JavaScript Intermediate Tutorial

Okey dokey. You've nailed the basics then, hmm? Good stuff. Have a cake. And once you've eaten it, try your hand at some areas of JavaScript that will push you a little further.

Contents

https://htmldog.com/guides/javascript/intermediate/

The DOM

The Document Object Model is a way to manipulate the structure and style of an HTML page. It represents the internals of the page as the browser sees it, and allows the developer to alter it with JavaScript.

If you'd like to have a look at the DOM for a page, open up the developer tools in your browser and look for the "elements" pane. It's a great insight into how the browser thinks, and in most browsers you can remove and modify elements directly. Give it a try!

Trees and Branches

HTML is an XML-like structure in that the elements form a structure of parents' nodes with children, like the branches of a tree. There is one root element (`html`) with branches like `head` and `body`, each with their own branches. For this reason, the DOM is also called the *DOM tree*.

Modifying the DOM, by picking an element and changing something about it, is something done often in JavaScript. To access the DOM from JavaScript, the `document` object is used. It's provided by the browser and allows code on the page to interact with the content.


New Examples Section! See all of this code stuff in action, and play around with it.

Getting an Element

The first thing to know is how to get an element. There are a number of ways of doing it, and browsers support different ones. Starting with the best supported we'll move through to the latest, and most useful, versions.

By ID

`document.getElementById` is a method for getting hold of an element - unsurprisingly - by its ID.

```
var pageHeader = document.getElementById('page-header');
```

The *pageHeader* element can then be manipulated - its size and color can be changed, and other code can be declared to handle the element being clicked on or hovered over. It's supported in pretty much all the browsers you need to worry about.

Notice that `getElementById` is a method of the *document* object. Many of the methods used to access the page are found on the document object.

By Tag Name

`document.getElementsByTagName` works in much the same way as `getElementById`, except that it takes a tag name (`a`, `ul`, `li`, etc) instead of an ID and returns a *NodeList*, which is essentially an array of the DOM Elements.

By Class Name

`document.getElementsByClassName` returns the same kind of NodeList as `getElementsByTagName`, except that you pass a class name to be matched, not a tag name.

By CSS Selector

A couple of new methods are available in modern browsers that make selecting elements easier by allowing the use of CSS selectors. They are `document.querySelector` and `document.querySelectorAll`.

```
var pageHeader = document.querySelector('#header');
var buttons = document.querySelectorAll(.btn);
```

`querySelector`, like `getElementById`, returns only one element whereas `querySelectorAll` returns a NodeList. If multiple elements match the selector you pass to `querySelector`, only the first will be returned.

Events and Callbacks

In the browser most code is *event-driven* and writing interactive applications in JavaScript is often about waiting for and reacting to events, to alter the behavior of the browser in some way. Events occur when the page loads, when user interacts (clicks, hovers, changes) and myriad other times, and can be triggered manually too.

To react to an event you *listen* for it and supply a function which will be called by the browser when the event occurs. This function is known as a *callback*.

Here's a group of the things needed to listen for an event; the callback function, an element and the call to listen for an event:

```
var handleClick = function (event) {
    // do something!
};
var button = document.querySelector('#big-button');
button.addEventListener('click', handleClick);
```

`addEventListener` is a method found on all DOM elements. Here it's being called on an element saved in the *button* variable. The first argument is a string - the name of the *event* to listen for. Here's it's `click` - that's a click of the mouse or a tap of the finger. The second is the *callback* function - here it's `handleClick`.

Unfortunately, Internet Explorer does not support `addEventListener` in versions earlier than 9. Instead `attachEvent` must be used.

```
button.attachEvent('onclick', handleClick);
```

Notice that it requires `onclick`, not just `click`. You'll find this is the same for most events. It's things like this that caused the creation of libraries like *jQuery* (which we'll come to) - they help you so that you don't have to care that the way of listening for events is different across browsers.

Data about a particular event is passed to the event *callback*. Take a look at `handleClick`, declared above. You can see its argument: *event* - it's an object whose properties describe what occurred.

Here's an example event you might see into a click event callback like `handleClick`. There are lots of properties giving you an idea of where the event occurred on the page (like `pageX` and

`offsetY`) - these are slightly different because they depend on the reference point used to measure from. You can also see the `target` which is a reference to the node that was clicked.

```
{
    offsetX: 74,
    offsetY: 10,
    pageX: 154,
    pageY: 576,
    screenX: 154,
    screenY: 489,
    target: h2,
    timeStamp: 1363131952985,
    type: "click",
    x: 154,
    y: 395
}
```

AJAX

In the early years of the web things were simple — a page was text, perhaps with styling, and it contained links to other pages. To get new content you moved from one page to the next. But as developers got more ambitious with the web, attempting to build interactive ("native-like" applications), it was obvious that there needed to be a way to load new content into a page without a full reload.

To retrieve new content for a page, like new articles on an infinite-scroll site or to notify you of new emails, a tool called an *XML HTTP Request* (*XHR*) is used. Web apps that do this are also called *AJAX apps*, AJAX standing for *Asynchronous JavaScript and XML*.

Almost all sites that pull in new content without a page reload (like Facebook, Gmail, Google Maps etc) use this same technique. In fact, it was Microsoft developing Outlook Web Access who originally created the XMLHttpRequest.

XML HTTP Request

So what does an `XMLHttpRequest` look like?

```
var req = new XMLHttpRequest();
req.onload = function (event) { . . . };
req.open('get', 'some-file.txt', true);
req.send();
```

The first thing to do is create a new `XMLHttpRequest` request, using the `new` keyword, and calling `XMLHttpRequest` like a function.

Then we specify a callback function, to be called when the data is loaded. It is passed information about the event as its first argument.

Then we specify how to get the data we want, using `req.open`. The first argument is the HTTP method (GET, POST, PUT etc). Next is the URL to fetch from - this is similar to the `href` attribute of a link.

The third is a boolean specifying whether the request is *asynchronous* - here we have it `true`, so the `XMLHttpRequest` is fired off and then code execution continues until a response from the server causes the `onload` callback to be fired.

The *asynchronous* parameter defaults to false - if it's false, execution of the code will pause at this line until the data is retrieved and the request is called *synchronous*. Synchronous

XMLHttpRequests are not used often as a request to a server can, potentially, take an eternity. Which is a long time for the browser to be doing nothing.

On the last line we tell the browser to fire off the request for data.

Using an `XMLHttpRequest` you can load HTML, JSON, XML and plain text over HTTP and HTTPS, and it also supports other protocols like FTP and file. All in all, they're very useful for a whole range of tasks involved in developing JavaScript apps.

AJAX and Libraries

The AJAX technique has been developed to the point now where single-page apps are possible - one main request loads JavaScript code which then loads, asynchronously, other content from the server. Whole libraries and frameworks have been built to help do so, and we'll take a look at them later.

JSON

JSON — **JavaScript Object Notation** — is a set of text formatting rules for storing and transferring data in a machine and human readable way. It looks a lot like the object literal syntax of JavaScript, and it is from there JSON originates.

But JSON is not JavaScript. Officially it's a totally different language with its own spec but it plays such a big part in JavaScript development that it's important to cover.

Here's some JSON:

```
{ "name": "Yoda", age: 894, "lightsaber" : { "color": "green" } }
```

Like in JavaScript, the brace notation is used.

Interestingly, the above example is actually valid JavaScript.

JSON is used to transfer information - between your browser to a server, or saved in text files for retrieval later - because it's simply text. That means you can't store complex data like a function, but you can store arrays, objects containing simple data, strings and numbers.

JSON is taking over from XML as the data-transfer format of the web, and many new web APIs are written exclusively serving JSON, which can mean that you can be using AJAX technology to grab JSON. But AJAJ ain't so catchy.

Using JSON

Data is either converted to or from JSON, using methods called `stringify` and `parse` respectively. JSON is an object available in pretty much all modern browsers but there are ways of adding to a browser that doesn't have it.

```
var jsonString = JSON.stringify({
    make: "McLaren",
    model: "MP4-12C",
    miles: 5023
});
```

`JSON.stringify` converts an object into a JSON string. In this example, *jsonString* becomes `{"make": "McLaren", "model": "MP4-12C", "miles": 5023 }`.

```
var car = JSON.parse(jsonString);
```

The string can then be converted back to a JavaScript object using `JSON.parse`. *car* is now usable as a normal JavaScript object, so you can set its properties:

```
car.model = "P1";
```

Scope

Scope is the term used to mean variable visibility — a variable's **scope** is the part of your code that can access and modify that variable. JavaScript has *function scope* — but what does that mean and how is it different to other languages?

Scope is, in many programming languages, dictated by the *block* in which the variable was declared. A block, in C-like languages, is anything between two curly braces, or indentation in a language like Python. For example, the *b* variable below is not available outside the curly braces in which it was declared:

```
var a = 10;

if (a > 5) {
    var b = 5;
}

var c = a + b; // Wouldn't work!
```

Global variables — that is, variables that can be read and modified anywhere in your application — are not good because they can expose security issues and make code much harder to maintain.

Remember that code is read much more than it's written. When reading code, if you can't determine where a variable came from and what its potential values are, there's a problem.

So it's best to limit the scope of a variable as much as possible, making it visible to as few parts of your code as possible.

 **New Examples Section!** See all of this code stuff in action, and play around with it.

Function scope

JavaScript does things a little differently. It uses *function scope*. This means that variables are not visible outside of the *function* in which they were declared. If they are not declared inside a function then they are available globally.

The example below demonstrates that a variable is only visible inside the function in which it was declared. The *doSomething* function is the variable *a*'s scope.

```
var doSomething = function () {
    var a = 10;
};
```

```
doSomething();

console.log(a); // a is undefined
```

Comparing this to the block scope example above, you can see that, in JavaScript, *b* is available:

```
var a = 10;

if (a > 5) {
    var b = 5;
}

var c = a + b; // c is 15
```

Child scopes

Variables are available in *child scopes* of their own scope. For example, *doSomethingElse* is a child of *doSomething*, so *a* is visible inside *doSomethingElse*.

```
var doSomething = function () {
    var a = 10;
var doSomethingElse = function () {
    console.log(a); // a is 10
};
doSomethingElse();
};

doSomething();
```

Functional scope is a very powerful tool for creating elegant code, as we'll see, but it can take some time to get your head around.

jQuery

The set of tools used to allow modification and control of the DOM are a bit of mess because they differ across browsers, generally for historical reasons. To make your job easier, a number of *libraries* have been created that hide these differences, providing a more uniform way of interacting with the DOM. Often they also provide AJAX functionality, taking the burden of that complexity away from you.

jQuery is far and away the most popular DOM library and it is used on a huge number of websites.

By the way, the way you interact with the DOM (or any service that you use via code) is called an Application Programming Interface or API.

jQuery has a very distinctive syntax, all based around the *dollar* symbol. Here's some:

```
$('.btn').click(function () {
    // do something
});
```

This code attaches a click handler to all elements with a class of *btn*. This *selector syntax* is core to jQuery.

jQuery is added to a page by simply including it as a file using a `script` element. Download it from jquery.com, or include it from a Content Delivery Network (CDN) such as CDNJS:

```
<script
src="http://cdnjs.cloudflare.com/ajax/libs/jquery/1.9.1/jquery.min.js"></script>
```

In the following pages we'll be taking a look at jQuery for selecting and manipulating elements, AJAX and at some of its other useful tricks.

It's not always the best idea to use a library, and it's best to choose whether to use a framework based on the specific project. For example, jQuery is a large file to introduce to a page and slow the downloading of that page, particularly on mobile browsers with potentially weak connections.

jQuery: DOM API

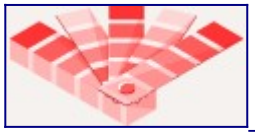
Selecting elements and interacting with the DOM are things you'll probably use jQuery for the most. Luckily it's really easy to do because the syntax for choosing an element, or set of elements, is exactly the same as element selection in CSS.

jQuery also makes performing actions on many elements at the same time simple, which is incredibly useful.

In the example below `$('.note')` selects all the elements with a class of *note* on the page and then we set the background of all of the *note* boxes to red and their heights to 100px.

```
$('.note').css('background', 'red').height(100);
```

jQuery uses a really neat *chainable* syntax that allows code like the above. This works because, for any kind of "setting" method, jQuery returns the same thing as the selector function ("$") does: `$` is a function that returns a jQuery *wrapper* around an element. `.css` is a method of that jQuery wrapper and it too returns the same wrapper. `.height` sets the height (duh) of the element selected, and of course there's an equivalent for width.

 **New Examples Section!** See all of this code stuff in action, and play around with it.

Getters and setters

In the above example we used `.css` and `.height` to set the value of the element but these methods are also *getters*. Calling `.height` without any value returns the current element's height and calling `.css` with just a CSS property name retrieves the property's value.

```
var currentHeight = $('.note').height(),
    currentColor = $('.note').css('color');
```

If you've got more than one element selected (you have lots of *note* elements, say) a getter will retrieve the value for the first of them.

Context

It's sometimes useful to limit the area of the DOM from which an element can be selected. The area of the DOM used is also known as a different *context*.

To tell jQuery which area you want to select from you pass a second argument that can be a *DOM element*, a *string selector* (selecting an element that jQuery will find and use) or a *jQuery object*. jQuery will only search within this context for your selector.

Here's an example. Notice that the variables that are used to store jQuery objects begin with a *dollar*. This a convention to help you and readers of your code understand that it's a jQuery object being saved.

```
var $header = $('header'),
    $headerBoxes = $('.note', $header);
```

jQuery: AJAX

jQuery has some AJAX helper methods that save time and are much easier to read. They are all properties of the `$` variable: `$.get`, `$.post`, and `$.ajax`.

`$.ajax` is the main method, allowing you to manually construct your AJAX request - the others are shortcuts to common configurations, like getting data or posting it.

Here's an example that gets some data from a server:

```
$.ajax({
    url: '/data.json',
    method: 'GET',
    success: function (data) {
        console.log(data);
    }
});
```

You can see that a configuration object is used to tell jQuery how to get the data. The basics are supplied: a URL, the method (which actually defaults to `get`) and a function to be called when the data is retrieved, named the *success callback*.

$.get

This example is just getting some data and, because this is a very common activity, jQuery provides a helper: `$.get`.

```
$.get('/data.json', function (data) {
    console.log(data);
});
```

You can also provide an error callback, which will let you know if something goes wrong and the server can't be reached:

```
$.get('/data.json', function (data) {
    console.log(data);
}).fail(function () {
    // Uh oh, something went wrong
});
```

$.post

Sending data to a server is just as easy, using the `$.post` method. The second argument is the data to be sent - it can be almost anything except a function: jQuery will figure out how to send it for you. How convenient!

```
$.post('/save', { username: 'tom' }, function (data) {
    console.log(data);
}).fail(function () {
    // Uh oh, something went wrong
});
```

$.ajax

Of course, if you want more control over how data is sent, use `$.ajax` to set the request up manually.

```
$.ajax({
    url: '/save',
    method: 'POST',
    data: { username: 'tom' },
    success: function (data) {
        console.log(data);
    }),
    error: function () {
        // Uh oh, something went wrong
    }
});
```

jQuery: Other Tricks

jQuery also helps you out with other common tasks, particularly where things are inconsistent across browsers.

DOMContentLoaded

Sometimes you will want to run JavaScript only when the DOM is loaded and ready (but before stylesheets are fully loaded) - for example, to move elements to a different location in the page, or create new ones. We can do this in pure JavaScript (although this will not work in all browsers):

```
var doSomething = function (event) { . . . };

window.addEventListener('DOMContentLoaded', doSomething);
```

But we can do it more easily with jQuery, and it will work cross-browser:

```
$(window).ready(doSomething);
```

This can be shortened further to:

```
$(doSomething);
```

In all the above examples, *doSomething* is a JavaScript function.

 **New Examples Section!** See all of this code stuff in action, and play around with it.

Load

In other situations it's better to wait for the page to fully load - that is, when all stylesheets and images have been downloaded.

To do this without jQuery, listen for the load event on the window:

```
window.addEventListener('load', doSomething);
```

But it's even easier with jQuery:

```
$(window).load(doSomething);
```

Type checking

Finding out what kind of data is stored in a variable in JavaScript is awkward at best so jQuery provides some tools to help you out:

```
$.isArray([1, 2, 3]);
```

```
true
```

```
$.isFunction(function () { });
```

```
true
```

```
$.isNumeric(10);
```

```
true
```

```
$.isPlainObject({ name: 'Tom' });
```

```
true
```