JavaScript Advanced Tutorial

JavaScript is huge. Like, really big. Like a bus. For elephants. Injected with growth hormones. In this guide we'll look at some of the numerous more advanced aspects of programming with the language.

Contents

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

Object-Oriented Code

Humans are good a categorizing. *Objects* are a way to categorize behavior and data, making large amounts of code easier to design and think about. In fact, *object-oriented* programming can be found in many languages used to build all kinds of software and is considered a very good way to write code.

We've already looked at objects in JavaScript - they're collections of named properties and methods, and they can be created on the fly using the object literal syntax:

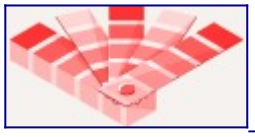
```
var user = {
    name: "tom",
    say: function (words) { alert(words); }
};
```

One reason using objects to hold data is useful is that, as mentioned, human brains are very good at categorizing things: a boat; a chair; a moose. Creating objects in code helps you to think about how the different parts of your code fit and (hopefully) work together.

In a large application you may have a great number of objects interacting with each other. Remember that objects are a grouping together of properties (like *name*) and methods (like *say*)? This grouping of data and behavior into one entity is called *encapsulation*.

A powerful tool available to you when building applications using objects is called *inheritance*. This is where an object *inherits* properties and methods from another object. An object can alter the properties and methods it inherits and add additional properties and methods too.

So, for example, you could create a *moose* object that inherits behavior from a *mammal* object. The *moose* could alter the *mammal*'s *furriness*, for example. You could then create a *pangolin* object that also inherits from a mammal. But, of course, pangolins aren't furry, and they might have a different *scaliness* property.

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

How do objects inherit?

JavaScript uses *prototypal inheritance*. This means that, when an object inherits from another, the parent object is known as the child's *prototype*.

There are a couple of other things to know: every object maintains a reference to its prototype and every object inherits from the global `object` object.

When you ask for a property of an object, JavaScript looks for the property on that object. If it doesn't find it there, it follows the link to the object's prototype and looks for the property there. It continues this, up the *prototype chain* until it finds it, or it returns `undefined`.

In other words, an object inherits properties and methods by maintaining a link to another object from which it wants to inherit. Multiple objects can inherit from one single object, but one single object cannot inherit from multiple other objects, although it can inherit from an object that inherits from another.

In practice

There are many ways to achieve inheritance in JavaScript. The most commonly used is the *constructor pattern*. In this pattern, a function called a *constructor* is used to create new objects. Constructors are used in combination with the `new` keyword to create objects.

Here's an example constructor. Note the capital letter on *Person* - this is an important convention. Generally, in JavaScript, if it's got a capital first letter, it's a constructor and should be used with the `new` keyword.

```
var Person = function (name) {
    this.name = name;
};
```

Now you can use the `new` keyword to create *Person* objects:

```
var tom = new Person('tom');
```

This produces an object like this:

```
{
    name: "Tom"
}
```

In the constructor pattern you manually create the object from which the new objects will inherit by adding properties and methods to the `prototype` property of the *constructor function*. Like this:

```
Person.prototype.say = function (words) {
    alert(this.name + ' says "' + words + '"');
};
```

Now, objects created from the constructor will have the `say` method.

```
var tom = new Person("tom");
tom.say("Hello");


// Produces an alert: tom says "Hello"
```

This is just the start

Inheritance and object-oriented programming are big, complex areas that whole books are written about. If you'd like to know more there are a few great resources out there.

If you'd like to know more about the `new` keyword, check out the thread on Stack Overflow. For a slightly more detailed article about constructors, check out Constructors considered mildly confusing.

Creating Elements

Creating elements using HTML tags isn't the only way to do it — in fact it's possible to create, modify and insert elements from JavaScript.

Here's an example that creates a `div`, adds some text to it and appends it as the last element in the body:

```
var div = document.createElement('div');
div.textContent = "Sup, y'all?";
div.setAttribute('class', 'note');
document.body.appendChild(div);
```

So `document.createElement` is used with an HTML tag to create the element. The `textContent` is then modified and then the class attribute is modified using `setAttribute`. This could also be used to add a data attribute or any other kind of attribute, like you can in HTML! Finally the element is appended to the body using the `body` element's `appendChild` method.

It's essentially equivalent to `<div class="note">Sup, y'all?</div>`.

In fact, all elements have the `appendChild` method, so having done the above it's possible to do the following:

```
var span = document.createElement('span');
span.textContent = "Hello!";
div.appendChild(span);
```

The `span` will be added to the end of the `div` element.

Unfortunately, the `textContent` property used is not supported by all browsers. Internet Explorer (sad face) only supports an equivalent property called `innerText`. Other browsers support `innerText` too but in the interests of web standards it's best to see which property the element supports, and use that:

```
var span = document.createElement('span');
if (span.textContent) {
    span.textContent = "Hello!";
} else if (span.innerText) {
    span.innerText = "Hello!";
}
```

Removing an element

Removing elements is just as fun. Each DOM element has a property that's actually the DOM element's parent. To remove an element, you call the `removeChild` method of the parent, passing the child element you want to remove. Something like this:

```
div.parentNode.removeChild(div);
```

Simple, eh?

Creating with jQuery

The other alternative is to use jQuery's element creation syntax. This example does the same as the above examples combined:

```
var div = $('<div/>').text("Sup, y'all?").appendTo(document.body);
$('<span/>').text('Hello!').appendTo(div);
```

Canvas

The HTML 5 specification has introduced a slew of new and exciting technologies to the web. Here we'll expand on just one: the **Canvas** API. It's for drawing pretty, whizzy things.

Canvas is a new DOM API for drawing on a 2- or 3-dimensional (you guessed it) canvas. What follows just looks at the 2D version, but the 3D technology is called *WebGL* and there are some incredible things being done with it.

The first thing to know is that `canvas` is a new element in the HTML 5 specification. To begin drawing on the `canvas`, you grab hold of it and retrieve a particular context - either `2d` or `webgl`:

```
var canvas = document.querySelector('canvas'),
    ctx = canvas.getContext('2d');
```

From there you can begin drawing, filling and stroking shapes and text on the canvas. Here's an example that draws a simple square on a canvas.

```
<canvas></canvas>

<script>
    var canvas = document.querySelector('canvas'),
        ctx = canvas.getContext('2d');
    ctx.fillRect(10, 10, 10, 10);
</script>
```

Try this out on a page to see a little black square!

The `getContext` function retrieved the correct context, which is an object whose methods are used to draw, fill and generally modify the canvas. Above we're using `fillRect`, which takes the *x* and *y* position of the top left corner of the rectangle as the first two arguments, and the *width* and *height* dimensions as the latter two.

Animation

`canvas` works a bit like a real canvas - pixels are drawn and layered on top of each other until you erase - *clear* - them. This means that if you want to animate something moving across a canvas you have to repeatedly *clear* the canvas (or a section of it) and redraw the scene.

In the next example we've animating the square bouncing around the `canvas`, which automatically resizes to fill the screen. *loop* is the business part of the code. The `clearRect` function is used to clear the entire canvas before each frame.

```
var canvas = document.querySelector('canvas'),
    ctx = canvas.getContext('2d');

var resize = function () {
  canvas.width = window.innerWidth;
  canvas.height = window.innerHeight;
};

window.addEventListener('resize', resize);

window.addEventListener('load', function () {

  resize();

  var pos, vel;
  pos = {
    x: 10,
    y: 10
  };
  vel = {
    x: 1,
    y: 1
  };

  var loop = function () {
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    pos.x += vel.x;
    pos.y += vel.y;
    if (pos.x < 5 || pos.x > canvas.width - 5) {
      vel.x *= -1;
    }
    if (pos.y < 5 || pos.y > canvas.height - 5) {
      vel.y *= -1;
    }
    ctx.fillRect(pos.x - 5, pos.y - 5, 10, 10);
  };

  setInterval(loop, 1000 / 60);
});
```

There’s lots to explore with canvas, this was just a very shallow look at it. There are many specialized resources on the web so, to find out more, Google is your best friend.

Local Storage

When building more complex JavaScript applications that run in a user's browser it's very useful to be able to store information in the browser, so that the information can be shared across different pages and browsing sessions.

In the recent past this would have only been possible with *cookies* - text files saved to a user's computer - but the way of managing these with JavaScript was not good. Now there's a new technology called *local storage* that does a similar thing, but with an easier-to-use interface.

Unlike cookies, local storage can only be read client-side — that is, by the browser and your JavaScript. If you want to share some data with a server, cookies may be a better option.

To save some data, use `localStorage.setItem`.

```
localStorage.setItem('name', 'tom');
```

The first argument is the identifier you'll later use to get the data out again. The second is the data you want to store.

Getting back out again is simple — just pass the identifier to `localStorage.getItem`.

```
var name = localStorage.getItem('name');
```

Local storage can only save strings, so storing objects requires that they be turned into strings using `JSON.stringify` - you can't ask local storage to store an object directly because it'll store "[object Object]", which isn't right at all!

That also means the object must be run through `JSON.parse` on the way out of local storage. You can see this in the example below.

```
// Object example

localStorage.setItem('user', JSON.stringify({
    username: 'htmldog',
    api_key: 'abc123xyz789'
}));

var user = JSON.parse(localStorage.getItem('user'));
```

Browser support for local storage is not 100% but it’s pretty good - you can expect it to work in all modern browsers, IE 8 and above, and in most mobile browsers too.

Errors and Exceptions

Errors are often used to notify another piece of code that something went wrong when trying to carry out a task and that error may then be passed on to the user.

When errors are created — **thrown** — they interrupt the code that's currently being run, unless they are **caught**.

For example, you can force an error to be thrown by giving `JSON.parse` invalid data:

```
JSON.parse("a");


// Produces a SyntaxError
```

You can account for exceptions being thrown using a construct called a *try-catch*. It looks like this:

```
try {
    JSON.parse("a"); // Produces a SyntaxError
} catch (error) {
    // Handle the error
    alert(error.message);
}
```

If an error is thrown in the `try` block it doesn't prevent continued code execution - the error is passed into the `catch` block for you to deal with. Here, for example, you could tell the user that the JSON that was passed in was invalid.

The error passed to the `catch` block (like a function) is an object with a message property you can use to see what went wrong.

In almost all cases you'll want to ensure your code handles errors gracefully. It's worth learning when errors are likely to occur so that you can defend your code against them.

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

Creating errors

It's possible to generate your own errors for someone else to deal with. This is done using the `throw` keyword:

```
throw new Error("I hungry. Fridge empty.");
```

Defensive code

Errors are most common at the front line where data from outside your code is brought in, and so those are the areas you need to be most careful.

Try-catch blocks are one way of handling errors, but you can also check the *type* of some data and provide sensible defaults if you do receive ill-formatted data. Being defensive in your code is a good way of avoiding errors.

Regular Expressions

Regular expressions are another type of data in JavaScript, used to **search and match strings** to identify if a string is a valid domain name or to replace all instances of a word within a string, for example.

Regular expressions have their own syntax and often you'll need to consult a regular expression reference if you're doing something complex. Just search the Internet for what you need!

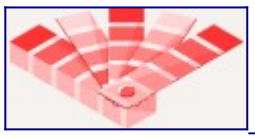
The following is a very high-level overview of a complex area, going through some simple examples to show how to do it in JavaScript.

Creating regular expressions

Like with objects and arrays, there is a *regular expression literal* syntax designated by two forward slashes. Everything between is the expression:

```
var regex = /^[a-z\s]+$/;
```

This expression matches strings that are made of lowercase letters and whitespace from beginning to end. The caret ("^") indicates the start of the string, and the brackets indicate that anything between the opening and closing brace - letter a to z ("a-z"), lowercase, and white-space, indicated by the special "\s" character - can be repeated one or more times up to the end of the string, indicated by the dollar ("$") symbol.

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

Strings have a few useful methods that accept regular expressions; here's a look at `match` and `replace`:

```
var lowerCaseString = 'some characters';

if (lowerCaseString.match(regex)) {
    alert('Yes, all lowercase');
}
```

`match` produces a truthy value if the regular expression *matches* the string (`lowerCaseString`) `match` is called on. The string matches, so execution drops into the `if` statement to show an alert.

Regular expressions can also be used to replace text:

```
var text = "There is everything and nothing.";
```

```
text = text.replace(/(every|no)thing/g, 'something');

// text is now "something and something"
```

This expression matches the word “everything” and the word “nothing” and replaces all occurrences of them with the word “something”. Notice the “g” after the closing forward slash (“/”) of the regular expression - this is the *global flag* that means the expression should match *all* occurrences, rather than just the first, which is what it does by default.

Flags always go after the closing slash. The other flag you might need to use is the *case-insensitive flag*. By default, regular expressions care about the difference between uppercase and lowercase letters, but this can be changed using the “i” flag.

```
var text = "Everything and nothing.";

text = text.replace(/(every|no)thing/gi, "something");

// text is now "something and something"
```

Closures

A **closure** is a function that returns a function. The function that is returned (the *inner* function) is created inside the called function (the **outer**) so - due to the scoping rules we've seen - the *inner* has access to the variables and arguments of the *outer*.

Closures are a powerful tool but can be tricky to understand. Mastering them, however, will allow you to write some very elegant code.

Here's an example:

```
var saver = function (value) {
    return function () {
        return value;
    };
};

var retriever = saver(10);

alert(retriever());
```

```
10
```

The *saver* function returns a function that, when called, retrieves the value that was passed into saver. The *inner* can do so because it was created in the scope of the *outer* and functions created in a particular scope retain access to variables in the scope even if they are called outside of it.

One way to use closures is for creating functions that act differently depending on what was passed to the outer function. For example, here's an *add* function. The value passed to the outer function is used to add to values passed to the inner.

```
var add = function (a) {
    return function (b) {
        return a + b;
    };
};

var addFive = add(5);

alert(addFive(10));
```

```
15
```

The inner, here saved as *addFive*, has access to *a* when it is called because it was created in the same scope as *a*. The inner adds *a* and *b* and returns the result.

In the above example, the inner is saved as *addFive* because it adds 5 to anything passed to it. A different adding function, using strings, could be created in the same way:

```
var hello = add('Hello ');

alert(hello('tom'));


Hello tom
```

Node.js

Much of the code we've seen so far has been with a web browser in mind, using DOM APIs that are only available in that environment. But JavaScript has found its way to the other side of the client/sever divide in the form of *Node.js* (also known as just *Node*).

Node is a platform for building servers in JavaScript, built on Google's V8 JavaScript engine. V8 is also found in the Chromium browser project and derivatives like Google Chrome.

If you have used preprocessors like PHP then Node may take some getting used to, but the main thing to remember is that it works like JavaScript in the browser: idling until an event occurs, responding to that event and returning to idle.

Install

You can grab Node from nodejs.org. You'll get *npm* in there too, which is the *Node Package Manager*, used to install others' modules to make developing things easier.

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

Example

Node comes with a core set of modules, one of which is "*http*", used to set up a web server. Here's a simple HTTP server example that serves one page to all requests:

```
var http = require('http');

var server = http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World\n');
})

server.listen(1337, '127.0.0.1');

console.log('Server running at http://127.0.0.1:1337/');
```

We grab the "http" module then create a server with a single callback function that serves all requests and then set it up to listen on localhost port 1337.

From there things get a whole lot more interesting. Node is used to do everything from clusters of servers to development tools like Grunt. A number of tools and utilities for web development use Node, so it's definitely something that's worth getting familiar with!

JS Apps

These days JavaScript is being used to build large-scale *web applications* and the web is maturing as a platform for doing so. These apps have the advantage of being able to be highly interactive and responsive, requiring fewer page loads and HTTP requests.

Ideas are being taken from the software engineering world on how to build such applications, so here's a few ideas on how to do it before a look at some JavaScript frameworks designed to assist building larger apps.

Modularity

A key concept in software engineering is the idea of *modularity* in code. Separating out the key functionality of your app improves its maintainability and has a number of development advantages.

For example, in a modular system a component can be swapped out at little cost, so long as the interface between the components remains the same. Similarly, different components can be worked on in isolation from each other, which is beneficial for development time and human resource management.

Modules can be connected in a number of different ways - one way is with modules communicating only with a central event and resource management module, sometimes referred to as a *sandbox*, and not communicating directly with each other. A module can publish a notification about an event occurring without caring who or what else will act on it. This means that if one component ceases to function the others may continue, and that a new module can be easily added without affecting the others.

If you want to begin building modular apps, a library called RequireJS is a good place to start.

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

MVC Architecture

Model-View-Controller (*MVC*) architecture is a way of structuring code within an application. A *model* is concerned with the data in your app; the *view* with displaying or outputting data and the *controller* with the business logic and coordinating the models and views. An app might have many of all three.

For example, you might have a *User* model that is responsible for retrieving information about a user. Then you'd have a *Login* controller that managed when to show a log-in screen to a visitor.

The *LoginView* would be responsible for showing the screen and handling form submission, the data from which would be passed to the *Login* controller through to the *User* model.

As with any system there are a number of variations on this, and a number of client-side frameworks have addressed this idea in JavaScript. We'll take a quick look at two: Backbone and Angular.

Backbone

Backbone.js (aka **Backbone**) is designed to add structure to client-side applications, to avoid a spaghetti-code mess.

In the words of Backbone.js:

> Backbone.js gives structure to web applications by providing models with key-value binding and custom events, collections with a rich API of enumerable functions, views with declarative event handling, and connects it all to your existing API over a RESTful JSON interface.

Models are where you keep your data, and in Backbone they are special JavaScript objects created from a Backbone's built-in `Model`, optionally extended by you to add functionality. Making a `Model` looks something like this:

```
var jedi = new Backbone.Model({
    name: 'Yoda',
    age: 874
});
```

You can then connect this model up to a `View` that displays it to the user, via a template. The template is just HTML with special tags or other syntax used to indicate where the model's data should go.

A view looks something like this, including adding it to the page and rendering it.

```
var jediView = new Backbone.View({
     model: jedi,
    tagName: 'div',
    jediTpl: _.template($('#jedi-template').html()),
     render: function () {
          this.$el.html(this.jediTpl(this.model.toJSON()));
          return this;
     }
});

jediView.$el.appendTo('body');

jediView.render();
```

A template, often kept in a `script` element with a type of “template”, contains the HTML that the view should use. For the above example, it might look something like this:

```
<script type="template" id="jedi-template">
     <h1><%- name %></h1>
     <h2><%- age %></h2>
</script>
```


Backbone is a very large, well-used and mature project, and covering it in any depth is way beyond the scope of this tutorial. If you’d like to learn more, Addy Osmani’s freely available book Developing Backbone.js Applications is a great resource.

One thing to consider about Backbone is that it’s dependent on two other JavaScript libraries: *jQuery* and *Underscore*. This makes it quite a heavy dependency to have in your project, and that’s something worth taking into account when deciding whether to use it. Mobile browsers may struggle to download all that code, making your site unusable.

Angular

AngularJS is an open-source client-side JavaScript framework that uses HTML as its templating language. It's notable for it's philosophy that declarative programming is better than imperative programming for wiring up user interfaces and creating modular components.

Angular is MVC but it works quite differently from Backbone.

Templates and *views* in Angular are analogous, and are just regular HTML with some added sugar. In fact, you can put together a (very) simple Angular app without a single line of JavaScript:

```
<input ng-model="name">

<h1>Hello {{ name }}</h1>
```

The `ng-model` attribute on the `input` element connects the value of the input to a *model* called "name". Angular creates this model for us; it doesn't need to be declared elsewhere. Then, in the `h1`, we bind to the value of the "name" model. When you update the `input`, the `h1` will update too.

When you want to build something useful in Angular, you use a controller, which is just a JavaScript function:

```
var AppCtrl = function ($scope) {
    $scope.name = "Yoda";
};
```

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

A controller has a *scope*, which is an area of DOM it has access to. Giving a controller a scope looks like this:

```
<div ng-controller="AppCtrl">
    <input ng-model="name">
     <h1>Hello {{ name }}</h1>
</div>
```

A model exists within a controller's scope, so in the above example the "name" model would be set to "Yoda" by default, but the `input` would update the model and the `h1` as you typed.

You'll notice the *$scope* variable that the controller takes as an argument, but that seems to come from nowhere. This is *injected* into the controller using a system called *dependency injection*. It's a complex area, but is a useful tool for building modular apps.

Angular doesn't depend on anything in the page, and so the file-size footprint tends to be smaller than Backbone. If you're worried about your HTML validating then Angular supports using data attributes instead of the `ng` attributes used above. For example, `data-ng-controller`.