<https://www.toptal.com/javascript/interview-questions>

What is the significance, and what are the benefits, of including 'use strict' at the beginning of a JavaScript source file?

The short and most important answer here is that use strict is a way to voluntarily enforce stricter parsing and error handling on your JavaScript code at runtime. Code errors that would otherwise have been ignored or would have failed silently will now generate errors or throw exceptions. In general, it is a good practice.

Some of the key benefits of strict mode include:

* **Makes debugging easier.** Code errors that would otherwise have been ignored or would have failed silently will now generate errors or throw exceptions, alerting you sooner to problems in your code and directing you more quickly to their source.
* **Prevents accidental globals.** Without strict mode, assigning a value to an undeclared variable automatically creates a global variable with that name. This is one of the most common errors in JavaScript. In strict mode, attempting to do so throws an error.
* **Eliminates this coercion**. Without strict mode, a reference to a this value of null or undefined is automatically coerced to the global. This can cause many headfakes and pull-out-your-hair kind of bugs. In strict mode, referencing a a this value of null or undefined throws an error.
* **Disallows duplicate property names or parameter values.** Strict mode throws an error when it detects a duplicate named property in an object (e.g., var object = {foo: "bar", foo: "baz"};) or a duplicate named argument for a function (e.g., function foo(val1, val2, val1){}), thereby catching what is almost certainly a bug in your code that you might otherwise have wasted lots of time tracking down.
* **Makes eval() safer.** There are some differences in the way eval() behaves in strict mode and in non-strict mode. Most significantly, in strict mode, variables and functions declared inside of an eval() statement are *not* created in the containing scope (they *are* created in the containing scope in non-strict mode, which can also be a common source of problems).
* **Throws error on invalid usage of delete.** The delete operator (used to remove properties from objects) cannot be used on non-configurable properties of the object. Non-strict code will fail silently when an attempt is made to delete a non-configurable property, whereas strict mode will throw an error in such a case.

What is the significance of, and reason for, wrapping the entire content of a JavaScript source file in a function block?

This is an increasingly common practice, employed by many popular JavaScript libraries (jQuery, Node.js, etc.). This technique creates a closure around the entire contents of the file which, perhaps most importantly, creates a private namespace and thereby helps avoid potential name clashes between different JavaScript modules and libraries.

Another feature of this technique is to allow for an easily referenceable (presumably shorter) alias for a global variable. This is often used, for example, in jQuery plugins. jQuery allows you to disable the $ reference to the jQuery namespace, using jQuery.noConflict(). If this has been done, your code can still use $ employing this closure technique, as follows:

(function($) { /\* jQuery plugin code referencing $ \*/ } )(jQuery);

What is NaN? What is its type? How can you reliably test if a value is equal to NaN?

The NaN property represents a value that is “not a number”. This special value results from an operation that could not be performed either because one of the operands was non-numeric (e.g., "abc" / 4), or because the result of the operation is non-numeric (e.g., an attempt to divide by zero).

While this seems straightforward enough, there are a couple of somewhat surprising characteristics of NaN that can result in hair-pulling bugs if one is not aware of them.

For one thing, although NaN means “not a number”, its type is, believe it or not, Number:

console.log(typeof NaN === "number"); // logs "true"

Additionally, NaN compared to anything – even itself! – is false:

console.log(NaN === NaN); // logs "false"

A *semi-reliable* way to test whether a number is equal to NaN is with the built-in function isNaN(), but even using [isNaN() is an imperfect solution](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/isNaN" \l "Confusing_special-case_behavior" \t "_blank).

A better solution would either be to use value !== value, which would *only* produce true if the value is equal to NaN. Also, ES6 offers a new [Number.isNaN()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/isNaN" \t "_blank) function, which is a different and more reliable than the old global isNaN() function.

Consider the following code snippet:

for (var i = 0; i < 5; i++) {

var btn = document.createElement('button');

btn.appendChild(document.createTextNode('Button ' + i));

btn.addEventListener('click', function(){ console.log(i); });

document.body.appendChild(btn);

}

(a) What gets logged to the console when the user clicks on “Button 4” and why?

(b) Provide one or more alternate implementations that will work as expected.

(a) No matter what button the user clicks the number 5 will *always* be logged to the console. This is because, at the point that the onclick method is invoked (for *any* of the buttons), the for loop has already completed and the variable i already has a value of 5. (Bonus points for the interviewee if they know enough to talk about how execution contexts, variable objects, activation objects, and the internal “scope” property contribute to the closure behavior.)

(b) The key to making this work is to capture the value of i at each pass through the for loop by passing it into a newly created function object. Here are three possible ways to accomplish this:

for (var i = 0; i < 5; i++) {

var btn = document.createElement('button');

btn.appendChild(document.createTextNode('Button ' + i));

btn.addEventListener('click', (function(i) {

return function() { console.log(i); };

})(i));

document.body.appendChild(btn);

}

What is a “closure” in JavaScript? Provide an example.

A closure is an inner function that has access to the variables in the outer (enclosing) function’s scope chain. The closure has access to variables in three scopes; specifically: (1) variable in its own scope, (2) variables in the enclosing function’s scope, and (3) global variables.

Here is a simple example:

var globalVar = "xyz";

(function outerFunc(outerArg) {

var outerVar = 'a';

(function innerFunc(innerArg) {

var innerVar = 'b';

console.log(

"outerArg = " + outerArg + "\n" +

"innerArg = " + innerArg + "\n" +

"outerVar = " + outerVar + "\n" +

"innerVar = " + innerVar + "\n" +

"globalVar = " + globalVar);

})(456);

})(123);

In the above example, variables from innerFunc, outerFunc, and the global namespace are **all** in scope in the innerFunc. The above code will therefore produce the following output:

outerArg = 123

innerArg = 456

outerVar = a

innerVar = b

globalVar = xyz

JSON Values VS JS Object

https://www.w3schools.com/js/js\_json\_syntax.asp

In JSON, keys must be strings, written with double quotes:

### JSON

{ "name":"John" }

In JavaScript, keys can be strings, numbers, or identifier names:

### JavaScript

{ name:"John" }

In JSON, *values* must be one of the following data types:

* a string
* a number
* an object (JSON object)
* an array
* a boolean
* null

In JavaScript values can be all of the above, plus any other valid JavaScript expression, including:

* a function
* a date
* undefined

In JSON, string values must be written with double quotes:

JSON: { "name":"John" }

In JavaScript, you can write string values with double or single quotes:

JavaScript: { name:'John' }

# AJAX

var xhttp = new XMLHttpRequest();

|  |  |
| --- | --- |
| readyState | Holds the status of the XMLHttpRequest. 0: request not initialized  1: server connection established 2: request received  3: processing request  4: request finished and response is ready |

# Plugin, library, API and framework

jQuery is a JS library / framework. jQuery functionality can be extended by jQuery plugins. These plugins wouldn't work without jQuery as they use its functionality.

A framework is a standardized set of concepts, practices and criteria for dealing with a common type of problem, which can be used as a reference to help us approach and resolve new problems of a similar nature.   
 The aim of frameworks is to provide a common structure so that developers don’t have to redo it from scratch and can reuse the code provided.   
it can be backend and frontend frameworks.

Framework is like a furnished house with all the basic/commonly used items. Though the house is furnished, there is always a need for something more based on individual needs. So you bring in new items to suite your requirement - these are plugins.   
  
APIs are the interfaces that define how you are going to use those items. For example, if we take a washing machine, all the knobs that you see are its APIs. They basically let you use an item without having to worry about its internal details.