript

```
// ...
var someText = " .. Some plain text";
var aTextNode = document.createTextNode(someText);

aNode.appendChild(aTextNode);
```

This simply appends text to the inside of the element whose node you have. However, it also gives the node a new child, and this child is of a peculiar type. This is a *text node*, which is not the same as a node produced from an HTML element. However, depending on which DOM methods you use for counting children and traversing nodes, they may or may not be counted in this node's list of children. When you manage to get a hold of a text node, you can change its text with **`textContent`**.

HTML

You can change HTML code directly, akin to **`textContent`**, and you can do it by creating and appending nodes. The node-creation method is sanctioned by the W3C, but all browsers support the direct changing of HTML text, in case you want to roll around in your own filth.

JavaScript

```
// ...
var aDiv = document.createElement("div");
aDiv.className = "MaybeSomeClass";
var textForTheDiv = "Some text in the div!";
var textNodeForTheDiv = document.createTextNode(textForTheDiv);

aDiv.appendChild(textNodeForTheDiv);

aNode.appendChild(aDiv);

// --OR--

// Note that this method clears out any existing HTML.
aNode.innerHTML = "<div class='MaybeSomeClass'>Some text in the div!</div>";
```

Due to software-engineering practices, though, you may want to go with the node method. We will talk more about JavaScript coding strategies later in the lesson.

Traversing the tree

From each node, and from the **document**, you can use the Object's properties to navigate around its immediate family tree. The following properties exist and have an intuitive meaning:

parentNode

firstChild *(including text nodes)*

lastChild *(including text nodes)*

childNodes *(Array) (including text nodes)*

children *(Array) (no-go for Firefox) (not including text nodes)*

In addition to this, nodes can refer to their siblings. These are the nodes "right next to" the reference node; they share the same parent.

previousSibling *(including text nodes)*

nextSibling *(including text nodes)*

Finally, note that you can chain the syntax, like this:

JavaScript

```javascript
// ...
var anotherNode = aNode.parentNode.parentNode.firstChild;
```

Events!

This is our final frontier in terms of JavaScript's core paradigms. Event-driven programming is just like it sounds: functions are not called in a scripted order, but they are called when an event

occurs. This can be something user-driven, like a mouse hover, a moue click, or a key press; or it can be browser-driven, like a page loading or window closing.

On page load

The mother of all events is the **load** event. It's not really that influential w/r.t. other events, but it is a great starting point. Let's just see an example.

HTML

```
<!-- ... -->
</head>
<body onload="callAFunction(1337, 'hello');">
  <div>
<!-- ... -->
```

It's that simple - at least for this event. Note that his event is fired after the document *and all its supporting, external content* are loaded. This includes images and CSS files and any other type of file.

User-driven

Most interesting events are user-driven, and they tend to have a slightly different syntax; you will see the resurgance of `this`.

On mouse hover

The event is **mouseover**, and it has uses HTML attribute of `onmouseover`. It goes like this:

```html
<img onmouseover="customHandlerForMouseOver(this, 1337);"
     src="asdf.gif"
     alt="Asdf">
```

When the event is fired and the function is called, the `this` keyword is passed. That refers to the *node of the* `<img>`. Convenient!

On mouse stop hovering

To catch the event of the mouse moving off of an image, the attribute is `onmouseout`. It goes like this:

HTML
```
<img onmouseover="customHandlerForMouseOver(this, 1337);"
     onmouseout="customHandlerForMouseOut(this);"
     src="asdf.gif"
     alt="Asdf">
```

On click

Some elements don't naturally track clicks - namely everything but `<a>`. To attribute a click event to these, the attribute is **`onclick`**.

On keypress

To track this event, you have to get a quick intro to form elements, such as text boxes. Surely, you have seen them before. You might be interested in this event to get immediate feedback on a change of a value in a text box. Here is the code for a text box, accompanied with the event, which has attribute **`onkeypress`**.

HTML
```
<input type="text" value="initial text" onkeypress="KeypressHandler();">
```

The handler for **keypress** is different in that it may need information about the event itself - i.e. which key was pressed. It gathers that information from an event Object. The event Object may be "global" (in **window** and thus globally accessible) and called simply **event**, or it may be passed to the handler as the first argument. You can look **online** for a depressing explanation of how this works.

In this case, you could at least retrieve the `value` property of the `<input>` Object to see the text inside.

On form-control change

With form controls, you often want to do something immediately after the user's input has changed, even before the form is submitted. This can be the changing of text in a text box or the changing of a radio button choice. For this, there is the `onchange` attribute. Note that this only fires after the focus (i.e. cursor) has moved *away* from the control that has just changed, so it would be inadequate for detecting keystrokes (until the user clicks outside of the text box).

On link click

Sometimes, links are adorned with JavaScript; in fact, often enough, JavaScript replaces the default behavior of a link. This is generally a bad idea, but you should know how it works, at least because it is ubiquitous.

In a link's **href**, you give the "javascript:" scheme and follow with some code, like so:

```html
<a href="javascript:window.alert('Hey!');">click</a>
```

This means that if JavaScript is turned off or doesn't exist for the user's browser, there is no real link action, which is quite annoying. To obtain a more harmonious balance, you can use **onclick** to fire a JavaScript event and then **href** to provide a link alternative. This requires event cancellation to work sensibly, however, which we will see soon.

An alternative method - slightly older - is to use "#" for the

link target and some JavaScript code for the `onclick`. A target of "#" will go to the top of the page. It may even make a "click" sound. As you might expect, if you give both `href` and `onclick` without any extra special code, then both will execute. Some browsers are aware of this idiom and will simply ignore the "#", while others will not; the less-aware ones may make a "click" sound or even add this to the browsing history. This inconsistent behavior can be a real pain.

There are some advantages to attributing JavaScript to links instead of just a `<span>` with an `onclick`. The user can tab through links with the keyboard, and browsers are trained to treat links specially in some ways, so you may want to latch on to that behavior.

Browser-driven

Aside from **load**, there are a few other browser-driven events. The inverse is **unload**, but really, there is one called **beforeunload** which gives you more power. It fires just before the window (tab) closes. Attach this to the **window** object.

Attributing events to nodes in JavaScript

This is all well and good, but what if you have 1,000 form controls on a page, each with two or three events? All of the "on*whatever*" HTML attributes would seriously weigh down your page. And if the event handler calls are so formulaic, shouldn't there be a way to automate this? There is a way to automate event registration with JavaScript, and there is more than one reason why you would want to do it.

Once you have a node Object in JavaScript, you can set one of its properties, like **`onclick`**, etc., equal to a function. Use a function name (unquoted) or the name of a variable which refers to a function (hint: same thing in this language).

However, that is only the most simple way to do it. If you have a self-contained site and aren't using any libraries, it may work. But what if you want to register multiple evetn handlers for the same exact event? You can, but with this syntax, whenever you set the event handler equal to a function, you overwrite whatever was there.

In Firefox, and according to the W3C, you can add an event from within JavaScript by using the **addEventListener()** method.

```
JavaScript
var aNode = document.getElementById("someElementID");
aNode.addEventListener('mouseover', myHandlerFunc, false);
// Basically always false for the last argument
```

What's also nice is that from within the **myHandlerFunc()** function, the **this** variable refers to the node with the event! Convenient. Don't use an Object method for your handler function, however, or the meaning of **this** is unnecessarily complicated.

See the Mozilla Developer Center page on **addEventListener()** for more.

Cancelling events

As we've seen, events can be attributed to elements that already have default behavior for those events - for example, **<a>** already has behavior for a mouse click. Another pertinent example, which we haven't yet seen, is that an HTML form

(`<form>`) already has a behavior for submission (**submit** is an event). Using JavaScript, we can add more event handlers to this event, but we also may use them to cancel the default behavior if our JavaScript code deems it appropriate.

You can cancel an event using a few different syntactic structures, but the most basic toy example is this:

```html
<!-- This link will never go to somePage.htm -->
<a href="somePage.htm" onclick="return false;">
```

Using this basic idea, we can bootstrap other syntactic forms. If you want a function to determine whether the event should be cancelled, you can do this:

```html
<!-- This link may go to somePage.htm, depending on the reckoner -->
<a href="somePage.htm" onclick="return EventHandlerAndReckoner();">
```

If the reckoner[4] returns false, then the event's default behavior is cancelled.

[4]They're not really called reckoners.

This can finally be composed into a nice structure that allows JavaScript if it's supported and alternative behavior if it isn't. Recall our link from before, which fired both the "#" and the `onclick`. If our **click** handler returns false, then the browser won't follow the link. If the browser doesn't support JavaScript, then the handler will be ignored, anyhow, and the default link behavior will work.

There is an even more advanced structure than this which offers more versatility. What if there is an error in the function? And/or what if you want the function to return true on success rather than false on success? Do this instead:

```html
<!-- This link may go to somePage.htm, depending on the reckoner -->
<a href="somePage.htm" onclick="return !InverseReckoner();">
```

This way, the function can work on "return true" semantics, *and*, if there's any problem within the function, the function will fail to return true (it returns void), so the negation will be true, and the event will not be cancelled. This defers to the fallback behavior - the `href` - and everything works as robustly as possible. Just think about it for a while. Be careful with

this, though, because you may have a silent failure and gloss over it in your debugging phase, since the link will still appear to "work" (do something).

Browser issues

We know by now that browsers are drama queens, and in case you didn't know, JavaScript is a Mexican soap opera. Since JavaScript execution is so complex - running an entire programming environment alongside the page rendering process and a hundred other things - browsers have engineered various ways to execute JavaScript. Feature compatibility is an issue, and there is even an issue as to how JavaScript is loaded and executed.

Compatibility

Browsers vary greatly on how well they support JavaScript's features. Or not JavaScript's language features so much, exactly, but the DOM. Recall the **acid3** test results to see a spectrum of unbelievable failure.

Check methods of **DOM traversal** and **event registration** (on QuirksMode) to see key differences on the features we have used so far.

More than anything, simply know that browsers may differ

in the way they implement the DOM, and be suspicious of browsers. We will see some more detailed strategies later.

Loading & execution behavior

When a browser encounters JavaScript in a page, what does it do? Well, generally, it stops everything and loads & executes the JavaScript. Because - think about it - the browser doesn't know if the JavaScript will fundamentally alter the page, so it can't really skip ahead.

What's more interesting is what browsers do when they encounter a link to an external JavaScript file. Some browsers will stop downloading all other JavaScript files while the first one is being downloaded, and some will stop downloading *all* files until the pending JavaScript is loaded and *executed*. This can be a staggering blow to your page's load time!

To mitigate this, you can employ a few strategies:

- Consolidate your JavaScript files into one (or at least fewer) files in order to reduce the number of synchronization barriers you have in the ideally parallel task of downloading your site's files.

- Include only minimal JavaScript at the top of your page and fetch the rest of it at the bottom. Or - what you likely want instead - on **load**. In an event handler, like for **load**, you can add a new `<script>` element to the end of `<body>`, where the `<script>` is loaded from an external file according to normal syntax.
- Employ caching policies to hope the JavaScript file is downloaded as infrequently as possible.

Browser detection

With all of this cross-browser trouble, perhaps you want a way to detect which browser is running and then vary your code accordingly. There are two common ways of doing this: with JavaScript and.. with HTML comments.

By JavaScript

Using JavaScript, you can naïvely check a certain part of the DOM, although, as this sentence implies, that is problematic. You can refer to the **navigator** Object, and its variously convoluted properties, for some semi-reliable values. View some documentation at the Mozila Developer Center **here**.

A strange fact is that you can never trust a browser to correctly report even its most basic information. Even still, if you have this information, what you usually want to know is whether a certain feature is supported - not exactly which browser the user has. You are one step away from the information you really want. What's worse is that each incarnation of a browser - on Windows, on Mac, on Linux, on each brand of mobile phone, etc. - may have a different layout engine, different support of JavaScript/the DOM, and different bugs.

What you should do instead is *test for the features you want*. Test for the symptoms rather than trying to identify the browser and then inferring the symptoms. Consider this example of event registration provided by the Mozilla Developer Center:

JavaScript

```
// For Firefox/DOM-standards-compliant browsers
if (aNode.addEventListener)
{
    aNode.addEventListener('click', modifyText, false);
}
// For Internet Explorer
else if (aNode.attachEvent)
{
    aNode.attachEvent('onclick', modifyText);
}
```

By comments

Out of some measure of guilt, Internet Explorer provides a way for you to detect it using HTML comments. Observe:

HTML

```
<!--[if IE 6]>
<script type="text/javascript" src="someHackIEScript.js"></script>
<![endif]-->
```

This looks like a comment to any other browser - including the encompassed HTML code - and so it is ignored by other browsers. You can actually build compound logic expressions and use less-than and greater-than operators for different versions of Internet Explorer. See the **Wikipedia entry** for more.

You may like this method, but you probably shouldn't; it can change the way that the programmer is intended to read HTML code and thus lead to problems in semantic clarity.

JavaScript frameworks

In recent years, there have been advances in JavaScript frameworks - code libraries that assist in the common functions of JavaScript and which even alleviate some cross-browser woes. The most prominent of these is jQuery, which is more or less archetypal of the others.

jQuery

jQuery is a framework which primarily assists in retrieving (querying) nodes from the DOM; it also has cross-browser features and provides a platform for hundred of plugins, which are commonly used for building standard HTML parts such as animations and menus. It is free and open-source, and large companies are starting to bundle it into their applications.

Powers & abilities

jQuery has a few strong selling points:

DOM selection

jQuery uses CSS selectors - even CSS 3 selectors - to retrieve nodes from the DOM. It actually emulates the logic behind CSS selectors, even though they are already executed elsewhere within the browser. Though somewhat absurd, the selectors can help, as they are a familiar interface rather than the yet-another interface of the DOM. For example:

JavaScript
```
var someNodes = $("div.SomeClass > p");
```

What is even better is that these selectors work on most all browsers, even old versions that do not support the CSS natively.

Advanced aesthetics

While many of jQuery's plugins are simply shortcuts, some of its aesthetic plugins are something more - they're huge shortcuts. As of summer 2010, not all layout engines support animation effects, but you can always fake it with enough painstaking loops. If you really need simple animation for your

page elements, then look into jQuery and its **plugins** rather than using Flash or coding it yourself.

Incorporation into your page

One extra nice thing about jQuery is that it is hosted on the Web by several large companies around the world, so you can actually point your `src` at a large company instead of at your own servers. As we have seen, this alleviates the concurrent-connection problem on your own servers, and the .js file can likely benefit from server locality and caching. Search for "**jQuery CDN**" for the latest.

Other frameworks

There are other frameworks - even some that cost thousands of dollars. Other frameworks may differ in the feature set that they support, such as by emulating the ability to create OOP classes in order to help the application programmer. See **Prototype**, **MooTools**, **Ext**, and **more**.

Survival advice

Here is some battle-hardened JavaScript and client-side-coding advice that should help you to program as smoothly as possible.

Cross-browser awareness

There are a ton of browsers out there - far more than the big two, or big three, or big six. Browsers are popping up all the time, and existing ones are devolving all the time. You can never be an expert on the behaviors of each browser for each feature, and you shouldn't try; it's a waste of your time. At best, you would become a master of trivia. The best you can do is to be *aware* of the possible problems that may occur in different browsers. Can an array of node children include text nodes? What about empty text nodes? Well, it *can conceivably* do either of those things, so you can bet that some client's browser will do it.

So, you should account for it either by testing explicitly or using a more robust technique. Whenever you feel suspicious about a feature, search for "cross-browser *(whatever)*" or "*(whatever)*

in IE", and check on any issues. Check as you need it, on a case-by-case basis. You will accumulate a sense of browser behavior and a sense of browser cynicism over time, which are valuable-enough assets that you don't need to go further and memorize browser features.

Use features that are as standards-compliant as possible (in addition to any robustness checks and workarounds that you need to do for specific features). Even if you know that some browsers are not standards compliant, the closest thing you can assume is that the client's browser will be standards-compliant for core features of the standard; otherwise, you can't proceed with any helpful assumptions to the contrary ("code it according to not the standard?").

In the grand scheme of website architecture, client-side programming is not that interesting. For some non-computer scientists, they think it is the tallest mountain they can climb, so they make a living out of being a client-side expert. Surely, others go into the field out of genuine curiosity. But it is probably a waste of your time to develop a thorough expertise of these quirks; the most interesting features lie beyond.

Development & debugging

JavaScript is particularly difficult to debug and can be overall pesky to write, though it is getting much better in recent years.

JavaScript code correctness

There is no compiler to suggest reasons for your mistakes, and sometimes, code works even though it shouldn't. There are browser plugins and other tools to help you debug, but *guess what?* JavaScript is browser-dependent. There is no universal debugger, and there are not debuggers for every browser, so you can't exactly use Firebug to debug your scripts' behavior in Internet Explorer.

To ensure correct code, start with a basic program architecture and check & refine incrementally. You should proceed conservatively, thinking of cross-browser correctness more than any other goal. Get a finished product (or a nearly-finished product) before you think about optimizations.

There are a few generic compilers out there that should help you, should you want a more aggressive approach. See the

Google Closure Compiler, which also optimizes (and maybe mangles) your JavaScript. See **JSLint**, by JavaScript guru Douglas Crockford, which is a sort of compiler and error checker.

HTML & CSS code correctness

With JavaScript, you have the option to write HTML code and write CSS code. For instance, you can use `innerHTML` or **document**.`write()`, and you can use the `style` property of elements, among other means. You should probably never, ever do this. You wouldn't write your entire web page inside a giant **printf()** call of a C program, would you? There is no checking of syntax, and its integration with your more static code can only be seen, well, dynamically. You may induce a parse error and never catch it.

Instead, you should use W3C methods for creating element nodes. Create functions whose responsibility it is to create a certain type of node (+ subtree), where the caller of the function has the job of integrating it into the DOM.

Prepare CSS classes in advance for all of the styles you will ever have. If you want, include the extra CSS styles in a file at the bottom of your page, so it does not block anything else from loading. Then, use JavaScript to alter the class (`className`) of elements and point them to these various styles. If and when the elements move back into their original state (e.g. **mouseout**), point them back. You may do something like this:

```javascript
// Change the state of the node; change its style.
aNode.className += " AlteredBeast";

// Revert the style of the node.
aNode.className = aNode.className.replace("AlteredBeast", "");
// Note that the trailing space delimiter between class names
// may be automatically removed from the string after this step,
// so don't search for " AlteredBeast"; this is more cautious.

// Or you can replace it with "NotAlteredBeast", if that makes
// sense in your application - if the new style is not so much
// an additive as a replacement.
```

Algorithmic complexity and performance

Your JavaScript does not generally have to be the fastest program in the world. It won't, anyway; it's running inside of a behemoth browser. As stated in the section on correctness,

you should code for correctness first and then worry about performance. Once you do look at performance, though, there are two general issues to consider: time and memory.

To reasonably optimize time, just try to make as few passes over the data as possible (i.e. smallest order of magnitude in asymptotic complexity). If there are n data items, try not to make n^2 comparisons or other operations, like if you compare every element in a list to every other element. You can get away with this if the list is small (just run experiments), but you should generally avoid algorithms that do this, even if it means avoiding features which require it. It is almost always possible to do this, though, in algorithms that are used in JavaScript. Simply put: go for the most coarse-grained time improvements.

To reasonably optimize memory, consider the amount of global data you have lying around. When it's not in use, set it to **null**. Perhaps have an Object which contains all the global data you use for temporary computations, so you can easily set the one Object to **null** and then destroy everything inside it. Note that since closure functions hold on to references to their variables,

they may tie up memory and prevent it from being destroyed. So, if this is not necessary behavior, don't create a million closures. Don't even make too many anonymous functions. If you sense a problem, investigate how many of them you create.

Markup integration

We have seen various ways of incorporating JavaScript code and events into HTML.

Code file consolidation

Since your code should be correct more than anything, you should probably avoid consolidating your JavaScript files too aggressively, since the benefits of modularity and simpler collaboration may outweigh the drawbacks of request overhead, especially with caching.

Code inlining and DOM retrieval

You should generally use IDs for your HTML elements to facilitate your DOM lookups. If you have a grouping of elements that you want to affect, perhaps have an ID on the

container. If several elements on your page fit into a pattern, then name them according to a pattern, such as "Submenu_1", "Submenu_2", etc. This becomes trivial when the page is programatically generated. Then, in your JavaScript, you can do something like this:

JavaScript

```
// Set this value somewhere;
// maybe simply write it in a <script> at the bottom of your  page.
var TotalSubmenus = 3;

// Iterate through all submenus and fetch their nodes.
for (var i = 1; i <= TotalSubmenus; i++)
{
    var aSubmenu = document.getElementById("Submenu_" + i);
    // Do something with the node; register events; etc.
}
```

If you only have one instance of an element grouping on a page - such as only one fancy search bar - then you can give an ID to everything in there that would help you. You can be specific, concrete, and technically inefficient. If you have an infinite number of element groupings, like product listings, then give yourself one or two IDs per grouping and then use node traversal to find the others.

Trivia: IDs and classes

Note that using IDs for CSS is slightly less flexible than using classes and does not offer any significant benefit[5]. You should maintain classes for your CSS use and IDs for your JavaScript use, even if they appear redundant (e.g. `id` is "Submenu" and `class` is "Submenu"). Quite often, a website grows by multiplicity of its existing elements, like if the client decides to add a second category of products to the page, add a second testimonial, or even a second store. You should be prepared for multiplicity as much as possible, and using classes for CSS is one way of doing this.

[5]Faster matching of selection rules, but that is slim.