JAVASCRIPT
TECHNICAL INTERVIEW QUESTIONS

PREPARE YOURSELF FOR THAT DREAM JOB

Javascript Technical Interview Questions

Prepare Yourself For That Dream Job

Xuanyi Chew

This book is for sale at http://leanpub.com/jsinterviewquestions

This version was published on 2014-03-19

Contents

Sample of Underhanded JavaScript

Hi there!

This is the sample of Underhanded JavaScript: How to Be A Complete Arsehole With Bad JavaScript. Included are three chapters (there are 12 more in the proper book when published). I hope you have fun reading it, and eventually, do buy it.

Introduction

The motivation for me to write this book was originally boredom. I was joking with my colleagues at my startup that JavaScript has so many pitfalls that I could write a book on it. I was then egged on to actually write it, and so I started drafting a series of blog posts when Hacker News went down and I had no other entertainment. Eventually, it became obvious that a collection of JavaScript quirks would be better suited in a book format than a blog post.

At the same time, I was doing a few friends a favour by helping them out with interviewing a few candidates for a JavaScript position. I found that the candidates were not able to answer questions derived from the first three chapters of the book. To be fair, I did ask rather obscure questions about JavaScript and I didn't expect the candidates to do well, for even the most experienced of JavaScript developers fall for some of these. For example:

Do you know why `"true" == true` will return `false`? What really happens behind the scenes when variables are "hoisted"? Why do numbers in JavaScript act so strangely?

This book aims to help potential candidates on their job interview. I do so by debunk the strangeness, and make it not strange. JavaScript doesn't have quirks once you understand the language. And it is only with mastery of the language will a technical interview go over smoothly. In the book, I also suggest ways to further a conversation in an interview.

I had fun writing this book. And I sincerely hope that it will, to some extent help you in getting that dream job of yours.

On IIFE Behaviour

Immediately Invoked Functions, in particular Immediately Invoked Function Expressions[1] are a commonly used Javascript design pattern. An IIFE is basically where a function is defined, and then immediately called. It is quite useful in compressing code and reducing scope lookups for more effective execution of code.

Many people also use IIFEs to prevent pollution of the global namespace. If function `foo()` is ever going to be called once, and it isn't ever going to be referenced again, why add `foo` to the global object? Another oft-quoted reason is for scope privacy. This is especially true in the case of Javascript libraries. By using IIFEs, the variables are all contained within the library's local scope.

Question: What do these return?

Example 1

```
function foo(x) {
        console.log(arguments)
        return x
}

foo(1, 2, 3, 4, 5)
```

Example 2

```
function foo(x) {
        console.log(arguments)
        return x
}(1, 2, 3, 4, 5)
```

Example 3

```
(function foo(x) {
        console.log(arguments)
        return x
})(1, 2, 3, 4, 5)
```

Answer:

[1] http://benalman.com/news/2010/11/immediately-invoked-function-expression/

Example 1	Example 2	Example 3
`[1, 2, 3, 4, 5]`	`5`	`[1, 2, 3, 4, 5]`
`1`		`1`

Why?

The Explanation

The reason why Example 2 returned 5 is because the function definition is actually a type of function definition called a *`FunctionDeclaration`*. The ECMAScript specification discriminates between two different kinds of function definitions: *`FunctionDeclaration`*, and *`FunctionExpression`*.

Both are function definitions, and a *`FunctionDeclaration`* and a *`FunctionExpression`* can both have the same syntax. However, how a function definition is parsed as a *`FunctionDeclaration`* depends on a few things. Firstly, the function definition must be named - anonymous functions are not *`FunctionDeclaration`*s. The function definitions in all three examples above has a name each. Check.

Secondly, the function definition cannot be part of an expression. This means that the function definition in `a = function a(){}` is a *`FunctionExpression`* and not a *`FunctionDeclaration`* because it's part of an assignment expression. In Example 3, the addition of the parenthesis before the function definition makes it part of an expression, specifically a grouping expression. However, both Example 1 and 2 are function definitions that stand on their own.

Thirdly, *`FunctionDeclaration`*s cannot be nested inside non-function blocks[2]. In V8/Chrome, the function definitions in Examples 1-3 are actually wrapped in a global function block. This is invisible to the end user, and it's only when one uses V8's AST output (`d8 --print-ast` is the command), one sees the global wrapping block.

These are defined in §13 of the ECMAScript 5.1 specification[3], for those who are interested in reading more. However the specification isn't very clear on the latter about *`FunctionExpression`*, and much of it was intuited from reading V8's source code.

Another factor in play is the role of scopes. Scopes in Javascript are called Environments. There are different kinds of Environments[4], but for simplicity, let's just treat them all the same. Environments hold all the variables defined in it. All code belong to an environment or another. If Example 1 is all there is to the code (i.e. `foo()` is not defined inside another function or object), then the environment that `foo()` belongs to is the environment of the global object (in most browsers, this would be `window`).

So, when the code is parsed, the definition of `foo()` in both Example 1 and 2 is treated as a *`FunctionDeclaration`*. What this means is that the function definition is "hoisted" to the top. This makes `foo()` a property of the global object.

[2]Not really, as we will see in a later chapter of this book.

[3]http://www.ecma-international.org/ecma-262/5.1/#sec-13

[4]There are two kinds of environments: **Lexical Environments** and **Variable Environments**. In general, Variable Environments is used to manage variable use and scope at execution time, while Lexical Environments are used at definition time. This is also commonly known in various other forms in other languages as static scoping. We're only concerned about the Lexical Environment. Lexical environments can be made of two different types of *`EnvironmentRecord`*: *`DeclarativeEnvironmentRecord`* and *`ObjectEnvironmentRecord`*. For the most part, most code languish in the *`DeclarativeEnvironmentRecord`*. The Global Object has its own *`ObjectEnvironmentRecord`* though, and when variables are "hoisted", they are hoisted into the Global Object's *`ObjectEnvironmentRecord`*

Why does this matter?

Because when the code in Example 2 is parsed, it's essentially being parsed as if you did this:

```
function foo(x){
        // code
};

(1, 2, 3, 4, 5) // group expression
```

Chrome and Firefox will happily parse it as a function definition followed by a group expression, which returns `5` in the above case.

So, the solution is to make your function definition as an expression (hence the term Immediately Invoked Function *Expression*). Expressions are evaluated and the results are output, then promptly thrown away (or its values are assigned to a variable). So you'd get a `ReferenceError` if you did this in Chrome:

Example 4

```
(function foo(x) {
        console.log(arguments)
        return x
}(1, 2, 3, 4))

foo() // this throws ReferenceError
```

It is precisely this behaviour which makes IIFEs so well-loved. An IIFE protects the variables inside it, and provides privacy from the external environments. An IIFE cannot have its name shared with another function declaration, and hence name collisions will not happen - it cannot be accidentally called. This is what is meant by preventing pollution of the global namespace.

Keen-eyed readers will note that Example 4 above is slightly different from Example 3 and wonder why. Example 4 uses the syntax that Douglas Crockford recommends and therefore more commonly seen in the wild. However, Example 3 is still correct, and does make clear the structural difference of a *`FunctionExpression`*.

Exceptions

The Node.js REPL console actually treats Example 2 as a *`FunctionExpression`* by wrapping it in parentheses. In fact, if you did call `foo` after executing that, a `ReferenceError` will be thrown. When executing a script in the file, however, Node.js will not treat Example 2 as a *`FuncionExpression`*. In fact nothing will be returned if you had put Example 2 in a file and run using `node foo.js`, exactly because there is nothing to be returned.

This is also the reason why after a function has been defined in Node.js REPL, it can be invoked thusly: `foo)(`. This however, has been fixed in the latest version of Node.js

In repl.js[5], the following can be found:

[5]https://github.com/joyent/node/blob/master/lib/repl.js

Node.js' REPL source code snippet, accurate as at 7th January 2014

```
if (!skipCatchall) {
     var evalCmd = self.bufferedCommand + cmd;
     if (/^\s*\{/.test(evalCmd) && /\}\s*$/.test(evalCmd)) {
       // It's confusing for `{ a : 1 }` to be interpreted as a block
       // statement rather than an object literal. So, we first try
       // to wrap it in parentheses, so that it will be interpreted as
       // an expression.
       evalCmd = '(' + evalCmd + ')\n';
     } else {
       // otherwise we just append a \n so that it will be either
       // terminated, or continued onto the next expression if it's an
       // unexpected end of input.
       evalCmd = evalCmd + '\n';
     }

     debug('eval %j', evalCmd);
     self.eval(evalCmd, self.context, 'repl', finish);
   } else {
     finish(null);
   }
```

The justification of wrapping REPL commands in parentheses are also listed in the source. It was mainly done to prevent examples like `{ a : 1 }` from being interpreted as a block statement. The purpose is noble - it prevents developers from shooting their own feet. But it also changes the code. While ambivalent to that idea, I think it's best to tolerate it.

During The Interview

Questions regarding the parsing of IIFEs are not the usual questions asked. However, I have found that it is unusually tough for candidates to answer questions like in the first example. Usually in panic and in the lack of computers to test code on, the little things like missing parentheses are often missed. Remember, when faced with these sorts of tricky questions, it's perhaps best to take a step back and take the question slowly. Just recall that there are differences in function definitions in JavaScript, and you'll be fine. Most interviewers will not ask about environments or only ever lightly touch on scope chains when discussing IIFEs.

Since IIFEs are a well trod pattern in JavaScript, you will be expected to know quite a bit about IIFEs. Perhaps you will be asked to correct a broken IIFE, or design a IIFE that will be namespaced globally. It shouldn't be too difficult. The topic of IIFEs also allow you to branch out the conversation into the module patterns and even the pros and cons of global variables.

On The Comma Operator

Arrays in Javascript are different from arrays in other languages. Arrays are merely Javasript objects with special properties. This allows for strange things to be done, such as setting the array index to a non-integer like so:

```
> a = [1,2,3,4]
> a[1.5] = 1.5
> a
[1, 2, 3, 4]
> a[1.5]
1.5
> a["foo"] = "bar"
> a["foo"]
"bar"
```

To access the i^{th} element of an array, the syntax is `a[i]`. Square brackets are also used for array literals, which declares, initializes, and populates the array, as seen in the example above.

Question: What does Example 1 below return?

Example 1

```
> a = [1, 2, 3, 4][1, 2, 3]
> b = [1, 2, 3, 4][3, 2, 1]
```

Answer `a` is `4`. `b` is `2`.

Why?

The Explanation

This was actually hinted in the previous chapter about IIFE behaviours. When multiple arguments were "passed into" the *`FunctionDeclaration`*, it was actually evaluated as a group expression: `(1,2,3,4,5)`. This evaluates to `5`.

In Example 1 above, given the answers, it should be quite clear that `[1, 2, 3]` evaluates to `3` and `[3, 2, 1]` evaluates to `1`. The obvious explanation is that only the last value is considered or evaluated. Hence `a[1, 2, 3]` is as if it were equivalent to `a[3]`. That's not entirely true. In fact, Example 1 is being parsed as this:

```
a = [1, 2, 3, 4][((1, 2), 3)]
b = [1, 2, 3, 4][((3, 2), 1)]
```

What does that mean? And why does it do that?

The parenthesis were added for clarity - to show the comma operator[6] at work. The comma operator is a rarely used operator, but it's often quite useful. In a Javascript developer's day-to-day work, perhaps the most commonly seen use of the comma is when declaring variables, like so:

Example 2: Variable declarations with the comma

```
function foo(){
        var a = 1,
            b = 2 // both a and b are variables local to foo()
}

function bar() {
        var a = 1 // no comma. a ; is automatically inserted by the parser
            b = 2 // oops. b is now a global variable.
}
```

Declaring variables is NOT actually using the comma operator. There is however, a general use case for the comma operator.

The comma operator is very much like the semicolon. Where the semicolon delimits statements, the comma operator separates expressions. Expressions are always evaluated and produce a result. What the comma operator does is discard unused results. Consider an expression that looks something like this: *`sub-expression1, sub-expression2`*. The comma operator indicates that *`sub-expression1`* is to be evaluated, and the result is thrown away, then *`sub-expression2`* is evaluated and returned (as it is the last expression to be evaluated). It essentially acts as a continuation of sorts - i.e. evaluate *`sub-expression1`* THEN evaluate *`sub-expression2`*. This behaviour can be seen everywhere where the comma operator is used: in array literals, function call arguments and variable declaration blocks.

The case of Example 1 is simply a case of abusing the syntax for added confusion. When `[1,2,3,4][1,2,3]` is parsed, a machine understands that the second `[` is the start of the array or object property accessor. Hence the expression inside the accessor is simply treated as a vanilla expression with the comma operator. The confusion mainly lies in that the reader expects `[1,2,3]` to be an array literal as well.

The key takeaway though, is that a sub-expression in an expression with comma operators will be evaluated even if no results were returned for that specific sub-expression. In some cases, it can be useful, but for most uses, this can lead to some nasty surprises.

[6] http://www.ecma-international.org/ecma-262/5.1/#sec-11.14

Step-by-Step

This is how Example 1 is evaluated:

1. Declare `a` as a variable (in the global scope)
2. Declare `b` as a variable (in the global scope)
3. Set `a` to the result of the evaluation of `[1, 2, 3, 4][1, 2, 3]`. It is evaluated as such:
    i. Allocate array `[1, 2, 3, 4]`
    ii. Evaluate the accessor `[1, 2, 3]`. It is parsed as if this were passed in: `[((1, 2), 3)]`
    iii. The first evaluation of `(1, 2)` happens. `1` is parsed as a literal expression, and the result is thrown away. `2` is then parsed as a literal expression and returned.
    iv. The second evaluation of `(2, 3)` happens. `2` is parsed as a literal expression, and the result is thrown away. `3` is then parsed as a literal expression and returned.
    v. The parsing of the expressions with comma operators is complete. The returned result is `3`
    vi. Since this is an array accessor, access element `3` of the array.
    vii. Assign the result of element `3`(`4`) to `a`.
4. Set `b` as the Step `3` above.

Miscelleny

As previously mentioned, the comma that are found in array and object literals, as well as declaration blocks[7] and parameters lists are not actually comma operators. While the commas in these cases are not the comma operator[8], in a way, they act exactly as if the comma operator were used, except that the results of the evaluated sub-expressions are not thrown away, but are allocated in memory instead.

Example 2 above is a very good demonstration of the use of the comma operator in the special case of Variable Declarations. A Javascript parser will parse `var a = 1, b = 2` as this[9]:

[7] `foo()` in Example 3 is an example of a variable declaration block. In `bar()` of the same example, the variable declaration block ends when the semicolon is automatically added at the end of `a`'s assignment.

[8] The comma tokens are being specially specified in the specification for array literals and object literals (in the spec they're called *`Elision`*), as well as function call arguments, and variable declaration

[9] The Javascript example used in here is illegal though. The use of the `{` after `var` is to illustrate the block when the comma is used. Basically, it's as if there were a *`Block Statement`* there.

Example 3: What a parser parses a variable declaration block as

```
var a = 1,
    b = 2,
    c

// this is parsed as if it's in a block, like so:
var {
    // declare a as a local var, then assign 1 to a
    a = 1, // evaluate this line, THEN

    // declare b as a local var, then assign 2 to b
    b = 2, // evaluate this line, THEN
    // b is declared as a local var because the comma operator
    // indicates that it's still within the var block

    // declare c as a local var. No assignments happen
    c
}
```

The comma in Example 3 plays the role of informing the parser that the Variable Declaration block is not complete yet, which is why `b` and `c` are declared as a local variables.

Likewise for an array literal. The comma operator too performs its task but does not throw away the result. Consider:

```
a = [1, 2+3, function(){return 4}(),]
```

A Javascript parser, when parsing the `[`, knows it's going to be an array literal. It evaluates `1`, and then stores the result as the 0th value of the array. Then it parses `2+3`, and stores the results as the 1st value of the array. Then it parses the Immediately Invoked Function Expression `function(){return 4}()`, and stores it as the 2nd value of the array.

When the parser reached the last `,` and finds no more expressions beyond that, because the array literal is a special case, it omits the last `,` it finds.

This only applies to array literals and object literals. In any other use where there is a trailing comma, a `SyntaxError` will be thrown instead, because the parse expects something after that.

Practice Questions

A lot of JavaScript code in real life will have confusing syntax. Consider this:

```
var a = [1, 2, 3, 4]

x = a[function(){
                a.push(5, 6, 7, 8)
        }(),
        function(){
                a[0] = math.random()
        }(),
        function(){
                a = a.toString()
        }(),
        7]
```

What will the expression above return? Why? Can you work through the expression manually, step-by-step? Can you rewrite the expression above to look saner?

During The Interview

A typical interview will not typically include questions about the comma operator, unless it's like the examples and practice question above. Most questions which involve the comma operator are disguised as problems about arrays or grouping operators. These are not meant to be tricky, as code that involve the comma operator exist in the wild, and occasionally you'd need to be able to understand what's going on in order to figure out a problem.

On Object Coercion

Consider a scenario where a developer has extended the properties of `String` and `Number`, providing a method to all strings and numbers to check whether they are really strings or numbers:

Example 1

```
> String.prototype.isString = function() {
        return this === String(this)
}

> "Hello World".isString()
```

Example 2

```
> Number.prototype.isInt = function() {
        return this === ~~this
}

> 1..isInt()
```

Question: What is the result of evaluating those

Answer: `false`, `false`.

What is perhaps interesting is that both examples above imply that `"Hello World"` is not a String object and `1` is not a Number object! Why?

Asides: Weird JavaScript Syntax?

To a casual JavaScript developer, the above code may seem wrong (such as `1..isInt()`), but rest assured, it is correct. Most parsers would parse the first `.` that comes after a number as part of the number literal. It's not an unreasonable assumption, given that it could be part of a decimal number. Therefore the second `.` is required to act as a member operator (i.e. to call the `isInt()` method). Essentially `1..isInt()` is as if you had written this: `1.0.isInt()`

`~~this` is a rather clever if confusing use of the binary negation operator. A `~` negates a number: `~1` is equal to `-1`. Using two binary negations converts any number in JavaScript into an integer quickly[a] since the binary operator only works on integers. It was added in the above example to be extra arseholish. This method has a caveat: it only works for positive numbers.

But I digress. Good programmers would just use `Math.floor(x)`.

[a]except in V8/Chrome, where it is an anti-optimization. In Chrome/V8, `Math.floor(x)` performs much quicker.

The Explanation

Like a good detective, we must follow clues as to why this does not work as intended. An obvious clue is in the === that was used. Using a strict equality comparator means that in both Example 1 and 2 there is a type mismatch somewhere. In fact, the following examples shows the type mismatch quite clearly.

Example 1's this

```
> String.prototype.isString = function(){
        console.log(this);
        console.log(String(this));
        return this === String(this)
}

> "Hello World".isString()
String {0: "H", 1: "e", 2: "l", 3: "l", 4: "o", 5: " ", 6: "W", 7: "o", 8: "r"\
, 9: "l", 10: "d", length: 11, isString: function}
        __proto__: String
        [[PrimitiveValue]]: "Hello World"
"Hello World"
false
```

Example 2's this

```
> Number.prototype.isInt = function() {
        console.log(this);
        console.log(~~this);
        return this === ~~this
}

> 1..isInt()
Number {isInt: function}
        __proto__: Number
        [[PrimitiveValue]]: 1
1
false
```

In Example 1, `this` is an instance of the String object. `String(this)` however, both provides a string primitive and a syntax that would confuse people. `typeof this` would return `"object"` while `typeof String(this)` would return `"string"`.

In Example 2, `this` is an instance of the Number object. `~~this` is an expression, which converts `this` which is an instance of the Number object, into its primitive value, a value with the number type.

In JavaScript, when a method call is made on a primitive type (such as a number, or string), the interpreter automatically converts the primitive into an object. In the case of Example 1, the string literal `"Hello World"` is of a primitive type (i.e. a string type). When `.isString()` is called on the string literal, the JavaScript engine creates a new object as if `new String("Hello World")` was called before applying the method onto the object. Likewise for `1`. A new object is created as if `new Number(1)` was called, and then `isInt()` is applied to the newly created object. Thus `this` in both methods actually refer to the newly created object, as primitive values cannot have methods or properties.

For those looking at the specification for documentation on these behaviours, they are strangely not actually documented on the specification of the String or Number object - instead the reason why automatic coercion to object happens is the effect of conforming to §11.2.1 in the ECMAScript specification[10]. Here is the gist in English:

1. Given an expression `MemberExpression.Expression` or `MemberExpressio[Expression]`, figure out what kind of expression `MemberExpression` is actually.
2. Call `GetValue()` for `MemberExpression`. Call it `baseValue`
3. Figure out what kind of expression `Expression` is.
4. Call `GetValue()` for `Expression`. Call it `propertyNameValue`.
5. Check that `baseValue` is actually coercible to an object.
6. Coerce `propertyNameValue` to a string.
7. Check whether the function code is actually in strict mode. Call it `strict`.
8. Return a *`Reference`* whose base value is `baseValue` and whose referenced name is `propertyNameString`, and whose strict mode flag is `strict`.

It's here where we can make a couple of very interesting observations. But firstly, let's examine what happens in Example 1 and 2. `MemberExpression` in Example 1 is a string literal, and is of `string` type. In Example 2, `MemberExpression` is a number literal, and is of `number` type. So far so good. Let's move on to Step 2.

`GetValue()` is something we've encountered in the previous chapter on the `new` operator. However we've never fully explored it[11]. It is in fact the `GetValue()` function which coerces primitives into objects. The algorithm for `GetValue()` is defined in §8.7.1 in the ECMAScript specification[12]. Here's the gist of the algorithm:

1. Check that the value being passed in is a reference.

[10]http://www.ecma-international.org/ecma-262/5.1/#sec-11.2.1

[11]mostly due to the fact that the Reference specification type documentation is really quite poorly written and would require a lot of explanation on the prior points. The Reference specification type is actually very well defined, but really poorly worded.

[12]http://www.ecma-international.org/ecma-262/5.1/#sec-8.7.1

2. Get the actual value of the reference - this is called the base value in the specification - call this `base`
3. Check if the reference is actually resolvable.
4. If `base` is a primitive value, then
    a. Set a special `[[Get]]` internal method as the internal getter function the object.
    b. Return the result of calling the method, with `base` as the `this` value, and the referenced name as the property name.

The `[[Get]]` method is an internal method of an object to retrieve the object's property. It takes an argument, which is the name of the property. At this point you may be wondering how would a primitive value have an internal method. The answer is in the first step of the special `[[Get]]` internal method, which converts `base` into an object:

> 1. Let `O` be `ToObject(base)`.
> 2. Let `desc` be the result of calling the `[[GetProperty]]` internal method of `O` with property name `P`.
> 3. … (the rest are the same as the normal `[[Get]]` internal method algorithm) …

It is the step after 4a that is rather confusing. However, to me it just means that when the `[[Get]]` internal method is called (i.e. when getting properties), the result is returned with `base` as the `this` value. The special `[[Get]]` internal method then handles the rest.

Therefore when a method of a primitive is called, it is converted into an object with an existing prototype containing all the methods of the object. A lookup the prototype chain is then performed and the result of the method (if found) is called. This is also the reason why the extension is added to the prototype of the object instead of the object itself. Here's what happens when you try to add a property to a primitive:

```
1  > x = "Hello World"
2  > x.logThis = function() {
3          console.log(this)
4   }
5  > x.logThis()
6  TypeError: Object hello world has no method 'logThis'
```

So why does this happen? When Line 2 is executed, `GetValue()` is called on `x`, and indeed, a new String object (let's call it `O`) is created. The `logThis` property is attached to that new String object, and the function is attached to the `logThis` property. But, because no reference is made to `O` is made, so as soon as the program moves to Line 3, `O` is garbage collected.

So we're now at Line 5. Again, `GetValue()` is called on `x`, and yet again a new String object (let's call this one `O2`) is created. `O2` is not `O`, and does not have the `logThis` property. Which is why a TypeError is raised.

Practice Questions

Can `null` or `undefined` be extended? What happens when `null` or `undefined` gets coerced into objects? Can you explain what happens step by step?

During The Interview

During a technical interview, it is usual to ask candidates if primitives can accept properties. This is a simpler form of the question in the examples above, and the explanations above will serve very well when explaining why a primitive can or cannot accept properties and methods.

The question is also asked in the form of extending the properties of built-in objects. Remember that primitives cannot take a property or method. They will have to be coerced into objects (either automatically or manually) before being able to take properties or methods.

This chapter is particularly useful for talking points discussing primitive-to-object coercion. You can also segue the topic into extension of built-in objects and host objects, as well as their pros and cons.