

DWA_02.8 Knowledge Check_DWA2

1. What do ES5, ES6 and ES2015 mean - and what are the differences between them?

ES5, ES6 and ES2015 are all different versions of ECMAScript specifications. ECMAScript is the standard upon which JavaScript is based.

ES5 was released in Dec 2009 and it introduced strict mode. It also added new methods to native objects like for example: `Array.prototype.forEach`, and `Object.keys`. It also enhanced support for JSON.

ECMAScript 5 was wholly supported by Firefox 4 (2011), Chrome 19 (2012), Safari 6 (2012), Opera 12.10 (2012) and Internet Explorer 10 (2012).

ES5 updates included:

- Getter/setters
- Trailing commas in array and object literals
- Reserved words as property names
- New **Object** methods (`create`, `defineProperty`, `keys`, `seal`, `freeze`, `getOwnPropertyNames`, etc.)
- New **Array** methods (`isArray`, `indexOf`, `every`, `some`, `map`, `filter`, `reduce`, etc.)
- `String.prototype.trim` and property access
- New **Date** methods (`toISOString`, `now`, `toJSON`)
- Function `bind`
- JSON
- Immutable global objects (`undefined`, `NaN`, `Infinity`)
- Strict mode
- Other minor changes (`parseInt` ignores leading zeroes, thrown functions have proper `this` values, etc.)

ES6 was released in Jun 2015. ES6 is also referred to as ES2015 because it was initially planned to release a new version of ECMAScript yearly. Thus followed the yearly release cycle.

The most significant features of ES6 were the introduction of “let” and “const” declarations. The addition of arrow functions for concise function syntax. Template literals for string interpolation. Classes and modules to make OOP convenient. The

destructuring assignment and spread/rest operators were also introduced. Enhancement of the handling of promises for async programming. And the for ... of loop.

ES6/2015 features:

- Let (lexical) and const (unrebindable) bindings
- Arrow functions (shorter anonymous functions) and lexical this (enclosing scope this)
- Classes (syntactic sugar on top of prototypes)
- Object literal improvements (computed keys, shorter method definitions, etc.)
- Template strings
- Promises
- Generators, iterables, iterators and for..of
- Default arguments for functions and the rest operator
- Spread syntax
- Destructuring
- Module syntax
- New collections (Set, Map, WeakSet, WeakMap)
- Proxies and Reflection
- Symbols
- Typed arrays
- Support for subclassing built-ins
- Guaranteed tail-call optimization
- Simpler Unicode support
- Binary and octal literals

2. What are JScript, ActionScript and ECMAScript - and how do they relate to JavaScript?

JScript is Microsoft's implementation of ECMAScript. It's a scripting language developed by Microsoft for use in Internet Explorer.

JScript and JavaScript share similarities. JScript has some differences to JavaScript in terms of features and behavior. JScript is associated with older versions of Internet Explorer whereas JavaScript has become the dominant implementation of ECMAScript.

ActionScript is a scripting language used for the development of Adobe Flash content. It's an object oriented language based on ECMAScript 4.

ActionScript and JavaScript share a common ancestry, as they both derive from ECMAScript. They were developed independently for different purposes. ActionScript was designed for creating interactive content within the Adobe Flash platform. With the decline of Flash and rise of standards, like HTML5 and JavaScript, ActionScript became less prominent.

ECMAScript is a scripting language specification that serves as the foundation for several scripting languages. It defines the core features and functionality that a scripting language should provide.

JavaScript is the most well-known implementation of ECMAScript. JavaScript follows the standard of ECMAScript. JavaScript is an implementation of ECMAScript specifications, with additional features and behavior.

3. What is an example of a JavaScript specification - and where can you find it?

A JavaScript specification is officially known as the ECMAScript specification.

To find the ECMAScript specification, I can visit their website(<https://tc39.es/ecma262/>) or go to ECMAScript's github repository(<https://github.com/tc39/ecma262>).

Here is an example of ECMAScript's specification for NaN -

6.1.6.1 The Number Type

The Number type has exactly 18,437,736,874,454,810,627 (that is, $2^{64} - 2^{53} + 3$) values, representing the double-precision 64-bit format IEEE 754-2019 values as specified in the IEEE Standard for Binary Floating-Point Arithmetic, except that the 9,007,199,254,740,990 (that is, $2^{53} - 2$) distinct "Not-a-Number" values of the IEEE Standard are represented in ECMAScript as a single special NaN value. (Note that the NaN value is produced by the program expression NaN.) In some implementations, external code might be able to detect a difference between various Not-a-Number values, but such behaviour is implementation-defined; to ECMAScript code, all NaN values are indistinguishable from each other.

NOTE

The bit pattern that might be observed in an ArrayBuffer (see 25.1) or a SharedArrayBuffer (see 25.2) after a Number value has been stored into it is not necessarily the same as the internal representation of that Number value used by the ECMAScript implementation.

There are two other special values, called **positive Infinity** and **negative Infinity**. For brevity, these values are also referred to for expository purposes by the symbols $+\infty_p$ and $-\infty_p$, respectively. (Note that these two infinite Number values are produced by the program expressions **+Infinity** (or simply **Infinity**) and **-Infinity**.)

The other 18,437,736,874,454,810,624 (that is, $2^{64} - 2^{53}$) values are called the finite numbers. Half of these are positive numbers and half are negative numbers; for every finite positive Number value there is a corresponding negative value having the same magnitude.

Note that there is both a **positive zero** and a **negative zero**. For brevity, these values are also referred to for expository purposes by the symbols **+0_F** and **-0_F**, respectively. (Note that these two different zero Number values are produced by the program expressions **+0** (or simply **0**) and **-0**.)

The 18,437,736,874,454,810,622 (that is, $2^{-64} - 2^{-53} - 2$) **finite** non-zero values are of two kinds:

18,428,729,675,200,069,632 (that is, $2^{-64} - 2^{-54}$) of them are normalized, having the form

$$s \times m \times 2^{-e}$$

where s is 1 or -1, m is an **integer** in the **interval** from 2^{-52} (inclusive) to 2^{-53} (exclusive), and e is an **integer** in the **inclusive interval** from -1074 to 971.

The remaining 9,007,199,254,740,990 (that is, $2^{-53} - 2$) values are denormalized, having the form

$$s \times m \times 2^{-e}$$

where s is 1 or -1, m is an **integer** in the **interval** from 0 (exclusive) to 2^{-52} (exclusive), and e is -1074.

Note that all the positive and negative **integers** whose magnitude is no greater than 2^{-53} are representable in the Number type. The **integer** 0 has two representations in the Number type: **+0_F** and **-0_F**.

A **finite** number has an *odd significand* if it is non-zero and the **integer** m used to express it (in one of the two forms shown above) is odd. Otherwise, it has an *even significand*.

In this specification, the phrase “the Number value for x ” where x represents an exact real mathematical quantity (which might even be an irrational number such as π) means a Number value chosen in the following manner. Consider the set of all **finite** values of the Number type, with **-0_F** removed and with two additional values added to it that are not representable in the Number type, namely 2^{-1024} (which is $+1 \times 2^{-53} \times 2^{-971}$) and -2^{-1024} (which is $-1 \times 2^{-53} \times 2^{-971}$). Choose the member of this set that is closest in value to x . If two values of the set are equally close, then the one with an even significand is chosen; for this purpose, the two extra values 2^{-1024} and -2^{-1024} are considered to have even significands. Finally, if 2^{-1024} was chosen, replace it with $+\infty_F$; if -2^{-1024} was chosen, replace it with $-\infty_F$; if **+0_F** was chosen, replace it with **-0_F** if and only if $x < 0$; any other chosen value is used unchanged. The result is the **Number value for x** . (This procedure corresponds exactly to the behaviour of the [IEEE 754-2019](#) roundTiesToEven mode.)

The **Number value for** $+\infty$ is $+\infty_F$, and the **Number value for** $-\infty$ is $-\infty_F$.

Some ECMAScript operators deal only with **integers** in specific ranges such as the **inclusive interval** from -2^{31} to $2^{31} - 1$ or the **inclusive interval** from 0 to $2^{16} - 1$. These operators accept any value of the Number type but first convert each such value to an **integer** value in the expected range. See the descriptions of the numeric conversion operations in [7.1](#).

4. What are v8, SpiderMonkey, Chakra and Tamarin? Do they run JavaScript differently?

V8, SpiderMonkey, Chakra and Tamarin are all JavaScript engines, which are responsible for interpreting and executing JS code in web browsers. Each engine has its own implementation of ECMAScript specifications and has differences in performance, optimization and features.

V8 is an adopter of JIT compilation, which compiles JS code into native machine code for improved performance. It has high performance and efficiency.

SpiderMonkey powers the Firefox browser and has gone through various optimizations.

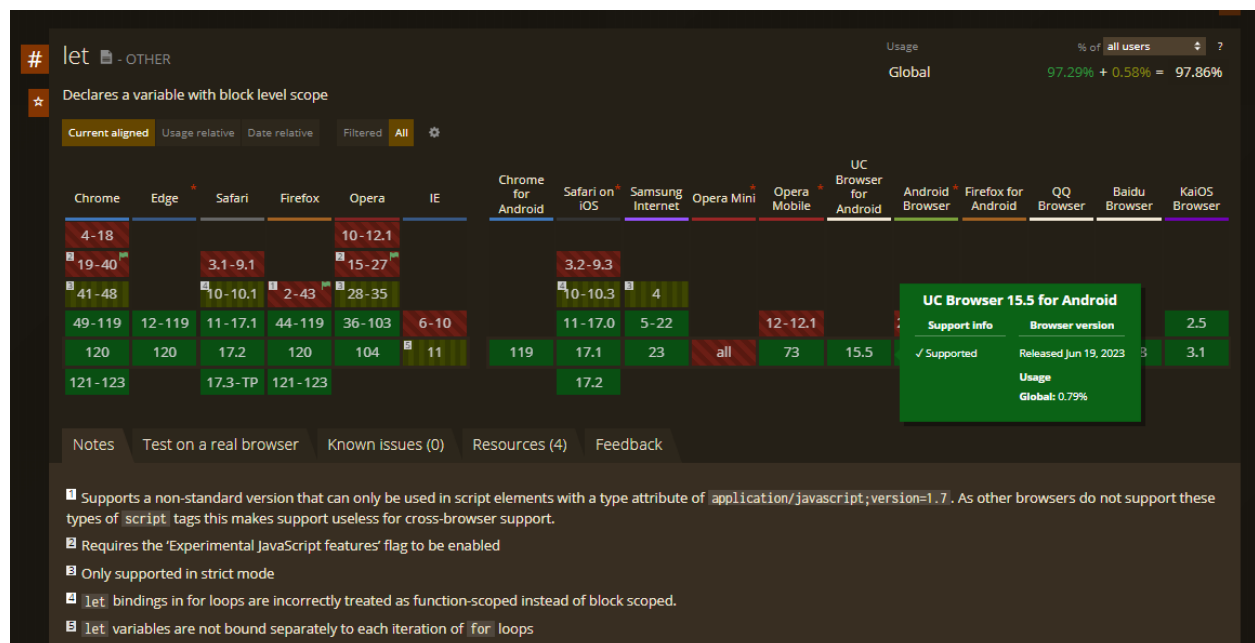
Chakra is supported by the ChakraCore project which allows embedding the engine in various applications as well as the browser. It features profile-guided optimization for improved performance.

Tamarin was designed for executing ActionScript within Adobe Flash platform.


There are variations in how they handle optimization, memory management and implementation details. They execute JS code differently due to variations in their implementation strategies, optimization techniques and design.

















Factors that contribute to differences are: Just-in-time(JIT) compilation, optimizations, garbage collection, concurrency and threading, feature support, platform integration, and evolution and updates.

5. Show a practical example using caniuse.com and the MDN compatibility table.




Browser compatibility

[Report problems with this compatibility data on GitHub](#) 

													
	 Chrome	 Edge	 Firefox	 Opera	 Safari	 Chrome Android	 Firefox for Android	 Opera Android	 Safari on iOS	 Samsung Internet	 WebView Android	 Deno	 Node.js
<code>let</code>	✓ 49 ...	✓ 14 ...	✓ 44 *	✓	✓ 10	✓ 49 ...	✓ 44 *	✓ 18	✓ 10	✓ 5.0 ...	✓ 49 ...	✓ 1.0	✓ 6.0.0

Tip: you can click/tap on a cell for more information.

✓ Full support  Partial support * See implementation notes. ... Has more compatibility info.