Extra notes - Client-side Design and Development

- Dr Nick Hayward

JS - Logic

A brief introduction to logic in JavaScript.

Contents

Intro

JavaScript is now a core, invaluable technology for client-side design and development. From plain JavaScript to the latest library, its growth as a development environment has exploded over the last few years. It is now being used as a powerful technology to help us rapidly prototype and develop web, mobile, and desktop applications. We can also use it with embedded systems.

Logic

A few underlying concepts for working with *logic* in JavaScript.

blocks

A natural coding style, for JS and other languages, is the simple act of grouping contiguous and related code statements together. Often known as **blocks**, in JS a block is defined by wrapping one or more statements together within a pair of curly braces, `{}`.

Such **blocks** are commonly attached to other forms of control statement, including conditional statements,

```
if (a > b) {
...do something useful...
}
```

conditionals

Conditionals, and by association conditional statements, inherently require a decision to be made. A code statement, and application, will often need to consult **state** and the answer will predominantly be a simple **yes** or **no**.

Within our JS applications, there are many different ways we can express **conditionals**. The most common example is the `if` statement. In essence, we use this statement to check, *if this given condition is true, do the following...*

```
if (a > b) {
console.log("a is greater than b...");
}
```

The `if` statement requires an expression between the parentheses that can be treated as either *true* or *false*.

We can add an additional option if this expression returns false, using a common `else` clause

```
if (a > b) {
console.log("a is greater than b...");
} else {
console.log("no, b is greater...");
}
```

As mentioned above, types that are not matching, in effect the expected type for the comparison, will be coerced by JS to the expected type. For an `if` statement, JS expects a `boolean`.

With this in mind, JS defines a list of values that it considers *false*. These values will become false when coerced to a `boolean`. For example, such values include `0` (and `""`). This means that any value not on this list of *false* values will be considered true, and therefore coerced to *true* when defined as a `boolean`.

Conditionals in JS also exist in another form, which includes the `switch` statement. Further details on conditionals later on.

loops

Programming in general, and JS in this instance, uses loops to allow repeating sets of actions until a given condition fails. In effect, this repetition continues whilst the requested condition holds.

Loops can take many different forms, but in essence they follow this basic behaviour.

A loop includes the *test condition* as well as a *block*, normally within curly braces. Each time this block executes, an iteration of the loop has occurred.

Good examples of this behaviour include the `while` and `do...while` loops. Each repeat a block of statements until a condition ceases to evaluate as `true`.

The basic difference between these loops, `while` and `do...while`, is whether the conditional tested is before the first iteration (`while` loop), or after the first iteration (`do...while`) loop.

If the conditional test returns as `false`, the next iteration of both of these loops will fail to execute. The loop stops.

So, if the condition is initially false, a `while` loop will never run, but a `do...while` will run through for the first time.

We can also stop a JS loop using the common `break` statement.

Another useful form of loop is known as the `for` loop. This loop has three clauses, including

- initialisation clause
- conditional test clause
- update clause

If the goal of the loop is counting, or iterating over a large list or array, it is often more efficient to use a `for` loop. It will often also be the easier option.

There are other specialised forms of loop that will be covered later on.

NB: don't forget, programming languages, and CS in general, start counting at `0`.

functions

Functions are a particularly useful and important part of programming in general. Again, this is no different with JavaScript. In fact, a function may often be considered one of the most important concepts in JavaScript.

In JS, functions are a type of object, and one that we use repeatedly, in different guises, throughout our applications.

In theory, functions, being a type of object, can also have properties. However, it is more common to use this type of object as a grouping of code. They allow us to define once, and then re-use as needed throughout our application. In effect, we can break up our code into more manageable, reusable pieces, and store them in functions with the benefit of abstraction.

In most instances, a **function** is a named grouping of code. This name can be called, and the code will be run each time.

JS functions can be designed with optional arguments, better known as **parameters**, which allow us to pass values to the function. These functions can also optionally return a value. e.g.

```
function outputTotal(total) {
  console.log(total);
}
var a = 49;
a = a * 3; // or use a *= 3;

outputTotal(a);
```

We can obviously update this example to better abstract the code by adding an additional function,

```
function outputTotal(total) {
  console.log(total);
}

function calculateTotal(amount, times) {
  amount = amount * times;
  return amount;
}

var a = 49;
a = calculateTotal(a, 3);
outputTotal(a);
```

- JSFiddle Demo

```javascript function outputTotal(total) { console.log(total); }
function calculateTotal(amount, times) {
amount = amount * times;
return amount;
}

var a = 49;
a = calculateTotal(a, 3);
outputTotal(a);
```

```
   * [JSFiddle Demo](http://jsfiddle.net/ancientlives/0432kzb0/)

  ##### scope
  Scope, or *lexical scope*, is, effectively, a collection of variables and
  associated rules for how we can access them by name.

  So, in JS, **each function gets its own scope**. Variables within a function's
  given **scope** can only be accessed by code inside that function. Also, a
```

```
  variable name has to be unique within a function's scope. However, the same
  variable name could appear in different scopes.

  We can also nest one scope within another. If we nest scopes, then code within
  the inner scope is able to access variables from either the parent or local
  scope.

  However, code only in the outer scope is unable to access any code in the inner
  scope. Effectively, it simply does not have access to this inner function code.

  ##### scope example
  ```javascript
  function outerScope() {
    var a = 49;
    //scope includes outer and inner
    function innerScope() {
      var b = 59;
      //output a and b
      console.log(a + b); //returns 108
    }
    innerScope();

    //scope limited to outer
    console.log(a); //returns 49
  }

  //run outerScope function
  outerScope();
```

- JSFiddle Demo

strict mode

With the introduction of ES5, JavaScript now includes the option to add a **strict mode** to ensure tighter code and better compliance with certain language behaviour and rules. Using this option is often considered worthwhile to ensure greater compatibility, and safer use of the language and its guidelines. It can also help optimise code for better use with rendering engines.

So, we can add it at different levels within our code. For example, we could restrict this **strict mode** to a function level, or enforce it for a whole file. We simply set the required **strict mode** pragma to the required level,

```
function outerScope() {
  "use strict";
  //code is strict

  function innerScope() {
  //code is strict

  }
}
```

If we set the **strict mode** pragma for the whole file, placed at the top of our file, all functions and code will be checked against this strict mode. A potential benefit of **strict mode** relates to the way JS can auto-create global variables for variable declarations missing a `var` keyword. We'll look at this later on. However, an example might be

```
function outerScope() {
  "use strict";
  a = 49; // `var` missing - ReferenceError
}
```
```

variables and scope

When we declare a variable in JS, we can use different keywords relative to intended scope. For example, the standard declarative keyword is

- `var`

which means that the variable will belong to the current scope. If we want to create a new variable with a global scope, outside of any function, we can use the

- `global`

keyword. Such declarations will, of course, influence the scope of usage for a given variable. As we've already seen, we can limit the scope of a variable to local or global usage, and consider a variable's value relative to its availability within that scope.

For example, we can consider such behaviour an example of **hoisting**, which defines the declaration of a variable as belonging to the entire scope, and by association accessible throughout that scope as well. In effect, a declared variable will be moved to the top of the scope. This concept also works with functions, where they are **hoisted** to the top of the scope. In fact, this is often the more common use of hoisting, and preferable to standard misuse with variables.

variables and block-level

We've also briefly discussed the concept of nesting, and scope specific variables. With the advent of ES6, we can increase the level of granularity, and fine control, to create variables that are restricted to the block level.

By using the keyword **let**, we can declare block level specific variables

```
if (a > 5) {
let b = a + 4;

console.log(b);

}
```

let enables us to specify the variable just within the scope of the `if` conditional statement, which means that `b` will not be available to the whole function. One of the obvious benefits is the negation of the possibility of scope pollution within an application. If a variable is only required in a given limited block, why expose it to a broader scope within the app.

variables and strict mode

As a parting thought on variables, let's return to the option to add **strict mode** to our code. As mentioned earlier, one of the inherent benefits of using **strict mode** is the ability to avoid the unfortunate JS habit of creating global variables, in the sense of scope, for variable declarations without the keyword `var`.

```
  function myScope() {
    a = 49;
  }
```

If we add strict mode, this will return a reference error, and `a` will not be defined. Without this strict mode option, we get a variable that will be hoisted to the top, and then either set as a globally available variable, although it could be deleted, or it will set a value for a variable with the matching name. It has basically bubbled up through the available layers of scope until it finds a match.

In effect, it becomes similar in essence to a declared global variable. This can create some strange behaviour in our applications, which can often be tricky to debug. So, remember to declare your variables correctly and at the top.

For example,

```
var a;

function myScope() {
    "use strict";
        a = 49;
}

myScope()
a = 59;
console.log(a);
```

References

- MDN
  - MDN - JS
  - MDN - JS Const
  - MDN - JS Data Types and Data Structures
  - MDN - JS Grammar and Types
- W3 - JS Performance