Unicode: What is it and why it matters

In the past I've visited Unicodes several times to discuss why it's important when working on the web:

- Empathy in Design and Development: Unicode and why it's important
- When is an é not an é? Working with Unicode on the web
- Unicode and multilingual support

In this post we'll revisit the history of languages on the web from ASCII to Unicode, pitfalls you may encounter and how to deal with them.

From ASCII to Unicode

In the beginning, the web and the internet were written in English and only needed 128 characters to describe all possible characters in use. This character set was known as ASCII (American Standard Code for Information Interchange).

As the reach of the internet grew, ASCII was no longer enough to describe all the characters needed to write content for the web. This 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 when you were working with a single language but there was no easy way to mix and match encodings to display them together on a web page. What happened if you were working in English (ISO 8859-1 or 8859-15) but had to reference modern Greek phrases (ISO 8859-7)?

Different encodings in the ISO-8859 family could use the same codepoint for two different characters, this would lead to all sorts of confusion and wrong characters appearing on the page based on the encoding in use.

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 glyphs.

Unicode is divided into 17 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 to keep in mind is that there are three version of Unicode, UTF-8, UTF-16, and UTF-32.

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.

Javasacript uses 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.

We mention UTF-32 for completeness but no one uses it because it’s inneficient.

Unicode Terminology

Taken from What every JavaScript developer should know about Unicode

Abstract character (or character) is a unit of information used for the organization, control, or representation of textual data.

Every abstract character has an associated name, and a rendered form (glyph).

Code point is a number assigned to a character. Code points are numbers in the range from U+0000 to U+10FFFF and are unique throughout Unicode.

U+<hex> is the format of code points, where U+ is a prefix meaning Unicode and <hex> is a number in hexadecimal (base 16) notation. For example, U+0041 and U+2603 are code points.

The magic happens because Unicode uniquely associates a code point with a character. The same codepoint will render the same glyph the same way everywhere.

Unicode planes and the Basic Multilingual Plane. Unicode is broken into 17 different planes (0 to 16) of 65,536 (or hex FFFF) contiguous Unicode code points from U+n0000 up to U+nFFFF, where n takes its value from the plane number in hexadecimal (base 16) form. If the plane is 0, the values go from 0000 to FFFF without adding an additional digit.

Plane 0 is a special one, the Basic Multilingual Plane or BMP. It contains characters from most modern languages (Basic Latin, Cyrillic, Greek, etc) and a big number of symbols.

As mentioned above, the code points from the Basic Multilingual Plane are in the range from U+0000 to U+FFFF and can have up to 4 hexadecimal digits

These are some characters from the BMP

```
console.log('\u0065');
console.log('\u007C');
console.log('\u25A0');
console.log('\u2602');
```

Now things get a little more complicated. Because UTF-16 only works with 16 bits at a time there are characters that will require more than one code unit to capture prioperly. This is where we use surrogate pairs.

A ***surrogate pair*** is a representation for a single abstract character that consists of a sequence of code units of two 16-bit code units, where the first value of the pair is a high-surrogate code unit and the second value is a low-surrogate code unit.

With those definitions we can look at some Unicode quirks and issues.

How we represent a character

Unicode can be very simple and very complicated at the same time.

Take the é character, an accented e that is used in Spanish and other Latin languages. It can be represented by:

- 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

Although both the single codepoint and the surrogate pair represent the same character, they are not equal and they don’t have the same length as demonstrated below:

```
console.log('\u00e9') // => é
console.log('\u0065\u0301') '\u0065\u0301'// => é\u00e9' === '\u0065\u030
co' === '// => false.length) // => 1
console.log('\u0065\u0301'.length) // => 2
'\u0065\u0301'// => 2
```

The same thing happens to characters with accents or other diacritical marks, and other special characters. Take the following as examples:

- ñ = n + tilde (‘\u006E\u0303’)
- ü = u + umlaut (‘\u0075\u0308’)

Emojis are even more complicated. We’ll use the Couple With Heart: Woman, Woman emoji as an example. It is made of four surrogate pairs:

- 👩 (\uD83D\uDC69)
- ❤️ (\u200D\u2764\uFE0F\u200D) (two surrogate pairs)
- 👩 (\uD83D\uDC69)

Using the surrogate pairs together to produce the emoji looks like this:

```
console.log('\uD83D\uDC69\u200D\u2764\uFE0F\u200D\uD83D\uDC69');
```

What’s the length of my string

Figuring out the length of a string using surrogate pairs can be complicated.

Using surrogate pairs will skew the length reported by the string.length method.

This is important in situations like minimal password length requirements or word count for text fields.

In the case of an é this is easy to fix because the character also has a single codepoint version, but sticking to surrogate pairs, the length of é is two.

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

The following example goes back to our `Emoji: Couple With Heart: Woman, Woman`. It reports the length as eight, again one for each element in a surrogate pair enven though it is represented as a single character.

```
console.log('\uD83D\uDC69\u200D\u2764\uFE0F\u200D\uD83D\uDC69'); // => 👩❤
console.log('\uD83D\uDC69\u200D\u2764\uFE0F\u200D\uD83D\uDC69'.length) '\
```

The final one is a pile of poo... literally. The character at codepoint U+1F4A9 is called A Pile of Poo. Because it uses a surrogate pair it has a length of two.

```
console.log('\uD83D\uDCA9') // => 💩
console.log('\uD83D\uDCA9'.length) '\uD83D\uDCA9'// => 2
```

For most situations a function like `countSymbols` below would give you the correct count of symbols, rather than codepoints.

The function uses Unicode Normalization to replace equivalent sequences of characters so that any text that are equivalent will be reduced to the same sequence of code points, called the normal form of the original text.

```
function countSymbols(string) {
  var normalized = string.normalize('NFC');
  return [...normalized].length;
}
```

However this will not work with larger sequences of surrogate pairs. The 👩❤️👩 emoji will still produce the incorrect result (6).

Mathias Bynens does a more thorough analyis of Unicode in pre-ES2015 Javascript in his JavaScript has a Unicode problem note, a strongly recommended reading.

Unicode escape codes

If you’re working with ES2015 or later you can take advantage of Unicode code point escapes as described in Mathias Bynens note on escapes.

Instead of using the surrogate pairs, we can use the full Unicode codepoint like in the examples below.

```
console.log('\u{1F4A9}') // => 💩
console.log('\u{1D306}') '\u{1D306}'// => �g('\u{1F925}') // => 🤥
'\u{1F925}'// => 🤥
```

You must find the Unicode codepoint for each character that you want to use in this fashion but, if supported, gives you much more flexibility on how to use Unicode in Javascript.

Using unicode in HTML entities

One last item to consider is how to add Unicode glyphs to your web content is to add them as entities.

You’ve likely seen entities in DevTools when looking at the elements panel or when you view the source of a document. They usually look like this

```
<h1>&nbsp;</h1>
<h1>&rdquo;</h1>
<h1>&ldquo;</h1>
```

These entities are built into the HTML spec and have been a part of it since the early days.

Now with Unicode we can tweak the way we write entities to include unicode characters. Taking the full codepoint for the glyph we want to show we add it to the entity declaration with an x before the number like the following examples

```
<h1>&#x2014;</h1>
<h1>&#x1F4A9;</h1>
<h1>&#x1D306;</h1>
```