Extra notes - JS - Core - Part 1

- Dr Nick Hayward

A brief introduction to some of the core concepts for working with JavaScript.

Contents

Intro A few of the primary, core concepts for working with JavaScript. Many of these concepts are applicable to client-side design, web-stack mobile development, and web-stack desktop application development.

Values and Types JS has typed values, and not typed variables. To help us, JS provides the following built-in types

- boolean
- null
- number
- object
- string
- symbol (new in ES6)
- undefined

Another helping hand is provided by JS's `typeof` operator, which lets us easily examine a value and return its type. We are asking JS for the type of value currently stored in the specified variable. For example,

```
var a = 49;
console.log(typeof a); //result is a number
```

So, as of ES6, there are 7 possible return types for JS. It's also useful to remember that in JS variables do not have types, they are mere containers for the values. It's these values that specify the type.

As a point of interest, if we run the following

```
var a = null;
console.log(typeof a); //result is object
```

The result is an object, and not the expected **null**. This is a known, long standing bug, and one that may never get squashed. Developers have often come to rely on this issue, and it can be seen used in different examples.

Objects Objects, as you might imagine, are particularly useful in JS. In essence, the **object** type includes a compound value, which JS can use to set properties, or named locations. Each of these properties holds its own value, and can be defined as any type. Hence its general flexibility in JS development, and its widespread usage.

```
var objectA = {
  a: 49,
  b: 59,
  c: "Philae"
};
```

We can then access these values using either **dot** or **bracket** notation,

```
//dot notation
objectA.a;
//bracket notation
objectA["a"];
```

Dot notation tends to be more common, and is therefore often preferred for JS usage.

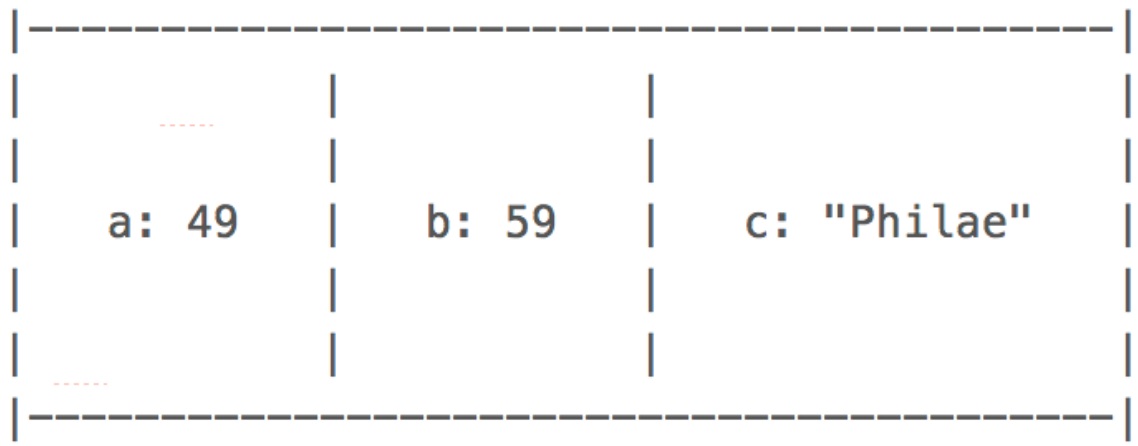

Figure 1: JS Object

Arrays In JS, an array is an object that contains values, again of any type, in numerically indexed positions.

So, we can store a number, a string, and the array will start at index position `0` . It will then increment by **1** for each new value.

These arrays can also have properties, for example the automatically updated **length** property.

```
var arrayA = [
  49,
  59,
  "Philae"
];
arrayA.length; //returns 3
```

Each value can be retrieved from its applicable index position,

```
arrayA[2]; //returns the string "Philae"
```

Due to the nature of arrays, as special objects, we could use them as a catch-all solution for storing our values. We could even add our own named properties, thereby mimicking the functionality of an object. However, this is often considered poor usage, or misuse in many respects, of the functionality of objects and arrays in JavaScript.

Therefore, we can use objects for named properties, and arrays for values with numerically indexed positions.

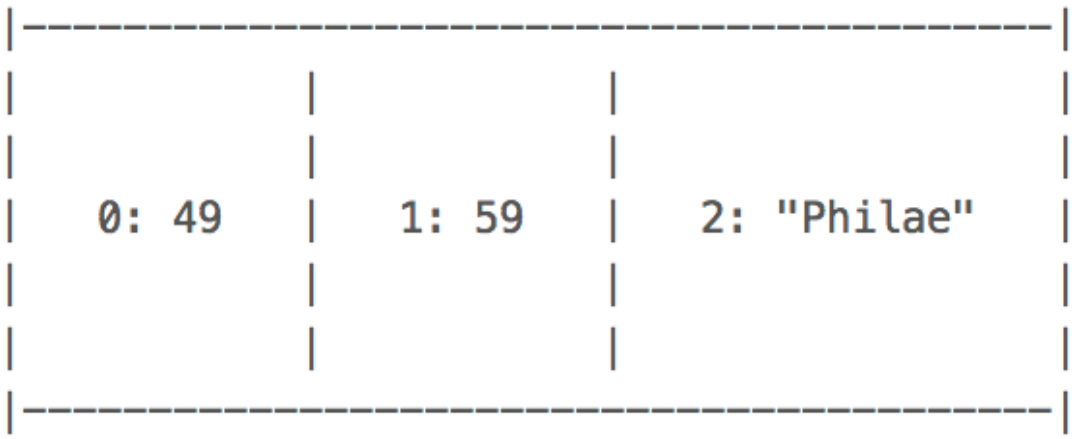

Figure 2: JS Array

Checking Equality In JS, there are four equality operators, which include two **not equal** examples. These include

- `==` , `===` , `!=` , `!==`

The first option, `==` , checks for value equality, whilst allowing coercion. The second option, `===` , will also check for value equality but without coercion. Therefore, this second option is also known as **strict equality**. For example,

```
var a = 49;
var b = "49";

console.log(a == b); //returns true
console.log(a === b); //returns false
```

Therefore, as the rules imply, for the first comparison JS will check the values, and if necessary will try to coerce one or both values to a different type until a match occurs. This allows JS to then perform a simple equality check, which results in `true` .

The second check, however, is far simpler. As coercion is not permitted, a simple equality check is performed, which results in the obvious `false` return. So, an obvious question is which comparison operator should we use. The following are often suggested as useful rules of thumb,

- use `===` if it's possible either side of the comparison could be true or false
- use `===` if either value could be one of the following specific values,
  - `0` , `""` , `[]`
- otherwise, it's safe to use `==` . It will also simplify code in a JS application due to the implicit coercion.

We can also use their **not equal** counterparts, `!` and `!==` . They work is a similar manner to the above.

Checking Inequality Known as **relational comparison**, we can use the operators,

- `<` , `>` , `<=` , `>=`

to check for inequality. Such inequality operators tend to be used to simply check comparable values like numbers, normally those that have an ordinal quality. For example,

```
49 < 59
```

However, we can also use these inequality operators to check strings. This comparison is based on typical alphabetical rules,

```
"hello" < "world"
```

Coercion also occurs with inequality operators, and it should be noted that we do not have to deal with the concept of **strict inequality**. For example,

```
var a = 49;
var b = "59";
var c = "69";

a < b; //returns true
b < c; //returns true
```

Again, if we consider the above results we can see how JS follows a set of prescribed rules and patterns, which informs its decision and outcome. So, in these examples for a `<` operator JS will check whether both values are strings. If true, then it will perform a comparison based upon alphabetical checks. If either value is not a string, it will coerce both sides to **numbers** and perform the comparison.

- we can encounter an issue when either value cannot be coerced into a number

```
var a = 49;
var b = "nice";

a < b; //returns false
a > b; //returns false
a == b; //returns false
```

- issue for `<` and `>` is string is being coerced into invalid number value, `NaN`
- `==` coerces string to `NaN` and we get comparison between `49 == NaN`

JS Best Practices As an end to our initial foray into JavaScript, there are a few guidelines for best practices that are worth considering.

variables There are a couple of useful guidelines for using both global and local variables.

Where at all possible, limit use of global variables in JavaScript. In JS, they are easy to override, can lead to unexpected errors and issues, and should be replaced with appropriate local variables or closures.

Local variables should always be declared with the keyword `var` to avoid the automatic global variable issue.

It's also useful to initialise variables as they are declared. This helps create cleaner code, single declaration and initialisation, and avoids unnecessary undefined values.

declarations As an act of good practice, and to avoid unnecessary or unwanted hoisting, add all required declarations at the top of the appropriate script or file. Whilst providing cleaner, more legible code, it also helps to avoid unnecessary global variables and the unwanted re-declarations.

types and objects Avoid declaring numbers, strings, or booleans as objects. These should be treated more correctly as primitive values, which helps increase the performance of our code, and decrease the possibility for issues and bugs.

type conversions and coercion Due to the weakly typed nature of JS, it's important to avoid accidentally converting one type to another. For example, converting a number to a string or mixing types to create a NaN (Not a Number). Also, we can often get a returned value set to NaN instead of generating an error. For example, if we try to subtract one string from another. However, if we try the following

```
"15" - 10
```

JS will convert the first string to a number, and then perform the subtraction.

comparison With comparisons, it is better to try and work with `===` (**equal value and equal type**) instead of `==` (**equal to**). As we've seen, the main issue that `==` tries to coerce a matching type before comparison. The second comparison, `===` forces comparison of values and type.

JS performance Finally, a few simple steps to help improve general code performance in JavaScript.

loops

Loops are a common feature of JavaScript programming, and it makes sense to limit the number of calculations, executions, and statements performed per iteration of a loop. Therefore, it's useful to check loop statements for assignments and statements that only need to be checked or executed once, rather than each time a loop iterates. The following `for` loop is a standard example of this type of quick optimisation

```
// bad
for (i = 0; i < arr.length; i++) {
...
}
// good
l = arr.length;
for (i = 0; i < l; i++) {
...
}
```

source - W3

DOM access Working with the DOM repetitively can be slow, and resource intensive. Therefore, either try to limit the number of times your code needs to access the DOM, or simply access once and then use as a local variable.

```
var testDiv = document.getElementById('test');
testDiv.innerHTML = "test...";
```

JavaScript loading As alluded to earlier, we do not always need to place our JS files in the `<head>` element. By adding our JS files to the end of the page's body, we allow the browser to load the page first, and importantly the DOM itself.

Traditionally, whilst a browser was downloading a script, it would not start any other downloads. This might also affect parsing and rendering of the page itself, thereby creating a delay in the overall page for the user.

However, whilst this modification in practice has now started to filter into most web app development, it is still not practical for all JS development. For example, if we start building desktop apps, and mobile cross-platform apps we cannot always implement this practice in our HTML.

References

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