HTML, CSS, JavaScript and jQuery notes

Contents

1. HTML & CSS 8
   1.1. HTML Structure 8
   1.2. HTML Trees 8
   1.3. HTML Syntax 9
   1.4. HTML Cheat Sheet 10
   1.5. CSS Rules and syntax 11
   1.6. CSS tips 11
   1.7. CSS Grid 14
      Grid container 14
      Grid Items 16
   1.8. CSS Cheat Sheet 18
2. Atom shortcuts 20
3. javaScript intro 21
   3.1. Console 21
   3.2. Data types & Variables 21
      3.2.1. Data types 21
         Null, Undefined, and NaN 23
      3.2.2. Operators 23
      3.2.3. Strict equality 24
      3.2.4. Implicit type coercion 24
         *Comments 25
      3.2.5. Variables 25
         Naming conventions 26
         Google JavaScript Style Guide 26
         Constant variables (cannot be reassigned) 26
         Let variables (can be reassigned) 27
         Indexing 27

Escaping strings 28
Special characters 29
Comparing strings 29
String Interpolation 30
3.3. Quizzes and examples 31
Quiz: Converting Temperatures 31
Quiz: All Tied Up 31
Quiz: Yosa Buson 32
Quiz: Out to Dinner 32
Quiz: Mad Libs 33
Quiz: One Awesome Message 34
4. javaScript Conditionals 35
4.1. If...else statements 35
Tue or false values (Truthy and Falsy) 36
Else if statements 38
Logical operators 40
4.2. Ternary Operator 42
4.3. Switch Statement 43
Break statement 44
Falling through 45
4.4. Quizzes and examples 46
Quiz: Even or Odd 46
Quiz: Musical Groups 47
Quiz: Murder Mystery 48
Quiz: Checking your Balance 50
Quiz: Ice Cream 52
Quiz: What do I Wear? 53

Quiz: Navigating the Food Chain 55
Quiz: Back to School 56
5. JavaScript Loops 58
5.1. While loops 58
5.2. For Loops 60
for Loops, Backwards 61
5.3. Nested Loops 62
5.4. Increment and decrement 63
5.5. Quizzes and examples 64
Quiz: JuliaJames 64
Quiz: 99 Bottles of Juice 65
Quiz: Countdown, Liftoff! 66
Quiz: Changing the Loop 67
Quiz: Fix the Error 1 68
Quiz: Fix the Error 2 68
Quiz: Factorials! 69
Quiz: Find my Seat 70
6. JavaScript functions 71
6.1. How to declare a function 72
Return statements 72
How to *run* a function 73
Parameters vs. Arguments 73
6.2. Returning with Logging 74
6.3. Scope 77
Shadowing 79
Using global variables 80
Hoisting 80

6.4. Function Expressions 82
Function expressions and hoisting 83
Named function expressions 84
6.5. Patterns with Function Expressions 84
Functions as parameters 84
Inline function expressions 84
Why use anonymous inline function expressions? 85
6.6. Quizzes and examples 86
Quiz: Laugh it Off 1 86
Quiz: Laugh it Off 2 86
Quiz: Build a Triangle 87
Quiz: Laugh 89
Quiz: Cry 89
Quiz: Inline 90
7. JavaScript Arrays 91
7.1. Array Index 93
7.2. Array Properties and Methods 94
Property: Length 94
Method: Push 95
Method: Splice 96
7.3. Array Loops 96
Method: forEach() 97
Method: map() 99
7.4. Quizzes and examples 101
Quiz: UdaciFamily 101
Quiz: Building the Crew 101
Quiz: Joining the Crew 102

Quiz: The Price is Right 103
Quiz: Colors of the Rainbow 103
Quiz: Quidditch Cup 104
Quiz: Another Type of Loop 105
Quiz: I Got Bills 106
Quiz: Nested Numbers 107
8. JavaScript Objects 108
8.1. Object Literals 110
8.2. Naming Conventions 111
8.3. Quizzes and examples 114
Quiz: Umbrella 114
Quiz: Menu Items 115
Quiz: Bank Accounts 1 116
Quiz: Facebook Friends 117
Quiz: Donuts Revisited 118
9. jQuery 119
9.1. Hosting jQuery 120
9.2. jQuery Selectors 121
9.3. Traversing the DOM 122
9.4. jQuery tricks 123
.ready() 123
.toggleClass() 124
.next() 124
.attr() 124
.css() 125
.html() .text() .val() 125
.remove() 127

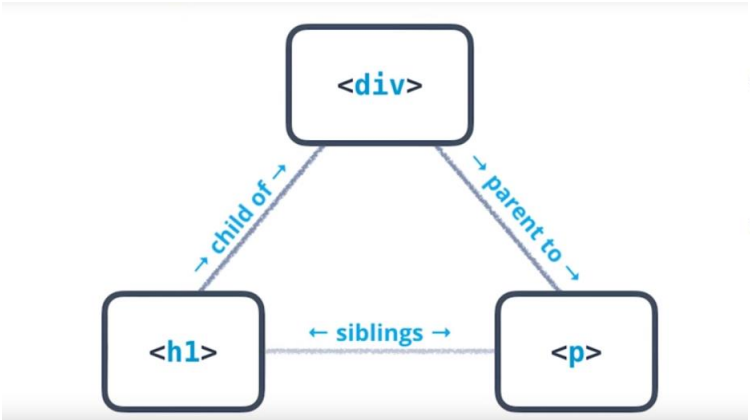
.append() .prepend() .insertBefore() .insertAfter() 128
.each() 129
9.5. Mouse Events 130
The Event Object 133
The Convenience Method 134
Event Delegation 134

1. HTML & CSS

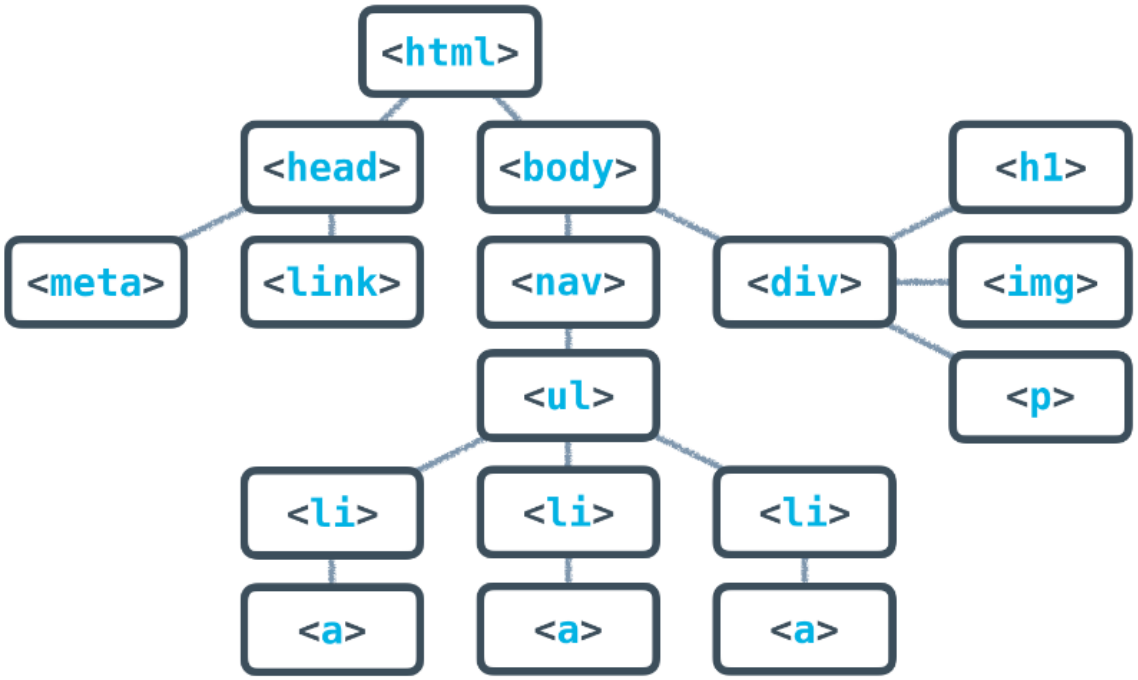
1.1. HTML Structure

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Title of the document</title>
</head>

<body>
The content of the document.....
</body>

</html>
```

1.2. HTML Trees




```
<div>
  <h1>Article Title</h1>
  <p>Paragraph of text.</p>
</div>
```



Full HTML Tree Structure

1.3. HTML Syntax

- Button

```
<button>a button!</button>
```

- Link

```
<a href="https://www.udacity.com">Udacity</a>
```

- Image

```
<img src="http://somewebsite.com/image.jpg" alt="short description">
```

- Paths

External

```
<a href="http://labs.udacity.com/fend/example/hello-world.html">Hello, world!</a>
```

Internal

```
<a
```

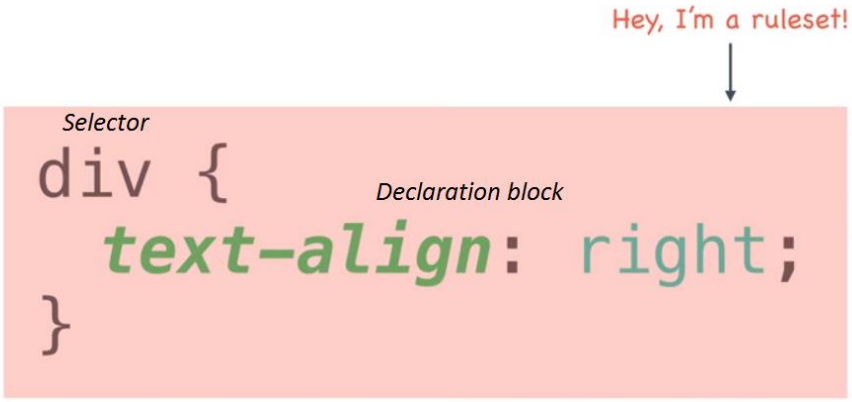
```
target="_blank" would
open in new tib/window
```

```
href="/Users/cameron/Udacity/etc/labs/fend/example/hello-world.html"> Hello, world!</a>
```

1.4. HTML Cheat Sheet

TAGS	MEANING	TAGS	MEANING	TAGS	MEANING
<a>	hyperlink	<h1>	heading level 1	<progress>	progress of a task
<abbr>	abbreviation	<h2>	heading level 2	<q>	short quotation
<address>	address element	<h2>	heading level 2	<rb>	marks the base text component of a ruby annotation.
<area>	area inside an image map	<h3>	heading level 3		
<article>	article	<h4>	heading level 4	<rp>	used for the benefit of browsers that don't support ruby annotations
		<h5>	heading level 5		
<aside>	content aside from the page content	<h6>	heading level 6	<rt>	ruby text component
<audio>	sound content	<head>	information about the document	<rtc>	marks a ruby text container for ruby text components in a ruby annotation.
<b>	bold text	<header>	group of introductory or navigational aids		
<base>	base URL for all the links in a page			<ruby>	ruby annotation
<bdi>	bi-directional text formatting	<hr>	horizontal rule	<s>	Indicates text that's no longer accurate or relevant.
		<html>	html document		
<bdo>	direction of text display	<i>	italic text	<samp>	sample computer code
<blockquote>	long quotation	<iframe>	inline sub window (frame)	<script>	script
<body>	body element	<img>	image	<section>	section
 	single line break	<input>	input field	<select>	selectable list
<button>	push button	<ins>	inserted text	<small>	small text
<canvas>	define graphics	<kbd>	keyboard text	<source>	media resources
<caption>	table caption	<keygen>	generates a key pair		section in a document
<cite>	citation	<label>	label for form control		strong text
<code>	computer code text	<legend>	title in a fieldset	<style>	style definition
<col>	table columns		list item	<sub>	subscripted text
<colgroup>	groups of table columns	<link>	resource reference	<summary>	summary / caption for the <details> element
<data>	Allows machine-readable data to be provided	<main>	main content area of an HTML document.	<sup>	superscripted text
<datalist>	"autocomplete" dropdown list	<map>	image map	<table>	table
		<mark>	marked text	<tbody>	table body
<dd>	definition description	<menu>	menu list	<td>	table cell
<del>	deleted text	<menuitem>	command that user can invoke from popup menu	<textarea>	text area
<details>	details of an element			<tfoot>	table footer
<dfn>	definition term	<meta>	meta information	<th>	table header
<dialog>	part of an application is interactive.	<meter>	measurement within a predefined range	<thead>	table header
				<time>	date/time
	section in a document	<nav>	navigation links	<title>	document title
<dl>	definition list	<noscript>	noscript section	<tr>	table row
<dt>	definition term	<object>	embedded object	<track>	text track for media such as video and audio
	emphasized text		ordered list	<u>	text with a non-textual annotation.
<embed>	external application or interactive content	<optgroup>	option group		unordered list
<fieldset>	fieldset	<option>	option in a drop-down list	<var>	variable
<figcaption>	caption for the figure element.	<output>	types of output	<video>	video
<figure>	group of media content, and their caption	<p>	paragraph	<wbr>	line break opportunity for very long words and strings of text with no spaces.
<footer>	footer section or page	<param>	parameter for an object		
<form>	specifies a form	<pre>	preformatted text		

1.5. CSS Rules and syntax



Tag Selector

```
h1 {
  color: green;
}
```

class Attribute Selector

```
.book-summary {
  color: blue;
}
```

id attribute selector

```
#site-description {
  color: red;
}
```

How to Link to a Stylesheet?

```
<head>
  <title>My Webpage</title>
  <!-- ... -->
  <link href="path-to-stylesheet/stylesheet.css" rel="stylesheet">
  <!-- ... -->
</head>
```

1.6. CSS tips

One of the most common mistakes beginners and intermediates fall victim to when it comes to CSS is not **removing the default browser styling**.

```
* {
 margin: 0px;
 padding: 0px;
 border: 0px;
 }
```

Keep a library of helpful CSS classes. For this part of the master stylesheet, you should define classes that make sense to you. Note that they are not set to any particular element, so you can assign them to whatever you need:

```
.floatLeft { float: left; }
.floatRight { float: right; }
.textLeft { text-align: left; }
.textRight { text-align: right; }
.textCenter { text-align: center; }
.textJustify { text-align: justify; }
.bold { font-weight: bold; }
.italic { font-style: italic; }
.underline { text-decoration: underline; }
.noindent { margin-left: 0; padding-left: 0; }
.nomargin { margin: 0; }
.nopadding { padding: 0; }
.width100 { width: 100%; }
.width75 { width: 75%; }
.width50 { width: 50%; }
```

Organize your CSS-styles, using flags. Divide your stylesheet into specific sections: i.e. Global Styles – (body, paragraphs, lists, etc), Header, Page Structure, Headings, Text Styles, Navigation, Forms, Comments, Extras.

```
/* ----------------------------------*/
/* ---------->>> GLOBAL <<<----------*/
/* ----------------------------------*/
```

Organize your CSS-styles, making a table of contents. At the top of your CSS document, write out a table of contents. For example, you could outline the different areas that your CSS document is styling (header, main, footer etc). Then, use a large, obvious section break to separate the areas.

```
Global Styles – (body, paragraphs, lists, etc)
Header
Page Structure
Headings
Text Styles
Navigation
Forms
Comments
Extras
```

It is important that you **define the basic rules for each area only once** so that the same default value is not being rewritten in every rule. If you know that all of the h2's will not

have a margin or padding, define that on the top level and let its effect cascade as it is supposed to.

Organize your CSS-styles, ordering properties alphabetically.. "I don't know where I got the idea, but I have been alphabetizing my CSS properties for months now, and believe it or not, it makes specific properties much easier to find."

```
body {
   background: #fdfdfd;
   color: #333;
   font-size: 1em;
   line-height: 1.4;
   margin: 0;
   padding: 0;
}
```

Keep properties to a minimum.. "Work smarter, not harder with CSS. Under this rule, there are a number of subrules: if there isn't a point to adding a CSS property, don't add it; if you're not sure why you're adding a CSS property, don't add; and if you feel like you've added the same property in lots of places, figure out how to add it in only one place."

Keep containers to a minimum.. "Save your document from structural bloat. New developers will use many div's similar to table cells to achieve layout. Take advantage of the many structural elements to achieve layout. Do not add more div's. Consider all options before adding additional wrappers (div's) to achieve an effect when using a little nifty CSS can get you that same desired effect."

Use CSS Constants for faster development.. "The concept of constants – fixed values that can be used through your code [is useful]. [..]

```
# /*
# Dark grey (text): #333333
# Dark Blue (headings, links) #000066
# Mid Blue (header) #333399
# Light blue (top navigation) #CCCCFF
# Mid grey: #666666
# */
```

To ensure that you see your various link styles, you're best off putting your styles in the order **"link-visited-hover-active"**, or "LVHA" for short.

```
a:link { color: blue; }
a:visited { color: purple; }
a:hover { color: purple; }
a:active { color: red; }
```

1.7. CSS Grid

Note: CSS Grid is supported in the most recent versions of many browsers, but it is not supported universally. To make sure that you can get the most out of this course, check your browser version and see if it supports CSS Grid. If CSS Grid is not supported in your browser, you should switch or update to a supported browser and version.

To set up a grid, you need to have both a *grid container* and *grid items*. The grid container will be a parent element that contains grid items as children and applies overarching styling and positioning to them.

Grid container

To turn an HTML element into a grid container, you must set the element's display property to grid (for a block-level grid) or inline-grid (for an inline grid).
We can define the columns of our grid by using the CSS property grid-template-columns. To specify the number and size of the rows, we are going to use the property grid-template-rows. This property is almost identical to grid-template-columns.

Below is an example of this property in action:

```
.grid {
  display: grid;
  width: 500px;
  height: 500px;
  grid-template-columns: 100px 200px;
  grid-template-rows: 10% 20% 600px;

}
```

grid-template-columns creates two changes. First, it defines the number of columns in the grid; in this case, there are two. Second, it sets the width of each column. The first column will be 100 pixels wide and the second column will be 200 pixels wide. grid-template-rows defines the number of rows and sets each row's height. In this example, the first row is 50 pixels tall (10% of 500), the second row is 100 pixels tall (20% of 500), and the third row is 600 pixels tall.
We can also define the size of our columns as a percentage of the entire grid's width or combine the units.

When using percentages, remember that rows are defined as a percentage of the grid's height, and columns are defined as a percentage of its width.

The property grid-template can replace the previous two CSS properties.

```
.grid {
 display: grid;
 width: 1000px;
 height: 500px;
 grid-template: 200px 300px / 20% 10% 70%;  /* row / column */
}
```

When using grid-template, the values before the slash will determine the size of each row. The values after the slash determine the size of each column. In this example, we've made two rows and three columns of varying sizes.

Fraction

By using the `fr` unit, we can define the size of columns and rows as a fraction of the grid's length and width. This unit was specifically created for use in CSS Grid. Using `fr` makes it easier to prevent grid items from overflowing the boundaries of the grid. Consider the code below:

```
.grid {
  display: grid;
  width: 1000px;
  height: 400px;
  grid-template: 2fr 1fr 1fr / 1fr 3fr 1fr;
}
```

In this example, the grid will have three rows and three columns. The rows are splitting up the available 400 pixels of height into four parts. The first row gets two of those parts, the second column gets one, and the third column gets one. Therefore, the first row is 200 pixels tall, and the second and third rows are 100 pixels tall.

Each column's width is a fraction of the available space. In this case, the available space is split into five parts. The first column gets one-fifth of the space, the second column gets three-fifths, and the last column gets one-fifth. Since the total width is 1000 pixels, this means that the columns will have widths of 200 pixels, 600 pixels, and 200 pixels respectively.

It is possible to use `fr` with other units as well. When this happens, each `fr` represents a fraction of the *available* space.

Repeat

```
.grid {
  display: grid;
  width: 300px;
  grid-template-columns: repeat(3, 100px);
}
```

The repeat function will duplicate the specifications for rows or columns a given number of times. In the example above, using the repeat function will make the grid have three columns that are each 100 pixels wide.

Repeat is particularly useful with `fr`. For example, `repeat(5, 1fr)` would split your table into five equal rows or columns.

Finally, the second parameter of `repeat()` can have multiple values.

```
grid-template-columns: repeat(2, 20px 50px);
```

This code will create four columns where the first and third columns will be 20 pixels wide and the second and fourth will be 50 pixels wide.

Grid Gap

```
.grid {
  display: grid;
  width: 320px;
  grid-template-columns: repeat(3, 1fr);
  grid-column-gap: 10px;
}
```

It is important to note that grid-gap does not add space at the beginning or end of the grid. In the example code, our grid will have three columns with two ten-pixel gaps between them.

There is a CSS property grid-gap that can set the row and column gap at the same time. grid-gap: 20px 10px; will set the distance between rows to 20 pixels and the distance between columns to 10 pixels. Unlike other CSS grid properties, this shorthand does not take a /between values! If only one value is given, it will set the column gap and the row gap to that value.

Grid Items

Using the CSS properties grid-row-start and grid-row-end, we can make single grid items take up multiple rows.

```
.item {
  grid-row-start: 1;
  grid-row-end: 3;
}
```

Row grid lines and column grid lines start at 1 and end at a value that is 1 greater than the number of rows or columns the grid has. For example, if a grid has 5 rows, the grid row lines range from 1 to 6.

The value for grid-row-start should be the row at which you want the grid item to begin. The value for grid-row-end should be one greater than the row at which you want the grid item to end. An element that covers rows 2, 3, and 4 should have these declarations: grid-row-start: 2 and grid-row-end: 5.

We can use the property grid-row as shorthand for grid-row-start and grid-row-end. Same rule applies for grid-column-start, grid-column-end - grid-column.

```
.item {
  grid-row: 4 / 6; /*start / end */
}
```

In this case, the starting row goes before the "/" and the ending row goes after it.

When an item spans multiple rows or columns using these properties, it will also include the grid-gap if any exists. For example, if an item spans two rows of height 100 pixels and there is a ten-pixel grid-gap, then the item will have a total height of 210 pixels.

When using these properties, we can use the keyword span to start or end a column or row relative to its other end.

```
.item {
  grid-column: 4 / span 2;
}
```

This is telling the item element to begin in column four and take up two columns of space. So item would occupy columns four and five. It produces the same result as the following code blocks:

```
.item {
  grid-column: 4 / 6;
}

.item {
  grid-column-start: 4;
  grid-column-end: span 2;
}
.item {
  grid-column-start: span 2;
  grid-column-end: 6;
}
```

We can refactor even more using the property grid-area. This property will set the starting and ending positions for both the rows and columns of an item.

```
.item {
  grid-area: 2 / 3 / 4 / span 5;
/* grid-row-start / grid-column-start / grid-row-end / grid-column-end */
}
```

grid-area takes four values separated by slashes. The order is important!
In the above example, the item will occupy rows two and three and columns three through eight.

1.8. CSS Cheat Sheet

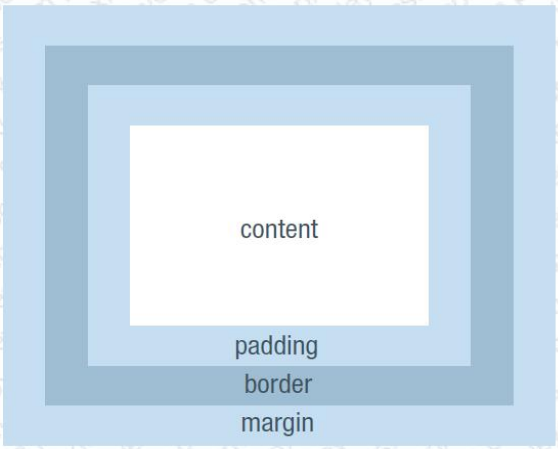
CSS Cheat Sheet

brought to you by pxleyes.com

Selectors

div	all DIV tags
div, span	all DIV tags and all SPAN tags
div span	all SPAN tags inside DIVs
#content	element with ID “content”
.box	all elements with CLASS “box”
ul#box	UL tag with ID “box”
span.box	all SPAN tags with CLASS “box”
*	all elements
#box *	all elements inside #box
a:link, a:active, a:visited	links in normal state, in clicked state, and in visited state
a:hover	link with mouse over it
div > span	all SPANs one-level deep in a DIV

Box Model



Positioning

position	places elements on screen, e.g. absolute, fixed, relative
float	stacks elements horizontally in a particular direction, e.g. left
top, left, right, bottom	specifies the offsets used in absolute, fixed, and relative positions, e.g. top:10px;left:10px
display	sets how the element is placed in the doc flow, e.g. block, inline, none
z-index	sets the stacking order of elements, e.g. z-index of 1 is below z-index of 2
overflow	sets what happens to content outside of container, e.g. auto, hidden

Text

font-family	font used, e.g. Helvetica, Arial
font-size	text size, e.g. 60px, 3em
color	text color, e.g. #000, #abcdef
font-weight	how bold the text is, e.g. bold
font-style	what style the text is, e.g. italic
text-decoration	sets a variety of effects on text, e.g. underline, overline, none
text-align	how text is aligned, e.g. center
line-height	spacing between lines, e.g. 2em
letter-spacing	spacing between letters, e.g. 5px
text-indent	indent of the first line, e.g. 2em
text-transform	applies formatting to text, e.g. upper-case, lowercase, capitalize
vertical-align	align relative to baseline, e.g. text-top

Borders and Lists

border	sets border style for all borders, in the format: border: (solid, dashed, dotted, double) (width) (color), e.g. border: solid 1px #000
border-top	sets border style for a specific
border-bottom	border (same property syntax used
border-left	for padding and margin, e.g.
border-right	margin-left)
list-style-type	sets style of bullets, e.g. square
list-style-position	sets how text wraps when bulleted, e.g. outside, inside
list-style-image	sets an image for a bullet, e.g. list-style-image:url(bullet.png)

Everything Else

background	sets background of an element, in the format: background: (color) (image) (repeat) (position), e.g. background: #000 url(bg.png) repeat-x top left
cursor	sets shape of cursor, e.g. pointer
outline	a border drawn around an element that doesn’t affect the box model
border-collapse	sets how borders within tables behave, e.g. collapse
clear	sets on what side a new line starts in relation to nearby floated elements, e.g. left, right, both

Always write <!doctype html> in your files!

CSS 3, Media Queries Cheat Sheet

Desktop - Tablet – Phone

```
<!DOCTYPE HTML>
<html lang="en-US">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, minimum-scale=1.0, maximum-scale=1.0" />
   <title>test</title>
   <style type="text/css">
      h3 {text-align:center;}

      /* PRINT VERSION */
      @media print {
         h3:after {content: ' - PRINT'; display: inline;}
      }
      /* PHONE VERSION */
      @media only screen and (min-width: 320px) {
         h3:after {content: ' - PHONE'; display: inline;}
      }
      /* TABLET VERSION */
      @media only screen and (min-width: 768px) {
         h3:after {content: ' - TABLET'; display: inline;}
      }
      /* DESKTOP VERSION */
      @media only screen and (min-width: 980px) {
         h3:after {content: ' - DESKTOP'; display: inline;}
      }
   </style>
</head>
<body>
   <h3>TEST</h3>
</body>
</html>
```

Animation Chaet Sheet

http://www.justinaguilar.com/animations/index.html

2. Atom shortcuts

Cut, Copy, Paste	Cut: Ctrl + X Copy: Ctrl + C Paste: Ctrl + V
Undo, Redo	Undo: Ctrl + Z Redo: Ctrl + ⇧ + Z or Ctrl + Y
Duplicate	Ctrl + ⇧ + D with or without text highlighted
Select	Select Word: Ctrl + D Select Line: Ctrl + L Select All: Ctrl + A Select Every Instance: Alt + F3
Move Cursor	Move Cursor to Previous Word: Ctrl + ← Move Cursor to Next Word: Ctrl + → Move Cursor to Start of Line: Home Move Cursor to End of Line: End Move Cursor to Start of Document: Ctrl + Home Move Cursor to End of Document: Ctrl + End
Jump	Jump to Opening/Closing Parentheses, Brackets, Braces: Ctrl + M Jump to Definition: Ctrl + R
Indent, Unindent	Indent: Tab or Ctrl +] Unindent: ⇧ + Tab or Ctrl + [
Toggle Comment	Ctrl + /
Tab Management	New Tab: Ctrl + N Switch Tab Left: Ctrl + PgUp Switch Tab Right: Ctrl + PgDn Close Tab: Ctrl + W Reopen Tab: Ctrl + ⇧ + T
Quick Open	Ctrl + P
Find	Ctrl + F

3. javaScript intro

JavaScript is a widely used web-based programming language that powers the dynamic behavior on most websites.

3.1. Console

Before you learn about data types and methods, you need to know how to print values to the console. The console is a tool that developers use to record the output of their JavaScript programs.

The console.log() command is used to print, or log, text to the console. Consider the following example:

```
console.log("Hello!");
```

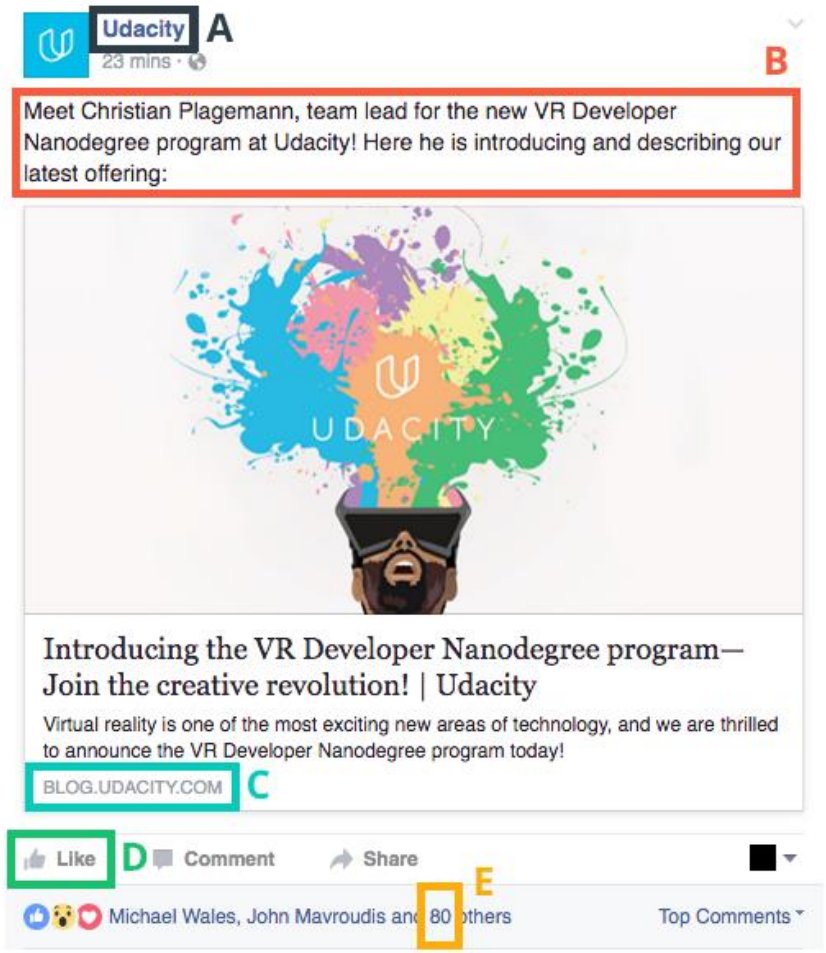
3.2. Data types & Variables

3.2.1. Data types

Below are examples of four *primitive data types* that lay the foundation for all JavaScript programs. Primitive data types, as their name implies, are the simplest built-in forms of data.

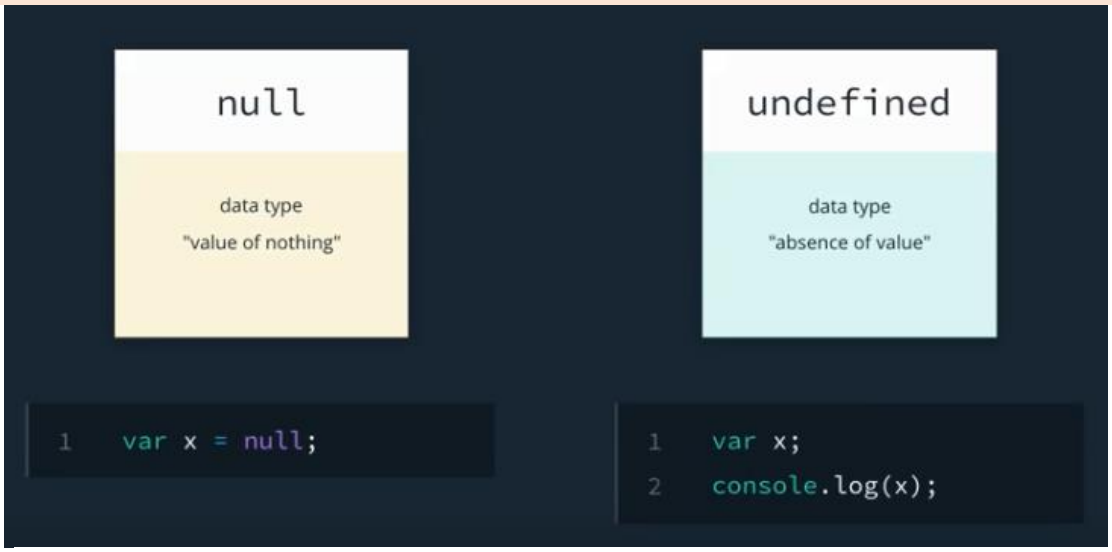
```
console.log('New York City');
console.log(40.7);
console.log(true);
console.log(null);
```

- *Strings* — Any grouping of keyboard characters (letters, spaces, numbers, or symbols) surrounded by single quotes ('Hello') or double quotes ("World!"). In the example above, 'New York City' is a string.
- *Numbers* — Any number, including numbers with decimals: 4, 1516, .002, 23.42. In the example above, 40.7 is a number.
- *Booleans* — Either true or false, with no quotations. In the example above, true is a boolean.
- *Null* — Can only be null. It represents the absence of value.



LETTER	TYPE
A - Udacity	String
B - Meet Christian Plagemann, team lead for the new VR Developer Nanodegree program at Udacity! Here he is introducing and describing our latest offering	String
C - blog.udacity.com	String
D - Whether something was "liked" or not.	Boolean
E - 80	Number

Null, Undefined, and NaN



NaN stands for "Not-A-Number" and it's often returned indicating an error with number operations. For instance, if you wrote some code that performed a math calculation, and the calculation failed to produce a valid number, NaN might be returned.

```
// calculating the square root of a negative number will return NaN
Math.sqrt(-10)

// trying to divide a string by 5 will return NaN
"hello"/5
```

What happens if you create a variable, but don't assign it a value?

JavaScript creates space for this variable in memory and sets it to undefined. *Undefined* is the fifth and final primitive data type. JavaScript assigns the undefined data type to variables that are not assigned a value.

```
let whatAmI;
```

In the example above, we created the variable whatAmI without any value assigned to it. JavaScript creates the variable and sets it equal to the value undefined.

3.2.2. Operators

JavaScript supports the following math *operators*: Add: +, Subtract: -, Multiply: *, Divide: /

Mathematical assignment operators that make it easy to calculate a new value and assign it to the same variable without writing the variable twice. See examples of these operators below.

```
let x = 4; x += 2; // x equals 6
let y = 4; y -= 2; // y equals 2
let z = 4; z *= 2; // z equals 8
let r = 4; r++; // r equals 5
let t = 4; t--; // t equals 3
```

In the example above, operators are used to calculate a new value and assign it to the same variable. Let's consider the first three and last two operators separately:

1. The first three operators (`+=`, `-=`, and `*=`) perform the mathematical operation of the first operator (`+`, `-`, or `*`) using the number on the right, then assign the new value to the variable.
2. The last two operators are the increment (`++`) and decrement (`--`) operators. These operators are responsible for increasing and decreasing a number variable by one, respectively.

Operator	Meaning
<	Less than
>	Greater than
<=	Less than or Equal to
>=	Greater than or Equal to
==	Equal to
!=	Not Equal to

3.2.3. Strict equality

In JavaScript it's better to use **strict equality** to see if numbers, strings, or booleans, etc. are identical in *type* and *value* without doing the *type conversion first (refer to the topic below)* . To perform a strict comparison, simply add an additional equals sign `=` to the end of the `==` and `!=` operators.

```
"1" === 1
```

Returns: false

This returns false because the string `"1"` is not the same type *and* value as the number `1`.

```
0 === false
```

Returns: false

This returns false because the number `0` is not the same type *and* value as the boolean `false`.

3.2.4. Implicit type coercion

JavaScript is known as a *loosely typed language*. Basically, this means that when you're writing JavaScript code, you do not need to specify data types. Instead, when your code is interpreted by the JavaScript engine it will automatically be converted into the "appropriate" data type. This is called *implicit type coercion* and you've already seen examples like

*A **strongly typed language** is a programming language that is more likely to generate errors if data does not closely match an expected type. Because JavaScript is loosely typed, you don't need to specify data types; however, this can lead to errors that are hard to diagnose due to implicit type coercion.*

this before when you tried to concatenate strings with numbers.

```
"julia" + 1
```

Returns: "julia1"

In this example, JavaScript takes the string "julia" and adds the number 1 to it resulting in the string "julia1". In other programming languages, this code probably would have returned an error, but in JavaScript the number 1 is converted into the string "1" and then is concatenated to the string "julia".
It's behavior like this which makes JavaScript unique from other programming languages, but it can lead to some quirky behavior when doing operations and comparisons on mixed data types.

Example of strongly typed programming language code

```
int count = 1;
string name = "Julia";
double num = 1.2932;
float price = 2.99;
```

Equivalent code in JavaScript

```
// equivalent code in JavaScript
var count = 1;
var name = "Julia";
var num = 1.2932;
var price = 2.99;
```

In the example below, JavaScript takes the string "1", converts it to true, and compares it to the boolean true.

```
"1" == true
```

Returns: true

When you use the == or != operators, JavaScript first converts each value to the same type (if they're not already the same type); this is why it's called "type coercion"! This is often not the behavior you want, and **it's actually considered bad practice to use the == and != operators when comparing values for equality**.

***Comments**

```
// this is a single-line comment

/*
this is
a multi-line
comment
*/
```

3.2.5. Variables

With variables, you no longer need to work with one-time-use data. Storing the value of a string in a variable is like packing it away for later use.

```
var greeting = "Hello";
```

Now, if you want to use "Hello" in a variety of sentences, you don't need to duplicate "Hello" strings. You can just reuse the greeting variable.

```
greeting + " World!";
```

Returns: Hello World!

Programmers use variables to write code that is easy to understand and repurpose.

Imagine you're writing a weather app. Your thermometer outside reports the temperature in Celsius, but your goal is to record the temperature in Fahrenheit.

You write a program that takes a temperature of 15degrees Celsius and calculates the temperature in Fahrenheit.

Once you've done this though, you see the temperature now reads 16 degrees Celsius. To find Fahrenheit again, you'd need to write a whole new program to convert 16 degrees Celsius to Fahrenheit.

That's where variables come in. Variables allow us to assign data to a word and use the word to reference the data. If the data changes (like degrees Celsius) we can replace the variable's value instead of re-writing the program.

Naming conventions

When you create a variable, you write the name of the variable using camelCase (the first word is lowercase, and all following words are uppercase). Also try to use a variable name that accurately, but succinctly describes what the data is about.

```
var totalAfterTax = 53.03; // uses camelCase if the variable name is multiple words
var tip = 8; // uses lowercase if the variable name is one word
```

Not using camelCase for your variables names is not going to necessarily *break* anything in JavaScript. But there are recommended style guides used in all programming languages that help keep code consistent, clean, and easy-to-read. This is especially important when working on larger projects that will be accessed by multiple developers.

Google JavaScript Style Guide

https://google.github.io/styleguide/jsguide.html

Constant variables (cannot be reassigned)

Here is how you declare a *constant variable*:

```
const myName = 'Arya';
console.log(myName);
// Output: Arya
```

Let's consider the example above:

Constant variables, as their name implies, are constant — you cannot assign them a different value.

1. const, short for constant, is a JavaScript *keyword*that creates a new variable with a value that cannot change.
2. myName is the variable's name. Notice that the word has no spaces, and we capitalized the N. Capitalizing in this way is a standard convention in JavaScript called *camelCasing*, because the capital letters look like the humps on a camel's back.
3. = is the *assignment operator*. It assigns the value ('Arya') to the variable (myName).
4. 'Arya' is the *value* assigned (=) to the variable myName.

After the variable is declared, we can print 'Arya' to the console with: console.log(myName).

You can save any data type in a variable. For example, here we save numbers:

```
const myAge = 11;
console.log(myAge);
// Output: 11
```

In the example above, on line 1 the myAge variable is set to 11. Below that, console.log() is used to print 11 to the console.

Let variables (can be reassigned)

```
let meal = 'Enchiladas';
console.log(meal);
meal = 'Tacos';
console.log(meal);
 // output: Enchiladas // output: Tacos
```

In the example above, the let keyword is used to create the meal variable with the string 'Enchiladas' saved to it. On line three, the mealvariable is changed to store the string 'Tacos'.

You may be wondering, when to use const vs let. In general, only use const if the value saved to a variable does not change in your program.

Indexing

Did you know that you can access individual characters in a string? To access an individual character, you can use the character's location in the string, called its **index**. Just put the index of the character inside square brackets (starting with [0] as the first character) immediately after the string. For example:

```
"James"[0];
```

Returns: "J"
or more commonly, you will see it like this, using a variable:

```
var name = "James";
name[0];
```

Returns: "J"

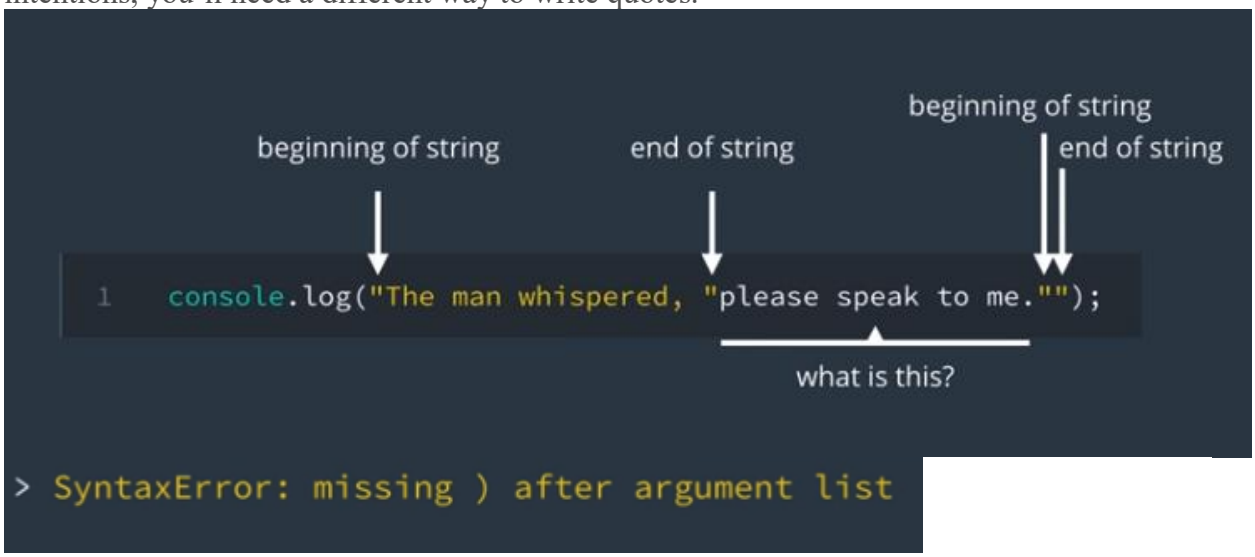
Escaping strings

There are some cases where you might want to create a string that contains more than just numbers and letters. For example, what if you want to use quotes in a string?

```
"The man whispered, "please speak to me.""
```

Uncaught SyntaxError: Unexpected identifier

If you try to use quotes within a string, you will receive a SyntaxError like the one above.
If you want to use quotes *inside a string*, and have JavaScript not misunderstand your intentions, you'll need a different way to write quotes.



Escaping a character tells JavaScript to ignore the character's special meaning and just use the literal value of the character. This is helpful for characters that have special meanings like in our previous example with quotes "…".
Because quotes are used to signify the beginning and end of a string, you can use the backslash character to escape the quotes in order to access the literal quote character.

```
"The man whispered, \"please speak to me.\""
```

Returns: "The man whispered, "please speak to me.""

This guarantees that the JavaScript engine doesn't misinterpret the string and result in an error.

Special characters

Code	Character
\\	\ (backslash)
\"	" (double quote)
\'	' (single quote)
\n	newline
\t	tab

The last two characters listed in the table, newline \n and tab \t, are unique because they add additional **whitespace** to your Strings. A newline character will add a line break and a tab character will advance your line to the next tab stop.

```
"Up up\n\tdown down"
```

Returns:

Up up
    down down

Using special characters in strings

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_types#Using_special_characters_in_strings

Comparing strings

Another way to work with strings is by comparing them. You've seen the comparison operators == and != when you compared numbers for equality. You can also use them with strings! For example, let's compare the string "Yes" to "yes".

```
"Yes" == "yes"
```

Returns: false

When you run this in the console, it returns false. Why is that? "Yes" and "yes" are the same string, right? Well not quite.

Case-sensitive

When you compare strings, case matters. While both string use the same letters (and those letters appear in the same order), the first letter in the first string is a capital Y while the first letter in the second string is a lowercase y.

```
'Y' != 'y'
```

Returns: true

String Interpolation

The JavaScript term for inserting the data saved to a variable into a string is *string interpolation*.

The + operator, known until now as the addition operator, is used to interpolate (insert) a string variable into a string, as follows:

```
let myPet = 'armadillo';
console.log('I own a pet ' + myPet + '.');
// Output: 'I own a pet armadillo.'
```

In the example above, we saved the value 'armadillo' to the myPet variable. On the second line, the + operator is used to combine three strings: I own a pet, the value saved to myPet, and . We log the result of this interpolation to the console as:

```
I own a pet armadillo.
```

In the newest version of JavaScript (ES6) we can insert variables into strings with ease, by doing two things:

1. Instead of using quotes around the string, use backticks (this key is usually located on the top of your keyboard, left of the 1 key).
2. Wrap your variable with ${myVariable}, followed by a sentence. No +s necessary.

ES6 string interpolation is easier than the method you used last exercise. With ES6 interpolation we can insert variables directly into our text.

It looks like this:

```
let myPet = 'armadillo' console.log(`I own a pet ${myPet}.`)
// Output: 'I own a pet armadillo.'
```

In the example above, the backticks (`) wrap the entire string. The variable (myPet) is inserted using ${}. The resulting string is:

```
I own a pet armadillo.
```

3.3. Quizzes and examples

Quiz: Converting Temperatures

```
/*
 * Programming Quiz: Converting Tempatures (2-2)
 *
 * Use the Celsius-to-Fahrenheit formula to set the fahrenheit varible:
 *
 *     F = C x 1.8 + 32
 *
 * Log the fahrenheit variable to the console.
 *
 */

var celsius = 12;
var fahrenheit = celsius * 1.8 + 32;

console.log(fahrenheit);
```

Quiz: All Tied Up

Build a single string that resembles the following joke.

```
Why couldn't the shoes go out and play?
They were all "tied" up!
```

Your joke should take the format of a **question** and **answer**. The first line should be a question and the second line should be an answer.
Hint: *You will need to use special characters to produce the following output.*

```
/*
 * Programming Quiz: All Tied Up (2-5)
 */

var joke = "Why couldn't the shoes go out and play?\nThey were all \"tied\" up!";
console.log(joke);
```

Returns: Why couldn't the shoes go out and play?

They were all "tied" up!

Quiz: Yosa Buson

Build a string using concatenation by combining the lines from this famous haiku poem by Yosa Buson.

```
Blowing from the west
Fallen leaves gather
In the east.
```

Each string should be printed on its own line.

Hint: *You will need to use special characters to produce the following output. For a refresher, feel free to review the previous* ***Escaping Strings*** *lesson in this course.*

```
/*
 * Programming Quiz: Yosa Buson (2-6)
 */

var haiku = "Blowing from the west" + "\nFallen leaves gather\nIn the east.";
console.log(haiku);
```

Returns:

Blowing from the west
Fallen leaves
gather In the east.

Quiz: Out to Dinner

Create a variable called bill and assign it the result of 10.25 + 3.99 + 7.15 (don't perform the calculation yourself, let JavaScript do it!). Next, create a variable called tip and assign it the result of multiplying bill by a 15% tip rate. Finally, add the bill and tip together and store it into a variable called total.
Print the total to the JavaScript console.

Hint: *15% in decimal form is written as* 0.15.

TIP: To print out the total with a dollar sign ($) use string concatenation. To round total up by two decimal points use the toFixed() method. To use toFixed() pass it the number of decimal points you want to use. For example, if total equals 3.9860, then total.toFixed(2) would return 3.99.

```
/*
 * Programming Quiz: Out to Dinner (2-10)
 */

// your code goes here
var bill = 10.25 + 3.99 + 7.15;
var tip = bill * 0.15;
var total = bill + tip;
console.log(total.toFixed(2));
```

Returns: 24.60

Quiz: Mad Libs

Mad Libs is a word game where players have fun substituting words for blanks in a story. For this exercise, use the adjective variables below to fill in the blanks and complete the following message.

```
"The Intro to JavaScript course is _____. James and Julia are so _____. I cannot wait to work through the rest of this _____ content!"
var adjective1 = "amazing";
var adjective2 = "fun";
var adjective3 = "entertaining";
```

Assign the resulting string to a variable called madLib.

```
/*
 * Programming Quiz: MadLibs (2-11)
 *
 * 1. Declare a madLib variable
 * 2. Use the adjective1, adjective2, and adjective3 variables to set the madLib variable to the message:
 *
 * 'The Intro to JavaScript course is amazing. James and Julia are so fun. I cannot wait to work through the rest of this entertaining content!'
 */

var adjective1 = 'amazing';
var adjective2 = 'fun';
var adjective3 = 'entertaining';
var madLib = 'The Intro to JavaScript course is ' + adjective1 + '. James and Julia are so ' + adjective2 + '. I cannot wait to work through the rest of this ' + adjective3 + ' content!';
console.log(madLib);
// your code goes here
```

Returns:

The Intro to JavaScript course is amazing. James and Julia are so fun. I cannot wait to work through the rest of this entertaining content!

Quiz: One Awesome Message

Here are two awesome messages:

> "Hi, my name is Julia. I love cats. In my spare time, I like to play video games."
> "Hi, my name is James. I love baseball. In my spare time, I like to read."

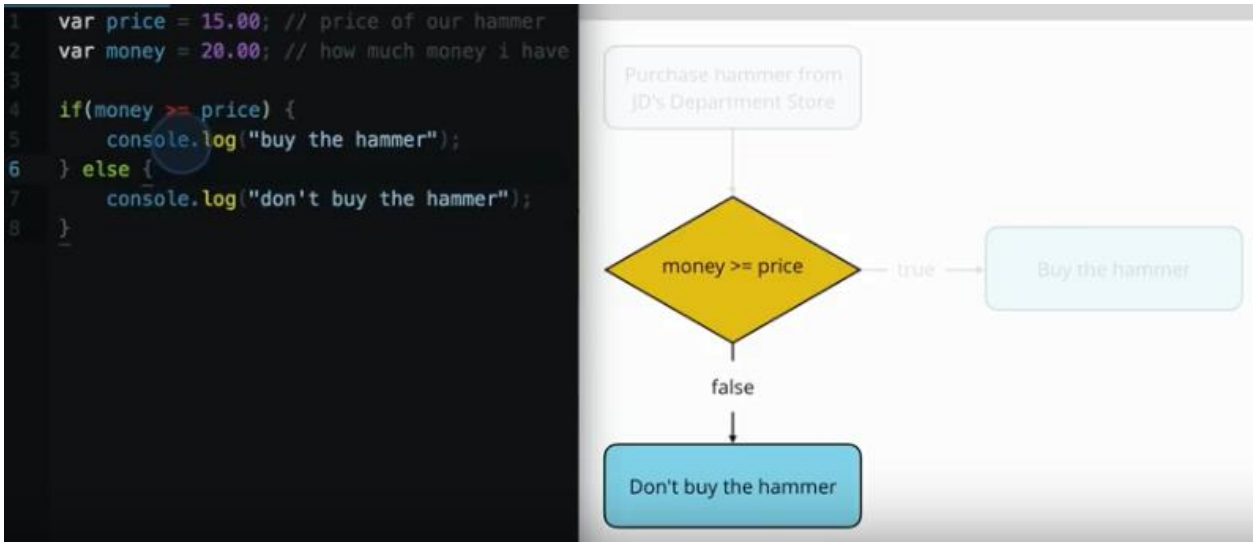
Declare and assign values to three variables for each part of the sentence that changes (firstName, interest, and hobby).
Use your variables and string concatenation to create your own awesome message and store it in an awesomeMessagevariable. Finally, print your awesome message to the JavaScript console.

```
/*
 * Programming Quiz: One Awesome Message (2-12)
 *
 * 1. Create the variables:
 *     - firstName
 *     - interest
 *     - hobby
 * 2. Create a variable named awesomeMessage and, using string concatenation
      and the variables above, create an awesome message.
 * 3. Print the awesomeMessage variable to the console.
 */

/*
 * Notes:
 * - Using the above as an example, firstName would have been assigned to
 *   "Julia", interest to "cats", and hobby to "to play video games".
 * - Be sure to include spaces and periods where necessary!
 */

// Add your code here
var firstName = 'Julia';
var interest = 'books';
var hobby = 'paint';
var awesomeMessage = 'Hi, my name is ' + firstName + '. I love ' + interest + '. In my spare time, I like to ' + hobby + '.';
console.log(awesomeMessage);
```

Returns: Hi, my name is Julia. I love books. In my spare time, I like to paint.

4. javaScript Conditionals

4.1. If...else statements



If...else statements allow you to execute certain pieces of code based on a condition, or set of conditions, being met.

```
if (/* this expression is true */) {
  // run this code
} else {
  // run this code
}
```

This is extremely helpful because it allows you to choose which piece of code you want to run based on the result of an expression. For example,

```
var a = 1;
var b = 2;

if (a > b) {
  console.log("a is greater than b");
} else {
  console.log("a is less than or equal to b");
}
```

Prints: "a is less than or equal to b"

Control flow *statements enable JavaScript programs to make decisions by executing code based on a condition. If a given condition is true, we execute one block of code. If the statement is false, we execute another block of code.*

A couple of important things to notice about if...else statements.
The value inside the if statement is always *converted* to true or false. Depending on the value, the code inside the if statement is run or the code inside the else statement is run, but not both. The code inside the if and else statements are surrounded by **curly braces** {...} to separate the conditions and indicate which code should be run.

TIP: When coding, sometimes you may *only* want to use an if statement. However, if you try to use only an else statement, then you will receive the error SyntaxError: Unexpected

token else. You'll see this error because else statements need an if statement in order to work. You can't have an else statement without first having an if statement.

Example:

```
let needsCoffee = true;
if (needsCoffee === true) {
console.log('Finding coffee');
} else {
console.log('Keep on keeping on!');
}
```

1. Lines of code between curly braces are called *blocks*. if/else statements have two code blocks. If the variable needsCoffee is true, the program will run the first block of code. Otherwise, it will run the other block of code.
2. needsCoffee is the *condition* we are checking inside the if's parentheses. Since it is equal to true, our program will run the code between the first opening curly brace { (line 2) and the first closing curly brace }(line 4). It will ignore the else { ... } part. In this case, we'd see Finding coffee log to the console.
3. If needsCoffee were false, only the console.log() statement in the else block would be executed.

if/else statements are how programs can process yes/no questions programmatically.

Tue or false values (Truthy and Falsy)

There are data types that are not booleans. Let's explore the concepts of true and false in variables that contain other data types, including strings and numbers.

In JavaScript, all variables and conditions have a *truthy*or *falsy* value.
value.

Falsy values

A value is **falsy** if it converts to false when evaluated in a boolean context. For example, an empty String "" is falsy because, "" evaluates to false. You already know if...else statements, so let's use them to test the truthy-ness of "".

```
if ("") {
  console.log("the value is truthy");
} else {
  console.log("the value is falsy");
}
```

Returns: "the value is falsy"

```
let variableOne = 'I Exist!';
if (variableOne) { // This code will run because variableOne contains a truthy value.
} else { // This code will not run because the first block ran.
}
```

In the first line of the program above, a variable is created and set. The value of this variable is a string rather than a boolean. How does this program determine which code block to run?

The second line of this program checks a condition if (variableOne). In the previous exercise, we checked if a variable was equal to true or false. By only writing the name of the variable as the condition, we are checking the *truthiness* of the variableOne. In this case, variableOne contains a truthy value.

If we changed `if (variableOne)` to say `if (variableTwo)`, that condition would evaluate to falsy because we have not created a variable called `variableTwo` in this program. In other words, `variableOne` is truthy and `variableTwo` is falsy.

All variables that have been created and set are truthy (and will evaluate to true if they are the condition of a control flow statement) unless they contain one of the seven values listed below:

- `false`
- `0` and `-0`
- `""` and `''` (empty strings)
- `null`
- `undefined`
- `NaN` (Not a Number)
- `document.all` (something you will rarely encounter)

There is an important distinction between a variable's value and its truthiness: `variableOne`'s value is 'I exist' because that is the data saved to the variable. `variableOne` is truthy because it exists and does not contain any of the seven falsy values listed above.

In programming, we often evaluate whether or not an expression is true or truthy. Conveniently, JavaScript provides a shorthand notation for this.

```
let isRaining = true;
 if (isRaining) {
console.log('Carry an umbrella!');
} else {
console.log('Enjoy the sun!');
}
```

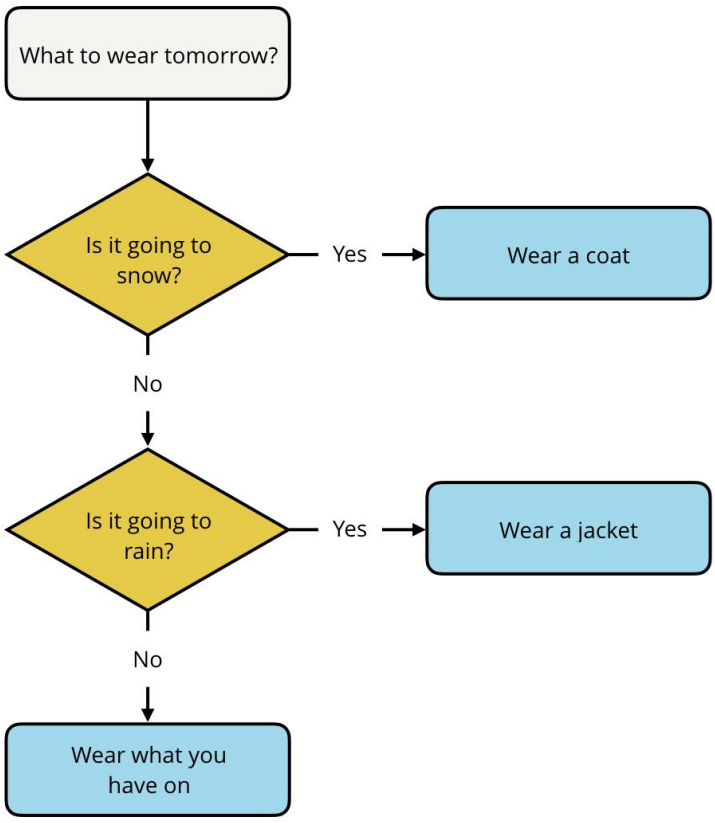
In the example above, the condition is simply `if (isRaining)`. In JavaScript, this is evaluating whether `isRaining` is truthy. If you read the code out loud to yourself, it sounds like a simple sentence: "If it's raining, carry an umbrella. Else, enjoy the sun!"

JavaScript provides an operator for swapping the truthiness and falsiness of values - the exclamation point (`!`). We can use this in conditional statements as shorthand to check if the value of a variable evaluates to false rather than true.

```
let isPhoneCharged = true;
 if (!isPhoneCharged) {
console.log('Plug in your phone!');
} else {
console.log('No need to charge!');
}
```

In the example above, the program checks if `isPhoneCharged` evaluates to `false`. Because `isPhoneCharged` is `true`, the second block of code will execute.

Else if statements



In some situations, two conditionals aren't enough. Consider the following situation.

You're trying to decide what to wear tomorrow. If it is going to snow, then you'll want to wear a coat. If it's not going to snow and it's going to rain, then you'll want to wear a jacket. And if it's not going to snow or rain, then you'll just wear what you have on.

In JavaScript, you can represent this secondary check by using an extra if statement called an **else if statement**.

```
var weather = "sunny";

if (weather === "snow") {
  console.log("Bring a coat.");
} else if (weather === "rain") {
  console.log("Bring a rain jacket.");
} else {
  console.log("Wear what you have on.");
}
```

Prints: Wear what you have on.

By adding the extra else if statement, you're adding an extra conditional statement.

If it's not going to snow, then the code will jump to the else if statement to see if it's going to rain. If it's not going to rain, then the code will jump to the else statement.

The else statement essentially acts as the "default" condition in case all the other if statements are false.

Examples:

```
let stopLight = 'green';
if (stopLight === 'red') {
console.log('Stop');
} else if (stopLight === 'yellow') {
console.log('Slow down');
} else if (stopLight === 'green') {
console.log('Go!');
} else { console.log('Caution, unknown!');
}
```

1. We created a variable named stopLight that is assigned to the string green.

2. Then, there's an if/else statement with multiple conditions, using else if. else if allows us to check multiple values of the stopLight variable and output different things based on its color.

3. The block ends with the singular else we have seen before. The else is a catch-all for any other situation. For instance, if the stopLight was blinking blue, the last else would catch it and return a default message.

```
var money = 100.50;
var price = 100.50;

if (money > price) {
  console.log("You paid extra, here's your change.");
} else if (money === price) {
  console.log("You paid the exact amount, have a nice day!");
} else {
  console.log("That's not enough, you still owe me money.");
}
```

Prints: You paid the exact amount, have a nice day!

```
var runner = "Kendyll";
var position = 2;
var medal;

if(position === 1) {
  medal = "gold";
} else if(position === 2) {
  medal = "silver";
} else if(position === 3) {
  medal = "bronze";
} else {
  medal = "pat on the back";
}

console.log(runner + " received a " + medal + " medal.");
```

Prints: Kendyll received a silver medal.

Logical operators

We can translate certain thoughts into JavaScript code such as, "Are these things equal?" with ===, or, "Is one thing greater than another thing?" with >.

In English, sometimes we say "both of these things" or "either one of these things." Let's translate those phrases into JavaScript with special operators called *logical operators*.

1. To say "both must be true," we use &&.
2. To say "either can be true," we use ||.

For example:

```
if (stopLight === 'green' && pedestrians === false) {
console.log('Go!');
} else {
console.log('Stop');
}
```

1. In the example above, we make sure that the stopLight is 'green' and (&&) there are no pedestrians before we log Go!.
2. If either of those conditions is false, we log Stop.

Just like the operators we learned previously, these logical operators will return either true or false.

These logical operators are helpful when writing if/else statements since they let us make sure multiple variables are true or false. We can combine these operators with all of the ones we have learned throughout this lesson.

Here's the logical expression used to represent Julia's weekend plans:

```
var colt = "not busy";
var weather = "nice";

if (colt === "not busy" && weather === "nice") {
  console.log("go to the park");
}
```

Prints: "go to the park"

Notice the && in the code above.
The && symbol is the logical AND operator, and it is used to combine two logical expressions into one larger logical expression. If **both** smaller expressions are *true*, then the entire expression evaluates to *true*. If **either one** of the smaller expressions is *false*, then the whole logical expression is *false*.
Another way to think about it is when the && operator is placed between the two statements, the code literally reads, "if Colt is not busy *AND* the weather is nice, then go to the park".

Logical expressions

Logical expressions are similar to mathematical expressions, except logical expressions evaluate to either *true* or *false*.

```
11 != 12
```

Returns: true

TIP: Logical expressions are evaluated from left to right. Similar to mathematical expressions, logical expressions can also use parentheses to signify parts of the expression that should be evaluated first.

You've already seen logical expressions when you write comparisons. A comparison is just a simple logical expression.

Similar to mathematical expressions that use +, -, *, / and %, there are logical operators &&, || and !that you can use to create more complex logical expressions.

Logical operators

Logical operators can be used in conjunction with boolean values (true and false) to create complex logical expressions.
By combining two boolean values together with a logical operator, you create a *logical expression* that returns another boolean value.

Here's a table describing the different logical operators:

Operator	Meaning	Example	How it works
&&	Logical AND	value1 && value2	Returns true if **both** value1 **and** value2 evaluate to true.
||	Logical OR	value1 || value2	Returns true if **either** value1 **or** value2 (**or even both!**) evaluates to true.
!	Logical NOT	!value1	Returns the **opposite** of value1. If value1 is true, then !value1 is false.

By using logical operators, you can create more complex conditionals like Julia's weekend example.

```
var colt = "not busy";
var weather = "nice";

if (colt === "not busy") {
  if (weather === "nice") {
    console.log("go to the park");
  }
}
```

```
var colt = "not busy";
var weather = "nice";

if (colt === "not busy" && weather === "nice") {
  console.log("go to the park");
}
```

4.2. Ternary Operator

Sometimes, you might find yourself with the following type of conditional.

```
var isGoing = true;
var color;

if (isGoing) {
  color = "green";
} else {
  color = "red";
}

console.log(color);
```

Prints: "green"

In this example, the variable color is being assigned to either "green" or "red" based on the value of isGoing. This code works, but it's a rather lengthy way for assigning a value to a variable. Thankfully, in JavaScript there's another way.

TIP: Using if(isGoing) is the same as using if(isGoing === true).
Alternatively, using if(!isGoing) is the same as using if(isGoing === false).

The **ternary operator** provides you with a shortcut alternative for writing lengthy if...else statements.

```
conditional ? (if condition is true) : (if condition is false)
```

To use the ternary operator, first provide a conditional statement on the left-side of the ?. Then, between the ? and :write the code that would run if the condition is true and on the right-hand side of the : write the code that would run if the condition is false. For example, you can rewrite the example code above as:

```
var isGoing = true;
var color = isGoing ? "green" : "red";
console.log(color);
```

Prints: "green"

This code not only replaces the conditional, but it also handles the variable assignment for color.

If you breakdown the code, the condition isGoing is placed on the left side of the ?. Then, the first expression, after the ?, is what will be run if the condition is *true* and the second expression after the, :, is what will be run if the condition is *false*.
Example:

```
isNightTime ? console.log('Turn on the lights!') : console.log('Turn off the lights!');
```

The code in the example above will operate exactly as the code from the previous example. Let's break this example into its parts:

1. isNightTime ? — the conditional statement followed by a question mark. This checks if isNightTime is truthy.
2. console.log ('Turn on the lights!') — this code will be executed if the condition is truthy.
3. : — a colon separates the two different blocks of code that can be executed.
4. console.log('Turn off the lights!'); — this code will be executed if the condition is falsy.

4.3. Switch Statement

If you find yourself repeating else if statements in your code, where each condition is based on the same value, then it might be time to use a switch statement.

```
if (option === 1) {
  console.log("You selected option 1.");
} else if (option === 2) {
  console.log("You selected option 2.");
} else if (option === 3) {
  console.log("You selected option 3.");
} else if (option === 4) {
  console.log("You selected option 4.");
} else if (option === 5) {
  console.log("You selected option 5.");
} else if (option === 6) {
  console.log("You selected option 6.");
}
```

A **switch statement** is another way to chain multiple else if statements that are based on the same value **without using conditional statements**. Instead, you just *switch* which piece of code is executed based on a value.

```
switch (option) {
  case 1:
    console.log("You selected option 1.");
  case 2:
    console.log("You selected option 2.");
  case 3:
    console.log("You selected option 3.");
  case 4:
    console.log("You selected option 4.");
  case 5:
    console.log("You selected option 5.");
  case 6:
```

```
  console.log("You selected option 6.");
}
```

Here, each else if statement (option === [value]) has been replaced with a case clause (case: [value]) and those clauses have been wrapped inside the switch statement.
When the switch statement first evaluates, it looks for the first case clause whose expression evaluates to the same value as the result of the expression passed to the switch statement. Then, it transfers control to that case clause, executing the associated statements.
So, if you set option equal to 3...

```
var option = 3;

switch (option) {
  ...
}
```

Prints:
You selected option 3.
You selected option 4.
You selected option 5.
You selected option 6.
...then the switch statement prints out options 3, 4, 5, and 6.

But that's not exactly like the original if...else code at the top? So what's missing?

Break statement

The **break statement** can be used to terminate a switch statement and transfer control to the code following the terminated statement. By adding a break to each case clause, you fix the issue of the switch statement *falling-through* to other case clauses.

```
var option = 3;

switch (option) {
  case 1:
    console.log("You selected option 1.");
    break;
  case 2:
    console.log("You selected option 2.");
    break;
  case 3:
    console.log("You selected option 3.");
    break;
  case 4:
    console.log("You selected option 4.");
    break;
  case 5:
    console.log("You selected option 5.");
    break;
  case 6:
    console.log("You selected option 6.");
    break; // technically, not needed
}
```

Prints: You selected option 3.

Falling through

In some situations, you might want to leverage the "falling-through" behavior of switch statements to your advantage.

For example, when your code follows a hierarchical-type structure.

```
var tier = "nsfw deck";
var output = "You'll receive "

switch (tier) {
  case "deck of legends":
    output += "a custom card, ";
  case "collector's deck":
    output += "a signed version of the Exploding Kittens deck, ";
  case "nsfw deck":
    output += "one copy of the NSFW (Not Safe for Work) Exploding Kittens card game and
";
  default:
    output += "one copy of the Exploding Kittens card game.";
}

console.log(output);
```

Prints: You'll receive one copy of the NSFW (Not Safe for Work) Exploding Kittens card game and one copy of the Exploding Kittens card game.

In this example, based on the **successful Exploding Kittens Kickstarter campaign** (a hilarious card game created by Elan Lee), each successive tier builds on the next by adding more to the output. Without any break statements in the code, after the switch statement jumps to the `"nsfw deck"`, it continues to fall-through until reaching the end of the switch statement.

Also, notice the `default` case.

```
var tier = "none";
var output = "You'll receive ";

switch (tier) {
  ...
  default:
    output += "one copy of the Exploding Kittens card game.";
}

console.log(output);
```

Prints: You'll receive one copy of the Exploding Kittens card game.

You can add a `default` case to a switch statement and it will be executed when none of the values match the value of the switch expression.

4.4. Quizzes and examples

Quiz: Even or Odd

Write an if...else statement that:

- prints "even" if the number is an even number
- prints "odd" if the number is an odd number

Hint: Use the % (**modulo**) operator to determine if a number is even or odd. The modulo operater takes two numbers and returns the remainder when the first number is divided by the second one:

```
console.log(12 % 3);
console.log(10 % 4);
```

Modulo checks if the number is odd or even. Example:
10 % 5 = 2 rest 0 12 % 3 = 4 rest 0
10 % 4 = 2 rest 2 12 % 5 = 2 rest 2

Result:
0
2

The answer for 12 % 3 is 0 because twelve divided by three has no remainder. 10 % 4 is 2 because ten divided by 4 has a remainder of two.

Make sure to test your code with different values. For example:

- If number equals 1, then odd should be printed to the console.
- If number equals 12, then even should be printed to the console.

```
/*
 * Programming Quiz: Even or Odd (3-2)
 *
 * Write an if...else statement that prints `even` if the
 * number is even and prints `odd` if the number is odd.
 *
 * Note - make sure to print only the string "even" or the string "odd"
 */

// change the value of `number` to test your if...else statement
var number = 7;

if (number % 2 === 0) {
   console.log("even");
} else {
   console.log("odd");
}
```

Returns: odd

Quiz: Musical Groups

Musical groups have special names based on the number of people in the group.

Directions:
Write a series of conditional statements that:

- prints "not a group" if musicians is less than or equal to 0
- prints "solo" if musicians is equal to 1
- prints "duet" if musicians is equal to 2
- prints "trio" if musicians is equal to 3
- prints "quartet" if musicians is equal to 4
- prints "this is a large group" if musicians is greater than 4

TIP: Make sure to test your code with different values. For example,

If musicians equals 3, then trio should be printed to the console.
If musicians equals 20, then this is a large group should be printed to the console.
If musicians equals -1, then not a group should be printed to the console.

```
/*
 * Programming Quiz: Musical Groups (3-3)
 */
// change the value of `musicians` to test your conditional statements
var musicians = 8;
// your code goes here
if (musicians <= 0) {
    console.log("not a group");
} else if (musicians === 1){
    console.log("solo");
} else if (musicians === 2){
    console.log("duet");
} else if (musicians === 3){
    console.log("trio");
} else if (musicians === 4){
    console.log("quartet");
} else if (musicians > 4){
    console.log("this is a large group");
} else {
}
```

Returns: this is a large group

Quiz: Murder Mystery

For this quiz, you're going to help solve a fictitious **murder mystery** that happened here at Udacity! A murder mystery is a game typically played at parties wherein one of the partygoers is secretly, and unknowingly, playing a murderer, and the other attendees must determine who among them is the criminal. It's a classic case of **whodunnit**.
Since this might be your first time playing a murder mystery, we've simplified things quite a bit to make it easier. Here's what we know! In this murder mystery there are:

- **four rooms**: the ballroom, gallery, billiards room, and dining room,
- **four weapons**: poison, a trophy, a pool stick, and a knife,
- and **four suspects**: Mr. Parkes, Ms. Van Cleve, Mrs. Sparr, and Mr. Kalehoff.

We also know that each weapon *corresponds* to a particular room, so...

- the poison belongs to the ballroom,
- the trophy belongs to the gallery,
- the pool stick belongs to the billiards room,
- and the knife belongs to the dining room.

And we know that each suspect was located in a specific room at the time of the murder.

- Mr. Parkes was located in the dining room.
- Ms. Van Cleve was located in the gallery.
- Mrs. Sparr was located in the billiards room.
- Mr. Kalehoff was located in the ballroom.

To help solve this mystery, write a combination of conditional statements that:

1. sets the value of weapon based on the room and
2. sets the value of solved to true if the value of room matches the suspect's room

Afterwards, print the following to the console if the mystery was solved:

> _____ did it in the _____ with the _____!

Fill in the blanks with the name of the suspect, the room, and the weapon. For example,

> Mr. Parkes did it in the dining room with the knife!

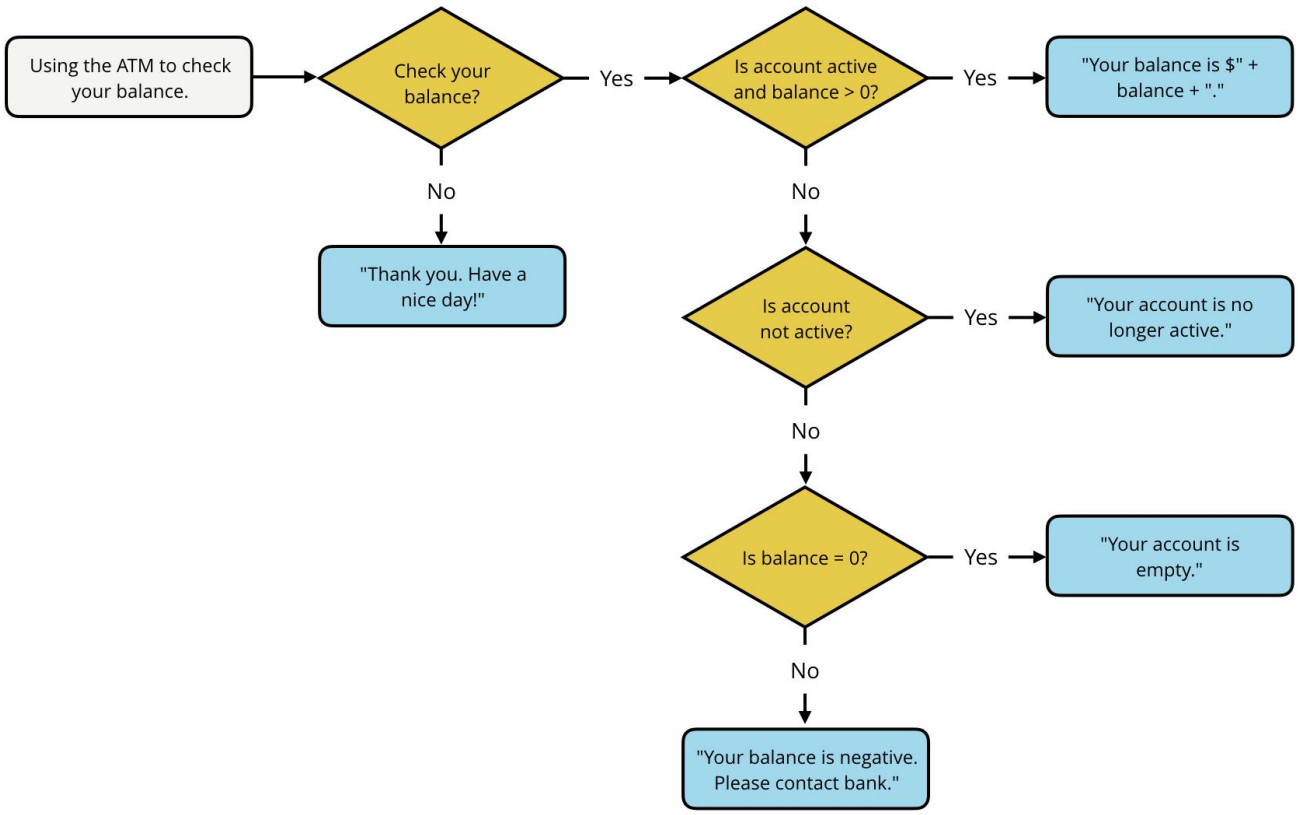
TIP: Make sure to test your code with different values.
For example,
If room equals gallery and suspect equals Ms. Van Cleve, then Ms. Van Cleve did it in the gallery with the trophy! should be printed to the console.

```
/*
 * Programming Quiz: Murder Mystery (3-4)
 */

// change the value of `room` and `suspect` to test your code
var room = "ballroom";
var suspect = "Mr. Kalehoff";

var weapon = "";

if (room === "dining room" && suspect === "Mr. Parkes") {
   var weapon = "knife";
   var solved = true;
} else if (room === "ballroom" && suspect === "Mr. Kalehoff") {
    var weapon = "poison";
    var solved = true;
} else if (room === "gallery" && suspect === "Ms. Van Cleve") {
    var weapon = "trophy";
    var solved = true;
} else {
    var weapon = "pool stick";
    var solved = true;
}

if (solved) {
        console.log(suspect+" did it in the "+room+" with the "+weapon+"!");
}
```

Returns: Mr. Kalehoff did it in the ballroom with the poison!

Quiz: Checking your Balance

Using the flowchart below, write the code to represent checking your balance at the ATM. The yellow diamonds represent conditional statements and the blue rectangles with rounded corners represent what should be printed to the console.



Use the following variables in your solution:

- balance - the account balance
- isActive - if account is active
- checkBalance - if you want to check balance

Hint: *The variable balance could be a value less than, greater than, or equal to 0. The variables isActive and checkBalance are booleans that can be set to true or false.*

TIP: To print out the account balance with decimal points (i.e. 325.00), use the .toFixed() method and pass it the number of decimal points you want to use. For example, balance.toFixed(2) returns 325.00.

TIP: Make sure to test your code with different values. For example,

If checkBalance equals true and isActive equals false, then Your account is no longer active. should be printed to the console.

```
/*
 * Programming Quiz - Checking Your Balance (3-5)
 */

// change the values of `balance`, `checkBalance`, and `isActive` to test your code
var balance = 0.00;
var checkBalance = true;
var isActive = true;

// your code goes here

if (!checkBalance) {
    console.log("Thank you. Have a nice day!");
} else if (isActive && balance>0 && checkBalance) {
   console.log("Your balance is $"+balance.toFixed(2)+".");
} else if (!isActive && checkBalance) {
    console.log("Your account is no longer active.");
} else if (balance === 0 && checkBalance && isActive) {
    console.log("Your account is empty.");
} else {
    console.log("Your balance is negative. Please contact bank.");
}
```

Prints: `Your account is empty.`

Quiz: Ice Cream

Ice cream is one of the most versatile desserts on the planet because it can be done up so many different ways. Using logical operators, write a series of complex logical expressions that prints **only if** the following conditions are **true**:

- if flavor is set to vanilla or chocolate and
- if vessel is set to cone or bowl and
- if toppings is set to sprinkles or peanuts

If the above conditions are true, then print out:

> I'd like two scoops of _____ ice cream in a _____ with _____.

Fill in the blanks with the flavor of the ice cream, vessel, and toppings. For example,

> I'd like two scoops of vanilla ice cream in a cone with peanuts.

TIP: Make sure to test your code with different values. For example,

If flavor equals "chocolate", vessel equals "cone" and toppings equals "sprinkles", then "I'd like two scoops of chocolate ice cream in a cone with sprinkles." should be printed to the console.

```
 * Programming Quiz: Ice Cream (3-6)
 * Write a single if statement that logs out the message:
 *
 * "I'd like two scoops of _____ ice cream in a _____ with _____."
 *
 * ...only if:
 *   - flavor is "vanilla" or "chocolate"
 *   - vessel is "cone" or "bowl"
 *   - toppings is "sprinkles" or "peanuts"
 *
 * We're only testing the if statement and your boolean operators.
 * It's okay if the output string doesn't match exactly.
 */

// change the values of `flavor`, `vessel`, and `toppings` to test your code
var flavor = "vanilla";
var vessel = "bowl";
var toppings = "peanuts";

// Add your code here
if ((flavor==="vanilla"||flavor==="chocolate")&&(vessel==="cone"||vessel==="bowl")&&(toppings==="sprinkles"||toppings==="peanuts")) {
   console.log("I'd like two scoops of "+flavor+" ice cream in a "+vessel+" with "+toppings+".");
}
```

Prints: `I'd like two scoops of vanilla ice cream in a bowl with peanuts.`

Quiz: What do I Wear?

If you're like me, finding the right size t-shirt can sometimes be a challenge. What size am I? What's the difference between S (small), M (medium), and L (large)? I usually wear L, but what if I need an XL (extra large)?

Thankfully, our friends at **Teespring** have got us covered because they've created a sizing chart to make things a lot easier.

SIZE	WIDTH	LENGTH	SLEEVE
S	18"	28"	8.13"
M	20"	29"	8.38"
L	22"	30"	8.63"
XL	24"	31"	8.88"
2XL	26"	33"	9.63"
3XL	28"	34"	10.13"

T-Shirt Sizing Chart

TIP: Make sure to test your code with different values. For example,

If shirtWidth equals 19, shirtLength equals 28 and shirtSleeve equals 8.21, then S should be printed to the console.
If shirtWidth equals 26, shirtLength equals 33 and shirtSleeve equals 9.63, then 2XL should be printed to the console.
If shirtWidth equals 18, shirtLength equals 29 and shirtSleeve equals 8.47, then N/A should be printed to the console.

Directions:
Use the sizing chart above, create a series of logical expressions that prints the **size** of a t-shirt based on the measurements of shirtWidth, shirtLength, and shirtSleeve. Valid sizes include S, M, L, XL, 2XL, and 3XL.
For example, if...

```
var shirtWidth = 23; // size L (large)
var shirtLength = 30; // size L (large)
var shirtSleeve = 8.71; // size L (large)
```

Then print L to the console.

Hint: *You will need to compare a range of values when checking for shirtWidth, shirtLength, and shirtSleeve. For example, if the shirt's **width** is at least **20"**, but no more than **22"**, then the t-shirt should be **medium (M)** — as long as the other values for the shirt's **length** and **sleeve** measurements match up.*

If shirtWidth, shirtLength, and shirtSleeve don't fit within the range of acceptable values for a specific **size**, then print N/A to the console.

For example, if...

```
var shirtWidth = 18; // size S (small)
var shirtLength = 29; // size M (medium)
var shirtSleeve = 8.47; // size M (medium)
```

Then print N/A to the console because the measurements don't all match up with one particular size.

```
 * Programming Quiz: What do I Wear? (3-7)
 */
// change the values of `shirtWidth`, `shirtLength`, and `shirtSleeve` to test your code
var shirtWidth = 18;
var shirtLength = 29;
var shirtSleeve = 8.47;

// your code goes here
if((shirtWidth>=18&&shirtWidth<20)&&(shirtLength>=28&&shirtLength<29)&&(shirtSleeve>=8.13&&shirtSleeve<8.38)){
   console.log("S");
} else
if((shirtWidth>=20&&shirtWidth<22)&&(shirtLength>=29&&shirtLength<30)&&(shirtSleeve>=8.38&&shirtSleeve<8.63)){
   console.log("M");
} else
if((shirtWidth>=22&&shirtWidth<24)&&(shirtLength>=30&&shirtLength<31)&&(shirtSleeve>=8.63&&shirtSleeve<8.88)){
   console.log("L");
} else
if((shirtWidth>=24&&shirtWidth<26)&&(shirtLength>=31&&shirtLength<33)&&(shirtSleeve>=8.88&&shirtSleeve<9.63)){
   console.log("XL");
} else
if((shirtWidth>=26&&shirtWidth<28)&&(shirtLength>=33&&shirtLength<34)&&(shirtSleeve>=9.63&&shirtSleeve<10.13)){
   console.log("2XL");
} else if((shirtWidth>=28)&&(shirtLength>=34)&&(shirtSleeve>=10.13)){
   console.log("3XL");
} else {
   console.log("N/A");
}
```

Prints: L

Quiz: Navigating the Food Chain

From the smallest of creatures to the largest of animals, inevitably every living, breathing thing must ingest other organisms to survive. This means that all animals will fall within one of the three consumer-based categories based on the types of food that they eat.

- Animals that eat only plants are called **herbivores**
- Animals that eat only other animals are called **carnivores**
- Animals that eat both plants and animals are called **omnivores**

Directions:

Write a series of ternary statements that sets the variable `category` equal to:

- `"herbivore"` if an animal eats plants
- `"carnivore"` if an animal eats animals
- `"omnivore"` if an animal eats plants and animals
- `undefined` if an animal doesn't eat plants or animals

Use the `eatsPlants` and `eatsAnimals` variables to test your code.

TIP: Make sure to test your code with different values. For example,

If `eatsPlants` equals `true` and `eatsAnimals` equals `false`, then `herbivore`should be printed to the console.

```
 * Programming Quiz - Navigating the Food Chain (3-8)
 *
 * Use a series of ternary operator to set the category to one of the following:
 *   - "herbivore" if an animal eats plants
 *   - "carnivore" if an animal eats animals
 *   - "omnivore" if an animal eats plants and animals
 *   - undefined if an animal doesn't eat plants or animals
 *
 * Notes
 *   - use the variables `eatsPlants` and `eatsAnimals` in your ternary expressions
 *   - `if` statements aren't allowed ;-)
 */
// change the values of `eatsPlants` and `eatsAnimals` to test your code
var eatsPlants = true;
var eatsAnimals = false;

var category = eatsPlants ? (eatsAnimals ? "omnivore" : "herbivore") : (eatsAnimals ?
"carnivore" : undefined);

console.log(category);
```

Returns: `herbivore`

Quiz: Back to School

In 2015, the U.S. Bureau of Labor Statistics **conducted research** to reveal how average salary is directly related to the number of years spent in school. In their findings, they found that people with:

- no high school diploma earned an average of **$25,636/year**,
- a high school diploma earned an average of **$35,256/year**,
- an Associate's degree earned an average of **$41,496/year**,
- a Bachelor's degree earned an average of **$59,124/year**,
- a Master's degree earned an average of **$69,732/year**,
- a Professional degree earned an average of **$89,960/year**,
- and a Doctoral degree earned an average of **$84,396/year**.

Directions:
Write a switch statement to set the average salary of a person based on their type of completed education.
Afterwards, print the following to the console.

> In 2015, a person with _____ earned an average of _____/year.

Fill in the blanks with the type of education and the expected average salary. Make sure to use correct grammar in your printed statement. For help, refer to the findings above.

> In 2015, a person with a Bachelor's degree earned an average of $59,124/year.

TIP: To print out the average salary with commas (i.e. 59,124), use the toLocaleString() method and pass it the locale "en-US". For example, salary.toLocaleString("en-US").

TIP: Make sure to test your code with different values. For example, If education equals "an Associate's degree", then In 2015, a person with an Associate's degree earned an average of $41,496/year. should be printed to the console.

```
/*
 * Programming Quiz: Back to School (3-9)
 */

// change the value of `education` to test your code
var education = "a Doctoral degree";

// set the value of this based on a person's education
var salary;

// your code goes here
switch (education) {
  case "no high school diploma":
    salary = 25636;
    break;
  case "a high school diploma":
    salary = 35256;
    break;
  case "an Associate's degree":
    salary = 41496;
    break;
  case "a Bachelor's degree":
    salary = 59124;
    break;
  case "a Master's degree":
    salary = 69732;
    break;
  case "a Professional degree":
    salary = 89960;
    break;
  case "a Doctoral degree":
    salary = 84396;
    break;
}

console.log("In 2015, a person with "+education+" earned an average of
$"+salary.toLocaleString("en-US")+"/year.");
```

Returns: `In 2015, a person with a Doctoral degree earned an average of $84,396/year.`

5. JavaScript Loops

5.1. While loops


```
var x = 1;
while (x <= 10000) {
    console.log(x + " mississippi!");
    x = x + 1;
}
```

As long as condition is true the loop will print out the number and increment the value of x by 1.

Three main pieces of information that any loop should have are:

1. **When to start:** The code that sets up the loop — defining the starting value of a variable for instance.
2. **When to stop:** The logical condition to test whether the loop should continue.
3. **How to get to the next item:** The incrementing or decrementing step — for example, `x = x * 3` or `x = x - 1`

Here's a basic while loop example that includes all three parts.

```
var start = 0; // when to start
while (start < 10) { // when to stop
  console.log(start);
  start = start + 2; // how to get to the next item
}
```

Prints:

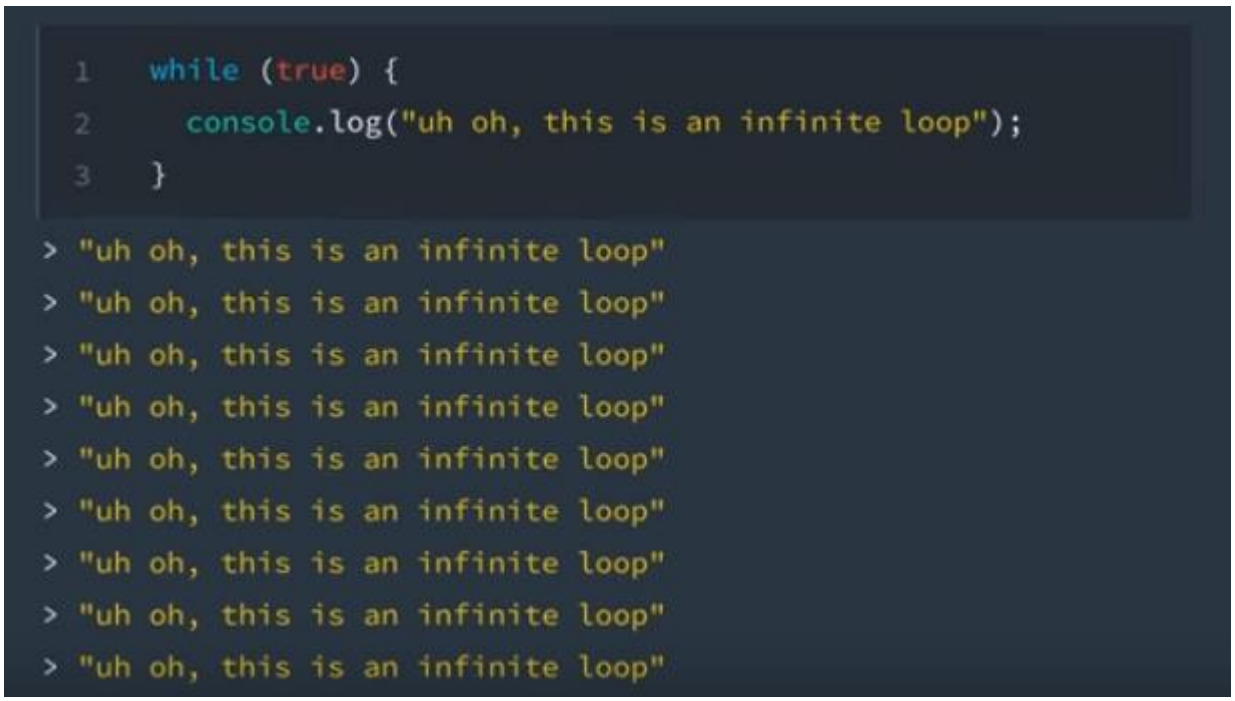
0
2
4
6
8

If a loop is missing any of these three things, then you might find yourself in trouble. For instance, a missing stop condition can result in a loop that never ends!

```
Don't run this code!
while (true) {
  console.log("true is never false, so I will never stop!");
}
```

If you did try to run that code in the console, you probably crashed your browser tab.

WARNING: You're probably reading this because you didn't heed our warnings about running that infinite loop in the console. If your browser tab has crashed or has become frozen/unresponsive, there are a couple ways to fix this. If you are using Firefox, the browser will popup a notification about your script being unresponsive, and will give you the option to kill the script (do that). If you're using Chrome, go to the taskbar and select Window > Task Manager. You can end the process for the particular tab you ran the script in through the task manager. **If you're not using Firefox or Chrome, download Firefox or Chrome ;)**.



Here's an example where a loop is missing how to get to the next item; the variable x is never incremented. x will remain 0 throughout the program, so the loop will never end.

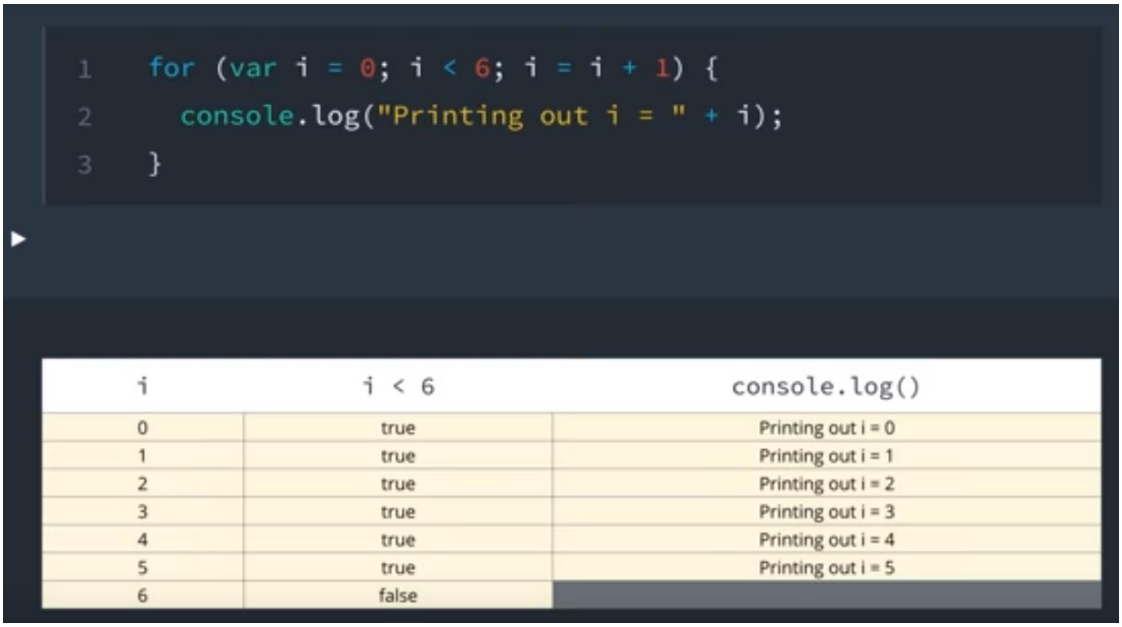
Don't run this code!

```
var x = 0;

while (x < 1) {
  console.log('Oops! x is never incremented from 0, so i will ALWAYS be less than 1');
}
```

This code will also crash your browser tab, so we don't recommend running it.

5.2. For Loops

For loops are the most common type of loop in JavaScript.



The for loop explicitly forces you to define the start point, stop point, and each step of the loop. In fact, you'll get an Uncaught SyntaxError: Unexpected token) if you leave out any of the three required pieces.

```
for ( start; stop; step ) {
  // do this thing
}
```

Here's an example of a for loop that prints out the values from 0 to 5. Notice the semicolons separating the different statements of the for loop: var i = 0; i < 6; i = i + 1

```
for (var i = 0; i < 6; i = i + 1) {
  console.log("Printing out i = " + i);
}
```

Prints:

Printing out i = 0
Printing out i = 1
Printing out i = 2
Printing out i = 3
Printing out i = 4
Printing out i = 5

Instead of writing out the same code over and over, let's make the computer loop through our array for us. We can do this with for loops.

The syntax looks like this:

```
let animals = ["Grizzly Bear", "Sloth", "Sea Lion"];

for (let animalIndex = 0; animalIndex < animals.length; animalIndex++) {
console.log(animals[animalIndex]);
 }
```

The output would be the following:

```
Grizzly Bear
Sloth
Sea Lion
```

Since this syntax is a little complicated, let's break it into four parts:

1. Within the `for` loop's parentheses, the *start condition* is `let animalIndex = 0`, which means the loop will start counting at `0`. `animalIndex` is called an *iterator variable*and it is a good practice to give this variable a descriptive name.
2. The *stop condition* is `animalIndex < animals.length`, which means the loop will run as long as `animalIndex` is less than the length of the animals array. When `animalIndex` is equal to the length of the animals array, the loop will stop executing.
3. The *iterator* is `animalIndex++`. This means that after each loop, `animalIndex` will increase by 1.
4. Finally, the code block is inside of the `{` and `}` curly braces. The block will execute each loop until the program reaches the stop condition.

The secret to loops is that `animalIndex`, the variable we created inside the `for` loop's parentheses, is equal to a number. To be more clear, the first loop, `animalIndex` will equal `0`, the second loop, `animalIndex` will equal `1`, and the third loop, `animalIndex` will equal `2`.

Loops make it possible to write `animals[0]`, `animals[1]`, `animals[2]` programmatically instead of by hand. We can write a `for` loop and replace the hard-coded number with the variable `animalIndex`, like this: `animals[animalIndex]`.

for Loops, Backwards

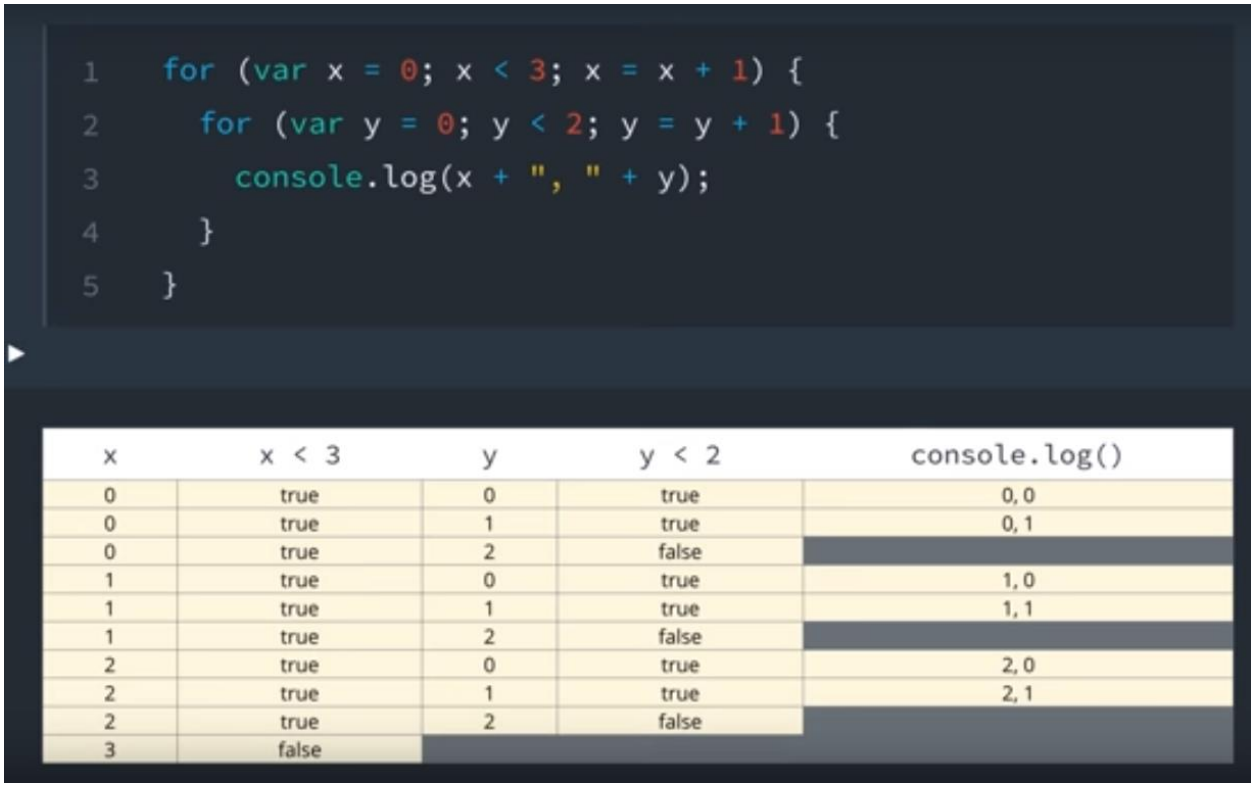
If we can make a `for` loop run forwards through an array, can we make it run backward through one? Of course!

We can make our loop run backward by modifying the start, stop, and iterator conditions.

To do this, we'll need to edit the code between the `for`loop's parentheses:

1. The start condition should set `vacationSpotIndex` to the length of the array.
2. The loop should stop running when `vacationSpotIndex` is less than `0`.
3. The iterator should subtract `1` each time, which is the purpose of `vacationSpotIndex--`.

5.3. Nested Loops



```
for (var x = 0; x < 3; x = x + 1) {
  for (var y = 0; y < 2; y = y + 1) {
    console.log(x + ", " + y);
  }
}
```

x	x < 3	y	y < 2	console.log()
0	true	0	true	0, 0
0	true	1	true	0, 1
0	true	2	false	
1	true	0	true	1, 0
1	true	1	true	1, 1
1	true	2	false	
2	true	0	true	2, 0
2	true	1	true	2, 1
2	true	2	false	
3	false			

Did you know you can also *nest* loops inside of each other? Paste this nested loop in your browser and take a look at what it prints out:

```
for (var x = 0; x < 5; x = x + 1) {
  for (var y = 0; y < 3; y = y + 1) {
    console.log(x + "," + y);
  }
}
```

Prints:

0, 0
0, 1
0, 2
1, 0
1, 1
1, 2
2, 0
2, 1
2, 2
3, 0
3, 1
3, 2
4, 0
4, 1
4, 2

Notice the order that the output is being displayed.

For each value of x in the outer loop, the inner for loop executes completely. The outer loop starts with x = 0, and then the inner loop completes it's cycle with all values of y:

```
x = 0 and y = 0, 1, 2 // corresponds to (0, 0), (0, 1), and (0, 2)
```

Once the inner loop is done iterating over y, then the outer loop continues to the next value, x = 1, and the whole process begins again.

```
x = 0 and y = 0, 1, 2 // (0, 0) (0, 1) and (0, 2)
x = 1 and y = 0, 1, 2 // (1, 0) (1, 1) and (1, 2)
x = 2 and y = 0, 1, 2 // (2, 0) (2, 1) and (2, 2)
etc.
```

5.4. Increment and decrement

```
x++ or ++x // same as x = x + 1
x-- or --x // same as x = x - 1
x += 3 // same as x = x + 3
x -= 6 // same as x = x - 6
x *= 2 // same as x = x * 2
x /= 5 // same as x = x / 5
```

5.5. Quizzes and examples

Quiz: JuliaJames

"Fizzbuzz" is a famous interview question used in programming interviews. It goes something like this:

- Loop through the numbers 1 to 100
- If the number is divisible by 3, print "Fizz"
- If the number is divisible by 5, print "Buzz"
- If the number is divisible by both 3 and 5, print "FizzBuzz"
- If the number is **not** divisible by 3 or 5, print the number
-

TIP: A number x is divisible by a number y if the answer to x / y has a remainder of 0. For example, 10 is divisible by 2 because 10 / 2 = 5 with no remainder. You can check if a number is divisible by another number by checking if x % y === 0.

We're going to have you program your own version of FizzBuzz called "JuliaJames" (yes, imaginative, right?) Keep in mind that in an interview, you would want to write efficient code with very little duplication. We don't want you to worry about that for this question. Just focus on practicing using loops.

Directions:

Write a while loop that:

- Loop through the numbers 1 to 20
- If the number is divisible by 3, print "Julia"
- If the number is divisible by 5, print "James"
- If the number is divisible by 3 and 5, print "JuliaJames"
- If the number is **not** divisible by 3 or 5, print the number

```
/*
 * Programming Quiz: JuliaJames (4-1)
 */

var x = 1;

while (x <= 20) {
   var output = x % 3 === 0 ? (x % 5 === 0 ? "JuliaJames" : "Julia") : (x % 5 === 0 ?
"James" : x);
   console.log(output);
   x = x+1;
}
```

Prints: `1 2 Julia 4 James Julia 7 8 Julia James 11 Julia 13 14 JuliaJames 16 17 Julia 19 James`

Quiz: 99 Bottles of Juice

Write a loop that prints out the following song. Starting at 99, and ending at 1 bottle.

> 99 bottles of juice on the wall! 99 bottles of juice! Take one down, pass it around... 98 bottles of juice on the wall!
> 98 bottles of juice on the wall! 98 bottles of juice! Take one down, pass it around... 97 bottles of juice on the wall!
> ...
> 2 bottles of juice on the wall! 2 bottles of juice! Take one down, pass it around... 1 bottle of juice on the wall!
> 1 bottle of juice on the wall! 1 bottle of juice! Take one down, pass it around... 0 bottles of juice on the wall!

Some Notes:

1. Note the pluralization of the word "bottle" when you go from 2 bottles to 1 bottle.
2. Your text editor may try to autocorrect your ellipses ... to the ellipses *character* … Do not use the ellipses *character* for this quiz.

```
/*
 * Programming Quiz: 99 Bottles of Juice (4-2)
 *
 * Use the following `while` loop to write out the song "99 bottles of juice".
 * Log the your lyrics to the console.
 *
 * Note
 *   - Each line of the lyrics needs to be logged to the same line.
 *    - The pluralization of the word "bottle" changes from "2 bottles" to "1 bottle" to "0 bottles".
 */

var num = 99;

while (num > 0) {
numMinus=(num-1);
bottles1= num===1 ? "bottle" : "bottles";
bottles2= numMinus===1 ? "bottle" : "bottles";
   console.log(num+" "+bottles1+" of juice on the wall! "+num+" "+bottles1+" of juice! Take one down, pass it around... "+numMinus+" "+bottles2+" of juice on the wall!");
   num = num - 1;
}
```

Prints:

```
99 bottles of juice on the wall! 99 bottles of juice! Take one down, pass it around... 98 bottles of juice on the wall!
(…)
1 bottle of juice on the wall! 1 bottle of juice! Take one down, pass it around... 0 bottles of juice on the wall!
```

Quiz: Countdown, Liftoff!

NASA's countdown to launch **includes checkpoints** where NASA engineers complete certain technical tasks. During the final minute, NASA has 6 tasks to complete:

- Orbiter transfers from ground to internal power (T-50 seconds)
- Ground launch sequencer is go for auto sequence start (T-31 seconds)
- Activate launch pad sound suppression system (T-16 seconds)
- Activate main engine hydrogen burnoff system (T-10 seconds)
- Main engine start (T-6 seconds)
- Solid rocket booster ignition and liftoff! (T-0 seconds)

NOTE: "T-50 seconds" read as "T-minus 50 seconds".

Directions:

Write a while loop that counts down from 60 seconds and:

- If there's a task being completed, it prints out the task
- If there is no task being completed, it prints out the time as T-x seconds

Use the task and time descriptions described above.

Your output should look like the following:

```
T-60 seconds
T-59 seconds
T-58 seconds
...
T-51 seconds
Orbiter transfers from ground to internal power
T-49 seconds
...
T-3 seconds
T-2 seconds
T-1 seconds
Solid rocket booster ignition and liftoff!
```

```
 * Programming Quiz: Countdown, Liftoff! (4-3)
 * Using a while loop, print out the countdown output above.
// your code goes here
var t = 60;
while (t<=60 && t>=0){
   textT = (t===50 ? "Orbiter transfers from ground to internal power" :
   (t===31 ? "Ground launch sequencer is go for auto sequence start" :
   (t===16 ? "Activate launch pad sound suppression system" :
   (t===10 ? "Activate main engine hydrogen burnoff system" :
   (t===6 ? "Main engine start" :
   (t===0 ? "Solid rocket booster ignition and liftoff!" : "T-"+t+" seconds"))))));
   console.log(textT);
   t = t-1;
}
```

Quiz: Changing the Loop

Rewrite the following while loop as a for loop:

```
var x = 9;
while (x >= 1) {
  console.log("hello " + x);
  x = x - 1;
}
```

```
/*
 * Programming Quiz: Changing the Loop (4-4)
 */

// rewrite the while loop as a for loop
var x = 9;

for (var x=9; x>=1; x--){
   console.log("hello " + x);
}
```

Prints:

```
hello 9 hello 8 hello 7 hello 6 hello 5 hello 4 hello 3 hello 2 hello 1
```

Quiz: Fix the Error 1

Here is a for loop that's supposed to print the numbers 5 through 9. Fix the error!

```
for (x < 10; x++) {
  console.log(x);
}
```

```
/*
 * Programming Quiz: Fix the Error 1 (4-5)
 */
// fix the for loop
for (var x = 5; x < 10; x++) {
    console.log(x);
}
```

Returns: `5 6 7 8 9`

Quiz: Fix the Error 2

The for loop below has an error. Fix it!

```
for (var k = 0 k < 200 k++) {
  console.log(k);
}
```

```
/*
 * Programming Quiz: Fix the Error 2 (4-6)
 */

// fix the for loop
for (var k = 0; k < 200; k++) {
    console.log(k);
}
```

Returns: `0 1 2 (…) 199`

Quiz: Factorials!

Write a for loop that prints out the factorial of the number 12:

A **factorial** is calculated by multiplying a number by all the numbers below it. For instance, 3! or "3 factorial" is 3 * 2 * 1 = 6

```
3! = 3 * 2 * 1 = 6
4! = 4 * 3 * 2 * 1 = 24
5! = 5 * 4 * 3 * 2 * 1 = 120
```

Save your final answer in a variable called solution and print it to the console.

```
/*
 * Programming Quiz: Factorials (4-7)
 */

// your code goes here
var solution = 12;
for (var k = 11; k >= 1; k--) {
   solution *= k;
}
console.log(solution);
```

Returns: `479001600`

Quiz: Find my Seat

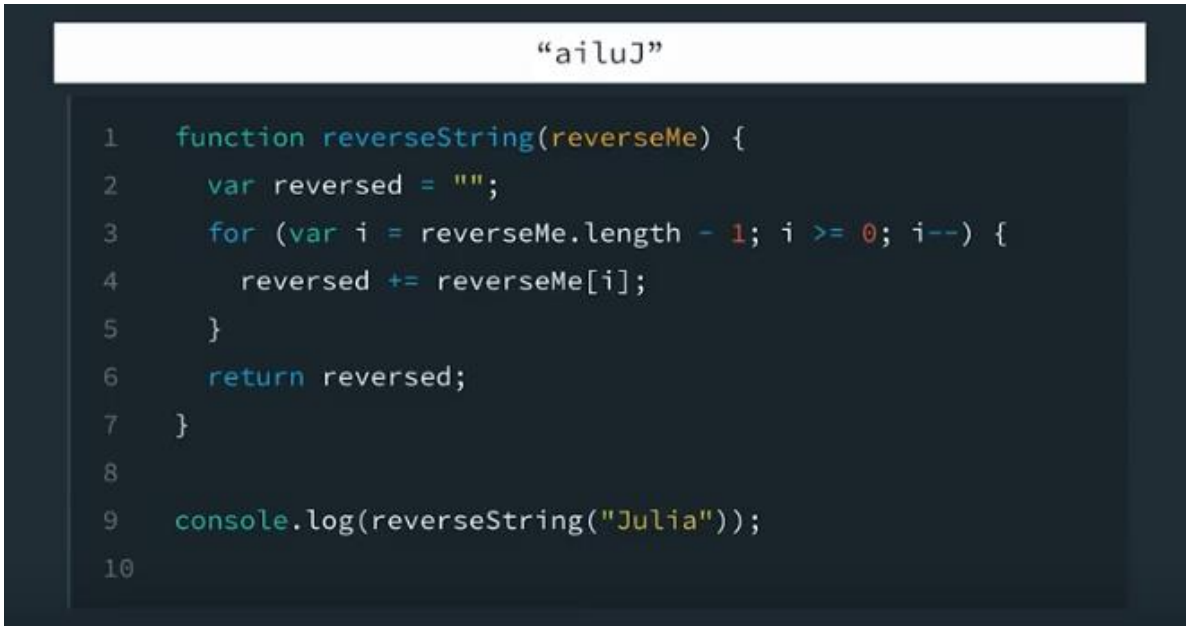
Theater seats often display a row and seat number to help theatergoers find their seats. If there are 26 rows (0 to 25) and 100 seats (0 to 99) in each row, write a nested for loop to print out all of the different seat combinations in the theater.

Example output for row-seat information: *output each row and seat number on a separate line*

```
0-0
0-1
0-2
...
25-97
25-98
25-99
```

```
/*
 * Programming Quiz: Find my Seat (4-8)
 *
 * Write a nested for loop to print out all of the different seat combinations in the theater.
 * The first row-seat combination should be 0-0
 * The last row-seat combination will be 25-99
 *
 * Things to note:
 *  - the row and seat numbers start at 0, not 1
 *  - the highest seat number is 99, not 100
 */

// Write your code here
for (var x = 0; x < 26; x++) {
  for (var y = 0; y < 100; y++) {
    console.log(x + "-" + y);
  }
}
```

6. JavaScript functions



Functions are like recipes. They accept data, perform actions on that data, and return a result. The beauty of functions is that they allow us to write a block of code once, then we can reuse it over and over without rewriting the same code.

How does this code work?

```
let calculatorIsOn = false;
const pressPowerButton = () => {
  if (calculatorIsOn) {
     console.log('Calculator turning off.');
   calculatorIsOn = false;
 } else {
   console.log('Calculator turning on.');
   calculatorIsOn = true;
 }
};

pressPowerButton();
// Output: Calculator turning on.
pressPowerButton();
// Output: Calculator turning off.
```

Let's explore each part in detail.

1. We created a function named `pressPowerButton`.

- `const pressPowerButton` creates a variable with a given name written in *camelCase*.
- The variable is then set equal `=` to a set of parentheses followed by an arrow token `() =>`, indicating the variable stores a function. This syntax is known as *arrow function* syntax.

- Finally, between the curly braces {} is the *function body*, or the JavaScript statements that define the function. This is followed by a semi-colon ;. In JavaScript, any code between curly braces is also known as a *block*.

2. Inside the function body is an if/else statement.

3. On the last few lines, we call the function by writing its name followed by a semi-colon pressPowerButton();. This executes the function, running all code within the function body.

4. We executed the code in the function body twice without having to write the same set of instructions twice. Functions can make code reusable!

6.1. How to declare a function

Functions allow you to package up lines of code that you can use (and often reuse) in your programs.
Sometimes they take **parameters**. The reverseString() function that you saw also had one parameter: the string to be reversed.

```
function reverseString(reverseMe) {
  // code to reverse a string!
}
```

The parameter is listed as a variable after the function name, inside the parentheses. And, if there were multiple parameters, you would just separate them with commas.

```
function doubleGreeting(name, otherName) {
  // code to greet two people!
}
```

But, you can also have functions that don't have any parameters. Instead, they just package up some code and perform some task. In this case, you would just leave the parentheses empty. Take this one for example. Here's a simple function that just prints out "Hello!".

```
// accepts no parameters! parentheses are empty
function sayHello() {
  var message = "Hello!"
  console.log(message);
}
```

If you tried pasting any of the functions above into the JavaScript console, you probably didn't notice much happen. In fact, you probably saw undefined returned back to you. undefined is the default return value on the console when nothing is *explicitly* returned using the special return keyword.

Return statements

In the sayHello() function above, a value is **printed** to the console with console.log, but not explicitly returned with a **return statement**. You can write a return statement by using the return keyword followed by the expression or value that you want to return.

```
// declares the sayHello function
function sayHello() {
  var message = "Hello!"
  return message; // returns value instead of printing it
}
```

How to *run* a function

Now, to get your function to *do something*, you have to **invoke** or **call** the function using the function name, followed by parentheses with any **arguments** that are passed into it. Functions are like machines. You can build the machine, but it won't do anything unless you also turn it on. Here's how you would call the sayHello() function from before, and then use the return value to print to the console:

```
// declares the sayHello function
function sayHello() {
  var message = "Hello!"
  return message; // returns value instead of printing it
}

// function returns "Hello!" and console.log prints the return value
console.log(sayHello());
```

Prints: "Hello!"

Parameters vs. Arguments

At first, it can be a bit tricky to know when something is either a parameter or an argument. The key difference is in where they show up in the code.

A **parameter** is always going to be a *variable* name and appears in the function declaration.

On the other hand, an **argument** is always going to be a *value* (i.e. any of the JavaScript data types - a number, a string, a boolean, etc.) and will always appear in the code when the function is called or invoked.

Parameters are variables in a function definition that represent data we can input into the function.

```
const multiplyByThirteen = (inputNumber) => {
  console.log(inputNumber * 13);
};
multiplyByThirteen(9);
// Output: 117
```

Let's explore how this function works:

1. We add inputNumber within the parentheses () => of the multiplyByThirteen function. inputNumber is a parameter.
2. Inside the multiplyByThirteen() function, we use console.log to print the inputNumber multiplied by 13.
3. When we call the multiplyByThirteen() function on the last line, we set the inputNumber parameter. Here, we set it to 9. Then, inside the function block, 9 is multiplied by 13, resulting in 117 printing to the console.
4. Note on terminology: inputNumber is a parameter, but when we call multiplyByThirteen(9), the 9 is called an *argument*. In other words, arguments are provided when you call a function, and parameters receive arguments as their value. When we set the value 9 as the argument, we *pass* a value to the function.

Parameters let us write logic inside functions that are modified when we call the function. This makes functions more flexible.

We can set as many parameters as we'd like by adding them when we declare the function, separated by commas, like this:

```
const getAverage = (numberOne, numberTwo) => {
  const average = (numberOne + numberTwo) / 2 ;
  console.log(average);
};
getAverage(365, 27);
// Output: 196
```

6.2. Returning with Logging

It's important to understand that **return** and **print** are not the same thing. Printing a value to the JavaScript console only displays a value (that you can view for debugging purposes), but the value it displays can't really be used for anything more than that. For this reason, you should remember to only use `console.log` to test your code in the JavaScript console.

Paste the following function declaration *and* function invocation into the JavaScript console to see the difference between logging (printing) and returning:

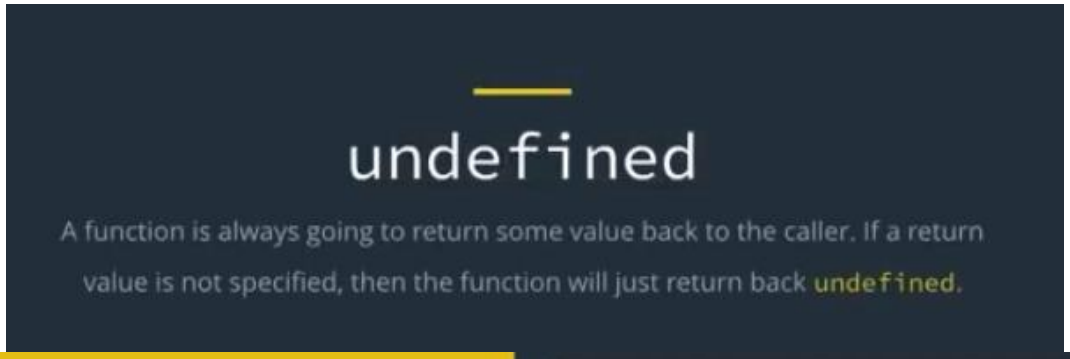
```
function isThisWorking(input) {
  console.log("Printing: isThisWorking was called and " + input + " was passed in as an argument.");
  return "Returning: I am returning this string!";
}

isThisWorking(3);
```

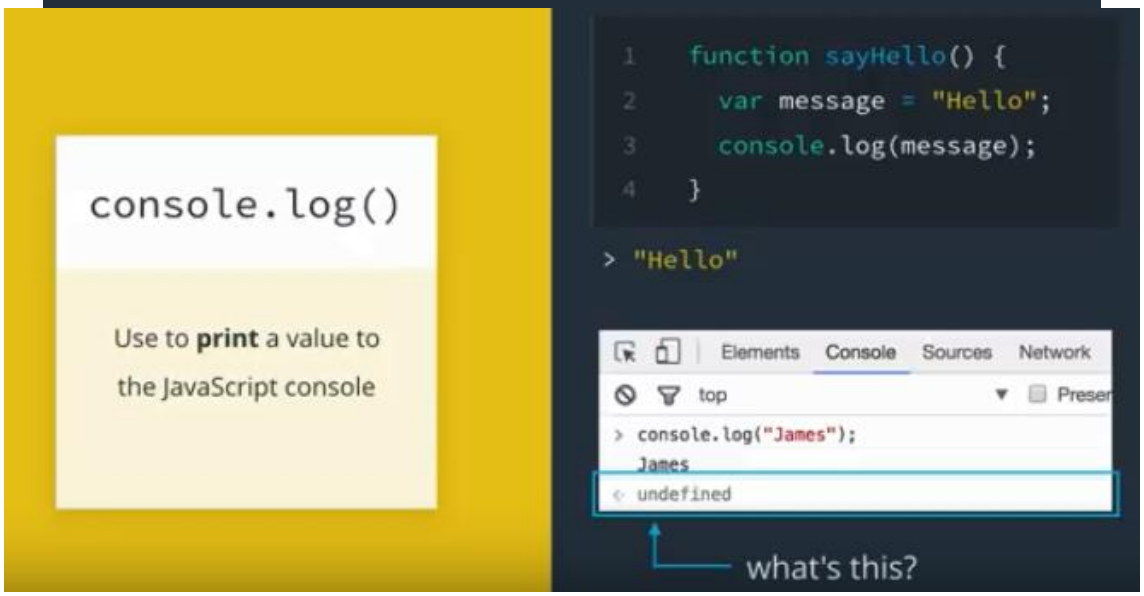
Prints: "Printing: isThisWorking was called and 3 was passed in as an argument"
Returns: "Returning: I am returning this string!"

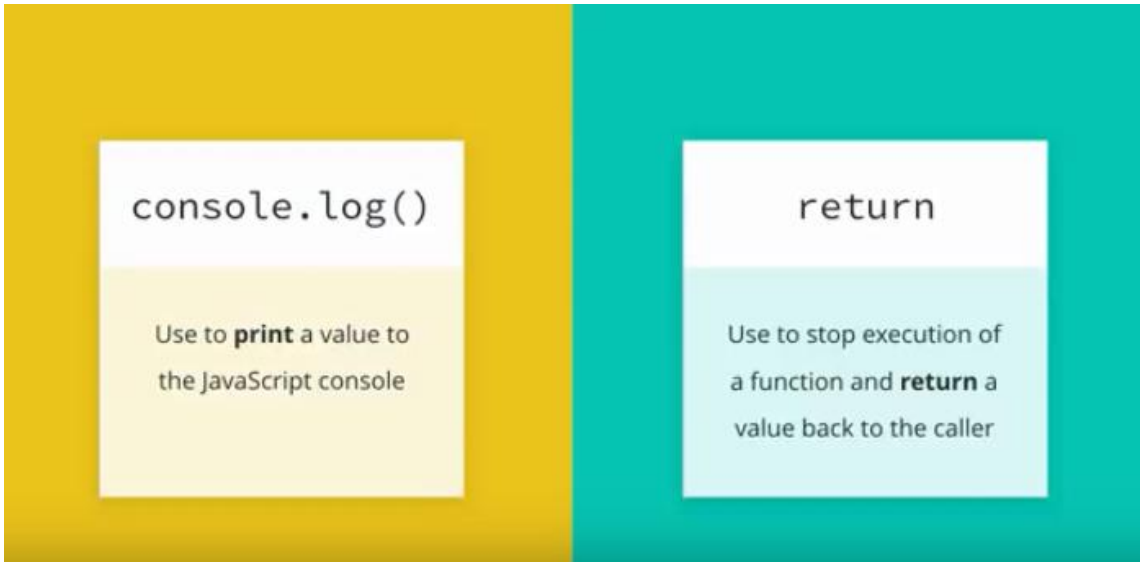
If you don't explicitly define a return value, the function will return `undefined` by default.

```
function isThisWorking(input) {
  console.log("Printing: isThisWorking was called and " + input + " was passed in as an argument.");
}
isThisWorking(3);
```

Prints: "Printing: isThisWorking was called and 3 was passed in as an argument"
Returns: undefined







The purpose of a function is to take some input, perform some task on that input, then return a result.

To return a result, we can use the return keyword. Take a look at our function from the last exercise, now re-written slightly:

```
const getAverage = (numberOne, numberTwo) => {
  const average = (numberOne + numberTwo) / 2;
  return average;
```

```
}

console.log(getAverage(365, 27));
// Output: 196
```

1. Instead of using console.log() inside the getAverage()function, we used the return keyword. return will take the value of the variable and return it.

2. On the last line, we called the getAverage() function inside of a console.log() statement, which outputted the result of 196.

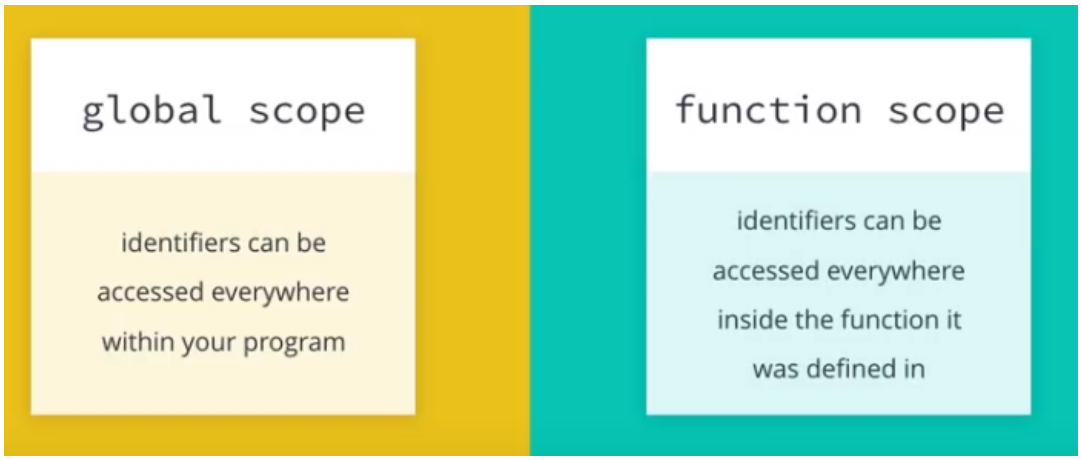
3. This code achieved the same output as before, however now our code is better. Why? If we wanted to use the getAverage() function in another place in our program, we could without printing the result to the console. Using returnis generally a best practice when writing functions, as it makes your code more maintainable and flexible.

A function's return value can be stored in a variable or reused throughout your program as a function argument. Here, we have a function that adds two numbers together, and another function that divides a number by 2. We can find the average of 5 and 7 by using the add() function to add a pair of numbers together, and then by passing the sum of the two numbers add(5, 7) into the function divideByTwo() as an argument.

And finally, we can even store the final answer in a variable called average and use the variable to perform even more calculations in more places!

```
// returns the sum of two numbers
function add(x, y) {
  return x + y;
}


// returns the value of a number divided by 2
function divideByTwo(num) {
  return num / 2;
}


var sum = add(5, 7); // call the "add" function and store the returned value in the "sum" variable
var average = divideByTwo(sum); // call the "divideByTwo" function and store the returned value in the "average" variable
```

6.3. Scope



variable james is defined in global scope

```
1  var james = "I'm looking for this book...";
2
3  function library() {
4        var librarian = "Oh, you'll want to look in the classic
5        literature section, follow me!";
6        function classicLiterature() {
7              var book = "Great Expectations";
8        }
9  }
```

variable librarian - is in function scope

```
1  var james = "I'm looking for this book...";
2
3  function library() {
4        var librarian = "Oh, you'll want to look in the classic
5        literature section, follow me!";
6        function classicLiterature() {
7              var book = "Great Expectations";
8        }
9  }
```

accessible in library function and classiLiterature function

function scope, but visible only in classicLiterature function

```
var james = "I'm looking for this book...";

function library() {
      var librarian = "Oh, you'll want to look in the classic
      literature section, follow me!";
      function classicLiterature() {
            var book = "Great Expectations";
      }
}
```

```
var james = "I'm looking for this book...";
console.log(book);
function library() {
      var librarian = "Oh, you'll want to look in the classic
      literature section, follow me!";
      function classicLiterature() {
            var book = "Great Expectations";
      }
}
```

accessing var book out of function classicLiterature results in the below error

```
Uncaught ReferenceError: book is not defined
```

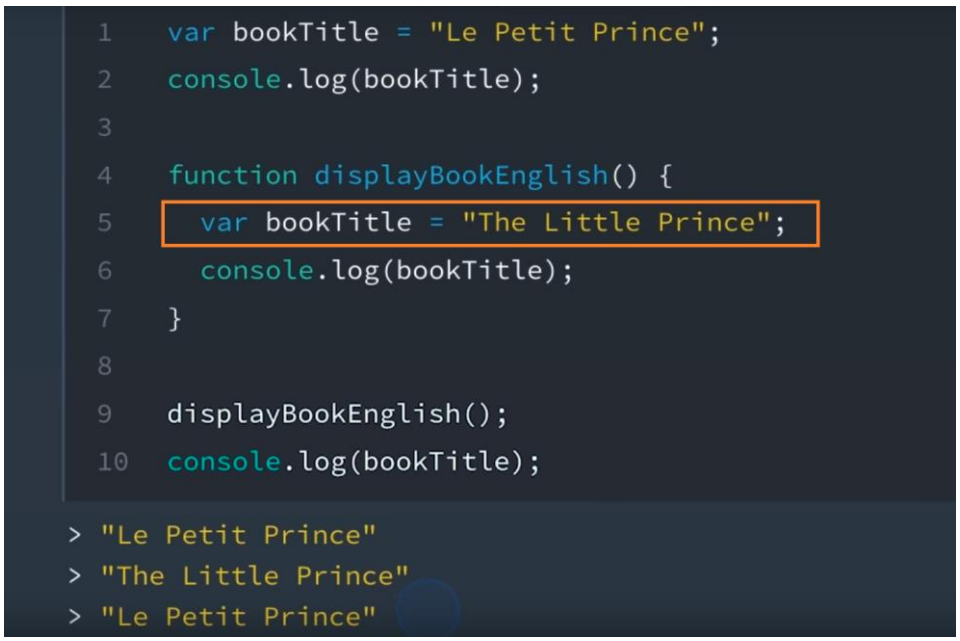
- If an identifier is declared in **global scope**, it's available *everywhere*.
- If an identifier is declared in **function scope**, it's available *in the function* it was declared in (even in functions declared inside the function).
- When trying to access an identifier, the JavaScript Engine will first look in the current function. If it doesn't find anything, it will continue to the next outer function to see if it can find the identifier there. It will keep doing this until it reaches the global scope.
- Global identifiers are a bad idea. They can lead to bad variable names, conflicting variable names, and messy code.

Shadowing

Shadowing, also called scope overriding.

When we reach the fifth line of the code, the bookTitle value has been reassigned to "The Little Prince". That means when we reach the last line of the code in the global scope, bookTitle had been changed, even though we exited the function scope.

To prevent this from happening, we should declare a new variable, instead of re-assigning it:



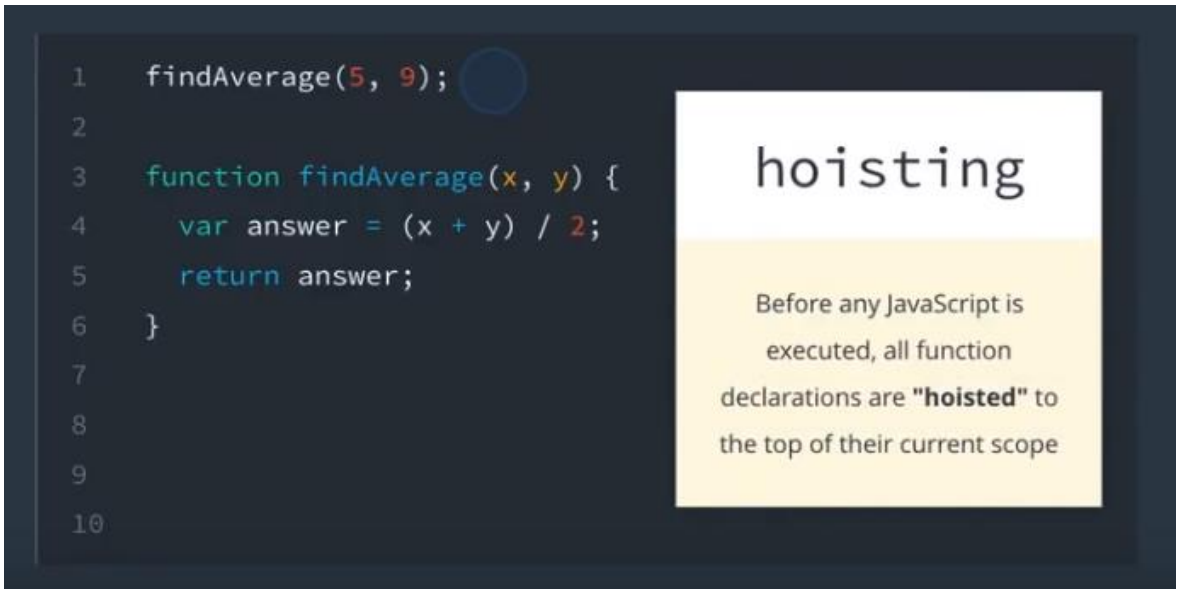
Using global variables

So, you might be wondering:

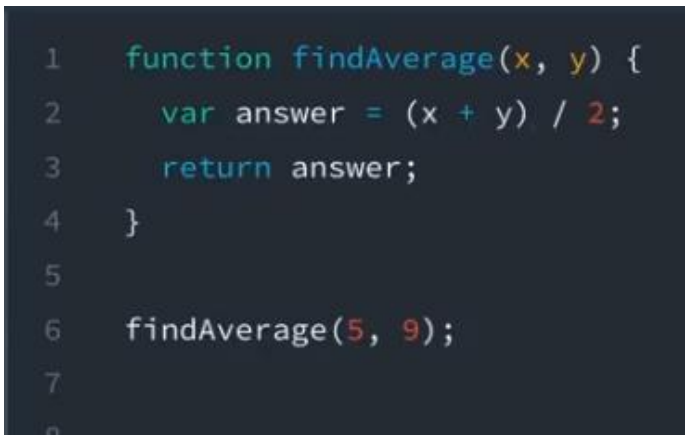
"Why wouldn't I always use global variables? Then, I would never need to use function arguments since ALL my functions would have access to EVERYTHING!"

Well... Global variables might seem like a convenient idea at first, especially when you're writing small scripts and programs, but there are many reasons why you shouldn't use them unless you have to. For instance, global variables can conflict with other global variables of the same name. Once your programs get larger and larger, it'll get harder and harder to keep track and prevent this from happening.

Hoisting



The code you write does not change, but the way it is interpreted as follows and return value of 7.



Another example:

Because the variable greeting I not declared anywhere, therefore, the error is returned:

```
function sayGreeting() {
  console.log(greeting);
}

sayGreeting();
```

```
> Uncaught ReferenceError: greeting is not defined
```

To fix the error, we can declare the variable greeting pretty much anywhere in the function, because we know that it will get hoisted to the top of the function scope.

```
function sayGreeting() {
  console.log(greeting);
  var greeting;
}

sayGreeting();
```

```
> undefined
```

However, if we put value to the variable greeting we will still get the undefined:

```
function sayGreeting() {
  console.log(greeting);
  var greeting = "hello";
}

sayGreeting();
```

```
> undefined
```

This is a bug because of hoisting: the variable greeting is getting hoisted to the top, but the assignment is staying where it is.

```
function sayGreeting() {
  var greeting;
  console.log(greeting);
  greeting = "hello";
}

sayGreeting();
```

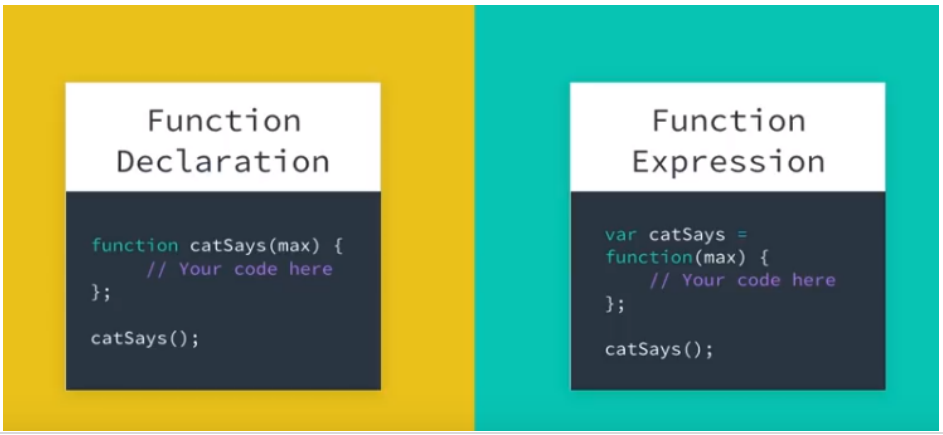
The good practice is to declare functions at the top of the script, and variables at the top of the functions.

```
function sayGreeting() {
  var greeting = "hello";
  console.log(greeting);
}

sayGreeting();
> "hello"
```

- ✓ JavaScript hoists function declarations and variable declarations to the top of the current scope.
- ✓ Variable assignments are not hoisted.
- ✓ Declare functions and variables at the top of your scripts, so the syntax and behavior are consistent with each other.

6.4. Function Expressions

You may store pretty much anything into a variable, even a function: when a function is stored inside a variable t is called function expression.



```
var catSays = function(max) {
  var catMessage = "";
  for (var i = 0; i < max; i++) {
    catMessage += "meow ";
  }
  return catMessage;
};
```

Notice how the function keyword no longer has a name.

```
var catSays = function(max) {
  // code here
};
```

It's an **anonymous function**, a function with no name, and you've stored it in a variable called catSays.
And, if you try accessing the value of the variable catSays, you'll even see the function returned back to you.

```
catSays;
```

Returns:

```
function(max) {
  var catMessage = ""
  for (var i = 0; i < max; i++) {
    catMessage += "meow ";
  }
  return catMessage;
}
```

Function expressions and hoisting

Deciding when to use a function expression and when to use a function declaration can depend on a few things, and you will see some ways to use them in the next section. But, one thing you'll want to be careful of, is hoisting.

All *function declarations are hoisted* and loaded before the script is actually run. *Function expressions are not hoisted*, since they involve variable assignment, and only variable declarations are hoisted. The function expression will not be loaded until the interpreter reaches it in the script.

Named function expressions

You may write an anonymous function expression as in the example above, but you also may create named function expressions. Note, in order to call the function, you should still use the variable name, not the function name:

6.5. Patterns with Function Expressions

Functions as parameters

Being able to store a function in a variable makes it really simple to pass the function into another function. A function that is passed into another function is called a **callback**. Let's say you had a helloCat() function, and you wanted it to return "Hello" followed by a string of "meows" like you had with catSays. Well, rather than redoing all of your hard work, you can make helloCat() accept a callback function, and pass in catSays.


```
// function expression catSays
var catSays = function(max) {
  var catMessage = "";
  for (var i = 0; i < max; i++) {
    catMessage += "meow ";
  }
  return catMessage;
};

// function declaration helloCat accepting a callback
function helloCat(callbackFunc) {
  return "Hello " + callbackFunc(3);
}

// pass in catSays as a callback function
helloCat(catSays);
```

Inline function expressions

A function expression is when a function is assigned to a variable. And, in JavaScript, this can also happen when you pass a function *inline* as an argument to another function. Take the favoriteMovie example for instance:

```
// Function expression that assigns the function displayFavorite
// to the variable favoriteMovie
var favoriteMovie = function displayFavorite(movieName) {
  console.log("My favorite movie is " + movieName);
};

// Function declaration that has two parameters: a function for displaying
// a message, along with a name of a movie
function movies(messageFunction, name) {
  messageFunction(name);
}

// Call the movies function, pass in the favoriteMovie function and name of movie
movies(favoriteMovie, "Finding Nemo");
```

Returns: My favorite movie is Finding Nemo

But you could have bypassed the first assignment of the function, by passing the function to the movies() function inline.

```
// Function declaration that takes in two arguments: a function for displaying
// a message, along with a name of a movie
function movies(messageFunction, name) {
  messageFunction(name);
}

// Call the movies function, pass in the function and name of movie
movies(function displayFavorite(movieName) {
  console.log("My favorite movie is " + movieName);
}, "Finding Nemo");
```

Returns: My favorite movie is Finding Nemo

This type of syntax, writing function expressions that pass a function into another function inline, is really common in JavaScript. It can be a little tricky at first, but be patient, keep practicing, and you'll start to get the hang of it!

Why use anonymous inline function expressions?

Using an anonymous inline function expression might seem like a very not-useful thing at first. Why define a function that can only be used once and you can't even call it by name?

Anonymous inline function expressions are often used with function callbacks that are probably not going to be reused elsewhere. Yes, you could store the function in a variable, give it a name, and pass it in like you saw in the examples above. However, when you know the function is not going to be reused, it could save you many lines of code to just define it inline.

6.6. Quizzes and examples

Quiz: Laugh it Off 1

Declare a function called laugh() that returns "hahahahahahahahaha!". Print the value returned from the laugh()function to the console.

```
/*
 * Programming Quiz: Laugh it Off 1 (5-1)
 */

// your code goes here

function laugh() {
  var message = "hahahahahahahahaha!";
  return message;
}
console.log(laugh());
```

Returns: `hahahahahahahahaha!`

Quiz: Laugh it Off 2

Write a function called laugh() that takes one parameter, num that represents the number of "ha"s to return.
TIP: You might need a loop to solve this!

Here's an example of the output and how to call the function that you will write:

```
console.log(laugh(3));
```

Prints: "hahaha!"

```
 * Programming Quiz: Laugh it Off 2 (5-2)
 *
 * Write a function called `laugh` with a parameter named `num` that represents the number of "ha"s to return.
 * Note:
 *  - make sure your the final character is an exclamation mark ("!")
 */
function laugh (num){
    var message = "";
    for (var x=1; x<=num; x++){
        message = message+"ha";
    }
    return message +"!";
}
console.log(laugh(3));
```

Returns: `hahaha!`

Quiz: Build a Triangle

For this quiz, you're going to create a function called buildTriangle() that will accept an input (the triangle at its widest width) and will return the string representation of a triangle. See the example output below.

```
buildTriangle(10);
```

Returns:

```
*
* *
* * *
* * * *
* * * * *
* * * * * *
* * * * * * *
* * * * * * * *
* * * * * * * * *
* * * * * * * * * *
```

We've given you one function makeLine() to start with. The function takes in a line length, and builds a line of asterisks and returns the line with a newline character.

```
function makeLine(length) {
  var line = "";
  for (var j = 1; j <= length; j++) {
    line += "* "
  }
  return line + "\n";
}
```

You will need to call this makeLine() function in buildTriangle().
This will be the most complicated program you've written yet, so take some time *thinking* through the problem before diving into the code. What tools will you need from your JavaScript tool belt? Professionals plan out their code before writing anything. Think through the steps your code will need to take and write them down in order. Then go through your list and convert each step into actual code. Good luck!

```
/*
 * Programming Quiz: Build A Triangle (5-3)
 */
// creates a line of * for a given length

function makeLine(length) {
    var line = "";
    for (var j = 1; j <= length; j++) {
        line += "* ";
    }
    return line + "\n";
}

function buildTriangle(length) {
    var raw = "";
    for (var x=1; x<=length; x++){
        raw += makeLine(x);
    }
    return raw;
}

console.log(buildTriangle(10));
// your code goes here.  Make sure you call makeLine() in your own code.
// test your code by uncommenting the following line
//console.log(buildTriangle(10));
```

Quiz: Laugh

Write an anonymous function expression that stores a function in a variable called "laugh" and outputs the number of "ha"s that you pass in as an argument.

```
laugh(3);
```

Returns: hahaha!

```
/*
 * Programming Quiz: Laugh (5-4)
 */
var laugh = function(y){
    var message="";
    for (var x=1; x<=y; x++){
        message += "ha";
    }
    return message + "!";
};

console.log(laugh(3));
```

Returns: `hahaha!`

Quiz: Cry

Write a named function expression that stores the function in a variable called cry and returns "boohoo!". Don't forget to call the function using the variable name, not the function name:

```
cry();
```

Returns: boohoo!

```
/*
 * Programming Quiz: Cry (5-5)
 */

// your code goes here
var cry = function cryingFunc(){
    return "boohoo!";
};
console.log(cry());
```

Returns: `boohoo!`

Quiz: Inline

Call the emotions() function so that it prints the output you see below, but instead of passing the laugh() function as an argument, pass an inline function expression instead.

```
emotions("happy", laugh(2)); // you can use your laugh function from the previous quizzes
```

Prints: "I am happy, haha!"

```
/*
 * Programming Quiz: Inline Functions (5-6)
 */

// don't change this code
function emotions(myString, myFunc) {
    console.log("I am " + myString + ", " + myFunc(2));
}

// your code goes here
// call the emotions function here and pass in an
// inline function expression

emotions("happy", function laugh(y){
    var message="";
    for(var x=1; x<=y; x++){
        message += "ha";
    }
    return message + "!";
});
```

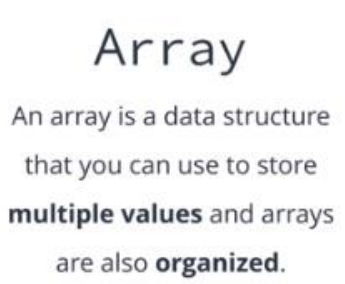
Returns: `I am happy, haha!`

7. JavaScript Arrays

A foundational concept of programming is how to organize and store data.
One way we organize data in real life is to make lists.

Let's make one here:

New Year's Resolutions:
1. Rappel into a cave
2. Take a falconry class
3. Learn to juggle



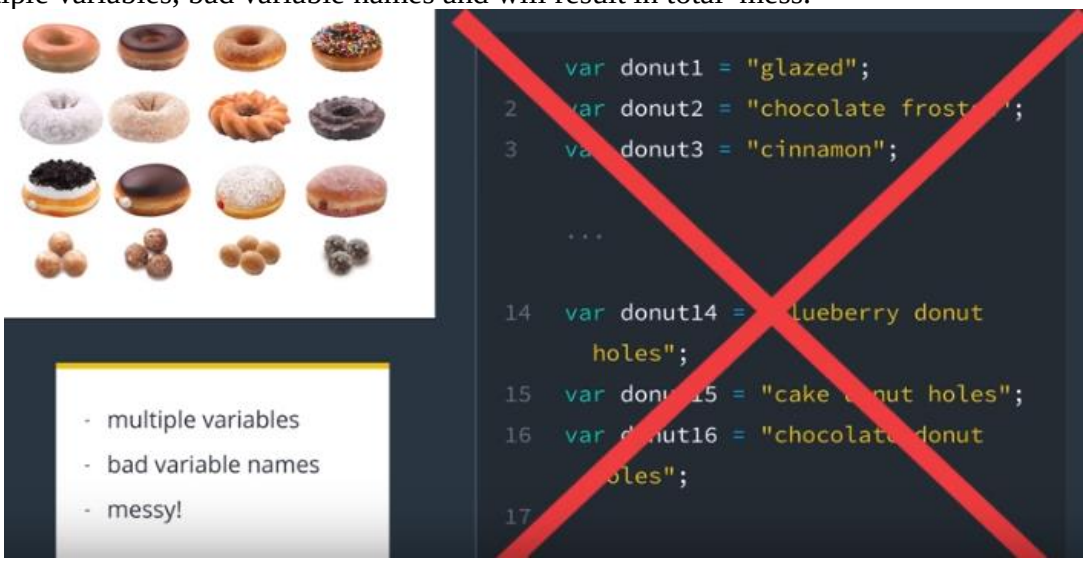
Let's now write this list in JavaScript, as an *array*:

```
let newYearsResolutions = ['Rappel into a cave', 'Take a falconry class', 'Learn to juggle'];
```

Arrays are JavaScript's way of making lists. These lists can store any data types (including strings, numbers, and booleans) and they are ordered, meaning each item has a numbered position.

Donuts example:
Let's assume we have a donuts shop with a number of donuts types we want to keep track of:

Obviously, creating a variable for each kind of donuts is not an option, as it will include: multiple variables, bad variable names and will result in total mess.



As it was mentioned, we could think of arrays as lists, just the first item starts with 0.


```
var donuts = ["glazed","chocolate frosted","cinnamon","sprinkled","powdered","cinnamon sugar","glazed cruller","
    chocolate cruller","cookies","Boston creme","powdered jelly filled","creme de leche","glazed donut holes","blueberry
    donut holes","cake donut holes","chocolate donut holes"];
```

An **array** is useful because it stores multiple values into a single, organized data structure. You can define a new array by listing values separated with commas between square brackets [].

```
// creates a `donuts` array with three strings
var donuts = ["glazed", "powdered", "jelly"];
```

But strings aren't the only type of data you can store in an array. You can also store numbers, booleans… and really anything!

```
// creates a `mixedData` array with mixed data types
var mixedData = ["abcd", 1, true, undefined, null, "all the things"];
```

You can even store an array in an array to create a **nested array**!

```
// creates a `arraysInArrays` array with three arrays
var arraysInArrays = [[1, 2, 3], ["Julia", "James"], [true, false, true, false]];
```

Nested arrays can be particularly hard to read, so it's common to write them on one line, using a newline after each comma:

```
var arraysInArrays = [
  [1, 2, 3],
  ["Julia", "James"],
  [true, false, true, false]
];
```

Accessing Array Elements

Each type of donuts from the above example is an element in the array:

7.1. Array Index

Remember that elements in an array are indexed starting at the position 0. To access an element in an array, use the name of the array immediately followed by square brackets containing the index of the value you want to access.

```
var donuts = ["glazed", "powdered", "sprinkled"];
console.log(donuts[0]); // "glazed" is the first element in the `donuts` array
```

Prints: "glazed"

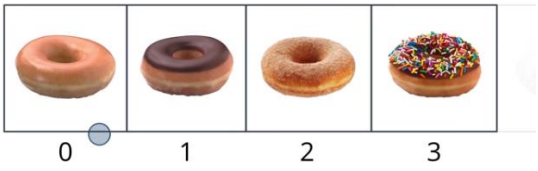
index

References the location, or **position**, of an element in an array.

One thing to be aware of is if you try to access an element at an index that does not exist, a value of undefined will be returned back.

```
console.log(donuts[3]); // the fourth element in `donuts` array does not exist!
```

Prints: undefined

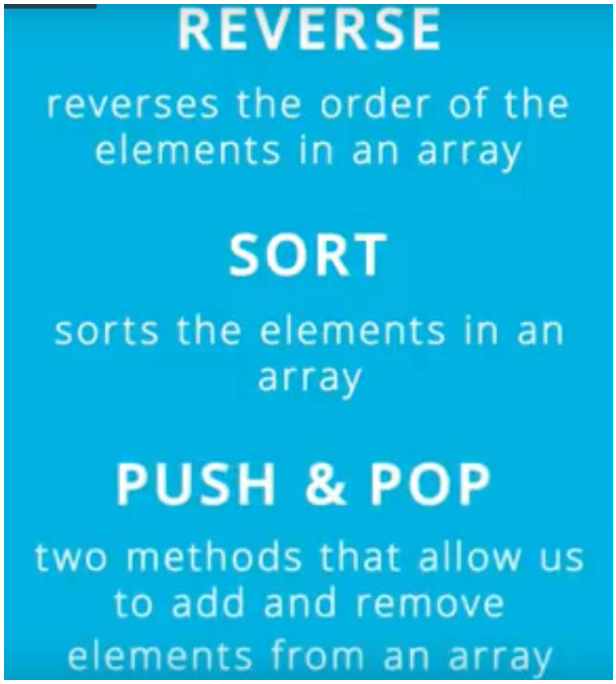


Finally, if you want to change the value of an element in array, you can do so by setting it equal to a new value.

```
donuts[1] = "glazed cruller"; // changes the second element in the `donuts` array to "glazed cruller"
console.log(donuts[1]);
```

Prints: "glazed cruller"

7.2. Array Properties and Methods



JavaScript provides a large number of built-in methods for modifying arrays and accessing values in an array, check out the MDN Documentation, or type []. into the JavaScript console for a list of all the available Array methods.

Property: Length

You can find the **length** of an array by using its length property.

```
var donuts = ["glazed", "powdered", "sprinkled"];
console.log(donuts.length);
```

Prints: 3

To access the length property, type the name of the array, followed by a period . (you'll also use the period to access other properties and methods), and the word length. The length property will then return the **number of elements** in the array.

TIP: Strings have a length property too! You can use it to get the length of any string. For example, "supercalifragilisticexpialidocious".length returns 34.

Method: Push

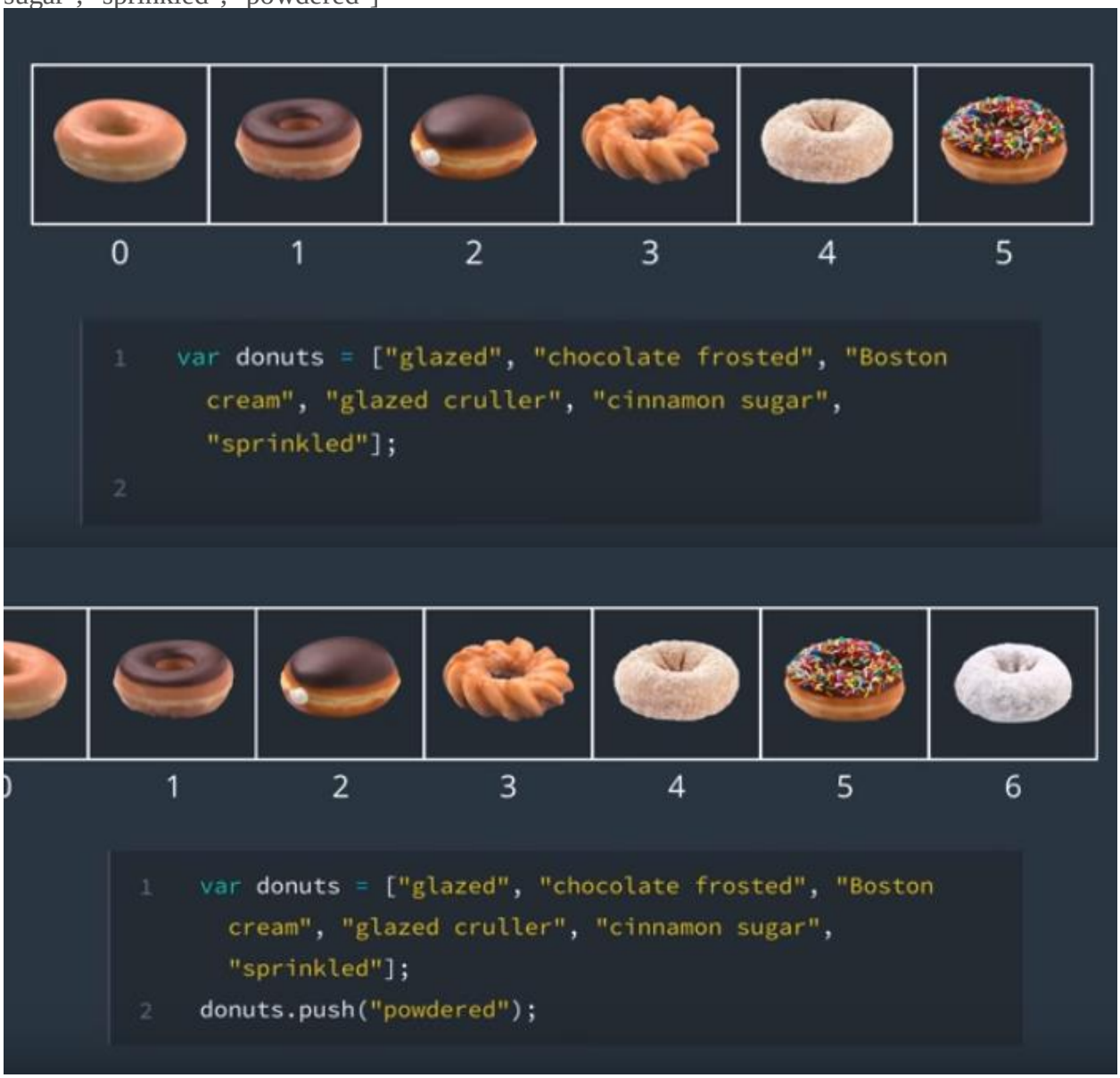
You can use the push() method to add elements to the *end of an array*.
You can represent the spread of donuts using an array.

```
var donuts = ["glazed", "chocolate frosted", "Boston creme", "glazed cruller", "cinnamon sugar", "sprinkled"];
```

Then, you can *push* donuts onto the end of the array using the push() method.

```
donuts.push("powdered"); // pushes "powdered" onto the end of the `donuts` array
```

Returns: 7
donuts array: ["glazed", "chocolate frosted", "Boston creme", "glazed cruller", "cinnamon sugar", "sprinkled", "powdered"]



Notice, with the push() method you need to pass the value of the element you want to add to the end of the array. Also, the push() method returns the length of the array after an element has been added.

```
var donuts = ["glazed", "chocolate frosted", "Boston creme", "glazed cruller", "cinnamon sugar", "sprinkled"];
donuts.push("powdered"); // the `push()` method returns 7 because the `donuts` array now has 7 elements
```

Returns: 7

Method: Splice

splice() is another handy method that allows you to add and remove elements from anywhere within an array.
While push() and pop() limit you to adding and removing elements from *the end of an array*, splice() lets you specify the index location to add new elements, as well as the number of elements you'd like to delete (if any).

```
var donuts = ["glazed", "chocolate frosted", "Boston creme", "glazed cruller"];
donuts.splice(1, 1, "chocolate cruller", "creme de leche"); // removes "chocolate frosted" at index 1 and adds "chocolate cruller" and "creme de leche" starting at index 1
```

Returns: ["chocolate frosted"]
donuts array: ["glazed", "chocolate cruller", "creme de leche", "Boston creme", "glazed cruller"]

The first argument represents the starting index from where you want to change the array, the second argument represents the numbers of elements you want to remove, and the remaining arguments represent the elements you want to add.

splice() is an incredibly powerful method that allows you to manipulate your arrays in a variety of ways. Any combination of adding or removing elements from an array can all be done in one simple line of code.

7.3. Array Loops

Once the data is in the array, you want to be able to efficiently access and manipulate each element in the array without writing repetitive code for each element.

For instance, if this was our original donuts array:

```
var donuts = ["jelly donut", "chocolate donut", "glazed donut"];
```

and we decided to make all the same donut types, but only sell them as *donut holes* instead, we could write the following code:

```
donuts[0] += " hole";
donuts[1] += " hole";
donuts[2] += " hole";
```

donuts array: ["jelly donut hole", "chocolate donut hole", "glazed donut hole"]

But remember, you have another powerful tool at your disposal, **loops**!
To loop through an array, you can use a variable to represent the index in the array, and then loop over that index to perform whatever manipulations your heart desires.

```
var donuts = ["jelly donut", "chocolate donut", "glazed donut"];

// the variable `i` is used to step through each element in the array
for (var i = 0; i < donuts.length; i++) {
   donuts[i] += " hole";
   donuts[i] = donuts[i].toUpperCase();
}
```

donuts array: ["JELLY DONUT HOLE", "CHOCOLATE DONUT HOLE", "GLAZED DONUT HOLE"]

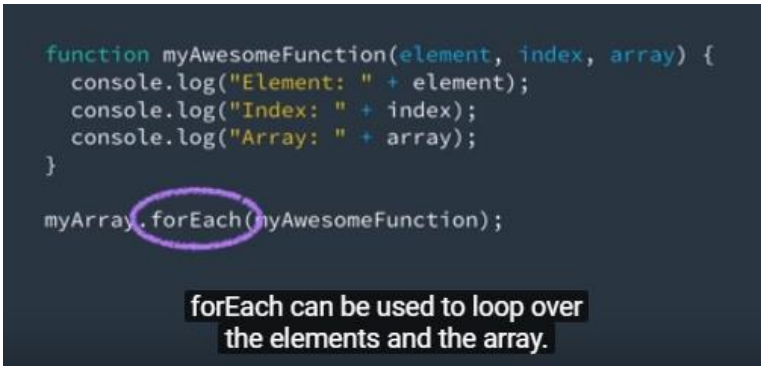
In this example, the variable i is being used to represent the index of the array. As i is incremented, you are stepping over each element in the array starting from 0 until donuts.length - 1 (donuts.length is out of bounds).

Method: forEach()

The forEach() method gives you an alternative way to iterate over an array, and manipulate each element in the array with an inline function expression.

```
var donuts = ["jelly donut", "chocolate donut", "glazed donut"];

donuts.forEach(function(donut) {
  donut += " hole";
  donut = donut.toUpperCase();
  console.log(donut);
});
```

Prints:
JELLY DONUT HOLE
CHOCOLATE DONUT HOLE
GLAZED DONUT HOLE



Notice that the forEach() method iterates over the array without the need of an explicitly defined index. In the example above, donut corresponds to the element in the array itself. This is different from a for or while loop where an index is used to access each element in the array:

```
for (var i = 0; i < donuts.length; i++) {
  donuts[i] += " hole";
  donuts[i] = donuts[i].toUpperCase();
  console.log(donuts[i]);
}
```

Parameters

The function that you pass to the forEach() method can take up to three parameters. These are called element, index, and array, but you can call them whatever you like.

The forEach() method will call this function once *for each* element in the array (hence the name forEach.) Each time, it will call the function with different arguments. The element parameter will get the *value* of the array element. The index parameter will get the *index* of the element (starting with zero). The arrayparameter will get a reference to the whole array, which is handy if you want to modify the elements.
Here's another example:

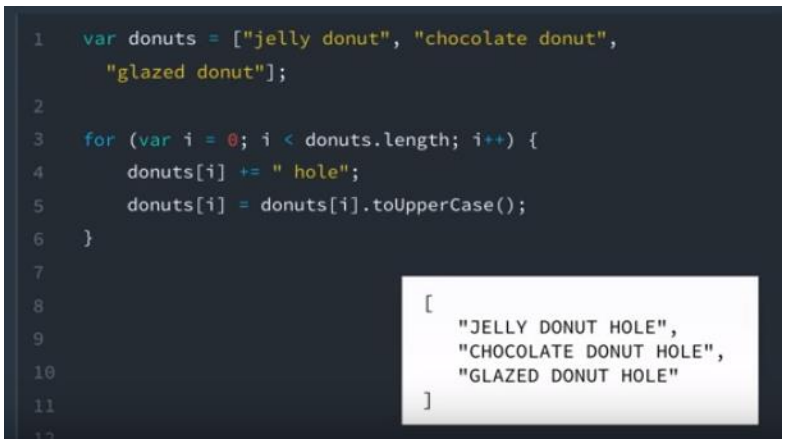
```
words = ["cat", "in", "hat"];
words.forEach(function(word, num, all) {
  console.log("Word " + num + " in " + all.toString() + " is " + word);
});
```

Prints:
Word 0 in cat,in,hat is cat
Word 1 in cat,in,hat is in
Word 2 in cat,in,hat is hat

For example:

If we want to sell donut holes of the same type as our donuts, we could write the following code:

However, this approach is horrible, resulting in a repetitive code. The better way is to use loop:



Otherrwise, we may re-organize the function using forEach() method:

```
var donuts = ["jelly donut", "chocolate donut",
  "glazed donut"];

function printDonuts(donut) {
  donut += " hole";
  donut = donut.toUpperCase();
  console.log(donut);
}

donuts.forEach(printDonuts);
```

We also, may define it as the inline function expression, resulting in:

```
var donuts = ["jelly donut", "chocolate donut",
  "glazed donut"];

donuts.forEach(function(donut) {
  donut += " hole";
  donut = donut.toUpperCase();
  console.log(donut);
});
```

Method: map()

Using forEach() will not be useful if you want to permanently modify the original array. forEach() always returns undefined. However, creating a new array from an existing array is simple with the powerful map()method.
With the map() **method**, you can take an array, perform some operation on each element of the array, and return a new array.

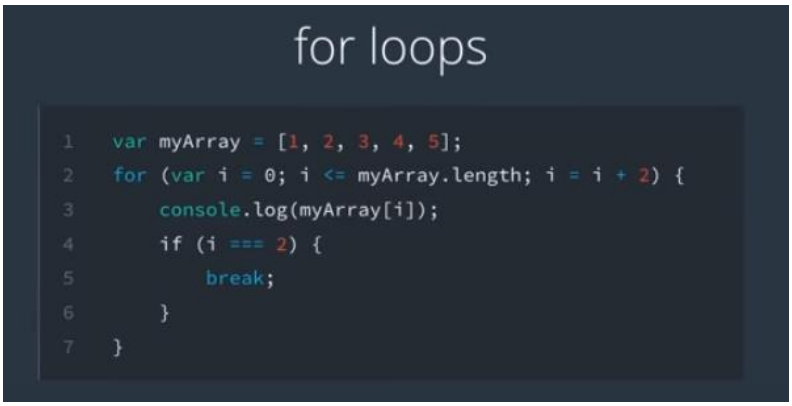
```
var donuts = ["jelly donut", "chocolate donut", "glazed donut"];

var improvedDonuts = donuts.map(function(donut) {
  donut += " hole";
  donut = donut.toUpperCase();
  return donut;
});
```

donuts array: ["jelly donut", "chocolate donut", "glazed donut"]
improvedDonuts array: ["JELLY DONUT HOLE", "CHOCOLATE DONUT HOLE", "GLAZED DONUT HOLE"]

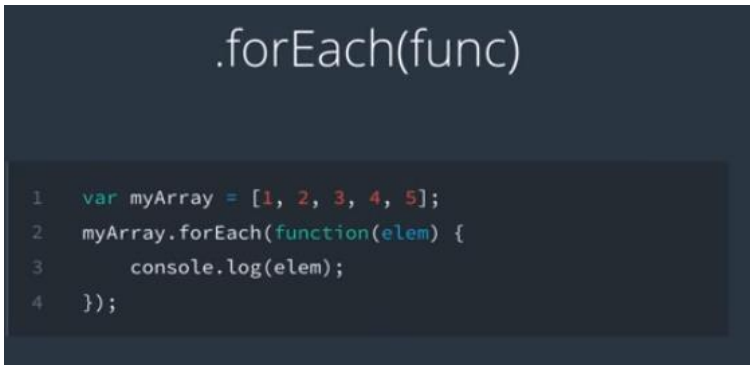
Oh man, did you just see that? The map() method accepts one argument, a function that will be used to manipulate each element in the array. In the above example, we used a function expression to pass that function into map(). This function is taking in one

argument, donut which corresponds to each element in the donuts array. You no longer need to iterate over the indices anymore. map() does all that work for you.

So, there are lots of ways to loop over arrays:



For loops give you complete control over the looping process: you define where to strat, where to stop, whether you want to skip over values in the array and whether to break out the loop early (early the break statement).



.forEach() is a vay of looping over an array. You would have lesss versatility than you do with a regular for loop, but you can access each element directly (and you don't need an index of an element to do that).

Similar to forEach() you may define a callback function that performs an operation on each element in the array. The differrence is map() returns the new array with new values your function calculated.

7.4. Quizzes and examples

Quiz: UdaciFamily

Create an array called udaciFamily and add "Julia", "James", and your name to the array. Then, print the udaciFamily to the console using console.log.

```
/*
 * Programming Quiz: UdaciFamily (6-1)
 */

// your code goes here
var udaciFamily = ["Julia", "James", "Julia"];
console.log(udaciFamily);
```

Returns: `[ 'Julia', 'James', 'Julia' ]`

Quiz: Building the Crew

Create an array called crew to organize the Serenity's crew and set it equal to the variables below . You don't need to type out the actual strings, just use the provided variables.

```
var captain = "Mal";
var second = "Zoe";
var pilot = "Wash";
var companion = "Inara";
var mercenary = "Jayne";
var mechanic = "Kaylee";
```

Then, print the crew array to the console.

```
/*
 * Programming Quiz: Building the Crew (6-2)
 */

var captain = "Mal";
var second = "Zoe";
var pilot = "Wash";
var companion = "Inara";
var mercenary = "Jayne";
var mechanic = "Kaylee";

// your code goes here
var crew=[captain,second,pilot,mechanic,mercenary,companion];
console.log(crew);
```

Returns: `[ 'Mal', 'Zoe', 'Wash', 'Kaylee', 'Jayne', 'Inara' ]`

Quiz: Joining the Crew

In an earlier exercise, you created a crew array to represent Mal's crew from the hit show Firefly.

```
var captain = "Mal";
var second = "Zoe";
var pilot = "Wash";
var companion = "Inara";
var mercenary = "Jayne";
var mechanic = "Kaylee";

var crew = [captain, second, pilot, companion, mercenary, mechanic];
```

Later in the show, Mal takes on three new crew members named "Simon", "River", and "Book". Use the push() method to add the three new crew members to the crew array.

```
var doctor = "Simon";
var sister = "River";
var shepherd = "Book";
```

```
/*
 * Programming Quiz: Joining the Crew (6-6)
 */

var captain = "Mal";
var second = "Zoe";
var pilot = "Wash";
var companion = "Inara";
var mercenary = "Jayne";
var mechanic = "Kaylee";

var crew = [captain, second, pilot, companion, mercenary, mechanic];

var doctor = "Simon";
var sister = "River";
var shepherd = "Book";

crew.push(doctor);
crew.push(sister);
crew.push(shepherd);

console.log(crew);
```

Returns: `[ 'Mal', 'Zoe', 'Wash', 'Inara', 'Jayne', 'Kaylee', 'Simon', 'River', 'Book' ]`

Quiz: The Price is Right

Starting with this array of prices, change the prices of the 1st, 3rd, and 7th elements in the array.

```
var prices = [1.23, 48.11, 90.11, 8.50, 9.99, 1.00, 1.10, 67.00];
```

TIP: The 1st element of any array has an index of 0.
Afterwards, print out the prices array to the console.

```
/*
 * Programming Quiz: The Price is Right (6-3)
 */

var prices = [1.23, 48.11, 90.11, 8.50, 9.99, 1.00, 1.10, 67.00];

// your code goes here
prices[0]=1.00;
prices[2]=90.99;
prices[6]=1.50;
console.log(prices);
```

Returns: `[ 1, 48.11, 90.99, 8.5, 9.99, 1, 1.5, 67 ]`

Quiz: Colors of the Rainbow

James was creating an array with the colors of the rainbow, and he forgot some colors. The standard rainbow colors are usually listed in this order:

```
var rainbow = ["Red", "Orange", "Yellow", "Green", "Blue", "Purple"];
```

but James had this:

```
var rainbow = ["Red", "Orange", "Blackberry", "Blue"];
```

Using only the splice() method, insert the missing colors into the array, and remove the color "Blackberry"by following these steps:

1. Remove "Blackberry"
2. Add "Yellow" and "Green"
3. Add "Purple"

```
 * Programming Quiz: Colors of the Rainbow (6-4)
 */
var rainbow = ["Red", "Orange", "Blackberry", "Blue"];


// your code goes here
var removed=rainbow.splice(2, 1, "Yellow", "Green");
var added=rainbow.splice(5, 0, "Purple");
console.log(rainbow);
```

Returns: `[ 'Red', 'Orange', 'Yellow', 'Green', 'Blue', 'Purple' ]`

Quiz: Quidditch Cup

Consider the following array.

```
var team = ["Oliver Wood", "Angelina Johnson", "Katie Bell", "Alicia Spinnet", "George Weasley", "Fred Weasley", "Harry Potter"];
```

Create a function called hasEnoughPlayers() that takes the team array as an argument and returns true or false depending on if the array has at least seven players.

```
/*
 * Programming Quiz: Quidditch Cup (6-5)
 */

// your code goes here

var team = ["Oliver Wood", "Angelina Johnson", "Katie Bell", "Alicia Spinnet", "George Weasley", "Fred Weasley", "Harry Potter"];
function hasEnoughPlayers(team){
return team.length >= 7;
}

console.log(hasEnoughPlayers(team));
```

Returns: true

Quiz: Another Type of Loop

Use the array's forEach() **method** to loop over the following array and add 100 to each of the values if the value is divisible by 3.

```
var test = [12, 929, 11, 3, 199, 1000, 7, 1, 24, 37, 4, 19, 300, 3775, 299, 36, 209, 148, 169, 299, 6, 109, 20, 58, 139, 59, 3, 1, 139];
```

Remember that the "Test Run" button will display any logged content, so feel free to use console.log() to test your code.

```
/*
 * Programming Quiz: Another Type of Loop (6-8)
 *
 * Use the existing `test` variable and write a `forEach` loop
 * that adds 100 to each number that is divisible by 3.
 *
 * Things to note:
 *  - you must use an `if` statement to verify code is divisible by 3
 *  - you can use `console.log` to verify the `test` variable when you're finished looping
 */
var test = [12, 929, 11, 3, 199, 1000, 7, 1, 24, 37, 4,
    19, 300, 3775, 299, 36, 209, 148, 169, 299,
    6, 109, 20, 58, 139, 59, 3, 1, 139];

// Write your code here
test.forEach(function mF(element, index, array) {
  if(element%3===0){
      array[index]+=100;
  }
});

console.log(test);
```

Returns: `[ 112, 929, 11, 103, 199, 1000, 7, 1, 124, 37, 4, 19, 400, 3775, 299, 136, 209, 148, 169, 299, 106, 109, 20, 58, 139, 59, 103, 1, 139 ]`

Quiz: I Got Bills

Use the map() **method** to take the array of bill amounts shown below, and create a second array of numbers called totals that shows the bill amounts with a 15% tip added.

```
var bills = [50.23, 19.12, 34.01, 100.11, 12.15, 9.90, 29.11, 12.99, 10.00, 99.22, 102.20, 100.10, 6.77, 2.22];
```

Print out the new totals array using console.log.

TIP: Check out the toFixed() method for numbers to help with **rounding** the values to a maximum of 2 decimal places. Note, that the method returns a string to maintain the "fixed" format of the number. So, if you want to convert the string back to a number, you can **cast** it or convert it back to a number:

```
Number("1");
```

Returns: 1

```
/*
 * Programming Quiz: I Got Bills (6-9)
 *
 * Use the .map() method to take the bills array below and create a second array
 * of numbers called totals. The totals array should contains each amount from the
 * bills array but with a 15% tip added. Log the totals array to the console.
 *
 * For example, the first two entries in the totals array would be:
 *
 * [57.76, 21.99, ... ]
 *
 * Things to note:
 *  - each entry in the totals array must be a number
 *  - each number must have an accuracy of two decimal points
 */
var bills = [50.23, 19.12, 34.01,
   100.11, 12.15, 9.90, 29.11, 12.99,
   10.00, 99.22, 102.20, 100.10, 6.77, 2.22
];

var totals = bills.map(function(bills) {
  bills *= 1.15;
  return Number(bills.toFixed(2));
});

console.log(totals);
```

Returns: `[ 57.76, 21.99, 39.11, 115.13, 13.97, 11.38, 33.48, 14.94, 11.5, 114.1, 117.53, 115.11, 7.79, 2.55 ]`

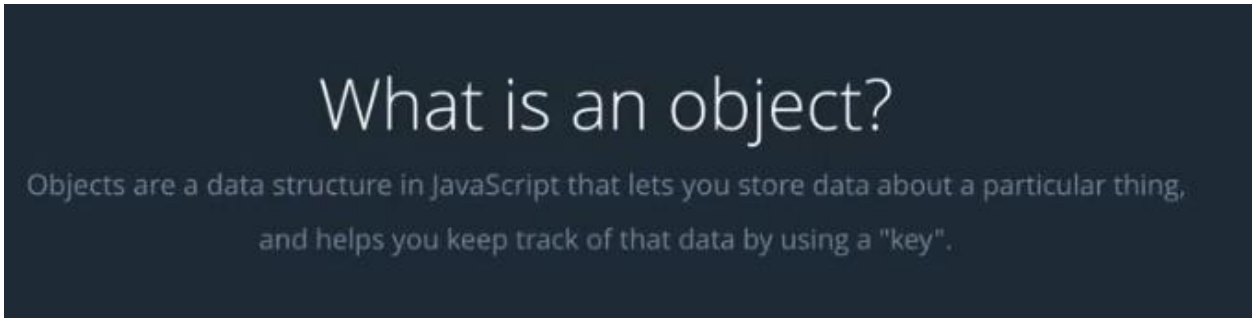
Quiz: Nested Numbers

Use a nested for loop to take the numbers array below and replace all of the values that are divisible by 2(even numbers) with the string "even" and all other numbers with the string "odd".

```
var numbers = [
   [243, 12, 23, 12, 45, 45, 78, 66, 223, 3],
   [34, 2, 1, 553, 23, 4, 66, 23, 4, 55],
   [67, 56, 45, 553, 44, 55, 5, 428, 452, 3],
   [12, 31, 55, 445, 79, 44, 674, 224, 4, 21],
   [4, 2, 3, 52, 13, 51, 44, 1, 67, 5],
   [5, 65, 4, 5, 5, 6, 5, 43, 23, 4424],
   [74, 532, 6, 7, 35, 17, 89, 43, 43, 66],
   [53, 6, 89, 10, 23, 52, 111, 44, 109, 80],
   [67, 6, 53, 537, 2, 168, 16, 2, 1, 8],
   [76, 7, 9, 6, 3, 73, 77, 100, 56, 100]
];
```

```
 * Programming Quiz: Nested Numbers (6-10)
 *   - The `numbers` variable is an array of arrays.
 *   - Use a nested `for` loop to cycle through `numbers`.
 *   - Convert each even number to the string "even"
 *   - Convert each odd number to the string "odd"
 */
var numbers = [
   [243, 12, 23, 12, 45, 45, 78, 66, 223, 3],
   [34, 2, 1, 553, 23, 4, 66, 23, 4, 55],
   [67, 56, 45, 553, 44, 55, 5, 428, 452, 3],
   [12, 31, 55, 445, 79, 44, 674, 224, 4, 21],
   [4, 2, 3, 52, 13, 51, 44, 1, 67, 5],
   [5, 65, 4, 5, 5, 6, 5, 43, 23, 4424],
   [74, 532, 6, 7, 35, 17, 89, 43, 43, 66],
   [53, 6, 89, 10, 23, 52, 111, 44, 109, 80],
   [67, 6, 53, 537, 2, 168, 16, 2, 1, 8],
   [76, 7, 9, 6, 3, 73, 77, 100, 56, 100]
];
// your code goes here
for (var row = 0; row < numbers.length; row++) {
   for (var column = 0; column < numbers[row].length; column++) {
      numbers[row][column]%2===0 ? numbers[row][column]="even" :
numbers[row][column]="odd";
  }
}
```

8. JavaScript Objects



JavaScript *objects* are containers that can store data and functions. The data we store in an object is not ordered — we can only access it by calling its associated *key*.
You can create an object with *key-value* pairs using the following syntax:

Arrays are a special kind of object in JavaScript

```
let restaurant = {
  name: 'Italian Bistro',
  seatingCapacity: 120,
  hasDineInSpecial: true,
  entrees: ['Penne alla Bolognese', 'Chicken Cacciatore', 'Linguine Pesto']
};
```

Let's consider the above example one step at a time:

1. let restaurant creates a variable named restaurant that stores the object.
2. We create the object between curly braces: {}.
3. name, seatingCapacity, hasDineInSpecial, and entrees are all keys.
4. We separate each key from its corresponding value by a colon (:).
5. The value is to the right of the colon. For example, seatingCapacity's value is 120.
6. Every key-value pair is separated by a comma ,.

An objects keys point to values that can be any data type, including other objects.

JavaScript comes with a number of built-in objects, also, you may create your own objects. The example with umbrella:

So, to control the presented umbrella, first of all we need to create an object to represent umbrella:
One way to define an object is to create a variable:

```
var umbrella = {};
```

You may verify whether the umbrella is an object by using the **typeof** operator.

NOTE: typeof is an operator that returns the name of the data type that follows it:

```
typeof "hello" // returns "string"
typeof true // returns "boolean"
typeof [1, 2, 3] // returns "object" (Arrays are a type of object)
typeof function hello() { } // returns "function"
```

Objects have **properties** and things they can do. To add this information, you can define **key-value** pairs for each piece of data. The below is defined property key called color, and it is **value** is “pink”. The umbrella is closed, therefore isOpen property is false.

So, we want the umbrella to open – something the object can do is a **method**. A method is just a function that is associated with an object (just like push and pop methods with the arrays).

```
var umbrella = {
    color: "pink",
    isOpen: false
};
```

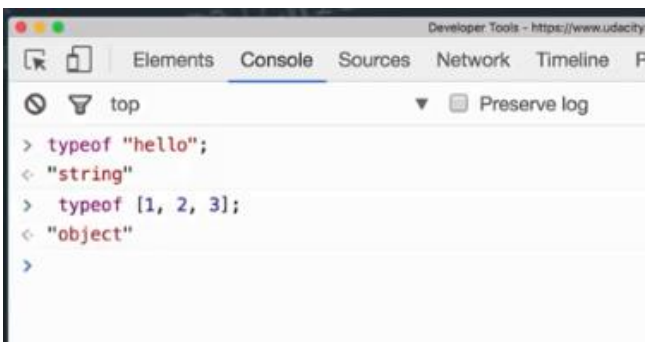
```
var umbrella = {
    color: "pink",
    isOpen: false,
    open: function() {
     if (umbrella.isOpen === true) {
       return "The umbrella is
       already opened!";
     } else {
       umbrella.isOpen = true;
       return "Julia opens the
       umbrella";
     }
    }
};
```

The JavaScript Console

```
umbrella.isOpen;
> false
umbrella.open();
> "Julia opens the umbrella!"
umbrella.isOpen;
> true
```

TIP: It’s worth noting that while we can represent real-world objects as JavaScript objects, the analogy does not always hold. This is a good starting place for thinking about the structure and purpose of objects, but as you continue your career as a developer, you’ll find that JavaScript objects can behave wildly different than real objects.

What happens when we put string or an array to the console:



8.1. Object Literals

```
var sister = {
  name: "Sarah",
  age: 23,
  parents: [ "alice", "andy" ],
  siblings: ["julia"],
  favoriteColor: "purple",
  pets: true
};
```

The syntax you see above is called **object-literal notation**. There are some important things you need to remember when you're structuring an object literal:

- The "key" (representing a **property** or **method** name) and its "value" are separated from each other by a **colon**
- The key: value *pairs* are separated from each other by **commas**
- The entire object is wrapped inside curly braces { }.

And, kind of like how you can look up a word in the dictionary to find its definition, the key in a key:valuepair allows you to look up a piece of information about an object. Here's are a couple examples of how you can retrieve information about my sister's parents using the object you created.

```
// two equivalent ways to use the key to return its value
sister["parents"] // returns [ "alice", "andy" ]
sister.parents // also returns ["alice", "andy"]
```

Using sister["parents"] is called **bracket notation** (because of the brackets!) and using sister.parents is called **dot notation** (because of the dot!).

What about methods?

The sister object above contains a bunch of properties about my sister, but doesn't really say what my sister *does*. For instance, let's say my sister likes to paint. You might have a paintPicture() method that returns "Sarah paints a picture!" whenever you call it. The syntax for this is pretty much exactly the same as how you defined the properties of the object. The only difference is, the value in the key:value pair will be a function.

```
var sister = {
  name: "Sarah",
  age: 23,
  parents: [ "alice", "andy" ],
  siblings: ["julia"],
  favoriteColor: "purple",
  pets: true,
  paintPicture: function() { return "Sarah paints!"; }
};

sister.paintPicture();
```

Returns: "Sarah paints!"

and you can access the name of my sister by accessing the name property:

```
sister.name
```

Returns: "Sarah"

8.2. Naming Conventions

There are some Naming Conventions that should be followed; failing to follow them may lead to bugs.

Feel free to use upper and lowercase numbers and letters, but don't *start* your property name with a number. You don't need to wrap the string in quotes! If it's a multi-word property, use camel case. Don't use hyphens in your property names

```
var richard = {
  "1stSon": true;
  "loves-snow": true;
};

richard.1stSon // error
richard.loves-snow // error
```

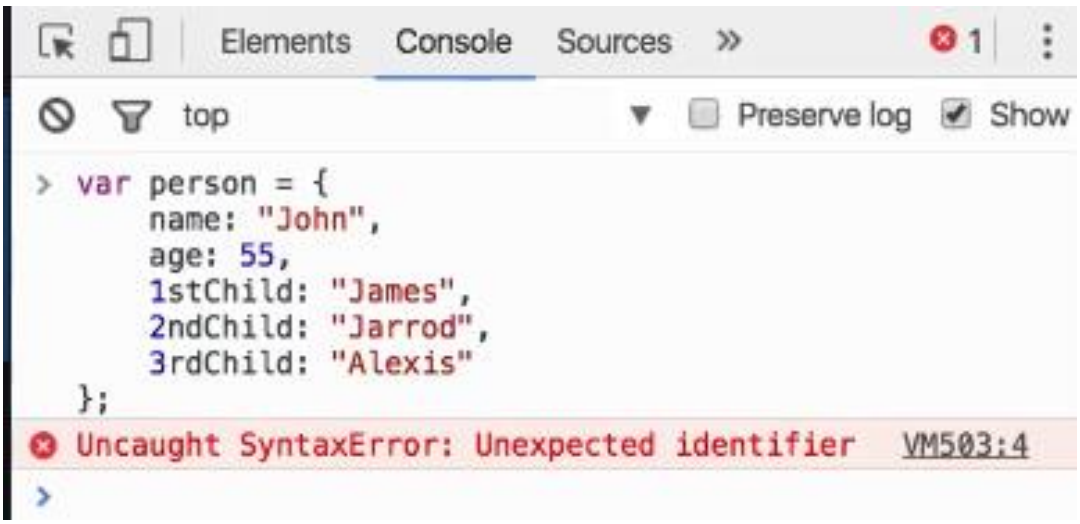
Example 1:

There is an object called “person”, including some properties. So he is John, he’s 55 and he’s got 3 children.

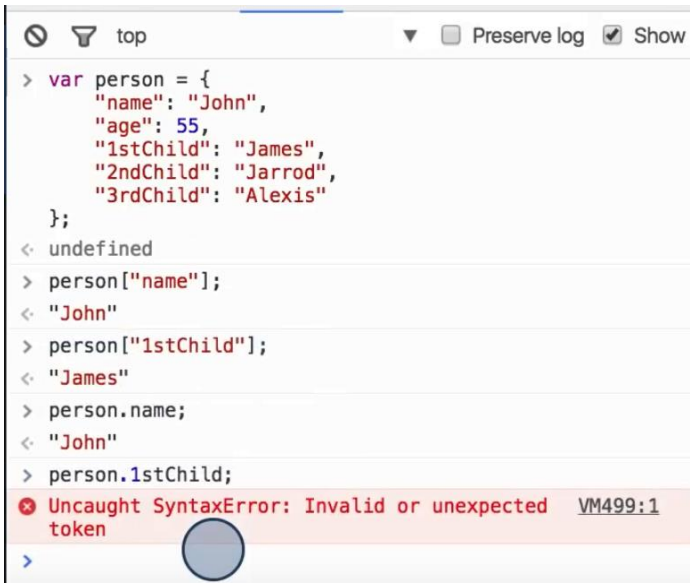
```
var person = {
    "name": "John",
    "age": 55,
    "1stChild": "James",
    "2ndChild": "Jarrod",
    "3rdChild": "Alexis"
};
```

It is important to use quotes around the property name. Even though quotes are not required, using them may mass some potential problems.

The above example without quotes results in error:

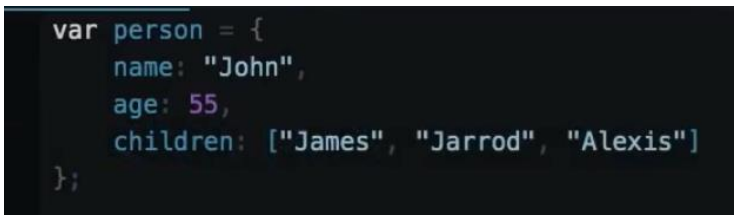


There are two ways to access properties: bracket notation or dot notation.



When we are trying to get the "1stChild" using dot notation, we are getting error – so, **don't use numbers as the first character in property names, it is bad practice in general.**

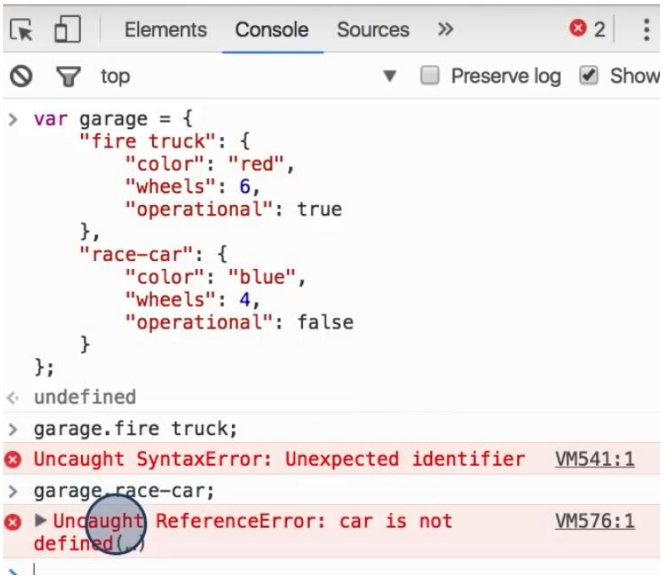
The best way to define the mentioned object would be:



Example 2:

The second example contains some problems: using spaces and using hyphens (minus sign), these should be avoided, as neither of these will work out when using dot notation.

So, if you want to use multi-word property name, instead of using space or hyphen, the prefer method is camelCasing.



So, the correct way of defining the above example:

8.3. Quizzes and examples

Quiz: Umbrella

Using the umbrella example from the previous video, see if you can follow the example open() method and create the close() method. It's alright if you have trouble at first! We'll go into more detail later in this lesson.

TIP: Remember to put all of your object's properties and methods inside curly braces: `var myObject = { greeting: "hello", name: "Julia" };`. Also, remember that an object is just another data type. Just like how you would put a semicolon after a string variable declaration `var myString = "hello";`, don't forget to put a semi-colon at the end of your object's declaration.

```
var umbrella = {
  color: "pink",
  isOpen: false,
  open: function() {
    if (umbrella.isOpen === true) {
      return "The umbrella is already opened!";
    } else {
      umbrella.isOpen = true;
      return "Julia opens the umbrella!";
    }
   }
};
```

```
 * Programming Quiz: Umbrella (7-1)
 */

var umbrella = {
    color: "pink",
    isOpen: true,
    open: function() {
        if (umbrella.isOpen === true) {
            return "The umbrella is already opened!";
        } else {
            umbrella.isOpen = true;
            return "Julia opens the umbrella!";
        }
    },
    close: function(){
        if(umbrella.isOpen === false){
            return "The umbrella is already closed!";
        } else {
            umbrella.isOpen = false;
            return "Julia closes the umbrella";
        }
    }
};
```

Quiz: Menu Items

Create a breakfast object to represent the following menu item:

```
The Lumberjack - $9.95
eggs, sausage, toast, hashbrowns, pancakes
```

The object should contain properties for the name, price, and ingredients.

```
/*
 * Programming Quiz: Menu Items (7-2)
 */

// your code goes here
var breakfast = {
    "name": "The Lumberjack",
    "price": 9.95,
    "ingredients": ["eggs", "sausage", "toast", "hashbrowns", "pancakes"],
};
```

Quiz: Bank Accounts 1

Using the given object:

```
var savingsAccount = {
  balance: 1000,
  interestRatePercent: 1,
  deposit: function addMoney(amount) {
    if (amount > 0) {
      savingsAccount.balance += amount;
    }
  },
  withdraw: function removeMoney(amount) {
    var verifyBalance = savingsAccount.balance - amount;
    if (amount > 0 && verifyBalance >= 0) {
      savingsAccount.balance -= amount;
    }
  }
};
```

add a printAccountSummary() method that returns the following account message:

```
Welcome!
Your balance is currently $1000 and your interest rate is 1%.
```

```
 * Programming Quiz: Bank Accounts 1 (7-3)
 var savingsAccount = {
    balance: 1000,
    interestRatePercent: 1,
    deposit: function addMoney(amount) {
        if (amount > 0) {
            savingsAccount.balance += amount;
        }
    },
    withdraw: function removeMoney(amount) {
        var verifyBalance = savingsAccount.balance - amount;
        if (amount > 0 && verifyBalance >= 0) {
            savingsAccount.balance -= amount;
        }
    },
    // your code goes here
    printAccountSummary: function(){
        return "Welcome!\nYour balance is currently $"+savingsAccount.balance+" and your
interest rate is "+savingsAccount.interestRatePercent+"%.";
    }
};

console.log(savingsAccount.printAccountSummary());
```

Returns: `Welcome! Your balance is currently $1000 and your interest rate is 1%.`

Quiz: Facebook Friends

Create an object called facebookProfile. The object should have 3 properties:

1. your name
2. the number of friends you have, and
3. an array of messages you've posted (as strings)

The object should also have 4 methods:

1. postMessage(message) - adds a new message string to the array
2. deleteMessage(index) - removes the message corresponding to the index provided
3. addFriend() - increases the friend count by 1
4. removeFriend() - decreases the friend count by 1

```
/*
 * Programming Quiz: Facebook Friends (7-5)
 */

// your code goes here
var facebookProfile = {
  name: "Julia",
  friends: 100,
  messages: ["Hi!", "How are you?", "What's new?", "Let's meet in an hour!"],
  postMessage: function message(messages){
      var message="";
      return facebookProfile.messages.push(message);
    },
  deleteMessage: function(index){
     return facebookProfile.messages.splice(index, 1);
  },
  addFriend: function() {
    return facebookProfile.friends++;
    },
  removeFriend: function() {
    return facebookProfile.friends--;
    },
};
```

Quiz: Donuts Revisited

Here is an array of donut objects.

```
var donuts = [
  { type: "Jelly", cost: 1.22 },
  { type: "Chocolate", cost: 2.45 },
  { type: "Cider", cost: 1.59 },
  { type: "Boston Cream", cost: 5.99 }
];
```

Use the forEach() method to loop over the array and print out the following donut summaries using console.log.

```
Jelly donuts cost $1.22 each
Chocolate donuts cost $2.45 each
Cider donuts cost $1.59 each
Boston Cream donuts cost $5.99 each
```

```
/*
 * Programming Quiz: Donuts Revisited (7-6)
 */

var donuts = [
    { type: "Jelly", cost: 1.22 },
    { type: "Chocolate", cost: 2.45 },
    { type: "Cider", cost: 1.59 },
    { type: "Boston Cream", cost: 5.99 }
];

// your code goes here
donuts.forEach(function(message){
   console.log(message.type+" donuts cost $"+message.cost+" each");
});
```

9. jQuery

JavaScript is the most widely-used language for adding dynamic behavior to web pages. The JavaScript community contributes to a collection of libraries that extend and ease its use.

jQuery, a JavaScript library that makes it easy to add dynamic behavior to HTML elements.

Let's look at an example of how JavaScript is used to add dynamic behavior to a web page (don't worry about understanding the code).

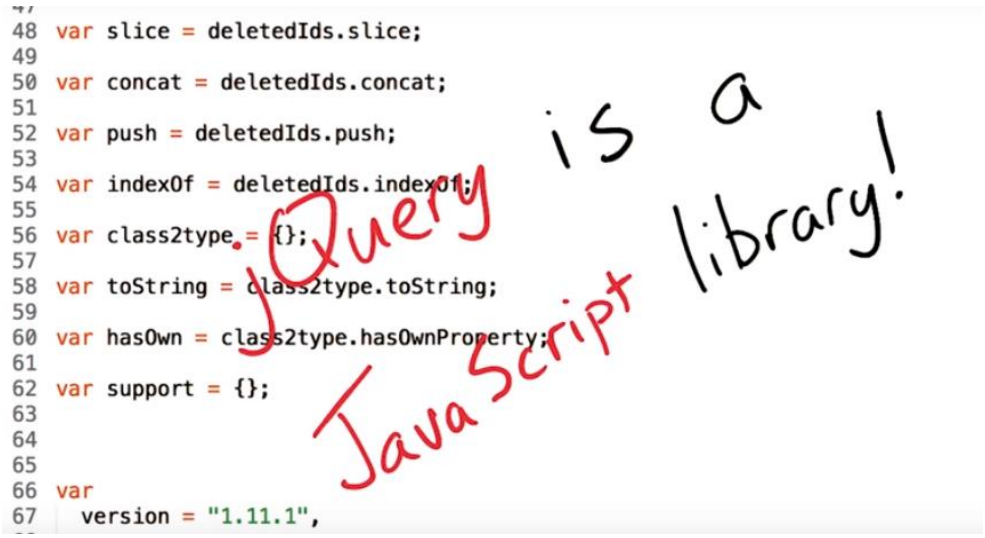
```
const login = document.getElementById('login');
const loginMenu = document.getElementById('loginMenu');

login.addEventListener('click', () => {
  if(loginMenu.style.display === 'none'){
    loginMenu.style.display = 'inline';
  } else {
    loginMenu.style.display = 'none';
  }
});
```

In this example, JavaScript is used to apply behavior to an HTML element with id login. The behavior allows a user to click a **LOGIN** button that toggles a login form.
The code below accomplishes the same behavior with jQuery.

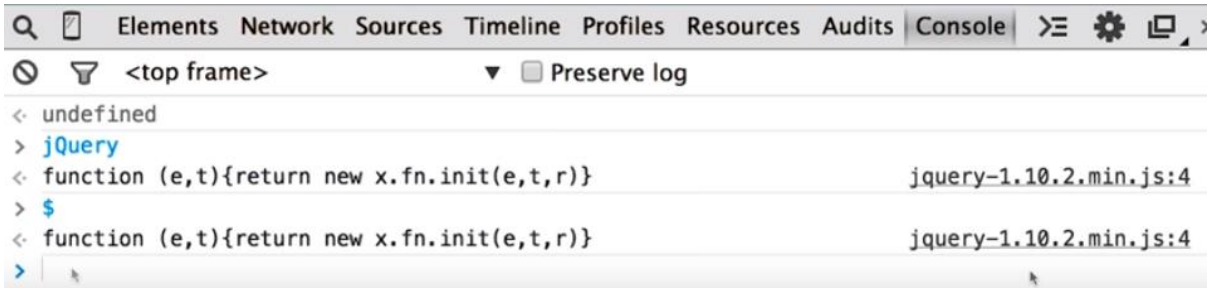
```
$('#login').click(() => {
  $('#loginMenu').toggle()
});
```

In this example, the same toggle functionality is accomplished using just three lines of code.

Over 60% of the top 100 000 visited web-sits take the advantage of jQuery simple syntax and powerful DOM manipulation methods. It is important to understand that jQuery is just JavaScript library (jQuery is not a language on its own!):

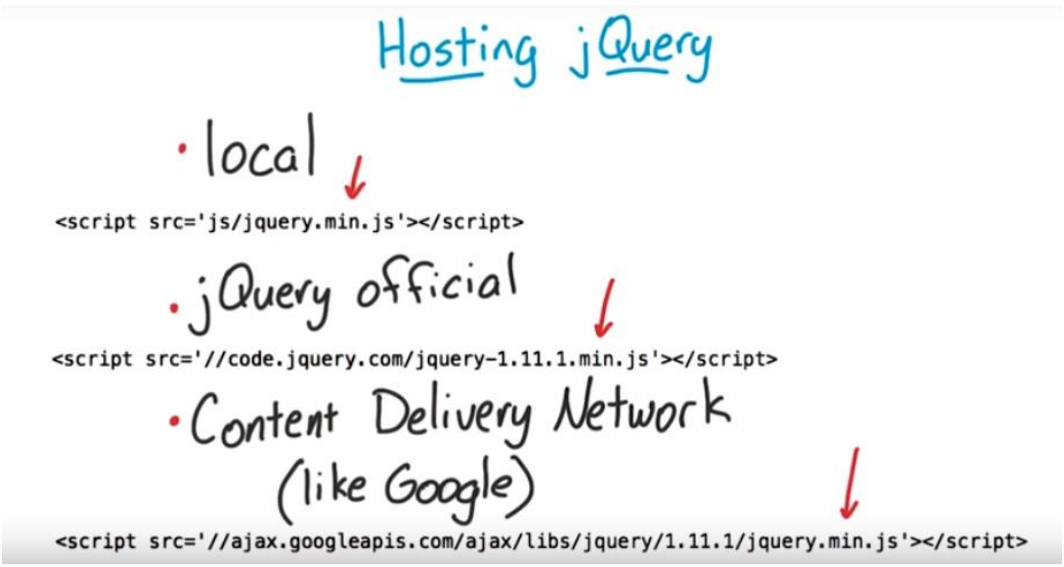


jQuery is a function and also an object.



$ and jQuery are mapped to exactly the same thing – JavaScript library.

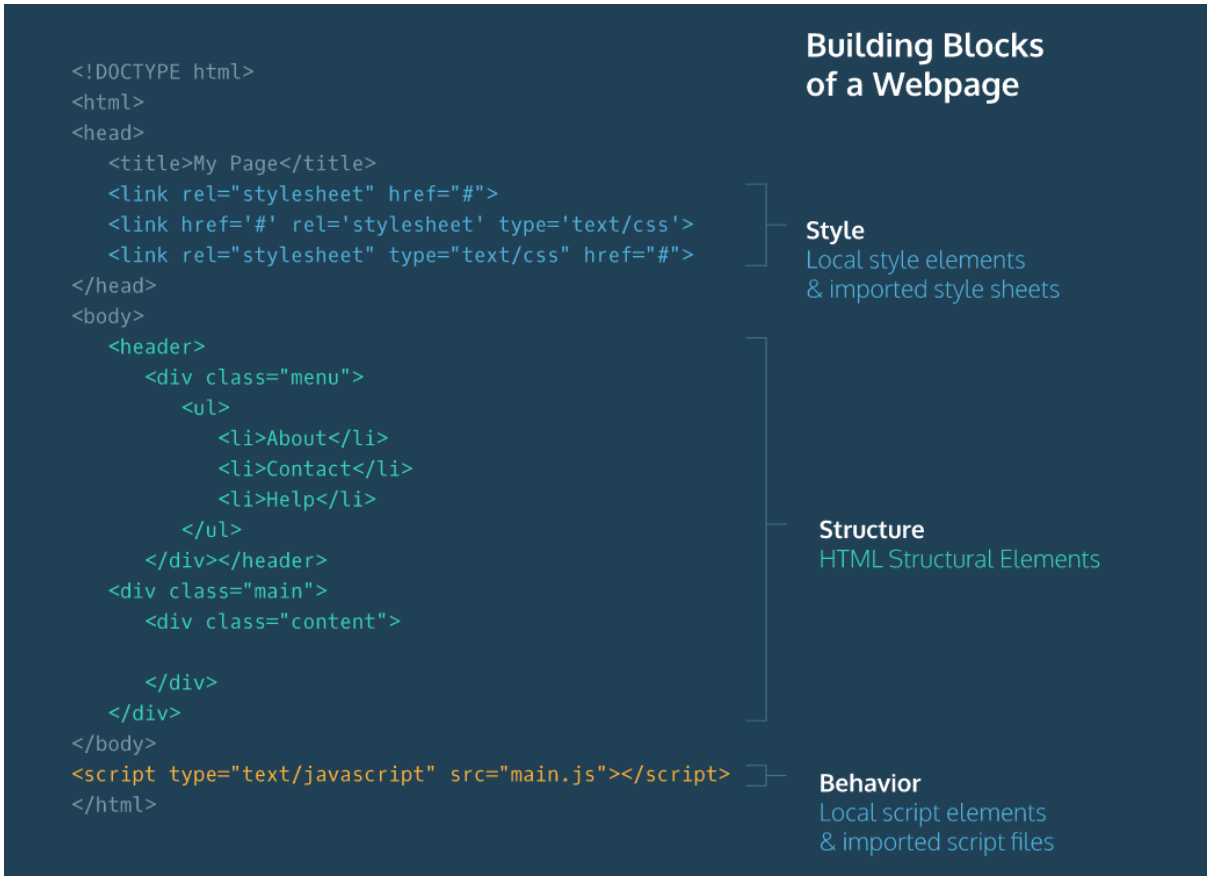
9.1. Hosting jQuery



jQuery may be added to any web-site, using <script> in HTML.

The jQuery Homepage
jQuery's CDN
jQuery hosted by Google

To use the jQuery library, index.html must load it with the other dependencies. The document is loaded from top to bottom. So the style dependencies in the <head> will load first, then the structural elements in the <body> will load next. It has become common practice to link the main JavaScript file at the bottom of the HTML document because a good deal of the content of the script will require that the dependencies, style sheets and elements exist before the browser can run the JavaScript that uses and references those things.



To include jQuery, we use a <script> tag as follows:

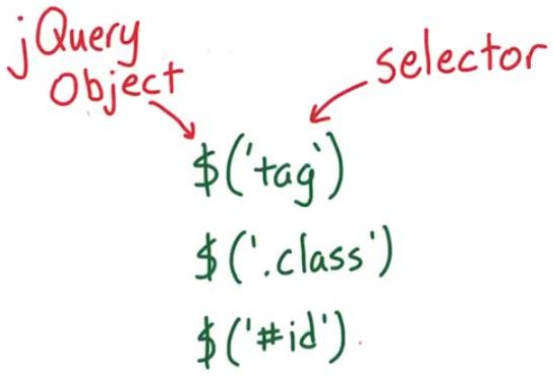
```
<script src="https://code.jquery.com/jquery-3.2.1.min.js" integrity="sha256-hwg4gsxgFZhOsEEamdOYGBf13FyQuiTwlAQgxVSNgt4="
crossorigin="anonymous"></script>
```

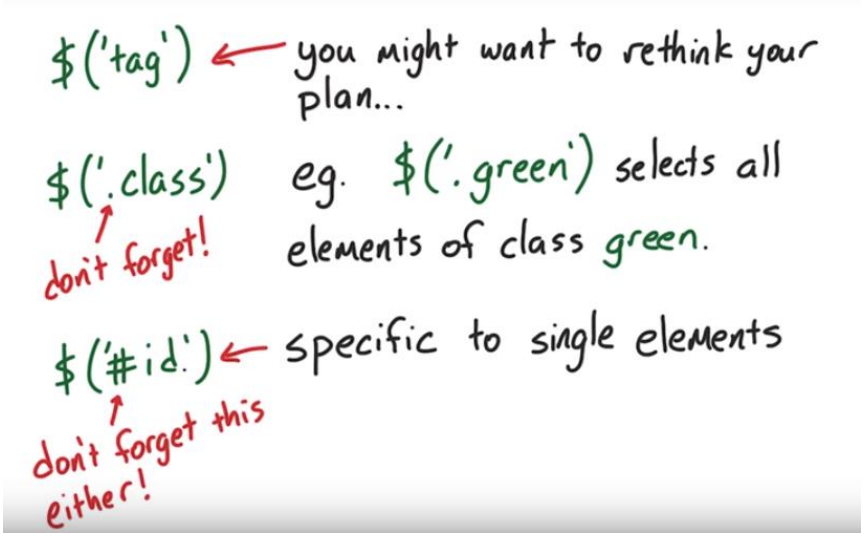
In this example, the jQuery library is loaded from the jQuery *content delivery network (CDN)*. A CDN is a collection of servers that can deliver content.
You must include the <script> tag in the HTML document before you link to a JavaScript file that uses the jQuery library. The integrity and crossorigin properties in the example ensure the file is delivered without any third-party manipulation.

9.2. jQuery Selectors

Documentation on selectors

You can use jQuery to select a collection of DOM elements based on tag name, class or id.





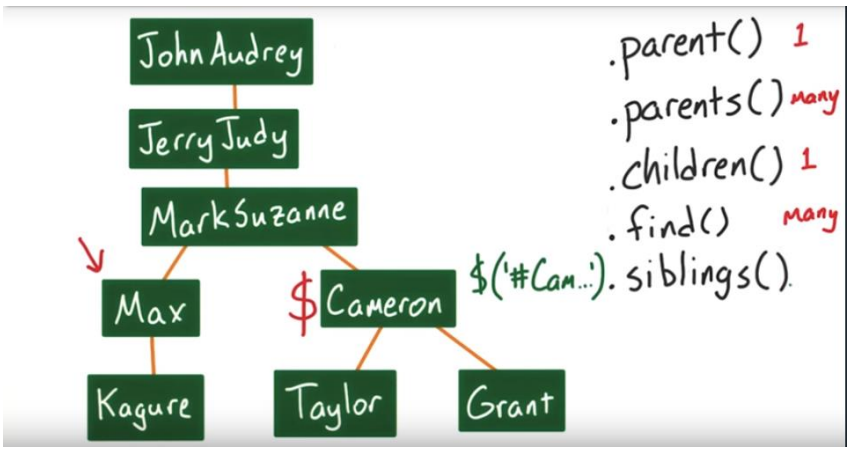
Typically, we pass a string into $() to target elements by id, class, or tag. Once targeted, we can use . notation to attach a handler method that triggers a callback function.

Let's consider how we can target elements by class. We can reference elements by class name with the following syntax:

```
$('.someClass').handlerMethod();
```

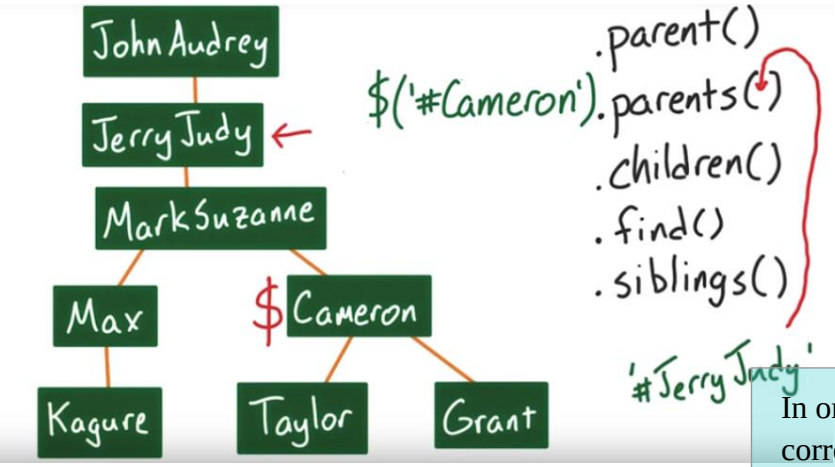
In the example above, every element with a class of 'someClass' is targeted. Note, we prepend the class name with a period (.someClass). Then, we call the .handlerMethod() on all of the referenced items.

9.3. Traversing the DOM



Documentation on traversal methods

jQuery makes it easy to target HTML elements by tag name, class, and id. Traversing the DOM – allows us to target elements based on their position relative to other elements:
$('#Cameron').parent() - would select one immediate parent of Cameron, i.e. MarkSuzanne
$('#Cameron').parents() - should be used to select Cameron's parents, grandparents, etc. It is possible to filter a specific parent by passing in another selector to the parents method



In order to use methods correctly, and to find the needed method, always refer to jQuery API Documentation

$('#Cameron').children() – would select Cameron's immediate children, i.e. Taylor and Grant
$('#Cameron').find() – would select Cameron's children's children
$('#Cameron').siblings() – would select Cameron's siblings (meaning that the elements have the same parent), i.e. Max.

9.4. jQuery tricks

The JavaScript language represents an infinite supply of Lego blocks — the possibilities are endless but time-consuming. The pre-made Lego structures are like jQuery methods. You can use these methods to add dynamic behavior, such as .hide(), .show(), .fadeIn(), .fadeOut() etc., to HTML elements.

The first example of the chapter took twelve lines of JavaScript, but was achieved with only three lines of jQuery's .click() and .toggle() methods.

.ready()

The jQuery library makes it quick and easy to add visual effects and interactivity to your web page. However, a web page must be rendered in a user's browser before it's possible to have any dynamic behavior. To solve this problem, we will use our first jQuery *method*.
The jQuery .ready() method waits until the HTML page's DOM is ready to manipulate. You should wrap all JavaScript behavior inside of the .ready() method. This will make sure the web page is rendered in the browser before any jQuery code executes.

```
$(document).ready(function() {
  /* All your code */
});
```

Or

```
$(document).ready(() => {
  /*All your code*/
});
```

In the example above, the .ready() method waits until the browser finishes rendering the HTML document before triggering a callback function. We will write all of our jQuery behavior inside this callback function.

.toggleClass()

Documentation on .toggleClass()

Add or remove one or more classes from each element in the set of matched elements, depending on either the class's presence or the value of the state argument.

Example:

```
var featuredArticle = $('.featured');
featuredArticle = featuredArticle.toggleClass();
```

.next()

Documentation on .next()

Get the immediately following sibling of each element in the set of matched elements. If a selector is provided, it retrieves the next sibling only if it matches that selector.

Example:

```
var article2, article3;

article2 = $('.featured');
article2 = article2.toggleClass('featured');

article3 = article2.next();
article3 = article3.toggleClass('featured');
```

.attr()

Documentation on .attr()

Get the value of an attribute for the first element in the set of matched elements.

Example (using .attr change the href of a tag to be equal to 1):

```
var navList;
navList = $("ul li a").first().attr("href","#1");
```

or

```
var navList, firstItem, link;
navList = $(‘.nav-list’);
firstItem = navList.children().first();
link = firstItem.find(‘a’);
link.attr(‘href’, ‘#1’);
```

.css()

Documentation on .css()

At this point, you likely know that jQuery can link user events to dynamic effects on a web page. One of the most powerful toolsets in jQuery allows you to edit CSS property values. With these tools, you can change multiple visual aspects of an element at once.

To modify CSS properties of an element, jQuery provides a method called .css(). This method accepts an argument for a CSS property name, and a corresponding CSS value.

It's syntax looks like:

```
$('.example-class').css('color', '#FFFFFF');
```

Let's break the example above into two pieces:

- We call the .css() method on .example-class. The first argument is the CSS property we want to change. In this case, that's 'color'.
- The second argument specifies the new value for the given property in the first argument. In this case, the .css() method changes the color of .example-class elements to '#FFFFFF'.

Notice, both of the inputs to the .css() method are strings.

Example (change the font-size of all the article-items to 20px):

```
var articleItems;

articleItems = $('.article-item');
articleItems.css('font-size', '20px');
```

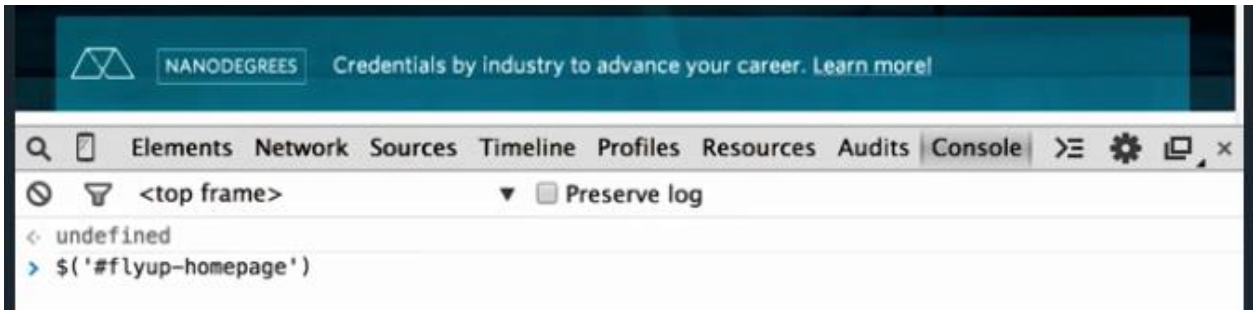
.html() .text() .val()

Documentation on .html()
Documentation on .text()
Documentation on .val()

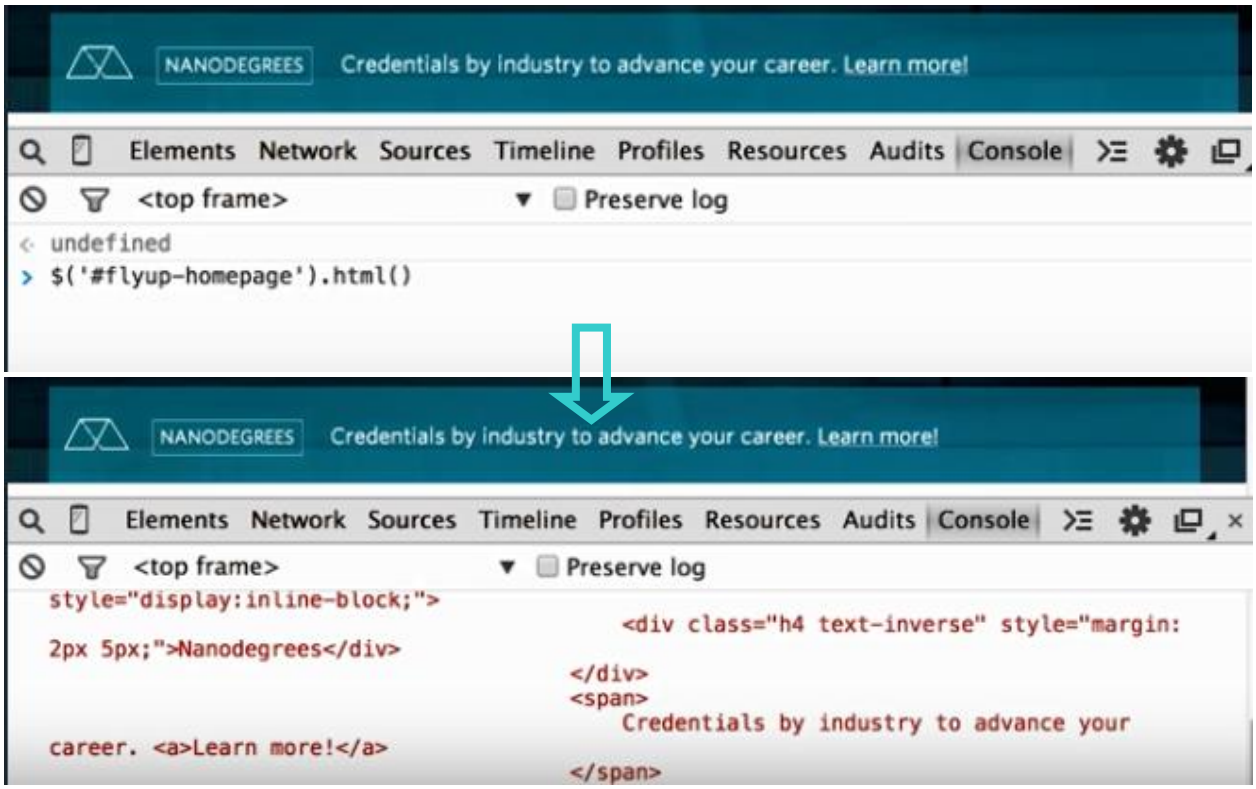
The two best options for finding all of the data between tags are .html() and .text()

Example:

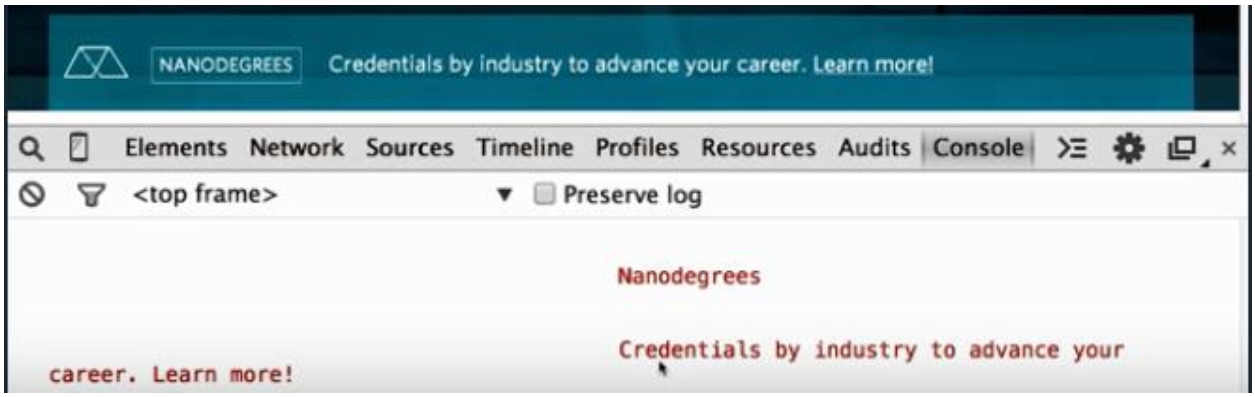
We have selected the element with the id of #flyup-homepage, it has other elements inside of it:



If you run .html() in this selected element, you'll see the html of everything inside of the #flyup-homepage, including all of the tags, classes and attributes.

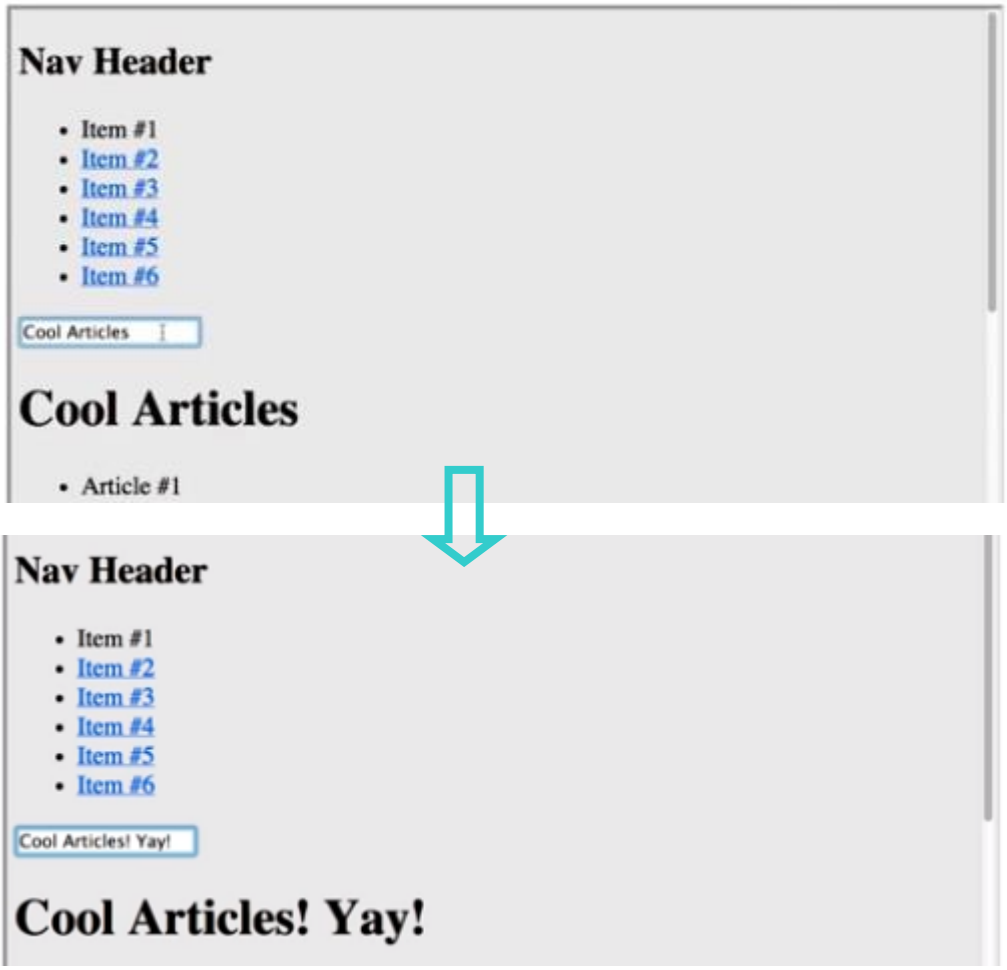


But if you run .text() on a selection `$('#flyup-homepage').text()`
You will see that jQuery has stripped out all of the html tags, and only return the text contained within.



Example (use jQuery's val method to make live changes to the 'Cool Articles' <h1>)

```
$('#input').on('change', function() {
   var val;
   val = $('#input').val();
   $('#input').next('h1').text(val);
});
```



.remove()

Documentation on .remove()

With jQuery you may create and delete DOM elements.

Example (remove the <ul> from the first article item):

```
var articleItems, ul;

articleItems = $('.article-item').first();
ul= articleItems.children('ul');
ul.remove();
```

```
1 var articleItems, ul;
2
3 articleItems = $('.article-item');
4
5 ul = articleItems.find('ul');
6
7 ul.remove();
```

.append() .prepend() .insertBefore() .insertAfter()

Documentation on .append()
Documentation on .prepend()
Documentation on .insertBefore()
Documentation on .insertAfter()

```
var div = document.createNode('div');
div.innerHTML = "Hello Udacity";
var parent = document.querySelector('#parent');
parent.appendChild(div);
```

It can be tricky to add elements to the DOM with JavaScript, because you first have to create d DOM node, add data to it, find a parent it, and finally add that node as a child to that parent. Each step happens pretty much independently:

With jQuery you can create new DOM nodes and simultaneously add them to the document with one simple method.

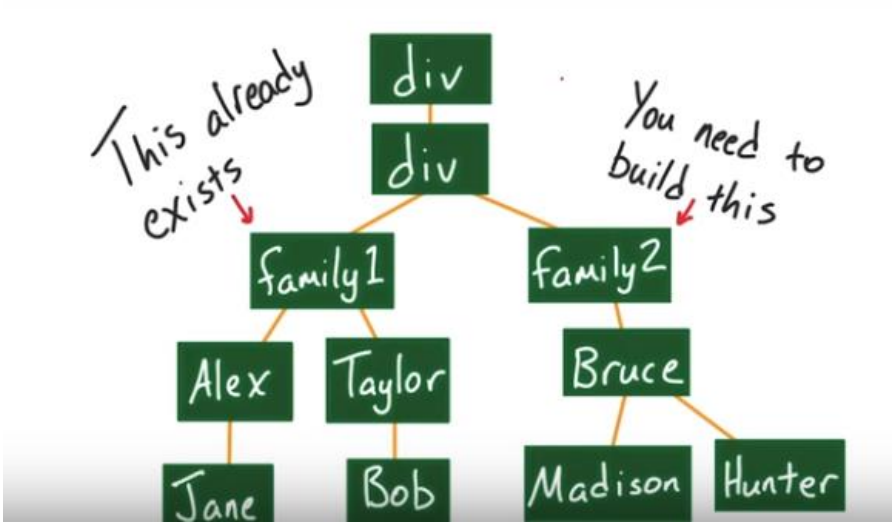
.append() and .prepend() add children to an element.

```
1 var firstArticleItem;
2
3 firstArticleItem = $('.article-item').first();
4
5 firstArticleItem.append('<img src="http://placepuppy.it/200/300">');
6
7 firstArticleItem.prepend('<img src="http://placepuppy.it/200/300">');
```

.append() - adds the new element as **the last child** of the selected item; while .prepend() would add it **as the first child.**

.insertBefore() and .insertAfter() create a sibling to the selected element.

Example (new DOM family tree):

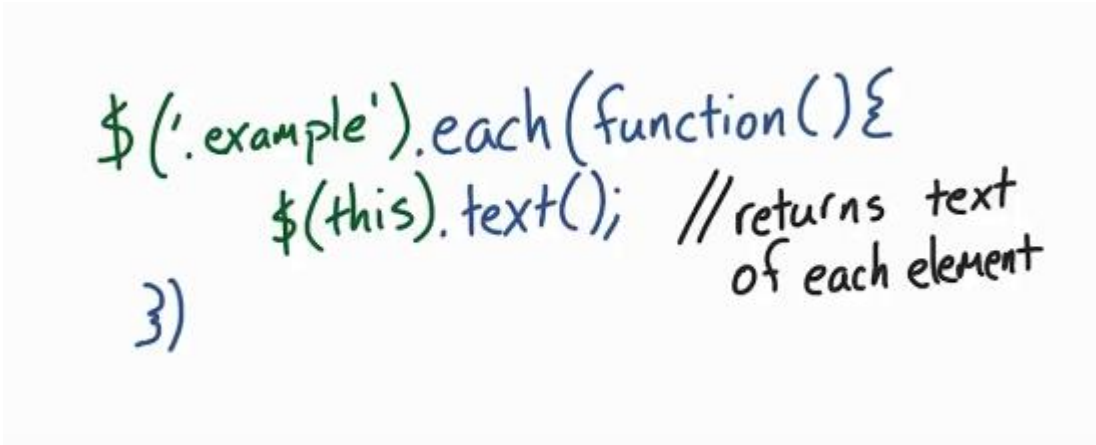


```
var family1, family2, bruce, madison, hunter;

family1 = $('#family1');
family2 = $('<div id="family2"><h1>Family 2</h1></div>');
bruce = $('<div id="bruce"><h2>Bruce</h2></div>');
madison = $('<div id="madison"><h3>Madison</h3></div>');
hunter = $('<div id="hunter"><h3>Hunter</h3></div>');

family2.insertAfter(family1);
family2.append(bruce);
bruce.append(madison);
bruce.append(hunter);
```

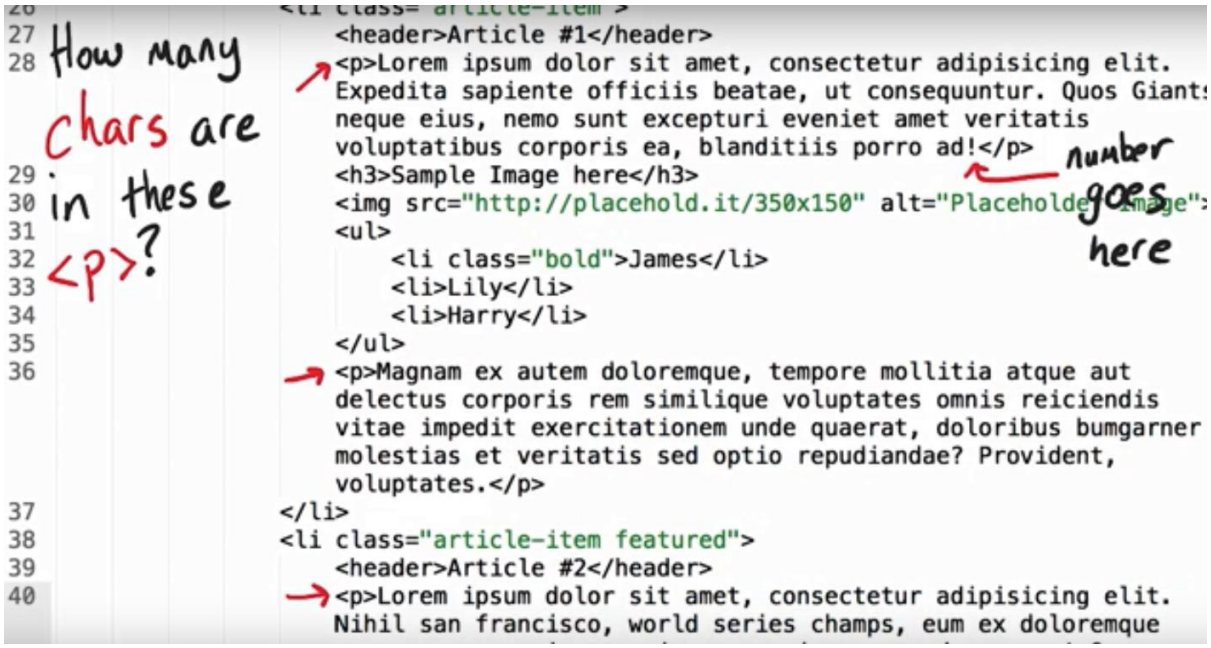
.each()

Documentation on .each()

The .each() method lets you to iterate through a jQuery collection and run a function, against each DOM element.



Example (jQuery's each() method to iterate through the <p>s, calculate the length of each one, and add each length to the end of each <p>):

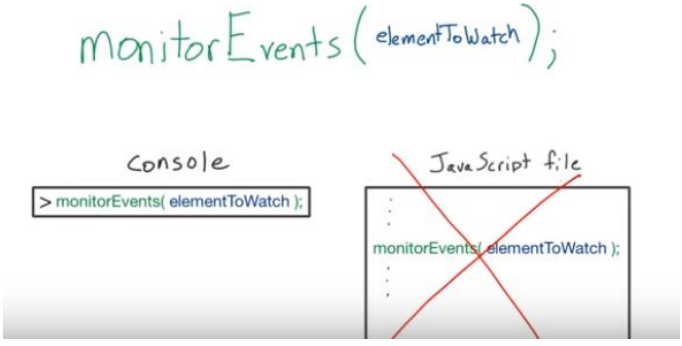



```
$('p').each(function WordCount(p) {

  var countWords = $(this).text().length;
  $(this).append(countWords);

});
```

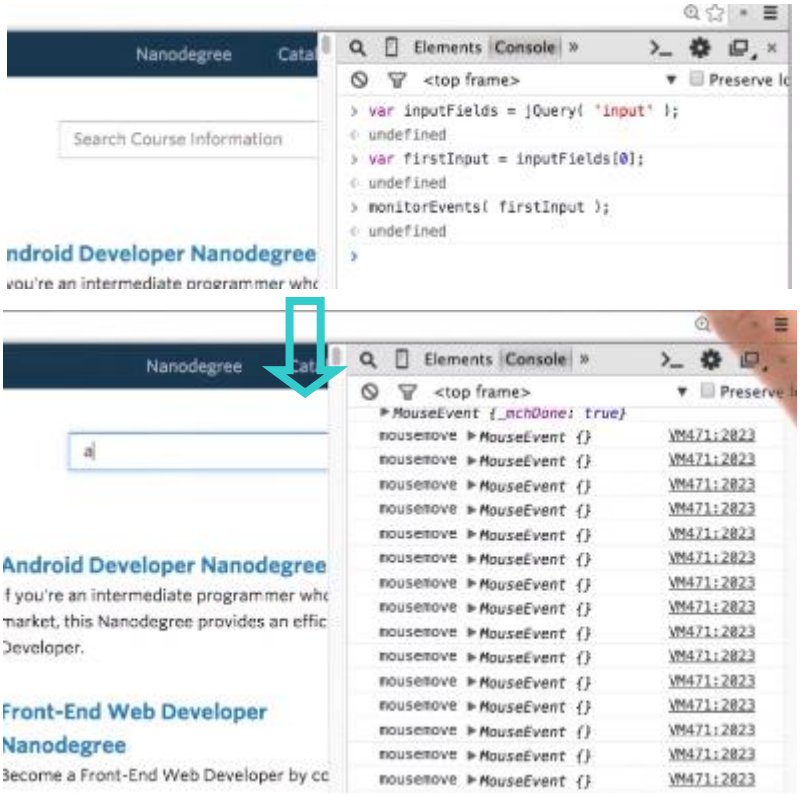
```
function numberAdder () {
    var text, number;

    text = $(this).text();

    number = text.length;

    $(this).text(text + " " + number);
}

$('p').each(numberAdder);
```

9.5. Mouse Events

What are browser events? Every time you move your mouse, click on a link, submit a form, or do really anything, your browser makes an announcement of the action you just took. What the browser is actually doing? monitorEvents function watches the element for events, and log them all out.



monitorEvents() Documentation



There are three items you need in order to listen for events and react to them:

1. the target element to listen to
2. the event we want to react to
3. the actions to take in response

```
1 $( '#my-input' ).on( 'keypress', function() {
2     console.log( 'The event happened!' );
3 });
```

So, the jQuery library provides a collection of methods that serve one of two purposes.

- To listen for an event — an event (e.g. clicking a button) is something the user does to trigger an action.
- To add a visual effect — a visual effect (e.g. a drop-down menu appearing when a user clicks a button) is something that changes the appearance of the web page. Events are often responsible for triggering a visual effect.

In this lesson, you will learn how to link a user event to a visual effect using *event handlers*. There are two parts to an event handler: an *event listener* and a *callback function*.

- An *event listener* is a method that listens for a specified event to occur, like a click event.
- A *callback function* is a function that executes when something triggers the event listener.

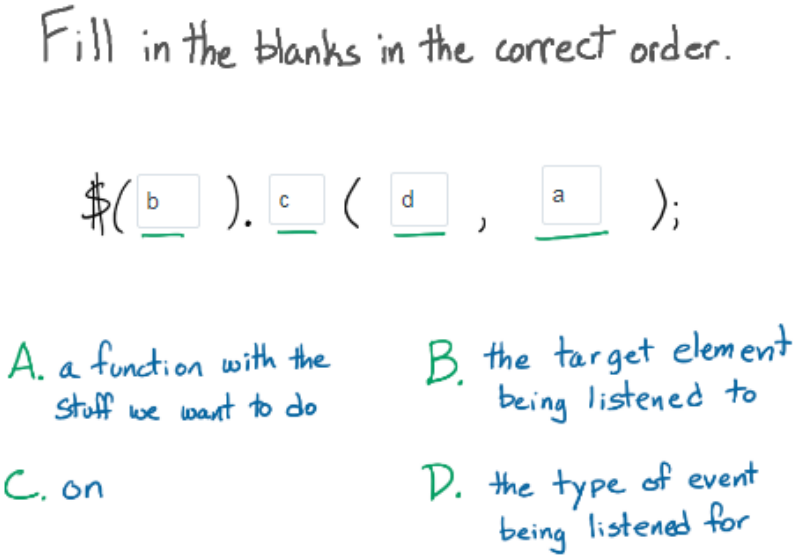
Both the event listener and callback function make up an event handler.

In code, this looks like:

```
$('.example-class').on('click', () => {
  // Execute code here when .example-class is clicked
});
```

Let's consider the example above step-by-step:

- `$('.example-class')` selects all HTML elements with a class of `example-class`.
- The `on('click')` method is the event listener. It checks if the user has clicked an `.example-class` HTML element.
- The second argument to `.on()` is a callback function. When a `'click'` occurs on an `example-class` element, this function executes.



Example:
Use jQuery to set up an event listener. Your event listener must:

1. listen to the #my-button element
2. listen for a `click` event
3. perform the following actions when the button is clicked:
   a. remove the #my-button element from the DOM
   b. add the `success` class to the body

```
$('#my-button').on('click',function(){
   $('#my-button').remove();
   $('body').addClass('success');
});
```

The Event Object

Reacting to events often requires knowledge about the event itself, so this is a quick breakdown of the event object which gets passed to an event listener's callback.

Remember that the target element calls the callback function when the event occurs. When this function is called, jQuery passes an event object to it containing information about the event. This object holds a ton of useful information that can be used in the body of the function. This object, which is usually referenced in JavaScript as e, evt, or event, has several properties that you can use to determine the flow of your code. Try logging the object to see what's available:

```
$( 'article' ).on( 'click', function( evt ) {
   console.log( evt );
});
```

You should notice a target property. The target property holds the page element that is the target of the event. This can be extremely useful if an event listener has been set up for a number of elements:

```
$( 'article' ).on( 'click', function( evt ) {
   $( evt.target ).css( 'background', 'red' );
});
```

In the example above, an event listener is set up for every article element on the page. When an article is clicked an object containing information about the event is passed to the callback. The evt.target property can be used to access just the clicked on element! jQuery is used to select just that one element from the DOM and update its background to red.
The event object also comes in handy when you want to prevent the default action that the browser would perform. For example, setting up a click event listener on an anchor link:

```
$( '#myAnchor' ).on( 'click', function( evt ) {
   console.log( 'You clicked a link!' );
});
```

Clicking on the #myAnchor link will log the message to the console, but it will also navigate to that element's href attribute - potentially redirecting to a new page. The event object can be used to prevent the default action:

```
$( '#myAnchor' ).on( 'click', function( evt ) {
   evt.preventDefault();
   console.log( 'You clicked a link!' );
});
```

In the code above, the evt.preventDefault(); line instructs the browser not to perform the default action.
Other uses include:

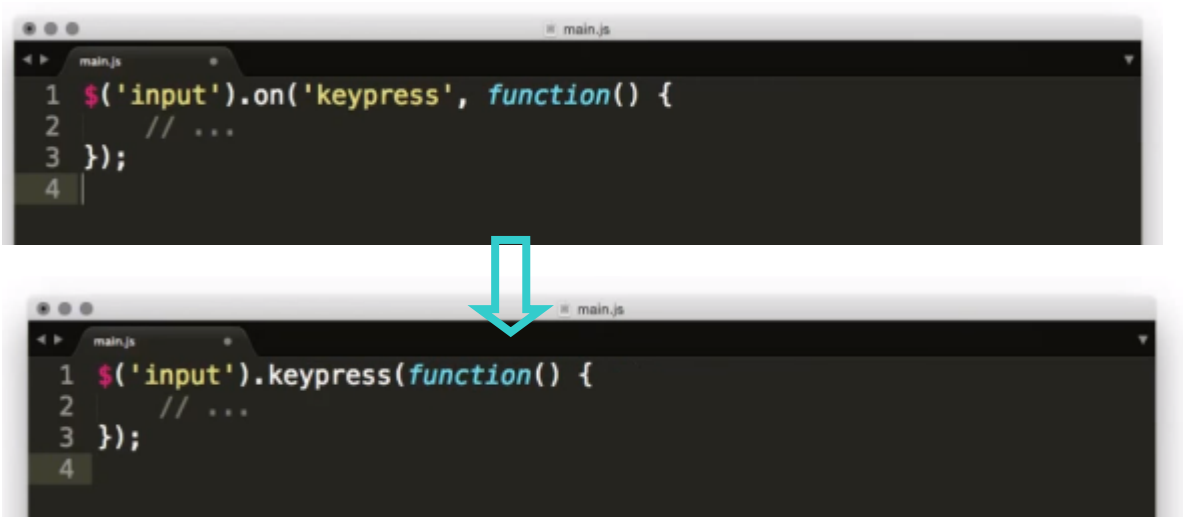
- event.keyCode to learn what key was pressed - invaluable if you need to listen for a specific key
- event.pageX and event.pageY to know where on the page the click occurred - helpful for analytics tracking

- event.type to find what event happened - useful if listening to a target for multiple events.

jQuery's Event Object
event.target property
DOM Level 3 Events

The Convenience Method

So far you have been setting up event listeners using jQuery's .on method. jQuery also has some convenience methods that you can use.



http://api.jquery.com/category/events/

Event Delegation

There is one more way to set up event listener with jQuery.

The jQuery event listener examples we've been looking at so far select the target item(s) using jQuery and then attach an event listener to that target directly. But what about when the target doesn't exist yet? This can happen in a lot of situations. For example, if you have a list of items, and you want to listen to clicks on any of them, what happens if you add an extra list item after your page is done?

Be careful when setting up an event listener and then creating the target item afterwards. For example:

```
$( 'article' ).on( 'click', function() {
   $( 'body' ).addClass( 'selected' );
});

$( 'body' ).append( '<article> <h1>Appended Article</h1> <p>Content for the new article </p> </article>' );
```

Clicking on the "appended" article will not add a class to the body because the "appended" article was created after the event listeners were set up. When we targeted the 'article', it didn't exist yet, so jQuery added the click listener to all ZERO of our articles!

But there is a way to make this scenario work by using Event Delegation. We'll listen to events that hit a parent element, and pay attention to the target of those events. Event Delegation with jQuery uses the same code we've been using, but passes an additional argument to the "on" method.

```
$( '.container' ).on( 'click', 'article', function() { … });
```

...this code tells jQuery to watch the .container element for clicks, and then if there are any, check if the click event's target is an article element.
Another advantage in using Event Delegation is that you can use it to consolidate the number of event listeners. For example, what if you had 1,000 list items on a page:

```
<ul id="rooms">
   <li>Room 1</li>
   <li>Room 2</li>
         .
         .
         .
   <li>Room 999</li>
   <li>Room 1000</li>
</ul>
```

The following code would set up an event listener for each 1,000 event listeners - one for each list item...that's 1,000 event listeners!

```
$( '#rooms li' ).on( 'click', function() {
   ...
});
```

Alternatively, we can use jQuery's event delegation to set the event listener on just one element (the ul#rooms) and check if the target element is a list item;

```
$( '#rooms' ).on( 'click', 'li', function() {
   ...
});
```

To find out more, check out jQuery's page on **Event Delegation**.