The Definitive JavaScript Handbook for your next developer interview

medium.freecodecamp.org/the-definitive-javascript-handbook-for-a-developer-interview-44ffc6aeb54e

JavaScript is the most popular programming language and has been since 2014, according to Stack Overflow Survey. It is no wonder that over 1/3rd of all developer jobs require some JavaScript knowledge. So, if you plan to work as a developer in the near future, you should be familiar with this extremely popular language.

The post's purpose is to bring together all JavaScript concepts that are frequently brought up in developer interviews. It was written so you can review everything you need to know about JavaScript in a single place.

Types & Coercion

There are 7 built-in types: `null`, `undefined`, `boolean`, `number`, `string`, `object` and `symbol` (ES6).

All of these are types are called primitives, except for `object`.

Null vs. Undefined

Undefined is the absence of a definition. It is used as the default value for uninitialized variables, function arguments that were not provided and missing properties of objects. Functions return `undefined` when nothing has been explicitly returned.

Null is the absence of a value. It is an assignment value that can be assigned to a variable as a representation of 'no-value'.

Implicit coercion

Take a look at the following example:

In this case, the string variable `name` is coerced to true and you have 'Joey doesn't share food!' printed in our console. But how do you know what will be coerced to true and what will be coerced to false?

Falsy values are values that will be coerced to `false` when forced a boolean coercion on it.

Falsy values: `""`, `0`, `null`, `undefined`, `NaN`, `false`.

Anything not explicitly on the falsy list is truthy—**boolean coerced to true**.

Yes. You read it right. Empty arrays, objects and functions are boolean coerced to true!

String & Number coercion

The first thing you need to be aware of is the `+` operator. This is a tricky operator because it works for both number addition and string concatenation.

But, the *, / , and `-` operators are exclusive for numeric operations. When these operators are used with a string, it forces the string to be coerced to a number.

== vs. ===

It is widely spread that `==` checks for equality and `===` checks for equality and type. Well, that is a misconception.

In fact, == checks for **equality with coercion** and === checks for equality without coercion—**strict equality**.

Coercion can be tricky. Take a look at the following code:

What would you expect for the following comparison?
`console.log(a == b);` `(1)`

This comparison actually returns True. Why?
What really happens under the hood is that if you are comparing a `boolean` with something other than a `boolean`, JavaScript coerces that `boolean` to a `number` and compares. `(2)`

This comparison is now between a `number` and a `string`. JavaScript now coerces that `string` to a `number` and compares both numbers. `(3)`

In this case, the final comparison `0 == 0` is True.

```
'0' == false   (1)
'0' == 0       (2)
 0  == 0       (3)
```

For a fully comprehension on how such comparisons are performed, you can check ES5 documentation here.

For a cheat sheet, you can click here.

Some tricky comparisons to look out for:

Value vs. Reference

Simple values (also known as primitives) are always assigned by value-copy: `null`, `undefined`, `boolean`, `number`, `string` and ES6 `symbol`.

Compound values always create a copy of the reference on assignment: objects, which includes arrays, and functions.

To copy a compound value by value, you need to **make** a copy of it. The reference does not point to the original value.

Scope

Scope refers to the execution context. It defines the accessibility of variables and functions in the code.

Global Scope is the outermost scope. Variables declared outside a function are in the global scope and can be accessed in any other scope. In a browser, the window object is the global scope.

Local Scope is a scope nested inside another function scope. Variables declared in a local scope are accessible within this scope as well as in any inner scopes.

You may think of Scopes as a series of doors decreasing in size (from biggest to smallest). A short person that fits through the smallest door — **innermost scope** — also fits through any bigger doors — **outer scopes**.

A tall person that gets stuck on the third door, for example, will have access to all previous doors — **outer scopes** — but not any further doors — **inner scopes**.

Hoisting

The behavior of "moving" `var` and `function` declarations to the top of their respective scopes during the compilation phase is called **hoisting**.

Function declarations are completely hoisted. This means that a declared function can be called before it is defined.

Variables are partially hoisted. `var` declarations are hoisted but not its assignments.

`let` and `const` are not hoisted.

Function Expression vs. Function Declaration

> **Function Expression**
> A Function Expression is created when the execution reaches it and is usable from then on—it is not hoisted.
>
> **Function Declaration**
> A Function Declaration can be called both before and after it was defined—it is hoisted.

Variables: var, let and const

Before ES6, it was only possible to declare a variable using `var`. Variables and functions declared inside another function cannot be accessed by any of the enclosing scopes—they are function-scoped.

Variables declared inside a block-scope, such as `if` statements and `for` loops, can be accessed from outside of the opening and closing curly braces of the block.

Note: An undeclared variable—assignment without `var`, `let` or `const`—creates a `var` variable in global scope.

ES6 `let` and `const` are new. They are not hoisted and block-scoped alternatives for variable declaration. This means that a pair of curly braces define a scope in which variables declared with either let or const are confined in.

A common misconception is that `const` is immutable. It cannot be reassigned, but its properties can be **changed**!

Closure

A **closure** is the combination of a function and the lexical environment from which it was declared. Closure allows a function to access variables from an enclosing scope—**environment**—even after it leaves the scope in which it was declared.

The above example covers the two things you need to know about closures:

1. Refers to variables in outer scope.
   The returned function access the `message` variable from the enclosing scope.
2. It can refer to outer scope variables even after the outer function has returned.
   `sayHiToJon` is a reference to the `greeting` function, created when `sayHi` was run. The `greeting` function maintains a reference to its outer scope—**environment**—in which `message` exists.

One of the main benefits of closures is that it allows **data encapsulation**. This refers to the idea that some data should not be directly exposed. The following example illustrates that.

By the time `elementary` is created, the outer function has already returned. This means that the `staff` variable only exists inside the closure and it cannot be accessed otherwise.

Let's go deeper into closures by solving one of the most common interview problems on this subject:
What is wrong with the following code and how would you fix it?

Considering the above code, the console will display four identical messages `"The value undefined is at index: 4"`. This happens because each function executed within the loop will be executed after the whole loop has completed, referencing to the last value stored in `i`, which was 4.

This problem can be solved by using IIFE, which creates a unique scope for each iteration and storing each value within its scope.

Another solution would be declaring the `i` variable with `let`, which creates the same result.

Immediate Invoked Function Expression (IIFE)

An IIFE is a function expression that is called immediately after you define it. It is usually used when you want to create a new variable scope.

The **(surrounding parenthesis)** prevents from treating it as a function declaration.

The **final parenthesis()** are executing the function expression.

On IIFE you are calling the function exactly when you are defining it.

Using IIFE:

- Enables you to attach private data to a function.
- Creates fresh environments.
- Avoids polluting the global namespace.

Context

Context is often confused as the same thing as Scope. To clear things up, lets keep the following in mind:
Context is most often determined by how a function is invoked. It always refers to the value of `this` in a particular part of your code.
Scope refers to the visibility of variables.

Function calls: call, apply and bind

All of these three methods are used to attach `this` into function and the difference is in the function invocation.

`.call()` invokes the function immediately and requires you to pass in arguments as a list (one by one).

`.apply()` invokes the function immediately and allows you to pass in arguments as an array.

`.call()` and `.apply()` are mostly equivalent and are used to borrow a method from an object. Choosing which one to use depends on which one is easier to pass the arguments in. Just decide whether it's easier to pass in an array or a comma separated list of arguments.

Quick tip: Apply for **A**rray — **C**all for **C**omma.

Note: If you pass in an array as one of the arguments on a call function, it will treat that entire array as a single element.
ES6 allows us to spread an array as arguments with the call function.

```
char.knows.call(Snow, ...["nothing", "Jon"]);  // You know nothing, Jon Snow
```

`.bind()` returns a new function, with a certain context and parameters. It is usually used when you want a function to be called later with a certain context.

That is possible thanks to its ability to maintain a given context for calling the original function. This is useful for asynchronous callbacks and events.

`.bind()` works like the call function. It requires you to pass in the arguments one by one separated by a comma.

`'this'` keyword

Understanding the keyword `this` in JavaScript, and what it is referring to, can be quite complicated at times.

The value of `this` is usually determined by a functions execution context. Execution context simply means how a function is called.

The keyword `this` acts as a placeholder, and will refer to whichever object called that method when the method is actually used.

The following list is the ordered rules for determining this. Stop at the first one that applies:

> **`new` binding** — When using the `new` keyword to call a function, `this` is the newly constructed object.

Explicit binding—When call or apply are used to call a function, `this` is the object that is passed in as the argument.
Note: `.bind()` works a little bit differently. It creates a new function that will call the original one with the object that was bound to it.

Implicit binding—When a function is called with a context (the containing object), `this` is the object that the function is a property of.
This means that a function is being called as a method.

Default binding—If none of the above rules applies, `this` is the global object (in a browser, it's the window object).
This happens when a function is called as a standalone function.
A function that is not declared as a method automatically becomes a property of the global object.

Note: This also happens when a standalone function is called from within an outer function scope.

Lexical this—When a function is called with an arrow function `=>`, `this` receives the `this` value of its surrounding scope at the time it's created.
`this` keeps the value from its original context.

Strict Mode

JavaScript is executed in strict mode by using the `"use strict"` directive. Strict mode tightens the rules for parsing and error handling on your code.

Some of its benefits are:

- **Makes debugging easier**—Code errors that would otherwise have been ignored will now generate errors, such as assigning to non-writable global or property.
- **Prevents accidental global variables**—Assigning a value to an undeclared variable will now throw an error.
- **Prevents invalid use of delete**—Attempts to delete variables, functions and undeletable properties will now throw an error.

- **Prevents duplicate property names or parameter values** — Duplicated named property in an object or argument in a function will now throw an error. (This is no longer the case in ES6)
- **Makes eval() safer** — Variables and functions declared inside an `eval()` statement are not created in the surrounding scope.
- **"Secures" JavaScript eliminating this coercion** — Referencing a `this` value of null or undefined is not coerced to the global object. This means that in browsers it's no longer possible to reference the window object using `this` inside a function.

`new` keyword

The `new` keyword invokes a function in a special way. Functions invoked using the `new` keyword are called **constructor functions**.

So what does the `new` keyword actually do?

1. Creates a new object.
2. Sets the **object's** prototypeto be the prototypeof the **constructor function**.
3. Executes the constructor function with `this` as the newly created object.
4. Returns the created object. If the constructor returns an object, this object is returned.

What is the difference between invoking a function with the `new` keyword and without it?

Prototype and Inheritance

Prototype is one of the most confusing concepts in JavaScript and one of the reason for that is because there are two different contexts in which the word **prototype** is used.

- **Prototype relationship**
  Each object has a **prototype**object, from which it inherits all of its prototype's properties.
  `.__proto__` is a non-standard mechanism (available in ES6) for retrieving the prototypeof an object *(*)*. It points to the object's

"parent"—the **object's prototype**.
All normal objects also inherit a `.constructor` property that points to the constructor of the object. Whenever an object is created from a constructor function, the `.__proto__` property links that object to the `.prototype` property of the constructor function used to create it.
(*) *`Object.getPrototypeOf()`* is the standard ES5 function for retrieving the prototype of an object.

- **Prototype property**
  Every function has a `.prototype` property.
  It references to an object used to attach properties that will be inherited by objects further down the prototype chain. This object contains, by default, a `.constructor` property that points to the original constructor function.
  Every object created with a constructor function inherits a constructor property that points back to that function.

Prototype Chain

The prototype chain is a series of links between objects that reference one another.

When looking for a property in an object, JavaScript engine will first try to access that property on the object itself.

If it is not found, the JavaScript engine will look for that property on the object it inherited its properties from—the **object's prototype**.

The engine will traverse up the chain looking for that property and return the first one it finds.

The last object in the chain is the built-in `Object.prototype`, which has `null` as its **prototype**. Once the engine reaches this object, it returns `undefined`.

Own vs Inherited Properties

Objects have own properties and inherited properties.

Own properties are properties that were defined on the object.

Inherited properties were inherited through prototype chain.

Object.create(*obj*) — Creates a new object with the specified **prototype**object and properties.

Inheritance by reference

An inherited property is a copy by reference of the **prototype object's**property from which it inherited that property.

If an object's property is mutated on the prototype, objects which inherited that property will share the same mutation. But if the property is replaced, the change will not be shared.

Classical Inheritance vs. Prototypal Inheritance

In classical inheritance, objects inherit from classes — like a blueprint or a description of the object to be created — and create sub-class relationships. These objects are created via constructor functions using the new keyword.

The downside of classical inheritance is that it causes:
inflexible hierarchy
tight coupling problems
fragile base class problems
duplication problems
And the so famous gorilla/banana problem — *"What you wanted was a banana, what you got was a gorilla holding the banana, and the entire jungle."*

In prototypal inheritance, objects inherit directly from other objects. Objects are typically created via `Object.create()`, object literals or factory functions.

There are three different kinds of prototypal inheritance:

- **Prototype delegation** — A delegate prototype is an object which is used as a model for another object. When you inherit from a delegate prototype, the new object gets a reference to the prototype and its properties.

This process is usually accomplished by using `Object.create()`.

- **Concatenative inheritance** — The process of inheriting properties from one object to another by copying the object's prototype properties, without retaining a reference between them.
  This process is usually accomplished by using `Object.assign()`.
- **Functional inheritance** — This process makes use of a *factory function(*)* to create an object, and then adds new properties directly to the created object.
  This process has the benefit of allowing data encapsulation via closure.
  (*)Factory function is a function that is not a class or constructor that returns an object without using the `new` keyword.

You can find a complete article on this topic by Eric Elliott here.

Favor composition over class inheritance

Many developers agree that class inheritance should be avoided in most cases. In this pattern you design your types regarding what they **are**, which makes it a very strict pattern.

Composition, on the other hand, you design your types regarding what they **do**, which makes it more flexible and reusable.

Here is a nice video on this topic by Mattias Petter Johansson

Asynchronous JavaScript

JavaScript is a single-threaded programming language. This means that the JavaScript engine can only process a piece of code at a time. One of its main consequences is that when JavaScript encounters a piece of code that takes a long time to process, it will block all code after that from running.

JavaScript uses a data structure that stores information about active functions named **Call Stack**. A Call Stack is like a pile of books. Every book that goes into that pile sits on top of the previous book. The last book to go into the pile will be the first one removed from it, and the first book added to the pile will be the last one removed.

The solution to executing heavy pieces of code without blocking anything is **asynchronous callback functions**. These functions are executed later —**asynchronously**.

The asynchronous process begins with an asynchronous callback functions placed into a **Heap or** region of memory. You can think of the Heap as an **Event Manager**. The Call Stack asks the Event Manager to execute a specific function only when a certain event happens. Once that event happens, the Event Manager moves the function to the Callback Queue. **Note**: When the Event Manager handles a function, the code after that is not blocked and JavaScript continues its execution.

The Event Loop handles the execution of multiple pieces of your code over time. The Event Loop monitors the Call Stack and the Callback Queue.

The Call Stack is constantly checked whether it is empty or not. When it is empty, the Callback Queue is checked if there is a function waiting to be invoked. When there is a function waiting, the first function in the queue is pushed into the Call Stack, which will run it. This checking process is called a 'tick' in the Event Loop.

Let's break down the execution of the following code to understand how this process works:

1. Initially the Browser console is clear and the Call Stack and Event Manager are empty.
2. `first()` is added to the Call Stack.
3. `console.log("First message")` is added to the Call Stack.
4. `console.log("First message")` is executed and the Browser console displays **"First message"**.
5. `console.log("First message")` is removed from the Call Stack.
6. `first()` is removed from the Call Stack.
7. `setTimeout(second, 0)` is added to the Call Stack.
8. `setTimeout(second, 0)` is executed and handled by the Event Manager. And after 0ms the Event Manager moves `second()` to the Callback Queue.

9. `setTimeout(second, 0)` is now completed and removed from the Call Stack.
10. `third()` is added to the Call Stack.
11. `console.log("Third message")` is added to the Call Stack.
12. `console.log("Third message")` is executed and the Browser console displays **“Third message”**.
13. `console.log("Third message")` is removed from the Call Stack.
14. `third()` is removed from the Call Stack.
15. Call Stack is now empty and the `second()` function is waiting to be invoked in the Callback Queue.
16. The Event Loop moves `second()` from the Callback Queue to the Call Stack.
17. `console.log("Second message")` is added to the Call Stack.
18. `console.log("Second message")` is executed and the Browser console displays **“Second message”**.
19. `console.log("Second message")` is removed from the Call Stack.
20. `second()` is removed from the Call Stack.

Note: The `second()` function is not executed after 0ms. The **time** you pass in to `setTimeout` function does not relate to the delay of its execution. The Event Manager will wait the given timebefore moving that function into the Callback Queue. Its execution will only take place on a future ‘tick’ in the Event Loop.

Thanks and congratulations for reading up to this point! If you have any thoughts on this, feel free to leave a comment.