

# Extra notes - Client-side Design and Development

- Dr Nick Hayward

## JS - Core

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

### Contents

- Intro
- Values and Types
- Objects
  - objects
  - arrays
- Checking equality
- Functions and values
- More conditionals
  - switch conditional
  - ternary or conditional operator
- Closures
  - examples 1 and 2
  - why use closures?
  - example 3
  - closures and scope persistence
- JS and `this`
  - global, window object
  - object literals
  - example - object literals
  - events
- JS best practices
- JS performance
- References

### 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.

## Image - JS Object

|       |       |             |
|-------|-------|-------------|
|       |       |             |
|       |       |             |
| a: 49 | b: 59 | c: "Philae" |
|       |       |             |
|       |       |             |

## 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.

## Image - JS Array

|       |       |             |
|-------|-------|-------------|
|       |       |             |
|       |       |             |
| 0: 49 | 1: 59 | 2: "Philae" |
|       |       |             |
|       |       |             |

## 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 in 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`

## Functions and values

So far, we've seen variables acting as groups of code and blocks, which act as one of the primary mechanisms for scope within our JS applications.

However, we can also use functions as values, effectively using them to set values for other variables, for example.

```
var a;

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

b = scope()*2;
console.log(b);
```

It's a useful and interesting aspect of the JS language, thereby allowing us to build values from multiple layers and sources.

## More conditionals

We've already briefly considered conditional statements using the `if` statement,

```
if (a > b) {  
  console.log("a is the best...");  
} else {  
  console.log("b is the best...");  
}
```

Switch statements in JS effectively follow the same pattern as `if` statements, but they are designed to allow us to check for multiple values in a more succinct manner. These statements enable us to check and evaluate a given expression, and then attempt to match a required value against an available `case`.

### switch conditional

The addition of `break` is important to ensure that only a matched case is executed, and then the application breaks from the switch statement. If not, execution after that case will continue. This is commonly known as **fall through** in programming. Be aware, however, that this may be an intentional feature of your code design as well. For example, we may wish to allow a match against multiple possible cases, therefore a premature break is not required within the code.

For example,

```
var a = 4;  
  
switch (a) {  
  case 3:  
    //par 3  
    console.log("par 3");  
    break;  
  case 4:  
    //par 4  
    console.log("par 4");  
    break;  
  case 5:  
    //par 5  
    console.log("par 5");  
    break;  
  case 59:  
    //dream score  
    console.log("record");  
    break;  
  default:  
    console.log("more practice");  
}
```

### ternary or conditional operator

As a final point, there is also a more concise way to write our conditional statements using what is known as the **ternary** or **conditional** operator. It's best to consider this operator as a more concise form of the standard `if...else` statement. For example,

```
var a = 59;  
var b = (a > 59) ? "high" : "low";
```

This is equivalent to the following standard `if...else` statement

```
var a = 59;  
  
if (a > 59) {  
  var b = "high";  
} else {  
  var b = "low";  
}
```

```
}
```

## Closures

Closures are a particularly important and useful aspect of JavaScript. Once more, we're dealing with variables and scope, but this time we're considering continued, broader access to ongoing variables via a function's scope.

In effect, we can think of closures as a useful construct to allow us to access a function's scope even after it has finished executing. This can give us something similar to a private variable, which we can then access through another variable using the relative scopes of outer and inner. The inherent benefit is that we are able to repeatedly access internal variables that would normally cease to exist once a function had executed.

### closures - example 1

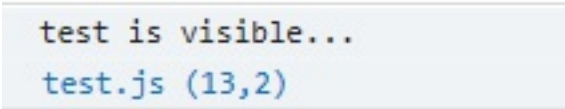
Our first example of explicitly using closures,

```
//value in global scope
var outerVal = "test1";

//declare function in global scope
function outerFn() {
  //check & output result...
  console.log(outerVal === "test1" ? "test is visible..." : "test not visible...");
}

//execute function
outerFn();
```

### Image - closures - global scope



```
test is visible...
test.js (13,2)
```

### closures - example 2

Second example of using closures,

```
"use strict";

function addTitle(a) {
  var title = "hello ";
  function updateTitle() {
    var newTitle = title+a;
    return newTitle;
  }
  return updateTitle;
}

var buildTitle = addTitle("world");
console.log(buildTitle());
```

So, as we can see in this code example, the new function `buildTitle` has become a `closure`. In effect, `buildTitle` is now a special type of object with both a function and the original function environment. The key here is that this function environment includes any local variables within the same scope.

### why use closures?

We use closures a lot in JavaScript, they are a real driving force behind Node.js, jQuery, and many other JS libraries. Closures can help reduce the sheer amount, and complexity, of code necessary to add advanced features to an app. However, they also help us add features to apps that would otherwise be difficult, or impossible, to implement without closures.

e.g. any task using callbacks, including event handlers, would be considerably more difficult without closures. Others, including support for private object variables, would effectively be impossible.

In essence, a closure allows us to work with a function that has been defined within another scope, and yet still has access to all the variables within the defined outer scope. This **allows basic encapsulated data**, in effect giving us a way to store data in a separate scope, and then share it where needed.

We can also use this same approach to repeatedly make new functions.

```
function count(a) {  
  return function(b) {  
    return a + b;  
  }  
}  
  
var add1 = count(1);  
var add5 = count(5);  
var add10 = count(10);  
  
console.log(add1(8));  
console.log(add5(8));  
console.log(add10(8));
```

- using one function to create multiple other functions, `add1`, `add5`, `add10`, and so on.

### closures - example 3

```
//variables in global scope  
var outerVal = "test2";  
var laterVal;  
  
function outerFn() {  
  //inner scope variable declared with value - scope limited to function  
  var innerVal = "test2inner";  
  //inner function - can access scope from parent function & variable innerVal  
  function innerFn() {  
    console.log(outerVal === "test2" ? "test2 is visible" : "test2 not visible");  
    console.log(innerVal === "test2inner" ? "test2inner is visible" : "test2inner is  
not visible");  
  }  
  //inner function now added to global scope - now able to access elsewhere & call  
  later  
  laterVal = innerFn;  
}  
//invokes outerFn, innerFn is created, and its reference assigned to laterVal  
outerFn();  
//innerFn is invoked using laterVal - can't access innerFn directly...  
laterVal();
```

### Image - closures - inner scope



```
test2 is visible
test.js (15,5)
test2inner is visible
test.js (16,5)
```

## closures and scope persistence

So, how is the `innerVal` variable still available when we execute the inner function. In particular, if we consider that the scope in which it was created has gone.

This is why **closures** are such an important and useful concept in JavaScript. It is this use of closures that creates a sense of persistence in the scope.

This scope persistence, and delayed access to functions and variables, is why closures are so useful in JavaScript. The closure creates a safe wrapper around the function, and any variables that are in scope as a function is defined. In effect, the closure ensures that the function has everything it needs to execute correctly.

This wrapper, including the function and variables, then persists as long as the function exists.

However, there may also be a cost to memory and performance if we construct all of our app using this pattern. There is no specifically defined closure object, for example, and each function comes with its own entourage, so to speak. These associated variables and values, for example, do not come free. There will be some memory usage, so you'll need to check if this is an issue with your app or not.

## JS and `this`

A commonly misunderstood feature of JavaScript is appropriate and, in many examples, correct usage of the keyword `this`.

Unlike many other programming and scripting languages, the value of `this` is not inherently linked with the function itself. Instead, it is **a response to how the function is called that determines the value of `this`**.

Therefore, the value itself can be dynamic, simply based upon how the function is called programmatically. If a function contains `this`, its reference will usually point to an `object`.

So, let's take a quick look at some of the initial ways we can use, and update, `this`.

There are a number of ways that the value of `this` can change, depending upon the originating context. When we call functions, the way we call them will inherently affect the resultant value of `this`.

## global, window object

When we call a function, we can bind the `this` value to the `window` object. The resultant object refers to the root, in essence the `global` scope. So, if we use `console.log()` to output the value of `this`, we should expect the resultant value to be our current `window` object.

```
function test1() {
  console.log(this);
}

test1();
```

**NB:** the above will return a value of `undefined` in strict mode.

- [JSFiddle - this - window](#)

We can also check for the value of `this` relative to the global object,

```
var a = 49;

function test1() {
    console.log(this.a);
}

test1();
```

- [JSFiddle - this - global](#)

### object literals

Within an object literal, the value of `this`, thankfully, will always refer to its own object.

```
var object1 = {
    method: test1
};

function test1() {
    console.log(this);
}

object1.method();
```

- [JSFiddle - this - literal](#)
- [JSFiddle - this - literal 2](#)

So, the return value for `this` will be the object itself. In the above simple example, we get the returned object with a property and value for the defined function. If the object contains other properties and values, these will be returned and available as well.

### example - object literals

```
var sites = {};
sites.name = "philae";

sites.titleOutput = function() {
    console.log("Egyptian temples...");
};

sites.objectOutput = function() {
    console.log(this);
};

console.log(sites.name);
sites.objectOutput();
sites.titleOutput();
```

We can see example output for this code in the following image,

### Image - Object Literals console output

```
philae
test.js (22,1)
  > [Object Object]      {name: "philae"}
    test.js (19,3)
Egyptian temples...
test.js (15,3)
```

## events

For events, the value of `this` points to the owner of the bound event. For example, if we bind the following event

```
<div id="test">click to test...</div>
```

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

function output() {
  console.log(this);
};

testDiv.addEventListener('click', output, false);
```

- [JSFiddle - this - events](#)

When this element is clicked, the value of `this` becomes this element.

## 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.

### defaults

Where parameters are required by a function, a function call with a missing argument can lead to it being set as **undefined**. Therefore, it is good coding practice to at least assign default values to arguments to help prevent issues and bugs.

### switches

As already mentioned, **fall-through** in switch statement can be useful but you still need to consider a default for the conditional statement. Therefore, ensure you always set a `default` to end a switch statement.

### 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](#)