**1. Is JavaScript Interpreted Language in its entirety? Check this Link and Make Up your justification.**

The short answer is no. JavaScript is not an interpreted language in its entirety.

It was designed in by Brendan Eich and introduced in the Netscape browser in 1995 with the goal of adding interactivity to web pages. Its purpose was to work inside the browser and provide API to manipulate the DOM. Performance wasn't the first concern. It was fully interpreted - read and executed line by line by an interpreter without needing it to be compiled to machine language.

But it was slow and needed improvement for different reasons, of which, one was the fact that it was unoptimized unlike compiled languages. It was redundant and inefficient. For example, when a loop containing a function call that invokes a calculation and returns a result is ran, it would read the code line by line, interpreting and executing each and every iteration even if it was the same function call with the same arguments, unlike complied languages where there'd be complier optimizations. For this case, one simple optimization could be replacing the function call with its return value, avoiding going to the function every time. Another issue was error handling. When a code is being executed in a line-by-line basis, say line 6 will executed before line 7 is read, which could potentially be a line with an exception and so this makes every execution a possible failure, a doomed partial execution, which might be undesirable. Additionally, since JavaScript was dynamically typed the type checking and computation made it very slow - iterating over an Array would be an example in point here showing, considering how arrays can consist of multiple data types. All this is not mentioning how it's relatively an English like, simple to understand, human-friendly, language, which contributed to making it interpreted language **at the time/in the first plac**e and widened the game between it and what a computer can understand. **But despite all that it is currently one of the most fast and efficient languages.**

Due to the resulting slowness of JavaScript, Google was having a hard time providing a Google Maps that was highly interactive, smooth and dynamic. And so they come up with better approaches and build the modern JavaScript engine V8 in 2008 and had made huge progress since then.

Today, the modern V8 engine and other browser engine with similar mechanisms run our JavaScript codes and they had improved running efficiency a whole lot. It uses lots heuristics (essentially self-teaching, data and result based) and optimizations to fasten up the process. Here are the steps from parsing () to execution:

1. The script is 'parsed' into tokens that are like the units or alphabets of the language and are used to make an Abstract Syntax Tree (AST) of the code, which is a tree-like structure representing the code and their relationships. Here the parser makes optimization choice like not parse functions that aren't called and will only parse them when they're called like with a button click.
2. The engine interpreter (V8 Ignition Interpreter for Chrome) interprets the AST to bytecode, which is a bit more abstract form of the low-level machine code. One optimization that's done here is the using of only one shared copy of the same meta-data/object properties of multiple instances to avoid wasting memory with duplicated information. Here a component called the profile monitors the codes that runs and watches if a certain snippet of code is run more than a certain number of times. Then it passes it to the 'optimizing' compiler which then tries to form an optimized machine code of it so as to save it and easily run it when needed without going through the whole process again. But that is assuming things like the type of a variable is not changed. The engine also has an cache where such repeated bytecodes and locations of objects in the bytecode are stored for the same purpose of make the process more fast and efficient.
3. The compile (called Turbofan) takes in the bytecode and changes it machine code. When using those saved code snippers, the compiler checks if the assumptions it made were true and if not the engine will go back to using the bytecode being generated. Finally, the JavaScript Virtual Machine executes the program.

These sets of optimizing techniques are called Just-in-Time compiling (JIT). And the steps make it clear that JavaScript engines use compilers and thus JavaScript is not a fully interpreted language. It would seem more of a hybrid - as there can be different implementations (like one that mostly use interpretation) - but leaning to compiled language as interpretated language hardly touches the half of what JavaScript is and how it shares similar execution process with Java and how it reports static (compile-time) errors (before execution).

**2. The history of “typeof null”.**

The "typeof" \_\_\_ classifies primitive data types and tells them apart from objects. The table below shows its output for different operands.

|  |  |
| --- | --- |
| **Operand** | **Result** |
| undefined | "undefined" |
| null | "object" |
| Boolean value | "boolean" |
| Number value | "number" |
| String value | "string" |
| Function | "function" |
| All other values | "object" |

The result for "null" is a bug. It is a primitive data type, not an object. This bug traces its roots the first version of JavaScript when Brendan Eich had very little time to write the source code. He wrote it in 10 days.

He was writing in C and wrote typeof to check every of the above possible data types except the null. Values used to be stored in 32-bit units, 1-3 of which were used to tell the data type. They were 1, 000, 010, 100, and 110. The 000 reference was for an object. But here’s the catch, the null was an object type tag with a reference of zero. Because of this the typeof saw the type tag of null and read it as an object.

**3. Explain in detail why hoisting is different with let and const?**

Hoisting is the when variable and function declarations are put into memory during the compilation phase. With var for example, the interpreter scans the entire code and puts its declaration in the memory (in the VO (Variable Object) to be exact). And after it does so, it will initialize it with a value of undefined, and so if you used a var before its declaration it won't throw an error. Instead, it will work normally with the value undefined.

What changes with let and const is that they won't be initialized with undefined or anything like var. Take the following example:

**console.log(x); //throws an error**

**let x;**

**console.log(x); //gives undefined**

**x = 1;**

**console.log(x);//gives 1**

The interpreter will first put the declaration of x to memory without initializing it and before going to the execution phase and starts executing each line. And it will throw a throw an error while running the first log as x isn't initialized. In the second line it would initialize x with undefined as it will do with any other uninitialized variable and that's what it will output in the second log. And finally, the third log would give 1 since it's been initialized in the previous line. I should note that the same can't be done with const as it must be initialized while it's declared. And so there won't be the middle log.

The period between line 1 and the initialization line above is called the temporal dead zone (TDZ), owing to the fact that the let variable can't be used in that span. The TDZ is about the time and not the order of the code lines because not their physical ordering but execution ordering that defines it. The code below depicts that.

**function func () {console.log(x)}; //no error**

**let x = 1;**

**func();**

Since the log line is executed after func is invoked and therefore after the let variable is initialized, there won't be any error thrown - the order of execution doesn't cause that so.