"Basic 13" Review

Solutions for the "Basic 13" algorithm challenges.

Zero Out Negative Numbers

Set negative array values to zero.

```
function setNegsToZero(arr)
{
   for (var idx = 0; idx < arr.length; idx++)
   {
      if (arr[idx] < 0)
      {
         arr[idx] = 0;
      }
   }
   return arr;
}
```

Shift Array Values

Shift array values: drop the first and leave `'0'` at end.

```
function arrShift(arr)
{
   for (var idx = 1; idx < arr.length; idx++)
   {
      arr[idx - 1] = arr[idx];
   }
   arr[arr.length - 1] = 0;
   return arr;
}
```

Swap String for Array Negative Values

Replace any negative array values with `'Dojo'`.

```
function numToStr(arr)
{
   for (var idx = 0; idx < arr.length; idx++)
   {
      if (arr[idx] < 0)
      {
         arr[idx] = "Dojo";
      }
   }
   return arr;
}
```

More About Strings

Of our basic JavaScript data types, strings are our third focus (after *Number* and *Boolean*).

Strings are arrays of characters (more accurately, you can *read* individual characters the same way you read specific values in a numerical array, and these individual values are *strings of length 1*). However, you cannot write individual characters in a string in this same way. Once a string is defined, individual characters can be *referenced* by [ ] but *not changed*. Strings are *immutable*: they can be completely replaced in their entirety, but not changed piecewise. To manipulate string characters, you must split the string into an *array*, make individual changes within that array, then join the array to reform a string.

Below are examples of declaring strings, referencing individual elements, using `String.length`, converting string to array with `String.split`, and converting array back to string with `Array.join`.

```
var funStr = "Emma shreds on her electric cello";
console.log(typeof funStr);              // "string"
var oneChar = funStr[26];                // "c"
console.log(typeof oneChar);             // "string"
```

`String.length` method

```
console.log(funStr.length);              // 33
console.log("".length);                  // 0
```

`String.split` (converts string to array, splitting on the provided parameter)

```
wordArray = funStr.split(" ");           // Note: " " never appears in result:
                                         // [ "Emma", "shreds", "on", "her",
                                         // "electric", "cello" ]
console.log(wordArray [5].split("");     // Split on every letter:
                                         // [ "c", "e", "l", "l", "o" ]
```

`Array.join` (converts array to string, using the provided parameter as a separator)

```
console.log(wordArray.join());           // Note: "," is used by default:
                                         // "Emma,shreds,on,her,electric,cello"
console.log(wordArray.join("-"));        // Param "-" inserted between words:
                                         // "Emma-shreds-on-her-electric-cello"
console.log(wordArray.join(""));         // "Emmashredsonherelectriccello"
```

Challenge: what is displayed by the following? Why?

```
console.log(1 + 2 + "3" + "4" + 5 + 6);
```

Chapter 4 – Strings and Associative Arrays

This chapter explores strings – a special case of the basic array – then associative arrays. By now you should be able to easily complete the "Basic 13" algorithm challenges in less than 2 minutes each.

☐ Remove Blanks

Create a function that, given a string, returns all of that string's contents, but without blanks. If given the string `" Pl  ayTha  tF  u  nkyM  usi  c  "`, return `"PlayThatFunkyMusic"`.

☐ String: Get Digits

Create a JavaScript function that given a string, returns the integer made from the string's digits. Given `"0s1a3y5w7h9a2t4?6!8?0"`, the function should return the number `1357924680`.

☐ Acronyms

Create a function that, given a string, returns the string's acronym (first letters only, capitalized). Given `" there's no free lunch - gotta pay yer way. "`, return `"TNFL-GPYW"`. Given `"Live from New York, it's Saturday Night!"`, return `"LFNYISN"`.

☐ Count Non-Spaces

Accept a string and return the number of non-space characters found in the string. For example, given `"Honey pie, you are driving me crazy"`, return `29` (not `35`).

☐ Remove Shorter Strings

Given a string array and value (length), remove any strings shorter than *length* from the array.

Chapter 4 – Strings and Associative Arrays

Switch/case statements

Think of `SWITCH` statements as a series of `IF` statements, based on a single `value` (a number or string). From the `switch`, execution jumps forward to the `case` that matches (or `default` if no match is found). Execution continues until it hits a `break`, exiting the switch statement. If you omit a `break`, execution continues even into a subsequent `case:`! Here are examples:

```
switch (favoriteLanguageString) {
   case 'JavaScript': console.log("Ah so, we thrive on chaos!"); break;
   case 'Python':     console.log("Parenthesis-haters, unite!"); break;
   case 'PL/I':       console.log("Wha? Who let you in here?");
   default:           console.log("Why don't you choose a different one.");
}
```

Note that if `favoriteLanguageString` has a value of `'PL/I'`, then we would log two messages. After the `console.log`, we would continue onward to the end of the `SWITCH` (since we hit no `break`). Switch statements are not always the right choice, but may prove valuable in some challenges below.

☐ String: Reverse

Implement `reverseString(str)` that, given string, returns that string with characters reversed. Given `"creature"`, return `"erutaerc"`. Tempting as it seems, do not use the built-in `reverse()`!

☐ Remove Even-Length Strings

Build a standalone function to remove strings of *even lengths* from a given array. For array containing `["Nope!","Its","Kris","starting","with","K!","(instead","of","Chris","with","C)",".",]`, change that same array to `["Nope!","Its","Chris","."]`.

☐ Integer to Roman Numerals

Given a positive integer that is less than 4000, return a string containing that value in Roman numeral representation. In this representation, `I` is 1, `V` is 5, `X` is 10, `L` = 50, `C` = 100, `D` = 500, and `M` = 1000. Remember that 4 is IV, 349 is CCCIL and 444 is CDXLIV.

☐ Roman Numerals to Integer

Sept 16, 2014 headline: "*Ancient Computer Found In Roman Shipwreck*". Comprising 30 bronze gears, its wooden frame features 2000 characters. Given a string containing a Roman numeral representation of a positive integer, return the integer. Remember that `III` is `3`, `DCIX` is `609` and `MXDII` is `1492`.

Challenge answered: `console.log(1 + 2 + "3" + "4" + 5 + 6)` will output `"33456"`. Num+num is a num, but num+str or str+num is a str: `1+2==3`. `3+"3"=="33"`. `"334"+5=="3345"`.

Chapter 4 – Strings and Associative Arrays

Getting the hang of strings? Good! This data structure is important in essentially every programming language.

☐ Parens Valid

Create a function that, given an input string `str`, returns a boolean whether parentheses in `str` are valid. Valid sets of parentheses always open before they close, for example. For `"Y(3(p)p(3)r)s"`, return `true`. Given `"N(0(p)3"`, return `false`: not every parenthesis is closed. Given `"N(0)t )0(k"`, return `false`, because the underlined `")"` is premature: there is nothing open for it to close.

☐ Braces Valid

Given a sequence of parentheses, braces and brackets, determine whether it is valid. Example: `"W(a{t}s[o(n{ c}o)m]e )h[e{r}e]!"` => `true`. `"D(i{a}l[ t]o)n{e"` => `false`. `"A(1)s[O(n]0{t) 0}k"` => `false`.

Strings like `"Able was I, ere I saw Elba"` or `"Madam, I'm Adam"` could be considered *palindromes*, because (if we ignore spaces, punctuation and capitalization) the letters are the same when reading from the back to the front.

☐ String: Is Palindrome

Create a function that returns a boolean whether the string is a *strict* palindrome. For `"a x a"` or `"racecar"`, return `true`. Do **not** ignore spaces, punctuation and capitalization: if given `"Dud"` or `"oho!"`, return `false`.

Second: now *do* ignore white space (spaces, tabs, returns), capitalization and punctuation.

☐ Longest Palindrome

For this challenge, we will look not only at the entire string provided, but also at the substrings within it. Return the longest palindromic substring. Given `"what up, daddy-o?"`, return `"dad"`. Given `"uh... not much"`, return `"u"`. *Include spaces* as well (i.e. be strict, as in previous challenge): given `"Yikes! my favorite racecar erupted!"`, return `"e racecar e"`. Strings *longer or shorter* than complete words are OK.

Second: re-solve the above problem, but ignore spaces, tabs, returns, capitalization and punctuation. Given `"Hot puree eruption!"`, return `"tpureeerupt"`.

Chapter 4 – Strings and Associative Arrays

Fast-Finish / Fast-Fail

The idea of *quickly exiting a function if a special case is detected* likely does not seem all that revolutionary. However, this not only simplifies the code, but make its average running time faster as well. Whether to apply them to failure (fast-fail) or success cases will depend on the specifics of the challenge, but in any case they can quickly narrow a problem to the mainline case that remains.

This chapter's challenges focused on *strings*, then *maps / hashes*. These concepts might be useful:

.length *.split* *.join* *.concat* *for...in loops* *switch/case*

☐ Is Word Alphabetical

Nikki, a queen of gentle sarcasm, loves the word *facetiously*. Lance helpfully points out that it is the only known English word that contains *all five vowels* in alphabetical order, and it even has a 'y' on the end! Nikki takes a break from debugging to turn and give him an acid stare – indeed a look that was delivered *arseniously*. Given a string, return whether all contained letters are in *alphabetical order*.

☐ D Gets Jiggy

Write a function that accepts as a parameter a string containing someone's name. Return a string containing the following oh-so-cool greeting: strip off the first letter of the name, capitalize this new word, and add `" to the` [first letter]`!"` Given `"Dylan"`, return `"Ylan to the D!"`

☐ Common Suffix

Lance is writing his opus: Epitome, an epic tome of beat poetry. Always ready for a good rhyme, he constantly seeks words that end with the same letters. Write a function that, when given a word array, returns the largest suffix (word-end) common to *all words* in the array. For inputs `["deforestation", "citation", "conviction", "incarceration"]`, return `"tion"` (not a very creative starting point). If it is `["nice", "ice", "baby"]`, return `""`.

☐ Book Index

Martin is writing his opus: a book of algorithm challenges, set as lyrics to a suite of *a cappella* fugues. Some of those fugueing challenges are less popular than others, so he needs an index. Given a sorted array of pages where a term appears, produce an index string. Consecutive pages should form ranges separated by a hyphen. For `[1,13,14,15,37,38,70]`, return string `"1, 13-15, 37-38, 70"`. Take care to get all the commas and spaces correct: Martin is palpably persnickety about patchy punctuation.

☐ Drop the Mike

Create a standalone function that accepts an input string, removes leading and trailing white space (at beginning and end only), capitalizes the first letter of every word, and returns that string. If original string contains the word `"Mike"` *anywhere*, immediately return `"stunned silence"` instead.

Chapter 4 – Strings and Associative Arrays

Before we dive into a new area, let's remind ourselves of some of the best practices mentioned previously. Make sure to understand the problem thoroughly – ask clarifying questions before rushing to write code. Challenge yourself to think of any special cases your solution might need to handle: can you trust the input data you are given? Note any interesting "corner cases" for later, when you test your code. Restate the problem back to the interviewer, and (again, before you start coding) note a few important test cases along with what output they should produce. *Then* start coding. Once you finish, verify your code using the test cases you identified earlier, perhaps using T-diagrams. OK, onward.

Associative Arrays (Objects)

"Regular" (numerically indexed) arrays are handy. They can contain many values, any of which can be instantly accessed simply by providing the index. Arrays have *order* – indices are numerically arranged. However, sometimes we want more than just a number to describe what is stored in an array cell. If we held `["John", "Watson", "221B Baker Street"]`, we *might* remember that `[0]` meant first name, `[2]` stored the address, etc., but numerical indices are often not descriptive enough.

What if, for each specific cell in our array, instead of associating it with an *index number*, we associated it with a *string* – any string we wanted? This is essentially the *associative array*. Just as we place a numerical index within square brackets to reference a cell in a numerical array, similarly with an associative array we place a string (whether literal or in a variable) within the same square brackets. As a structure that organizes and stores data, most programming languages have this concept, referring to it as an associative array, a dictionary, a map, a hashtable, or (in JavaScript) simply an *object*.

This data structure consists of key-value pairs: *keys* (strings) that are associated with *values* (the contents of the array cell). Just as (regular) arrays are initialized by `[]`, JavaScript objects are initialized by `{}`. The syntax for assigning values to keys *during object initialization* is this:

```
var myAssocArr = { fName: "Kaitemma", "lName": "Claiben"};
// notice that keys can be strings (quoted) or symbols (without quotes)
```

Once created, we access an object's keys as *array indices* (with `[]`) or *object properties*:

```
myAssocArr["fun"] = "shreds on electric cello";
console.log(myAssocArr);      // { fName: "Kaitlemma", lName: "Claiben",
                              //   fun: "shreds on her electric cello" }
myAssocArr.IQ = 144;
console.log(myAssoc["IQ"]     //   144
console.log(myAssoc.fun);     //   "shreds on her electric cello"
```

☐ Coin Change with Object

As before, given a number of U.S. cents, return the optimal configuration of coins, in an object.

☐ Max/Min/Average with Object

Given an array, return an object containing the array's max, min and average values.

FOR ... IN Loops

When working with arrays (whether they are associative or numerical), one of the most common tasks is to iterate through all keys or values in the array. With this type of array, the keys are not numerical: there isn't a predictable first index, like `[0]`. Furthermore, there is no obvious last index such as `[arr.length-1]`. Without something new that we have not yet learned, we don't know how many keys an object contains.

Fortunately for insiders like us, JavaScript has just what we need: the **`FOR...IN`** loop. Objects do not have a **`.length`** property, but with **`FOR...IN`** we can still iterate through each of the object's keys. There is no real guarantee on the *order* in which we will be handed these keys, but we will be handed each key exactly once. Many students, when encountering **`FOR...IN`** or **`FOREACH`** for the first time, are confused about whether the loop iterator represents a *key* or a *value*. For the JavaScript **`FOR...IN`**, *always think of it as* **`FOR (key IN obj)`**. The loop iterator represents *keys*, not *values*. If you need to iterate values within an object, then within a **`FOR (key IN obj)`** loop, reference **`obj[key]`**.

Using your new knowledge and skills with JavaScript objects (the equivalent in other programming languages to an associative array, dictionary or hashtable), try your hand at the following challenges:

☐ Zip Arrays into Map

Associative arrays are sometimes called *maps* because a key (string) maps to a value. Given two arrays, create an associative array (map) containing keys of the first, and values of the second. For **`arr1 = ["abc", 3, "yo"]`** and **`arr2 = [42, "wassup", true]`**, return **`{"abc": 42, 3: "wassup", "yo": true}`**.

☐ Invert Hash

Associative arrays are also called *hashes* (we'll learn why later). Build **`invertHash(assocArr)`** to convert hash keys to values, and values to keys. Example: given **`{"name": "Zaphod", "charm": "high", "morals": "dicey"}`**, return object **`{"Zaphod": "name", "high": "charm", "dicey": "morals"}`**.

☐ Array: Number of Values (without .Length)

Without using the **`.length`** property that is present on all arrays, determine and return the number of values in the given array. If we were to do this on a numerical array, we might check to see whether the element at a certain numerical index was **`undefined`**. Unfortunately we can't do that here because the keys don't have any sort of predictable order or first value.

So, for object **`{ band: "Travis Shredd & the Good Ol' Homeboys", style: "Country/Metal/Rap", album: "668: The Neighbor of the Beast" }`**, you should return the value **`3`**, because there are three keys in this object: **`band`**, **`style`** and **`album`**.

Why Don't We Allow Built-In Functions?

Knowing available services for a language or framework is essential for unlocking its value. That said, there is power in knowing how to recreate those services – if they don't work as expected, or when you must extend them for new scenarios. Furthermore, having a sense for how services work 'under the hood' deepens your understanding about how/when to use them. Knowing for example that `push()` and `pop()` are *significantly* faster than `splice()` might make a difference in which you choose.

For extra algorithm practice, recreate these built-in functions from JavaScript's string library.

☐ String.concat

`String.concat(str2,str3,...,strX)` - add string(s) to end of existing one. Return new string.

☐ String.slice

`String.slice(start,end)` - extract part of a string and return in a new one. *Start* and *end* are indices into the string, with the first character at index 0. *End* param is optional and if present, refers to one beyond the last character to include.

Bonus: include support for negative indices, representing offsets from string-end. Example: `String.slice(-1)` returns the string's last character.

☐ String.trim

`String.trim()` - remove whitespace (spaces, tabs, newlines) from both sides, and return a new string. Example: `" \n  hello goodbye \t  ".trim()` should return `"hello goodbye"`.

☐ String.split

`String.split(separator,limit)` - split a string into array of substrings, returning the new array. `Separator` specifies where to divide substrings and is not included in any substring. If `""` is specified, split the string on every character. `Limit` is optional and indicates number of splits; additional post-limit items should be discarded. Note: existing string is unaffected.

☐ String.search

`String.search(val)` - search `string` for the given `val` (another string). Return the index position of the first match found (or `-1` if not found).

Chapter 4 – Strings and Associative Arrays

☐ Short Answer Questions: Strings and Associative Arrays

What is a string? How is it different than an array?
What is a data type? Is this what `typeof` tells us? What JavaScript data types have we learned?
What does `typeof` return, if given a string? What does `typeof` return, if given an array?
How do you quickly determine the number of characters in a string?
Are spaces counted toward the length of a string?
What are a few of the built-in (method) functions available on every string?
Is there a built-in function to easily convert a string to an array? Show me how to do this.
Is there a built-in function to easily convert an array to a string? Show me how to do this.
Is there a built-in function to easily convert a string to a boolean? Show me how to do this.
Is there a built-in function to easily convert a number to a string? Show me how to do this.
What is a `switch` statement, and when would you best use one?
What is a fast-finish check? Does it actually make your code faster?
What is an associative array? How does one differ from a traditional 'array classic'?
What is a JavaScript object? Featurewise, how does it differ from an associative array?
Is an object the closest thing – in JavaScript – to an associative array? What is its data type?
What does 'immutable' mean? Is a string immutable? Is an array immutable? Is an object?
To manually iterate through the keys and values in an object, what type of loop do I need?
Does this type of loop give you the keys, or the values?
Why does the Dojo frown on the use of built-in functions, during most algorithm challenges?

☐ Weekend Challenge: Strings and Associative Arrays

This weekend, go online and find a long text file (such as http://www.classicreader.com/book/206/1, for example – although any multi-page text would suffice). We want to do some analysis on this text! Find a way to get the text into your JavaScript code, and then determine the following:

- How many letters (including spaces) are in the text? How many words?
- How many *unique* letters are in the text? Ignoring punctuation, how many *unique* words?
- What are the unique-letter and unique-word results if you ignore capitalization?
- List all the unique words in alphabetical order. Put them into an array in this order.
- What are the ten *most common* words in this text? How frequently do each of them occur?
- Create an array of unique words and number of appearances, *in ascending count order*. What is your best choice of data structure here?

Strings and Associative Arrays Review

In this chapter we learned about the **string** data type: how to access specific characters, determine the length, and use built-in string functions. We learned about the **`switch`** statement, as well as the concepts of **fast-finish** and **fast-fail**. We introduced the concept of associative arrays (and objects, which are the closest concept in JavaScript to an associative array). We also discovered the **`FOR..IN`** loop and used it to iterate through the *keys* (not *values*) of an object.