

Lecture 7: Strings in JavaScript

Written by Ben Yan, from materials by Jerry Cain

JavaScript Strings

A string is an ordered collection of characters, and we can define a string using double quotations in JavaScript, i.e. `let str = "hello world"`. Each character has an index position starting from 0, i.e.

h	e	l	l	o		w	o	r	l	d
0	1	2	3	4	5	6	7	8	9	10

You can ascertain the number of characters in a string via its `length` property, i.e. `str.length`. For the string above, `str.length` would evaluate to **11**, which includes the space character " ".

Here's a circus of string methods you may find useful and/or entertaining, preferably the latter.

charAt(index)	Returns the character at the specified index of the string. You can alternatively do <code>str[index]</code> , with square brackets.
substring(start, end)	Returns the substring between indices start (inclusive) and end (exclusive).
indexOf(substr) or indexOf(substr, start)	Returns the index where the first instance of substr appears in the string, and -1 if the substring never appears. An optional 2 nd parameter start specifies the index to begin searching from.
lastIndexOf(substr)	Similar to above, but it searches backwards from the end.

For the string `str` above ("hello world"), we'd have `str.substring(1,4) => "ell"`, `str.indexOf("l") => 2`, `str.charAt(2) => "l"`, and `str.lastIndexOf("l") => 9`.

You can compare strings lexicographically—that is, by dictionary ordering—using operators such as `==`, `<`, `>`, etc. For instance, `"black" < "pink"` and `"red" !== "RED"` both evaluate to **true**.

A very useful operation is **concatenation**, which entails combining two or more strings end to end, with no intervening characters, e.g., `"hollow" + "purple"` produces the new string `"hollowpurple"`.

Reversing a String

Here's a petite function that accepts an input string `str`, and returns a string that's the reverse of the original. For instance, `reverse("stressed")` returns `"desserts"`, which I guess makes sense ☺.

```
function reverse() {
  let result = "";
  for (let i = str.length - 1; i >= 0; i--) { // iterates in reverse
    result += str.charAt(i);
  }
  return result;
}
```

Note that we build up a **new, initially empty string**, as we can't modify existing strings. This is a pattern that you'll see throughout string tasks: stitching a new string from an old one, character by character.

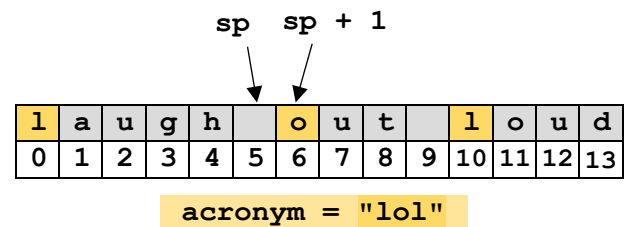
Acronyms

An *acronym* is a word formed by taking the first letter of each word in a sequence, as in:

- `acronym("National Aeronautics and Space Administration")` returns `"NASA"`
- `acronym("not in my back yard")` returns `"nimby"`
- `acronym("laugh out loud")` returns `"lol"`
- `acronym("oll korrekt")` returns `"ok"`, yes this is the origin 🍑

Our task is to implement an acronym-generating function `acronym(str)`. The first approach I can think of searches for spaces in a string, and extracts the character that immediately follows each space.

```
function acronym(str) {
  let result = str.charAt(0);
  let sp = str.indexOf(" ");
  while (sp !== -1) {
    result += str.charAt(sp + 1);
    sp = str.indexOf(" ", sp + 1);
  }
  return result;
}
```



Note that, at the beginning, we make sure to add the very first character, `str.charAt(0)`, as the only part of the acronym that does not follow a space. Furthermore, while this handles acronyms where each word is separated by a space, e.g., `"lol"`, it fails for acronyms such as `"scuba"` (*self-contained underwater breathing apparatus*), where some words are separated by hyphens instead of spaces.

Acronyms: Take II

Our second approach will correctly handle `scuba`, as well as strings with leading, trailing or multiples spaces between words, by identifying each word composed of a **sequential run of alphabet-only letters**.

- `acronym("self-contained underwater breathing apparatus")` returns `"scuba"`
- `acronym("ben! bin-yan")` returns `"bby"`
- `acronym("random-access memory")` returns `"ram"`

Key Idea: We keep track of a Boolean `inWord`. This indicates whether the character we're looking at is part of a word or not. Whenever `false` switches to `true`, we include this letter as part of the acronym!

acronym = "bby"

str	b	e	n	!		b	i	n	-	y	a	n
index	0	1	2	3	4	5	6	7	8	9	10	
inWord	T	T	T	F	F	T	T	T	F	T	T	T

First, we write a helper function that returns `true` only if the input is a valid alphabet letter. Note that, by the lexical ordering performed by comparison operators on strings, `ch >= "A" && ch <= "Z"` ensures `ch` is among letters `"A"`, `"B"`, `"C"`, ..., `"Y"`, `"Z"`, and the same applies for lowercase letters!

```
function isLetter(ch) {
  return ch.length === 1 && ((ch >= "A" && ch <= "Z") ||
    (ch >= "a" && ch <= "z"));
}
```

Next, we implement **acronym** itself! Pay attention to when **inWord** transitions from **false** to **true**.

```
function acronym(str) {
  let result = "";
  let inWord = false;
  for (let i = 0; i < str.length; i++) {
    let ch = str.charAt(i);

    if (isLetter(ch)) {
      // checks if first letter in word
      if (!inWord) result += ch;
      inWord = true;
    } else {
      inWord = false;
    }
  }

  return result;
}
```

str	b	e	n	!		b	i	n	-	y	a	n
index	0	1	2	3	4	5	6	7	8	9	10	
inWord	T	T	T	F	F	T	T	T	F	T	T	T

acronym = "bby"

Note that **inWord** is initialized to **False** to ensure we include the first letter of the starting word!

Hello World, but in JSGraphics

Revisiting your favorite library of **JSGraphics** for a moment, you can display a string graphically using the **GLabel** class. The program below etches the string "**hello, world**" on the graphics window.

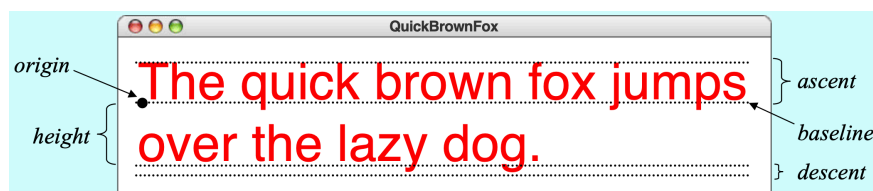
```
const GWINDOW_WIDTH = 500;
const GWINDOW_HEIGHT = 200;

function HelloWorld() {
  let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
  let label = GLabel("hello, world", 100, 75);
  label.setFont("36px Helvetica");
  label.setColor("Red");
  gw.add(label);
}
```



The function to create a label is **GLabel(text, x, y)**, and you can set the displayed font with **label.setFont(font)**, where **font** is a CSS fragment such as "**italic 36px Helvetica**", "**24px 'Times New Roman'**", and others described in our JavaScript textbook, pp. 129-131.

Note: A **GLabel**'s origin (**x, y**) differs slightly from other **GObjects**, as illustrated above, and relies on the following concepts. The **baseline** is the imaginary line on which the characters rest, and the **origin is the point on the baseline at which the label begins**, i.e. approximately the bottom-left corner of the first line.



Vowels and Consonants

As a precursor to the **Pig Latin** program, which scintillates the final third of today's lecture, we'll compose a duet of string functions related to vowels. The first is **isEnglishVowel**, which returns **true** if the input character (i.e. a length-one string) is either an uppercase or lowercase vowel.

```
const VOWELS = "AEIOUaeiou"; // excludes "y", sorry linguists!

function isEnglishVowel(ch) {
  return ch.length === 1 && VOWELS.indexOf(ch) !== -1;
}
```

The second function relies on the first, and, given a string **word**, returns the index of the first vowel that appears in the word. If no vowel appears (e.g., "**syzygy**", one of my favorite words), we return **-1**.

```
function findFirstVowel(word) {
  for (let i = 0; i < word.length; i++) {
    if (isEnglishVowel(word.charAt(i))) {
      return i;
    }
  }
  return -1;
}
```

For instance, **findFirstVowel("apple")** returns **0**, and **findFirstVowel("scram")** returns **3**.

Pig Latin Duolingo

For this whimsical exercise, we introduce the Pig Latin language (more of a playground game, really).

There are 3 rules that enable the conversion of an English word to a Pig Latin one. Are you ready? ☺

1. If the word begins with a consonant, we move the initial consonant string (i.e. the part before the first vowel) to the end of the word, and add the suffix "**ay**". For instance, **scram** -> **amscray**.
2. If the word begins with a vowel, we simply add the suffix "**way**" to alchemize its pig Latin equivalent. For instance, **apple** -> **appleway**, and **orange** -> **orangeway**.
3. If the word contains no vowel at all, we leave the original word unchanged, e.g., **cwm** -> **cwm**.

Here's the pseudocode for a function **wordToPigLatin** which accomplishes the above translation.

```
function wordToPigLatin(word) {
  Find the first vowel in the word.
  If there are no vowels, return the original word unchanged.
  If the vowel appears in the first position, return the word concatenated with "way".
  Otherwise, divide the string into 2 parts: head and tail, before and after the vowel, respectively.
  Return the result of concatenating the tail, the head, and the string "ay".
}
```

The translation (!!) of this pseudocode into pure, beautiful, flowing JavaScript is written below.

```
function wordToPigLatin(word) {
  let vp = findFirstVowel(word);

  if (vp === -1) { // no vowel in word at all
    return word;
  } else if (vp === 0) { // word begins with a vowel
    return word + "way";
  } else { // word begins with a consonant
    let head = word.substring(0, vp); // before the first vowel
    let tail = word.substring(vp); // after and including this vowel
    return tail + head + "ay";
  }
}
```

Note that **vp** is the **index** of the first vowel, and **.substring(start, end)** excludes the **end** index.

Pig Latin: Full Sentence Translation

We expand upon **wordToPigLatin** above to translate not just individual words, but entire sentences to Pig Latin. That green Duolingo Owl would be proud 😊, even if I just lost my streak in French oops.

Our new function, **toPigLatin(str)**, should detect each English word in **str**, and translate it, e.g.,

- **toPigLatin("stout plunder lover")** returns "outstay underplay overlay"
- **toPigLatin("hello there!!")** returns "ellohay erethay!!"
- **toPigLatin("i ... am iron man")** returns "iway ... amway ironway anmay"

We define a word as a sequential run of alphabet letters, and use a similar approach to **Acronyms: Take II** to parse each word. That is, as we iterate over each character in **str**, we keep track of an **inWord** boolean that's **true** only if we're currently in a word, and **false** otherwise. We also maintain a **start** souvenir that recalls the index of the first character of the word we're currently in, and is **-1** if we're not in one. That way, **when a word ends, we'll remember where it started**, and can thus extract the word!

An example is shown below for input **"stout plunder lover"**. Notice how a new word begins when **inWord** goes from **false** to **true**, and the word ends when **inWord** transitions back to **false**.

str		s	t	o	u	t		p	l	u	n	d	e	r		l	o	v	e	r
index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
inWord	F	T	T	T	T	T	F	T	T	T	T	T	T	F	T	T	T	T	T	T
start	-1	0	0	0	0	0	-1	6	6	6	6	6	6	-1	14	14	14	14	14	14

Here's a timeline of how each of the three words, **stout**, **plunder**, **lover**, gets parsed out of the sentence.

- When **index = 5**, **start = 0**, so **str.substr(start, index)** isolates **"stout"**.
- When **index = 13**, **start = 6**, so **str.substr(start, index)** isolates **"plunder"**.
- When the loop is over, **start = 14**, so **str.substr(start,)** lastly isolates **"lover"**.

We can use **wordToPigLatin** to translate each word 😊, which is what the pseudocode below exploits.

```

function toPigLatin(str) {
  Initialize a variable called result to hold the growing string
  for (each character position in str) {
    if (the current character is a letter) { // character is a part of a word
      if (we're not yet scanning a word) Remember the start of this word.

      } else { // is a separator character, e.g., space, punctuation
        if (we were scanning a word) {
          Extract word using the remembered start, and the current end position.
          Call wordToPigLatin to translate the word.
          Append the translated word to the result variable
        }
        Append this separator character to the result variable.
      }
    }
  }
  if (we're still scanning a word at the end) {
    Call wordToPigLatin and append the translated word to result.
  }
  Return the result string, and voila, we're done!
}

```

And lastly, here's the translation of the above pseudocode into a full-fledged JavaScript implementation.

```

function toPigLatin(str) {
  let result = "";
  let start = -1;
  for (let i = 0; i < str.length; i++) {
    let ch = str.charAt(i);

    if (isLetter(ch)) { // character is part of a word
      if (start === -1) {
        start = i;
      }
    } else { // is a separator character, e.g., space, punctuation
      if (start >= 0) {
        let currWord = str.substring(start, i);
        result += wordToPigLatin(currWord);
      }
      start = -1;
      result += ch;
    }
  }
  if (start >= 0) { // if still scanning a word at the end
    result += wordToPigLatin(str.substring(start));
  }
  return result;
}

```

Note that when **ch** is not an alphabet letter, we add it to the **result** unchanged, i.e. **result += ch**, so that all the spacing, punctuation, etc, is kept intact between the transfigured words.