Extra notes - Client-side Design and Development

- Dr Nick Hayward

JS - Intro & Basics

A brief introduction to 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.

We can use libraries such as jQuery, React, AngularJS, and Node.js, to name but a few, to help us develop powerful, scaleable client and server applications. Using frameworks such as Apache Cordova, we can develop cross-platform applications for mobile devices. Finally, with GitHub's recent Electron project, for example, we can now leverage web technologies to build powerful desktop applications. And, using libraries such as Espruino and Tessel, we can develop applications, controllers, and environments for use with embedded systems.

Basics

We'll now work our way through some of the fundamental concepts for JavaScript development

operators

Operators are a particularly useful, and fundamental aspect of programming. It is no different with JavaScript.

They allow us to perform mathematical calculations, assign one thing to another, compare and contrast, and so on.

With the simple `*` operator, we can perform multiplication,

```
2 * 4
```

Likewise, we can add, subtract, and divide numbers as required within the logic of our applications. We can mix mathematical with simple assignment,

```
a = 4;
b = a + 2;
```

So, in this basic example, we are simply assigning the value `4` to a variable called `a`, and then calculate a value for the variable `b` using `a + 2` resulting in a new total of `6` for `b`.

some common operators

The following is a short summary of the more common operators currently used in JavaScript.

assignment

- `=`
- eg: `a = 4`

comparison

- `<`, `>` `<=`, `>=`
- eg: `a <= b`

compound assignment

- `+=`, `-=`, `*=`, `/=`
- compound operators are used to combine a mathematical operation with assignment
- same as `result = result + expression`
- eg: `a += 4`

equality

operator	description
==	loose equals
===	strict equals
!=	loose not equals
!==	strict not equals

- eg: `a != b`

increment/decrement

- increment or decrement an existing value by 1
  - `++`, `--`
  - eg: `a++` is equal to `a = a + 1`

logical

- used to express compound conditionals - **and**, **or**
  - `&&`, `||`
  - eg: `a || b`

mathematical

- `+`, `-`, `*`, `/`

- eg: `a * 4` or `a / 4`

object property access

- properties in objects are specific named locations for holding values and data
- effectively, values within values
  - `.`
  - eg: `a.b` means object `a` with a property of `b`

values and types

In JavaScript, as with most forms of programming, we are able to express different representations of values often based upon a need or intention for that value. Such representations are known as **types** in common syntax.

JavaScript has **built-in** types, which allow us to represent **primitive** values. For example, if we need to perform a mathematical calculation we need and use **numbers**. For textual documents and output, we use **strings**, and to simply offer an option, **yes** or **no**, right or wrong, we can use a standard **boolean**, which allows us to represent a *true* or a *false* value.

Such values included in the source code are simply known as **literals**, and we can represent them as follows,

- string literals use double or single quotes
  eg: `"some text"` or `'some more text'`
- *numbers* and *booleans* are represented without being escaped, ie: they don't require encapsulating quotes...
  eg: `49`, `true;`

We can also consider and include arrays, objects, functions, and so on within our consideration of values and types for JavaScript. Each will be considered in detail later on.

type conversion

In JS, we have the option and ability to convert, or more correctly **coerce** our values and types from one type to another. For example, if we have a number, which then needs to be printed, we can convert it, or coerce it, to a string. And, logically, we can perform the reverse for a string to a number.

JS provides different options for enforcing such coercion, including the following

```
var a = "49";
var b = Number(a);
```

In JS, variable `a` will be a string, and the resultant number coercion will create a new variable `b` as a number. The built-in JS function, `Number()`, is an explicit coercion, and allows us to convert any type to a number type.

There is also a less specific implicit coercion, which JS will often perform as part of a comparison. For example, if we compare

```
"49" == 49
```

JS will do its best to force the correct answer by implicitly **coercing the left string to a matching number**, and then performing the comparison.

Whilst this is possible in JS, it's often considered bad practice and is something you should try to avoid where possible. It's better to explicitly convert the type, and then perform the comparison.

However, there are rules that JS follows in trying to implement this implicit coercion. We'll look at these rules later on.

comments

As with other languages, JS naturally includes the option to add comments within our code. There are currently two permitted implementations for comments,

single line comment

```
//single line comment
var a = 49;
```

multi-line comment

```
/* this comment has more to say...
we'll need a second line */
var b = "forty nine";
```

Variables

Often referenced as a **symbolic** container for values, and data, applications use such containers to keep track and update values during the various stages of an application. The easiest way to achieve this goal is to use a **variable** as a container for such values and data.

Variables allow values to vary over time, and JS is no different. However, one of the major differences lies in the way it declares its variables and assigns `type`. JS can emphasize types for such values, and does not enforce them on the variable itself.

Known as **weak typing**, or **dynamic typing**, JS permits a variable to hold a value of any type. It can often be a benefit of the language, and a quick way to maintain flexibility in design and development.

In JS, we declare a variable using the keyword `var`. and this declaration does not include any further necessary **type** information.

```
var a = 49;
//double var a value
var a = a * 2;
//coerce var a to string
var a = String(a);
//output string value to console
console.log(a);
```

In the above example, we can see how `var a` maintains a running total of the value of `a`. Therefore, it is able to keep a record of these changes, and effectively the **state** of the value and its small part of the application.

In other words, **state** is keeping track of changes to any values in the application.

We can also use variables in JS to enable central, common references to our values and data. Better known in most languages simply as **constants**, such variables allow us to define and declare a variable with a value that is not intended to change throughout the application.

In JavaScript, the concept of a constant is similar. It creates a read-only reference to a value. However, the value itself is not immutable, it's simply the identifier that cannot be reassigned. The value may be updated, for example, if it's set as an object.

Such perceived **constants** are often declared together, and form a store for values that can be abstracted for use throughout an app. If the value is later updated, this change ripples through the app to each reference to the variable.

As a convention, JS normally defines perceived constants using uppercase letters,

```
var NAME = "Philae";
```

We can also use multiple words in the naming convention for constants, and each word is separated using an underscore, `_`.

```
var TEMPLE_NAME = "Philae";
```

With the advent of ECMAScript 6, or ES6, there is now a new way to declare a constant, which uses the keyword `const` instead of `var`.

```
const TEMPLE_NAME = "Philae";
```

There are many different benefits to using constants, foremost amongst them are the benefits of abstraction, and ensuring that the value is not accidentally changed. For example, if we tried to change the value of the above constant whilst the application was running, it would reject the change. In strict mode, this rejection would lead to the application failing with an error.

Such JS constants are also bound by scoping rules, which means that the value may be updated at block-level, including conditional statements, loops, &c.

identifiers

We've already seen how to declare a variable in JS, but there are also a few rules and best practices for naming valid **identifiers**. Using typical ASCII alphanumeric characters, we can consider such naming rules as follows,

- an identifier must begin with `a–z, A–Z, $, _`
- may contain any of those characters, plus `0–9`

Property names follow this same basic pattern, although we need to be careful not to use certain keywords, or reserved words. Reserved words can include such examples as,

- `break, byte, delete, do, else, if, for, this, while` and so on
- further details are available at the W3 Schools site

References

- MDN
  - MDN - JS
- W3 Schools