# Append() and Prepend()

# Pop() and Push()

var sampleArray = [0,0,7];

var incrementLastArrayElement = function(\_array)

{

var newArray = [];

// Your code should make newArray equal to an array that has the same

// values as \_array, but the last number has increased by one.

// For example:

// \_array = [1, 2, 3];

// turns into:

// newArray = [1, 2, 4];

// Your code goes in here!

newArray = \_array.slice(0);

var lastNumber = newArray.pop();

newArray.push(lastNumber + 1);

// Don't delete this line!

return newArray;

};

// Did your code work? The line below will tell you!

console.log(incrementLastArrayElement(sampleArray));

newArray = \_array.slice(0);

Just like string.slice(begin, [end]) separates the characters of a string, array.slice(begin, [end]) separates the elements of an array from the index of the begin up to but not including end. We want to make a copy of the original array, so we won't include an [end]. At this point, newArray is a copy of the original \_array.

var lastNumber = newArray.pop();

The array.pop() method conveniently gives us (or returns) the last element of the array, which in this case is the number we want to increase by 1. However! Be careful because array.pop() actually removes the last element of the array. This is why we made a copy in the previous line, so we wouldn't modify the original \_array.

newArray.push(lastNumber + 1);

Just how the jQuery .append() method adds an element to the end of an HTML block, the array.push() method adds an element to the end of an array. Here, we're .push()ing the lastNumber + 1, which is exactly what we wanted to do.

# Split Array + Upper and Lower Case

var name = "AlbERt EINstEiN";

function nameChanger(oldName)

{

var finalName = oldName;

// Your code goes here!

var names = oldName.split(" ");

names[1] = names[1].toUpperCase();

names[0] = names[0].slice(0,1).toUpperCase() + names[0].slice(1).toLowerCase();

finalName = names.join(" ");

// Don't delete this line!

return finalName;

};

// Did your code work? The line below will tell you!

console.log(nameChanger(name));

var names = oldName.split(" ")

Here, we're creating an array of names by breaking the original name at the space. At this point for our original example, names === ["AlbERt", "EINstEiN"]

names[1] = names[1].toUpperCase();

The string.toUpperCase() method does exactly what its name describes. It's acting on names[1], which is "EINstEiN" in the original example. So here, we're reassigning the second element in the names array to the all caps version of "EINSTEIN".

names[0] = names[0].slice(0,1).toUpperCase() + names[0].slice(1).toLowerCase();

This line builds Albert. names[0].slice(0,1).toUpperCase() starts by acting on the first element in the names array, which is "AlbERt" in the example. Then, we grab just the first letter by using .slice(0,1). Then we simply use the .toUpperCase() method again to make sure the first letter is capitalized. Also, notice how we chained two methods together here.

The next part, names[0].slice(1).toLowerCase() acts similarly, except this time we're using .slice(1) to grab the rest of the first string, which is "lbERt" in the example. Then we simply chain the .toLowerCase() method to make sure that the rest of the letters in the first name are lower case. Once we have the first letter capitalized and the rest of the name lower case, we concatenate them together with the +.

finalName = names.join(" ");

array.join([chars]) lets us put array elements together into a single string. Each element will be separated by the optional chars. In this case, we want a space between the two names, so we made the chars a single space, " ". With that, we've joined "Albert" and "Einstein" to form "Albert EINSTEIN"!

# What is JSON?

**J**ava**S**cript **O**bject **N**otation. JSON is a popular and simple format for storing and transferring nested or hierarchal data. It's so popular that most other programming languages have libraries capable of parsing and writing JSON (like Python's [JSON library](https://docs.python.org/2/library/json.html)). Internet GET and POST requests frequently pass data in JSON format. JSON allows for objects (or data of other types) to be easily encapsulated within other objects. See the [MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON) or [JSON.org](http://json.org/) for more details.

[This](http://www.copterlabs.com/blog/json-what-it-is-how-it-works-how-to-use-it/) is a fantastic deep dive from Jason Lengstorf about JSON and its ubiquitous use in the form of [AJAX requests](http://en.wikipedia.org/wiki/Ajax_(programming)).

## Why should I lint my JSON?

With a mix of nested curly braces, square brackets and commas, it's easy to make mistakes with JSON. And mistakes mean bugs. Seriously, I mess up JSONs all the time. You might even be able to spot a bug in one of my JSONs in a video in this course...

If you're generating JSON by hand, you should copy and paste your code into a JSON linter like [jsonlint.com](http://jsonlint.com/) to quickly and easily find syntax errors. A linter is a piece of software that analyzes code for syntax errors. Some text editors, like Sublime Text, will automatically lint (or highlight) most syntax errors. But a JSON linter won't miss any syntax errors and you can rest assured that your JSONs will be properly formatted.

Click "Next" to dive into JSON!

**Important Note:**  
For the Online Resume project you'll be using javaScript Object Literals rather than JSON to define your objects. The syntax is very similar, but javaScript Object Literals permit the inclusion of functions as properties and JSON does not.

# What is the DOM?

This course isn't about HTML and CSS but they keep showing up. Why is that? HTML, CSS and JavaScript are the three components that make almost every website. As part of the process of building websites, browsers convert all of the HTML they receive into a JavaScript object called the Document Object Model (DOM).

In fact, from the console, you can examine the DOM for any website (including this one!). By examining the DOM, you can learn pretty much everything about a website.

### Your Challenge

For this challenge, I want you to find out the height of the classroom window (in pixels).

To do so, you'll first need to open the console. Got it open? Select the classroom window. Try typing document.getElementsByClassName("text-atom")[0] and press enter.

See the HTML that comes up? That's the HTML that makes up the classroom. It's actually a JavaScript object called an Element with properties you can access.

Read up on [Element properties](https://developer.mozilla.org/en-US/docs/Web/API/Element). **Can you figure out the height of the classroom window?** Your answer will depend on the size of your browser.

If the resume you're building doesn't have data in a certain section, like Education, the empty section is automatically hidden. At the bottom of index.html, you'll find a few lines of JavaScript that follow this pattern:

if (document.getElementsByClassName("education-entry").length === 0) {

document.getElementById("education").style.display = "none";

}

After we break down the way this code works, I'm going to challenge you to modify these scripts to turn empty resume sections black instead of making them disappear.

## How Resume Sections Get Hidden

While we're adding sections to the resume using jQuery, these scripts are using regular JavaScript to hide them.

First off, there's an if statement. You'll be learning about if statement syntax in the next lesson. Suffice it to say, we'll be executing the code that comes between the curly braces if a resume section is empty. In this example, the education section is empty so we execute document.getElementById("education").style.display = "none";

### document

document refers to the web page in its current state. You may have heard of the DOM, which contains all of the information about a website's visible elements. document is the [DOM](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model). We need to change the DOM if we want to add or remove page elements.

### getElementById("education")

getElementById(id) is a DOM manipulation method that will grab a single page element object with the id given. Notice how the word Element is singular in the method name. That means it returns a single page element. Other DOM methods containing Elements, like getElementsByClassName(), will return an array of page elements.

In this example, we've grabbed the entire education section of the resume. In the HTML, this is everything inside <div id="education">

### .style

.style is a [DOM property](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement.style). It is used to change a CSS style of the selected page element. The property that follows .style is the CSS style that will be modified by this piece of code. [Here's a list](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Properties_Reference) of all the possible CSS properties you can change.

### .display = "none"

Every page element has a display CSS property, which normally controls how that page element interacts with others. If display is set to "none", however, then the element is removed entirely from the page.

### Your Challenge

Rather than hiding resume sections when they're empty, can you modify the scripts at the bottom of the resume so that they turn each section black when the section is empty? Here's a hint, you'll need to change the background color property of each element.

------------------------------------------------------------------------------------------------------------------------------------------

document.getElementById("education").style.backgroundColor = "black";

Looking at [style API](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement.style), I see that there's a link to the [CSS Properties Reference](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Properties_Reference). Considering that we need to change the background color, I looked through the list and found the backgroundColor property of style. I set it to "black" but you could also set backgroundColor to a string of black's [hex code](http://www.color-hex.com/), "#000000", instead.

Then, I simply copied the same code to each section, making sure to change the id in .getElementById(id) to the right one for each!

Let's imagine that instead of building just for your own resume, you're building a webapp that takes in data from other users on the internet and turns it into a resume that they can use.

How might you make sure that the resume will still display correctly? Or even worse, imagine someone sets their name to equal <script src="http://hackyourwebsite.com/eviljavascript.js"></script>. Can you make sure your resume doesn't run their malicious script?

### Your Challenge

For this quiz, your goal is to make sure that if a user puts HTML in their resume's JSON, it doesn't break the resume!

How? You need to make sure that the < and > from their HTML get turned into harmless strings.

When you're ready to [replace](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/replace) all of the < and >, click "Continue to Quiz!"

var html = '<script src="http://hackyourwebsite.com/eviljavascript.js"></script>';

var charEscape = function(\_html) {

var newHTML = \_html;

// How will you make sure that newHTML doesn't contain any < or > ?

// Your code goes here!

newHTML = \_html.replace(/</g, "&lt;");

newHTML = newHTML.replace(/>/g, "&gt;");

// Don't delete this line!

return newHTML;

};

// Did your code work? The line below will tell you!

console.log(charEscape(html));

var charEscape = function(\_html) {

var newHTML = \_html;

newHTML = \_html.replace(/</g, "&lt;");

newHTML = newHTML.replace(/>/g, "&gt;");

return newHTML;

};

There are a few ways to remove < and > from code. The simplest is simply replacing them with their [character entity references](http://en.wikipedia.org/wiki/List_of_XML_and_HTML_character_entity_references#Character_entity_references_in_HTML) (&lt; and &gt;).

To do so, we can use string.replace(old, new). Note, however, that if you pass in a string as old, string.replace(old, new) will only replace the first instance of the old string.

You must pass in a [regular expression](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions) as old to replace every instance of old in the string.

In the example above, we passed /</g and />/g as old into string.replace(old, new), which are regular expressions that grab all instances of < and >.

Alternatively, you could pass an HTML string into a function like encodeURIComponent(string) to remove instances of < and >. But it isn't [intended for situations like this](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/encodeURIComponent), possibly leading to unexpected consequences.

Special thanks to [Michael Händel](http://forums.udacity.com/questions/100231121/spoiler-i-disagree-with-the-solution#ud804) for correcting my inaccurate solution!

Example

var weirdObject = {

"property": "Time for an astronomy lesson!",

"property1": "Cameron's minor in college was astronomy",

"property-2": "The 4 Galilean largest moons of Jupiter are:",

"property 3": "Io, Ganymede, Callisto, Europa",

"property$": "Saturn's moon Enceladus has liquid water ocean under its icy surface",

" property": "The Sun contains 99.87% of the mass of the entire solar system",

"property()": "There are 5 dwarf planets in our solar system:",

"property[]": "Pluto, Ceres, Eris, Haumea, Makemake",

"8property": "Mars has two tiny moons: Phobos and Deimos"

};

// Use console.log() to figure out if dot and/or bracket notation

// will work to access the properties below. Mark true if you can use dot/bracket

// notation to access the property, otherwise mark false.

// For example, uncomment the line below to see if you can use dot notation to access `property1`.

// console.log(weirdObject.property1);

// I'll give you the first answer. The rest are set to false. Try out each property and

// if you can use dot or bracket notation to access it, change the answer to true!

// property

var dotNotation0 = true;

var bracketNotation0 = true;

// property1

var dotNotation1 = false;

var bracketNotation1 = false;

// property-2

var dotNotation2 = false;

var bracketNotation2 = false;

// property 3

var dotNotation3 = false;

var bracketNotation3 = false;

// property$

var dotNotation4 = false;

var bracketNotation4 = false;

// \*space\*property

var dotNotation5 = false;

var bracketNotation5 = false;

// property()

var dotNotation6 = false;

var bracketNotation6 = false;

// property[]

var dotNotation7 = false;

var bracketNotation7 = false;

// 8property

var dotNotation8 = false;

var bracketNotation8 = false;

### Solution Summary:

Bracket notation always works. Dot notation requires properties that begin with a letter and do not include special characters.

Let's go through these one-by-one.

### property

**dot notation**: true

**bracket notation**: true

This one is normal. Either syntax is fine.

### "property1"

**dot notation**: true

**bracket notation**: true

A number attached to the end of a property is acceptable for dot and bracket notation.

### "property-2"

**dot notation**: false

**bracket notation**: true

Some special characters like the - are not acceptable with dot notation but will still work with bracket notation.

### "property 3"

**dot notation**: false

**bracket notation**: true

Spaces are generally bad form in programming. Don't use them except within strings. But you can still access a property name with a space using bracket notation.

### "property$"

**dot notation**: true

**bracket notation**: true

Surprisingly, you actually can use $ within property names and still access them with dot notation.

### " property"

**dot notation**: false

**bracket notation**: true

In dot notation, the space actually gets ignored, so you are accessing "property" instead. But bracket notation still works.

### "property()"

**dot notation**: false

**bracket notation**: true

Without quotes, property() is a function call. This is just plain bad. While you can access a property like this one with bracket notation, there's no reason you should ever include () within an object property.

### "property[]"

**dot notation**: false

**bracket notation**: true

Like the last one, this is bad form too. [] already have a specific purpose in JavaScript and should never be used within a property.

### "8property"

**dot notation**: false

**bracket notation**: true

Dot notation fails to work if the property starts with a number. This is also bad form. Properties should never start with numbers.

# Flow Control

## If Statement

**Strict equality (===) vs Loose equality (==)**

When you use three equal signs, ===, no type conversion is done prior to the comparison. If the values are different types, for example, a String and a Number, they can't ever be equal. To return true, the values must be equal and the types must be the same.

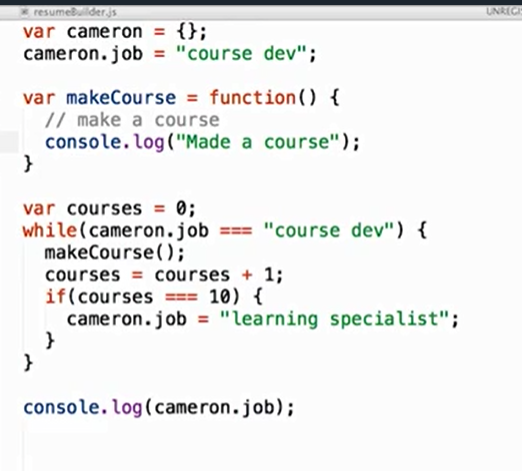
Loose equality, ==, checks to see if the two values are the same type and if not, converts to a common type before the conversion. If the types are already the same, there is no difference between the result of === and ==. When they aren't it can cause unexpected results.

if (document.getElementsByTagName('h1').length === 0) {

document.getElementById('header').style.display = 'none';

}

## While Loop



## For Loop

## For In Loops

Clarification

Use forEach or for to iterate over arrays like:

countries = ['Argentina', 'China', 'England'];

Use for-in to loop over objects like

countries = {'country1':'Argentina', 'country2':'China','country3':'England'};

being careful to wrap the content of the for-in in a conditional statement that tests if the key is part of the object:

myObj = {'country1':'Germany', 'country2':'Argentina'};

for (key in myObj){

if (myObj.hasOwnProperty(key)) {

console.log(myObj[key]);

}

}

## Functions

## Collecting Click Locations

Here's the code shell for you:

$(document).click(function(loc) {

// your code goes here

});

**$(document).click()**  
$(document).click() is a jQuery event handler on the page, which is a fancy way of saying that it will hold some code that runs every time a user clicks on the page. The function (that doesn't have a name, making it an anonymous function) that gets passed into .click() will be run every time a user clicks on the page.

## Split

var work = {

"jobs": [

{

"employer": "Udacity",

"title": "Course Developer",

"location": "Mountain View, CA",

"dates": "Feb 2014 - Current",

"description": "Who moved my cheese cheesy feet cauliflower cheese. Queso taleggio when the cheese comes out everybody's happy airedale ricotta cheese and wine paneer camembert de normandie. Swiss mozzarella cheese slices feta fromage frais airedale swiss cheesecake. Hard cheese blue castello halloumi parmesan say cheese stinking bishop jarlsberg."

},

{

"employer": "LearnBIG",

"title": "Software Engineer",

"location": "Seattle, WA",

"dates": "May 2013 - Jan 2014",

"description": "Who moved my cheese cheesy feet cauliflower cheese. Queso taleggio when the cheese comes out everybody's happy airedale ricotta cheese and wine paneer camembert de normandie. Swiss mozzarella cheese slices feta fromage frais airedale swiss cheesecake. Hard cheese blue castello halloumi parmesan say cheese stinking bishop jarlsberg."

},

{

"employer": "LEAD Academy Charter High School",

"title": "Science Teacher",

"location": "Nashville, TN",

"dates": "Jul 2012 - May 2013",

"description": "Who moved my cheese cheesy feet cauliflower cheese. Queso taleggio when the cheese comes out everybody's happy airedale ricotta cheese and wine paneer camembert de normandie. Swiss mozzarella cheese slices feta fromage frais airedale swiss cheesecake. Hard cheese blue castello halloumi parmesan say cheese stinking bishop jarlsberg."

},

{

"employer": "Stratford High School",

"title": "Science Teacher",

"location": "Nashville, TN",

"dates": "Jun 2009 - Jun 2012",

"description": "Who moved my cheese cheesy feet cauliflower cheese. Queso taleggio when the cheese comes out everybody's happy airedale ricotta cheese and wine paneer camembert de normandie. Swiss mozzarella cheese slices feta fromage frais airedale swiss cheesecake. Hard cheese blue castello halloumi parmesan say cheese stinking bishop jarlsberg."

}

]

};

// Your code goes here! Let me help you get started

function locationizer(work\_obj) {

var localArray = [];

for(job in work\_obj.jobs)

{

var newlocation = work\_obj.jobs[job].location;

localArray.push(newlocation);

}

return localArray;

}

// Did locationizer() work? This line will tell you!

console.log(locationizer(work));

## Anonymous functions

Anonymous functions are functions that don't have a name and are often returned by other functions and objects.   
  
Some JavaScript libraries ask for a callback function to be executed once they have have the results of a task. Anonymous functions are used in these cases because there is not a need to call the function by name outside the confines of the enclosing function.  
  
For example, the code below reads a JSON file from the server. After loading, it executes an anonymous function to print out the data.

$.getJSON("test.json", function(data) {

console.log(data);

});

Anything that uses an anonymous function could also use a named function. The following code is also valid and is equivalent to what's listed above:

var printData = function(data){

console.log(data)

};

$.getJSON("test.json", printData);

## Google Maps API Key

1. Uncomment the last block of code in helper.js. The code you need starts with window.addEventListener('load', initializeMap); and goes until the end of the file.
2. Uncomment the <script> tag for Google Maps API in the <head> of index.html.

**Google Maps API Key**  
Google is increasingly requiring an API key to make Google Map requests. You can obtain your own Google Maps API key [**here**](https://developers.google.com/maps/documentation/javascript/get-api-key#types-of-api-keys).

Once obtained, you can add the key to the Google Maps API script request in index.html:   
  
<script src="http://maps.googleapis.com/maps/api/js?libraries=places&key=YOUR\_API\_KEY\_HERE"></script>   
  
Want to learn about HTML and CSS? Try out [**Intro to HTML and CSS**](https://www.udacity.com/course/ud304).   
Want to check out your classmates' resumes? Check [**them out on GitHub**](https://github.com/search?utf8=%E2%9C%93&q=frontend-nanodegree-resume)

## Example 1

### Example 1

var outsideExample = "First string";

function example() {

var outsideExample = "Second string";

}

example();

console.log(outsideExample); // "First string"

### Example 2

var outsideExample = "First string";

function example() {

outsideExample = "Second string";

}

example();

console.log(outsideExample); // "Second string"

In the first example, notice that we're using var within example(). var means we're declaring a new outsideExample variable within example(). There are two outsideExamples in our program, one with a global scope ("First string") and one with a function-level scope inside example().

After we run example() and try to log outsideExample, we'll log the global version of outsideExample, which is "First string" because console.log() doesn't have access to the version of outsideExample that was created within example().

A global scope means that the variable is accessible anywhere inside our program, which is why we don't need to use var in example 2. In example 2, there's only one version of outsideExample because we aren't declaring a new variable inside example(). Instead we're simply modifying the value of the original global variable outsideExample, which is why we log "Second string" at the end of example 2.

To learn more, check out [this in-depth article](http://www.adequatelygood.com/JavaScript-Scoping-and-Hoisting.html) on JavaScript scope.

## Example 2

var outsideExample = "First string";

if (true) {

var outsideExample = "Second string";

console.log(outsideExample);

}

console.log(outsideExample);

The first console.log() obviously logs "Second string" as it comes right after we set the value of outsideExample to "Second string". But what about the second console.log()?

Remember, if statements do **not** create their own scope. Unlike the last quiz, where we created a totally new variable inside a function, the if statement does not create a new variable. It simply overwrites the value of outsideExample to "Second string".

So, the second time we console.log(), we see "Second string" again.

## Example 3

### Example 1

example1();

function example1() {

console.log("Ran the example");

}

You should see "Ran the example!" when you run this code.

### Example 2

example2();

var example2 = function() {

console.log("Ran the example");

}

You should see an undefined error when this code gets run.

In both examples, the interpreter modifies the code. This is effectively how this code is interpreted.

### Example 1 Interpreted

var example1;

example1 = function() {

console.log("Ran the example");

}

example1();

### Example 2 Interpreted

var example2;

example2();

example2 = function() {

console.log("Ran the example");

}

In both examples, the declarations get moved to the top of the scope. But notice how the definition comes along too in the first example, which allows us to use example1() right away.

## Example 4

Let's start by reading through the infoWindow constructor in the [Google Maps API](https://developers.google.com/maps/documentation/javascript/reference#InfoWindow).

It says that "After constructing an InfoWindow, **you must call open to display it on the map.**"

In other words, the infoWindow has an open method encapsulated in it that we must call to open each overlay.

This is a good place to start. Let's take a look at [the example](https://developers.google.com/maps/documentation/javascript/infowindows) to see how they open each infoWindow.

Looking down through the example, it looks like they're opening the window here:

google.maps.event.addListener(marker, 'click', function() {

infowindow.open(map, marker);

});

They're calling the open() method on the infowindow object (in helper.js, we call it infoWindow) and pass in the map object and the marker object.

But how did they know to pass in map and marker? The open(map, anchor) method described in the API explains that it needs a map first and then an anchor point. The anchor point is simply the position to open the map, which we obviously want to be the same as the map marker.

So, in the end we simply add the line

infoWindow.open(map, marker)

to the event listener for each map marker and we've got overlays!

Next step, try formatting the content of the infoWindow, which is simply a string that contains the overlay's HTML. Right now it's set to name, which is just the name of the city. Try learning some [HTML and CSS](https://www.udacity.com/course/ud304) to make the overlay really stand out!

# Project

## How do I complete this project?

1. Go to the [Javascript Basics course](https://www.udacity.com/course/ud804) and select "View Course Materials."
2. Go through the videos and assignments in this course to learn the JavaScript necessary to build your resume.
3. Review your work against the Project Rubric (on the next page).
4. When you are satisfied with your project, submit it according to the Submission Instructions on the next page.

### By the end:

Your resume will look something like this

And your repository will include the following files:

* **index.html**: The main HTML document. Contains links to all of the CSS and JS resources needed to render the resume, including resumeBuilder.js.
* **js/helper.js**: Contains helper code needed to format the resume and build the map. It also has a few function shells for additional functionality. More on helper.js further down.
* **js/resumeBuilder.js**: This file is empty. You should write your code here.
* **js/jQuery.js**: The jQuery library.
* **css/style.css**: Contains all of the CSS needed to style the page.
* **README.md**: The GitHub readme file.
* and some images in the images directory.

## Your starting point...

### js/helper.js

Within helper.js, you’ll find a large collection of strings containing snippets of HTML. Within many snippets, you’ll find placeholder data in the form of %data% or %contact%.

Each string has a title that describes how it should be used. For instance, HTMLworkStart should be the first <div> in the Work section of the resume. HTMLschoolLocation contains a %data% placeholder which should be replaced with the location of one of your schools.

### Your process:

The resume has four distinct sections: work, education, projects and a header with biographical information. You’ll need to:

1. Build four JSONs, each one representing a different resume section. The objects that you create need to follow the names within the schema below exactly. Make sure your JSONs are formatted correctly using [JSONlint.com](http://jsonlint.com/).
2. bio contains:
3. name : string
4. role : string
5. contacts : an object with
6. mobile: string
7. email: string
8. github: string
9. twitter: string
10. location: string
11. welcomeMessage: string
12. skills: array of strings
13. biopic: url
14. display: function taking no parameters
15. education contains:
16. schools: array of objects with
17. name: string
18. location: string
19. degree: string
20. majors: array of strings
21. dates: integer (graduation date)
22. url: string
23. onlineCourses: array with
24. title: string
25. school: string
26. date: integer (date finished)
27. url: string
28. display: function taking no parameters
29. work contains
30. jobs: array of objects with
31. employer: string
32. title: string
33. location: string
34. dates: string (works with a hyphen between them)
35. description: string
36. display: function taking no parameters
37. projects contains:
38. projects: array of objects with
39. title: string
40. dates: string (works with a hyphen between them)
41. description: string
42. images: array with string urls
43. display: function taking no parameters
44. Iterate through each JSON and append its information to index.html in the correct section.
    * First off, you’ll be using jQuery’s selector.append() and selector.prepend() functions to modify index.html. selector.append() makes an element appear at the end of a selected section. selector.prepend() makes an element appear at the beginning of a selected section.
    * Pay close attention to the ids of the <div>s in index.html and the HTML snippets in helper.js. They’ll be very useful as jQuery selectors for selector.append() and selector.prepend()
45. You’ll also be using the JavaScript method string.replace(old, new) to swap out all the placeholder text (e.g. %data%) for data from your resume JSONs.
46. Here’s an example of some code that would add the location of one your companies to the page:
    * var formattedLocation = HTMLworkLocation.replace("%data%", work.jobs[job].location);
    * $(".work-entry:last").append(formattedLocation);
      + Use the mockup at the page of this document as a guide for the order in which you should append elements to the page.
47. The resume includes an interactive map. To add it, append the googleMap string to <div id=”mapDiv”>.
48. All of your code for adding elements to the resume should be within functions. And all of your functions should be encapsulated within the same objects containing your resume data. For instance, your functions for appending work experience elements to the page should be found within the same object containing data about your work experience.
49. Your resume should also console.log() information about click locations. On line 90 in helper.js, you’ll find a jQuery onclick handler that you’ll need to modify to work with the logClicks(x,y) function above it.
50. It’s possible to make additional information show up when you click on the pins in the map. Check out line 174 in helper.js and the Google Maps API to get started.

We learned about relational operators and how they can classify the relationship between two values. Your job is to write the function getRelationship(x,y) function, which should return a string representing whether x is >, < or = y. For example:

var rel = getRelationship(2, 3);

console.log(rel); // <

If one or both of the values aren't numbers, your function should output:

"Can't compare relationships because [this value] and [that value] [is]/[are] not [a] number[s]."

where [this value] and [that value] are replaced with the non-numerical values. The sentence should be grammatically correct by outputting either is or are and pluralizing number if necessary.

For example:

var rel1 = getRelationship("this", 2);

console.log(rel1); // "Can't compare relationships because this is not a number"

var rel2 = getRelationship("that");

console.log(rel2) // "Can't compare relationships because that and undefined are not numbers"

Notice in the second example, because the y value was missing, the output said that undefined was not a number.