

Lecture 10: ASCII Character Codes

Written by Ben Yan, from materials by Jerry Cain

ASCII and Unicode Methods

Computers use numeric encodings to represent character data inside the physical memory of a machine. That is, what you see as text characters, e.g., "B", "Y", "!", "@", are really numbers in camouflage.

Each character is assigned an integer value, called a *code*, and back in the old days and medieval Silicon Valley, they weren't exactly standardized ☹️. Different computer manufacturers used their own character-to-code mappings, and sharing data across different machines was truly, truly painful.

A great renaissance was needed, and the first widespread lingua franca of character encoding was **ASCII** (*American Standard Code for Information Interexchange*). Goodness, computer scientists love their acronyms. **The ASCII standard used 128 code points, 0 to 127.** Out of them, **95 codes were for printable characters**, e.g., *alphabet letters, numeric digits, punctuation*, and **33 for control characters**, e.g., *backspace, delete, carriage return*, etc., which were critical to typewriters and early printer devices.

You can look up the table on the web, and you're not required to memorize them all—but you should know that neighboring uppercase letters have consecutive codes, which string algorithms often leverage.

Character	A	B	C	D	E	F	G	H	I	...	Y	Z
Code	65	66	67	68	69	70	71	72	73	...	89	90

That is, "A" is 65, "B" is 66, "C" is 67 🙌🙌, and so on, up to "Z", which is 90. Lowercase letters also have contiguous codes, although they start later at 97 ("a"), and skyscraper up to 122 ("z").

Character	a	b	c	d	e	f	g	h	i	...	y	z
Code	97	98	99	100	101	102	103	104	105	...	121	122

If you're curious, the character codes of numeric digits ("0" to "9") are also laid out contiguously 😊.

Character	0	1	2	3	4	5	6	7	8	9
Code	48	49	50	51	52	53	54	55	56	57

With only 128 characters, the ASCII system proved inadequate in bringing together the many alphabets in use throughout the globe, not to mention emojis 🙄. It has therefore been superseded by Unicode, which defines 172,808 characters in 175 scripts, **though the first 128 characters are identical to ASCII.**

As some fun Unicode examples, the melting face emoji, 🍷, has character code **129760**; the Euro currency sign, €, has code **8364**; and the Chinese/Japanese character for world, 世, has code **19990**.

Unicode String Methods in JavaScript

There are two methods you'll want to remember, which enable us to translate between JavaScript characters, e.g., "B", and their Unicode numeric representations, e.g., 66. Those Unicode methods are:

String.fromCharCode (code)	Returns the one-character string whose Unicode value is code . String is hardcoded here, and refers to the string library itself.
charCodeAt (index)	Returns the Unicode value of the character at the specified index of the string calling charCodeAt .

For example, **String.fromCharCode (65)** returns "A", while **"A".charCodeAt (0)** returns 65.

The **charCodeAt** method can be used for strings of any length, but handles one index at a time. For instance, **"ABC".charCodeAt (1)** returns 66, the Unicode of the character at index 1 of "ABC".

One useful application of these methods is computing an offset or index within the alphabet. That is,

```
let offset = ch.charCodeAt(0) - "a".charCodeAt(0);
```

is an expression that, for a lowercase letter **ch**, calculates its alphabetic distance from "a". For instance, if **ch = "b"**, the offset is 1, and if **ch = "z"**, the offset is 25. If we replace "a" with upper "A", i.e.

```
let offset = ch.charCodeAt(0) - "A".charCodeAt(0);
```

we have an expression that works similarly for uppercase letters ☺. If we want to go the other way around, and find the letter that is **offset** positions after "a" in the alphabet, we can write

```
let ch = String.fromCharCode("a".charCodeAt(0) + offset);
```

Character to Upper Case

As we saw earlier, lowercase and uppercase letters have contiguously laid out character codes. One can think of them like parallel train tracks, "A" (65) to "Z" (90), and "a" (97) to "z" (122). We *capitalize* (no pun intended) on this fact to **write a function that converts a character to uppercase**.

- **charToUpperCase ("a")** returns "A".
- **charToUpperCase ("b")** returns "B".
- **charToUpperCase ("Y")** returns "Y".
- **charToUpperCase ("!")** returns "!".

A	B	C	D	E	...	Z
65	66	67	68	69	...	90



a	b	c	d	e	...	z
97	98	99	100	101	...	122



If the input character is not a lowercase letter between "a" and "z", we simply leave it as-is.

To demonstrate how we can do this for "d", we can first compute its offset from "a" using character codes. That is, **"d".charCodeAt (0) - "a".charCodeAt (0)** tells us "d" is 3 letters above "a".

Note that upper "D" is also 3 letters above upper "A", meaning we can compute the character code of "D" by the expression **"A".charCodeAt (0) + 3**, which turns out to be 68. Lastly, we convert 68 to its character equivalent via **String.fromCharCode (68)**, which yields "D", and we're done! ☺

Below is an implementation of `charToUpperCase`, using the method described above. Note we use `ch >= "a" && ch <= "z"` to check if the input is a lowercase letter ("a", "b", "c", ..., "z").

```
function charToUpperCase(ch) {
  // check if input is a lowercase alphabet letter (a, b, c, ..., z)
  if (ch >= "a" && ch <= "z") {
    let offset = ch.charCodeAt(0) - "a".charCodeAt(0);
    let upperCode = "A".charCodeAt(0) + offset;
    ch = String.fromCharCode(upperCode);
  }
  return ch; // if not a lowercase letter, character is kept as-is
}
```

Alternatively, you can use the fact that the **ASCII code of each uppercase letter is precisely 32 plus the ASCII code of the corresponding lowercase letter**. It's almost like Fahrenheit to Celsius, I think. For instance, "a" and "A" have codes 65 and 97, respectively, and "z" and "Z" have codes 90 and 122.

```
const LOWER_TO_UPPER = "A".charCodeAt(0) - "a".charCodeAt(0); // is 32

function charToUpperCase(ch) {
  if (ch >= "a" && ch <= "z") { // is lowercase letter
    ch = String.fromCharCode(ch.charCodeAt(0) + LOWER_TO_UPPER);
  }
  return ch;
}
```

String to Upper Case

Lastly, we pretend the real `toUpperCase` method doesn't exist, and write a version that returns the same result. This time, **we accept a string of any length as input**, and return an uppercase version.

- `toUpperCase("The names Bond, James Bond.")` returns `"THE NAMES BOND, JAMES BOND."`
- `toUpperCase("oh no! the band is out on the field!!")` returns `"OH NO! THE BAND IS OUT ON THE FIELD"`. This is from the legendary radio call of the 1982 Big Game.
- `toUpperCase("Harry, did you put your name in the Goblet of Fire, Dumbledore said calmly")` returns `"HARRY, DID YOU PUT YOUR NAME IN THE GOBLET OF FIRE, DUMBLEDORE SAID CALMLY"`.

Using the helper `charToUpperCase` above, the implementation of `toUpperCase` is quite gentle. We loop over each character in the input `str`, convert it to uppercase, and add it to a new running string.

```
function toUpperCase(str) {
  let result = "";
  for (let i = 0; i < str.length; i++) {
    let ch = str.charAt(i);
    result += charToUpperCase(ch);
  }
  return result;
}
```