expect("💩").length .toBe(1)

No. It’s not just strings

Where’s the problem?

- "".length == 0
- "a".length == 1
- "ä".length == 1

Others (PHP) already fail here

- strlen("") == 0
- strlen("a") == 1
- strlen("ä") == 2

I cheated - my editor was in UTF-8 mode. I can also make strlen("ä") be 1. (or 3 or 4)

Ah. But PHP sucks! Let's use Ruby.

```
pilif@miscweb ~ % ruby --version
ruby 1.8.7 (2011-02-18 patchlevel 334) [i686-linux]
pilif@miscweb ~ % irb
irb(main):001:0> "ä".length
=> 2
irb(main):002:0> _
```

Yes. It's unfair to use an outdated version of Ruby. 1.9 has (generally) fixed this.

Whatever. We're doing JS and JS does it right. Right?

```
>>> "a".length
1
>>> "ä".length
1
>>> "ザ".length
1
```

```
>>> "𝄠".length
2
>>> "💩".length
2
```

What gives?

You know. Historical reasons

What is a string?

- Compound type
- Array of characters
- C says char*
- char is defined as the "smallest addressable unit that can contain *basic character set*". Integer type. Might be signed or unsigned
- Ends up being a byte

Traditional string APIs

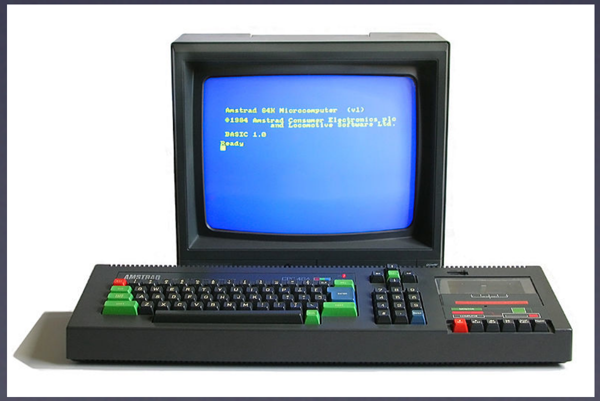
- Length of a string? count bytes until the end (\0) and divide by sizeof(char)
- Accessing the n-th character? Add n*sizeof(char) to the pointer
- Remember: sizeof(char) usually is 1 and guess how people "optimized"

Interacting with the world

- Just dump the contents of the memory into a file
- Read back the same contents and put it in memory
- Problem solved.
- Until you need to do this across machines

Interoperability

- char is inherently implementation dependent
- So is by definition the file you dump your char* into
- Can’t move files between machines



ASCII

- "**American** Standard Code for Information Interchange"
- Published 1963
- Uses 7 bits per character (circumventing the signedness-issue)
- Perfectly fine for what everybody is using (English)

But I need ümläüte

- Machines were used where people speak strange languages (i.e. not English)
- ASCII is 7bit. Adding a bit gives us another 127 characters!
- Depending on your country, these upper 127 characters had different meanings
- No problem as texts usually don't leave their country

remember “chcp 850”?

Thüs wäs nöt

pюssïѢlę!

I apologize to all Russians for butchering their script.

Then the Internet happened

Unicode 1.0

- 16 bits per character
- Published in 1991, revised in 1992
- Jumped on by everybody who wanted "to do it right"
- APIs were made Unicode compliant by extending the size of a character to 16 bits. Algorithms stayed the same

65K characters are enough for everybody

640K are enough for everybody

Still just dumping memory

- wchar is 16 bits
- Endianness? See if we care!
- To save to a file: Dump memory contents.
- To load from a file: Read file into memory
- Note they didn't dare extending char to 16 bits
- Let's call this "Unicode"

16 bits everywhere

- Windows API (XxxxXxxW uses wchar which is 16 bit wide)
- Java uses 16 bits
- Objective C uses 16 bits
- And of course, JavaScript uses 16 bits
- C and by extension Unix stayed away from this.

That's perfect. By using 16 bit characters, we can store all of Unicode!

It didn’t work out so well

- By just dumping memory, there’s no way to know how to read it back
- Heuristics suck (try typing “Bush hid the facts” in Windows Notepad, saving, reloading)
- Most protocols on the internet allow to specify a character set

BOM

No. Really

- Implementations lie.
- Legacy software had (well. has.) huge problems with wide characters
- Issues with updating old file formats
- 65K characters are not nearly enough

We learned

- UTF has happened
- specifically UTF-8 happened
- Unicode 2.0 happened
- Programming environments learned

Unicode 2.0+

- Theoretically unlimited code space
- Doesn’t talk about bits any more
- The terminology is code point.
- Currently 1.1M code points
- The old characters (0000 - FFFF) are on the BMP

Unicode Transformation Format

- Specifies how to store Unicode on disk
- Specifies exact byte encoding for every Unicode code point
- Available for 8-, 16- and 32 bit encodings per code point
- Not every byte sequence is a valid UTF byte sequence (finally!)

UTF-8

- Uses an 8bit encoding to store code points
- Is the same as ASCII for whatever's in ASCII
- Uses multiple bytes to encode code points outside of ASCII
- The old algorithms don't work any more

UTF-16

- Combines the worst of both worlds
- Uses 16bit to encode a code point
- Uses multiple of 16bits to encode a code point outside of the BMP
- Wastes memory for ASCII, has byte-ordering-issues and **still** breaks the old algorithms.
- Is the only way for these 16bit bandwagon jumpers to support Unicode 2.0 and later

UTF-32

- 4 bytes per character
- Byte ordering issues
- Still breaking the old algorithms due to combining marks

Strings are not bytes

- A string is a sequence of characters
- A byte array is a sequence of bytes
- Both are incompatible with each other
- You can encode a string into a byte array
- You can decode a byte array into a string

Which brings us back to JS

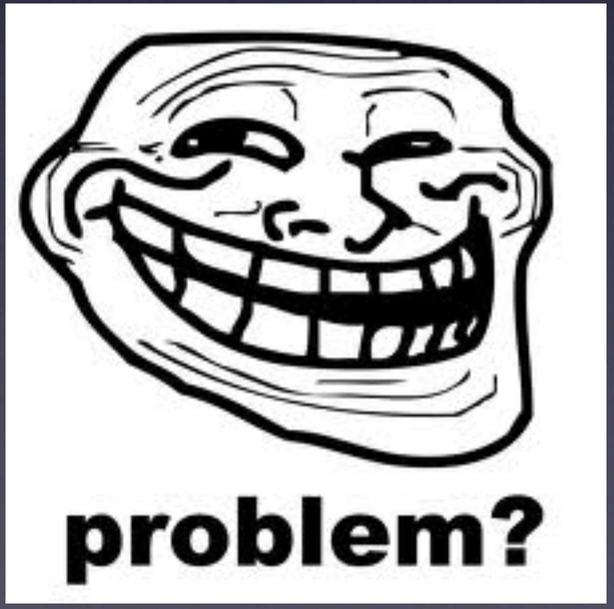
- Lives back in 1996
- Strings specified as being stored in UCS-2 (Fixed 16 bits per character)
- Leaks its implementation in the API
- Doesn’t know about Unicode 2.0

Browsers cheat

- Browsers of course support Unicode 2.0
- We need to display these piles of poo!
- Browsers expose Unicode strings to JS using UTF-16
- The JS API doesn’t know about UTF-16 (or Unicode 2.0)

String methods are leaky

- String.length returns mish-mash of **byte length** and **character length** for strings outside the BMP
- substr() can break strings
- charAt() can return non-existing code-points
- and let’s not talk about to*Case



Samples

```
>>> "Foo💩Bar".length
8
>>> "Foo💩Bar".charAt(3)
"[D83D]"
>>> "Foo💩Bar".substr(0, 4)
"Foo[D83D]"
```

That D8 3D is half of the UTF-16 encoding of U+1F4A9
which is 3d d8 a9 dc

Et tu RegEx?

- Character classes don't work right
- Counting characters doesn't work right
- Can break strings

```
>>> "a".match(/\w/)
["a"]
>>> "[illegible]".match(/\w/)
null
>>> "[illegible]".match(/./)
["D840"]
>>> "[illegible]".match(/.{2}/)
["[illegible]"]
```

Intermission: Digraphs

- ä is not the same as ä
- ä can be “LATIN SMALL LETTER A WITH DIAERESIS”
- it can also be “LATIN SMALL LETTER A” followed by “COMBINING DIAERESIS”
- both look exactly the same

No Normalization

```
pilif@kosmos:~|⇒ tail -n 4 poo-utf8.html | head -n 2 | hexdump -C
00000000  20 20 20 20 3c 73 70 61  6e 20 69 64 3d 22 6f 6e  |    <span id="on|
00000010  65 2d 63 6f 64 65 70 6f  69 6e 74 22 3e c3 a4 3c  |e-codepoint">..<|
00000020  2f 73 70 61 6e 3e 0a 20  20 20 20 3c 73 70 61 6e  |/span>.    <span|
00000030  20 69 64 3d 22 74 77 6f  2d 63 6f 64 65 70 6f 69  | id="two-codepoi|
00000040  6e 74 73 22 3e 61 cc 88  3c 2f 73 70 61 6e 3e 0a  |nts">a..</span>.|
00000050
```

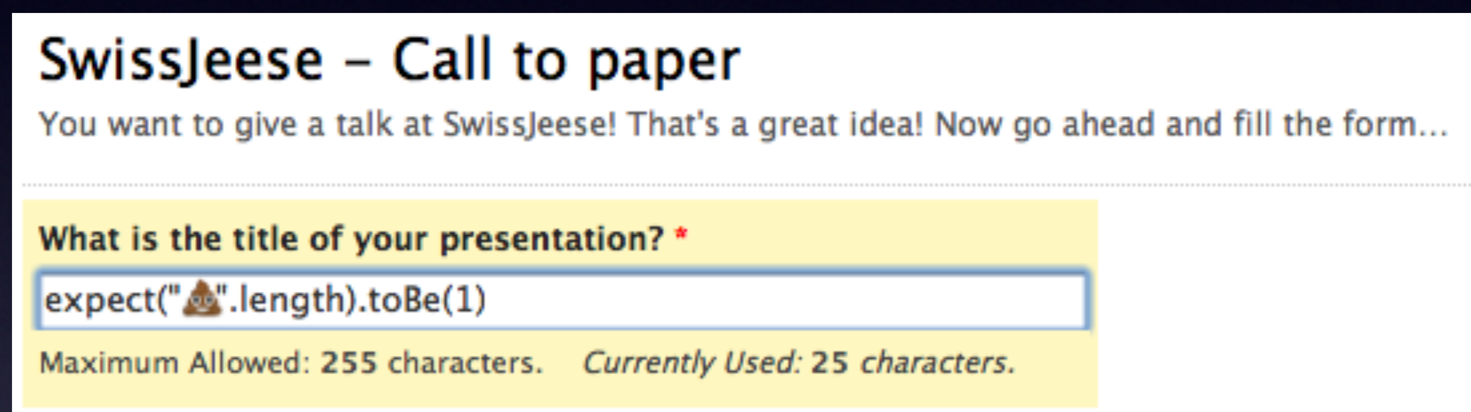
```
<body>
  <span id="one-codepoint">ä</span>
  <span id="two-codepoints">ä</span>
</body>
```

```
>>> one_codepoint = document.getElementById('one-codepoint').innerHTML;
"ä"
>>> two_codepoints = document.getElementById('two-codepoints').innerHTML
"ä"
>>> one_codepoint == two_codepoints
false
```

To add insult to injury

```
>>> two_codepoints.length
2
```

Real-World example



SwissJeese – Call to paper

You want to give a talk at SwissJeese! That's a great idea! Now go ahead and fill the form…

What is the title of your presentation? *

expect("💩".length).toBe(1)

Maximum Allowed: **255** characters. *Currently Used:* **25** *characters.*

The title of this talk has 24 characters :-)

Others screwed it up too

PHP

- At least you get to chose the internal encoding.
- PHP only does bytes by default. strlen() means bytelen()
- Forget a /u in preg_match and you’ll destroy strings. \s matches UTF-8 ä (U+00EF is 0xa420 and 0x20 is ASCII space)
- use any non mb_* function on a utf-8 string to break it

Python < 3.3

- They do clearly separate bytes and strings
- Use str.encode() to create bytes and bytes.decode() to go back to strings
- Unfortunately, UCS2 (mostly)

```
Python 3.1.1+ (r311:74480, Nov  2 2009, 14:49:22)
[GCC 4.4.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> len("a")
1
>>> len("ä")
1
>>> len("𝜑")
2
>>>
```

Some did it ok

- Python 3.3 (PEP 393)
- Ruby 1.9 (avoids political issues by giving a lot of freedom)
- Perl (awesome libraries since forever)
- ICU, ICU4C (http://icu-project.org/)

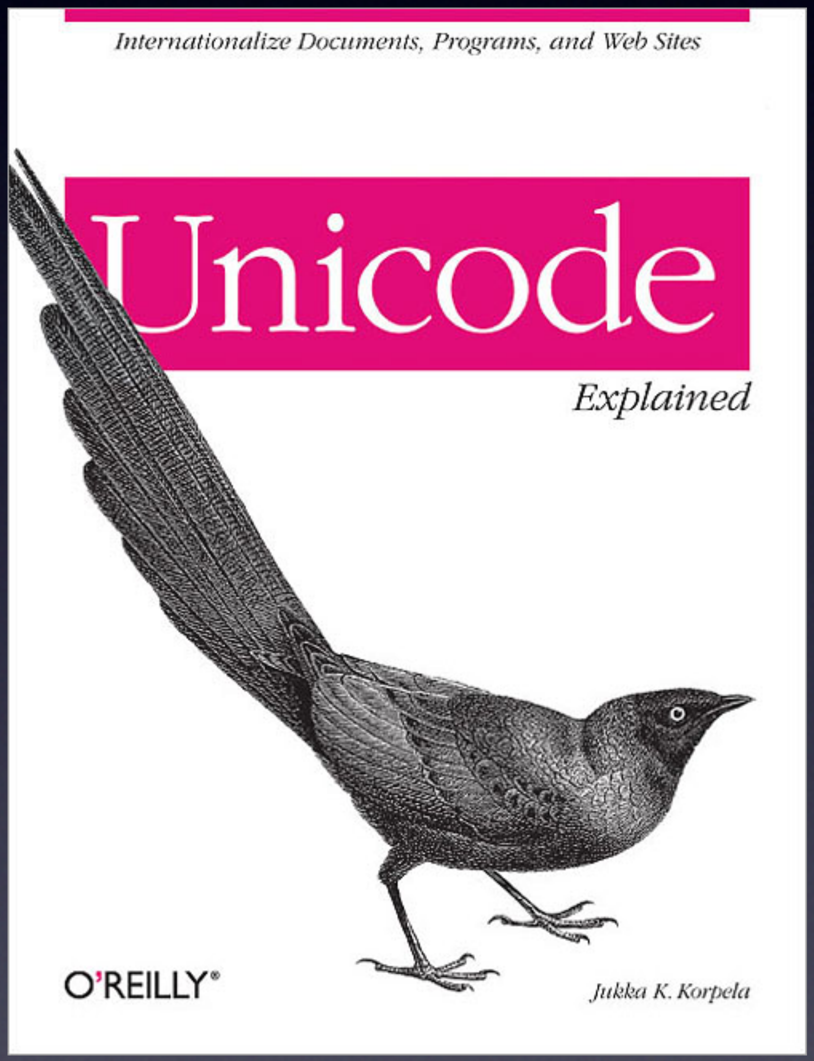
Solutions for JS

- Discussions happening for ES6
- Usable by 2040 or later I guess
- On the server: Use ICU
  - Only normalization currently available at https://github.com/astro/node-stringprep
- Manual bit-twiddling
- Regular expressions will still be broken
- Problem safe to ignore?

This was just the tip of the iceberg!

- Localization issues (Collation, Case change)
- Security issues (Encoding, Homographs)
- Broken Software (including “US UTF-8”)

Highly recommended Literature



Thank you!

- @pilif on twitter
- https://github.com/pilif/

Also: We are looking for a front-end designer with CSS skills. Send them to me if you know them (or are one)