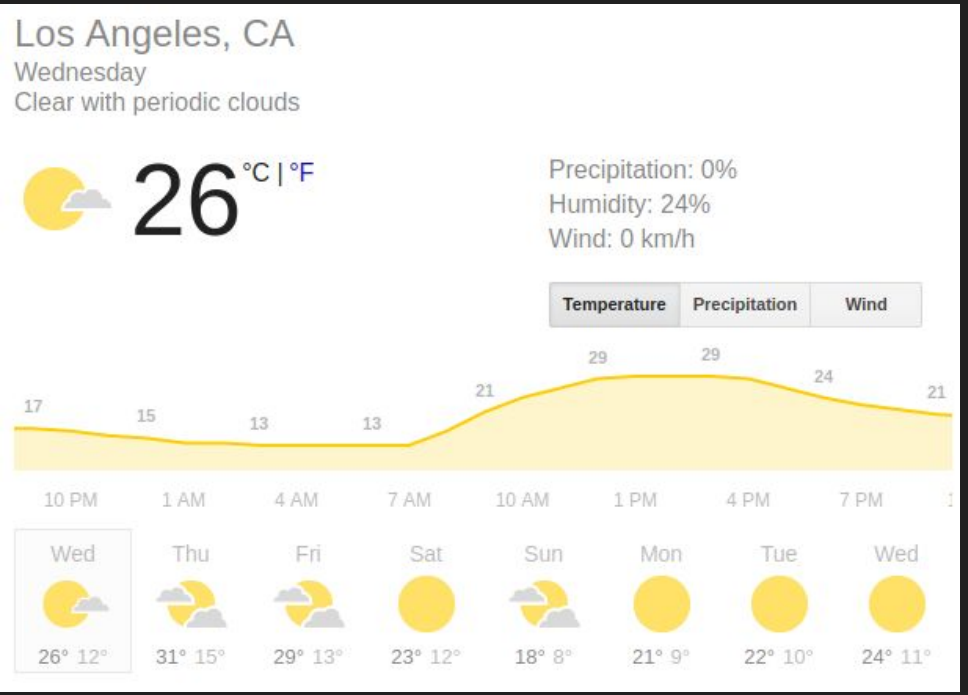
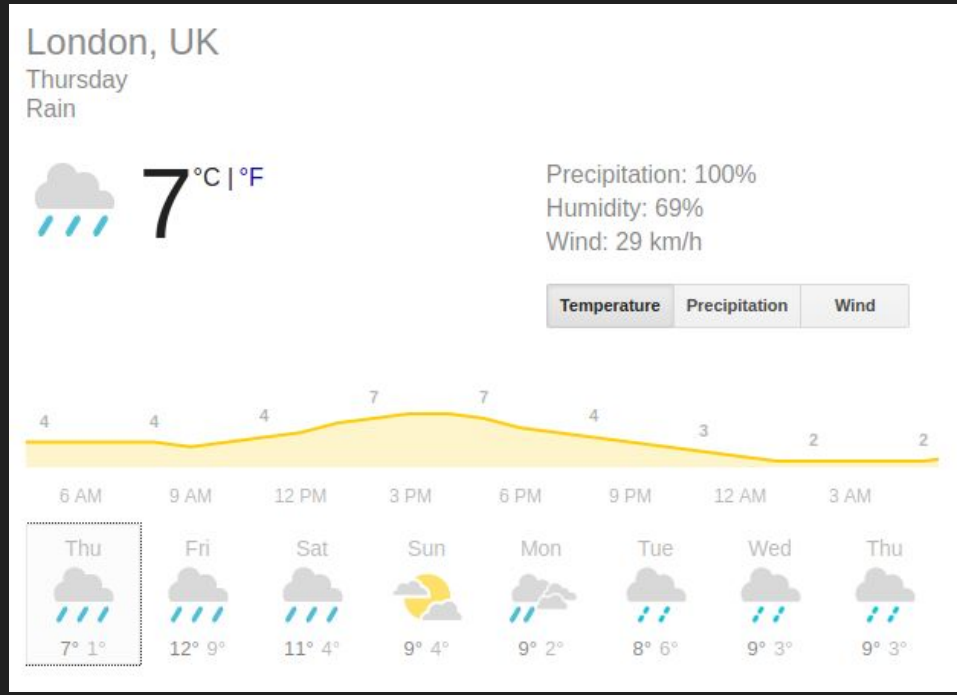
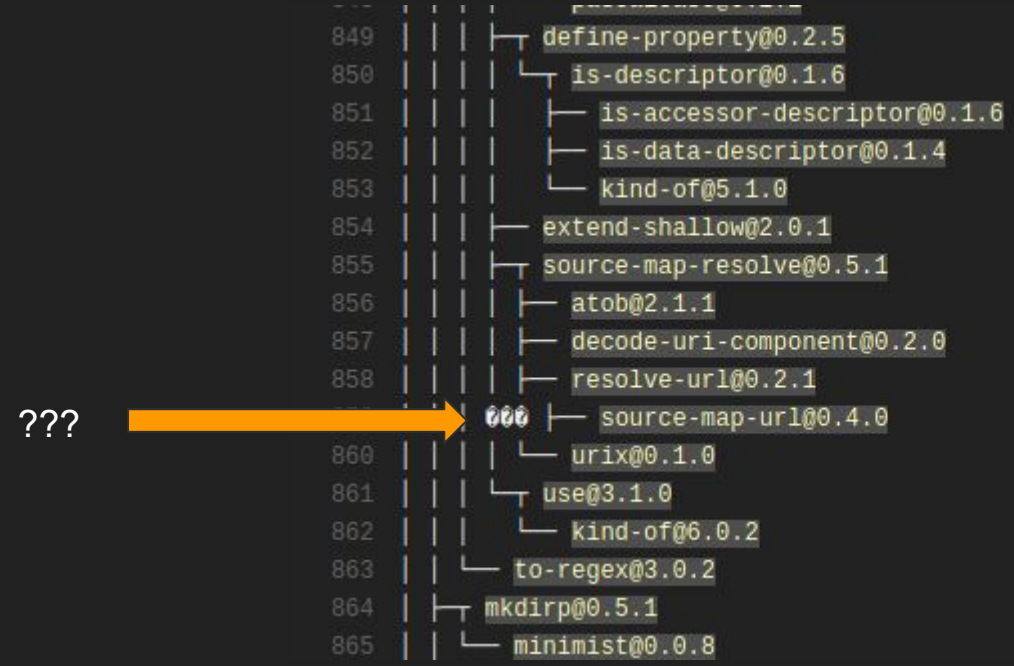
JS  Character Encodings

Anna Henningsen · @addaleax · she/her

It’s good to be back! 😉



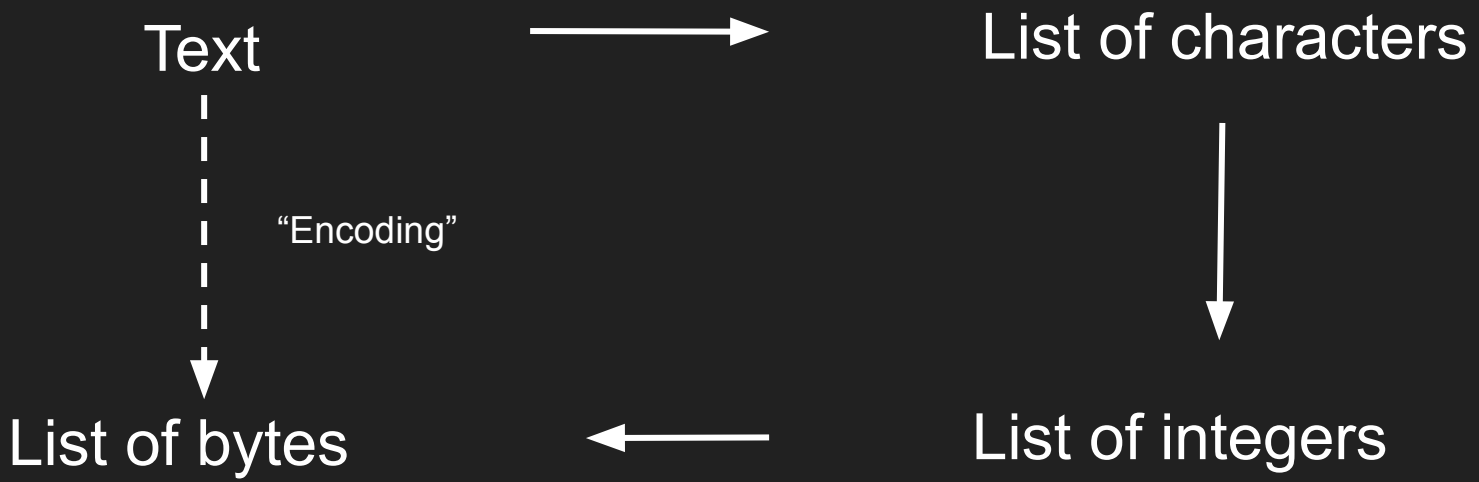




https://travis-ci.org/node-ffi-napi/get-symbol-from-current-process-h/jobs/641550176

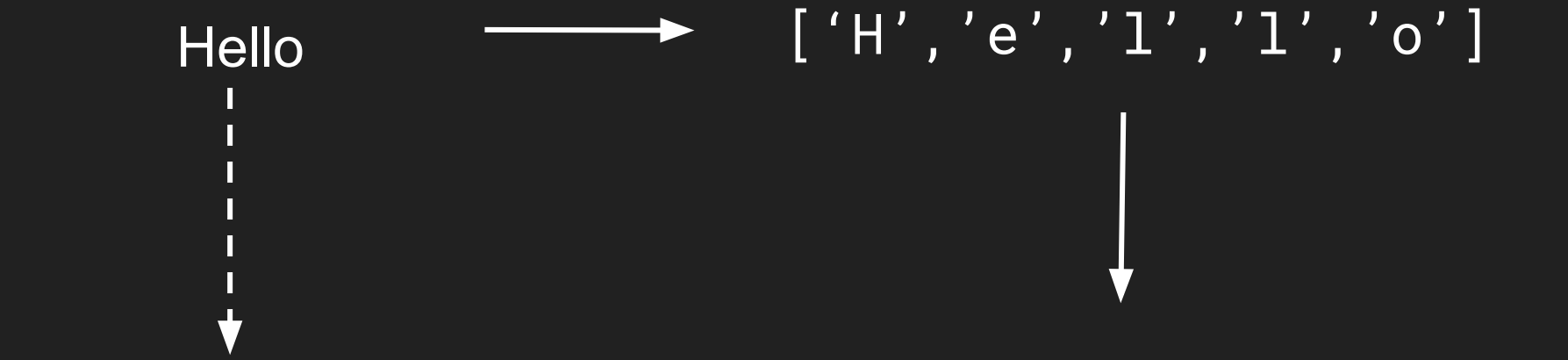
So … what's a character encoding?

People are good with text, computers are good with numbers



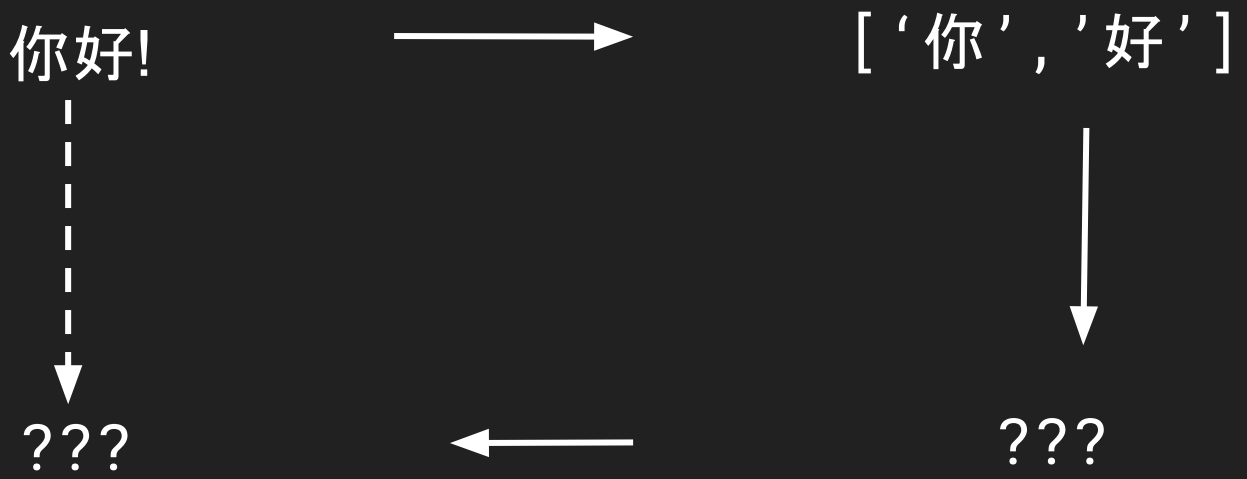
So … what's a character encoding?

People are good with text, computers are good with numbers



So … what's a character encoding?

People are good with text, computers are good with numbers



ASCII

0	0x00	<NUL>
…	…	…
65	0x41	A
66	0x42	B
67	0x43	C
…	…	…
97	0x61	a
98	0x62	b
…	…	…
127	0x7F	<DEL>

ASCII

- 7-bit
- Covers most English-language use cases
- … and that’s pretty much it

ISO-8859-*, Windows code pages

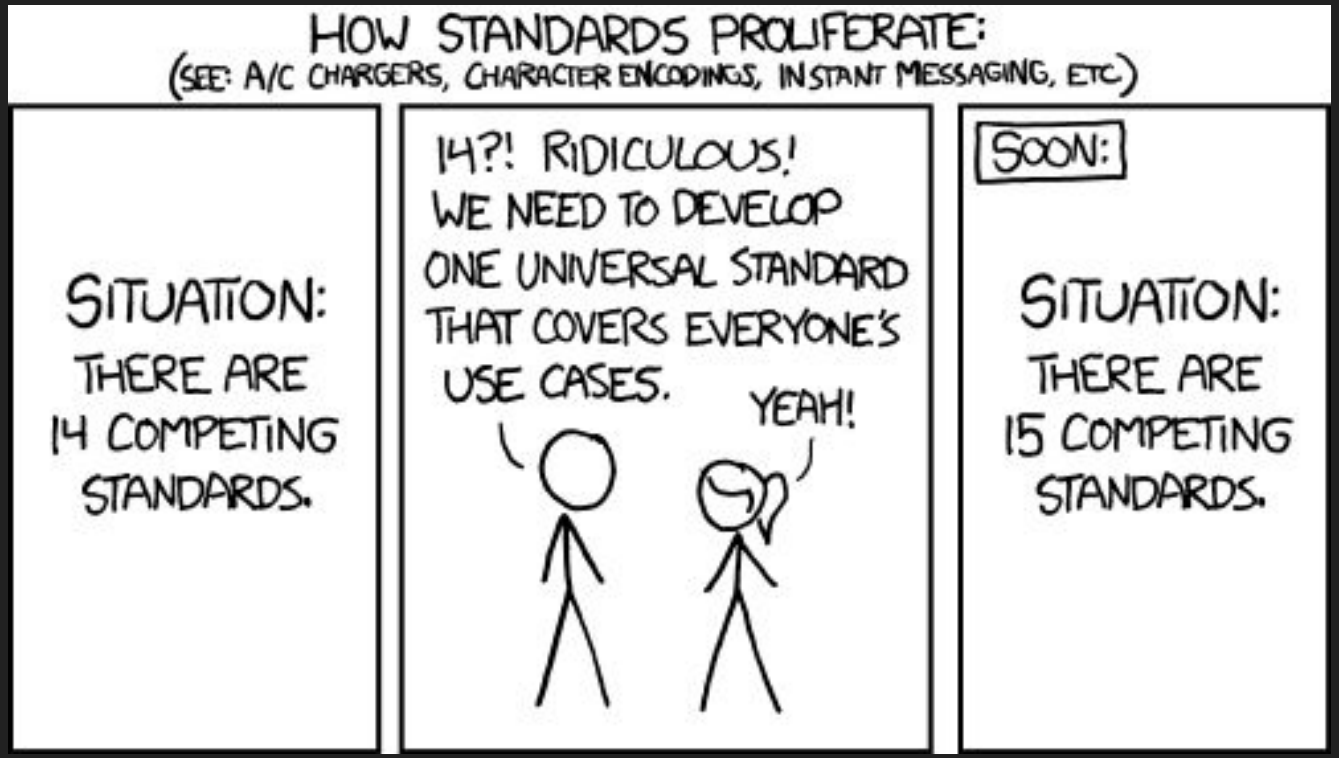
- Idea: Usually, transmission has 8 bit per byte available, so create ASCII-extending charsets for more languages

	ISO-8859-1 (Western) (aka Latin-1)	ISO-8859-5 (Cyrillic)	Windows-1251 (Cyrillic)
…	…	…	…
0xD0	Ð	а	Р
0xD1	Ñ	б	С
0xD2	Ò	в	Т
…	…	…	…

GBK

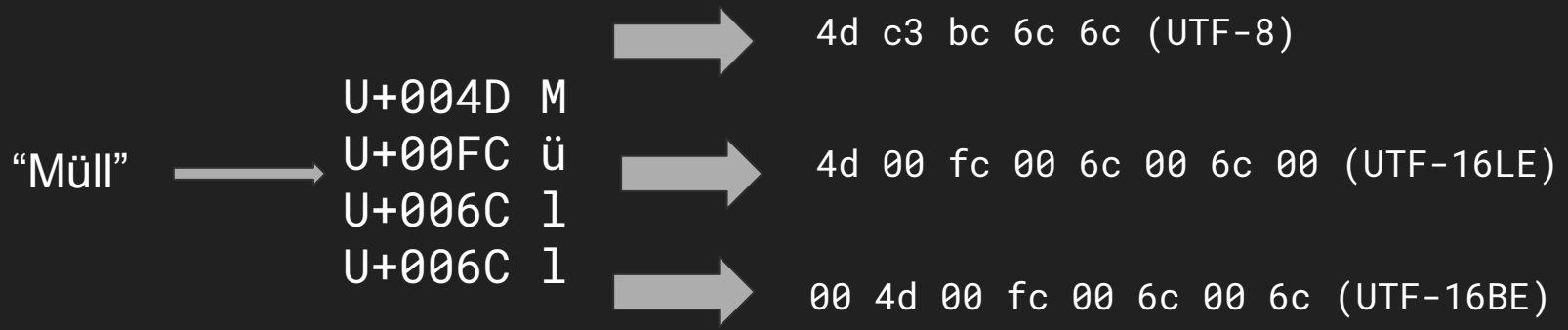
- Idea: Also extend ASCII, but use 2-byte for Chinese characters

…	…
0x41	A
0x42	B
…	…
0xC4 0xE3	你
0xC4 0xE4	匿
…	…



https://xkcd.com/927/

Unicode: Multiple encodings!



Unicode

- New idea: Don’t create a gazillion charsets, and drop 1-byte/2-byte restriction
- Shared character set for multiple encodings: U+XXXX with 4 hex digits, e.g. U+0041 = A
- Character numbering backwards-compatible with ISO-8859-1
- Goes up to U+10FFFF > 1M characters
- … Emoji! 🎉 😍 😸
- Special replacement character: U+FFFD 🞜
- Supported in HTML as &#x????; (hex) or &#????; (decimal)
- Supported in JS as \u???? or \u{?????}

UTF-8

Variable-length encoding with single-byte code units:

U+0000 - U+007F: 0xxxxxxx
U+0080 - U+07FF: 110xxxxx 10xxxxxx
U+0800 - U+FFFF: 1110xxxx 10xxxxxx 10xxxxxx
U+10000 - U+1FFFFF: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

- ASCII-compatible
- “Lead bytes” are >= 0xC0
- “Trailing bytes” are >= 0x80 and < 0xC0
- Missing/invalid bytes do not break decoding

UTF-8 broken decoding example

UTF-16

- Uses 2-byte code units
- Characters > U+FFFF split into two units from 0xD800 to 0xDFFF (“surrogate pairs”)
- Comes in Little Endian and Big Endian variants
- Maybe use special character U+FEFF (“BOM”) to distinguish LE/BE

→ `(0xFEFF) 0xD83C 0xDF89`

→ `(FF FE) 3C D8 89 DF`

→ `(FE FF) D8 3C DF 89`

“JavaScript uses UTF-16”

Well … yes and no:

- JavaScript does *not* perform any conversion of strings into bytes
- The underlying memory may or may not be formatted in UTF-16
  - (JS Engines are clever about this!)
- JavaScript *does* use character codes in the range 0 – 65535
- JavaScript strings *do* use surrogate pairs in the style of UTF-16

```
‘🎉’.length === 2
'🎉' === '\uD83C\uDF89’
```

Side note: What actually happens

- Both V8 and SpiderMonkey distinguish between Latin-1-only strings and strings requiring full 2-byte code units
- String representations are *complicated* anyway
- Don’t overthink it

Converting back and forth in JS

Node.js:

```
const buf = Buffer.from('Hi!', 'utf8');
console.log(buf.toString('utf8'));
```

Browser (or Node.js 12+ or Node.js 10 with `require('util')`):

```
const uint8arr = new TextEncoder().encode('Hi!');
console.log(new TextDecoder('utf8').decode(uint8arr);
```

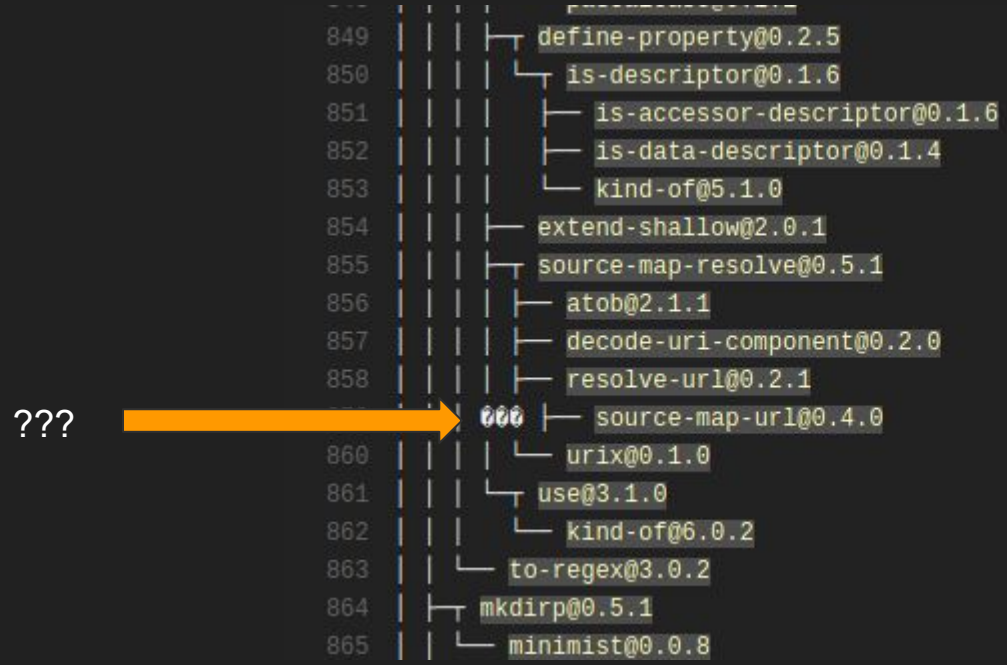
⚠️ `TextDecoder` supports a range of encodings, `TextEncoder` only UTF-8! ⚠️

Dealing with decoding errors

`TextDecoder` has a `fatal` option that makes it throw exceptions:

```
> new TextDecoder('utf-8').decode(new Uint8Array([0xff]))
'�'
> new TextDecoder('utf-8', {
  fatal: true
}).decode(new Uint8Array([0xff]))
TypeError [ERR_ENCODING_INVALID_ENCODED_DATA]: The encoded
data was not valid for encoding utf-8
```

Generally, it is okay to leave `�` when it happens.



https://travis-ci.org/node-ffi-napi/get-symbol-from-current-process-h/jobs/641550176

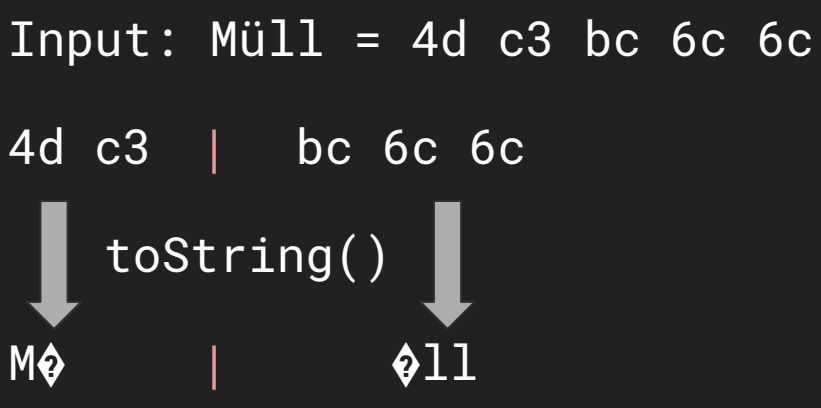
What’s wrong with this? (Node.js variant)

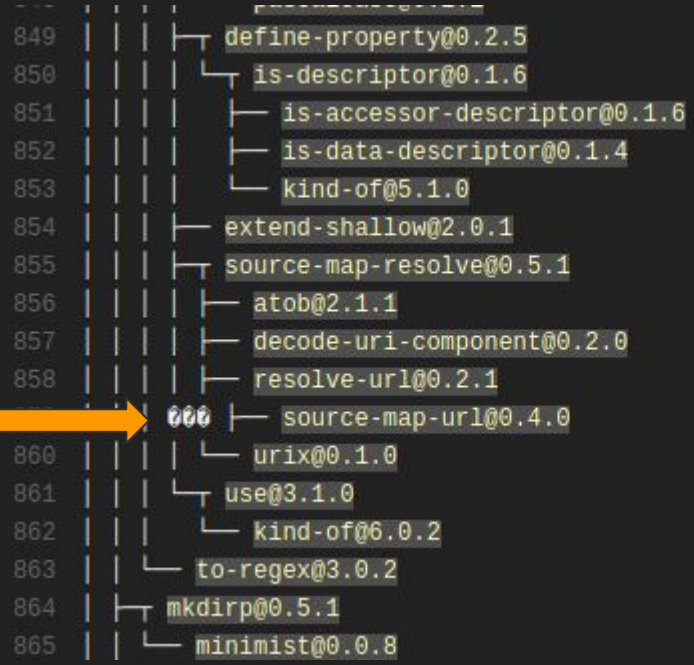
```
const data = ‘’;
process.stdin.on(‘data’, (buffer) => {
  data += buffer;
});
process.stdin.on(‘end’, () => {
  process.stdout.write(data);
});
```

What’s wrong with this? (Node.js variant)

```
const data = ‘’;
process.stdin.on(‘data’, (buffer) => {
  data += buffer; // Implicit buffer.toString() call
});
process.stdin.on(‘end’, () => {
  process.stdout.write(data);
});
```

Imagine that this happens…





https://travis-ci.org/node-ffi-napi/get-symbol-from-current-process-h/jobs/641550176

Let’s fix it:

```
const data = ‘’;
process.stdin.setEncoding(‘utf8’);
process.stdin.on(‘data’, (string) => {
  data += string;
});
process.stdin.on(‘end’, () => {
  process.stdout.write(data);
});
```

Under the hood: Streaming decoders

```
const decoder = new StringDecoder('utf8');  // Node.js
const str1 = decoder.write(buffer1);
const str2 = decoder.write(buffer2);
const str3 = decoder.end();
```

```
const decoder = new TextDecoder('utf8');  // Browser + Node
const str1 = decoder.decode(buffer1, { stream: true });
const str2 = decoder.decode(buffer2, { stream: true });
const str3 = decoder.decode(new Uint8Array());
```

Let’s talk a bit more about surrogates in JS…

- `‘🤡’ === ‘\uD83E\uDD21’`
- So, `‘🤡’.length === 2`
- How do we get the number of *characters*? How do we figure out the actual characters?

Option 1: Strings are iterables

```
const str = ‘Clown 🤡’;
console.log([...str]);  // [‘C’,‘l’,‘o’,‘w’,‘n’,‘ ’,‘🤡’]

let len = 0;
for (const char of str) len++;
console.log(len);
```

Option 2: Manual work

```
const str = '🤡';
console.log(str.charCodeAt(0));   // 0xD83E
console.log(str.charCodeAt(1));   // 0xDD21
console.log(str.codePointAt(0));  // 0x1F921
console.log(str.codePointAt(1));  // 0xDD21

// This also gives us the reverse transformation:

String.fromCharCode(0xD83E, 0xDD21) === '🤡';
String.fromCodePoint(0x1F921) === '🤡';
```

Regular expressions are fun

```
> /e{2,4}/.test('beehive')
true
> /🐈{2,4}/.test('two cats: 🐈🐈')
false
```

Regular expressions are fun

`/🐈{2,4}/` expands to `/\uD83D\uDC08{2,4}/` 😞

Luckily, there's an easy solution:

```
> /🐈{2,4}/.test('two cats: 🐈🐈')
false
> /🐈{2,4}/u.test('two cats: 🐈🐈')
true
```

Regular expressions are *even more* fun

Not yet supported everywhere, but:

```
‘This is a cat: 🐈’.match(/\p{Emoji_Presentation}/gu)
> [ '🐈' ]
```

Just because two strings look the same…

```
> 'André' === 'André'
false
> '한글' === '한글'
false
```

Unicode is a bit too clever here…

Just because two strings look the same…

```
> [...'André'].map(c =>
    c.codePointAt(0).toString(16).padStart(4, 0))
[ '0041', '006e', '0064', '0072', '0065', '0301' ]

> [...'André'].map(...)
[ '0041', '006e', '0064', '0072', '00e9' ]

> '한글'.length
6
> '한글'.length
2
```

Unicode normalization

Four normalization modes that can be used with `String.prototype.normalize()`:

1. NFC: “Canonical” decomposition + “Canonical” composition,
   e.g. ‘é’ or or ‘한’ are single characters
2. NFD: “Canonical” decomposition
   e.g. ‘é’ is composed out of 2 characters (e + ´), ‘한’ out of three characters (ㅎ + ㅏ + ㄴ)

You may want to use this when comparing strings

Unicode normalization, cont’d

Four normalization modes that can be used with `String.prototype.normalize()`:

1. NFKC: “Compatibility” decomposition + “Canonical” composition,
   e.g. ‘𝐇𝐄𝐋𝐋𝐎’ turns into ‘HELLO’
2. NFKD: “Compatibility” decomposition
   e.g. ‘𝐇𝐄𝐋𝐋𝐎’ turns into ‘HELLO’ (but ‘ã’ is turned into a + ̃)

You may want to use this for e.g. search parameters

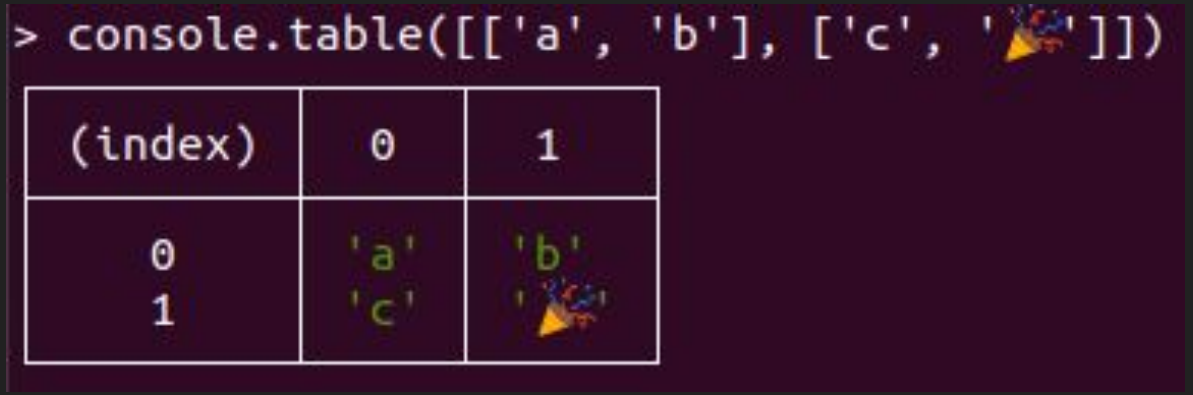
So … what does str.length actually tell us?

Not a lot:

- *Not* the number of characters – characters can be composed
- *Not* the number of Unicode code points – characters can be split into UTF-16-style surrogate pairs
- *Not* the string “width” – remember, `'한글'.length === 6`
- Basically only half the byte length when encoded as UTF-16… 😕

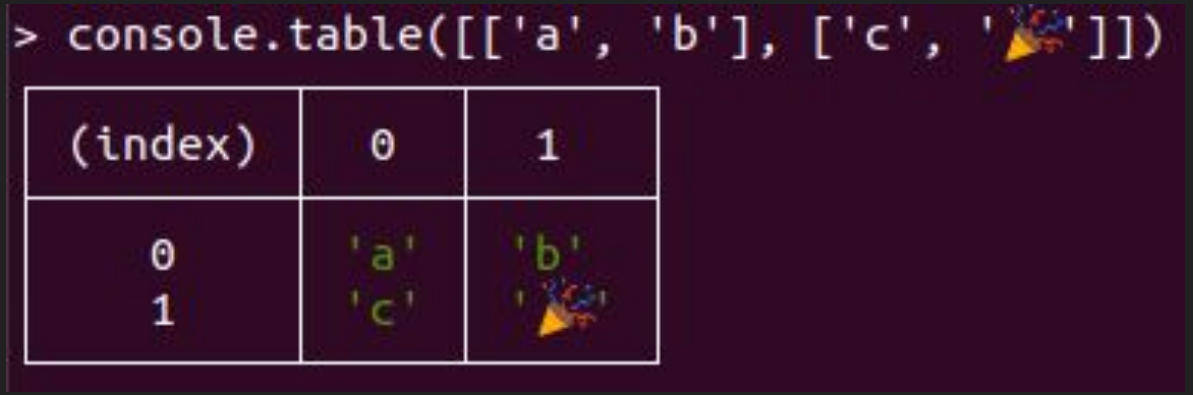
Àpropos string width…

How does this work?



Àpropos string width…

How does this work?




```
require('string-width')('🎉') === 2
```

Side note: Node.js v13.x REPL bug up for grabs?

```
> '한글'.let  engths
6
```

Our string width implementation doesn’t account for the way that the Hangul characters are composed… do we need to call `str.normalize(‘NFC’)` first? Does that always do the right thing? Why is this only problematic on v13.x?

So… about that `binary` Node.js encoding

- A long, long time ago … we didn’t have `Uint8Array`
- Binary data was still real, though
- The only good sequence type besides arrays were strings, so…

So… about that `binary` Node.js encoding

- A long, long time ago … we didn't have `Uint8Array`
- Binary data was still real, though
- The only good sequence type besides arrays were strings, so…

```
» gzippedDataAsBinaryString
← "\u001f\u008b\u0008\u0000\u0000\u0000\u0000\u0000\u0000\u0003uRANÃ0\u0010¼÷\u0015s\"ETá\u0001%\u0007\u0010\u0088V\u0008\u0081
\u0012ÇÈM×\u0089"
```

Use U+0000 through U+00FF to represent bytes 0 through 255

So… about that `binary` Node.js encoding

- We have something better: Uint8Array/Buffer
- There's actually a better name for the encoding: latin1!
- Most importantly: The name is *really* misleading – *all* character encodings convert strings to bytes, 99 % of modern usage is based on misunderstanding
- This is (was) kind of the big issue with Python 2 vs Python 3

(One use case for "binary strings" that remains: `atob()` / `btoa()` in the browser)

Side note: Node.js character encodings

Node.js supports:

- ascii
- utf8
- utf16le (a.k.a. ucs2)
- latin1 (a.k.a. binary)
- base64 (this is a binary-to-text encoding, *not* a character encoding)
- hex (this is a binary-to-text encoding, *not* a character encoding)

base64 + hex

⚠️ Warning:

- For character encodings, string → bytes is *encoding* and bytes → string is *decoding*
- For text-to-binary encodings, string → bytes is *decoding* and bytes → string is *encoding*
- So, depending on the parameters `Buffer.from()` can encode or decode, and `buffer.toString()` can decode or encode

Everybody uses UTF-8 now anyway, right?

- Legacy code and legacy websites exist…
- People sometimes don’t notice that they *don’t* use UTF-8 (e.g. in the binary case)
- We added `Buffer` support to the Node.js file system API because we had to
- The native Windows API is a *big* fan of UTF-16 😞
- Even when using UTF-8, things can still go wrong
- The speaker website couldn’t get this talk’s title right at first 😬
- Character encodings are part of your APIs!

Why is UTF-8 so popular anyway?

1. Backwards compatibility with ASCII
2. That’s it.

Why is UTF-8 so popular anyway?

1. Backwards compatibility with ASCII
2. That’s it.

Applications built for ASCII work with UTF-8 99 % of the time. Allowing for the other 1 % won over having to re-write tons of text handling code.

Resources

- `iconv(1)`
- `unicode(1)`
- MDN:
  - Binary strings - Web APIs | MDN
  - Intl - JavaScript | MDN
  - RegExp - JavaScript | MDN
  - Unicode property escapes - JavaScript | MDN
  - TextDecoder - Web APIs | MDN
  - TextEncoder - Web APIs | MDN
- https://nodejs.org/api/buffer.html … to some degree

Thank you!

Slides will be published soon!
@addaleax