Why does "👩🏾‍🌾" have a length of 7 in JavaScript?

by Evan Hahn, posted May 27, 2023

In short: 👩🏾‍🌾 is made of 1 grapheme cluster, 4 scalars, and 7 UTF-16 code units. That's why its length is 7.

The `length` property is used to determine the length of a JavaScript string. Sometimes, its results are intuitive:

```
"E".length;
// => 1

"♬".length;
// => 1
```

…sometimes, its results are surprising:

```
"🌸".length;
// => 2

"👩🏾‍🌾".length;
// => 7
```

To understand why this happens, you need to understand a few terms from the Unicode glossary.

The first term is the **extended grapheme cluster**. This is probably what most people would call a character. E, ♬, 🌸, and 👩🏾‍🌾 are examples of extended grapheme clusters.

Extended grapheme clusters are made up of **scalars**. Scalars are integers between `0` and `1114111`, though many of these numbers are currently unused.

Many extended grapheme clusters contain just one scalar. For example, 🌸 is made up of the scalar `127800` and E is made up of scalar `69`. 👩🏾‍🌾, however, is made up of *four* scalars: `128105`, `127998`, `8205`, and `127806`.

(Scalars are usually written in hex with a "U+" prefix. For example, the scalar for ♬ is `9836`, which might be written as "U+266C".)

Internally, JavaScript stores these scalars as **UTF-16 code units**. Each code unit is a 16-bit unsigned integer, which can store anything between 0 and 65,535. Many scalars fit into a single code unit. Scalars that are too big get split apart into two 16-bit numbers. These are called **surrogate pairs**, which is a term you might see.

For example, ♬ is made up of the scalar 9836. That fits into a single 16-bit integer, so we just store 9836.

The scalar for 🌸 is 127800. That's too big for a 16-bit integer so we have to break it up. It gets split up into 55356 and 57144. (I won't discuss *how* this splitting works, but it's not too complicated—the bits are divided in the middle and a different number is added to each half.)

That's why `"🌸".length === 2`—JavaScript is interrogating the number of UTF-16 code units, which is 2 in this case.

👩🏾‍🌾 is made up of four scalars. One of those scalars fits in a single UTF-16 code unit, but the remaining three are too big and get split up. That makes for a total of 7 code units. That's why `"👩🏾‍🌾".length === 7`.

To summarize our examples:

Extended grapheme cluster	Scalar(s)	UTF-16 code units
E	69	69

Extended grapheme cluster	Scalar(s)	UTF-16 code units
🎵	9836	9836
🌸	127800	55356, 57144
👩🏾‍🌾	128105, 127998, 8205, 127806	55357, 56425, 55356, 57342, 8205, 55356, 57150

Most JavaScript string operations also work with UTF-16.

`slice()`, for example, works with UTF-16 code units too. That's why you might get strange results if you slice in the middle of a surrogate pair:

```
"The best character is X".slice(-1);
// => "X"

"The best character is 🌸".slice(-1);
// => "\udf38"
```

However, not all JavaScript string operations use UTF-16 code units. For example, iterating over a string works a little differently:

```
// The spread operator uses an iterator:
[..."👩🏾‍🌾"];
// => ["👩","🏾","","🌾"]
```

```
// Same for `for ... of`:
for (const c of "👩‍🌾") {
  console.log(c);
}
// => "👩"
// => "‍"
// => ""
// => "🌾"
```

As you can see, this iterates over scalars, not UTF-16 code units.

`Intl.Segmenter()`, an object that doesn't work in all browsers, can help you iterate over extended grapheme clusters if that's what you need:

```
const str = "farmer: 👩‍🌾";

// Warning: this is not supported on all browsers!
const segments = new Intl.Segmenter().segment(str);
[...segments];
// => [
//      { segment: "f", index: 0, input: "farmer: 👩‍🌾" },
//      { segment: "a", index: 1, input: "farmer: 👩‍🌾" },
//      { segment: "r", index: 2, input: "farmer: 👩‍🌾" },
//      { segment: "m", index: 3, input: "farmer: 👩‍🌾" },
//      { segment: "e", index: 4, input: "farmer: 👩‍🌾" },
//      { segment: "r", index: 5, input: "farmer: 👩‍🌾" },
//      { segment: ":", index: 6, input: "farmer: 👩‍🌾" },
//      { segment: " ", index: 7, input: "farmer: 👩‍🌾" },
//      { segment: "👩‍🌾", index: 8, input: "farmer: 👩‍🌾" }
//    ]
```

For more on this tricky stuff, check out "It's Not Wrong that "🤦‍♂️".length == 7" and "JavaScript has a Unicode problem".