When is an é not an é? Working with Unicode on the web

There are times when working with non English languages on the web can be a real problem. We'll look at what's the historical background for the issue, how different encodings have tried to solve it, and how Unicode addresses the problem.

The web, and the larger Internet, started in the US in English and, as such needed, only 7 bit characters to represent all the characters needed for the English alphabet. This character set was known as ASCII (American Standard Code for Information Interchange).

While the web was primarily an American endeavor written in English everything was finem but as the web became international the 7-bit ASCII character set was no longer enough to represent the characters in non-English languages.

That led to the introduction of the ISO 8859 family of 8 bit character sets and encodings, each representing a group of related languages or dialects.

The different language groups needed their own encoding to display characters in supporting fonts appropriately. It worked well but there was no way to mix and match encodings to display them together on a web page.

Also to consider is that two encodings in the ISO-8859 family could use the same codepoint (number for a given character) for two different characters so it would lead to all sorts of confusion and wrong characters appearing on the page.

And that brings us to Unicode.

Rather than work on specific characters, Unicode provides a unique identifier (called a code point) to each character from most of the world's languages. The only limiting factor is the device having fonts capable to display the character.

Unicode is divided in 16 planes and each plane contains one or more blocks, with each block representing a language or dialect.

But, as good as Unicode is working with multiple languages and character sets in the same document, it’s not free of issues and pitfalls.

Chief among these issues that we need to keep in mind is that there is more than one version of Unicode, UTF-8 and UTF-16 (we’ll ignore UTF-32 for now as it’s not supported in browsers and developers are discouraged from using it).

UTF-8, what most of us use when thinking about Unicode, uses between 1 and 4 bytes to represent all characters it supports. It’s a superset of ASCII, so the first 128 characters are identical to those in the ASCII table.

UTF-16 (which Javascript uses to represent characters) that uses either 2 or 4 bytes. This difference in how many bytes an element takes makes the two encodings incompatible.

But even if we stick to UTF-8 eclusively, there’s more than one way to represent a character. For example, the letter é could be represented using either:

- A single code point U+00E9
- The combination of the letter e and the acute accent, for a total of two code points: U+0065 and U+0301

While we get the same acute accent e character, they are not the same in terms of equality or in terms of character length as demonstrated below:

```
console.log('\u00e9') // => é
console.log('\u0065\u0301') '\u0065\u0301'// => é\u00e9' == '\u0065\u0301
co' == '// => false
console.log('\u00e9'.length) // => 1
console.log('\u0065\u0301'.length) // => 2
```

Same thing happens to characters with accents or other diacritical marks.

- n + ˜ = ñ
- u + ¨ = ü

You may be wondering why is this important. When you’re doing string matching and measurements (is this password 6 character or longer) then some ways to represent characters will give false positives and retunr a value that is either larger or shorter than the actual string.

So, how do we solve the problem?

Since ES2015/ES6 there is a method in the string object called normalize that takes the following signature: `String.prototype.normalize([form])` The form argument is the string identifier of the normalization form to use; it can take one of four values:

- **NFC** — Normalization Form Canonical Composition. This is the default is no form is provided
- **NFD** — Normalization Form Canonical Decomposition.
- **NFKC** — Normalization Form Compatibility Composition.
- **NFKD** — Normalization Form Compatibility Decomposition.

Rather than bore you with the details of what each of the forms mean and how do they work, I'll refer you to the Unicode Annex #15: Unicode Normalization forms

Going back to our é example we'll add normalization to the mix. For each version of the string we want to work with we run it through the `normalize` methods and then test if they are equal (they are), what are their lengths (they are both 1) and if their lengths are equal (they are the same length).

```
const e1 = '\u00e9';
const e1n = e1.normalize();

const e2 = '\u0065\u0301';
con'\u0065\u0301'rmalize();


console.log(e1n == e2n) // true
console.log(e1n.length) // 1
console.log(e2n.length) // 1
console.log(e1n.length == e2n.length) // True
```

So, when working with international characters, particularly if you're working with user-provided input, you should take normalization into account to make sure that the characters are the same throughout the application and that you won't get unexpected results when using that data as part of your results.

Links and resources

- Introducing Character Sets and Encodings
- Character encodings: Essential concepts
- Choosing & applying a character encoding
- JavaScript has a Unicode problem
- When “Zoë” !== “Zoë”. Or why you need to normalize Unicode strings
- The Unicode portion of Ponyfoo’s ES6 Strings (and Unicode, ❤) in Depth