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 invoke 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. Rather, 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
```

• 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.

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.
• We’ll discuss character encodings more 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”:

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

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

  ```javascript
  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.

- `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.

- `toLowerCase()`
  Returns a copy of this string converted to lower case.

- `toUpperCase()`
  Returns a copy of this string converted to upper case.

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.

   ```javascript
   let result = ";
   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.

   ```javascript
   let result = ";
   for (whatever limits are appropriate to the application) {
       result += ch;
   }
   ```

Reversing a String

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

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

```plaintext
function acronym(str) {
    let result = str.charAt(0);
    let sp = str.indexOf(' ');
    while (sp >= 0) {
        result += str.charAt(sp + 1);
        sp = str.indexOf(' ', sp + 1);
    }
    return result;
}
```

str result sp

"not in my back yard" "nimby" -1

acronym("not in my back yard")

nimby

---

Translating Pig Latin to English

Section 7.5 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:

   ```plaintext
   str --> amcroy
   ```

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

   ```plaintext
   apple --> appleyway
   ```

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

---

Pseudocode for the Pig Latin Program

```plaintext
function wordToPigLatin(word) {
    let head = word.split('').shift();
    let tail = word.split('').join('');
    let result = tail + head + 'ay';
    return result;
}
```

---

Translating Pig Latin to English

"stout plunder lover"

```plaintext
<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>k</th>
<th>l</th>
<th>m</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
</tr>
</tbody>
</table>
```

```plaintext
i
inWord
start
```

```plaintext
j
```

```plaintext
k
```

```plaintext
l
```

```plaintext
m
```

```plaintext
n
```

---

Simulating the PigLatin Program

```plaintext
function PigLatin(word) {
    let result = word.split('').join('');
    let head = word.split('').shift();
    let tail = word.split('').join('');
    result = tail + head + 'ay';
    return result;
}
```

---

acronym("In My Humble Opinion")

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

```plaintext
function isVowel(ch) {
    return ALPHABET.toLowerCase().indexOf(ch.toLowerCase()) !== -1;
}
```

```plaintext
"mHBO" 20 true "n" 13
```

---

"nimby": Take II

```plaintext
> acronym("In My Humble Opinion")
"nimby"
```

---

Translating Pig Latin to English

"stout plunder lover"
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:

```javascript
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

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

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

Methods specific to the GLabel class

```javascript
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 quick brown fox jumps over the lazy dog.