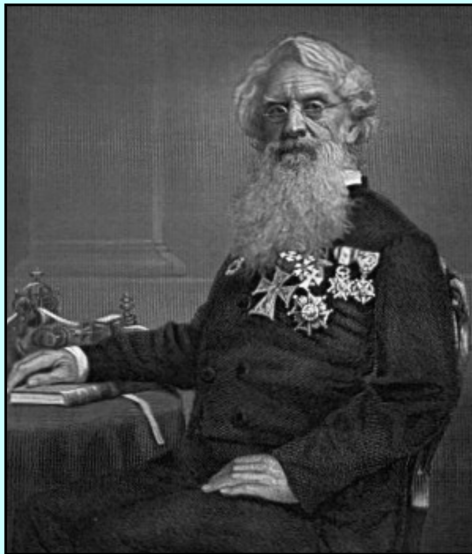
Strings in JavaScript

Jerry Cain
CS 106AJ
October 22, 2018
slides courtesy of Eric Roberts

Once upon a time . . .

Early Character Encodings

- The idea of using codes to represent letters dates from before the time of Herman Hollerith, as described in Chapter 7.
- Most of you are familiar with the work of Samuel F. B. Morse, inventor of the telegraph, who devised a code consisting of dots and dashes. This scheme made it easier to transmit messages and paved the way for later developments in communication.



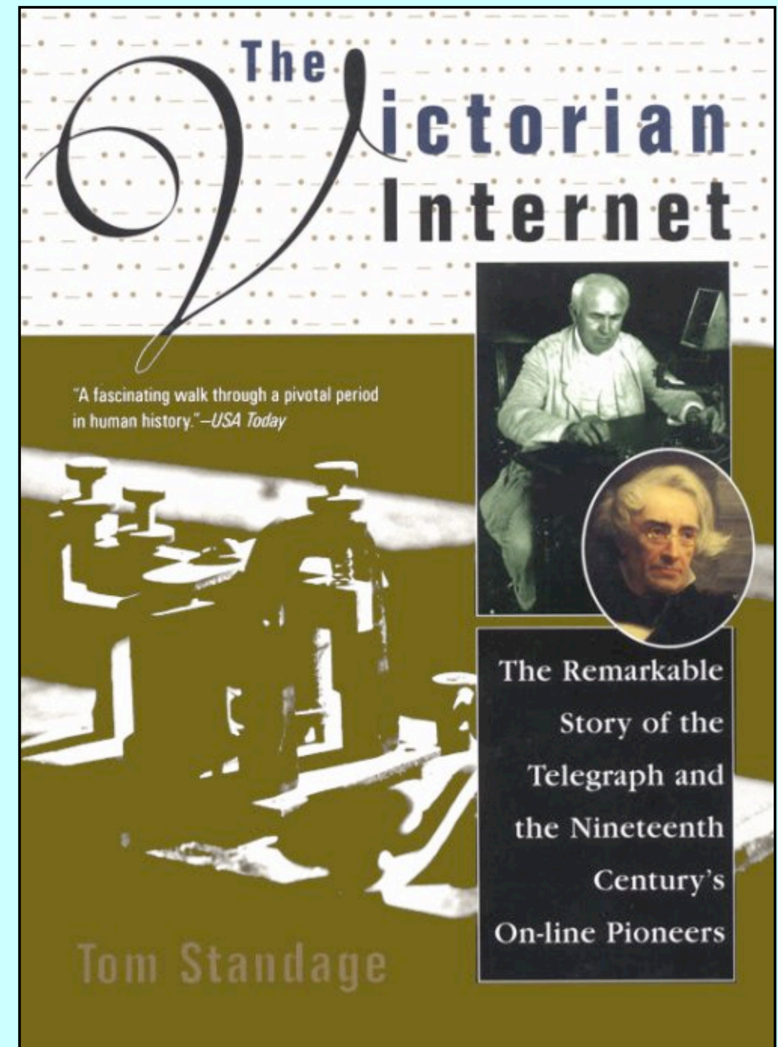
Samuel Morse (1791-1872)

A	· —	J	· — — —	S	· · ·
B	— · · ·	K	— · —	T	—
C	— · — ·	L	· — · ·	U	· · —
D	— · ·	M	— —	V	· · · —
E	·	N	— ·	W	· — —
F	· · — ·	O	— — —	X	— · · —
G	— — ·	P	· — — ·	Y	— · — —
H	· · · ·	Q	— — · —	Z	— — · ·
I	· ·	R	· — ·		

Alphabetic Characters in Morse Code

The Victorian Internet

What you probably don't know is that the invention of the telegraph also gave rise to many of the social phenomena we tend to associate with the modern Internet, including chat rooms, online romances, hackers, and entrepreneurs—all of which are described in Tom Standage's 1998 book, *The Victorian Internet.*



Strings in JavaScript

Review: Strings as an Abstract Idea

- Characters are most often used in programming when they are combined to form collections of consecutive characters called ***strings***.
- As you will discover when you have a chance to look more closely at the internal structure of memory, strings are stored internally as a sequence of characters at consecutive memory addresses.
- The internal representation, however, is really just an implementation detail. For most applications, it is best to think of a string as an abstract conceptual unit rather than as the characters it contains.
- JavaScript emphasizes the abstract view by defining a built-in string type that defines high-level operations on string values.

Using Methods in the `String` Class

- JavaScript defines many useful methods that 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 call string methods. Thus, if `str` is a string, you would 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. What happens instead is that these methods ***return*** a new string on which the desired changes have been performed.
- 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:

h	e	l	l	o	,		w	o	r	l	d
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 only if at least one of the operands is a string. If both operands are numbers, the **+** operator signifies 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 **`"ell"`** 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 similar to 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 ordering***.

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*.

`charCodeAt(`*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.

Exercise: Implementing `endsWith`

- The `startsWith` and `endsWith` methods did not exist in early versions of JavaScript. The text includes an implementation of `startsWith` written as a client function. How would you do the same for `endsWith`?

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 . . .
}
```

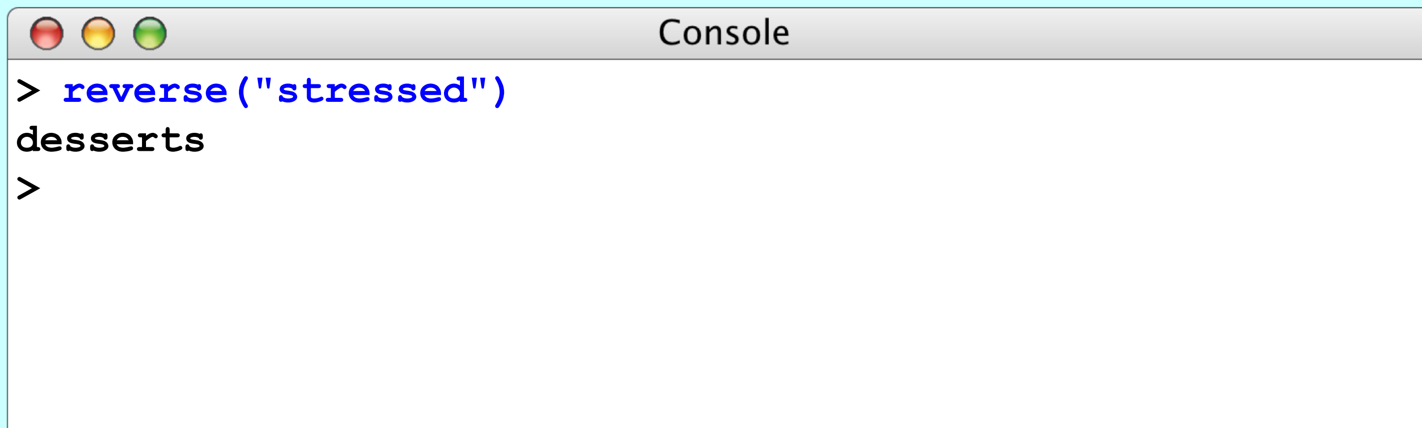
2. Growing a new string character by character.

```
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	result	i
"stressed"	"desserts"	-1



Exercise: Implementing `toUpperCase`

- Suppose that the **`toUpperCase`** method did not exist. How would you implement a **`toUpperCase`** function that returns the same result?

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.

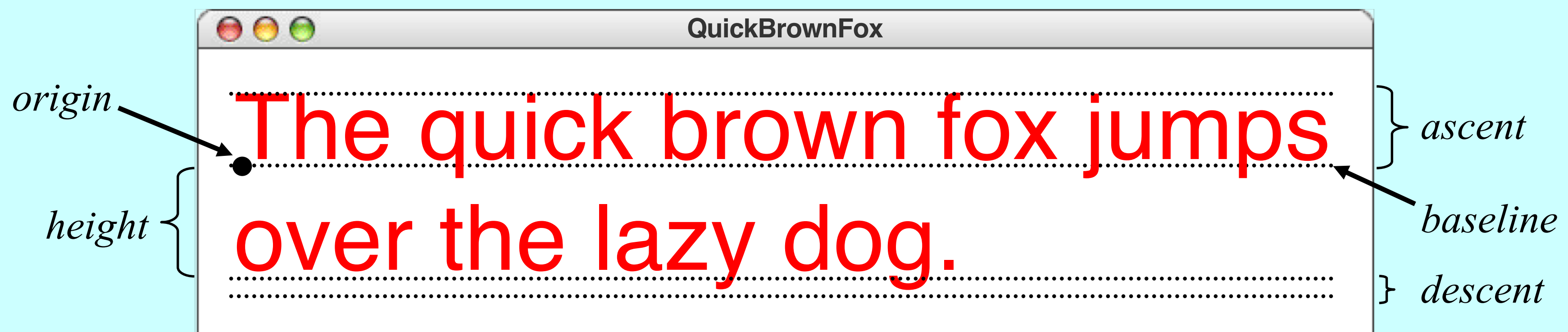
The font is specified as a CSS fragment, the details of which are described in the course reader, pp. 129-131.

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