

# Strings in JavaScript

Jerry Cain

CS 106AX

October 11, 2023

*slides leveraged from those constructed by Eric Roberts*

# Strings in JavaScript

# Using Methods in the `String` Class

- JavaScript defines many useful methods to operate on strings. Before trying to use those methods individually, it is important to understand how those methods work at a more general level.
- Because strings are objects, JavaScript uses the receiver syntax to invoke string methods. So, if `str` is a string, you invoke the *name* method using `str.name(arguments)`.
- **None** of the methods in JavaScript's `String` class change the value of the string used as the receiver. Each method returns a **new** string reflecting the outcome of the method's execution.
- Classes that prohibit clients from changing an object's state are said to be *immutable*. Immutable types have many advantages and play an important role in programming.

# Selecting Characters from a String

- Conceptually, a string is an ordered collection of characters.
- In JavaScript, the character positions in a string are identified by an *index* that begins at 0 and extends up to one less than the length of the string. For example, the characters in the string "hello, world" are arranged like this:

<b>h</b>	<b>e</b>	<b>l</b>	<b>l</b>	<b>o</b>	<b>,</b>		<b>w</b>	<b>o</b>	<b>r</b>	<b>l</b>	<b>d</b>
0	1	2	3	4	5	6	7	8	9	10	11

- You can obtain the number of characters by checking the **length** property, as in `str.length`.
- You can select an individual character by calling `charAt(k)`, where  $k$  is the index of the desired character. The expression

`str.charAt(0);`

returns the one-character string "h" that appears at index 0.

# Concatenation

- One of the most useful operations available for strings is *concatenation*, which consists of combining two strings end to end with no intervening characters.
- As you know from earlier in the quarter, concatenation is built into JavaScript in the form of the + operator.
- It is also important to recall that JavaScript interprets the + operator as concatenation if one or both operands are strings. If both operands are numbers, the + operator signifies traditional addition.

# Extracting Substrings

- The `substring` method makes it possible to extract a piece of a larger string by providing index numbers that determine the extent of the substring.
- The general form of the `substring` call is

```
str.substring(p1, p2);
```

where `p1` is the first index position in the desired substring and `p2` is the index position immediately following the last position in the substring.

- As an example, if you wanted to select the substring "ello" from a string variable `str` containing "hello, world" you would make the following call:

```
str.substring(1, 4);
```

# Comparing Strings

- JavaScript allows you to call the standard relational operators to compare the values of two strings in a natural way. For example, if `s1` and `s2` are strings, the expression

```
s1 === s2
```

is `true` if the strings `s1` and `s2` contain the same characters.

- String comparisons involving the operators `<`, `<=`, `>`, and `>=` are implemented in a fashion like traditional alphabetic ordering: if the first characters match, the comparison operator checks the second characters, and so on.
- Characters are compared numerically using their Unicode values. For example, `"cat" > "CAT"` because the character code for `"c"` (99) is greater than the code for `"C"` (67). This style of comparison is called *lexicographic comparison*.
- We'll revisit character encodings and Unicode next week.

# Searching in a String

- The `indexOf` method takes a string and returns the index within the receiver at which the first instance of that string begins. If the string is not found, `indexOf` returns `-1`. For example, if `str` contains the string `"hello, world"`:

```
str.indexOf("h")    returns 0
str.indexOf("o")    returns 4
str.indexOf("ell")  returns 1
str.indexOf("x")    returns -1
```

- The `indexOf` method takes an optional second argument that indicates the starting position for the search. Thus:

```
str.indexOf("o", 5) returns 8
```

- The `lastIndexOf` method works similarly except that it searches backward from the end of the receiving string.



# Other Methods in the `String` Class

**`String.fromCharCode (code)`**

Returns the one-character string whose Unicode value is *code*.

**`charAt (index)`**

Returns the Unicode value of the character at the specified index.

**`toLowerCase ()`**

Returns a copy of this string converted to lower case.

**`toUpperCase ()`**

Returns a copy of this string converted to upper case.

**`startsWith (prefix)`**

Returns **true** if this string starts with *prefix*.

**`endsWith (suffix)`**

Returns **true** if this string ends with *suffix*.

**`trim ()`**

Returns a copy of this string with leading and trailing spaces removed.

# Simple String Idioms

When you work with strings, there are two idiomatic patterns that are particularly important:

1. Iterating through the characters in a string.

```
for (let i = 0; i < str.length; i++) {  
  let ch = str.charAt(i);  
  ... code to process each character in turn ...  
}
```

2. Growing a new string character by character.

```
let result = "";  
for (whatever limits are appropriate to the application) {  
  ... code to determine the next character to be added ...  
  result += ch;  
}
```

# Reversing a String

```
reverse("stressed");
```

```
function reverse(str) {
```

```
  let result = "";
```

```
  for (let i = str.length - 1; i >= 0; i--) {
```

```
    result += str.charAt(i);
```

```
  }
```

```
  return result;
```

```
}
```

str

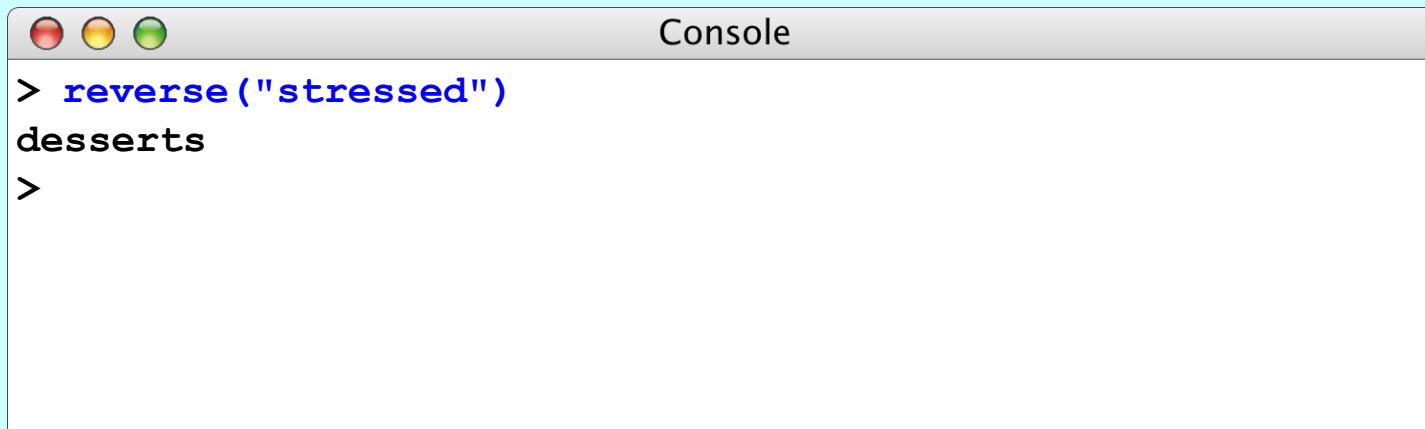
"stressed"

result

"desserts"

i

-1



The screenshot shows a console window titled "Console" with three window control buttons (red, yellow, green) in the top-left corner. The console contains the following text:

```
> reverse("stressed")  
desserts  
>
```

# String Calisthenics

Let's review some `String` methods before continuing:

- ✓ `"AEIOUaeiou".length` 10
- ✓ `"ABCDEFGFG".charAt(6)` "G"
- ✓ `"Harry Potter".indexOf("a")` 1
- ✓ `"Harry Potter".indexOf("a", 6)` -1
- ✓ `"Harry Potter".lastIndexOf("rr")` 2
- ✓ `"bumfuzzle".substring(3, 7)` "fuzz"
- ✓ `"cabotage".substring(1, 1)` ""
- ✓ `"agelast".substring(3)` "last"

# Generating Acronyms

- An *acronym* is a word formed by taking the first letter of each word in a sequence, as in
  - "North American Free Trade Agreement" → "NAFTA"
  - "not in my back yard" → "nimby"
  - "self-contained underwater breathing apparatus" → "scuba"
- The text describes and implements two versions of a function `acronym(str)` that generates an acronym for `str`:
  - The first version searches for spaces in the string and includes the following character in the acronym. This version, however, fails for acronyms like *scuba*, in which some of the words are separated by hyphens rather than spaces.
  - The second version looks at every character and keeps track of whether the algorithm is scanning a word formed composed of sequential letters. This version correctly handles *scuba* as well as strings that have leading, trailing, or multiple spaces.

# acronym , Take I

```
acronym("not in my back yard")  
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;  
}
```

str

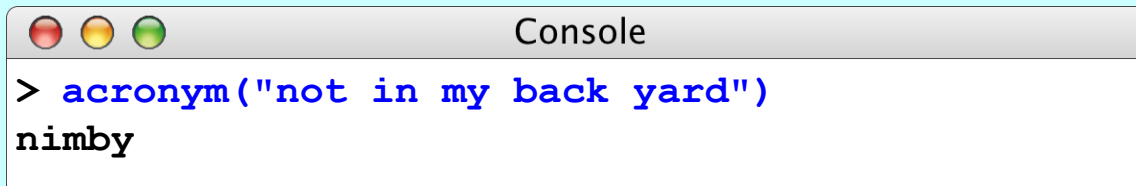
"not in my back yard"

result

"nimby"

sp

-1



```
Console  
> acronym("not in my back yard")  
nimby
```

# acronym, Take II

```
acronym("In My Humble Opinion")
```

```
function acronym(str) {
```

```
  const ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

```
  function isLetter(ch) {
```

```
    return ch.length === 1 &&
```

```
      ALPHABET.indexOf(ch.toUpperCase()) !== -1;
```

```
  }
```

-1

"IMHO"

20

true

"n"

```
Console
> acronym("In My Humble Opinion")
IMHO
```

# Translating Pig Latin to English

Section 7.4 works through the design and implementation of a program to convert a sentence from English to Pig Latin. In this dialect, the Pig Latin version of a word is formed by applying the following rules:

1. If the word begins with a consonant, the `wordToPigLatin` function moves the initial consonant string to the end of the word and then adds the suffix *ay*, as follows:

*scram* → *amscray*

2. If the word begins with a vowel, `wordToPigLatin` generates the Pig Latin version simply by adding the suffix *way*, like this:

*apple* → *appleway*

3. If the word contains no vowels at all, `wordToPigLatin` returns the original word unchanged.



# Translating Pig Latin to English

"stout plunder lover"

<b>i</b>		0	1	2	3	4	5	6		12	13	14		18
<b>inWord</b>	F	T	T	T	T	T	F	T		T	F	T		T
<b>start</b>	-1	0	0	0	0	0	-1	6		6	-1	14		14

- **inWord** is **true** if and only if we're in a word, and **start** is the index of the first character of the word we're currently in (or **-1** if we're not in a word).
- **inWord** is now **true** and **start** is set equal to **0**. We set assign the value of **i** to **start** at the same time **inWord** is transitioning from **false** to **true**, so we can remember where the current word of interest begins.
- This is an interesting transition, since the current word we're in is just now ending. We can isolate the word by calling **str.substring(start, i)**, where **str** is assumed to be the entire sentence or fragment to be translated.
  - Right now, **str.substring(start, i)** produces "**stout**".
  - And now, **str.substring(start, i)** produces "**plunder**".

# Pseudocode for the Pig Latin Program

```
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) {  
            if (we're not yet scanning a word) Remember the start of this word.  
        } else {  
            if (we were scanning a word) {  
                Call wordToPigLatin to translate the word.  
                Append the translated word to the result variable.  
            }  
            Append the separator character to the result variable.  
        }  
    }  
    if (we're still scanning a word) {  
        Call wordToPigLatin and append the translated word to result.  
    }  
}
```

```
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".  
    Divide the string into two parts (head and tail) before the vowel.  
    Return the result of concatenating the tail, the head, and the string "ay".  
}
```

# Simulating the PigLatin Program

```
toPigLatin("this is pig latin")
```

```
function toPigLatin(str) {
```

```
function wordToPigLatin(word) {
```

```
let vp = findFirstVowel(word);
```

```
if (vp === -1) {
```

```
return word;
```

```
} else if (vp === 0) {
```

```
return word + "way";
```

```
} else {
```

```
let head = word.substring(0, vp);
```

```
let tail = word.substring(vp);
```

```
return tail + head + "ay";
```

```
}
```

```
}
```

"atinlay"

word

vp

head

tail

"isthay isway igpay atinlay" "

"atin"

```
Console
> toPigLatin("this is pig latin")
isthay isway igpay atinlay
```

# The GLabel Class

You can display a string in the graphics window using the `GLabel` class, as illustrated by the following function that displays the string `"hello, world"` on the graphics window:

```
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);  
}
```



# Operations on the `GLabel` Class

## Function to create a `GLabel`

`GLabel` (*text*, *x*, *y*)

Creates a label containing the specified text that begins at the point (*x*, *y*).

## Methods specific to the `GLabel` class

*label*.`setFont` (*font*)

Sets the font used to display the label as specified by the font string.

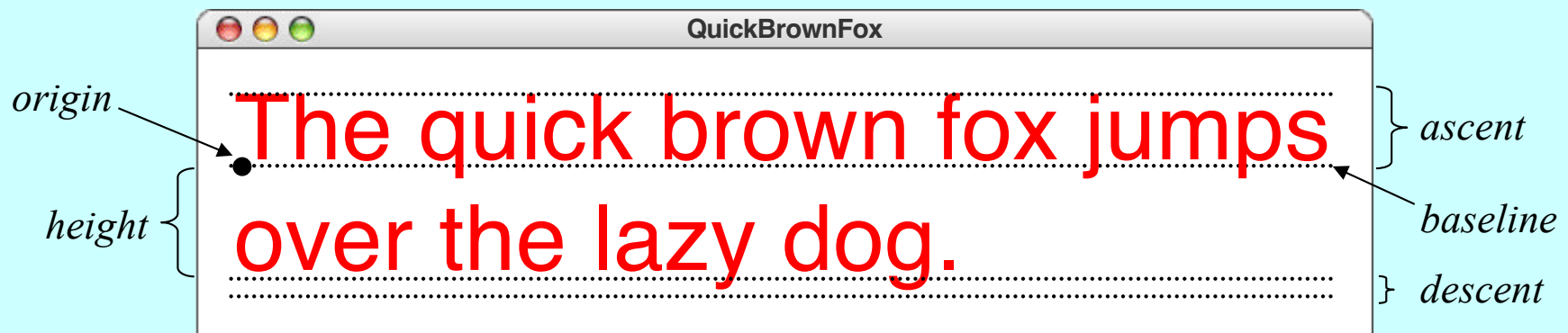
The font is specified as a CSS fragment, the details of which are described in the JavaScript textbook, pp. 129-131.

Examples of legal font strings:

- `"italic 36px Helvetica"`
- `"24px 'Times New Roman'"`
- `"bold 14px Courier, 'Courier New', Monaco"`
- `"oblique bold 44px 'Lucida Blackletter', serif"`

# The Geometry of the `GLabel` Class

- The `GLabel` class relies on a set of geometrical concepts that are derived from classical typesetting:
  - The *baseline* is the imaginary line on which the characters rest.
  - The *origin* is the point on the baseline at which the label begins.
  - The *height* of the font is the distance between successive baselines.
  - The *ascent* is the distance characters rise above the baseline.
  - The *descent* is the distance characters drop below the baseline.
- You can use the `getHeight`, `getAscent`, and `getDescent` methods to determine the corresponding property of the font. You can use the `getWidth` method to determine the width of the entire label, which depends on both the font and the text.



The End