web222

WEB222 - Week 3

Suggested Readings

- SpeakingJS, Chapter 12. Strings
- SpeakingJS, Chapter 18. Arrays
- SpeakingJS, Chapter 19. Regular Expressions

Introduction to Objects and Object-Oriented Programming

In languages like C, we are used to thinking about data types separately from the functions that operate upon them. We declare variables to hold data in memory, and call functions passing them variables as arguments to operate on their values.

In object-oriented languages like JavaScript, we are able to combine data and functionality into higher order types, which both contain data and allow us to work with that data. In other words, we can pass data around in a program, and all the functionality that works on that data travels with it.

Let's consider this idea by looking at strings in C vs. JavaScript. In C a string is a null terminated (`\0`) array of `char` elements, for example:

```
const char name1[31] = "My name is Arnold";
const char name2[31] = {'M','y',' ','n','a','m','e',' ','i','s',' ','A','r','n','o','l','d','\0'};
```

With C-style strings, we perform operations using standard library functions, for example `string.h` :

```c
#include <string.h>

int main(void)
{
    char str[31];        // declare a string
    ...
    strlen(str);         // find the length of a string str
    strcpy(str2, str);   // copy a string
    strcmp(str2, str);   // compare two strings
    strcat(str, "..."); // concatenate a string with another string
}
```

JavaScript also allows us to work with strings, but because JavaScript is an object-oriented language, a JavaScript `String` is an `Object` with various properties and methods we can use for working with text.

One way to think about `Object`s like `String` is to imagine combining a C-string's data type with the functions that operate on that data. Instead of needing to specify *which* string we want to work with, all functions would operate a particular *instance* of a string. Another way to look at this would be to imagine that the data and the functions for working with that data are combined into one more powerful type. If we could do this in C, we would be able to write code that looked more like this:

```c
String str = "Hello"; // declare a string

int len = str.len;    // get the length of str
str.cmp(str2);        // compare str and str2
str = str.cat("..."); // concatenate "..." onto str
```

In the made-up code above, the *data* (`str`) is attached to functionality that we can call via the `.*` notation. Using `str.*`, we no longer need to indicate to the functions which string to work with: all string functions work on the string data to which they are attached.

This is very much how `String` and other `Object` types work in JavaScript. By combining the string character data and functionality into one type (i.e., a `String`), we can easily create and work with text in our programs.

Also, because we work with strings at a higher level of abstraction (i.e., not as arrays of `char`), JavaScript deals with memory management for us, allowing our strings to grow or shrink at runtime.

JavaScript's `String`

Declaring JavaScript Strings

Here are a few examples of how you can declare a `String` in JavaScript, first using a string literal, followed by a call to the `new` operator and the `String` object's constructor function:

```
/*
 * JavaScript String Literals
 */
var s = 'some text';  // single-quotes
var s1 = "some text"; // double-quotes
var s2 = `some text`; // template literal using back-ticks
var unicode = "中文 español Deutsch English देवनागरी العربية português বাংলা русский 日本語 ਪੰਜਾਬੀ 한국어 தமிழ் עברית" // non-ASCII c

/*
 * JavaScript String Constructor: `new String()` creates a new instance of a String
 */
var s3 = new String("Some Text");
var s4 = new String('Some Text');
```

If we want to convert other types to a `String`, we have a few options:

```
var x = 17;
var s = '' + x;        // concatenate with a string (the empty string)
var s2 = String(x);    // convert to String. Note: the `new` operator is not used here
var s3 = x.toString(); // use a type's .toString() method
```

Whether you use a literal or the constructor function, in all cases you will be able to use the various functionality of the `String` type.

String Properties and Methods

- `s.length` - will tell us the length of the string (UTF-16 code units)
- `s.charAt(1)` - returns the character at the given position (UTF-16 code unit). We can also use `s[1]` and use an index notation to get a particular character from the string.
- `s.concat()` - returns a new string created by concatenating the original with the given arguments.
- `s.includes("tex")` - returns `true` if the search string is found within the string, otherwise `false` if not found.
- `s.startsWith("some")` - returns `true` if the string starts with the given substring, otherwise `false` .
- `s.endsWith("text")` - returns `true` if the string ends with the given substring, otherwise `false` .
- `s.match(regex)` - tries to match a regular expression against the string, returning the matches. See discussion of RegExp below.
- `s.replace(regex, "replacement")` - returns a new string with occurrence(s) of a matched RegExp replaced by the replacement text. See discussion of RegExp below.
- `s.slice(2, 3)` - returns a new string extracted (sliced) from within the original string. A beginning index and (optional) end index mark the position of the slice.
- `s.split()` - returns an Array (see discussion below) of substrings by splitting the original string based on the given separator (`String` or `RegExp`).
- `s.toLowerCase()` - returns a new string with all characters converted to lower case.
- `s.toUpperCase()` - returns a new string with all characters converted to upper case.
- `s.trim()` - returns a new string with leading and trailing whitespace removed.

> JavaScript Version Note: modern JavaScript also supports template literals, also sometimes called template *strings*. Template literals use back-ticks instead of single- or double-quotes, and allow you to interpolate JavaScript expressions. For example:

```
var a = 1;
var s = "The value is " + (1 * 6);
var templateVersion = `The value is ${1*6}`
```

JavaScript's `Array`

An `Array` is an `Object` with various properties and methods we can use for working with lists in JavaScript.

Declaring JavaScript Arrays

Like creating a `String`, we can create an `Array` in JavaScript using either a literal or the `Array` constructor function:

```
var arr = new Array(1, 2, 3); // array constructor
var arr2 = [1, 2, 3]; // array literal
```

Like arrays in C, a JavaScript `Array` has a length, and items contained within it can be accessed via an index:

```
var arr = [1, 2, 3];
var len = arr.length; // len is 3
var item0 = arr[0]; // item0 is 1
```

Unlike languages such as C, a JavaScript `Array` can contain any type of data, including mixed types:

```
var list = [0, "1", "two", true];
```

JavaScript `Array`s can also contain holes (i.e., be missing certain elements), change size dynamically at runtime, and we don't need to specify an initial size:

```
var arr = [];  // empty array
arr[5] = 56;   // element 5 now contains 56, and arr's length is now 6
```

> NOTE: a JavaScript `Array` is really a **map**, which is a data structure that associates values with unique keys (often called a key-value pair). JavaScript arrays are a special kind of map that uses numbers for the keys, which makes them look and behave very much like arrays in other languages. We will encounter this **map** structure again when we look at how to create `Object`s.

`Array` Properties and Methods

- `arr.length` - a property that tells us the number of elements in the array.

Methods that modify the original array

- `arr.push(element)` - a method to add one (or more) element to the end of the array. Using `push()` modifies the array (increasing its size). You can also use `arr.unshift(element)` to add one (or more) element to the *start* of the array.
- `arr.pop()` - a method to remove the last element in the array and return it. Using `pop()` modifies the array (reducing its size). You can also use `arr.shift()` to remove the *first* element in the array and return it.

Methods that do not modify the original array

- `arr.concat([4, 5], 6)` - returns a new array with the original array joined together with other arrays or values provided.
- `arr.includes(element)` - returns `true` if the array includes the given element, otherwise `false`.
- `arr.indexOf(element)` - returns the index of the given element in the array, if it exists, otherwise `-1` (meaning not found).
- `arr.join("\n")` - returns a string created by joining (concatenating) all elements in the array with the given delimiter (`String`).

Methods for iterating across the elements in an Array

JavaScript's `Array` type also provides a long list of useful methods for working with list data. All of these methods work in roughly the same way:

```
// Define an Array
let list = [1, 2, 3, 4];

// Define a function that you want to call on each element of the array
```

```
function operation(element) {
    // do something with element...
}

// Call the Array method that you want, passing your function operation
list.arrayOperation(operation);
```

JavaScript will call the given function on every element in the array, one after the other. Using these methods, we are able to work with the elements in an Array instead of only being able to do things with the Array itself.

As a simple example, let's copy our `list` Array and add 3 to every element. We'll do it once with a for-loop, and the second time with the `forEach()` method:

```
// Create a new Array that adds 3 to every item in list, using a for-loop
let listCopy = [];

for(let i = 0; i < list.length; i++) {
    let element = list[i];
    element += 3;
    listCopy.push(element);
}
```

Now the same code using the `Array`'s `forEach()` method:

```
let listCopy = [];

list.forEach(function(element) {
    element += 3;
    listCopy.push(element);
});
```

We've been able to get rid of all the indexing code, and with it, the chance for off-by-one errors. We also don't have to write code to get the element out of the list: we just use the variable passed to our function.

These `Array` methods are so powerful that there are often functions that do exactly what we need. For example, we could shorten our code above even further but using the `map()` method. The `map()` method takes one `Array`, and calls a function on every element, creating and returning a new `Array` with those elements:

```
let listCopy = list.map(function(element) {
    return element + 3;
});
```

Here are some of the `Array` methods you should work on learning:

- `arr.forEach()` - calls the provided function on each element in the array.
- `arr.map()` - creates and returns a new array constructed by calling the provided function on each element of the original array.
- `arr.find()` - finds and returns an element from the array which matches a condition you define.
- `arr.filter()` - creates and returns a new array containing only those elements that match a condition you define in your function.
- `arr.every()` - returns `true` if all of the elements in the array meet a condition you define in your function.

There are more `Array` methods you can learn as you progress with JavaScript, but these will get you started.

Iterating over String, Array, and other collections

The most familiar way to iterate over a `String` or `Array` works as you'd expect:

```
var s = 'Hello World!';
for(var i = 0; i < s.length; i++) {
    var char = s.charAt(i);
    console.log(i, char);
    // Prints:
```

```
    // 0, H
    // 1, e
    // 2, l
    // ...
}

var arr = [10, 20, 30, 40];
for(var i = 0; i < arr.length; i++) {
    var elem = arr[i];
    console.log(i, elem);
    // Prints:
    // 0, 10
    // 1, 20
    // 2, 30
    // ...
}
```

The standard `for` loop works, but is not the best we can do. Using a `for` loop is prone to various types of errors: off-by-one errors, for example. It also requires extra code to convert an index counter into an element.

An alternative approach is available in ES6, `for...of`:

```
var s = 'Hello World!';
for(var char of s) {
    console.log(char);
    // Prints:
    // H
    // e
    // l
    // ...
}

var arr = [10, 20, 30, 40];
for(var elem of arr) {
    console.log(elem);
```

```
    // Prints:
    // 10
    // 20
    // 30
    // ...
}
```

Using `for...of` we eliminate the need for a loop counter altogether, which has the added benefit that we'll never under- or over- shoot our collection's element list; we'll always loop across exactly the right number of elements within the given collection.

The `for...of` loop works with all collection types, from `String` to `Array` to `arguments` to `NodeList` (as well as newer collection types like `Map`, `Set`, etc.).

JavaScript's RegExp

A regular expression is a special string that describes a pattern to be used for matching or searching within other strings. They are also known as a *regex* or *regexp*, and in JavaScript we refer to `RegExp` when we mean the built-in `Object` type for creating and working with regular expressions.

You can think of regular expressions as a kind of mini programming language separate from JavaScript. They are not unique to JavaScript, and learning how to write and use them will be helpful in many other programming languages.

Even if you're not familiar with regular expression syntax (it takes some time to master), you've probably encountered similar ideas with wildcards. Consider the following Unix command:

```
ls *.txt
```

Here we ask for a listing of all files whose filename *ends with* the extension `.txt`. The `*` has a special meaning: *any character, and any number of characters*. Both `a.txt` and `file123.txt` would be matched against this pattern, since both end with `.txt`.

Regular expressions take the idea of defining patterns using characters like `*`, and extend it into a more powerful pattern matching language. Here's an example of a regular expression that could be used to match both common spellings of the word `"colour"` and `"color"`:

```
colou?r
```

The `?` means that the preceding character `u` is optional (it may or may not be there). Here's another example regular expression that could be used to match a string that starts with `id-` followed by 1, 2, or 3 digits (`id-1`, `id-12`, or `id-999`):

```
id-\d{1,3}
```

The `\d` means a digit (0-9) and the `{1,3}` portion means *at least one, and at most three*. Together we get *at least one digit, and at most three digits*.

There are many special characters to learn with regular expressions, which we'll slowly introduce.

Declaring JavaScript `RegExp`

Like `String` or `Array`, we can declare a `RegExp` using either a literal or the `RegExp` constructor:

```
var regex = /colou?r/;              // regex literal uses /.../
var regex2 = new RegExp("colou?r");
```

Regular expressions can also have advanced search flags, which indicate how the search is supposed to be performed. These flags include `g` (globally match all occurrences vs. only matching once), `i` (ignore case when matching), and `m` (match across line breaks, multi-line matching) among others.

```
var regex = /pattern/gi;                  // find all matches (global) and ignore case
var regex2 = new RegExp("pattern", "gi"); // same thing using the constructor instead
```

Understanding Regular Expression Patterns

Regular expressions are dense, and often easier to write than to read. It's helpful to use various tools to help you as you experiment with patterns, and try to understand and debug your own regular expressions:

- regexr.com
- Regulex
- regexpal.com

Matching Specific Characters

- `\ ^ $ . * + ? ( ) [ ] { } |` all have special meaning, and if you need to match them, you have to escape them with a leading `\`. For example: `\$` to match a `$`.
- Any other character will match itself. `abc` is a valid regular expression and means *match the letters abc*.
- The `.` means *any character*. For example `a.` would match `ab`, `a3`, or `a"`. If you need to match the `.` itself, make sure you escape it: `.\.` means *a period followed by any character*
- We specify a set of possible characters using `[]`. For example, if we wanted to match any vowel, we might do `[aeiou]`. This says *match any of the letters a, e, i, o, or u* and would match `a` but not `t`. We can also do the opposite, and define a negated set: `[^aeiou]` would match anything that is *not* a vowel. With regular expressions, it can often be easier to define your patterns in terms of what they are not instead of what they are, since so many things are valid vs. a limited set of things that are not. We can also specify a range, `[a-d]` would match any of `a, b, c, d` but not `f, g` or `h`.
- Some sets are so common that we have shorthand notation. Consider the set of single digit numbers, `[0123456789]`. We can instead use `\d` which means the same thing. The inverse is `\D` (capital `D`), and means `[^0123456789]` (i.e., *not one of the digits*). If we wanted to match a number with three digits, we could use `\d\d\d`, which would match `123` or `678` or `000`.

- Another commonly needed pattern is *any letter or number* and is available with `\w`, meaning `[A-Za-z0-9_]` (all upper- and lower-case letters, digits 0 to 9, and the underscore). The inverse is available as `\W` and means `[^A-Za-z0-9_]` (everything *not* in the set of letters, numbers and underscore).

- Often we need to match blank whitespace (spaces, tabs, newlines, etc.). We can do that with `\s`, and the inverse `\S` (anything not a whitespace). For example, suppose we wanted to allow users to enter an id number with or without a space: `\d\d\d\s?\d\d\d` would match both `123456` and `123 456`.

- There are lots of other examples of pre-defined common patterns, such as `\n` (newline), `\r` (carriage return), `\t` (tab). Consult the MDN documentation for character classes to lookup others.

Define Character Matching Repetition

In addition to matching a single character or character class, we can also match sequences of them, and define *how many* times a pattern or match can/must occur. We do this by adding extra information *after* our match pattern.

- `?` is used to indicate that we want to match something *once or none*. For example, if we want to match the word `dog` without an `s`, but also to allow `dogs` (with an `s`), we can do `dogs?`. The `?` follows the pattern (i.e., `s`) that it modifies, and indicates that it is optional.

- `*` is used when we want to match *zero or more* of something. `number \d*` would match `"number "` (no digits), `"number 1"` (one digit), and `"number 1234534123451334466600"`.

- `+` is similar to `*` but means *one or more*. `vroo+m` would match `"vroom"` but also `"vroooooooom"` and `"vroooooooooooooooooooooooooooooooom"`

- We can limit the number of matches to an exact number using `{n}`, which means *match exactly `n` times*. `vroo{3}m` would only match `"vroooom"`. We can further specify that we want a match to happen *match `n` or more times* using `{n,}`, or use `{n,m}` to indicate we want to match *at least `n` times and no more than `m` times: `\w{8,16}` would match 8 to 16 word characters, `"ABCD1234"` or `"zA5YncUI24T_3GHO"`

Define Positional Match Parameters or Alternatives

Normally the patterns we define are used to look *anywhere* within a string. However, sometimes it's important to specify *where* in the string a match is located. For example, we might care that an id number *begins* with some sequence of letters, or that a name doesn't *end* with some set of characters.

- `^` means start looking for the match at the *beginning* of the input string. We could test to see that a string begins with a capital letter like so: `^[A-Z]`.
- Similarly `$` means make sure that the match ends the string. If we wanted to test that string was a filename that ended with a period and a three letter extension, we could use: `\.\w{3}$` (an escaped period, followed by exactly 3 word characters, followed by the end of the string). This would match `"filename.txt"` but not `"filename.txt is a path"`.
- Sometimes we need to specify one of a number of possible alternatives. We do this with `|`, as in `red|green|blue` which would match any of the strings `"red"`, `"green"`, or `"blue"`.

Using `RegExp` with `String`s

So far we've discussed how to declare a `RegExp`, and also some of the basics of defining search patterns. Now we need to look at the different ways to use our regular expression objects to perform matches.

- `RegExp.test(string)` - used to test whether or not the given string matches the pattern described by the regular expression. If a match is made, returns `true`, otherwise `false`. `/id-\d\d\d/.test('id-123')` returns `true`, `/id-\d\d\d/.test('id-13b')` returns `false`.
- `String.match(regexp)` - used to find all matches of the given `RegExp` in the source `String`. These matches are returned as an `Array` of `String`s. For example, `'This sentence has 2 numbers in it, including the number 567'.match(/\d+/g)` will return the `Array` `['2', '567']` (notice the use of the `g` flag to find all matches globally).
- `String.replace(regexp, replacement)` - used to find all matches for the given `RegExp`, and returns a new `String` with those matches replaced by the replacement `String` provided. For example, `'50 , 60,75.'.replace(/\s*,\s*/g, ', ')` would return `'50, 60, 75.'` with all whitespace normalized around the commas.

- `String.split(RegExp)` - used to break the given `String` into an `Array` of sub-strings, dividing them on the `RegExp` pattern. For example, `'one-two--three---four----five-----six'.split(/-+/)` would return `['one', 'two', 'three', 'four', 'five', 'six']`, with elements split on any number of dashes.

There are other methods you can call, and more advanced ways to extract data using RegExp, and you are encouraged to dig deeper into these concepts over time. Thinking about matching in terms of regular expressions takes practice, and often involves inverting your logic to narrow a set of possibilities into something you can define in code.

Practice Exercises

1. Write a function `log` that takes an `Array` of `String`s and displays them on the `console`.
2. An application uses an `Array` as a Stack (LIFO) to keep track of items in a user's shopping history. Every time they browse to an item, you want to `addItemToHistory(item)`. How would you implement this?
3. Write a function `buildArray` that takes two `Number`s, and returns an `Array` filled with all numbers between the given number: `buildArray(5, 10)` should return `[5, 6, 7, 8, 9, 10]`
4. Write a function `addDollars` that takes an `Array` of `Number`s and uses the array's `map()` method to create and return a new `Array` with each element having a `$` added to the front: `addDollars([1, 2, 3, 4])` should return `['$1', '$2', '$3', '$4']`
5. Write a function `tidy` that takes an `Array` of `String`s and uses the array's `map()` method to create and return a new `Array` with each element having all leading/trailing whitespace removed: `tidy([' hello', ' world '])` should return `['hello', 'world']`.
6. Write a function `measure` which takes an `Array` of `String`s and uses the array's `forEach()` method to determine the size of each string in the array, returning the total: `measure(['a', 'bc'])` should return `3`. Bonus: try to rewrite your code using the `Array`'s `reduce()` method.
7. Write a function `whereIsWaldo` that takes an `Array` of `String`s and uses the array's `forEach()` method to create a new `Array` with only the elements that contain the text `"waldo"` or `"Waldo"` somewhere in them: `whereIsWaldo(['Jim Waldorf', 'Lynn Waldon', 'Frank Smith'])` should return `'Jim Waldorf', 'Lynn Waldon']`. Bonus: try to rewrite your code using the `Array`'s `filter()` method.
8. Write a function `checkAges` that takes two arguments: an `Array` of ages (`Number`); and a cut-off age (`Number`). Your function should return `true` if all of the ages in the `Array` are at least as old as the cut-off age: `checkAges([16, 18, 22, 32, 56], 19)` should return

`false` and `checkAges([16, 18, 22, 32, 56], 6)` should return `true`. Bonus: try to rewrite your code using the `Array`'s `every()` method.

9. Write a function `containsBadWord` that takes two arguments: `badWords` (an `Array` of words that can't be used), and `userName` (a `String` entered by the user). Check to see if any of the words in `badWords` are contained within `userName`. Return the `badWord` that was found, or `null` if none are found.
10. A `String` contains a Key/Value pair separated by a `":"`. Using `String` methods, how would you extract the two parts? Make sure you also deal with any extra spaces. For example, all of the following should be considered the same: `"colour: blue"`, `"colour:blue"`, `"colour : blue"`, `"colour: blue "`. Bonus: how could you use a `RegExp` instead?
11. A `String` named `addresses` contains a list of street addresses. Some of the addresses use short forms: `"St."` instead of `"Street"` and `"Rd"` instead of `"Road"`. Using `String` methods, convert all these short forms to their full versions.
12. Room booking codes must take the following form: room number (`1-305`) followed by `-` followed by the month as a number (`1-12`) followed by the day as a number (`1-31`). For example, all of the following are valid: `"1-1-1"`, `"250-10-3"`, `"66-12-12"`. Write a `RegExp` to check whether a room booking code is valid or not, which allows any of the valid forms.
13. Write a function that takes a `String` and checks whether or not it begins with one of `"Mr."`, `"Mrs."`, or `"Ms."`. Return `true` if it does, otherwise `false`. Bonus: try writing your solution using regular `String` methods *and* again as a `RegExp`.
14. Write a function that takes a password `String`, and validates it according to the following rules: must be between 8-32 characters in length; must contain one Capital Letter; must contain one Number; must contain one Symbol (`!@#$%^&*-+{}`). Return `true` if the given password is valid, otherwise `false`.
15. Write a `RegExp` for a Canadian Postal Code, for example `"M3J 3M6"`. Allow spaces or no spaces, capitals or lower case.

A Larger Problem Combining Everything:

You are asked to write JavaScript code to process a `String` which is in the form of a Comma-Separated Values (CSV) formatted data dump of user information. The data might look something like this:

```
0134134,John Smith,555-567-2341,62 inches
0134135   ,    June    Lee    ,  5554126347 ,        149 cm
0134136,       Kim Thomas       , 5324126347, 138cm`
```

Write a series of functions to accomplish the following, building a larger program as you go. You can begin with exercise.js:

1. Split the string into an `Array` of separate rows (i.e., an `Array` with rows separated by `\n`). Bonus: how could we deal with data that includes both Unix (`\n`) and Windows (`\r\n`) line endings?
2. Each row contains information user info: `ID`, `Name`, `Phone Number`, and `Height` info all separated by commas. Split each row into an `Array` with all of its different fields. You need to deal with extra and/or no whitespace between the commas.
3. Get rid of any extra spaces around the `Name` field
4. Using a `RegExp`, extract the Area Code from the `Phone Number` field. All `Phone Number`s are in one of two formats: `"555-555-5555"` or `"5555555555"`.
5. Check if the `Height` field has `"cm"` at the end. If it does, strip that out, convert the number to inches, and turn it into a `String` in the form `"xx inches"`. For example: `"152 cm"` should become `"59 inches"`.
6. After doing all of the above steps, create a new record with `ID`, `Name`, `Area Code`, `Height In Inches` and separate them with commas
7. Combine all these processed records into a new CSV formatted string, with rows separated by `\n`.

A sample solution is provided in solution.js.
