FRONTEND

- **HTML**
  - **Document** (individual elements combined to form an entire HTML page)
    - **Doctype**: <!DOCTYPE html>
      - Required as the first line of an HTML document (historical artifact).
    - **Root Element**: <html>
      - Follows the "doctype" and wraps around all content on the entire page.
    - **Head Element**: <head>
      - Container for things that do not appear as viewable content (e.g., keywords and descriptions that will appear in search results, CSS, character set declarations, etc.).
    - **Character Set**: <meta charset="UTF-8">
      - Allows document to use "utf-8" character set, which includes most characters from all known human languages (nests within head element).
    - **Title**: <title>
      - Sets the title that appears in browser tab (nests within head element).
      - Also appears as the search result in Google.
    - **Body**: <body>
      - Contains all of the content that will be shown to the viewer.
  - **Elements** (content + opening/closing tags)
    - **Block Elements** form a visible block on a page (e.g., paragraphs, lists, navigation menus, footers, etc.):
      - **Paragraph**: <p>
      - **Divider**: <hr>
      - **Headings**: <h1> through <h6>
        - **NOTE**: As a general rule, try to have only one <h1> tag in your HTML document, and it should be the biggest text element on the page.
      - Generic **Container**:
      - **Lists** (each item within a type of list needs to be identified by the "<li>" tag):
        - Ordered Lists (lists that are numbered):
        - Unordered Lists (lists composed of bullet points:
      - **Tables**: <table>
        - Table Row: <tr>
        - Table Header (consists of one cell within a row): <th>
          - Should be nested within <thead> under main table (semantics).
        - Table Data (consists of one cell within a row): <td>
          - Should be nested within <tbody> under main table (semantics).
        - Borders can be added by entering <table border="1">, although this is discouraged, as CSS should be used for styling.
      - **Forms** (interactive controls to submit information to a web server): <form>
        - Typically contain the "action" (the URL to send form data to) and "method" (the type of HTTP request, such as "GET" to receive information from the server and "POST" to add information to the serve) attributes, e.g.:
          <form action="/my-form-submitting-page" method="POST">
        - **Input** (used to accept data from the user): <input>

- The operation of <input> depends upon its type attribute. For a complete list of attributes, view **Form Input Types**. Examples:
  - **Text** (can be used for user names): type="text"
  - **Password**: type="password"
  - **Placeholder** (temporary text in input fields; used with "text" and "password" attributes): placeholder="insert-text-here"
  - **Button**: type="button" value="insert-text-here"
    - Simple **Submit** button: type="submit"
      - Alternatively, if placed at the end of a form, use the following to create an even simpler submit button:
        <button>insert-text-here</button>
  - **Checkbox** (square box for multiple choices): type="checkbox"
    - To have the checkbox already marked upon loading, add the attribute "checked" to the input.
  - **Radio Button** (circular icon for one choice): type="radio"
    - In order to make the user only able to select one choice, you must add the "name" attribute, which must be common among all choices.
    - The "value" attribute is necessary for the query string to understand the meaning behind each choice; otherwise, it will simply state "name=on".
    - Example:
      <label for="cats">Cats:</label>
      <input name="pet-choice" id="cats" type="radio" value="CATS">
      <label for="dogs">Dogs:</label>
      <input name="pet-choice" id="dogs" type="radio" value="DOGS">
- **Dropdown Menus**: <select>
  - For every possible option to select, use an <option> tag.
  - In order for the query string to understand that an option has been selected from the dropdown menu, the "name" attribute must be included in the <select> tag, e.g.:
    <select name="color">
      <option>White</option>
      <option>Black</option>
    </select>
  - If you want the query string to contain text other than "White" or "Black" in the example above, use the "value" attribute in the <option> tag, e.g.:
    <option value="happy">☺</option>
- **Text Areas** (multi-line plain-text editing control): <textarea>
  - You can specify how large the text area is by using the "rows" and "cols" attributes.

            - In order for the query string to process the data in the text area, you must use the "name" attribute.
            - Example:
              <textarea name="paragraph" rows="10" cols="50"></textarea>
        - **Labels** (add captions for individual items in a form): <label>
            - A label can be used by placing the control element inside the <label> element, or by using the "for" and "id" attributes:
                - For example,
                  <label>Username<input type="text"></label>
                  …is identical to:
                  <label for="username">Username</label>
                  <input id="username" type="text">
        - **Validations** ensure that users fill out forms in the correct format, e.g.:
            - The Boolean attribute "required" makes a field mandatory:
              <label>Username<input type="text" required></label>
                - Only works if the browser (like Chrome) allows it.
            - By changing type from "text" to "email", the browser will ensure that the field contains an @ symbol.
    - **Inline Elements** are contained within block level elements and do not cause new lines to appear:
        - Italics:
        - Bold:
        - Generic Container:
    - **BUT NOTE**: **Empty Elements** contain only a single tag:
        - Image: <img src="https://www.google.com/logo.png">
            - **NOTE**: Image width can be modified like so…
              <img width="50" src="https://www.google.com/logo.png">
              …but is discouraged, as styling should be done by CSS.
        - Input: <input type="text">
    - For a complete list of elements, view the **MDN Element Reference**.
    - **TIP**: In Sublime, press "Ctrl+Shift+D" to replicate an entire line of an element.
- **Attributes** (extra info; does not appear in content; target of style info)
    - Components:
        - Space between it and the element name (or prior attribute),
        - Attribute name followed by an equals sign, and
        - Attribute value surrounded by quotation marks.
            - Double or single quotes may be used, but must be done consistently. You can nest a single quote within a double quote (and vice versa), but if you want to nest the same type of quote, you must use **HTML Entities** (e.g., " or ').
    - Examples:
        - Class: content
        - Can be paired with the "anchor" element: <a>
            - Hyperlink with Title:
              content
    - **BUT NOTE**: **Boolean Attributes** can be written without a value, but technically always have only one value (generally the same as the attribute name:
        - Disabled: <input type="text" disabled="disabled">

        - Creates a text box in which typing is disabled.
        - May also be written as:
          <input type="text" disabled>
    - For a complete list of attributes, view the **MDN Attribute Reference**.
- **Entity References** (make special HTML syntax characters appear as normal text):
    - < = <
    - > = >
    - " = "
    - ' = '
    - & = &
- **HTML Comments** (write comments in the code that are invisible to the user by wrapping them in special markers):
    - <!-- and -->
    - **TIP**: In Sublime, you can (1) select text, and (2) hold "Ctrl+/" to turn text into comment.

- **CSS**
    - **The General Rule**

```
selector {
    property: value;
    anotherProperty: value;
}
```

        - For example, make all <h1> tags purple with 56-pixel font:

```
h1 {
    color: purple;
    font-size: 56px;
}
```

        - **Three Basic Selectors**
            - **Element** selectors select all instances of a given element. For example, "div" is a CSS element selector that will modify the properties of all HTML tags.
            - The **ID** selector selects a single element with an octothorp ("#") ID (only one per page). For example, the following HTML/CSS combination will result in the word "hello" appearing in yellow, while the word "goodbye" will remain as is:

```
<div>
    <p id="special">hello</p>
</div>
<div>
    <p>goodbye</p>
</div>
#special {
    color: yellow;
}
```

            - The **Class** selector selects all elements in a given class by functioning just like an ID selector; however, a class is instead prefaced with a period (".") . For example, the following items marked as "completed" on a "To Do List" will be crossed out with a line:

```
<div>
    <p class="completed">TASK #1</p>
</div>
```

```
<div>
    <p class="completed">TASK #2</p>
</div>
.completed {
    text-decoration: line-through;
}
```

- An element can be modified by multiple class or ID tags by simply adding a space between the two tag names, e.g.:
  `<p class="completed uncompleted">Text</p>`

- **Five More Advanced Selectors**
  - The **Star (*)** selector applies to every element on the page.
  - The **Descendant** selector applies to selectors that have been "nested" under another. For example, if you want to modify the color of only those <a> tags that are nested within the <li> tags of a <ul> list, use the following:

```
ul li a {
    color: red;
}
```

    - In addition to HTML tags, CSS selectors such as "ID" or "Class" may be used within a Descendant selector.
    - **HOWEVER**: If, for example, you have a second-tier <ul> nested within a first-tier <ul> that is nested within , and you only want to select the first-tier <ul> and not the second-tier, then you must use the ">" combinator to select only the **DIRECT** first-tier "child" (rather than the second-tier "grandchild") of :

```
#box > ul {
    color: red;
}
```

  - The **Adjacent (+)** selector will select only the element that comes **IMMEDIATELY** after another element (a "sibling" element, rather than a "nested" element). For example, to modify the font size of all <ul> tags that follow an <h4> tag (which are typed on the same "level" as the <ul> tags, and not nested under them), use the following:

```
h4 + ul {
    font-size: 24px;
}
```

    - If, in the above example, you want to select **ALL** tags after any <h4> tag, then use the more generalized sibling combinator of "~" instead of "+".
  - The **Attribute** selector will allow the selection of any element based off of any attribute. For example, to change the font family of all <a> tags that link to Google, use the following:

```
a[href="https://www.google.com/"] {
    font-family: cursive;
}
```

    - This selector an also be used to select all images of a particular source, or all inputs of a particular type, such as all checkboxes:

```
input[type="checkbox"] {
```

```
        border: 2px solid green;
    }
```

- **TIP**: See the complete list of **Attribute Selectors**.

- The **Nth-of-Type** selector takes a specific number and selects the "-nth" instance of an element. For example, to change the background color of every second <li> tag in every list (literally meaning the second tag, not every other tag), use the following:

```
li:nth-of-type(2) {
    background-color: rgba(100, 175, 225, 0.5);
}
```

- **NOTE**: To select every other tag, use the phrases (even) or (odd) instead of a specific number.

- For more advanced selectors, view **The 30 CSS Selectors You Must Memorize**.

- **CSS Location**
  - CSS should generally be saved to its own file, but can also be included in the HTML head by using the <style> tag:

```
<style type="text/css">
    li {
        color: red;
    }
</style>
```

  - The preferred method is to use a <link> tag in the HTML head to refer to the separate file containing CSS:

```
<link rel="stylesheet" type="text/css" href="directory/filename.css">
```

- **Specificity** is the means by which browsers decide which CSS property values are the most relevant to an element and, therefore, will be applied (e.g., if the body is styled to have red text, but a paragraph within the body is styled to have green text, then the text will be green because the green text style is more relevant to the specific paragraph than the general body).
  - The following list of selector types increases by specificity (in magnitudes of 10):
    1. Type selectors (e.g., li) and pseudo-element (e.g., :before)
    2. Class selectors (e.g., .hello), attributes selectors (e.g., [type="text"]) and pseudo-classes (e.g., :hover)
    3. ID selectors (e.g., #hello)
  - This **Specificity Calculator** may be used to test CSS specificity rules.
- **The Box Model**
  - In a document, each element is represented as a rectangular box. In CSS, each of these boxes is described using the standard "box model." Each box has four edges: (1) **Content Edge**, (2) **Padding Edge**, (3) **Border Edge**, and (4) **Margin Edge**. Padding is the space between the border and the element within the border, and the margin is the space between the border and everything outside of the border.
    - The content edge can be controlled by setting the "width" and "height" properties in "px" or "%" (with percentage in relation to the parent element), which in turn pushes out the border edge as well, as there is direct contact between the content and border (if no padding has yet been set).
      - **NOTE**: By using the "max-width" property in conjunction with "width", you can tell the browser to make an element's width a certain percentage, but then also cap that width to a maximum number of pixels, e.g.:

```
#container {
```

```
        width: 66.66%;
        max-width: 700px;
    }
```

- Space can be added between the content edge and border edge (and between the border edge and the next element's edge) by using the "padding" and "margin" properties respectively (in "px" or "%").
    - By default, padding and borders are set to go around all edges of an element, but can be limited by using more specific properties for top, right, bottom, and left—such as "padding-left" or "margin-top".
    - Alternatively, rather than typing a line of CSS for all four sides, the following shorthand can be used (with the first value setting the top property, and the remainder moving in a clockwise fashion):

```
p {
    margin: 20px 40px 60px 80px;
}
```

- **NOTE**: By setting the "margin" property to "auto" on the left and right, an element will automatically be horizontally centered:

```
p {
    margin: 0 auto 0 auto;
}
```

    - The above syntax can also be shorted as (with the first value representing the vertical axis, and the second value representing the horizontal axis):

```
p {
    margin: 0 auto;
}
```

- **Colors**
    - Colors can be created through the **Hexadecimal** system by combining the octothorp (#) with a string of 6 hexadecimal "numbers" from 0-F, e.g.:
      `color: #FF1493;`
        - This system follows an RGB scheme in which the first two numbers modify the amount of "red," the second two modify "green," and the last two modify "blue."
    - Alternatively, colors can be created through the **RGB** system: 3 channels consisting of red, green, and blue, with each ranging from 0-255, e.g.:
      `color: rgb(0, 255, 0);`
    - Colors can be made **Transparent** through the RGBA system. Just like RGB but with an alpha (transparency) channel ranging from 0.0-1.0, e.g.:
      `color: rgba(11, 99, 150, .6);`
- **Backgrounds**
    - Follows the same format as colors, e.g., `background: #FF6789;`
    - The background property can also set a background image, e.g.:

```
body {
    background: url(http://www.website.com/image.png);
}
```

        - To prevent the background from **Repeating** an image, add the following property:
          `background-repeat: no-repeat;`

- To allow the background image to **Stretch** out across the entire body, use:
  background-size: cover;

- **Borders**
  - Borders have three key properties: "width" (typically in pixels), "color" (as noted above), and "style" (generally solid, dotted, or dashed). All three properties must be present in order for a border to take effect, e.g.:

```
h1 {
    border-width: 5px;
    border-style: solid;
    border-color: purple;
}
```

  - The alternative shorthand syntax may also be used (in the specified order):

```
border: 5px solid purple;
```

- **Fonts**
  - **Font-Family** specifies the font for an element:

```
p {
    font-family: Arial;
}
```

    - While not always necessary, you may sometimes have to put quotation marks around the font name—particularly when the font name begins with a number.
    - **CSS Font Stack** shows what percentages of operating systems have a given system font (useful for choosing a safe bet on system compatibility).
      - However, rather than using those limited fonts, it is better to use **Google Fonts**, choose a font, and embed the font's stylesheet link in your HTML <head> prior to the CSS link, e.g.:
        <link href="https://fonts.googleapis.com/css?family=Esteban" rel="stylesheet">
  - **Font-Size** specifies how big the font appears (typically in pixels or "px"):

```
h1 {
    font-size: 25px;
}
```

    - Another size unit is "em", which dynamically sets font size in relation to a parent element. For example, if you want to make a section of a paragraph's text in tags be twice the size of the rest of the text in the <p> tags, you would say:

```
span {
    font-size: 2em;
}
```

      - **BUT NOTE**: What constitutes the "standard" 1em (i.e., the default font size on a page without CSS markup) varies from browser to browser, although the size is typically around **16 PIXELS**. To ensure uniformity among browsers, it is useful to set the body's font size at the outset.
      - **ALSO**: Similar to "em" is "rem", which—rather than setting font size in relation to the parent element—sets the font size in relation to the "root" element on the page (i.e., the default font size discussed above).
  - **Font-Weight** specifies how thick or thin the font appears.

            - Typically involves absolute values of "normal" or "bold", or relative (to parent) values of "lighter" and "bolder", but can also be assigned a numeric value in increments of 100 generally from "100" to "800" depending on the font itself.
        - **Line-Height** controls the height of a given line (similar to changing line spacing in Word to 1.5 or 2.0, which means that a larger font will result in larger spacing).
        - **Text-Align** controls where an element's text is aligned on the page (typically "left", "right", and "center").
        - **Text-Decoration** is used to give text effects such as "underline", "overline", or "line-through".
    - **Float**
        - Normally, block level elements (such as) are stacked directly underneath the preceding element on the page. To change this, use the "float" property and specify a value of the direction in which the element should float ("left", "right", "none").
        - When an element is floated, it is taken out of the normal flow of the document (though still remaining part of it), and it is shifted to the left or right until it touches the edge of its containing box, or another floated element.
        - When <img> tags are laid out in a consecutive sequence, HTML automatically places some small amount of white space between the images. If you want to remove the white space, you can use "float" to remove that white space.

- **Bootstrap (v.3)**
    - **About**
        - Bootstrap is the most popular HTML, CSS, and JS framework for developing responsive, mobile-first projects.
        - To incorporate Bootstrap's CSS into your project, you can do either one of the following:
            (1) Download the bootstrap.css file, placing it into your project directory, and creating a <link> to the bootstrap.css file; or
            (2) Paste the following <link> to the bootstrap.css file, which is hosted online: <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" crossorigin="anonymous">
        - **NOTE**: The semantics in Bootstrap have been criticized as being sometimes meaningless, and another contender for performing these tasks is **Semantic UI**. However, Bootstrap is more popular and widely accessible through tutorials.
    - **Important Pieces**
        - **Buttons** require that they be identified first by a "btn" class followed by a second specific class.
            - There are seven specific classes:
                - Default (standard white button): btn-default
                - Primary (provides extra visual weight; set to be the primary action in a set of buttons): btn-primary
                - Success (indicates successful or positive action): btn-success
                - Info (contextual button for informational alerts): btn-info
                - Warning (indicates caution should be taken): btn-warning
                - Danger (indicates a potentially dangerous action): btn-danger
                - Link (deemphasizes the button to look like a link while behaving like a button): btn-link

- Buttons classes can be added to the <a>, <button>, or <input> elements, e.g.:
  Link
  <button class="btn btn-default" type="submit">Button</button>
  <input class="btn btn-default" type="button" value="Input">
  <input class="btn btn-default" type="submit" value="Submit">
- Button sizes can be reduced or increased by adding a third size class.
  - Large: btn-lg
  - Small: btn-sm
  - Extra Small: btn-xs
- To make a button appear "active," add the "active" class:
  <button class="btn btn-success btn-xs active">Text</button>
- To disable a button, add a "disabled" attribute and set its value to "disabled":
  <button class="btn btn-success btn-xs" disabled="disabled">Text</button>

- **Jumbotron** (class="jumbotron") extends the entire viewport to enlarge and showcase key content within.
  - By default, the Jumbotron will extend across the entire width of the screen. To prevent this behavior, place the within a , as the "container" class in Bootstrap will add padding and margins to center the Jumbotron.
- **Forms**
  - **Basic Form**
    - Applying the class="form-group" attribute to all elements in the form optimizes the spacing between the elements (e.g., between the username and password field).
    - Applying the class="form-control" attribute to an <input> element improves the style of the normal default appearance (adds rounded corners, padding, spacing, focus effects, etc.), but also makes changes to how the element behaves within the grid system.
    - Applying the class="help-block" to a <p> element modifies the text of a helpful hint to be more subtle and subdued in appearance.
  - **Inline Form**
    - A basic form will lay its contents by stacking them on top of each other. However, by applying the class="form-inline" to your form (which doesn't necessarily have to be a <form> element) will place its contents in a line from left to right.
- **Navbar**
  - Navbars serve as navigation headers for an application or site. Navbars must be placed within a <nav> element, and, like buttons, require a general "navbar" class followed by a second specific class (typically "navbar-default", but may also be "navbar-inverse" for an inverted color scheme).
  - A navbar may contain a "**Brand**" image (e.g., company name or logo) as the first object in the navbar. This is constructed by creating a "navbar-header" and placing an anchored "navbar-brand" within the header (to add a link to the homepage, typically), and then an <img> within the brand, if desired:
    <nav class="navbar navbar-default">

        <img src="#">

```
    </div>
</nav>
```

- The remaining content on the **Left Side** of the navbar should appear outside of the containing the "navbar-header", and it should be placed in <ul class="nav navbar-nav">. Each item within the navbar should be marked with <li> tags (and those should contain an <a> tag if a link is desired).
    - **NOTE**: The <li> tags will function normally even if contained in a rather than <ul>; however, the <ul> should be used for semantics.
- To add additional content to the **Right Side** of the navbar, use <ul class="nav navbar-nav navbar-right">.
    - To ensure that the content on the right side does not press too squarely on the right edge of the browser, everything within the <nav> should be placed within a (fixed width) or (full width).
- To get the full benefit out of a navbar (such as dynamic dropdown menus), you must install the Bootstrap **JavaScript** file in your HTML <body>. Like "bootstrap.css", you can either use the "bootstrap.js" file downloaded from Bootstrap, or you can use the following link:

```
<script
src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"
integrity="sha384-
Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Tx
a" crossorigin="anonymous"></script>
```

    - **NOTE**: Bootstrap's JavaScript requires **jQuery** to work, and its <script> must be placed BEFORE Bootstrap's:

```
<script  src="http://code.jquery.com/jquery-3.2.1.js" integrity="sha256-
DZAnKJ/6XZ9si04Hgrsxu/8s717jcIzLy3oi35EouyE="
crossorigin="anonymous"></script>
```

- To have items **Collapse** when the browser hits mobile size, place everything that you want to collapse inside .
    - To add the "hamburger" icon that provides a dropdown icon for the collapsed items, (1) change the above to read , and (2) place the following <button> within the "navbar-header":

```
<button type="button" class="navbar-toggle collapsed" data-
toggle="collapse" data-target="#bs-example-navbar-collapse-1" aria-
expanded="false">
    <span class="sr-only">Toggle navigation</span>
    <span class="icon-bar"></span>
    <span class="icon-bar"></span>
    <span class="icon-bar"></span>
</button>
```

        - The three "icon-bar" classes are responsible for making the three bars in the icon.
        - The "data-target" attribute is responsible for showing/hiding the content when the icon is clicked. The value of this attribute should be set to whatever (by ID) you want to show/hide.

- To keep the navbar **Fixed to Top**, include the "navbar-fixed-top" class, and add a minimum of 50px of padding to the top of the <body> (as the navbar itself is 50px high), although it may better to add a little more for extra space.

▪ **Glyphicons**

- To place icons, use the link above to find the class of icon you wish to use (e.g., an envelope icon), and insert it according to the following syntax:

  - The icon should contain no text or child elements.
  - The aria-hidden="true" attribute is used to hide decorative icons on assistive technologies (e.g., **Screen Readers**). If the icon is not decorative and is intended to convey meaning, then that meaning can be added through by using , e.g.:

    Error: Enter a valid email address

  - If you're creating controls with no text (such as a <button> that only contains an icon), you can add an "aria-label" attribute to the control and make its value be a description that will appear on screen readers.
- For additional icons, install **Font Awesome** by placing the entire directory into your project, and then inserting the following into your <head>:
  <link rel="stylesheet" href="path/to/font-awesome/css/font-awesome.min.css">

○ **The Grid System**

▪ Bootstrap includes a fluid grid system that appropriately scales up to 12 columns as the device or viewport size increases. These 12 columns are contained within every instance of . You can then place additional elements within each row and divide them by the following syntax: .

- The "**#**" option determines how many columns (out of 12) the will occupy. In the following example, the first row will have 3 evenly spaced groups that are each 4 columns wide, and the second row will have 2 groups, with the first being 9 columns in width and the second being 3 columns:

```
<div class="container>
    <div class="row">
        <div class="col-md-4">First Symmetrical Group</div>
        <div class="col-md-4">Second Symmetrical Group</div>
        <div class="col-md-4">Third Symmetrical Group</div>
    </div>
    <div class="row">
        <div class="col-md-9">First Asymmetrical Group</div>
        <div class="col-md-3">Second Asymmetrical Group</div>
    </div>
</div>
```

- The "**Size**" option determines the width of the , which determines the "breaking point" at which the elements in a single row will take up a full 12 columns and stack on top of each other, rather than appear side-by-side.
  - Four sizes and breaking points:
    - Extra Small ("col-xs-#") for mobile phones: Auto width.

        - Small ("col-sm-#") for tablets: 750 pixels.
        - Medium ("col-md-#") for smaller monitors: 970 pixels.
        - Large ("col-lg-#") for larger monitors: 1170 pixels.
    - Given the sizes above, if you were to have 4 elements, the 4 elements would appear side-by-side until the screen width dropped below 970 pixels, at which point all 4 would stack on top of each other. If, however, you wanted tablet users (below 970 pixels) to see the first two elements appear side-by-side and the last two elements below, you would say: .
- This system also allows for **Nesting** additional "row" classes within other existing grid elements, e.g.:

```
<div class="container">
    <div class="row">
        <div class="col-md-3 col-sm-6">
            <div class="row">
                <div class="col-sm-6">First half of Group "1"</div>
                <div class="col-sm-6">Second half of Group "1"</div>
            </div>
        </div>
        <div class="col-md-3 col-sm-6">Group "2"</div>
        <div class="col-md-3 col-sm-6">Group "3"</div>
        <div class="col-md-3 col-sm-6">Group "4"</div>
    </div>
</div>
```

- Notes for **Image Galleries**:
    - Visit **Unsplash** for free high-quality images.
    - In order for images to be appropriately sized within its grid, a quick method is to nest the image within the "thumbnail" class, e.g.:

```
<div class="col-lg-4 col-md-6 col-sm-6">
    <div class="thumbnail">
        <img src="files/01.jpg">
    </div>
</div>
```

    - **NOTE**:
        - If you're looking for Pinterest-like presentation of thumbnails of varying heights and/or widths, you'll need to use a third-party plugin such as **Masonry**, **Isotope**, or **Salvattore**.
        - If you don't mind the images being of varying heights, but want to get rid of the white space bug, then see here.
        - If you want to crop all of the images to the same size, then see an example of this here.

- **JavaScript (Basic)**
    - **Writing a JavaScript File**
        - JavaScript must be saved to a file ending in the ".js" extension.
        - To add the JS file to your website, you must use a <script> tag in your HTML file:
          <script src="filename.js"></script>

        - **NOTE**: Use double slashes "//" to make unexecuted **Comments** within your JS file (similar to comments that can be included in HTML and CSS).
    - **The JavaScript Console**
        - Accessible on the "Console" tab in "Developer Tools" in Chrome (accessed by F12).
        - Just as with using the Developer Tools for testing and understanding how your HTML/CSS work (rather than actually writing your HTML/CSS), the JS console serves the same function.
        - **TIP**: Pressing the "up" arrow in the console will bring back previously lines of code, so you do not have to retype it.
    - **Primitives**
        - There are 5 low-level, basic types of data used in JS:
            - **Numbers**
                - Whole, fractional, or negative numbers are all treated the same way.
                - In the console, the browser can perform mathematical processes in the same manner as a calculator by simply typing, e.g., "4 + 10" and pressing "Enter" to have the result returned to you.
                    - One function known as **Modulo** (a.k.a., the Remainder Operator) uses the "%" placed between two numbers (like a "+" for addition), then it will divide the second number into the first number as many times as it can as a whole number, and then gives you the remainder as a result, e.g.:
                      10 % 3 = 1
                      20 % 5 = 0
                      15 % 11 = 4
            - **Strings**
                - All of the text within one pair of **Quotation Marks** constitutes one string. This is also true of numbers that appear within quotes.
                    - You can use either double quotes ("), single quotes ('), or backticks (`), but the pair must match or an error will result.
                        - **IMPORTANT**: If you want to have a string include **Newlines** (i.e., the character that causes a line break to occur when you press Enter), then you must use the **Backtick**. Other types of strings will remain on one line unless you use special escape characters (described below).
                        - Backticks can also be used in conjunction with ${} to **Embed Values** within a string, e.g.:
                          `Half of 100 is ${100/2}.`
                          …translates to:
                          "Half of 100 is 50."
                    - **BUT NOTE**: You can have a single quote within a pair of double quotes (e.g., if you are typing a word has an apostrophe), and it will still be valid.
                - **Concatenation**:
                    - Strings can be added together formulaically to create a single string, e.g.:
                      'Charlie ' + 'Brown' = 'Charlie Brown'
                - **Escape Characters**:

        - Escape characters all start with a backslash ("\"), and they are used to "escape" out of the string to write special characters that would not otherwise be valid within the string. For example, if you want the string to say, "She said "Goodbye!"" without using single quotes around "Goodbye!", then say:
          'She said \"Goodbye!\"'
        - The following link includes a **List of Special Escape Notations**.
    - **Length Property**:
        - Every string has a "length property," which refers to the number of characters (including spaces, numbers, etc.) within that string. For example, by typing the following, the console will tell you that the string "hello" is 5 characters long:
          'hello'.length;
    - You can also access individual characters by using **Brackets**, and placing a number (beginning with zero) within the brackets that corresponds to the position in the string in which the character appears (i.e., its "**Index**"). For example, JS will access the first letter in the string "hello" and return "h" if you say:
      'hello'[0];
        - **TIP**: Rather than manually counting each character, you can determine the index number of the very last character by using ".length" and subtracting by 1.
    - **NOTE**: To determine if something is a number or a string, you can use:
      console.log(typeof var);
- **Booleans**
    - Only two options: true or false
    - **TIP:** To invert the value of a variable from true to false or vice versa, simply use the NOT ("!") operator, e.g.:
      var x = true; // x will return true
      x = !x; // x will return false
      x = !x; // x will return true
- **Null** & **Undefined**
    - These are actually values rather than a true category (no multiple types of "null" or "undefined").
    - Undefined variables are those which are declared but not initialized, whereas null variables are "explicitly nothing."
        - For example, if you run "var age;" without a value, then JavaScript is simply told to make some space for something called "age" but without storing anything there. If you then ask the console to give you a return on "age", it will come back "undefined." If you ask the console to return something that doesn't exist at all, like "color," then it will return an error.
        - By contrast, if you are making a game and making a string for the name of the current player as "var currentPlayer = 'charlie';" and that player dies (game over), you would set "currentPlayer = null;" to make it clear that there is no longer any "current player."

- **Variables**
    - A variable is simply a named container with data stored within. Variables are structured according to the following syntax:

```
var yourVariableName = yourValue;
var name = 'Rusty';
var secretNumber = 73;
var isAdorable = true;
```

- **NOTE**: The naming convention used in JS relies on a system of capitalization known as **camelCase**, in which the first word of the name starts with a lower-case letter, and subsequent words begin with an upper-case letter.

- The name "variable" means that the data stored within that container can vary. For instance, you can begin by setting the following in the console:
  `var name = 'Rusty';`
  However, if you later decide that you want to change the name, then you can simply state "`name = 'Tater';`" which will result in the console returning the value "Tater" when you simply type "name" into the console.
- Math can also be performed with the name of a variable that has a number for its value. For example, if you set a variable with the name of "num" and set its value to 73, you can type "`num + 7`" to receive the result of 80.

- **Useful Built-In Methods**
  - **Alert**
    - Creates a pop-up message for the user when running the following:
      `alert('hello world');`
  - **Console.log**
    - "Prints" data to the JavaScript console (so only those viewing the console can see the text/data):
      `console.log('hello world');`
  - **Prompt**
    - Allows the browser to get input from a user via a pop-up message with a text field in which the user can input information:
      `prompt('What is your name?');`
    - The information obtained through Prompt can also be stored as a variable for later use. For example, the browser may run the following operation:
      `var userName = prompt('What is your name?');`
      Then after the user inputs his name, that name can be recalled by running "username".
    - **NOTE**: When a number is entered into a prompt by the user, it is stored as a string (e.g., 7 becomes "7"). To recall the stored string as a number, use the following syntax: `Number(storedString);`
      - If the prompt intends that the answer only be an answer and not contain any text, it would be most efficient to have the starting variable be something to the effect of:
        `var numberAnswer = Number(prompt('Enter a number.'));`
        or
        `var answer = prompt('Enter a number.');`
        `var numberAnswer = Number(answer);`
      - **ALSO NOTE**: Alternatively, you may use the **Unary Plus** operator ("`+`") to convert a string to a number: `+storedString;`
      - **SEE ALSO**: **parseFloat()**
  - **Clear**
    - Run "`clear()`" to clear the console.

- **Boolean Logic**
  - Boolean logic relies upon **Comparison Operators** in determining whether a particular result should be labeled as "true" or "false". For example, assuming the value of "x = 5", then the following operators would produce the following results:

Operator	Name	Example	Result
>	Greater than	x > 10	False
>=	Greater than or equal to	x >= 5	True
<	Less than	x < -50	False
<=	Less than or equal to	x <=100	True
==	Equal to	x == '5'	True
!=	Not equal to	x != 'b'	True
===	Equal value and type	x === '5'	False
!==	Not equal value or type	x !== '5'	True

    - **Double Equals** performs **Type Coercion**, which takes the two numbers, strings, or variables and tries to turn them into a similar type to compare them. **Triple Equals**, by contrast, does not perform type coercion, so a pure number 5 will not be treated the same way as a string containing the number "5".
      - As a general rule, triple equals should always be preferred, as it is more specific than double equals. For example, if you were to run "var y = null;" and then "y == undefined", the console would return a value of "true" because null and undefined are similar, even though the two are not technically true equivalents.
      - Additional quirks in JS demonstrating the anomalies that arise when using double equals:

```
true == "1"          // true
0 == false           // true
null == undefined    // true
NaN == NaN           // false ("NaN" stands for "not a number")
```

- **Logical Operators**
  - Logical operators are used to "chain together" expressions of Boolean logic. For example, where "x = 5", and "y = 9", you would receive the following results:

Operator	Name	Example	Result
&&	AND	x < 10 && x !==5	False
||	OR	y > 9 || x === 5	True
!	NOT	!(x === y)	True

    - **AND** requires both sides of the equation to be true to return a result of "true." On the other hand, **OR** only requires one or the other to be true to return a result of "true."
    - **NOT** negates the truth (or falsity) of whatever is contained in its parentheses. In the example above, the parenthetical statement says that "5 = 9", which is false. Adding the NOT operator is tantamount to saying, "It is NOT TRUE that '5 = 9' is true," which is ultimately a true statement, thereby returning a result of "true."
  - The following values are treated as being "**Falsy**" in JS (whereas everything else is considered "**Truthy**"):

```
false (NOTE: This false is not within a string.  Any text within a string is "truthy.")
0
""
null
```

undefined
NaN

- **Conditionals**
  - Conditionals are the means by which you add "decisions" to your code. For example, when you log in to a website, there is code that checks the password you type in with the password stored on the site's server (and you are logged in if they match, and given an error message if they do not match).
  - Three Key Conditionals
    - **If**
      - "If you are younger than 18, you cannot enter the venue":
        ```
        if (age < 18) {
            console.log('Sorry, you are not old enough to enter the venue.');
        }
        ```
    - **Else If** always follows an "if" statement as a secondary condition, as it will only run if the preceding "if" statement is false.
      - "If you are between 18 and 21, you can enter but cannot drink":
        ```
        else if (age < 21) {
            console.log('You can enter, but cannot drink.');
        }
        ```
    - **Else** will only run as a last-ditch effort when all preceding "if" and any subsequent "else if" statements are false.
      - "Otherwise, you can enter and drink":
        ```
        else {
            console.log('Come on in.  You can drink.');
        }
        ```
  - **Ternary Operator**
    - The ternary operator is a shorthand method of creating a simple "if/else" conditional by the following syntax:
      condition ? ifStatement : elseStatement
      - For example, to perform an age check, you could state:
        ```
        age >= 18 ? console.log('You can enter.') : console.log('You must leave.');
        ```
- **Loops**
  - **While Loops** repeat code while a condition is true. They are very similar to if statements, except they repeat a given code block instead of just running it once:
    ```
    while (someCondition) {
        // run some code
    }
    ```
    - For example, to have the console print numbers 1-5, you would say:
      ```
      var count = 1;
      while (count < 6) {
          console.log(count);
          count++;
      }
      ```
      - **NOTE**: Using "++" will add a value of 1, but you can add more value, for example, 4, by using "+= 4". For subtracting, just use a "-" sign instead of a "+" sign.
    - If you want the user to enter a certain phrase into a prompt, but do not need exact specificity (only that the user provide the phrase somewhere in the

response to the prompt), then you can use the **Index Of** method. This method returns the first index (starting at 0) at which a given element can be found in the array (in this case, the user's response), or -1 if it is not present.

- For example, if a prompt is set to loop until the user types "yes" (variable named "answer"), and the user instead types "hell yes", then running answer.indexOf('yes') will return a result of 5, because the phrase "yes" begins at index 5 of the user's answer of "hell yes". If the user typed "yes sir", then the result would be 0, because the phrase "yes" begins at index 0. If the user fails to include the phrase "yes" anywhere in his response, then a result of -1 will be returned, due to the fact that the phrase "yes" appears nowhere in the user's answer. Therefore, to make a prompt loop until the user types the word "yes" (and thereby returning a positive index to terminate the loop), the following expression may be used:

```
var answer = prompt('Are we there yet?');
while (answer.indexOf('yes') === -1) {
    var answer = prompt('Are we there yet?');
}
alert('Yay! We made it!');
```

- **SEE ALSO**: **Do…While Loops**

- **Infinite Loops** occur when the terminating condition in a loop is never false. For example, the code below will print 0 forever because "count" is never incremented:

```
var count = 0;
while (count < 10) {
    console.log(count);
}
```

- **For Loops** operate similarly to while loops, but with a different syntax. Most significantly, the variable that is set to initialize the for loop is typically only meant to exist within the confines of that particular for loop:
  - **Syntax**:

```
for (init; condition; step) {
    // run some code
}
```

    - "**Init**" (for "initialize") declares a variable and sets it to some initial value at the start of the loop, "**Condition**" states when this loop should keep running (for as long as the condition returns "true"), and "**Step**" states what should be done at the end of every iteration.
  - Example of how to print numbers 1-5 with a for loop:

```
for (var i = 1; i < 6; i++) {
    console.log(i);
}
```

    - This statement "initializes" by setting "i" as 1, and provides that "i" will be printed to the console and then increased by an increment of 1 for as long as "i" is less than 6.
  - Example of how to print each character in the world "hello":

```
var str = 'hello';
for (var i = 0; i < str.length; i++) {
    console.log(str[i]);
}
```

        - This statement initializes by setting "i" as 0, and provides that the corresponding index of "hello" (as defined by the number at which "i" is currently set) will be printed to the console and then increased by an increment of 1 for as long as the numerical value of "i" is less than the numerical length of the "str" variable (in this case, the word "hello").
    - **SEE ALSO**: **For…In Loops**
- **Functions**
    - Functions allow you to wrap bits of code up (with a suggested limit of 20 lines) into reusable packages. They function by first "declaring" a function, e.g.:

```
function doSomething() {
    console.log("HELLO WORLD");
}
```

      …and then "calling" the function as many times you want, e.g.:

```
doSomething();
doSomething();
doSomething();
doSomething();
```

        - **NOTE**: It is necessary to include the empty parentheses after "doSomething" to actually **Execute** its code; otherwise, the console will simply spit the code back out for you to see in its entirety (i.e., give you the value of "doSomething").
        - Another Syntax: As an alternative to a **Function Declaration**, you can also use a **Function Expression**:

```
var doSomething = function() {
    console.log('HELLO WORLD');
}
```

            - **BUT NOTE**: If the value of doSomething is modified, the function is lost.
    - **Arguments** are how we can write functions that take inputs. Rather than using empty parentheses, you insert the name of an argument within the parentheses following the function name.
        - For example, in order to create a function that can dynamically return the square of any numerical input, you would state:

```
function square(num) {
    console.log(num * num);
}
```

          …and then "**Call**" square with whatever value you "pass in" (i.e., input), e.g.:

```
square(10);     // prints 100
square(4);      // prints 16
square(3);      // prints 9
```

        - Functions can have as many arguments as needed and can generally be given any name. Simply separate the arguments with commas. For example, to calculate the area of a rectangle, you must multiply length and width:

```
function area(length, width) {
    console.log(length * width);
}
```

          …and then call the function, e.g.:

```
area(2, 4)      // prints 8
```

            - **NOTE**: If you want a function to have **Default Parameters** in the event the user does not specify a value, use the following syntax:

```
function area(length = 200, width = 400) {
    console.log(length * width);
}
```

- Other examples of useful functions include (1) being able to add or subtract from a player's score during a game (by inserting a positive or negative number as an argument); or (2) checking a user's login credentials by making their e-mail and password into two arguments, and then running an If statement to determine if they match correctly, or else return an error.
- **NOTE**: All functions have an **Arguments Object**, which is an **Array-Like Object** that can be used to access the arguments in a manner similar to accessing an array item. For example, if you have the following function…

```
function logNumbers(a, b, c) {
    console.log(arguments[0]);
    console.log(arguments[1]);
    console.log(arguments[2]);
}
```

…the console will print the following upon executing logNumbers(2, 4, 6):

```
2
4
6
```

  - **ALSO NOTE**: The arguments object can be useful for constructing functions in which a varied number of arguments may be passed. For example, if you want to create a function that will return the sum of any set of numbers that have been passed as arguments, you could say:

```
function addNumbers() {
    var sum = 0;
    for (var i = 0; i < arguments.length; i++) {
        sum += arguments[i];
    }
    return sum;
}
```

- **The Return Keyword**
  - Every function returns something, but if you don't tell it what to return, it just returns "undefined." In previous examples, functions have been used to print to the console, which is always an undefined return and cannot be used for anything. For example, in the "square()" function above, if you ran square(4), the console would print 16, but it would also return "undefined". Additionally, if you ran…

```
'4 squared is: ' + square(4);
```

…it would return: '4 squared is: undefined' rather than '4 squared is: 16'. That is because the return keyword was not used in the original function. Instead, you must state:

```
function square(num) {
    return num * num;
}
```

    - The use of the return keyword in this example also provides the additional benefit of allowing the returned result to be stored as a variable. For example, if you were to next run var result = square(104), then you would receive the returned value of 10816 after calling "result".

- For another example of how the return keyword allows you to capture a returned value in a variable, consider this function which capitalizes the first character in a string:

```
function capitalize(str) {
    return str.charAt(0).toUpperCase() + str.slice(1);
}
```

This is a functioned named "capitalize" that takes an argument named "str", converts the first character (at index 0) of "str" to upper case, and then adds that first character to the beginning of the remaining letters in "str" beginning at the "slice" that starts at index 1 (the second character). If you run the following:

```
var city = "paris";
var capital = capitalize(city);
```

…then you would receive an output of "Paris" upon calling "capital", because the capitalize() function performed its return on the argument "str" (in the form of the substituted variable of "city").

  - **IMPORTANT:** Once a return keyword is triggered, it will **STOP THE EXECUTION OF A FUNCTION**. Therefore, if a function includes an if statement, such as:

```
function capitalize(str) {
    if (typeof str === 'number') {
        return "That's not a string."
    }
    return str.charAt(0).toUpperCase() + str.slice(1);
}
```

…if the user inputs a number rather than a word to the "str" argument, he will receive a return of "That's not a string.", and there will be no further execution of the second return statement.

- **Higher Order Functions** are functions into which another function can be "**Passed**" as an argument. For example, JS includes a built-in function called setInterval(). Within the parentheses of this function, you first place the name of another function that you want setInterval() to run, and then follow that with the interval (in milliseconds) that the script should execute until terminated. Thus, if you want to have another function (which has already been previously set) run every second, you would state:

```
setInterval(anotherFunction, 1000);
```

Upon executing this statement, setInterval() will run anotherFunction() every 1000 milliseconds.

  - **NOTE**: When the example above is executed, it will return a number (e.g., 661) in the console. To stop the function from running, simply run clearInterval(661).
  - **ALSO NOTE**: If, in the above example, you want a function to run every second, but you don't want to name that function ahead of time, you can insert an **Anonymous Function**:

```
setInterval(function() {
    // code that you want to run every second;
}, 1000);
```

- **Scope**
  - Scope refers to the context in which JavaScript is being executed (i.e., which properties and variables are "visible" within a particular function, yet are excluded by or different in another function). For example, if you declare the

following function:

```
function getNumber() {
    var x = 40;
    console.log(x);
}
```

…the console will print 40 if you call the function. However, if you simply call "x", then the console will return an error saying "x" is undefined. This is because the scope in which "x" exists in the function differs from the "**Global**" scope, where "x" does not exist. Conversely, if you were to later define "x" as a variable with a value in the global scope, and then run the function, the function will still print 40 notwithstanding the value of "x" in the global scope.

- **HOWEVER**: If you were to have set "`var x = 40;`" in the global scope, and then created the following:

  ```
  function getNumber() {
      console.log(x);
  }
  ```

  …the Console will still print 40. This is because when a variable is defined outside of a function, you still have access to it inside of that function **AS LONG AS** the interior of the function does not redefine that variable. In other words, a more-specific "child" scope is governed by values set in the "parent" scope unless otherwise stated. The converse, however, is not true, and the parent scope will not be governed by anything set in the child scope.

  - **BUT NOTE**: If the variable was set FIRST in the parent scope, then directly modified in the function, e.g.:

    ```
    var x = 40;
    function getNumber() {
        x = 80;
        console.log(x);
    }
    ```

    …then simply calling "x" in the console will print a value of 80. This is because the function is not declaring a "new" variable "x" within itself; rather, it recognized that there was already a variable "x" existing outside of itself and redefined that variable externally while incorporating the redefined value internally. **HOWEVER**, had the function stated "`var x = 80;`", then the function would have created its own "x" as an internal variable, and calling "x" globally would print 40.

- **Arrays**
  - **Basics**
    - Arrays are high-level data structures (ways of holding information using JS) containing lists of data. For example, rather than making four "friend" variables:

      ```
      var friend1 = 'Mac';
      var friend2 = 'Dennis';
      var friend3 = 'Charlie';
      var friend4 = 'Dee';
      ```

      ...you could use an array by making a comma-separated list, like so:

      ```
      var friends = ['Mac', 'Dennis', 'Charlie', 'Dee'];
      ```

- Arrays are indexed starting at 0 (just like characters in a string). The index method is used to get data out of the array, e.g.:
  ```
  console.log(friends[0])              // "Mac"
  friends[1] + ' <3 ' + friends[2]     // "Dennis <3 Charlie"
  ```
- Arrays are useful in that they can be updated by simply stating, for example:
  ```
  friends[0] = 'Frank';
  ```
  …and all lines of code that would have otherwise produced "Mac" now say "Frank".
- Arrays are also useful in that new data can be added to them without modifying the original line of "var friends …" Rather, you can simply assign a new value/string (or any other data, including "true," "null," or even other arrays) to the next available index, and it will be added to the array, e.g.:
  ```
  friends[4] = 'Cricket';
  ```
- To initialize an array without yet adding data to the array, you can simply state:
  ```
  var friends = [ ];
  ```
  …or…
  ```
  var friends = new Array();
  ```
- Arrays also have a length property that can be called by using ".length", e.g.:
  ```
  var friends = ['Frank', 'Dennis', 'Charlie', 'Dee', 'Cricket'];
  friends.length;      // returns 5 (because there are 5 friend values)
  ```
- Arrays can be nested within arrays like so:
  ```
  var friendGroups = [
      ['Friend 1', 'Friend 2'],
      ['Friend A', 'Friend B']
  ];
  ```
  - **NOTE**: When arrays are nested, you can call a specific index within a nested array like so :
    ```
    console.log(friendGroups[1][0]);      // "Friend A" (takes the second
                                          // index of the first array, and the
                                          // first index of the nested array)
    ```

- **Built-In Array Methods**
  - For a complete list of Array Methods, refer to the "Methods" section in the MDN link to "Arrays" above.
  - Five Important Methods:
    - **Push/Pop**
      - Use push() to add a value to the **END** of an array (rather than having to count or use "length" to figure out which index in the array you would have to manually add the value to), e.g.:
        ```
        var colors = ['red', 'orange', 'yellow'];
        colors.push('green');        // ['red', 'orange', 'yellow', 'green']
        ```
      - If you were to then state "colors.pop();", then "green" would be removed, because pop() removes the last item in an array.
        - **NOTE**: When pop() is used, the console actually returns the removed variable. Because a value is returned, that means it can be stored as a variable if so stated:
          ```
          var removedColor = colors.pop();      // stores "green"
          ```
    - **Unshift/Shift**

        - unshift() and shift() operate in the same way as push() and pop(), except that they add/remove values to the **BEGINNING** of an array.
            - **NOTE**: While counterintuitive, remember that unshift() is used to ADD, not to remove.
    - **indexOf**
        - indexOf() takes an argument and tries to find that argument in a given array. If it finds it, then it will return the FIRST index of where it's found in the array, e.g.:
          var friends = ['Frank', 'Dennis', 'Charlie', 'Dee', 'Cricket'];
          friends.indexOf('Dennis'); // 1
            - **NOTE**: Remember that indexOf() will only return the **FIRST** index, so if the same argument appears elsewhere in the array, that index will not be returned.
            - **ALSO**: If an argument is **NOT** present, "-1" will be returned.
    - **Slice**
        - slice() is used to copy different portions of an array, e.g.:
          var fruits = ['Banana', 'Orange', 'Lemon', 'Apple', 'Mango'];
          var citrus = fruits.slice(1, 3);
          …leads to…
          var fruits = ['Banana', 'Orange', 'Lemon', 'Apple', 'Mango'];
          var citrus = ['Orange', 'Lemon'];
            - **NOTE**: The first number in slice argument signifies the first index that will be copied ("Orange"), and the second number represents where the slice will end **AND IS NON-INCLUSIVE** ("Apple"). Thus, only "Orange" and "Lemon" are copied out.
            - **TIP**: If you want to **Duplicate** an entire array, simply omit the numbers in the slice argument, and all data in the array will be copied over.
    - **Splice**
        - splice() changes the contents of an array by removing and/or adding elements at any point in the index.
            - To **Remove** an element, use this syntax:
              var fruits = ['Banana', 'Orange', 'Lemon', 'Apple', 'Mango'];
              fruits.splice(1, 2);
              …leads to…
              var fruits = ['Banana', 'Apple', 'Mango'];
                - The splice syntax works by using the first number as the index number at which to start changing the array (in this case, index 1, which is "Orange"). The second number is the **Delete Count**, which can be 0 (to not delete the starting index) or any other positive number to indicate how many indices (beginning from your starting index) are to be deleted (in this case, delete two indices starting with index 1, which means deleting indices 1 and 2, i.e., "Orange" and "Lemon").

- To **Add** an element, use this syntax:
  ```
  var fruits = ['Banana', 'Apple', 'Mango'];
  fruits.splice(1, 0, 'Orange', 'Lemon');
  ```
  …leads to…
  ```
  var fruits = ['Banana', 'Orange', 'Lemon', 'Apple', 'Mango'];
  ```
  - In this case, splice() is told to:
    (1) look at index 1 (starting as "Apple"),
    (2) abstain from deleting the element at index 1,
    (3) insert "Orange" as the new index 1, and
    (4) insert "Lemon" after "Orange" (this addition process can be repeated indefinitely and will ensure that "Mango" remains as the last index).

- **Array Iteration** is the process in which a code will run through every element in an array, rather than just one element. For example, message board comments are stored in an array, and they are only displayed when some code runs through that array and creates HTML content for each comment stored in the array.
  - **For Each Iteration** (newer method):
    ```
    var colors = ['red', 'orange', 'yellow', 'green'];
    colors.forEach(function(color) {        // "color" is a placeholder to represent a
        console.log(color);                 // singular instance of any given element in
    });                                     // the collection; call it whatever you want
    ```
    - Essentially, the "color" placeholder argument merely represents the value of every element in the array. Thus, the code is calling the console.log() function "for each" element, and then passing in each element's value.
    - Alternatively, you could set a function beforehand, and then run it through forEach():
      ```
      var colors = ['red', 'orange', 'yellow', 'green'];
      function printColors(color) {
          console.log(color);
      }
      colors.forEach(printColors);
      ```
    - **IMPORTANT:** forEach() executes a callback function once for each element in the array in ascending order. The callback function is expected to have **AT LEAST 1 (BUT UP TO 3) ARGUMENTS** separated by commas). The first argument represents the **Element** in the array. The second argument represents the **Index** of said element. The third argument represents the **Array** that forEach() was called on (which will be the same for every iteration of the loop). Thus, if you want to print both the contents of a list along with its index number, you would state:
      ```
      var colors = ['red', 'orange', 'yellow', 'green'];
      function printColors(x, i) {
          console.log(i + ': ' + x);
      }
      colors.forEach(printColors);
      ```
      …which would yield:
      0: red
      1: orange

2: yellow
3: green

- **For Loop Iteration** (older method):

```
var colors = ['red', 'orange', 'yellow', 'green'];
for (var i = 0; i < colors.length; i++) {
    console.log(colors[i]);
}
```

    - The above code will start by printing index 0 of "colors" to the console, and repeat the process until reaching index 3. This is possible because the length of "colors" is 4, and the final index value is 3. As the final index in an array will always be one digit smaller than the length of the array, this loop will necessarily run until all array values are printed.

- **Objects**
    - **Basics**
        - An object, like an array, is a collection of related data (of any type) and/or functionality. Unlike an array, objects are meant to handle data that does not follow a logical and progressive order. Objects have no built-in order. Objects use the following **Syntax** (**Key-Value Pairs** where the "key" (name) is separated from the value by a colon, and all key-value pairs are separated by commas):

```
var objectName = {
    member1Name: member1Value,
    member2Name: member2Value,
    member3Name: member3Value
};
```

        - **Retrieving Data** from an object is done by either **Bracket or Dot Notation**:

```
var dog = {
    name: 'Rusty',
    breed: 'Mutt',
    age: 3
};
…bracket notation:
console.log(dog['name']);       // "Rusty"
…dot notation:
console.log(dog.name);          // "Rusty"
```

            - **Usage Notes:**
                - You **CANNOT** use dot notation if the property starts with a number:

```
someObject.1blah          // invalid
someObject['1blah']       // valid
```

                - You **CANNOT** use dot notation if the property name has a space in it (not a good practice anyway):

```
someObject.fav color      // invalid
someObject['fav color']   // valid
```

                - You **CAN** look up things using a variable when using bracket notation:

```
var str = 'name';
someObject.str            // doesn't look for "name"
someObject[str]           // evaluates str and looks for "name"
```

- **Updating Data** in an object is done just like an array (access a property and reassign it):
  ```
  objectName['propertyName'] = newPropertyValue;
  ```
  …or…
  ```
  objectName.propertyName = newPropertyValue;
  ```
- **Creating Objects** can be done in one of three ways:
  - Make an empty object and then add to it:
    ```
    var person = {};
    person.name = 'Travis';
    person.age = 21;
    person.city = 'LA';
    ```
  - Make an object by executing a built-in function and then add to it:
    ```
    var person = new Object();
    person.name = 'Travis';
    person.age = 21;
    person.city = 'LA';
    ```
  - Make an object all at once:
    ```
    var person = {
        name: 'Travis',
        age: 21,
        city: 'LA'
    };
    ```
- Objects can be inserted within arrays, so that each item within an array is its own object. This is typical when working with an online array of comments, posts, friends, etc.:
  ```
  var posts = [
      {
          title: 'Article 1',
          author: 'Author 1',
          comments: ['Awesome post.', 'Terrible post.']
      },
      {
          title: 'Article 2',
          author: 'Author 2',
          comments: ['I love this.', 'I hate this.']
      }
  ];
  ```
  - To access the title of the first post, you would state:
    ```
    posts[0].title
    ```
  - To access the second comment of the second post, you would state:
    ```
    posts[1].comments[1]
    ```

- **Adding Methods to Objects**
  - A "method" (e.g., alert(), prompt(), indexOf(), etc.) is a function that is a property inside of an object. console.log() is an example of a method that has been added to an object. "console" is a built-in object with "log()" as a built-in function. By putting the two together (console.log), you are having JS look at the "console" object and run its "log()" function.

- Adding methods to objects (rather than having them stand alone) is useful for grouping things together logically to avoid a "**Namespace Collision**." For example, to have a function return "WOOF" when a "dog" object "speaks," but have it return "MEOW" when a "cat" object speaks," you would state:

```
var pets = [
    {
        type: 'dog',
        speak: function() {
            return 'WOOF';
        }
    },
    {
        type: 'cat',
        speak: function() {
            return 'MEOW';
        }
    }
];
```

…so that pets[0].speak() will return "WOOF", and pets[1].speak() will return "MEOW".

- **This**
  - The "this" keyword behaves differently depending on the context. The following example illustrates how "this" behaves when used in methods that have been added to an object.
    - Suppose you're making an app that has some comments, and you want to have some comment data stored in an object:

```
var comments = {
    data: [
        'Good',
        'Bad',
        'Okay'
    ]
};
```

    - Now suppose you want to make a method called "print()" and add it to "comments". You could define a separate function, and then "print" the data by running "print(comments.data)":

```
function print(arr) {
    arr.forEach(function(elem) {
        console.log(elem)
    });
}
print(comments.data);          // "Good,Bad,Okay"
```

    - But if you want to add the print function to the comments object, you can use "this" to refer to the data that is contained in the same object (i.e., "this" object):

```
var comments = {
    data: [
        'Good',
        'Bad',
```

```
            'Okay'
        ],
        print: function() {
            this.data.forEach(function(elem) {
                console.log(elem)
            });
        }
    };
```

        - The "this" keyword essentially allows you to use the values contained in comments.data inside of comments.print without the need for any additional argument. Thus, by writing "this" inside of a method, it refers to the object that the method is defined in.
- **DOM ("Document Object Model") Manipulation**
    - **Background**
        - The DOM is the interface between JS and HTML/CSS (for making content interactive). When you load an HTML page, the browser turns every HTML tag into a JS object that we can select and manipulate, and all of these are stored inside of a "document" object. To view the HTML elements converted into JS objects (i.e., the DOM), type the following into the console:
          console.dir(document);
        - **NOTE**: When performing DOM manipulation, be sure that the JS files are loaded after all HTML is written (or else you will be trying to manipulate HTML that does not yet exist to the JS files, and it will not work.)
    - **Important DOM Selector Methods** (the following five methods have been added into the "document" object):
        - **document.getElementById()**
            - Takes in an ID name and returns the one element that matches that name (because an ID can only be associated with one element in a page). For example, if you have an HTML element with the ID of "highlight", you can return that element by stating:
              `var tag = document.getElementById('highlight');`
                - You can then use `console.dir(tag)` to see all of the properties contained in the "highlight" object.
        - **document.getElementsByClassName()**
            - Operates in the same way as `getElementById()`, but applies to all elements sharing the same class. If you want to return a list of all elements with the class "bolded", you would state:
              `var tags = document.getElementsByClassName('bolded');`
                - You can then simply state "tags" to see all of the properties for all "bolded" objects, and you can use `console.log(tags[0])` to access the first index of the **Node List** (not technically an array, but is array-like) containing the "bolded" objects.
                    - The node list is "array-like" in the sense that you can access individual indices, but you **CANNOT** apply `forEach()` to it (because there is no `forEach()` function defined for these node lists; they are defined for arrays).

        - **BUT NOTE**: You can use the "**Array.from()**" method to convert node lists and HTML collections into arrays.
    - **NOTE**: You can also access a specific index (e.g., the first index) more directly by simply stating:
      ```
      var tags = document.getElementsByClassName('bolded')[0];
      ```
- **document.getElementsByTagName()**
    - Operates in the same way as the two above, but it refers to a general HTML tag (such as <li>, <ul>, <h1>, <head>, <body>) rather than a specific class/ID. If you want to return a list of all elements with the <li> tag, state:
      ```
      var tags = document.getElementsByTagName('li');
      ```
- **document.querySelector()**
    - Operates in the same fashion as the methods above, but does so by using a **CSS**-Style Selector (i.e., "#" for ID, and "." for class). **BUT**, it differs significantly by only returning the **FIRST MATCH**. For example, to if you want to return an HTML element with the ID of "highlight", you would state:
      ```
      var tag = document.querySelector('#highlight');
      ```
        - If "highlight" were a class, you would use a period instead of an octothorp, but note that you will only return the first match.
        - This works for any syntax that would be valid in CSS, e.g.:
          ```
          var tags = document.querySelector('li a.special');
          ```
        - query.Selector can also take in basic **HTML** tag names like <h1>:
          ```
          var tag = document.querySelector('h2');
          ```
          …but, again, only for the first match.
        - If you want to select a **Specific Type** of element (e.g., a numerical input), you would use brackets and state:
          ```
          var tag = document.querySelector('input[type="number"]');
          ```
- **document.querySelectorAll()**
    - Operates in the same manner as query.Selector, except it returns a node list of **ALL** elements (as objects) that match a given CSS-style selector.

- **Manipulating Style**
    - A DOM object's **Style** property is one way to manipulate an HTML element's style, e.g.:
      ```
      // select your object:
      var tag = document.getElementById('highlight');
      // manipulate your object:
      tag.style.color = 'blue';
      tag.style.border = '10px solid red';
      tag.style.fontSize = '70px';
      tag.style.background = 'yellow';
      tag.style.marginTop = '200px';
      ```
        - **NOTE**: This is not actually the best way to manipulate an object, as it is very repetitious (not "dry" code), but more significant is the "**Separation of Concerns**" (the principle that HTML, CSS, and JS should each be responsible for their own separate domain, and crossover between the

three should be avoided – HTML should be pure structure, CSS should be pure presentation, and JS should be pure behavior).

- Rather than changing multiple style properties in JS, you should turn them on/off by using the CSS file. For example, you can define a CSS class, select an element in JS, and add the class to its **classList**:

```
// define a class in CSS:
.toggleWarning {
    color: red;
    border: 10px solid red;
}
// add the .toggleWarning class to a select object in JS:
var tag = document.getElementsByTagName('h1');
tag.classList.add('toggleWarning');
```

  - If you so choose, you can later remove the class by stating: `tag.classList.remove('toggleWarning');`
  - A similar useful method is **toggle**, which operates by (1) adding the specified class if it is not present in the class list, and (2) removing the specified class if it is present in the class list: `tag.classList.toggle('toggleWarning');`
  - **NOTE**: A class list is technically **NOT AN ARRAY**. This is why you must use add() and remove() rather than push() and pop().

- **Manipulating Text and Content**
  - The **textContent** property retrieves the text inside of an HTML element. "Text" is defined as anything between the HTML tags but not including any tags contained within (it extracts only **Plain Text**). The method can be used to alter the text by the following syntax:
    `tag.textContent = 'blah blah blah';`
    - **NOTE**: Because this method only works in plain text, no modifiers like or will be preserved.
  - The **innerHTML** property is used to manipulate both **HTML Text and Inner Element**s.
  - **IMPORTANT SYNTAX NOTE**: When manipulating text and content (or style), it is not necessary to set a variable first. Instead, you can simply add the textContent or innerHTML property after the selector method, e.g.:
    `document.querySelector('h1').textContent = 'Heading 1';`
- **Manipulating Attributes**
  - The attributes of an HTML element (e.g., href, src, or anything else following the name="text" syntax, such as id, class, etc.) can be modified by using **getAttribute()** and **setAttribute()** to read and write attributes. For example, to return the URL for the following:
    `<a href="https://www.google.com/">Search</a>`
    …you would state:

```
var link = document.querySelector('a');
link.getAttribute('href');
```

    …and to modify the link, you would state the attribute as the first argument in the method, and the new URL as the second argument:
    `link.setAttribute('href', 'https://www.yahoo.com/');`

- **NOTE**: When manipulating **Images**, the "src" attribute may be ignored if the "srcset" attribute has also been set. In such cases, srcset must be modified.

- **DOM Events**
  - The **MDN Event Reference** contains all of the different events that are recognized by the DOM application programming interface (**API**). Some common events to be most familiar with are "click", "change", "mouseover", and "mouseout".
  - Events trigger the running of code when a certain action has been taken, as opposed to automatically executing when the page loads (e.g., clicking a button, hovering over a link, pressing a key, etc.). The event must be selected and then attached to a specific element by means of an **Event Listener** (e.g., "Listen for a click on this <button>." To add a listener, use the **addEventListener** method through this syntax:
    element.addEventListener('type', functionToCall);
    - For example, to have a message print to console when the first button in a document is clicked, you would state:

```
var button = document.querySelector('button');      // select element
button.addEventListener('click', function() {       // add event listener
    console.log('The button has been clicked.');    // run code
});
```

      - It is not necessary to use an anonymous function. The same code could be executed as follows:

```
var button = document.querySelector('button');
button.addEventListener('click', printConsole);
function printConsole() {
    console.log('The button has been clicked.');
}
```

        However, unless there is a need to use a named function again somewhere else outside of the click listener, it is better to simply use the anonymous function.
    - **NOTE**: You can have more than one listener on a given element. They will execute in the order that they were added.
    - **THIS NOTE**: Inside of a listener, the "this" keyword refers to the item that was clicked on (or hovered upon, or which a keypress referred to, etc.). So whatever element is attached to the addEventListener is the event to which "this" refers. This is useful in situations where you want a large number of similar but individual (sibling) elements to run specific code only upon itself when activated. For example, if you have a <ul> with a dozen <li> tags within, and you want each individual <li> to change its color when clicked, you can use a for loop and "this" as follows:

```
var lis = document.querySelectorAll('li');
for (var i = 0; i < lis.length; i++) {
    lis[i].addEventListener('click', function() {
        this.style.color = 'pink';
    });
}
```

- **JavaScript (Intermediate)**

- **This**
  - "This" is a reserved keyword, which means it cannot be set as the value of any variable. The value of "this" is determined upon execution (i.e., "**Execution Context**").
  - There are **Four Rules** governing how the value of "this" is determined:
    - **Global Context**
      - This rule applies when you see "this" applied outside of a declared object. When you see the word "this" in the global context, its value refers to the **Global Object**, which is the **Window** in the browser environment. Thus, if you were to console.log(this), you would return "window".
      - In essence, every variable that you declare in the global context is in fact attached to the window object, which is what allows "this" to work in the global context without an object being declared. For example, you can create the following variable:
        ```
        var person = 'Ellie';
        ```
        …which can then be accessed as:
        ```
        window.person
        ```
        …and which is also synonymous with:
        ```
        this.person
        ```
      - **NOTE:** Even when you use "this" inside of a **Function**, its value is still the global object if no other object is declared within the function, e.g.:
        ```
        function variablesInThis() {
            this.person = 'Ellie';        // still the same as var person = 'Ellie'
        }
        ```
        - **HOWEVER**: The above process is considered bad practice for setting global variables, as global variables should be declared at the top of your code (with the value either declared immediately or at a later time within a function). If you want to prevent the accidental creation of global variables in a function, you can state **"use strict"** (in quotes) at the top of your JS file, which will return a **TypeError** indicating that "this" is **Undefined** upon execution.
    - **Object/Implicit**
      - When "this" is found inside of a **Declared Object**, its value will always be the **Closest Parent Object**. For example, given the following object…
        ```
        var person = {
            firstName: 'Ellie',
            sayHi: function() {
                return 'Hi ' + this.firstName;
            }
        }
        ```
        …executing "person.sayHi()" will return "Hi Ellie" because "this.firstName" is equivalent to stating "person.firstName" due to "person" being the closest parent object to "this".
      - **NOTE**: Caution must be exercised with **Nested Objects**, as a reference to properties beyond the scope of the nested object may result in undefined values being returned when using "this" within the nested object.
    - **Explicit**
      - In order to avoid problems associated with nested objects, you can set the context of "this" by using the **Call**, **Apply**, or **Bind** methods (which can only be used by **Functions**, and not by any other data type).

- **Call** uses the following syntax:
  function.call(thisArg, arg1, arg2, …);
  - The first argument is whatever object you want the value of "this" to be. The subsequent arguments are for any optional parameters to be used in the function in which you are changing the context of "this".
    - **NOTE**: When the call method is used on a function, that function is immediately invoked.
  - Consider the following example:

```
var firstPerson = {
    firstName: 'John',
    lastName: 'Doe',
    fullName: function() {
        return this.firstName + ' ' + this.lastName;
    }
}
var secondPerson = {
    firstName: 'Mary',
    lastName: 'Jane'
}
```

    - If you want to return the full name of Mary Jane, there is no need to duplicate the fullName function from the first person (John Doe). Rather, you need merely state…
      firstPerson.fullName.call(secondPerson);
      …because using the "secondPerson" argument in call will result in the "this" keyword from the "firstPerson" object actually being applied to the "secondPerson" object instead (i.e., Mary Jane).
- **Apply** uses the following syntax:
  function.apply(thisArg, [argsArray]);
  - Apply is almost identical to call except it only takes two arguments. The first argument operates the same as call, while the second is an **Array** (or **ARRAY-LIKE OBJECT**) that will be passed to the function in which you are changing the value of "this".
    - **NOTE**: When the apply method is used on a function, that function is immediately invoked.
  - Consider the following example:

```
var firstP = {
    name: 'John',
    addNumbers: function(a, b, c, d) {
        return this.name + ' calculated ' + (a+b+c+d);
    }
}
var secondP = {
    name: 'Jane'
}
```

- Stating "firstP.addNumbers(1, 2, 3, 4)" will return "John calculated 10". You can do the same for Jane by either using call:
  ```
  firstP.addNumbers.call(secondP, 1, 2, 3, 4);
  ```
  …or by using apply:
  ```
  firstP.addNumbers.apply(second, [1, 2, 3, 4]);
  ```

- **Bind** uses the following syntax:
  ```
  function.bind(thisArg, arg1, arg2, …);
  ```
  - Bind is almost identical to call except that instead of invoking the function immediately, bind returns a function definition. This allows you to save a function with different values of "this" and invoke them at a later time (known as **Partial Application**), as you may not initially know all of the arguments that will be passed to a function.
    - In the example for apply above, you might know that you always want to add 1 and 2 for the second person, but not know what other numbers will need to be calculated. You could first state:
      ```
      var sPCalc = firstP.addNumbers.bind(secondP, 1, 2);
      ```
      …and then include more numbers on execution:
      ```
      sPCalc(3, 4)     // returns "Jane calculated 10"
      ```
  - Bind is also important when working with asynchronous code (e.g., **setTimeout**), as it allows you to set the context of "this" for a function to be called at a later point in time. Consider the following:
    ```
    var person = {
        name: 'John',
        sayHi: function() {
            setTimeout(function() {
                console.log('Hi ' + this.name)
            }, 1000);
        }
    }
    ```
    - Although you may expect "Hi John" to be displayed in the console after 1 second, this does not occur. This is because setTimeout() does not run console.log() concurrently with the creation of the "person" variable; rather, it only executes after the timer expires. As a result, setTimeout() is run in the **Global Context** as a **Window Object**. Thus, "this" actually refers to window rather than "person", because window is the value of "this" upon execution. You can fix this by stating:
      ```
      var person = {
          name: 'John',
          sayHi: function() {
              setTimeout(function() {
                  console.log('Hi ' + this.name)
      ```

```
        }.bind(this), 1000);
    }
}
```

- This may appear to be confusing, as you are using "this" to change the value of "this". However, at the time "person" is **CREATED**, "this" does in fact refer to "person" because "person" (not window.setTimeout) is the nearest parent object to "bind(this)". Here, "bind(this)" is equivalent to "bind(person)".

- **New**
  - The new operator is used together with a **Constructor Function** to create a new object. Inside the constructor function definition, the keyword "this" refers to the object that is created. When the new operator is used, an implicit return "this" is added to the constructor function.
    - **NOTE**: A popular convention in JS is to capitalize the names of **Constructors**, which basically operate as model templates upon which new objects will be based.
  - For example, you can create a constructor function for a person's name:

```
function Person(first, last) {
    this.firstName = first;
    this.lastName = last;
}
```

    - In this context, "this" would apply to window in the global context (as there is no object has been declared). However, when paired with the new operator, "this" will instead refer to the newly created object that has been stored as a variable. For example…

```
var john = new Person('John', 'Doe');
```

      …results in "this.firstName" being equal to "john.firstName" (as "john" is the new object) and "this.lastName" being equal to "john.lastName".

- **Object-Oriented Programming ("OOP")**
  - **About**
    - OOP is a programming model that uses blueprints to create objects. Such blueprints are conventionally called **Classes**, and the created objects are called **Instances**. The goal is to make classes modular so they can be reused and shared among all parts of an application. JS does not have "built-in" classes in version ES5, but they can be implemented via constructor functions and objects.
  - **New Operator**
    - As noted in the "this" section above, the new operator is used together with a constructor function to create a new object. The new operator works by:
      (1) creating an empty object;
      (2) setting the keyword "this" to be that empty object;
      (3) adding an implicit "return this" line to the end of the function; and
      (4) adding a property onto the empty object called "__proto__" (a.k.a., "**Dunder Proto**", which links the created object to the prototype property on the constructor function.
  - **Multiple Constructors**

- When working with multiple constructors, you can make use of "this" and call to optimize your code, e.g.:

```
function Dog(name, age) {
    this.name = name;
    this.age = age;
    this.bark = function() {
        console.log(this.name + ' just barked!');
    }
}
function Cat(name, age) {
    Dog.call(this, name, age);
    this.meow = function() {
        console.log(this.name + ' just meowed!');
    }
}
```

    - **NOTE**: It may be confusing to see that "this" is used as the first argument in Dog.call() of Cat() to redefine "this" as used in Dog(). However, remember that the constructor will be used with the new operator. This means that, when new is applied to Cat(), a **NEW OBJECT** will be created in which all "this" keywords in Cat() will refer to the new cat object (to which "this" will now apply because the cat is the closest parent object). Thus, you can use "this" as the first argument in Dog.call() to redefine all "this" keywords in Dog() to refer to the newly created Cat() object.
- For further optimization, you can use "this" and apply together with a built-in JS object called "**arguments**", which is an array-like object corresponding to the arguments passed to a function. As "arguments" is an array-like object, it can be used with apply as a shorthand, e.g.:

```
function Cat(name, age) {
    Dog.apply(this, arguments);
    this.meow = function() {
        console.log(this.name + ' just meowed!');
    }
}
```

- **Prototypes**
    - **Basics**
        - Prototypes are the mechanism by which JS objects inherit features from one another. Upon creation, all constructors possess a prototype property, which is an object that can have properties and methods attached to it. The prototype property has two properties: (1) "constructor", which relates back to the definition of the constructor itself; and (2) "__proto__", which allows properties or methods of the prototype to be accessed by any object created from the constructor.
            - **NOTE**: The "__proto__" property of a created object is identical to the prototype property of the constructor. Thus, if you created an object from a Cat() constructor, the following would be true: catObject.__proto__ === Cat.prototype
    - **Prototype Chain**

- If you want to add a property or method to all objects that will be (or, more significantly, already have been) created by a constructor, you need only add that property/method to the prototype of the constructor. For example, you have already created two cats (named "button" and "kiwi") from a Cat() constructor, and the only properties in the constructor are "name" and "age". However, you now want to add a property called "isMammal", which is to be set as "true" for all cats. You can state…
  Cat.prototype.isMammal = true;
  …which then results in the following statements returning true:
  button.isMammal; // true
  kiwi.isMammal; // true
- In the example above, if you were to access "button" to view its properties and methods, you would not immediately see "isMammal" as a property directly under "button" as you would with "name" and "age". Rather, you would have to access the "__proto__" property under "button" to see "isMammal". Nevertheless, you are still able to access "button.isMammal" without saying "button.__proto__.isMammal". This is because JS follows the prototype chain (by following a line of "__proto__" properties) to locate an object's properties or methods.
  - The chain here works as follows:
    (1) If the requested property/method is not found directly in the cat object, check the cat's "__proto__".
    (2) If the cat's "__proto__" (which itself an object constructed by JS's built-in **Object()** constructor) does not contain the property/method, then refer to the Object() constructor's "__proto__" nested under the cat's "__proto__".
    (3) If the Object() constructor's "__proto__" does not contain the property/method, then the result is "undefined" and the inquiry goes no further, as the Object() constructor is the last link in the chain.
  - **NOTE**: This manner in which JS finds methods can be applied to all created objects. For example, all arrays are created by JS's built-in Array() constructor, which contains a prototype object to which the "push" method (among others) is attached. If you create an array named "cats" and you want to push a new cat into the array, you are able to say "cats.push('Stumpy')" rather than "cats.__proto__.push('Stumpy')" due to prototype chaining.
- The prototype chain allows for further **Code Optimization**. For example, if you have a Dog() constructor and you want to apply a bark() method to each dog created by Dog(), you would have a lot of repetitive code if you inserted the bark() method directly into the Dog() constructor, as the bark() method would be repeated in every created dog object with the exact same value each time. But by taking advantage of the linking capabilities of the prototype chain, you could simply state:

```
function Dog(name, age) {
    this.name = name;
    this.age = age;
}
Dog.prototype.bark = function() {
```

```
        console.log(this.name + ' just barked!');
    }
```

- **Closures**
  - **About**
    - A closure is a function that makes use of variables defined in outer functions that have previously returned. Consider the following example:

```
function outerFunc() {
    var outerData = 'closures are ';
    return function innerFunc() {
        var innerData = 'awesome';
        return outerData + innerData;
    }
}
```

      - In this example above, if you were to simply call outerFunc(), you would only return the definition of the inner function, which would appear as:

```
function innerFunc() {
    var innerData = 'awesome';
    return outerData + innerData;
}
```

      - However, as the result of outerFunc() is itself a function, this gives you the ability to call innerFunc() immediately upon calling outerFunc() by simply stating **outerFunc()()**, which returns, "closures are awesome". Because innerFunc() makes use of the returned outerData variable defined in outerFunc(), that makes innerFunc() a closure.
        - **NOTE**: If an inner function does not make use of any returned external variables, then it is merely a **Nested Function**.
    - **NOTE**: When working with closures, you may either call the inner function right away (via "yourFunc()()"), or you can store the result of the outer function to be used by the inner function at a later time (similar to how bind() works), e.g.:

```
function outer(a) {
    return function inner(b) {
        return a + b;
    }
}
var storeOuter = outer(5);
storeOuter(10)      // returns 15
```

  - **Usage**
    - Closures are useful for creating **Private Variables**, which are variables that cannot be modified externally.
      - Consider the following example:

```
function counter() {
    var count = 0;
    return function() {
        return ++count;    // Pre-Increment
    }
}
```

        - If you were to call counter()() repeatedly, you would return "1" each time, without the count ever going up. This is because there is no "container" to keep track of the incremented count variable

that exists only within the counter() function. However, if you were to save counter() to a new variable in the global scope:
var counter1 = counter();
…then there is now a persistent variable (i.e., counter1) to "house" the increments to "count" within counter1's own counter() function.

- The primary benefit of using closures in this way is that it is no longer possible for the value of count within counter1 to be externally modified by any other function. Thus, you are free to repeat this process with a new variable named "counter2", which, in turn, will have its own internal count variable insulated from any external modification. Neither will have their count value increase except through their own internal operations.

- Consider the following for another more "practical" example:

```
function classroom() {
    var instructors = ['Adam', 'Ben'];
    return {
        getInstructors: function() {
            return instructors;
        },
        addInstructors: function(instructor) {
            instructors.push(instructor);
            return instructors;
        }
    }
}
```

- Essentially, the classroom() function operates as a model template upon which new "course" objects can be created, and in which all courses will have Adam and Ben as instructors. You can create two courses as variables (e.g., "course1" and "course2"), and you can add an instructor to course 1:
course1.addInstructor('Charlie'); // ['Adam', 'Ben', 'Charlie']
…but this addition will not be reflected in course2:
course2.getInstructors(); // ['Adam', 'Ben']

- **jQuery**
  - **About**
    - jQuery is a DOM manipulation library. Although it is a powerful tool, it does add to the "weight" of a project, so be aware of what circumstances exist in which **You Might Not Need jQuery**.
    - **Syntax**:

```
// when a user clicks the button with ID "trigger":
$('#trigger').click(function() {
    //change the body's background to yellow
    $('body').css('background', 'yellow');
    //fade out all "img" elements over 3 seconds
    $('img').fadeOut(3000, function() {
        // remove "img" elements from page when fadeOut is done:
        $(this).remove();
```

```
    });
});
```

- **NOTE**: The **$** sign is defined as a function in jQuery, but otherwise has no unique characteristic in plain JS.

▪ **NOTE**: In order to access a specific DOM element (such as the first on a page) after, you must access it via $('div')[0];. This will allow you to work with the specific DOM element instead of the whole jQuery object containing all elements.

○ **Selecting with jQuery**

▪ Selecting with jQuery is very similar to querySelectorAll() in the sense that we provide a CSS style selector and jQuery will return ALL matching elements:
$('selectorGoesHere');

▪ Selecting can be done on HTML tags, classes, and IDs, and specific selections can be performed in the same manner as in CSS (e.g., all <a> tags inside of <li> tags):
$('li a');

▪ **TIP**: See here for a list of all **jQuery Selectors**.

○ **Manipulating Style**

▪ The **.css()** method is jQuery's interface to styling:
$('selector').css('property', 'value');

- For example, to place a border around an element with the ID "special":
$('#special').css('border', '2px solid red');
- In addition to individual properties, an entire **JavaScript Object** containing multiple properties can be passed into an element by setting a variable:

```
var styles = {
    color: 'red',
    background: 'pink',
    border: '2px solid purple'
};
$('#special').css(styles);
```

○ **Common Methods**

▪ **text()**

- (1) Get the combined text contents of each element in the set of matched elements, including their descendants, or (2) set the text contents of the matched elements. (Basically, the jQuery equivalent of "textContent".)
  - A value passed into text() will update the text content, e.g.:
  $('h1').text('New Text');
- **NOTE**: All text within the elements will be returned in one string.

▪ **html()**

- (1) Get the HTML contents of the first element in the set of matched elements or (2) set the HTML contents of every matched element. (Similar to "innerHTML".)
  - If you have a set of three <li> tags, saying $('li').html() will return the inner HTML of the first <li>, but if you say $('li').html('New Text'), then that will change the inner HTML of all three.
  - **NOTE**: To get the value for each element individually, use a **Looping Construct** such as jQuery's **.each()** or **.map()** methods.
  - **NOTE**: To set the contents of just the **first** , you can use CSS pseudo-selector ":first-of-type", or jQuery's ":first" shorthand selector, or jQuery's .first() method. Technically, the former is more resource efficient (due to being built-in by JS) and is preferred.

        - **NOTE**: To set the contents of the **last** , you can use CSS pseudo-selector ":last-of-type" or jQuery's .last() method, e.g.:
          $('li').last().css('border', '1px solid red');

- **attr()**
    - (1) Get the value of an attribute for the first element in the set of matched elements or (2) set one or more attributes for every matched element.
        - **Syntax**:
            - To "get" an attribute's value (return the value), simply pass through the name of the attribute:
              $('selector').attr('attributeName')
            - However, to "set" an attribute's value, you must pass through both the name of the attribute and the new value:
              $('selector').attr('attributeName', 'value')
                - **NOTE:** To set multiple attributes at once, pass through an object containing the attributes:
                  ```
                  $('#photo').attr({
                      alt: 'text',
                      title: 'text'
                  });
                  ```

- **val()**
    - (1) Get the current value of the first element in the set of matched elements or (2) set the value of every matched element. This allows you to extract the value from an **Input** in the same way that .value() does in JavaScript.
        - **BUT NOTE**: Although most useful for inputs, this method actually works on ALL elements that have a value attribute (e.g., checkboxes, dropdown menus).
    - Like the methods above, .val() uses the **Getter/Setter Dynamic**. If run without an argument, it will return the value. If run with an argument passed through, it will set that argument as the new value.
    - **NOTE**: While it's not common that you would want to change the value of an input (this is usually the user's job), this method is useful for clearing the value in the input box after the user has entered the input, i.e.:
      $(this).val('');

- **addClass()**
  removeClass()
  toggle()
    - These work in the same manner as their vanilla JavaScript counterparts.
    - **NOTE**: "add" and "remove" can be used in tandem like so:
      $('p').removeClass('firstClass').addClass('secondClass');

- **ready()**
    - The .ready() method offers a way to run JS code as soon as the page's DOM becomes safe to manipulate:
      ```
      $(document).ready(function() {
          // code to run after DOM has been loaded
      });
      ```
        - **NOTE**: As of jQuery 3.0, the above syntax has been **Deprecated**, and the **Recommended Syntax** has been abbreviated as follows:
          ```
          $(function() {
          ```

```
    // code to run after DOM has been loaded
});
```

- **IMPORTANT**: jQuery's .ready() method differs from JavaScript's built-in window.onload() method. The latter fires when **ALL** content on the page has loaded (including the DOM, asynchronous JS, frames, and images), whereas the former executes after just the **DOM** has loaded (i.e., the function will execute after all HTML tags and scripts have loaded; it will not wait for the images and frames to fully load). Thus, if it is more important that you execute a function prior to your images being loaded, use the .ready() method. Conversely, if it is more important that your images load first before a less-essential function, then use window.onload().

- **Common Events**
  - **click()**
    - Bind an event handler to the "click" JS event, or trigger that event on an element.
      - **Syntax**:

```
$('button').click(function() {
    alert('You clicked a button.');
});
```

      - **THIS NOTE**: If you want to use "this" in jQuery (to, for example, specify that the .click() event changes the background color of the particular button pressed among a collection of buttons), you have to wrap "this" in jQuery language (i.e., $(this)), because you are applying a jQuery method to a jQuery object, not a vanilla JS object, e.g.:

```
$('button').click(function() {
    $(this).css('background', 'green');
});
```

  - **keypress()**
    - Bind an event handler to the "keypress" JS event, or trigger that event on an element.
      - **NOTE**: .keypress() focuses on which character is entered, rather than focusing on the key's physical state. If you want to trigger an event specifically when the key is pressed down, use **.keydown()** instead. Conversely, use **.keyup()** to trigger the even when the key is released. This distinction is important for inputs where a user may press the Shift key to capitalize his name. If you use .keydown(), the browser will treat the down-pressed "Shift" as its own entry and the lowercase "a" as a subsequent entry.
    - When working with a text **Input** and you want to have the user's data be taken when the user hits the Enter key, use the following syntax that works with the **Event Object**:

```
$('input[type="text"]').keypress(function(event ) {
    if (event.which === 13) {
        //run code when Enter key is pressed
    }
});
```

      - Here, the "event" object (call it whatever you want) contains all information about the .keypress() event. (**NOTE**: The information is

always logged, but never captured as a variable unless you include the argument.) One property in this information is called "**Which**", and that corresponds to the unique key code that jQuery looks at to know "which" key is pressed. Here, the Enter key has a "which" value of 13.

- Refer to **JavaScript Char Codes** for a table for all key codes.

- **on()**
  - Attach an event handler function for one or more events to the selected elements. Behaves in the same manner as ".addEventListener()" by letting you specify the type of event to listen for, e.g.:

```
$('li').on('mouseenter', function() {
    $(this).css('fontWeight', 'bold');
});
$('li').on('mouseout', function() {
    $(this).css('fontWeight', 'normal');
});
```

  - **VERY IMPORTANT NOTE**: In you are using click listeners, why use .on() rather than just .click()? Because .click only adds click listeners to objects that are **on the page at that time**, whereas .on() will add listeners to **all FUTURE elements if Event Delegation is used**.

- **Common Effects**
  - **Fading**
    - **fadeOut()** hides the matched elements by fading them to transparent.
      - **NOTE**: You can modify the rate at which the element fades by including an argument stating the desired time (in ms) to complete the fade effect (the default is 400ms, and you can also use "slow" or "fast" to operate as 600ms or 200ms, respectively):

```
$('li').fadeOut(800);
```

      - **NOTE**: You can modify the "easing" of the fade effect by specifying a second (or first if rate is not specified) argument of either "linear" or "swing", with the latter being the default:

```
$('li').fadeOut(200, 'linear');
```

      - **NOTE**: Finally, you can specify a function as a third (or second if easing not specified, or first if neither rate nor easing are specified) argument to trigger when the animation is complete:

```
$('li').fadeOut('fast', function() {
    // run your code
});
```

        - **IMPORTANT**: When an element is faded out, it is not removed from the DOM. It's "display" property is merely set to "none". If you want to remove the element, then use a callback function in tandem with the .remove() method:

```
$('li').fadeOut(function() {
    $(this).remove();
});
```

    - **fadeIn()** displays the matched elements by fading them to opaque. It operates exactly the same way as .fadeOut() (including the ability to use all three arguments), except in reverse (by removing "display: none" from the element).

- **fadeToggle()** display or hide the matched elements by animating their opacity. It operates the same as above (like using .toggle()).

- **Sliding**
  - Operates the same as fading, except by sliding the element up (and hiding it) or down (and displaying it) by animating the element's height (rather than opacity). The three effects are .slideUp(), .slideDown(), and .slideToggle().

- **Other Methods**
  - **stopPropagation()**
    - If a parent element has an event listener (e.g., a click listener), and its child element has its own event listener, modern browsers will first trigger the child element's event listener, then the parent's event listener, and will continue to "bubble" up the chain until it reaches the <html> element itself. (This is a phenomenon known as **Event Bubbling**.) This can be problematic when you only want a child element's event listener to fire without triggering any of its parent's event listeners. jQuery allows you to prevent this behavior by its .stopPropagation() method:

```
$('p').click(function(event) {
    event.stopPropagation();      // run code for child element only
});
```

      - **NOTE**: Similar to the .which() method used in the .keypress() method above, .stopPropagation() requires the use an "event" object that can be called whatever you want.
      - **ALSO NOTE**: Although .stopPropagation() is a jQuery method in this particular context, it should be noted that .stopPropagation() is also a **built-in method in plain JS** (as are various other jQuery methods).
  - **parent()**
    - Get the immediate parent of each element in the current set of matched elements. The method optionally accepts a selector expression. If the selector is supplied, the elements will be filtered accordingly. For example, if you want to use the .remove() method to remove an <li> when a within the <li> is clicked (as opposed to just removing the):

```
$('span').on('click', function() {
    $(this).parent().remove();
});
```

  - **append()**
    - Insert content to the end of each element in the set of matched elements. For example, to add an <li> to the end of all <ul> tags on the page:

```
$('ul').append(<li>Text</li>);
```

BACKEND

- **Basics**
  - **HTTP**
    - Hypertext Transfer Protocol (HTTP) is a protocol which allows the fetching of resources, such as HTML documents. HTTP follows a classical client-server model, with a client opening a connection to make a request, then waiting until it receives a response.
  - **Postman**

        - Postman can be used to make HTTP **Requests** (e.g., a "GET" request for receiving information, or a "POST" request for submitting information to the database) and receive HTTP responses from outside of a traditional browser.
        - There are three important parts of every **Response** received by Postman:
            - (1) **Body**: Essentially the payload containing the HTML, CSS, and JS that you would see in "View Page Source").
            - (2) **Headers**: The metadata about the response (e.g., content-type, date, status).
            - (3) **Status Code**: Just a number indicating the status of the request (e.g., "200" means the request is OK, "404" means the content requested does not exist).
    - **Cloud9**
        - C9 is a complete developer environment that contains everything you need in the browser (e.g., a text editor, terminal, node.js, etc.).
        - **NOTE**: This course's publicly visible Cloud9 workspace is available here.

- **The Command Line**
    - **Resources**
        - **Getting to Know the Command Line**
            - **Finding the Command Line in Windows**
                - There is no native command line in Windows. Install **Babun**.
            - **Command Syntax**
                - All commands have three parts: (1) the utility, (2) the flags, and (3) the arguments. The utility always comes first, and the other two are only needed depending on the command that is used. Example:
                ls -l ~/Desktop
                (1) "ls" is a utility that is used to list the contents of directories; (2) "-l" is a flag used to indicate to the utility that we want more information than usual, and so it should show the directory in its "long" format; flags alter how the utility operates; flags always start with either one or two dashes, and they usually come between the utility and arguments; (3) "~/Desktop" is an argument to the utility that tells it which directory's contents to list; arguments are used when the utility needs to know exactly what you want for a certain action and there's no clear default setting; arguments usually come at the end of the command.
            - **Basic Utilities**
                - **Manual**: man $UTIL
                    - Get information for how to use any utility. Replace $UTIL with any utility, like ls, cd, or even man. Press the up and down arrows to scroll through the documentation. Press "Q" to quit and go back to the command line.
                - **List**: ls $DIR
                    - Lists the contents of the directory $DIR. If no directory is specified, lists the contents of the current working directory. Use the -l flag to get more information.
                - **Change Directory**: cd $DIR
                    - Changes the current working directory to the directory $DIR. In effect, moves you around the computer.
                - **Print Working Directory**: pwd

            - If you ever get lost in the computer, run this command to get a trail of breadcrumbs all the way down from the top level of the computer to see where you are.
        - less $FILE
            - Displays the contents of a file. Press the up and down arrows to scroll though the file. Press "Q" to quit and go back.
        - **Copy**: cp $FILE $LOCATION
            - Copies the $FILE to the $LOCATION.
        - **Move**: mv $FILE $LOCATION
            - Moves the $FILE to the $LOCATION.
        - **Remove**: rm $FILE
            - Deletes a file **permanently**: there is no way to get it back. Be careful when using this command!
            - **NOTE**: Use the -rf (recursive force) flag to remove an entire directory (including all subdirectories and their contents).
        - **Super User Do**: sudo $CMD
            - When you use this utility, you use an entire command as a single argument: for example, sudo ls -l ~/Desktop. Sudo asks for your user account password. As a security measure, the screen does not display anything as you type, not even asterisks (*). If the password is typed in correctly, sudo executes the $CMD with elevated permissions. Be careful when using this command!
    - **Survival Guide for Unix Newbies**
    - **Learn Enough Command Line to Be Dangerous**
- **Other Command Line Utilities**
    - **Create File**: touch $FILE.ext
        - Creates a new file.
    - **Make Directory**: mkdir $DIR
        - Creates a new folder.

- **Node.js**
    - **About**
        - Previously, all JS had to run in the browser (which means it was all front-end code and not server-side). However, Node.js now allows JS to be run server-side.
    - **Usage**
        - The **Node Console** is purely server-side and therefore has no HTML/CSS to interact with. Rather, all JS is executed in the command line via the terminal.
            - **NOTE**: Not all JS features are available in the Node console (e.g., "alert", "document", or any DOM manipulation), because such functions come with the browser.
        - To **Enter** the Node console from the terminal in Cloud9, type "node". To **Exit**, hit "CTRL + C".
        - For the most part, you will not be running code out of the Node console; rather, a more common task is to run JS files via Node by typing "node fileName.js".
    - **NPM (Node Package Manager)**
        - **About**
            - To use libraries (e.g., jQuery or Bootstrap's JS library) on the front end, you would use a <script> tag to include the library. However, there is no HTML on

the server-side to perform these functions. NPM allows you to include those libraries (a.k.a. "packages") in a repository on the NPM website, and which can be installed via its command line tool.

- **Installing NPM Packages**
  - Use "npm install packageName" to **Install** a package, e.g.:
    npm install cat-me
  - Upon installation, a new directory entitled "node_modules" should appear that contains the specified package (and any subsequent packages that are installed will appear inside the same directory).
  - After the installation, you must then **Import** the package into your application via the "require" command in order to use the package:
    var catMe = require('cat-me');
    - **NOTE**: "catMe" can be any name you want it to be.
  - Once the package has been imported into a variable, refer to the NPM's packages documentation to learn how to use the package.
- **Package.json**
  - **About**: All npm packages contain a file (usually in the project root) called package.json (JSON: JavaScript Object Notation), which holds various metadata relevant to the project (re: identification, dependencies, description, version, license, configuration). Node itself is only aware of two fields in the package.json file: name and version.
    - **NOTE**: Dependencies specified in package.json will be installed to a folder called **node_modules** upon package installation.
  - Use "npm init" to **Create** a new package.json file.
  - When you use the "--save" flag in tandem with "npm install", it will take the package name and version and automatically save it into your **Dependencies** in your package.json file, if you have one.
    - **IMPORTANT**: When naming your project, be sure that the project is not given the same name as that of any NPM library that your project will use as a dependency. If you do, you will receive an error upon attempting to install the library as a dependency. To correct this problem, you will have to access the package.json file and modify the "name" property.

- **Express.js**
  - **About**
    - Express is a web development **Framework**, which is like a library/package in the sense that a framework is a set of external code included in your application. However, libraries differ in the sense that you are in full control (e.g., you can pick and choose which methods you wish to use), whereas you give up much more control when using a framework (i.e., the framework gives you the "scaffolding", and you get to fill it in).
    - **NOTE**: Other web development frameworks include Flask, Django, Rails, etc. Express is known as a "light-weight" framework in which there is more "white space" for you to fill in yourself, as opposed to a "heavy-weight" framework like Rails, in which most of the work has already been done for you.
  - **Installation**
    - Use "npm install express" in the Cloud9 terminal, and then "require" it in your application, e.g.:
      var express = require('express');

- **NOTE**: Unlike "cat-me", Express cannot simply be run by simply saying, e.g., express(). This is because, unlike cat-me, Express is not just one simple function but rather a collection of many different methods. Therefore, the best approach is to save its execution as a variable (conventionally named "app") to which methods can later be appended (app.whatever) as needed, e.g.:
  var app = express();
- When working outside of the Cloud9 terminal, use the "--save" flag to include Express in your package.json dependencies:
  "npm install express --save"

- **Routing**
  - **Basics**
    - Express relies on routing to determine how an application responds to a client request to a particular endpoint, which is a **Uniform Resource Identifier** (or path) and a specific HTTP request method (e.g., GET, POST, etc.). Each route can have one or more handler functions, which are executed when the route is matched.
    - Routes follow the syntax below:
      app.$METHOD($PATH, $HANDLER)
      - (1) app is an instance of Express;
      - (2) METHOD is an HTTP request method, in lowercase;
      - (3) PATH is a path on the server; and
      - (4) HANDLER is the function executed when the route is matched.
    - Specifically, a route could provide a message when a user accesses the home page:

```
app.get('/', function(req, res) {
    res.send('hello');
});
```

      - **NOTE**: "req" (**Request**) and "res" (**Response**) are actually objects (and can be named whatever you want, although req and res are conventions). "Request" is an object containing all the information about the request that was made and which triggered the route. "Response" is an object containing all of the data that the server is going to respond with.
  - **Parameters**
    - The **Splat (*)** parameter acts as a wildcard, and it will trigger whenever the user attempts to access an undefined or unexpected route, e.g.:

```
app.get('*', function(req, res) {
    res.send('Error.  Page not found.');
});
```

      - **NOTE**: The splat parameter should appear at the **END** of your routes, as it will supersede all other subsequent matching routes (the first route that matches a given request is the only route that will be run). The order of routes always matters (like the order of execution in a series of if/else if conditions).
    - **Route Parameters** (a.k.a., route variables or path variables) are used to define and identify patterns in a route by using a colon in front of anything that you want to be a variable. For example, in the case of Reddit, the following pattern is used:

```
app.get('/r/:subredditName', function(req, res) {
```

```
    var subreddit = req.params.subredditName;
    res.send('Welcome to the ' + subreddit + ' subreddit.');
});
```

…or…

```
app.get('/r/:subredditName/comments/:postId/:postTitle/', function(req, res) {
    res.send('<h1>Welcome to the comments page.</h1>');
});
```

- **NOTE**: In order to know what is the value of ":subredditName", it is possible to access that data by entering the "req" object, which contains all of the information about the incoming request. The simplest way to access that data is to use console.log(req), and then search for the value listed in "**params**" – or simply use console.log(req.params) to see all route parameters.
- **ALSO NOTE**: HTML tags can be included in the sent response.

- **Listening**
  - In order for Express to know when to serve a response to a request, it must be told to **Listen** for requests. This is done by using "app.listen()" together with the port number that should be listened to, e.g.:

    ```
    app.listen(3000);
    ```

    - **NOTE**: On Cloud9, you have to use the website's own assigned port and IP address, which can vary. Therefore, use the following syntax to obtain the variable port/IP:

      ```
      app.listen(process.env.PORT, process.env.IP);
      ```

- **Templates and EJS**
  - **Basics**
    - As noted above, HTML elements can be used in sending a response; however, transmitting an entire HTML page in this manner would be cumbersome. Instead, you can use a method called **Render** (as appended to the "res" object), e.g.:

      ```
      app.get('/', function(req, res) {
          res.render('home.html');
      });
      ```

    - However, when working with Express, you typically will not write plain HTML files (standard static files); rather, you will likely be using dynamic HTML files that are called **Templates**, which exist as **EJS** (Embedded JavaScript) files, and **<u>WHICH MUST BE LOCATED IN A DIRECTORY CALLED "VIEWS"</u>** to be rendered by Express, e.g.:

      ```
      app.get('/', function(req, res) {
          res.render('home.ejs');
      });
      ```

      - **NOTE**: Using EJS files requires installing EJS: npm install ejs --save
      - **ALSO NOTE**: Express's special handling of the "views" directory also includes the option to specify ahead of time what type of files will be served from "views". Thus, rather than needing to specify res.render('love.ejs'), you can simply say res.render('love'). To do so, simply include the following line if your instance of Express:

        ```
        app.set('view engine', 'ejs');
        ```

- To understand the **Syntax** of EJS, refer to the **EJS Documentation**. But it is important to note that variables set in the GET callback function do not automatically pass through to the variable names used in the EJS file. Rather, such information must be passed through the res.render() method as an object (which can contain multiple pieces of data that you want available in your template), e.g.:

```
app.get('love/:thing', function(req, res) {
    var thing = req.params.thing;
    res.render('love.ejs', {thingVar: thing});
});
```

…then in the love.ejs file, you can use the following **EJS Tags**:

```
I love <%= thingVar %>!
```

…and the end result for a route such as "love/cats" will be, "I love cats!"

  - **NOTE**: If you want your EJS to execute **HTML** tags rather than just display them if contained in a given variable, then use <%- instead. However, caution should be exercised, as <script> tags can be executed in HTML. This problem can be remedied by **Sanitizing** your inputs. One popular package is **Express Sanitizer**.
    - To install Express Sanitizer, run:

      ```
      npm install express-sanitizer --save
      ```

      …require it:

      ```
      var expressSanitizer = require('express-sanitizer');
      ```

      …and place the following **AFTER YOUR BODY PARSER** (explained below in the "POST Requests" section):

      ```
      app.use(expressSanitizer());
      ```

      …then place the following in your create and update routes prior to running the actual create and update methods:

      ```
      req.body.propertyToSanitize =
      req.sanitize(req.body.propertyToSanitize);
      ```

- **Conditionals**
  - To include conditional if/else logic in an EJS file, wrap each individual line of JS in the following EJS tags, e.g.:

```
<% if (thingVar === 'dogs') { %>
    <p>Good choice!  Dogs are the best!</p>
<% } else { %>
    <p>Bad choice!  You should love dogs!</p>
<% } %>
```

    - **NOTE**: EJS tags with the **Equals Sign** are used to take the returned value and render it to the page as HTML. On the other hand, if you are simply executing logic (conditionals/loops) that should not be displayed in the HTML, then use EJS tags without the equals sign.
- **Loops**
  - Loops operate in the same manner as described above. Create your set of data in your instance of Express to be passed to the EJS file, e.g.:

```
app.get('/posts', function(req, res) {
    var posts = [
        {title: 'My dog has fleas.', author: 'Alex'},
        {title: 'All cows eat grass.', author: 'Bob'},
        {title: 'Every good boy deserves fudge.', author: 'Charlie'},
```

```
    ];
    res.render('posts.ejs', {postsVar: posts});
});
```

…and then in the posts.ejs file:

```
<h1>Posts</h1>
<% postsVar.forEach(function(post) { %>
  <p><strong>Title:</strong> <%= post.title %>
  <strong>Author:</strong> <%= post.author %> </p>
<% }); %>
```

- **Serving Custom Assets (CSS/JS)**
  - To serve a response that includes more than just plain HTML, Express requires that custom assets (such as CSS and JS files) be configured in a particular manner. Convention calls for giving your workspace's asset folder the name "**Public**". If you are working with a CSS file, you would then create a new filed named, for example, "style.css" in "public". To link to this resource, you need not specify both the directory and file name. Rather, you should just use the file name in the link, and then include the following line in your instance of Express to tell it where to find custom assets:
    app.use(express.static('public'));
    - **NOTE**: The default behavior of Express is to not serve anything aside from the "views" directory, unless it is specifically instructed to do so. This is why it is necessary to tell Express explicitly to serve the "public" directory, although not required for the "views" directory.
    - **ALSO NOTE**: Out of the abundance of caution, you can state the following as an alternative:
      app.use(express.static(**__dirname +** '/public'));
      By incorporating "**dirname**", you are specifically instructing Node to look for "public" in the same directory name that the app.js file is saved within. But note that, when doing so, you must add the **Backslash** before "public".
  - However, since each page will require standard HTML boilerplate (in which links to custom assets will be included), it is necessary to create a template for your HTML "header" (everything above and including the opening <body> tag) and "footer" (everything to appear at the end of all pages on the website until the closing </html> tag). Such headers and footers are called **Partials**, and they are typically named "header.ejs" and "footer.ejs" respectively, and they are stored in the "**views/partials**" directory. Once created, you can simply include the following at the top and bottom of each of your EJS files:
    <% include partials/header %>
    …
    <% include partials/footer %>
    - **IMPORTANT**: When linking to an asset (such as "style.css" in "public"), even though Express knows to look in the "public" directory to find the asset, it will only be recognized to exist in your workspace's root directory (e.g., www.website.com/), and not whatever subdirectory the user may currently be in (e.g., www.website.com/info/). Therefore, if you want all pages on your site to share a common CSS file, it is **CRITICAL** to ensure that all stylesheet links in your HTML header state that the resource originates in the **ROOT DIRECTORY**, i.e.:
      <link rel="stylesheet" type="text/css" href="**/**style.css">
      …**NOT**…
      <link rel="stylesheet" type="text/css" href="style.css">

        - **NOTE**: If you were to put your CSS file into a subdirectory called "stylesheets" in your "public" directory, then you would say, href="/stylesheets/style.css". The main thing is to include the root "/" character.
    - **NOTE**: Depending on the location of your workspace, it may be necessary to say **./partials/header** rather than just partials/header. The **Dot-Slash** is necessary should you want Node to use whatever your **CURRENT DIRECTORY** is as its point of reference, as opposed to the root directory (i.e., tell Node to find the header file in the partials directory of whatever directory you're currently in, not the header file in the partials directory of the root directory).
        - Should it be necessary to move **BACK ONE DIRECTORY**, then instead use **Dot-Dot-Slash**: **../partials/header**. You can use the dot-dot-slash as many times as necessary to move back as many directories as you need to, e.g.: ../../partials/header.
- **POST Requests**
    - To create a new POST request, you must use the **app.post()** method in your Express instance, and route the request to a particular location, e.g.:

```
app.post('/friends', function(req, res) {
    // insert code here
});
```

    ….and then use that same location in, for example, the HTML form in the EJS file that will be used to transmit the POST request:

```
<form action="/friends" method="POST">
    <input type="text" name="newfriend" placeholder="New Friend">
    <button>Add!</button>
</form>
```

        - **NOTE**: The input must be given a name, as that will be the "key" by which we can look it up inside of the route. When the request is sent, the value of the single "newfriend" property will be sent in the body of the request. To see this illustrated, you can use console.log(req.body), in which "req.body" is an object that contains all of the data from the request body. **HOWEVER**, Express lacks the ability to create "req.body" by default; therefore, it is necessary to use "**npm install body-parser --save**", as this will take the request body and turn it into a JS object that can be used. **Body Parser** can be used when there is a form from which you would like to extract data on the server side.
            - **ALSO**: The following lines must be used for the body parser to work:

```
var bodyParser = require('body-parser');
app.use(bodyParser.urlencoded({ extended: true }));
```

                - **NOTE**: Refer to the documentation for further explanation.
    - Upon the form's submission, the following can be run to (1) create a new variable containing the desired value from req.body, (2) push the value to an array of the list of friends, and (3) **Redirect** the user back to the friends list / submission page:

```
app.post('/friends', function(req, res) {
    var newFriend = req.body.newfriend;
    friendsList.push(newFriend);
    res.redirect('friends.ejs');
});
```

        - **NOTE**: See the section below on RESTful routing for additional information on naming conventions.

- **ALSO**: If you want to redirect a user **Back** to the **HTTP Referer**, use:
  res.redirect('back');

- **IMPORTANT**: Because the request body will be returned as an object, that means you can take advantage of **BRACKET NOTATION WHEN NAMING YOUR INPUTS**. For example, if you have a form that will submit a new blog post with three properties (title, image, and body), you can set the name of each input as follows: name="blog[title]", name="blog[image]", and name="blog[body]". The result will be the compilation of a single "blog" object with three properties for title, image, and body (with each having its value defined by the user). When used together with Body Parser, the entire object can be accessed by simply stating: req.body.blog

- **PUT Requests**
  - As a preliminary matter, making a PUT request will require installing **Method-Override**, which allows you to use HTTP verbs such as PUT or DELETE in places where the browser does not support such requests by default. Most notably, **HTML Forms** were only designed to recognize GET and POST requests. Thus, editing database content via a form element will require the installation and use of method-override as follows:
    ```
    npm install method-override --save
    ```
    …together with:
    ```
    var methodOverride = require('method-override');
    app.use(methodOverride('_method'));
    ```
    - **NOTE**: The underscore in '_method' is known as a "**Getter**," which is used to look up the overridden request. The '_method' getter is used to override the method via a query string, which is detailed further in the next bullet point. If you fail to implement these steps, your PUT request will default to a GET request.
  - A PUT request is used to replace the data contained in a given resource. To create a new PUT request, you must use the **app.put()** method and route the request, e.g.:
    ```
    app.put('/friends/:id', function(req, res) {
        // insert code here
    });
    ```
    …and then use that same location in the HTML form in the EJS file that will be used to transmit the PUT request to a specific database object based upon its ID:
    ```
    <form action="/friends/<%= friend.id %>?_method=PUT" method="POST">
        <input type="text" name="friend[name]" value="<%= friend.name %>">
        <button>Update!</button>
    </form>
    ```
    - **NOTE**: When using method-override, you have to set the "normal" method to POST. Once you set the value of the "_method" query string to "PUT", the subsequent method="POST" is overridden to be a true PUT request.

- **DELETE Requests**
  - DELETE requests are essentially handled in the same manner as PUT requests, e.g.:
    ```
    app.delete('/friends/:id', function(req, res) {
        // insert code here
    });
    ```
    …although they differ insofar as the only data that needs to be passed through is the parameter associated with the data object to be deleted (which, in this case, would be the ID tag):
    ```
    <form action="/friends/<%= friend.id %>?_method=DELETE" method="POST">
        <button>Delete!</button>
    ```

```
</form>
```

- **Refactoring Routes**
  - Rather than having all of your routes in your app.js file, you can break them up by functionality across multiple JS files, which can then be required by the app.js file. Convention calls for placing each file in a directory named "routes". Each route file can be given a specific name, although an "all-purpose" routes file is typically named "index.js".
  - After your routes have been divided into separate JS files, you can use the **Express Router** by placing the following code at the top of each file:
    ```
    var express = require('express');
    var router = express.Router();
    ```
    … then (1) substitute "app" for "router" in each route, and (2) export the output at the end of the file by using the following code:
    ```
    module.exports = router();
    ```
  - After your JS files have been configured and all required dependencies have been included, the exported data can then be assigned to a variable in app.js, which can be used by the application, e.g.:
    ```
    var friendRoutes = require('./routes/friend');
    …
    app.use(friendRoutes);
    ```
    - **NOTE**: If you want to further "dry" up your code, you can specify in the "app.use()" line what particular language should appear in each route, and then you can append any additional required language in the route's JS file. For example, if your "friends" routes all start with "/friends" (e.g., "/friends/new", "/friends/:id", "/friends/:id/edit"), you can simplify your code by stating the following in your app.js file:
      ```
      app.use('/friends', friendRoutes);
      ```
      … and then remove '/friends' from the beginning of all routes in your friend.js routes file, as they will no longer be necessary (all that is needed is to include "/new", "/:id", "/:id/edit", and so forth).
    - **IMPORTANT**: To pass **Parameters** (such as ":id" above) from the app.js file to the route's JS file, it is necessary to include the following in your router variable:
      ```
      var router = express.Router({ mergeParams: true });
      ```

- **RESTful ("Representational State Transfer") Routing**
  - **Basics**
    - RESTful routing can be described as an architectural style that provides a map between HTTP verbs (e.g., GET, POST, etc.) and CRUD (create, read, update, delete) actions. There are seven RESTful route conventions:

Name	URL	Verb	Description	Mongoose Method
INDEX	/dogs	GET	List all dogs.	Dog.find()
NEW	/dogs/new	GET	Show new dog form.	N/A
CREATE	/dogs	POST	Create a new dog (then redirect).	Dog.create()
SHOW	/dogs/:id	GET	Show information about one dog.	Dog.findById()
EDIT	/dogs/:id/edit	GET	Show edit ("update") form for one dog.	Dog.findById()
UPDATE	/dogs/:id	PUT	Update one dog (then redirect).	Dog.findByIdAndUpdate()
DESTROY	/dogs/:id	DELETE	Delete one dog (then redirect).	Dog.findByIdAndRemove()

        - RESTful conventions call for giving the CREATE route the **SAME NAME** as that given to the INDEX route that will be appended by the POST request. For example, if you have a route called "/dogs" that will display an array of dogs upon rendering "index.ejs" when a GET request is made, then you would similarly name the POST route "/dogs", even if the POST request is made from a different page (as it is ultimately appending the same array that will be displayed on "/dogs" pursuant to a GET request). The fact that both the GET and POST route have the same name will not create conflicts, as one is a GET and the other a POST. The same applies to SHOW, UPDATE, and DESTROY.
            - **NOTE**: Each of the non-GET requests (i.e., POST, PUT, and DELETE) should always be followed up with a GET request redirect.
            - **ALSO NOTE**: It is common for most websites to have their home page ('/') redirect to the route that renders the index, e.g.:

```
app.get('/', function(req, res) {
    res.redirect('/dogs');
});
app.get('/dogs', function(req, res) {
    res.render('index.ejs');
});
```

    - **Nested Routes**
        - When one collection of data will be associated with a document from another data collection, one should use nested routes. For example, in reference to the chart above, you may want to allow users to add comments on each dog's "show" route. This would require making a "new" route for the comment form, and then a "create" route to post comments. If the dog and comment routes were created independently of each other, you could setup the following routes:

NEW	/dogs/new	GET
CREATE	/dogs	POST
NEW	/comments/new	GET
CREATE	/comments	POST

        - However, comments do not exist independently from dogs. When a comment is created, it is associated with a particular dog by its ID; therefore, it is essential to have the dog's ID in that particular route. Thus, the "comments" route should be nested under the "dogs" route by ID:

NEW	/dogs/:id/comments/new	GET
CREATE	/dogs/:id/comments	POST

- **Application Programming Interfaces (APIs)**
    - **Basics**
        - APIs are interfaces for programs to interact with each other. It broadly refers to code that is made for other types of code to communicate with. Web APIs are more specialized for web applications (e.g., to have a program extract certain sets of information from Reddit), and they generally communicate via HTTP (via a URL).
    - **XML and JSON**
        - APIs don't respond with HTML (which contains information about the structure of a page). Rather, APIs respond with data (not structure) and therefore use simple data formats like XML and JSON.
        - Extended Markup Language (**XML**) is similar to HTML in syntax but does not describe page presentation; rather it encodes key value pairs, e.g.:

```
<person>
```

```
    <age>21</age>
    <name>Travis</name>
    <city>Los Angeles</city>
</person>
```

- JavaScript Object Notation (**JSON**) looks exactly like JavaScript objects, except everything is a **STRING IN DOUBLE QUOTES** (no single quotes allowed), e.g.:

```
{
    "person": {
        "age": "21",
        "name": "Travis",
        "city": "Los Angeles"
    }
}
```

  - **IMPORTANT**: The fact that all data provided in a JSON **Body** is a string means that you cannot readily access the data in the same fashion as a traditional JS object. In order to turn the JSON body into an object, you must **Parse** it and save the data to a variable. JavaScript comes with a built-in parser for JSON, e.g.:
    `var data = JSON.parse(body);`
- **NOTE**: By default, JSON is rendered as one large block of text in the Chrome browser. You can use the **JSONView** extension to remedy this issue.

- **Making API Requests with Node (and Express)**
  - The most common way to make an API request with Node is to install a package called **Request**. A sample request format is provided as follows:

```
var request = require('request');
request('http://www.google.com', function (error, response, body) {
    // Print the Error if one occurred:
    console.log('error:', error);
    // Print the response Status Code if a response was received:
    console.log('statusCode:', response && response.statusCode);
    // Print the HTML Body for the Google homepage:
    console.log('body:', body);
});
```

  - The above request() format can be placed into a GET request for the purpose of running the request once a user requests a certain page. Thus, for example, if a search form on the home directory takes the user to a "/results" subdirectory to view the results, then you would state:

```
app.get('/results', function(req, res) {
    // insert request() here
});
```

    ...then any form that directs to 'results' (via its "action" property) will submit all name-value pairs as properties that can be accessed within a single object called **req.query** (in Express). These are the same name-value pairs that appear after the "?" in the URL. Thus, if a text input with the name of "t" (for title) and a numerical input with the name of "y" (for year) are used in a form, the submission of "Frozen" for the title and "2013" for the year would produce "/results?t=frozen&y=2013" in the URL, and this data can be accessed and assigned to a variable and used accordingly:

```
app.get('/results', function(req, res) {
    var title = req.query.t;
    var year = req.query.y;
```

```
    request('http://www.omdbapi.com/?apikey=thewdb&t=' + title + '&y=' + year,
        function (error, response, body) {
            var data = JSON.parse(body);
            res.render('results', { data: data });
        }
    }
});
```

- **NOTE**: You can extract API data straight out of the command line by using: curl <insert API link here>

- **Databases**
  - **Basics**
    - There are two broad categories of databases: **SQL** (sequel) and **NoSQL** (non-sequel).
      - **SQL Basics**
        - SQL databases are considered to be **Relational** and organized in tabular form (a flat data structure), e.g.:

USERS TABLE			
id	name	age	city
1	Tim	57	NYC
2	Ira	24	Missoula
3	Sue	40	Boulder

COMMENTS TABLE	
id	text
1	"First comment."
2	"Second comment."
3	"Third comment."

        - If you want there to be a relationship between tables, you create a **Join Table**, e.g.:

USERS/COMMENTS JOIN TABLE	
userId	commentId
1	3
2	1
2	2

      - **NoSQL Basics**
        - By contrast, NoSQL databases are **Non-Relational** and organized in BSON (Binary JSON) form (a nested data structure), e.g.:

```
{
    name: "Tim",
    age: 24,
    city: "Missoula",
    comments: [
        { text: "First comment." },
        { text: "Second comment." }
    ]
}
```

    - **NOTE:** NoSQL is newer than SQL, but not necessarily better in all cases. Other popular databases that are SQL include MySQL and PostgreSQL.

- **MongoDB**
  - **About**
    - MongoDB is the most popular NoSQL database. MongoDB is currently the most popular database for Node as part of the "**MEAN**" stack, which stands for (1) MongoDB, (2) Express, (3) Angular, and (4) Node.
  - ~~**Installation and Use (FOR CLOUD9)**~~ **[OUTDATED]**
    - ~~Preliminarily, installation requires running the following command:~~
      ~~sudo apt-get install -y mongodb-org~~
    - ~~Subsequently, run the following commands (refer to the above link for more information about the parameters used):~~
      ~~mkdir data~~
      ~~echo 'mongod --bind_ip=$IP --dbpath=data --nojournal --rest "$@"' > mongod~~
      ~~chmod a+x mongod~~
  - **Installation and Use (FOR CLOUD9) [v.3.6.2]**
    - Preliminarily, installation requires running the following commands in order:
      1) sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 2930ADAE8CAF5059EE73BB4B58712A2291FA4AD5
      2) echo "deb [arch=amd64] https://repo.mongodb.org/apt/ubuntu trusty/mongodb-org/3.6 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-3.6.list
      3) sudo apt-get update
      4) sudo apt-get install -y mongodb-org
    - Subsequently, run the following commands in order:
      1) mkdir data
         - **NOTE**: This directory is where Mongo stores its data.
      2) echo "mongod --dbpath=data --nojournal" > mongod
      3) chmod a+x mongod
    - Start Mongo by running the "mongod" script (the Mongo Daemon) on your project root:
      ./mongod
      - **IMPORTANT**: Once mongod is running, **THE TERMINAL RUNNING MONGOD MUST REMAIN OPEN FOR THE DATABASE TO REMAIN UP**. Thus, you must create a new terminal for performing all other tasks while the mongod terminal remains untouched in its own terminal.
      - **HOWEVER**: Once you are done working on your project, **SHUT DOWN MONGOD**, or else it can crash if/when Cloud9 times out.
  - **Basic Mongo Commands**
    - **Mongo Shell**: mongo
      - Opens the shell, which is used for testing and debugging.
    - **Help**: help
      - Displays a list of Mongo's basic features.
    - **Show** databases: show dbs
      - Displays a list of all databases (the two default databases are "admin" and "local").
    - **Use** or make a database: use
      - If a database does not exist, you can use the use command to create a database; otherwise, the command will use the database, e.g.:
        use demo

        - **NOTE**: All databases are empty by default and will not appear on the show dbs list until there is data within them.
- **Add** data (by creating a new "**Collection**" or adding to an existing one): insert
    - All data are stored in groups called collections. If, for example, you want to create a collection of data about dogs, you could create a collection called "dogs" and add data to it concurrently by stating:
      db.dogs.insert({name: 'Rusty', breed: 'Mutt'})
        - **NOTE**: Once the collection is created, additional data can be added to the collection by using the same syntax above:
          db.dogs.insert({name: 'Lucy', breed: 'Mutt'})
          db.dogs.insert({name: 'Lulu', breed: 'Poodle'})
        - **ALSO NOTE**: When an object is added to a collection, Mongo automatically assigns each object its own unique **ObjectId** in hexadecimal.
- **Find** (or "reading") data in a collection: find
    - Displays either (1) a list of all objects in a given collection, or (2) a specific object within a given collection. As to the first operation, you can state the following:
      db.dogs.find()
      ...which will return ALL objects in the collection. However, if you want to find a specific object within the collection, you need only pass through one property as an argument:
      db.dogs.find({name: 'Rusty'})
- **Modify** or add new properties to an existing object in a collection: update
    - Modifies an object's property by passing through two arguments in which you (1) first specify a unique property that identifies a particular object to be modified (e.g., a "name" or "_id"), and (2) then modify the value of a specific property within that object. For example, you can change Lulu's breed from "Poodle" to "Labradoodle", and also add a new Boolean property called "isCute" by saying:
      db.dogs.update({name: 'Lulu'}, {$set: {breed: 'Labradoodle', isCute: true} })
        - **IMPORTANT**: If you do not use "$set" and simply say:
          db.dogs.update({name: 'Lulu'}, {breed: 'Labradoodle', isCute: true})
          ...then you will **OVERWRITE ALL PROPERTIES** in the object with whatever properties are contained in the second argument (which, in this case, would delete Lulu's name property).
- **Delete DATA** from a collection: remove
    - Removes all matching objects according to the following syntax:
      db.dogs.remove({breed: 'Mutt'})
      …which, in this case, would result in the deletion of both Rusty and Lucy, because they are both "Mutt" breeds.
        - **NOTE**: The default behavior is to delete all objects that match; however, the number can be limited by using the **Limit** method:
          db.dogs.remove({breed: 'Mutt'}).limit(1)
- **Delete AN ENTIRE COLLECTION**: drop

        - Removes an entire collection:
          db.dogs.drop()
    - **Security**
        - For information regarding **Security Best Practices**, refer to the following links:
            - Security Best Practices for Express in Production
            - MongoDB Security Best Practices
- **Mongoose**
    - **About**
        - Mongoose is an NPM package that helps you interact with MongoDB inside of your JS files. Mongoose is known as an **Object-Document Mapper** (ODM), which basically means it is a way to write JS to interact with a database (i.e., a JS "layer" on top of MongoDB). All of its operations could be written without Mongoose; however, Mongoose makes it easier to interact with MongoDB (in a similar way that jQuery makes it easier to interact with the DOM).
    - **Installation**
        - Install: npm install mongoose
        - Require: var mongoose = require('mongoose');
    - **Configuration**
        - **Connect**
            - To connect Mongoose to your MongoDB, state the following in your application:
              mongoose.connect('mongodb://localhost/cat_app');
                - **NOTE**: If the specified database name (noted as "cat_app" above) does not exist, then it will automatically be created.
        - **Schema**
            - To add data to the database, it is first necessary to define a schema (i.e., Mongoose's "blueprint" of how a data object in a specific collection will be organized). For example, if you are creating a database of cats, the following schema could be used:

```
var catSchema = new mongoose.Schema({
    name: String,
    age: Number,
    temperament: String
});
```

                - **NOTE**: You are not forbidden from adding new properties or omitting the listed properties when adding objects to the collection, but this practice is not advised.
                - See here for a **Complete List of Schema Types**.
                    - The "**Date**" schema type can be configured to automatically set its value to be whatever the current time on the server is upon execution, e.g.:

```
var blogSchema = new mongoose.Schema({
    title: String,
    body: String,
    date: { type: Date, default: Date.now }
});
```

                        - **NOTE**: This option is not just limited to date properties, but can be applied in any situation in

which you want a **Default** value to be applied where none is submitted by the user, e.g.:
image: { type: String, default: 'defaultimage.jpg' }

- **Model**
  - Once a schema is defined, it can be complied into a model (returned as an object with predefined methods) that can be used to find, create, update, and delete objects of the given type (once saved to a variable), e.g.:
    var Cat = mongoose.model('Cat', catSchema);
    - **NOTE**: The first argument must be the **SINGULAR** name of the collection that will be created for your model (Mongoose automatically looks for the plural version of your model name). The second argument is the schema you want to use in creating the model.
    - **IMPORTANT**: The reason why "Cat" is **CAPITALIZED** is because Mongo collection names are case sensitive ("Cats" is different from "cats"). Mongo convention calls for having collection names pluralized and lower-case, and model names singular and upper-case. Thus, the collection would be called "cats", whereas an individual model would be called "Cat". Mongoose will automatically lower-case and pluralize a model name so that it can access the collection (i.e., "Cat" > "cats").
- **Module Exports**
  - Rather than having your schema and model in the same file as your app.js, you can break down each schema-model pair into its own JS file and export the output to app.js. This is useful for modularizing those components that are likely to be used in other projects.
  - For example, if you have an online forum that would have "users" and "posts" as schema and models, you can create a separate JS file called "user.js" for your user schema-model and a file called "post.js" for your post schema-model. These files should be placed in a directory called "**Models**". You would configure your "post.js" file as follows:

```
var mongoose = require('mongoose');
var postSchema = new mongoose.Schema({
    title: String,
    content: String
});
module.exports = mongoose.model('Post', postSchema);
```

    - **NOTE**: Because the schema and model are created by Mongoose, it is necessary to "require" it for every file.
  - The "module.exports" essentially operates as a return function, which will export the returned value of "mongoose.model('Post', postSchema)" to app.js once post.js is "required" in the app.js file, e.g.:
    var Post = require('./models/post');
    - **IMPORTANT**: When referencing directories in Node, you must use **Dot-Slash** (./) to reference the **CURRENT DIRECTORY**.
  - **TIP**: Module exports are also useful for creating a modular **Seeds File**, which can be used to "seed" a database with placeholder data. Having such a file to populate a database helps expedite testing and debugging.

The seeds file should be "required" in the main app.js file and stored as a variable that can be executed as a function, e.g.:

```
var seedDB = require('./seeds');
seedDB();
```

… However, for this to work, it is necessary to have the seeds file export a function, e.g.:

```
function seedDB() {
    // insert code here
}
module.exports = seedDB;
```

- **Basic Methods**
  - **Create & Save**
    - Once a model has been compiled, you can create and save **Documents**, which are instances (i.e., the actual data object) of your model. You create the document by first stating, e.g.:

```
var newCat = new Cat({
    name: 'Stumpy',
    age: 9,
    temperament: 'Timid'
});
```

…and then save the model by stating, e.g.:

```
newCat.save(function(err, cat) {
    if (err) {
        console.log(err);
    } else {
        console.log('Saved:');
        console.log(cat);
    }
});
```

      - **NOTE**: The reason why you want to use the callback function when saving is that you may not always succeed if there is a problem with the database or your connection to it. Because this process may take time, it is worth having a callback function let you know whether the document saved or whether there was an error. The first argument (any name you want, but conventionally called "err") in the save method's function contains any **Error** message that was generated, and the second argument (any name you want) contains the **Result**, which is an object added to your database (the data with an ObjectId).
    - **BUT NOTE**: Rather than creating and saving in two separate steps, it is possible to use the **create()** method to perform both steps at once, e.g.:

```
Cat.create( {
    name: 'Koko',
    age: 9,
    temperament: 'Silly'
}, function(err, cat) {
    if (err) {
        console.log(err);
    } else {
```

```
        console.log('Saved:');
        console.log(cat);
    }
});
```

- **Find**
    - Once data has been added to the database, you can find data by using the following model method, e.g.:

```
Cat.find( {}, function(err, cats) {
    if (err) {
        console.log(err);
    } else {
        console.log('All Cats:');
        console.log(cats);
    }
});
```

    - The **find()** method can be used in tandem with Express to render a page that contains items in the database, e.g.:

```
app.get('/cats', function(req, res) {
    Cat.find( {}, function(err, allCats) {
        if (err) {
            console.log(err);
        } else {
            res.render('index.ejs', { cats: allCats });
        }
    });
});
```

- **Find By ID**
    - While find() may return one or more documents that fit the specified criteria, **findById()** will only return the single document that matches the specified _id. For example, if you want to render a page that shows the details about a specific cat in a collection, you can use a cat's _id property as a URL parameter (e.g., www.mycats.com/cats/a1b2c3/, in which "a1b2c3" is the _id), and access that parameter via Express's "req.params.id" statement and return the document object that matches said parameter through the following expression:

```
app.get('/cats/:id', function(req, res) {
    Cat.findById(req.params.id, function (err, foundCat) {
        if (err) {
            console.log(err);
        } else {
            res.render('show.ejs', { cat: foundCat });
        }
    });
});
```

- **Find By ID And Update**
    - When making a **PUT** request, you can use the **findByIdAndUpdate()** method to concurrently find the ID of the document to be modified (as defined by the first argument), immediately modify the document with a new data set (as defined by the second argument), and execute a callback

(as defined by the third argument). For example, if you used a form to edit the details about a specific cat in a collection, you can use "req.params.id" to return the matching document, and then run a command that will replace the data in that document with the new data object submitted in the body of the request (e.g., "req.body.cat"):

```
app.put('/cats/:id', function(req, res) {
    Cat.findByIdAndUpdate(req.params.id, req.body.cat, function(err,
    updatedCat) {
        if (err) {
            console.log(err);
        } else {
            console.log('Updated:');
            console.log(updatedCat);
        }
        res.redirect('/cats/' + req.params.id);
    });
});
```

- **Find By ID And Remove**
    - The **findByIdAndRemove()** method operates in relatively the same manner as findByIdAndUpdate(), except there is no need to include a new body of information:

```
app.delete('/cats/:id', function(req, res) {
    Cat.findByIdAndRemove(req.params.id, function(err, deletedCat) {
        if (err) {
            console.log(err);
        } else {
            console.log('Deleted:');
            console.log(deletedCat);
        }
        res.redirect('/cats/');
    });
});
```

        - **NOTE**: If you want to **REMOVE ALL DATA IN A COLLECTION**, use the **remove()** method with an empty set of curly brackets, e.g.:

```
Cat.remove({}, function(err) {
    if (err) {
        console.log(err);
    } else {
        console.log('Deleted all cats.');
    }
});
```

- **Data Associations**
    - **Basics**
        - A data association is a grouping of related elements. For example:
            - You can have a group of users that each have a property to define the user's spouse. As each user can only have one spouse, and a spouse can be married to one user, this is a **One-To-One** association.

        - You can have a group of businesses that each have a property to define the business's inventory. As each business can have multiple pieces of inventory (which can only belong to that one business), this is a **One-To-Many** association.
        - You can have a group of students that each have a property to define the courses they are enrolled in, and a group of courses that each have a property of the students who are enrolled. As each student can be enrolled in multiple courses, and each course can enroll multiple students, this is a **Many-To-Many** association.
- **Embedding Data**
    - Embedding data is one way of creating data associations. In the context of an online forum, you will have collections of users and collections of posts. With Mongoose, you can create schema for posts and users. To associate a post with a particular user, it becomes necessary to embed the post schema as an **Array** within the user schema, e.g.:

```
var postSchema = new mongoose.Schema({
    title: String,
    content: String
});
var Post = mongoose.model('Post', postSchema);
var userSchema = new mongoose.Schema({
    name: String,
    email: String,
    posts: [postSchema]
});
var User = mongoose.model('User', userSchema);
```

        - **NOTE**: For this process to work, the embedded schema must **PRECEDE** the schema in which it is embedded (i.e., the post schema must first be created before it can be embedded within the user schema).
    - Once a user has been created and you want to associate a post with that user, you must (1) **Find** the user (most likely by using **findOne()** if you are attempting to find by the user's name rather than ID), (2) **Push** the post to the user's "posts" property, and (3) **Save** the user, e.g.:

```
User.findOne({ name: 'John Smith' }, function(err, user) {
    if (err) {
        console.log(err);
    } else {
        user.posts.push({
            title: 'Post Title',
            content: 'Post Content'
        });
        user.save(function(err, user) {
            if (err) {
                console.log(err);
            } else {
                console.log('Saved:');
                console.log(user);
            }
```

```
        });
    }
});
```

- **NOTE**: The "second" user (in user.save) is different than the first (in User.findOne) due to scope.

- **Object References**
  - An alternative way to create data associations is via object references. This method is similar to embedding data to the extent that data is stored in an array; however, rather than storing posts in their entirety, the array will only store **IDs** that are references to the post objects (instead of embedding the entire post), e.g.:

```
var userSchema = new mongoose.Schema({
    name: String,
    email: String,
    posts: [
        {
            type: mongoose.Schema.Types.ObjectId,
            ref: 'Post'
        }
    ]
});
```

    - The syntax above essentially states that the user schema will contain a property called "posts", which is an array of object IDs (i.e., mongoose.Schema.Types.ObjectId) belonging to a 'Post' ("ref" refers to the model type that relates to the "ObjectId").
  - Once the applicable schema and models have been established, you can concurrently (1) **Create** a post, (2) **Push** the post to a specified user, and (3) **Save** the modified user data, e.g.:

```
Post.create({
    title: 'Post Title 2',
    content: 'Post Content 2'
}, function(err, post) {
    if (err) {
        console.log(err);
    } else {
        User.findOne({ email: 'john@smith.com' }, function(err, user) {
            if (err) {
                console.log(err);
            } else {
                user.posts.push(post.id);
                user.save(function(err, user) {
                    if (err) {
                        console.log(err);
                    } else {
                        console.log(user);
                    }
                });
            }
        });
```

```
    }
});
```

- Once you have associated a collection of post IDs with a particular user, you can find the entire data content of all posts (rather than just IDs) associated with the user by stating the following:

```
User.findOne({ email: '@' }).populate('posts').exec(function(err, user) {
    if (err) {
        console.log(err);
    } else {
        console.log(user);
    }
});
```

  - The syntax above essentially finds the specified user and then "**Populates**" the user's "posts" property with all of the data tied to all post IDs associated with the user. It is necessary to run "exec()" (**Execute**) at the end of the chain to actually start the query because no callback function is used in the "findOne()" method (as it would be pointless to do so until the "posts" property has been populated). The "exec()" method allows you to run the query after all the chaining and populating have finished.

- **Authentication**
  - **Passport.js**
    - Passport.js provides a simple user authentication process for all Express-based applications. Passport includes over 300 types of "**Strategies**", which are methods of authentication that include, *inter alia*, "local" authentication (i.e., traditional e-mail/password logins) or authentication via services such as Twitter, Google, etc.
    - One popular **Local** Passport strategy for simple **Username/Password** authentication is **Passport-Local**. If using Mongoose, you can implement such authentication even faster by also adding **Passport-Local-Mongoose**.
  - **Sessions**
    - Authentication is made possible through "sessions." HTTP is supposed to be a stateless protocol, which means that when a request is made, it does not contain any information about your history of activity on the site (i.e., a request does not have a "state"; it is a one-time transaction). However, a session applies a persistent "state" to a user's HTTP requests when that user is logged in to the server, thereby allowing encrypted information about the user to be saved to each HTTP request. Passport will then translate that encrypted login information when configuring its HTTP response.
    - **Express-Session** is an npm package that facilitates session creation in Express apps.
  - **Node/Express/Mongoose Setup**
    - **Installation**
      - Apply the standard npm installation commands for (1) Body-Parser, (2) EJS, (3) Express, (4) Express-Session, (5) Mongoose, (6) Passport, (7) Passport-Local, and (8) Passport-Local-Mongoose:

```
npm install express body-parser ejs express-session mongoose passport passport-local passport-local-mongoose --save
```

    - **Configuration**
      - **(1)** Set the standard "require" variables, but note that convention calls for naming the Passport-Local variable "**LocalStrategy**", and the Express-Session variable

"**session**":

```
var express              = require('express'),
    session              = require('express-session'),
    bodyParser           = require('body-parser'),
    mongoose             = require('mongoose'),
    passport             = require('passport'),
    LocalStrategy        = require('passport-local'),
    passportLocalMongoose = require('passport-local-mongoose'),
    app                  = express();
```

- **NOTE**: Generally, variables start with a lowercase letter. However, convention calls for capitalizing **Constructors**, which are "blueprints" for creating many objects of the same "type." In this case, LocalStrategy is one such constructor and thus begins with an uppercase letter. You can typically know that something is a constructor if it is used in tandem with the "**new**" operator to create an object (e.g., **new** mongoose.**Schema**…).

(2) Instruct Express to use Express-Session, e.g.:

```
app.use(session({
    secret: 'whatever you want';
    resave: false,
    saveUninitialized: false
}));
```

- **NOTE**: The **Secret** phrase is used to encode and decode the session; thereby allowing encrypted data to be stored during the session, rather than storing the username and password in plain text. Refer to the **Express-Session Documentation** for more information on "resave" and "saveUninitialized". However, for basic usage purposes, simply know that those properties will default to "true" if left unspecified, and you will generally want those changed to "false".

(3) Instruct Express to use Passport by entering the following commands in app.js:

```
app.use(passport.initialize());
app.use(passport.session());
```

- **NOTE**: The second line is required for applications that use **Persistent Login Sessions** (which is recommended). This "session" actually refers to a strategy that is bundled with Passport (**NOT** the variable "session" for "express-session").
- **IMPORTANT**: These two lines of code must appear **AFTER THE EXPRESS-SESSION** code in step 2 above. Otherwise, authentication will not work.

(4) Configure Mongoose, create your model file(s) (e.g., models/user.js), and add Passport to your model(s) to be exported back to your app, e.g.:

```
var mongoose = require('mongoose');
var passportLocalMongoose = require('passport-local-mongoose');
var userSchema = new mongoose.Schema({
    username: String,
    password: String
});
userSchema.plugin(passportLocalMongoose);
module.exports = mongoose.model('User', userSchema);
```

- **NOTE**: The fourth line adds methods from the passport-local-mongoose package to the User schema, and it works in tandem with step 5 below.

(5) Create a User variable that requires the exported data from step 4 above, and instruct your app use Passport's **Serialization** commands to encode and decode each User session:

```
var User = require('./models/user');
passport.use(new LocalStrategy(User.authenticate()));
passport.serializeUser(User.serializeUser());
passport.deserializeUser(User.deserializeUser());
```

- **NOTE**: The second line creates a new LocalStrategy (via passport-local) by using the authenticate() method that was imported from the User model (via passport-local-mongoose).
- **ALSO NOTE**: serializeUser() and deserializeUser() are methods added from passport-local-mongoose via the plugin method in step 4 above.

- **Routing**
  - **Registration**
    - After creating a registration form (with, for example, a post request to "/register" and with two inputs named "username" and "password") for accessing a "secret" page, create the form's **POST Route** as follows:

```
app.post('/register', function(req, res) {
    User.register(new User({ username: req.body.username }),
        req.body.password, function(err, user) {
        if (err) {
            console.log(err);
            return res.redirect('/register');
        }
        passport.authenticate('local')(req, res, function() {
            res.redirect('/secret');
        });
    });
});
```

      - **IMPORTANT**: The **register()** method is used to create the username and password according to the User model. However, it is important to note that the password entered by the user is **NOT SAVED TO THE USER MODEL**; rather, it is passed as the second argument in the register method. This allows the password to be encoded and saved to the User model in an encoded format (by means of **Salted Password Hashing**).
      - **ALSO NOTE**: The **authenticate()** method is used to specify which Passport strategy you are using (in this case, "local").
  - **Login**
    - After creating a login form (with, for example, a post request to "/login" and with two inputs named "username" and "password") for accessing a "secret" page, create the form's **POST Route** as follows:

```
app.post('/login', passport.authenticate('local', {
    successRedirect: '/secret',
    failureRedirect: '/login'
}), function(req, res) {});
```

      - **NOTE**: The **authenticate()** method is being used here as **Middleware**, which is the name given to any code that runs immediately after the route is accessed but before the final

callback at the end. This particular piece of middleware is set up (at least in part) by the second line in step 5 of the configuration section above. In this case, authenticate() attempts to log the user in. If the attempt is successful, then the user will be redirected to the "secret" page. If the attempt fails, then the user will be redirected back to the "login" page.

- **ALSO NOTE**: Technically, the final callback at the end is not required, but it should be retained to remind you that the authenticate() method is middleware.

- **Logout**
  - After creating a logout link that directs to "/logout", create a **GET Route** as follows:

```
app.get('/logout', function(req, res) {
    req.logout();
    res.redirect('/');
});
```

- **Authentication (Middleware)**
  - To stop someone from accessing a "secret" page if they are not logged in, it is necessary to create your own **Middleware Function** to the "secret" route. For example, if the user is logged in, then the route will render the "secret" page; otherwise, the user will be directed to the index page:

```
function isLoggedIn(req, res, next) {
    if (req.isAuthenticated()) {
        return next();
    }
    res.redirect('/login');
}
```

…which can then be added as middleware to the "secret" route as follows:

```
app.get('/secret', isLoggedIn, function(req, res) {
    res.render('secret');
});
```

    - **NOTE**: The three arguments for the "isLoggedIn" function are standard arguments that are recognized by Express. The "next" argument is understood by Express to require that the code following the middleware be executed if the isAuthenticated() method returns true.
  - If you want to pass through information about the user to be manipulated as a JS object in your rendered templates, this can be done by accessing "**req.user**", which contains all of the information about the logged-in user who is submitting a particular request, e.g.:

```
app.get('/', function(req, res) {
    res.render('index', { currentUser: req.user });
});
```

…which then allows you to customize page display depending on whether or not a user is logged in (e.g., display login/registration links if a user is not signed in, but otherwise display the user's name and a logout link if a user is signed in):

```
<% if (!currentUser) { %>
    <a href="/login">Login</a>
    <a href="/register">Sign Up</a>
<% } else { %>
```

```
        <span>Welcome, <%= currentUser.username %></span>
        <a href="/logout">Logout</a>
    <% } %>
```

- **IMPORTANT**: If you want to have access to req.user on **EVERY ROUTE** without having to manually pass through an object each time a page is rendered, then simply use the following **Middleware Function** (to be run on every Express route):

```
app.use(function(req, res, next) {
    res.locals.currentUser = req.user;
    next();
});
```

  - Essentially, whatever value is assigned to "res.locals.object" will be made available inside of every template.
  - **NOTE**: This function must go **AFTER** the code listed in step 3 of the configuration section above.
- **ALSO IMPORTANT**: If you are performing **Authorization** (i.e., only a user who created a comment has the authority to edit it), then you will have to compare req.user._id with the ID of the creator of the comment. However, the comment creator's ID (generated pursuant to a schema) is stored as a Mongoose **OBJECT**, whereas req.user._id is stored as a **STRING**. Therefore, it is necessary to convert the former to a string to make a comparison (via the **String()** function). Alternatively, Mongoose comes with a built-in **Equals** method that performs this task, e.g.:

```
if (foundComment.author.id.equals(req.user._id)) {
    // insert code here
}
```

- **NOTE**: For more information about authentication in Node, refer to the following presentations by Randall Degges:
  - Everything You Ever Wanted to Know about Node Authentication (2014)
  - Everything You Ever Wanted to Know About Web Authentication in Node (2017)

- **Refactoring Middleware**
  - Rather than keeping middleware in separate files, you can combine them into a single JS file. In that file, you can create a **Middleware Object** to which you can append all of the middleware functions as **Methods**. (In JS, every function is an object. An object is a collection of key:value pairs. When the value is a primitive (i.e., integer, string, or Boolean) or another object, the value is called a "property." When the value is a function, it is called a "method.") Example:

```
var middleware = {
    firstMiddleware: function() {
        // do something
    },
    secondMiddleware: function() {
        // do something
    }
}
module.exports = middleware;
```

    - **NOTE:** Alternatively, you may create an empty middleware object (i.e., var middleware = {};) and then separately add each method to the middleware object, e.g.:

```
middleware.firstMiddleware = function() {
```

```
        // do something
    };
```

- When using Node, convention calls for naming the compiled middleware file **Index.js**, and saving it to an appropriately titled directory (e.g., "middleware"). This is because Node always uses the file named index.js by **Default** when a directory has been required. Thus, the "require" code need only state the directory path rather than the directory and specific file name, e.g.:
  `var middleware = require('../middleware');`

- **Flash Messages**
  - **Connect-Flash**:
    - Connect-flash uses the **Flash**, which is a special area of the session used for storing messages. Messages are written to the flash and cleared after being displayed to the user. The flash is typically used in combination with redirects, ensuring that the message is available to the next page that is to be rendered.
  - **Installation & Configuration**:
    - Apply the standard npm command, require the package, and use it (**PRIOR TO** Passport):
      ```
      npm install connect-flash --save
      var flash = require('connect-flash');
      app.use(flash());
      ```
    - **NOTE**: The installation instructions on the npm page also include references to "cookieParser" and "session". However, this inclusion is not necessary if you are already using express-session.
  - **Usage**:
    - When you want to display a message, use the following syntax (following a key-value pair format) **BEFORE** redirecting to the page in which the message will be displayed:
      ```
      req.flash('keyName', 'valueMessage');
      res.redirect('/pageName');
      ```
    - For example, if you want to have an error message be displayed to an unauthenticated user and then redirect the user to the login page, you could add the following to your authentication middleware…
      ```
      req.flash('error', 'Please login first.');
      res.redirect('/login');
      ```
      …and then you must **Handle** the flash message in the actual login route…
      ```
      router.get('/login', function(req, res) {
          res.render('login', { message: req.flash('error') });
      });
      ```
      …and then add the message to the appropriate login.ejs **Template**, e.g.:
      ```
      <div><%= message %></div>
      ```
      - **IMPORTANT**: Per the connect-flash docs, you may either set a flash message on the req.flash object before returning res.**redirect** **OR** you may pass the req.flash object into the res.render function; **HOWEVER**, it will not work if you attempt to set a flash message on the req.flash object before returning res.**render**.
      - **NOTE**: If you want to have access to req.flash on **EVERY ROUTE** without having to manually pass through an object each time a page is rendered, then simply use the following **Middleware Function** that is also used for req.user as noted in the "Authentication" section above:
        ```
        app.use(function(req, res, next) {
            res.locals.currentUser = req.user;
            res.locals.message = req.flash('error');
        ```

```
    next();
});
```

- **ALSO NOTE**: The "message" need not be named as such, and can be given any name to distinguish it from **Multiple** flash messages, e.g.:

```
app.use(function(req, res, next) {
    res.locals.currentUser = req.user;
    res.locals.error = req.flash('error');
    res.locals.success = req.flash('success');
    next();
});
```

- **Git**
  - **About**
    - Git is a free and open source distributed version control system.
    - **NOTE**: If you are using Cloud9, you do not have to install Git because it comes pre-installed. Otherwise, follow the **Installation** instructions.
  - **Basic Commands**
    - **Init**
      - To tell Git which directory it should monitor for changes, you must first cd $DIR into the directory that you want to track, at which point Git will monitor that directory and all subdirectories upon executing the following code:
        `git init`
      - Upon execution, Git will create a **Hidden Folder** named **/.git** in your workspace. This hidden folder is where Git will track all changes to your workspace.
        - **NOTE**: To view hidden folders in the terminal, type: ls -a
        - **ALSO**: If you accidentally git init in the wrong directory, you can remove /.git like any other directory: rm -rf .git
    - **Status**
      - Asks Git to provide a status report on your project. It will include information such as which **Branch** of the project you are on (e.g., master), which **Commit** you are on, whether any changes will be committed, whether any files have been modified since the last commit, and whether there are any **Untracked Files**.
    - **Add**
      - With respect to untracked files, Git does not automatically track every file within the directories monitored by init. Rather, a file must be specifically tracked by issuing the following command:
        `git add $FILE`
    - **Commit**
      - Once a file has been tracked, you can save the changes to a new version checkpoint in Git by running the following command:
        `git commit -m "Explanatory Message (in Present Tense)"`
        - **NOTE**: Every commit must include a message explaining the changes that you are making. The -m flag is used to signal the inclusion of a message. If you do not include the -m flag with the message in quotes, you will be prompted to enter a message in the terminal.
        - **IMPORTANT**: After you submit a commit for any given file, Git will not automatically stage that file for new commits in the future. Instead, you must use add each time (i.e., a two-step process of add and commit). If

you want to add **ALL FILES** that have yet to be staged for a commit, use:
git add .

- **Log**
  - Displays a log of all the commits made to the project:
    git log
  - For each commit, the log will include a long **Commit String**, which is the unique identifier for viewing each commit by using the following command.
- **Checkout**
  - Primarily used for displaying a previous commit or may be used to change to a different **Branch**:
    git checkout $COMMIT
    …or…
    git checkout -b $BRANCH
    - To return to the **Master Branch**, simply type:
      git checkout master
- **Revert**
  - There are multiple ways to revert back to (rather than simply view) a previous commit (see, e.g., How to revert Git repository to a previous commit?), but one of the safest methods is as follows, which will revert everything back from the current HEAD to the specified commit string:
    git revert --no-commit $COMMIT..HEAD
    …at which point you will then have to commit the reversion:
    git commit -m "Explanatory Message"
  - One of the main benefits to this method is that there are no actual deletions of any previous commits, and they can still be accessed through the log if desired.

- **Additional Resources**
  - Learn Git and GitHub Basics for Free
  - Linking GitHub to Cloud9

- **Deployment**
  - **Heroku**
    - **About**
      - Heroku is a cloud computing service that provides a platform to develop, run, and manage web applications without the complexity of building and maintaining the infrastructure typically associated with deploying an application.
    - **Basic Deployment (VIA CLOUD9)**
      - Ordinarily, to manage your application with Heroku, it is necessary to download and install the **Heroku Command Line Interface** (CLI), which was formerly known as the Heroku Toolbelt. However, the Heroku CLI is built into Cloud9, and can be accessed by typing the following into the Cloud9 console:
        heroku
      - To deploy an application from Cloud9 to Heroku, you must first **Log In** to your Heroku account from the Cloud9 console:
        heroku login
      - After logging in, ensure that you are currently in the directory that contains your application (i.e., the application that contains your package.json file) and run the following code to determine whether your workspace has been made into a **Git Repository:**

git status
…and if it has not been made into a repository, then do so as follows (as git is used in determining which files are sent to Heroku):
git init

- After initializing your repository, use "git status" to see all of the files and directories that are not yet tracked, and use the "git add $FILE/DIRECTORY" command in the console to **Add** any untracked files and then **Commit**:
  git commit -m "commit message"
- After committing your project, you must next **Create** an application on Heroku (with a randomized name), which prepares Heroku to receive your source code:
  heroku create
  - **NOTE**: After completing this step, the console will display a URL that will be the location to which your app will be deployed. You can later specify your own **Custom Domain Name** (e.g., https://myname.herokuapp.com/) or configure Heroku use your own private domain.
- After creating the app on Heroku, a Git **Remote** will be created and can be accessed via the Cloud9 console. Git remotes are versions of your repository that live on other servers. You deploy your app by pushing its code to a special Heroku-hosted remote that is associated with your app. You can use the following command to confirm that a remote named "heroku" has been made:
  git remote -v
- After creating a remote, your application will be ready to deploy; however, you must first configure your package.json file to include a **Start Script**. This script is responsible for telling Heroku what command must be run to start the application. Assuming you are using node.js, and assuming your application is named "app.js", then you would include following language:

```
"scripts": {
    "start": "node app.js"
}
```

- After successfully creating a remote and configuring your "start" script, you can use the **Push** command to push your code from your local repository's "master" branch to your Heroku remote:
  git push heroku master
  - **NOTE**: If your application crashes upon deployment, a non-specific **Error** message will be displayed at the herokuapp.com URL. To actually view the specific information regarding the error, check the **Logs** from Cloud9:
    heroku logs
  - **ALSO NOTE**: If you need to **View** which files/directories have been exported to Heroku, run the following command in Cloud9:
    heroku run ls
    - Essentially, you can execute any CLI command in Heroku by starting with the phrase "heroku run" followed by the command.
  - **IMPORTANT**: If you needed to modify a file in your application to correct the error, then you cannot merely use the push command immediately after changing the file. Rather, such changes must first be **ADDED AND COMMITTED**.

- **mLab**
  - **About**

- mLab (formerly known as "MongoLab") is a fully-managed cloud database service that hosts MongoDB databases. mLab will provide you with a persistent URI to your hosted MongoDB database that will replace the "localhost" connection link in your Heroku-hosted application.

- **Usage**
  - After creating an account on mLab, access your dashboard for "**MongoDB Deployments**" and select "**Create New**".
  - After creating a database, you will receive a **URI** to be used in connecting your application to the database, e.g.: mongodb://<dbuser>:<dbpassword>@ds123456.mlab.com:12345/your-database-name
  - Upon receiving the URI, it is necessary to add a database user. Select the "Users" tab and select "**Add Database User**".
  - After creating a database user, **Replace** <dbuser> and <dbpassword> in the above URI in your application with the username and password.
  - Finally, add and commit the changes to your application, and push to Heroku.

- **Environment Variables**
  - In the section entitled "Listening" under "Express.js" above, it is noted that the following code must be included in the application:
    app.listen(process.env.PORT, process.env.IP);
  - The above-referenced "PORT" and "IP" are environment variables, which are values that can vary depending on the environment in which the application is being run.
  - When deploying an application with a database, it is beneficial to use an environment variable in reference to your database **URI**. For example, you may have your application hosted on Cloud9 for testing, and also hosted on Heroku for deployment. However, if each copy has its data hosted on the same mLab database, then modifications made in testing will also carry over to the deployed application.
  - To ensure that your testing and deployment applications connect to separate databases, you must convert the value in the following line:
    mongoose.connect('mongodb://localhost/your_application');
    …to an environment variable, e.g.:
    mongoose.connect('process.env.DATABASEURI');
    - **NOTE**: As a matter of convention, environment variables are typed in **ALL CAPS**.
  - Once this change has been made, you can then set the value of DATABASEURI in **Cloud9** by using the **Export** command:
    export DATABASEURI=mongodb://localhost/your_application
    …and you can set the value in **Heroku** by running the following in the Cloud9 console:
    heroku config:set DATABASEURI= mongodb://<dbuser>:<dbpassword>@ds123456.mlab.com:12345/your-database-name
    - **NOTE**: Alternatively, you can set environmental variables in the Heroku dashboard by going to "**Settings**" and selecting "**Config Vars**".
    - **SEE Heroku Node.js Support** for more information on deploying a Node.js application on Heroku.
  - **TIP**: Environment variables are also useful for concealing sensitive information.
  - **NOTE**: If you want to have a **Backup URI**, you can use the following procedure, e.g.:
    var uri = process.env.DATABASEURI **||** '**mongodb://your/backup/uri**';
    mongoose.connect(uri);
    - In the event that the value of DATABASEURI is undefined, your application will still connect to the backup URI that has been explicitly provided.