



# Front End Developer

# Table of Contents

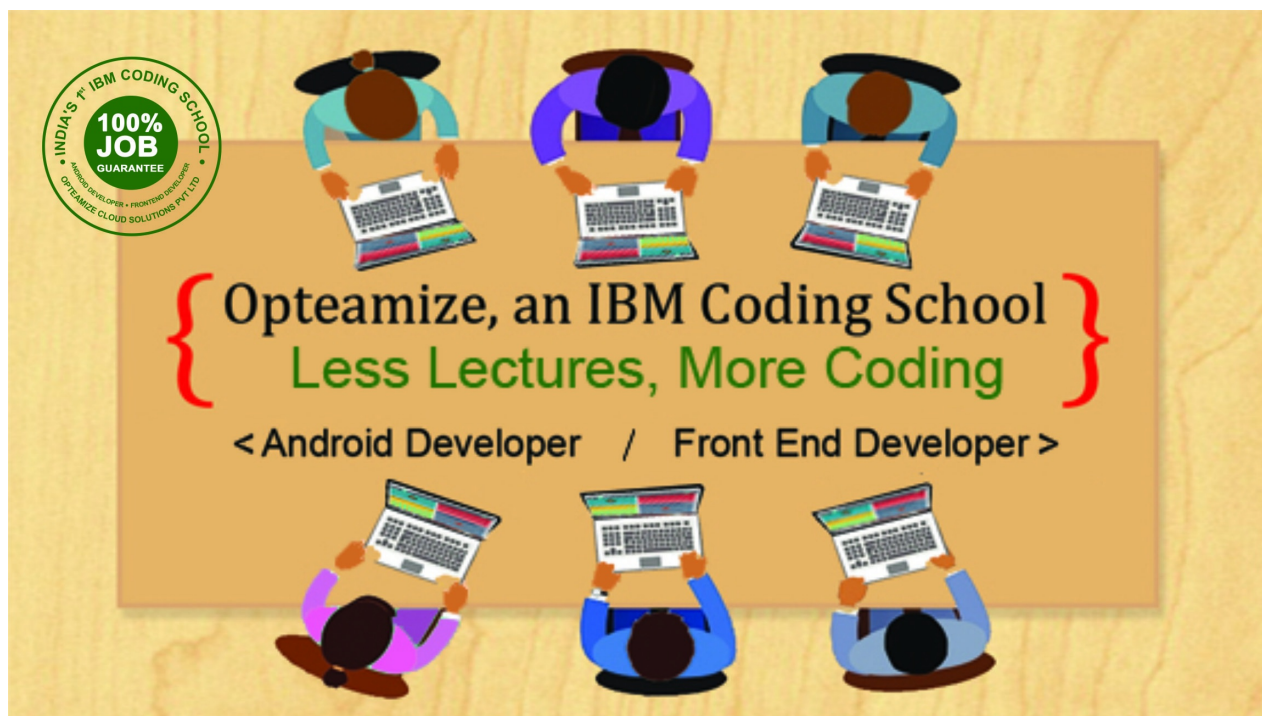
Introduction	0
Coding School Enrolment Document	0.1
How to use CourseBook	0.2
HTML and CSS basics	1
HTML and CSS objectives	1.1
Text editor	1.2
HTML Layouts	1.3
HTML Tags	1.4
Indentation and spacing	1.5
Block elements	1.6
Block elements practice	1.7
Inline elements	1.8
Inline elements practice	1.9
Styling text	1.10
Styling text practice	1.11
Styling with classes	1.12
Classes practice	1.13
Organizing with divs and spans	1.14
Div and span practice	1.15
Laying out with floats	1.16
Floats practice	1.17
The box model	1.18
Box model practice	1.19
Cascading	1.20
Cascading practice	1.21
Applying other people's styles	1.22
Theme practice	1.23
Grid system and responsive design	1.24
Grid, theme, and media query practice	1.25
HTML and CSS code review	1.26

HTML and CSS Project	1.27
More HTML and CSS resources	1.28
Introduction to Git	2
README template	2.1
Git clone	2.2
Git with Teams	2.3
GitHub Pages	2.4
Git cheat sheet	2.5
JavaScript basics	3
JavaScript basics objectives	3.1
Arithmetic	3.2
Arithmetic practice	3.3
Variables	3.4
Variables practice	3.5
Methods	3.6
Methods practice	3.7
Strings	3.8
String practice	3.9
Functions	3.10
Functions practice	3.11
Writing functions	3.12
Practice writing functions	3.13
JavaScript Branching and Looping Objectives	3.14
Branching	3.15
Branching Practice	3.16
More Branching	3.17
More Branching Practice	3.18
Arrays	3.19
Arrays Practice	3.20
JavaScript Reserved Words	3.21
Looping	3.22
Looping Practice	3.23
Looping With For	3.24
For Loop Practice	3.25

---

Debugging In JavaScript	3.26
Javascript Code Review	3.27
JavaScript JSON	3.28
jQuery	4
jQuery Objectives	4.1
Introducing jQuery	4.2
jQuery Practice	4.3
Simple Effects	4.4
jQuery Effects Practice	4.5
DOM Manipulation and Traversal	4.6
DOM Manipulation and Traversal Practice	4.7
Forms	4.8
Forms Practice	4.9
Attributes	4.10
jQuery Attributes Practice	4.11
jQuery Code Review	4.12
More jQuery Practice	4.13
Entry to Level 2 Test	5

# Front End Developer



So you want to learn how to code? Great decision! What programming language should you start learning first?

We at **Opteamize, an IBM Coding School** teach programming and our platform is Javascript-centric. We feel that most of our students will have great success with **JavaScript**. Let us explain why you should start with JavaScript.

## **It's easy to get started**

JavaScript is the easiest language to start learning with since there's nothing to install or complex "runtime" environments to configure.

## **It's everywhere**

JavaScript is the only language that runs in web browsers like Chrome, Internet Explorer, Safari and Firefox and because the web is everywhere, JavaScript is everywhere.

## **One true language**

Whether you're using MongoDB for data storage, Node.js for your web backend, jQuery/Angularjs for your web frontend, d3.js for visualization, at all levels you'll be thinking/writing/mastering JavaScript

## **Excellent Salary**

Evidence from major job sites suggests that programmers good at advanced JavaScript will be commanding some of the highest salaries in the years ahead compared to other web-

based languages.

### **Only Language you will need**

What's great about JavaScript in the web world is that it is the only language that you have to learn. No other language is required if all you want to create is exciting web applications.

### **Still not convinced? Here are more reasons:**

[The first programming language you should learn in JavaScript](#)

[6 reasons to learn JavaScript](#)

[10 reasons why you will find JavaScript as the best programming language](#)

# Opteamize, an IBM Coding School Enrolment Document

This Job Guarantee Agreement is signed on the xxth day of Month 2016 between Opteamize Cloud Solutions Private Limited, a Private Limited Company with its offices located at 20/3B Swamy Apartments, Ganpatraj Nagar Main Road, Virugambakkam, Chennai 600092 hereby referred to as Opteamize and student name..... S/o D/o aged 23 years address ..... hereby referred to as student

Whereas Opteamize is in the business of software development, educational technology and software consulting and is offering its software platform to be used to deliver software training to the student by its service partner ...location .....

Opteamize & student herewith agree to the following terms and conditions for eligibility to the Job Guarantee Program run by Thinksoft R&D using the Opteamize Coding Platform.

## General Enrolment Conditions

- You shall be prepared to dedicate yourself entirely to the learning being imparted: 6 hours on weekends (Sat/Sun) for first 9 weekends & 10 hours a day, 5 days a week for last 4 weeks.
- You will need to bring your own laptop preferably with recommended configuration - 4 GB RAM, Original Windows or Linux OS, Working Video & Sound Cards.
- Before the class begins, you will complete several lessons in CourseBook which can be accessed by you. Depending on your experience and proficiency, you should expect to spend 30 to 40 hours on these lessons.
- You will come to class as per calendar, arriving on time and staying until class is over (except for one hour lunch break). \*
- When we assign homework, you will complete it on time. There is typically 30-45 minutes of homework each evening and weekend.
- During class, you will not use your cell phones, tablets, laptops, and any other electronic devices for personal purpose or to check email, Facebook, Whatsapp or any other website that does not relate to your class.
- You shall update your personal and professional profile on the Opteamize Platform regularly and accurately.

- You shall not share your login, course content or other course related material with any 3rd party.

### **Job Guarantee Conditions**

- Attendance will be taken at the beginning of each class. Failure to get 80% attendance may result in not being allowed entry into Job Guarantee program.
- You shall clear Level 1 and Level 2 Examinations in a maximum of 2 attempts to gain eligibility into Job Guarantee program. Pass marks for both these Examinations will be 60%.
- You shall complete Live Project successfully scoring atleast 70% marks.
- You shall submit original copies of your Marks Sheet and/or Degree Certificate with us for safekeeping during the duration of Job Guarantee program. The original copies shall be returned upon full payment of any pending fees.
- You shall be allowed a maximum of 4 personal interviews within a period of 3 months after graduation. You shall attempt to clear and join one of those companies.
- You will agree to pay all fees promptly and as per agreed terms and conditions Option (a) Full Payment (b) Partial Payment + 15% of Annual CTC.
- You shall agree to pay 7.5% of the annual CTC within 7 days of receipt of offer letter. Balance 7.5% shall be paid within 30 days after payment of first instalment.
- No refund in partial or full is payable to you if you fail to clear and/or attend any of the 4 interviews arranged for you.
- Failure to comply with any of above General Conditions & Job Guarantee Conditions shall result in not being eligible for Job Guarantee program.

### **Our Commitment to you**

- Our support team shall support you through email, Chat, phone call, forum etc. within a reasonable time frame.
- During class time, your instructors will be available as much as possible to answer your questions, review your code, help you with problems, and guide you through the process of learning software programming.
- Once a week, one of your instructors will have a one-on-one session with you.
- We shall strive to provide you atleast 99.5% uptime of the Opteamize Coding Platform. However we cannot guarantee the same as it is dependent on 3rd party hosting services.
- If classes get canceled due to unfavourable conditions (like weather, power cuts etc.) then we will try to compensate with alternate class.



- We will do our best to help you find a job but we cannot guarantee how long it will take you to find a job, or what salary you will receive.
- Interviews shall be relevant to the Android or Front End Developer job roles envisaged for you based on the course chosen and graduated from.
- We feel a personal obligation to help you find a job you love, and as the reputation of Opteamize depends on your success in the job market, it is in our best interest to do whatever we can to help you succeed.

All the best!

Signature of Student      Signature of Parent

# How to use Course Book

Embedded Video: <https://www.youtube.com/watch?v=UQ-HD5YVRIA?rel=0>

## Curriculum and Levels

Hi! Welcome to Opteamize, an IBM Coding School. This is a step-by-step guide to take you from zero experience to Front End Developer in 3 months, and contains everything you need, including lessons and practice exercises. The content you see here is the same curriculum at Epicodus, a Coding School in Portland, OR (USA). Thank you Michael for agreeing to share the curriculum with Opteamize thus benefiting hundreds of Indian students.

### Curriculum

The curriculum has a strong focus on all of the skills required of Front End Developers today and is crafted to simulate a real job environment, exposing students to broken tests, incomplete documentation and other surprises that give them experience with the sorts of real-world challenges that many Engineers only encounter late in their careers.

If learning how to program sounds a bit overwhelming, don't worry! The curriculum is broken down so that each step along the way is small and manageable. All you need to do is follow along and do the exercises. Note: if you don't do the exercises, all of this is pointless. **You can't learn to program by watching somebody else. You have to actually do it yourself.**

Check out the table of contents to see an overview of all the sections.

### Level 1 - Technical

Level 1 is designed to take you from zero experience to being able to create a basic web page that displays information with the presentation language HTML and style it with CSS. Understand the basics of Javascript and get a good understanding of jQuery.

### Level 2 - Technical

By the second level of the course, students would have honed the ability to thrive within the constantly shifting software engineering landscape, frequently working on projects based on technologies not explicitly covered in our curriculum. Get a better understanding of Javascript and discover the wonderful framework created by Google Engineers, Angularjs.

### **Level 3 - Career Level**

The program culminates with student live projects and soft skills training, so that students have the ability to articulate their skills to employers and get placed at first attempt.

Okay, let's get going; All the best!

Support Team

Opteamize, an IBM Coding School

Chennai, Bengaluru & Coimbatore.

044-22282472

[www.opteamize.in](http://www.opteamize.in)

[info@opteamize.in](mailto:info@opteamize.in)

# HTML and CSS basics

- HTML is an abbreviation for Hyper Text Markup Language.
- CSS is an abbreviation for Cascading Style Sheets.

HTML is actually used to create the webpage and the content that it has, like the images and writing. CSS is used to design the webpage and tell the writing and images to be in a certain place, like an image on the top of the page, another in the center etc..

**Anything on the internet can be modified, deleted or added just with the use of HTML and CSS!**

# HTML and CSS basics objectives

Welcome to Introduction to Programming!

In this Level, we will be learning the building blocks of making responsive web pages. You will be working with the following tools:

- Git
- Command line/Terminal
- Markdown
- HTML
- CSS
- Bootstrap
- JavaScript Console

At the end of this Level, you should be able to:

Navigate the command line to access and create directories and files

Use Git to track the creation of static webpages

Create a static webpage using both Markdown and HTML

Style a static webpage using custom CSS rules in a linked stylesheet

Apply Bootstrap rules and classes

Adapt a webpage for responsive design

# Text editor

Embedded Video: <https://www.youtube.com/watch?v=bplRxHNQzYM?rel=0>

The first thing you need to do is open a text editor. You shouldn't use a word processor like Microsoft Word for this. Text editors are different from word processors because they just display raw, plain text - you can't apply any styles like center-justification or font size. This is perfect for programming, where styles are meaningless and just get in the way.

If you're using a Mac, your computer comes with a text editor called TextEdit that's in your Applications folder. There's just one thing you need to change to make it work well for our purposes: open Preferences and switch Format from Rich text to Plain text. Then, if a new document window is open, close it and open a new one. Now you have a text editor!

If you're using Windows, your computer comes with a text editor called Notepad. To open it, click the Start menu, All Programs, Accessories, and then Notepad.

I prefer to use a text editor called Sublime Text. It has a lot of features that make programming easier. Sublime Text is free to try out, although it will bug you to buy a license (which costs \$70). If you'd like, you can download and install it now.

Open up your text editor and save a new file called my-first-webpage.html. Notice the .html extension: that tells your computer this is an HTML file and should be opened with your web browser. When you save a file with the .html extension, you might get a warning from your text editor asking if you're sure you don't want to save it with the .txt extension. Tell it that yes, you are sure.

Now, double-click the file you just created (in Finder if you're on a Mac, or in Windows Explorer if you're on a PC) to open it in your web browser. You should have a blank page. Now you're ready to start writing your first HTML!

## Summary

Free text editors:

Notepad (included with Windows) TextEdit (included with Macs) Professional text editor:

Sublime Text Costs \$70, but free to try indefinitely

# HTML Layouts

Websites often display content in multiple columns (like a magazine or newspaper).

## HTML Layout Using Elements

This example uses four elements to create a multiple column layout:

### Example

```
<body>

<div id="header">
<h1>City Gallery</h1>
</div>

<div id="nav">
London<br>
Paris<br>
Tokyo
</div>

<div id="section">
<h1>London</h1>
<p>London is the capital city of England. It is the most populous city in the United King
with a metropolitan area of over 13 million inhabitants.</p>
<p>Standing on the River Thames, London has been a major settlement for two millennia,
its history going back to its founding by the Romans, who named it Londinium.</p>
</div>

<div id="footer">
Copyright © opteamize.in
</div>

</body>
```

**The CSS:**

```
<style>
#header {
  background-color:black;
  color:white;
  text-align:center;
  padding:5px;
}
#nav {
  line-height:30px;
  background-color:#eeeeee;
  height:300px;
  width:100px;
  float:left;
  padding:5px;
}
#section {
  width:350px;
  float:left;
  padding:10px;
}
#footer {
  background-color:black;
  color:white;
  clear:both;
  text-align:center;
  padding:5px;
}
</style>
```

## Website Layout Using HTML5

HTML5 offers new semantic elements that define different parts of a web page:



- `<header>` - Defines a header for a document or a section
- `<nav>` - Defines a container for navigation links



- `<section>` - Defines a section in a document
- `<article>` - Defines an independent self-contained article
- `<aside>` - Defines content aside from the content (like a sidebar)
- `<footer>` - Defines a footer for a document or a section
- `<details>` - Defines additional details
- `<summary>` - Defines a heading for the `<details>` element

This example uses `<header>` , `<nav>` , `<section>` , and `<footer>` to create a multiple column layout:

### Example

```
<body>

<header>
<h1>City Gallery</h1>
</header>

<nav>
London<br>
Paris<br>
Tokyo
</nav>

<section>
<h1>London</h1>
<p>London is the capital city of England. It is the most populous city in the United King
with a metropolitan area of over 13 million inhabitants.</p>
<p>Standing on the River Thames, London has been a major settlement for two millennia,
its history going back to its founding by the Romans, who named it Londinium.</p>
</section>

<footer>
Copyright © opteamize.in
</footer>

</body>
```

### The CSS:

```
<style>
header {
  background-color:black;
  color:white;
  text-align:center;
  padding:5px;
}
nav {
  line-height:30px;
  background-color:#eeeeee;
  height:300px;
  width:100px;
  float:left;
  padding:5px;
}
section {
  width:350px;
  float:left;
  padding:10px;
}
footer {
  background-color:black;
  color:white;
  clear:both;
  text-align:center;
  padding:5px;
}
</style>
```

## HTML Layout Using Tables

Layout can be achieved using the element, because table elements can be styled with CSS:

### Example

```
<body>

<table class="lamp">
<tr>
  <th>
    
  </th>
  <td>
    The table element was not designed to be a layout tool.
  </td>
</tr>
</table>

</body>
```

## The CSS:



```
<style>
table.lamp {
  width:100%;
  border:1px solid #d4d4d4;
}
table.lamp th, td {
  padding:10px;
}
table.lamp th {
  width:40px;
}
</style>
```

# HTML Tags

element

Tag	Description
<code>&lt;!-- . . . --&gt;</code>	Defines a comment
<code>&lt;!DOCTYPE&gt;</code>	Defines the document type
<code>&lt;a&gt;</code>	Defines a hyperlink
<code>&lt;abbr&gt;</code>	Defines an abbreviation or an acronym
<code>&lt;acronym&gt;</code>	Not supported in HTML5. Use <code>&lt;abbr&gt;</code> instead. Defines an acronym
<code>&lt;address&gt;</code>	Defines contact information for the author/owner of a document
<code>&lt;applet&gt;</code>	Not supported in HTML5. Use <code>&lt;embed&gt;</code> or <code>&lt;object&gt;</code> instead. Defines an embedded applet
<code>&lt;area&gt;</code>	Defines an area inside an image-map
<code>&lt;article&gt;</code>	Defines an article
<code>&lt;aside&gt;</code>	Defines content aside from the page content
<code>&lt;audio&gt;</code>	Defines sound content
<code>&lt;b&gt;</code>	Defines bold text
<code>&lt;base&gt;</code>	Specifies the base URL/target for all relative URLs in a document
<code>&lt;basefont&gt;</code>	Not supported in HTML5. Use CSS instead. Specifies a default color, size, and font for all text in a document
<code>&lt;bdi&gt;</code>	Isolates a part of text that might be formatted in a different direction from other text outside it
<code>&lt;bdo&gt;</code>	Overrides the current text direction
<code>&lt;big&gt;</code>	Not supported in HTML5. Use CSS instead. Defines big text
<code>&lt;blockquote&gt;</code>	Defines a section that is quoted from another source
<code>&lt;body&gt;</code>	Defines the document's body
<code>&lt;br&gt;</code>	Defines a single line break
<code>&lt;button&gt;</code>	Defines a clickable button
<code>&lt;canvas&gt;</code>	Used to draw graphics, on the fly, via scripting (usually JavaScript)
<code>&lt;caption&gt;</code>	Defines a table caption
<code>&lt;center&gt;</code>	Not supported in HTML5. Use CSS instead. Defines centered text
<code>&lt;cite&gt;</code>	Defines the title of a work

<code>&lt;code&gt;</code>	Defines a piece of computer code
<code>&lt;col&gt;</code>	Specifies column properties for each column within a
<code>&lt;colgroup&gt;</code>	Specifies a group of one or more columns in a table for formatting
<code>&lt;datalist&gt;</code>	Specifies a list of pre-defined options for input controls
<code>&lt;dd&gt;</code>	Defines a description/value of a term in a description list
<code>&lt;del&gt;</code>	Defines text that has been deleted from a document
<code>&lt;details&gt;</code>	Defines additional details that the user can view or hide
<code>&lt;dfn&gt;</code>	Represents the defining instance of a term
<code>&lt;dialog&gt;</code>	Defines a dialog box or window
<code>&lt;dir&gt;</code>	Not supported in HTML5. Use <code>&lt;div&gt;</code> instead. Defines a directory list
<code>&lt;div&gt;</code>	Defines a section in a document
<code>&lt;dl&gt;</code>	Defines a description list
<code>&lt;dt&gt;</code>	Defines a term/name in a description list
<code>&lt;em&gt;</code>	Defines emphasized text
<code>&lt;embed&gt;</code>	Defines a container for an external (non-HTML) application
<code>&lt;fieldset&gt;</code>	Groups related elements in a form
<code>&lt;figcaption&gt;</code>	Defines a caption for a element
<code>&lt;figure&gt;</code>	Specifies self-contained content
<code>&lt;font&gt;</code>	Not supported in HTML5. Use CSS instead. Defines font, color, and size for text
<code>&lt;footer&gt;</code>	Defines a footer for a document or section
<code>&lt;form&gt;</code>	Defines an HTML form for user input
<code>&lt;frame&gt;</code>	Not supported in HTML5. Defines a window (a frame) in a frameset
<code>&lt;frameset&gt;</code>	Not supported in HTML5. Defines a set of frames
<code>&lt;h1&gt; to &lt;h6&gt;</code>	Defines HTML headings
<code>&lt;head&gt;</code>	Defines information about the document
<code>&lt;header&gt;</code>	Defines a header for a document or section
<code>&lt;hr&gt;</code>	Defines a thematic change in the content
<code>&lt;html&gt;</code>	Defines the root of an HTML document

<code>&lt;i&gt;</code>	Defines a part of text in an alternate voice or mood
<code>&lt;iframe&gt;</code>	Defines an inline frame
<code>&lt;img&gt;</code>	Defines an image
<code>&lt;input&gt;</code>	Defines an input control
<code>&lt;ins&gt;</code>	Defines a text that has been inserted into a document
<code>&lt;kbd&gt;</code>	Defines keyboard input
<code>&lt;keygen&gt;</code>	Defines a key-pair generator field (for forms)
<code>&lt;label&gt;</code>	Defines a label for an  element
<code>&lt;legend&gt;</code>	Defines a caption for a  element
<code>&lt;li&gt;</code>	Defines a list item
<code>&lt;link&gt;</code>	Defines the relationship between a document and an external resource (most used to link to style sheets)
<code>&lt;main&gt;</code>	Specifies the main content of a document
<code>&lt;map&gt;</code>	Defines a client-side image-map
<code>&lt;mark&gt;</code>	Defines marked/highlighted text
<code>&lt;menu&gt;</code>	Defines a list/menu of commands
<code>&lt;menuitem&gt;</code>	Defines a command/menu item that the user can invoke from a popup menu
<code>&lt;meta&gt;</code>	Defines metadata about an HTML document
<code>&lt;meter&gt;</code>	Defines a scalar measurement within a known range (a gauge)
<code>&lt;nav&gt;</code>	Defines navigation links
<code>&lt;noframes&gt;</code>	Not supported in HTML5. Defines an alternate content for users that do not support frames
<code>&lt;noscript&gt;</code>	Defines an alternate content for users that do not support client-side scripts
<code>&lt;object&gt;</code>	Defines an embedded object
<code>&lt;ol&gt;</code>	Defines an ordered list
<code>&lt;optgroup&gt;</code>	Defines a group of related options in a drop-down list
<code>&lt;option&gt;</code>	Defines an option in a drop-down list
<code>&lt;output&gt;</code>	Defines the result of a calculation

# Indentation and spacing

Whether you are writing HTML, CSS, JavaScript or any other programming language, well-written code follows consistent indentation and spacing patterns. In some cases, indentation and spacing are required for the code to function properly. In other cases, inconsistent indentation and spacing will not impact the functionality but will cause your code to be difficult to read and understand.

Here is some HTML with inconsistent indentation and spacing:

```
<!DOCTYPE html>
<html><head>
  <title>  Example title</title>
    </head>
<body>
  <h1>Example header</h1>
  <p>  Page content  </p>
    </body>
</html>
```

It is difficult to see where the various elements begin and end and which elements are nested within other elements. Here is the same code with consistent indentation and spacing:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example title</title>
  </head>
  <body>
    <h1>Example header</h1>

    <p>Page content</p>
  </body>
</html>
```

In HTML, the beginning and ending tags of an element on multiple lines should be left-aligned. For example, when you see `<html>` , you should be able to visually scan the page straight down to find its closing tag (the same for `<head>` and `<body>` ). One way to implement this is to create opening and closing tags at the same time and then add the contents.

When elements are nested within another element, they should be indented two spaces from the opening tag. For example, `<head>` is indented two spaces from `<html>` and `<title>` is indented two spaces from `<head>` .

If you use the tab key to space over, make sure it is set to two spaces. Some defaults are set to other amounts such as four spaces which leaves too much white space and is not standard practice.

Also, notice that there is no space between the text content of an element and its tags. For example, `<h1>Example header</h1>` has no spaces between `<h1>` and the word Example.

Regardless of the language you use to write code, learn the indentation and spacing standards for it. Practice consistency so that your code is easily readable to other developers as well as your future self.



# Block elements

Embedded Video: [https://www.youtube.com/watch?v=Rp7zt-d\\_wHs?rel=0](https://www.youtube.com/watch?v=Rp7zt-d_wHs?rel=0)

The first thing we need to learn is how to make simple web pages using HTML, which stands for HyperText Markup Language. HyperText is a nerdy word for a link, a markup language is a language that lets you insert instructions within the text that you write, and HTML is the language web browsers translate into the web pages you see on your screen. Now, we're going to start writing our first HTML!

With my-first-webpage.html open in a text editor, let's start by typing the following code. Don't type the my-first-webpage.html bit at the top - that's just telling you what file the following code goes in.

my-first-webpage.html

```
<p>This is my first web page!</p>
<p>Isn't it nice?</p>
```

The `<p>` is called a p tag, and it means that everything that follows is a paragraph, until `</p>` is reached. The `</p>` is called a closing tag or end tag.

Now, if we open the file in a web browser (by double-clicking it), we have two paragraphs of text. We just wrote our first HTML code!

Now, let's add some more stuff to the page:

my-first-webpage.html

```
<h1>My first web page</h1>
<h2>Written with the guidance of Opteamize Coding School</h2>

<p>This is my first web page!</p>
<p>Isn't it nice?</p>
```

Here, we've added a main header ( `<h1>` ) and a subheader ( `<h2>` ). You can actually add up to six different levels of headers using `<h1>` through `<h6>` .

HTML ignores spaces and blank lines, which together are called whitespace. That's why the extra line between our headers and paragraphs doesn't do anything. We could take it out, or make it three spaces, and it would look exactly the same in the browser.

Let's add some elements to our page:

## my-first-webpage.html

```
<h1>My first web page</h1>
<h2>Written with the guidance of Epicodus</h2>

<p>This is my first web page!</p>
<p>Isn't it nice?</p>

<p>Here are some things I'm going to learn about coding:</p>
<ul>
  <li>HTML</li>
  <li>CSS</li>
  <li>JavaScript</li>
  <li>And a lot more!</li>
</ul>
```

Here we've added an unordered list ( `<ul>` ) with four list items ( `<li>` ). If we wanted to make an ordered list that uses numbers and letters instead of bullets, we would use an `<ol>` tag instead of a `<ul>` .

Notice how I've indented the

- s within the

s. This makes it much easier to read which tags are nested, and makes it easy to see where the closing tag goes when it's not on the same line. It's a good practice to indent using two spaces (tabs are often rendered as different widths, whereas spaces are standard). It's also a good idea to write your closing tag just after you write your opening tag, and then go back and fill in whatever comes in the middle. This way, you won't forget to close a tag after you've opened it.

We just have a couple last things to add to make this a valid HTML web page:

## my-first-webpage.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello world!</title>
  </head>
  <body>
    <h1>My first web page</h1>
    <h2>Written with the guidance of Epicodus</h2>

    <p>This is my first web page!</p>
    <p>Isn't it nice?</p>

    <p>Here are some things I'm going to learn about coding:</p>
    <ul>
      <li>HTML</li>
      <li>CSS</li>
      <li>JavaScript</li>
      <li>And a lot more!</li>
    </ul>
  </body>
</html>
```

The `<!DOCTYPE html>` tag tells the browser that this document contains HTML, and specifically that it contains the newest version of HTML, HTML5. (An example of a doctype for an older version of HTML is `<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">.`)

The `<head>` tag contains information about the page, which for now is just the `<title>` tag. And the `<title>` tag sets the title for the web page - if you look in your browser, you can see that the title of the window is now Hello world!.

Finally, the actual content of the page is now wrapped in a `<body>` tag.

Here's another example of a web page that I've saved as favorite-things.html. I've added a couple new things to the page; see if you can figure them out.

favorite-things.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Michael's favorite things</title>
  </head>
  <body>
    <h1>My favorite things</h1>
    <h2>These are a few of my favorite things.</h2>

    <h3>People</h3>
    <p>Here are some of my favorite people!</p>
    <ul>
      <li>My brother Christopher</li>
      <li>My mom and dad:
        <ul>
          <li>Steve</li>
          <li>Carol</li>
        </ul>
      </li>
      <li>My friend Jessica</li>
      <li>My friend Candy</li>
    </ul>

    <h3>Flavors of ice cream</h3>
    <p>These are my three favorites, in order:</p>
    <ol>
      <li>Pistachio</li>
      <li>Malted milk chocolate</li>
      <li>Black licorice (really!)</li>
    </ol>

    <p>Thanks for reading about me!</p>
  </body>
</html>
```

## Summary

```
<h1>I am a main heading.</h1>
<h2>I am a subheading.</h2>
<h6>I am the smallest heading.</h6>

<p>I am a paragraph.</p>

<ul>

<li>This</li>
<li>is an</li>
<li>unordered list.</li>
</ul>

<ol>

<li>This</li>
<li>is an</li>
<li>ordered list.</li>
</ol>

Valid HTML document:

<!DOCTYPE html>
<html>
  <head>
    <title>Example title</title>
  </head>
  <body>
    <h1>Example header</h1>

    <p>Page content</p>
  </body>
</html>
```

If the content inside a tag spans more than one line, indent everything within the tag two spaces in. Put the closing tag on the same indentation level as the opening tag.

# Block elements practice

In the Block elements video, we learned:

- The meaning of HTML
- How to create a static webpage with headers, paragraphs and lists
- The purpose of `<head>` tags and `<title>` tags

Now it's your turn! Here are some web pages for you to create.

- Follow along with the lesson to create your own `my-first-webpage.html`, just like the one I made. Make sure you pay close attention to getting the indentation correct, closing all of the tags you open, and making your page look just like mine. Create a Git repository in this project folder and track your changes with commits. Make sure you push your project to a remote repository on GitHub.
- Create a web page of your own favorite things. Follow along with my example, but make it about you.
- Make a homepage for your cat. You can get creative here! Maybe include information about their favorite toys and their favorite activities. (If you don't have a cat, pretend like you do.)
- Build a blog for your dog with a couple entries. Of course, you're more than welcome to make webpages about anything you want - these are just suggestions.

## 21. Inline elements

**Embedded Video:** [https://www.youtube.com/watch?v=91R\\_tlKHgAE?rel=0](https://www.youtube.com/watch?v=91R_tlKHgAE?rel=0)

All of the tags we learned about in the previous lesson are (at least by default) block elements, which means that each element is displayed on a new line. I'm going to modify our first webpage to include an inline element:

### my-first-webpage.html

```
<p>This is my <strong>first web page</strong>!</p>
```

The `<strong>` tag indicates that something is important. By default, it will make the text inside it bold (although we'll learn how to customize this soon). If we want to emphasize something, we'll use the `<em>` tag, which by default will italicize it:

### my-first-webpage.html

```
<h2>Written with the guidance of <em>Epicodus</em></h2>
```

Now, let's learn about perhaps the most important HTML tag of them all:

### my-first-webpage.html

```
<p>Check out <a href="http://www.epicodus.com/">Epicodus</a>. It's a great school for learning web programming!</p>
```

The `<a>` (for anchor) creates a link, and the href (for hypertext reference) attribute tells where the link should go to.

In the previous lesson, we made two files: my-first-webpage.html and favorite-things.html. They should be saved in the same folder. Here's how we can link from one to another:

### my-first-webpage.html

```
<p>Here is a link to my <a href="favorite-things.html">favorite things</a>.</p>
```

Instead of supplying a full path the address, we've provided a relative path. If a link doesn't start with http:// (or https://, or a few other specific protocols you may come across), your web browser will look in the same folder for the file you're linking to.

This brings up a pitfall you should be careful to avoid: if you write a link like `<a href="www.epicodus.com">link</a>`, it will look for a file called www.epicodus.com in the current directory. So if you're linking to another website, don't forget to put http:// in front!

Finally, let's make our webpage a bit prettier and include an image. Save an image in the same folder as your HTML files and then include it like this:

## my-first-webpage.html

```

```

The `<img>` tag is different from the other tags we've used in that it is self-closing: you don't need a closing tag. Also notice the `alt` attribute, which the browser displays if it can't render the image. This attribute is optional, but it's a really good idea to include it, so that blind people using software that reads web pages to them can know what the image is.

Typically, webpages will store their images in a separate folder from their HTML. Let's create a folder called `img` and put the image file in there. Now, we need to update our code to tell the browser where to find the image:

## my-first-webpage.html

```

```

`img/kitten.png` tells the browser to look in the `img` folder in the current directory and then look for the `kitten.png` file.

Let's update the page of favorite things to include some of the concepts we just learned about:

## favorite-things.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Michael's favorite things</title>
  </head>
  <body>
    <h1>My favorite things</h1>
    <h2>These are a few of my favorite things.</h2>

    <h3>People</h3>
    <p>Here are some of my favorite <em>people</em>!</p>
    <ul>
      <li>
        <a href="http://www.example.com/christopher.html">
          My brother <strong>Christopher</strong>
          
        </a>
      </li>
      <li>
        My mom and dad:
        <ul>
          <li>
            <a href="http://www.example.com/steve.html">
              <strong>Steve</strong>
              
            </a>
          </li>
        </ul>
      </li>
    </ul>
  </body>
</html>
```



```
</li>
<li>
  <a href="http://www.example.com/carol.html">
    <strong>Carol</strong>
    
  </a>
</li>
</ul>
</li>
<li>
  <a href="http://www.example.com/jessica.html">
    My friend <strong>Jessica</strong>
    
  </a>
</li>
<li>
  <a href="http://www.example.com/candy.html">
    My friend <strong>Candy</strong>
    
  </a>
</li>
</ul>

<p>Thanks for reading about me!</p>
</body>
</html>
```

## Summary

```
<strong> I am important and bold by default. </strong>
```

```
<em> I am emphasized and italic by default. </em>
```

```
<a href="http://www.epicodus.com"> I link to www.epicodus.com. </a>
```

```
<a href="another_file.html"> I link to another file in the same folder as myself. </a>
```

```
<a href="anotherfolder/anotherfile.html"> I link to another file in a folder called
another_folder. </a>
```

```

```

## 22.Inline elements practice

In the Inline elements video, we learned:

- Headers, paragraphs and list elements are considered block elements
- Inline elements are elements included within block element tags
- Some inline elements:

`<a>` - with the href attribute to make a hypertext link

`<img>` - with the alt attribute

Here are some exercises for you to practice using inline HTML elements. Continue to make sure you are using proper indentation to keep your code clean and readable!

Add `<strong>` and `<em>` tags to your `my-first-webpage.html` , like I did in the lesson.

Link from your `my-first-webpage.html` to your `favorite-things.html` .

Add photos of your favorite things to your `favorite-things.html` page. Make an img folder to hold these photos and make sure the path is correctly written on your webpage.

In your `favorite-things.html` page, add links to your favorite websites.

Update your cat's homepage and your dog's blog with photos, links, and `<strong>` and `<em>` tags.

## 23. Styling text

Embedded Video: [https://www.youtube.com/watch?v=\\_M7W89FS9Bw?rel=0](https://www.youtube.com/watch?v=_M7W89FS9Bw?rel=0)

I'll be the first to admit it: the web pages we've made so far are pretty ugly. Let's learn our first **CSS**, which stands for **Cascading Style Sheets**. They're called cascading because often a single element on a webpage will have multiple sources that style it, and CSS resolves those differences to create the style you see on the screen. But we'll get into more of that later.

Side note: if trying to make something look pretty is your idea of an awful time, don't worry: this isn't a design course. We're going to learn enough about styling to get you to the point where you can apply styles that other people develop without having to do any design yourself. And if you love design and want to learn more, I'll point you in the right direction at the end of this section.

Let's get going and make a folder called `css` and create a new file in it called `styles.css`. The first thing we need to do is to tell our HTML document to use this file for its CSS:

### favorite-things.html

```
<head>
  <link href="css/styles.css" rel="stylesheet" type="text/css">
  <title>Michael's favorite things</title>
</head>
```

I'm leaving out everything above and below the `<head>` tag for the sake of brevity.

In our `styles.css` file, let's add our first bit of CSS:

### styles.css

```
h1 {
  color: blue;
}
```

Now, if we refresh the page, our header is blue!

What we just wrote is called a CSS rule. The `h1` part is called a **selector**, because it selects the HTML elements that the rule applies to. The `color` is called a **property**, and then `blue` is called a **value**.

Once again, notice the indentation: the property and value are indented two spaces, because they are inside the selector.

Let's add some more rules:

### **styles.css**

```
h1 {
  color: blue;
  text-align: center;
}

h2 {
  font-style: italic;
  text-align: center;
}

p {
  font-family: sans-serif;
}

ul {
  font-size: 20px;
  line-height: 30px;
}
```

This should be mostly self-explanatory, but here are a couple notes.

`px` is short for pixels, which are a unit of measurement.

Sans Serif is a kind of font.

Named colors, like the `blue` we used for our `h1`, aren't used very commonly. Instead, it's more typical to use a three- or six-digit hexadecimal code like this:

```
h1 {
  color: #00f;
}
```

or like this:

```
h1 {
  color: #0000ff;
}
```

Both of these are equivalent to the named color `blue`. If you use a graphics program, it usually can tell you the hex code for the colors you're using. There are also a lot of online tools for picking colors and getting their hex codes.

Finally, you might be wondering why we made our `<h2>` s italic in CSS instead of just adding the `<em>`. The reason is two-fold. First, we can modify the `<em>` tag just like any other HTML tag. Remember, `<em>` stands for emphasize, and at some point, we might decide that things that need emphasis should be bolded in addition to being italicized, which we might not want for our `<h2>` s. Second, one of the great advantages of CSS is that it makes it easy to change styles in many places at once. Let's say we use an `<em>` tag inside our `<h2>` s to make them italic. If we later change our mind and want them to be bold, we have to remove all of the `<em>` tags and replace them with `<strong>` tags. By using CSS instead, we only have to change the one CSS rule, and every single `<h2>` on every single web page that uses this CSS file will be updated.

My apologies for the long lecture, but this last point is really important. Building web sites is fundamentally different from building physical things. When you finishing building a bridge, you're more or less done: there's not much you can change besides a coat of paint. But when you finish building a website, you've only just begun. There will be additions, deletions, changes, redesigns, and a whole lot more. CSS gives us some powerful tools to make it easier to change what we've built. If you remember nothing else, remember this: good code is code that is easy to change.

To see the beauty of using CSS, check out CSS Zen Garden. On the right side of the page, clicking one of the links will apply a different stylesheet to the same HTML. The page is totally transformed - no changes to the HTML needed. Your goal in writing CSS should be to allow to Zen Garden-style changes to your styles.

If you're following along and your CSS file isn't actually adding styles to your web page, here's your first lesson in debugging. To keep things concise, I'll only give instructions for Google Chrome. If you aren't using Chrome, I'd suggest you download and install it now. In Chrome, click on the menu button (it looks like ☰ and is on the right side of the address bar), go to the Tools menu, and click JavaScript Console. (As a shortcut, you can also press `Cmd + Alt + J`.) We're not actually using JavaScript (yet!), but this is where all errors in your web page are logged. If you see something that looks like Failed to load resource file:///Users/michael/epicodus/css/styles.css, Chrome is telling you that it you told it to look for your CSS file at Users/michael/epicodus/css/styles.css, but that the file wasn't there. In other words, you either gave the wrong path to the file, or you put the file in the wrong place.

And since we're talking about putting files in the right place, one easy way to make sure you're working on the right files is to open your project folder in Sublime. Just drag the folder from Finder onto the Sublime icon, and it will open the whole thing in Sublime.

# Summary

Link a stylesheet to your HTML document:

```
<head>
  <link href="path/to/my/stylesheet.css" rel="stylesheet" type="text/css">
</head>
```

Example CSS rule:

```
h1 {
  text-align: center;
}
```

In this example, h1 is the selector, text-align is the property, and center is the value.

Common CSS text rules:

```
h1 {
  color: #0000ff;
  text-align: center;
  font-style: italic;
  font-weight: bold;
  font-family: sans-serif;
  font-size: 20px;
  line-height: 30px;
}
```

Use the JavaScript console (Cmd + Alt + J) to debug when your CSS isn't linked correctly.

## 24. Styling text practice

In the Styling text video, we learned:

- How to set up a styles.css file and link to it in our HTML page
- CSS rules, selectors and properties
- Hexadecimal codes
- Basic debugging techniques

Now it's your turn! If you would like to access the JavaScript Console from a Chrome browser just type `Cmd + Alt + J`.

- Create a blank CSS stylesheet for your `my-first-webpage.html` . Put it in a sub-folder called `css` . Link the HTML document to the stylesheet (make sure the path to the file is correct!).
- Add a single CSS rule that makes the main header ( `<h1>` ) red. Refresh the page in the browser to verify that the stylesheet is linked correctly. If it doesn't change the page, make sure that you've put the CSS file in the correct folder, and specified the right path to the CSS file. Open the JavaScript console to debug if necessary.
- Now, use the following properties in your CSS: `color` , `text-align` , `font-style` , `font-weight` , `font-family` , `font-size` , and `line-height` .
- Make a new webpage called `favorite-places.html` and make it about your favorite places to visit. Add images, links and inline styling.
- Switch to your `favorite-places.html` page. Make another stylesheet (make sure to name it differently than the `my-first-webpage.html` stylesheet!) and apply CSS rules.
- Make a new stylesheet for both your `my-first-webpage.html` and `favorite-places.html` . Remove the other stylesheets and link to the single stylesheet from both pages. You should have a consistent style between the two documents.
- Finally, update your cat's homepage and dog's blog to have some styles as well. (If you haven't created these yet, go ahead and do that.)

Be sure to make commits and push those commits to GitHub as you are working on your pages.

## 27. Styling with classes

Embedded Video: <https://www.youtube.com/watch?v=-6M0CfJfGII?rel=0>

Let's make a new page with two types of paragraphs: regular paragraphs, and a summary at the top of the page:

### paragraphs.html

```
<!DOCTYPE html>
<html>
  <head>
    <link href="css/styles.css" rel="stylesheet" type="text/css">
    <title>Paragraphs!</title>
  </head>
  <body>
    <h1>Paragraphs of text</h1>
    <h2>Plus an intro at the top!</h2>

    <p>Here is an intro. Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do

    <p>And here is the full version. Ut enim ad minim veniam, quis nostrud exercitation u

    <p>Here is a bit more stuff. At vero eos et accusamus et iusto odio dignissimos ducim

    <p>And, this is the end. Lorem ipsum dolor sit amet, consectetur adipisicing elit, se

    <p>Here is a bit about the author at the end. Nam libero tempore, cum soluta nobis es
  </body>
</html>
```

The dummy text I put in here is called Lorem Ipsum text. It looks like Latin but is actually just a bunch of fake words. Lorem Ipsum text is often used as a placeholder when doing design; you can search online for Lorem Ipsum and find many websites where you can copy the text from.

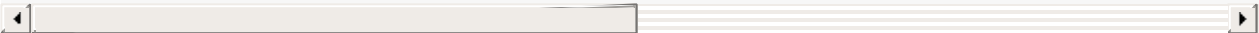
This page is nice, but I'd like for the top paragraph to look more like an intro. I want it to be italicized, and to be skinnier than the other paragraphs. Here's how we do that without changing the other paragraphs. First, we add a class to the

tag just for that paragraph:

### paragraphs.html



```
<p class="intro">Here is a summary. Lorem ipsum dolor sit amet, consectetur adipisicing e
```



Then, we make a CSS rule for `<p>` tags with the intro class :

### **styles.css**


```
p.intro {  
  width: 50%;  
  font-style: italic;  
}
```

And now our intro paragraph has a width of only 50% and is italicized, while the rest of the paragraphs stay the same.

Similarly, we might make a class for the final paragraph about the author:

### **paragraphs.html**

```
<p class="author">Here is a bit about the author at the end. Nam libero tempore, cum solu
```



And style it like this:

### **styles.css**

```
p.author {  
  width: 75%;  
  font-style: italic;  
  font-weight: bold;  
}
```

You can also apply a class to multiple tags. Here's another example of web page that shows you how:

### **fido.html**

```
<!DOCTYPE html>
<html>
  <head>
    <link href="css/styles.css" rel="stylesheet" type="text/css" media="all">
    <title>How to take care of Fido</title>
  </head>
  <body>
    <h1>Instructions for Fido's babysitter</h1>

    <p>Thanks so much for watching Fido this weekend! Here's everything you need to know

    <p>Here's a bit of history about Fido. Lorem ipsum dolor sit amet, consectetur adipis

    <p class="important">Fido is allergic to a lot of foods. Ut enim ad minim veniam, quis

    <p>Fido likes to play. Sed ut perspiciatis unde omnis iste natus error sit voluptatem

    <p>Here are a list of things that you should do with Fido every day:</p>

    <ul class="important">
      <li>Walk him</li>
      <li>Feed him</li>
      <li>Pet him</li>
    </ul>

    <p>If you have any questions while I'm out of town, I <strong class="important">won't

    <p>Thanks again!</p>
  </body>
</html>
```

Here we've added the important class to a paragraph, an unordered list, and a `<strong>` tag. Now, we can style everything with the important class like this:

### fido-styles.css

```
.important {
  color: red;
}
```

Also, check out how I linked to an email address in my HTML using the syntax `<a href="mailto:michael@epicodus.com">` .

## Summary

Add a class to one or more HTML tags:

## webpage-with-classes.html

```
<p class="important">Important stuff should be:</p>
<ul class="important">
  <li>red,</li>
  <li>bold, and</li>
  <li>big, but only for paragraphs.</li>
</ul>
```

Create CSS rules for that class:

## styles-for-classes.css

```
.important {
  color: red;
  font-weight: bold;
}
```

Create a CSS rule for a tag with only a certain class:

## styles-for-classes.css

```
p.important {
  font-size: 24px;
}
```

## 28.Classes practice

In the Styling with classes video, we learned:

- How to add classes to style large chunks of our webpage
- Classes can be applied to multiple HTML tags
- How to link to an email address with `<a href="mailto:michael@epicodus.com">`

Try out using classes yourself:

- Create a new website for a cupcake shop. Your site should have images, text and links. Be creative!
- Pick three HTML elements and give them a class called `flashy`, and create a CSS rule to make them red and italic.
- Create a class called `best` and add it to the most popular cupcake featured on your site. Apply a style of your choice to make it stand out.
- Create a new project called `boring-lecture.html` that is a fake page about a boring subject of your choice. Use lorem ipsum text to fill up the page using `<p>` tags.
- Create a section of the page at the beginning for an introduction to the lecture; in that intro, include a `<h2>` subheading, a paragraph or two, and a picture. Give each of these elements the class `intro`.
- Create CSS rules for `h2.intro`, `p.intro`, and `img.intro`. Remember that in CSS, classes are prefaced with a `.`

Add some styles of your own choosing to your cat's homepage and your dog's blog.

## 29.Organizing with divs and spans

Embedded Video: <https://www.youtube.com/watch?v=tBR4cft-5-c?rel=0>

Classes are great for selectively applying styles to certain parts of your web pages. But sometimes you'll have entire sections of your pages that you want to style a certain way, and adding a style to every tag gets annoying. Here's a new page I made about fish. Notice how several elements in a row all have the same class:

```
<!DOCTYPE html>
<html>
  <head>
    <link href="css/styles.css" rel="stylesheet" type="text/css" media="all">
    <title>Everything about fish</title>
  </head>
  <body>
    <h1>Everything about fish</h1>

    <p>Here is everything you need to know about fish. First, the important stuff:</p>

    <h2 class="important">Types of fish</h2>

    <p class="important">There are many types of fish. Lorem ipsum dolor sit amet, consec

    <p class="important">Here are some types of fish:</p>

    <ul class="important">
      <li>Goldfish</li>
      <li>Catfish</li>
      <li>And lots, lots more!</li>
    </ul>

    <h2 class="important">Other important stuff</h2>

    <p class="important">Here is some other really important stuff about fish. Ut enim ad

    <p>Now, for some less important stuff about fish.</p>

    <h2 class="unimportant">Fish like to play golf</h2>

    <p class="unimportant">Little-known fact: fish like to play golf. Sed ut perspiciatis

    <h2 class="unimportant">Fish are not mammals</h2>

    <p class="unimportant">Duis aute irure dolor in reprehenderit in voluptate velit esse
  </body>
</html>
```

Let's learn a new tag that will simplify all of this: the `<div>` tag.

## fish.html

```
<!DOCTYPE html>
<html>
  <head>
    <link href="css/styles.css" rel="stylesheet" type="text/css">
    <title>Everything about fish</title>
  </head>
  <body>
    <h1>Everything about fish</h1>

    <p>Here is everything you need to know about fish. First, the important stuff:</p>

    <div class="important">
      <h2>Types of fish</h2>

      <p>There are many types of fish. Lorem ipsum dolor sit amet, consectetur adipisicing
      </p>

      <p>Here are some types of fish:</p>

      <ul>
        <li>Goldfish</li>
        <li>Catfish</li>
        <li>And lots, lots more!</li>
      </ul>

      <h2>Other important stuff</h2>

      <p>Here is some other really important stuff about fish. Ut enim ad minim veniam, q
      </div>

      <p>Now, for some less important stuff about fish.</p>

      <div class="unimportant">
        <h2>Fish like to play golf</h2>

        <p>Little-known fact: fish like to play golf. Sed ut perspiciatis unde omnis iste n
        </p>

        <h2>Fish are not mammals</h2>

        <p>Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
        </div>
    </body>
  </html>
```

Then, we can style entire areas of the page using selectors for `.important` and `.unimportant`, for example:

### styles.css

```
.important {  
  color: red;  
}  
  
.unimportant {  
  color: blue;  
}
```

The `<div>` tag does nothing on its own: it's just a place to add a class that is then applied to everything that is nested within it. It's very common to apply styles by using `<div>` tags to separate the content of your pages. Typically, in your css, you won't even explicitly label a selector as being for a `<div>`; in other words, you'd write `.important`, rather than `div.important`.

Remember my little lecture about how good code is code that is easy to change? We just made our code easier to change: if we want to change the class name, we only have to change one `<div>` instead of all of the elements that are now nested inside of it. Nice!

For inline elements, there's a similar tag called `<span>`:

### fish.html

```
<p>There are many types of <span class="highlight">fish</span>. For example, there are <s
```



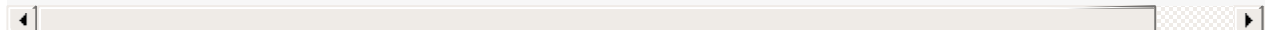
And we can style these inline elements with `.highlight` and `.fish-type`.

## Summary

Group elements together with similar styles using `<div>` s. For example, change this:

### stuff.html

```
<h2 class="important">You better read this</h2>  
<p class="important">Here are the things you really need to know.</p>  
<img class="important" src="scary.png" alt="This is what will happen if you don't read it
```



to this:

### stuff.html



```
<div class="important">
  <h2>You better read this</h2>
  <p>Here are the things you really need to know.</p>
  
</div>
```

Create in-line styles using `<span>` `s`:

### **different.html**

```
<p>Something is <span class="special">different</span> in this sentence.</p>
```

Style `<div>` `s` and `<span>` `s` just like you would any other element, without explicitly using `div` or `span` in the selector:

### **styles.css**

```
.important {
  color: red;
}
```

## 30.Div and span practice

In the Organizing with divs and spans video, we learned:

- `<div>` tags are places to add classes to large parts of our page and separate out content
- `<span>` tags are places to add classes to content within block elements

Try out using divs and spans yourself.

- Rewrite the boring-lecture page and remove the intro class from the elements themselves, and create a `<div>` with the intro class to hold the elements.
- Create a new blogging website about whatever you want and use `<div>` s to group together and style each blog post.
- Create an "about me" section that's styled differently from the rest of the page.
- Start a new webpage to house your resume. Use `<div>` s and `<span>` s to style sections for your education, work experience, skills, etc.

## 31.Laying out with floats

Embedded Video: <https://www.youtube.com/watch?v=ibI2W1URUTl?rel=0>

So far, everything we've done has been laid out very simply, from top to bottom for block elements, and left to right for inline ones. But often we'll want to have text wrap around an image, or have a sidebar on one side, or create a page with multiple columns. We can design all of these layouts using **floats**.

Here's an example of how to have text wrap around an image. First, the HTML for a new page about walrus:

### walrus.html

```
<!DOCTYPE html>
<html>
  <head>
    <link href="css/styles.css" rel="stylesheet" type="text/css">
    <title>The Walrus: a strange and noble creature</title>
  </head>
  <body>
    
    <p>The walrus is truly a majestic animal. Lorem ipsum dolor sit amet, consectetur adi
  </body>
</html>
```

And then the CSS magic:

### styles.css

```
img {
  float: left;
}
```

Check it out: the image "floats" to the left, and the text wraps around it. If your web browser isn't wide enough that the text extends below the bottom of the image, resize the window so that you can see how the text flows at the bottom of the page.

You can of course make the image float the right, if you'd prefer. And if you'd like only certain images to float, you can just add a class to those images, and only float images with that class.

Now let's use floats to create a sidebar:

## walrus.html

```
<!DOCTYPE html>
<html>
  <head>
    <link href="css/styles.css" rel="stylesheet" type="text/css">
    <title>The Walrus: a strange and noble creature</title>
  </head>
  <body>
    <h1>All about walruses</h1>
    <div class="sidebar">
      <h2>IRL</h2>
      <p>Want to see walruses in real life? Here are some places to check out:</p>
      <ul>
        <li>The beach</li>
        <li>The ocean</li>
        <li>The bay</li>
        <li>Rocks</li>
      </ul>
    </div>

    <div class="main">
      <p>The walrus is truly a majestic animal. Lorem ipsum dolor sit amet, consectetur a
    </div>
  </body>
</html>
```

And the CSS:

## styles.css

```
.sidebar {
  width: 30%;
  float: right;
  background-color: yellow;
}
```

Are you starting to see how this works? Let's do one more thing with floats and create a column layout:

## columns.html

```
<!DOCTYPE html>
<html>
  <head>
    <link href="css/styles.css" rel="stylesheet" type="text/css" media="all">
    <title>Columns</title>
  </head>
  <body>
    <h1>Check out these columns</h1>
    <div class="column">
      <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor
    </div>

    <div class="column">
      <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor
    </div>

    <div class="column">
      <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor
    </div>
  </body>
</html>
```



CSS:

### **styles.css**

```
.column {
  width: 300px;
  float: left;
}
```

And now we have columns.

Floats are incredibly powerful and an important way to lay out your web pages.

## Summary

Use floats to make elements float to one side of the page. Other elements will wrap around that element.

Float an image to the left:

### **floating-image.html**

```

```

## styles.css

```
img.main {  
  float: left;  
}
```

Create a sidebar:

## sidebar.html

```
<h1>Title</h1>  
<div class="sidebar">  
  <h2>Sidebar</h2>  
  <p>Some sidebar stuff.</p>  
</div>  
<p>Main page stuff.</p>
```

## sidebar-styles.css

```
.sidebar {  
  float: right;  
}
```

Make a column layout:

## columns.html

```
<div class="column">  
  <p>Column 1</p>  
</div>  
<div class="column">  
  <p>Column 2</p>  
</div>  
<div class="column">  
  <p>Column 3</p>  
</div>
```

## column-styles.css

```
.column {  
  width: 300px;  
  float: left;  
}
```

# Floats practice

In the Laying out with floats video, we learned how to use floats to:

- Wrap text around an image
- Create sidebars
- Create pages with multiple columns

Practice using floats to position images, create sidebars, and make column layouts.

- Create a website for your favorite band.
- Add a sidebar for upcoming shows and style it so it stands out on the page.
- Add images to the webpage and float them either to the left or right.
- Create a webpage for an interior decorator. Use columns to display information and pictures for "before", "during" and "after" renovations.

# The box model

Embedded Video: [https://www.youtube.com/watch?v=I\\_Cs6JKu1ow?rel=0](https://www.youtube.com/watch?v=I_Cs6JKu1ow?rel=0)

There's one last major piece of HTML and CSS we need to cover: the **box model**. Each element on a web page takes up space, and the box model describes the space around the element. Let's start with some unstyled HTML and then add CSS as we go to illustrate the box model:

## box-model.html

```
<!DOCTYPE html>
<html>
  <head>
    <link href="css/styles.css" rel="stylesheet" type="text/css" media="all">
    <title>Box model</title>
  </head>
  <body>
    <div class="my-class">Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed d
  </body>
</html>
```

Now, let's add a border around this text:

## styles.css

```
.my-class {
  border: 2px solid green;
}
```

Here's how to increase the space between the text in the `<div>` and the border around it:

## styles.css

```
.my-class {
  border: 2px solid green;
  padding: 4px;
}
```

Finally, we might want to make some room between the margin and the elements around it (which are, in this simple case, just the edges of the page):

## styles.css



```
.my-class {  
  margin: 12px;  
  border: 2px solid green;  
  padding: 4px;  
}
```

Now, let's apply a background color to this part of the page:

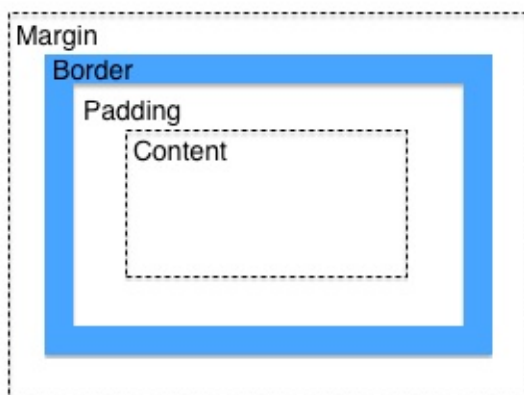
### **styles.css**

```
.my-class {  
  margin: 12px;  
  border: 2px solid green;  
  padding: 4px;  
  background-color: orange;  
}
```

It just colors the content itself and the padding around it.

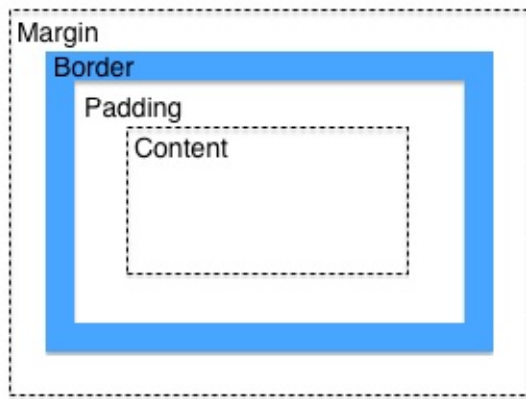
The box model is perhaps easiest to understand with a picture:

A diagram of the CSS box model.



You'll run into the box model a lot when building web pages, so it's a good idea to get familiar with it.

## **Summary**



```
.my-class {  
  margin: 12px;  
  border: 2px solid green;  
  padding: 4px;  
  background-color: orange;  
}
```

# Box model practice

In the Box model video, we learned:

- The space around each element on our web page is described by the box model
- The box model can add a margin, border and padding around web page content

Here are some exercises for you to practice using margins, borders, and padding:

- For the sidebar to your band's webpage, add a border and a bit of padding and margin. Give the sidebar a background color, if you haven't already.
- Add some padding around the images on your band's webpage.
- Between each entry on your blog project, add a bit of space. Use `margin-top` and `margin-bottom` so that you don't increase the spacing on the sides.
- Give different background colors to each column of your interior decorator site.

# Cascading

Embedded Video: <https://www.youtube.com/watch?v=yKMzAcE3U8s?rel=0>

You may have noticed that, in earlier lessons, we defined a style for `<ul>`, and it was applied to the text that was inside of `<li>` tags. This is called **inheritance**: since the `<li>` tags are nested within the `<ul>` tags, the `<li>` tags inherit the styles given to the `<ul>` s.

We can use inheritance to change the font for the entire web page like this:

```
body {  
  font-family: sans-serif;  
}
```

Since everything else in the web page is nested within the `<body>` tag, all of the text for the page will be Sans Serif now.

You can also directly specify how certain child elements are styled. For example, let's say you have this HTML:

**digital-vs-analog.html**

```
<!DOCTYPE html>
<html>
  <head>
    <link href="css/styles.css" rel="stylesheet" type="text/css">
    <title>Digital vs Analog</title>
  </head>
  <body>
    <h1>Digital vs Analog: The Showdown</h1>

    <p>Welcome to the ultimate face-off between digital and analog. May the best one win!

    <div class="digital">
      <p>Here's the good stuff about digital. Lorem ipsum dolor sit amet, consectetur adi

      <p>Find out more about digital <a href="http://www.example.com/digital.html">here</a>
    </div>

    <div class="analog">
      <p>Here's the good stuff about analog. Duis aute irure dolor in reprehenderit in vo

      <p>Find out more about analog <a href="http://www.example.com/analog.html">here</a>
    </div>

    <p>Now: YOU pick the winner!</p>
  </body>
</html>
```

Here's how we can style the paragraphs for the digital and analog sections differently:

### styles.css

```
.digital p {
  background-color: red;
}

.analog p {
  background-color: yellow;
}
```

This will give a red background to paragraphs that are nested within a tag that has the `digital` class, and a yellow background to paragraphs that are nested within a tag that has the `analog` class.

Here's an example of nesting even farther down:

### styles.css

```
.digital p a {  
  font-weight: bold;  
}  
  
.analog p a {  
  font-style: italic;  
}
```

You can nest as far down as you'd like.

Now, what if you have a couple rules like this:

### **styles.css**

```
.digital p a {  
  font-weight: bold;  
}  
  
a {  
  font-weight: normal;  
}
```

In this case, the most specific wins: in this case, a link in a paragraph in an element with the `digital` class will be bold, not normal, because the `.digital p a` selector is more specific than the `a` selector. This is called specificity, and it's both very powerful and, when things get complicated, a pain in the butt.

Here's one more example of specificity:

### **styles.css**

```
h2 {  
  font-size: 32px;  
}  
  
.countries h2 {  
  font-style: italic;  
}
```

Let's apply those rules to the following HTML:

### **greenland-iceland.html**

```
<!DOCTYPE html>
<html>
  <head>
    <link href="css/styles.css" rel="stylesheet" type="text/css" media="all">
    <title>How to tell apart Greenland and Iceland</title>
  </head>
  <body>
    <h1>How to tell apart Greenland and Iceland</h1>
    <h2>It's not easy, but I'll teach you how</h2>

    <p>If you're like me, you can never keep those two islands straight. But here's a foo

    <div class="countries">
      <h2>Greenland</h2>
      <p>Here's the thing about Greenland. Lorem ipsum dolor sit amet, consectetur adipis

      <h2>Iceland</h2>
      <p>Here's the thing about Iceland. Duis aute irure dolor in reprehenderit in volupt
    </div>

    <p>You'll never be confused again!</p>
  </body>
</html>
```

The `<h2>` s inside of the countries `<div>` will be italicized, but the `<h2>` s outside will not.

There's one final, important principle to understanding cascading: if two rules have the same specificity, the last one wins - in other words, the one that's the farthest down in the CSS file takes precedence.

Occasionally, you'll run into HTML that looks like this:

### inline-style.html

```

```

Here, the style is embedded in the HTML. This is called an inline style, and it is very, very bad! You lose all of the power of CSS when you do this, as you now can't re-use your styles and centralize them in one place. Fortunately, this is becoming rare, but you still may run into it every once in a while. You just need to know that, if you see an inline style, it takes precedence over what's defined in your CSS.

There are some additional rules about which style takes precedence in more complicated situations, but that's beyond the scope of what I'm teaching here.

# Summary

Elements inherit styles from their parents. This rule gives every element on the page the San Serif font:

```
body {  
  font-family: sans-serif;  
}
```

You can nest selectors:

**nesting-styles.css** .sidebar p img{ display: block; } nesting.html

```
<div class="sidebar">  
  <p>This image:  will be on its own line.</p>  
</div>
```

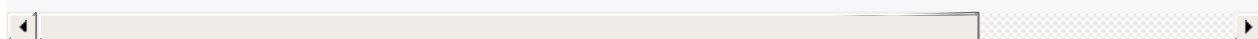
More specific rules take precedence over less specific rules:

**specificity-styles.css**

```
p {  
  background-color: green;  
}  
  
p.slow {  
  background-color: yellow;  
}  
  
p.slow .stop {  
  background-color: red;  
}
```

**specificity.html**

```
<p>This will have a green background.</p>  
<p class="slow">This will have a yellow background. <span class="stop">And this a red bac
```



In the case of conflicting rules, the last one wins. There are more complicated rules that we won't get into.



# Cascading practice

In the Cascading video, we learned:

- Styles can be inherited by elements nested within other elements
- In the CSS stylesheet, `body` can be used to style the entire page
- Elements within other elements are known as `child` elements
- There is no limit to nesting elements
- The most specific CSS rule takes precedence
- Avoid inline styling because it invalidates the power of CSS

Now is your turn to practice with inheritance and cascading styles.

- Create a new web page for an animal shelter. List out the animals available to be adopted, including pictures and descriptions of each animal. Have at least 3-5 animals listed.
- Display each animal's "profile" in a column.
- Change the styles of elements nested within the columns using the idea of cascading.
- Find other places in your web pages where you can change elements on the page according to cascading precedence.

Check out the CSS reference on the Mozilla Developer Network for ideas about what selectors and properties you can change with CSS.

# Applying other people's styles

Embedded Video: <https://www.youtube.com/watch?v=akiroGPO4v4?rel=0>

Often, you'll work on a team where somebody else gives you some CSS and you have to apply the correct classes to your HTML to use it properly. It's also very common that you'll be given some CSS, but then need to make small tweaks to it, which is why I think it's so important for developers to understand at least the basics of CSS.

One common theme is Bootstrap. Check out its homepage, which is of course, styled with Bootstrap itself. Here's a really powerful trick I want to show you for learning CSS and HTML. You can do this in almost any web browser, but I'll just give instruction for Chrome. Now, right-click anywhere on Bootstrap's homepage and click Inspect Element. A little window will pop open on the bottom of the screen that shows you the source code for the page, with the element you inspected selected. This window is called the Chrome Developer Tools, or DevTools. As a shortcut, you can open it with Cmd + Alt + I.

On the left side of the DevTools, you can select a different element, and expand or collapse all of the elements on the page to make it easier to see the layout. On the right side, the inspector shows you the CSS for the current element. One of the really cool things about this is that it shows you all of the different rules that have been applied to the current element, including those that have been overridden due to specificity or order. You can also uncheck a rule to disable it, or double-click a rule to edit it and see the change live. You can even add your own rules to see what they will do live.

Now, let's move on to using Bootstrap to style a page of our own. Download Bootstrap and then copy the **bootstrap.css** file into the **css** folder of your project directory. We'll start with the following HTML:

**bootstrap-playground.html**

```
<!DOCTYPE html>
<html>
  <head>
    <link href="css/bootstrap.css" rel="stylesheet" type="text/css">
    <title>A bit of Bootstrap</title>
  </head>
  <body>
    <h1>A bit of Bootstrap</h1>
    <h2>Let's see what Bootstrap can do for us</h2>

    <p>Bootstrap is pretty great, and we're going to explore why. Stay tuned for more exc

    <p>Here's a paragraph. Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed

    <p>Now, let's list some stuff out:</p>

    <ul>
      <li>Stuff</li>
      <li>More stuff</li>
      <li>Even more stuff</li>
    </ul>
  </body>
</html>
```

If we open this in our web browser, we can see that we already get some styling from Bootstrap. Try inspecting elements on this page to see what CSS rules are being applied.

Bootstrap's website looks much nicer than ours, so let's check out what classes they use and add them to our page. If we right click and inspect towards the top of any page, we can see that Bootstrap's pages are separated into three parts using `<header>`, `<main>`, and `<footer>` tags. Like `<div>` s, these tags don't do anything on their own, but represent logical grouping on the page. You can think of them just as `<div>` s with special names that can only be used once on a page. Within the `<header>` and `<main>`, there are `<div>` s with the class `container`; the `container` class is also applied to the `<footer>`. Let's keep our page simple and not bother with the `<header>`, `<main>`, and `<footer>` tags, but let's wrap the rest of the page in a `<div>` with the class `container`. Let's just add this container to our page and see how it looks:

### bootstrap-playground.html

```
<body>
  <div class="container">
    <h1>A bit of Bootstrap</h1>
    <h2>Let's see what Bootstrap can do for us</h2>

    <p>Bootstrap is pretty great, and we're going to explore why. Stay tuned for more exc

    <p>Here's a paragraph. Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed

    <p>Now, let's list some stuff out:</p>

    <ul>
      <li>Stuff</li>
      <li>More stuff</li>
      <li>Even more stuff</li>
    </ul>
  </div>
</body>
Ah, that's much better.
```

Let's apply one more Bootstrap style. If you go to any page on the Bootstrap website other

bootstrap-playground.html

```
<body>
  <div class="container">
    <h1>A bit of Bootstrap</h1>
    <h2>Let's see what Bootstrap can do for us</h2>

    <p class="lead">Bootstrap is pretty great, and we're going to explore why. Stay tuned

    <p>Here's a paragraph. Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed

    <p>Now, let's list some stuff out:</p>

    <ul>
      <li>Stuff</li>
      <li>More stuff</li>
      <li>Even more stuff</li>
    </ul>
  </div>
</body>
```

Doesn't that look nice?

The Bootstrap website actually documents how to use all of the styles it provides if you take the time to read through the docs, but I just wanted to show you how to figure out what class something is when it isn't well-documented, which you'll often run into. Also, there are many,

many themes available online for free or to buy, and many people have built on Bootstrap to create themes of their own (search for bootstrap themes and you'll get a huge number of results). This site and the Epicodus homepage are both built with Bootstrap themes.

Often, you'll start with a theme like Bootstrap and then want to change some of its styles. You could go into Bootstrap's CSS and change them yourself, but this isn't a good idea. If a new version of Bootstrap is released that fixes some bugs, you would have to go and find all of the changes you made to your Bootstrap CSS file and port them over to the new file. Instead, it's a better idea to leave the original file untouched, and create your own stylesheet to override Bootstrap's default styles.

Here's an example of how I would override Bootstrap's styles to make the main headers bold:

### **bootstrap-playground.html**

```
<!DOCTYPE html>
<html>
  <head>
    <link href="css/bootstrap.css" rel="stylesheet" type="text/css">
    <link href="css/styles.css" rel="stylesheet" type="text/css">
    <title>A bit of Bootstrap</title>
  </head>
  <body>
    <h1>A bit of Bootstrap</h1>
  </body>
</html>
```

### **styles.css**

```
h1 {
  font-weight: bold;
}
```

Make sure that your stylesheet is linked after Bootstrap's so that it takes precedence.

## **Summary**

Right-click a page and click Inspect Element to see what styles are applied and to change or add styles live.

Use Bootstrap or another theme to get nice styling for your web pages with little work on your own.

If you want to change the style that a theme provides you, don't edit their CSS directly. Instead, add your own styles to a separate file and link it after the origin theme:

```
<link href="css/bootstrap.css" rel="stylesheet" type="text/css">  
<link href="css/styles.css" rel="stylesheet" type="text/css">
```

# Theme practice

In the Applying other people's styles video, we learned:

- Bootstrap is a great way to add front-end design to webpages with minimal effort
- DevTools can be opened with Cmd + Alt + I to inspect an element
- The bootstrap.css file should be linked in the `<head>` tag, above your custom CSS stylesheet.

Let's get some practice using other people's themes. Browse all that you can do in Bootstrap:

- Make a new webpage about a place you'd like to go on vacation. Your webpage should include images, text and links. Use Bootstrap for your styles and remember you can use Inspect Element to see how other websites are styled.
- Override some of Bootstrap's styles with your own. Use a second stylesheet; don't change Bootstrap's code! Make sure your stylesheet is linked after Bootstrap in your HTML document.
- Make an image of your vacation destination the background of a **jumbotron** element on your page.
- Search for Bootstrap themes, and pick out another theme built with Bootstrap. Replace Bootstrap's origin CSS with the theme's CSS.

# Grid system and responsive design

Embedded Video: <https://www.youtube.com/watch?v=j2zAyX4TCEY?rel=0>

One of Bootstrap's most powerful features is its grid system, which lets you create rows and columns on your pages. Check out the Bootstrap CSS documentation. The documentation is all put in a `<div>` with class **row**; the sidebar is in a `<div>` with class **col-md-3**, and the documentation itself is in a `<div>` with class **col-md-9**. Bootstrap's grid has a 12-column layout, and you can choose how many columns each vertical section of the page takes up. So you could divide a row up into 4 columns of **col-md-3**, or a column of **col-md-4** and a one of **col-md-8**, and so on.

If you resize the Bootstrap homepage to make the window skinnier, you'll see something really cool: the page layout changes as the window gets narrower. More and more websites are using responsive design to make pages that work well on desktop, tablet, and mobile devices, regardless of screen size. Some websites serve a special page when they detect that the visitor is using a mobile browser, but the advantage of responsive design is that you don't have to maintain a separate version of your site for desktop and mobile (and tablet, and all of the different size variations of tablet and mobile and desktop) - you have a single website that works well regardless of screen size.

To understand how responsive design works its magic, right-click and use Inspect Element on the sidebar, and make sure the `<div class="col-md-3">` tag is selected. You should see a CSS rule that looks something like:

```
@media (min-width: 992px)
.col-md-3 {
  width: 25%;
}
```

When you resize the page so that it is less than 992 pixels wide, the sidebar will snap up to the top of the row. The CSS rule also disappears from the Chrome inspector.

The **@media** bit is a media query that tells the CSS rule to only apply when certain criteria are met; in this case, that the screen is no narrower than 992 pixels.

If you want to write a rule yourself, it's pretty simple. Here's an example:



```
@media (min-width: 768px) and (max-width: 991px) {  
  .my-class {  
    display: inline;  
  }  
}
```

## Summary

Example using the Bootstrap grid:

```
<div class="container">  
  <div class="row">  
    <div class="col-md-4">  
      I'm a column! Perhaps I'm a right sidebar.  
    </div>  
    <div class="col-md-8">  
      I'm another column! Perhaps I'm the main body text.  
    </div>  
  </div>  
</div>
```

Example media query:

```
@media (min-width: 768px) and (max-width: 991px) {  
  .my-class {  
    display: inline;  
  }  
}
```

# Grid, theme, and media query practice

In the Grid system and responsive design video, we learned:

- The grid system allows us to create rows and columns on our pages
- Responsive design makes pages that work well on desktop and mobile devices
- Media queries tell the CSS rule to only apply when certain criteria are met

Now it's your turn!

Practice using the Bootstrap grid:

- Add columns and rows to your vacation page.
- Rework one of your existing pages to use Bootstrap's grid.

Bootstrap is very popular, but there are several other great alternatives.

- Make a page about your favorite sports team with Skeleton, which is very easy to set up.
- Make a page about your best friend with Foundation, which may take a bit more reading and fiddling to figure out, but which is also very popular.

Practice your responsive design:

- Pick one of your pages that has images on it. Create a class for images that should not be shown on small screens. Add a media query that hides these images (using **display: none**) on smaller screen sizes.

# HTML and CSS code review

Your project: About Me page

Tell your instructor a bit more about yourself! Create a webpage using HTML and CSS to display who you are. You should have the following sections on your page:

- Your name
- An image of you or something that represents who you are
- A list of your top 3 websites with active links to them
  - use CSS to make the links look like links
- Your background (education, job experience, or why did you decide to take a class on programming, etc)
- Your current hobbies/skills
  - use bullet points for these

**Important:** Use CSS to style each HTML selector differently. For example, `<h3>` tags could be italic or red or underlined. Get creative!

Don't forget your Git commits and README!

When you have reviewed your code for all of the objectives, submit a link to your GitHub repository under the the HTML and CSS Code Review on Epicenter.

Below are the objectives the instructor will use to review your code.

## This Week's Code Review Objectives

- Use of all of the HTML tags you have learned so far:
  - p, h, ul, ol, li, em, strong, a, img
- A custom made stylesheet that uses at least 2 HTML selectors, properties and values
- Commits made regularly with clear messages that finish the phrase "It will..."
  - A project README that includes:
    - author name
    - program name
    - description
    - setup instructions
    - link to site on GitHub Pages
    - copyright and license information

# HTML and CSS Project

This week you are going to be given more time and freedom to explore HTML and CSS. You have the tools to create a great-looking static website. Brainstorm ideas of websites that you would like to make using your HTML and CSS skills.

## Project Requirements:

Everyone will be working in a team of 4. You may select your teams, but make sure you join a team with a project that you would like to work on for the entire time. For example, don't do a project that you find boring just because you would be working with your friends. If you need recommendations for a team, ask the instructor.

## Each project should:

- Have a main page and 2-3 subpages that are linked to the main page (remember to put these all in the same folder)
- Make use of all of the HTML tags you have learned so far:
  - p, h, ul, ol, li, em, strong, a, img, div, span
- Have a custom made stylesheet that uses selectors, properties and values
- Use divs and spans to keep the page organized
- Use at least one float
- Make use of the box model to create custom border, padding and margins around an HTML block element
- Use nesting in the stylesheet to add different style to a child element
- Use Bootstrap to add a navigation bar
- Use Git to track your website's progress. Please read and review the Git with teams lesson.

If you have extra time, add a media query to make your website responsive.

**Presentations:** There will be 5 minute presentations of team projects happening on Wednesday during class. Remember that this is a supportive environment and we are here to cheer each other on and learn from each other's struggles and successes. We've got an overhead projector and a whole lot of enthusiasm!

Please briefly cover the following information in your presentation:

- **Project Name and Objective** - Why did you choose this project? What purpose does it fulfill?
- **Team Name/Members** - Who worked on the project? What pieces did each person focus on?

- **Process** - Describe how the team development process worked in your group.
- **Demonstration** - Show and describe your work.
- **Challenges** - Share the biggest challenges you faced.
- **Surprises** - Was there anything that surprised you? What did you learn?

### **Project possibilities**

- Blog site for your interest/hobby with pages for how-to instructions
- University faculty directory with pages for individual faculty members
- Favorite band's website with pages for tour dates and band news
- A zoo's website with pages for each animal type
- Whatever you would like to do!

## More HTML and CSS resources

This is the end of the lessons on HTML and CSS. If you want to continue learning these topics, I'd suggest you go through [Learn CSS Layout](#) and then the [HTML & CSS book](#). Most Opteamize students' weakest skills are in CSS, so this is a good place to stand out!

To continue your learning, here is a compilation of some things we've found:

- [Learn Layout](#)
- [HTML & CSS book](#)
- [A Book Apart](#)
- [Interactive design syllabus](#)
- [Google material design guidelines](#)
- [Responsive web fundamentals course](#)
- [CSS](#)
- [HTML elements](#)
- [HTML attributes](#)

# Introduction to Git

Embedded Video: <https://www.youtube.com/watch?v=r63f51ce84A?rel=0>

## GitLab

GitLab is an online Git repository manager with a wiki, issue tracking, Continuous Integration (CI) and Continuous Deployment (CD). It is a great way to manage git repositories on a centralized server. GitLab gives you complete control over your repositories or projects and allows you to decide whether they are public or private for free.

**What is Git?** Git is a system where you can create projects of different sizes with speed and efficiency. It helps you manage code, communicate and collaborate on different software projects. Git will allow you to go back to a previous status on a project or to see its entire evolution since the project was created. You could think of it as a time machine which will allow you to go back in time to whenever you'd like in your project.

With Git, 3 basic issues were solved when working on projects:

- It became easier to manage large projects.
- It helps you avoid overwriting the team's advances and work. With git, you just pull the entire code and history to your machine, so you can calmly work in your own little space without interference or boundaries.
- It's much simpler and much more light-weight.

**What is Repository?** The place where the history of your work is stored.

**What is Remote repository?** It's a repository that is not-on-your-machine, so it's anything that is not your computer. Usually, it is online, GitLab.com for instance. The main remote repository is usually called "Origin".

**What is Source control or revision control software?** It's a system that records and manages changes to projects, files and documents. It helps you recall specific versions later. It also makes it easier to collaborate, because it shows who has changed what and helps you combine contributions.

**What is Continuous Integration (CI)?** It's the system of continuously incorporating the work advances with a shared mainline in a project. Git and GitLab together make continuous integration happen.

**What is Continuous deployment (CD)?** It means that whenever there is a change to the code, it is deployed or made live immediately. This is in contrast to continuous integration, where code is continuously being merged in the mainline and is always ready to be deployed, rather than actually deployed. When people talk about CI and CD what they usually mean to say is that they are constantly and automatically testing their code against their tests using a tool such as GitLab CI and upon passing to a certain action. That action could be merging the code into a branch (master, production, etc), deploying it to a server or building a package / piece of software out of it. Non-continuous integration would be everyone working on something and only integrating all the work as the very last step. Obviously, that results in many conflicts and issues, which is why CI is adopted widely nowadays.

**What is a Commit?** It's the way you call the latest changes of source code that you made on a repository. When changes are tracked, commits mark the changes on a document.

**What is a Master?** It's how you call the main and definitive branch (the independent line of development of a project).

**What is a Branch?** It's an independent line of development. They are a brand new working directory, staging area, and project history. New commits are recorded in the history for the current branch, which results in taking the source from someone's repository (the place where the history of your work is stored) at certain point in time, and apply your own changes to it in the history of the project.

**What is a Fork?** It's a copy of an original repository (the place where the history of your work is stored) that you can put somewhere else or where you can experiment and apply changes that you can later decide if publishing or not, without affecting your original project.

**What is a Clone?** It's to get a copy of a git project to look at or to use the code.

**What is to Merge?** It's integrating separate changes that you made to a project, on different branches.

**What is markdown?** It's a plain text format that will make any document easy-to-write and easy-to-read.

**What is to Push a repository?** It's to incorporate a local branch (the independent line of development of a project) to a remote repository (online version of your project).

**What is a README.md?** It's a file in a simple format which summarizes a repository. If there's also a README (without the .md), the README.md will have priority.

**What is SSH (secure shell protocol)?** It's how you call the commands that help communicate through a network and that are encrypted and secure. It's used for remote logins and it helps users connect to a server in a secure way.



**What is to stage a file?** It's how you call the act of preparing a file for a commit (the latest changes of source code in a repository).

# README template

Every GitHub repository should include a README.md file that provides the visitor to the repo with the general information about the repository and the code it stores. The README.md file should be stored at the top level of the project folder. GitHub will look for this file and present it on the main viewing page of the repository.

## README Information

Every README should have these five parts:

- A description of the project's purpose
- Complete setup instructions
- Names of contributors
- A copyright with the date
- License information

And could additionally include sections for:

- Technologies used
- Known bugs
- Contact information
- Support or Contribution instructions

## README Template

Here is some README text for you to use in your apps. It's already written in Markdown, so it will be nicely formatted on Github. If you aren't familiar with Markdown check out [Markdown Live Preview](#) to get familiar.

It's worth taking the time to make your README look nice because it will be the first thing anyone will see in any of your projects. If people see that you have a messy or incomplete README, they will assume that your code is also messy or incomplete. But if you have a

README with clean formatting and clear setup instructions, your users will have more confidence in trying out your software.

If you copy and paste the text below into a new file and replace the relevant parts with information about your project, then it will look something like this at the end.

```
# _{Application Name}_

##### _{Brief description of application}, {Date of current version}_

#### By _**{List of contributors}**_

## Description

_{This is a detailed description of your application. Give as much detail as needed to ex

## Setup

* _This is a great place_
* _to list setup instructions_
* _in a simple_
* _easy-to-understand_
* _format_

_{Leave nothing to chance! You want it to be easy for potential employers and collaborato

## Technologies Used

_{Tell me about the languages you used to create this app. Assume that I know you probabl

### Legal

*{This is boilerplate legal language. Read through it, and if you like it, use it. There

Copyright (c) 2015 **_{List of contribtors}**

This software is licensed under the MIT license.

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.
```

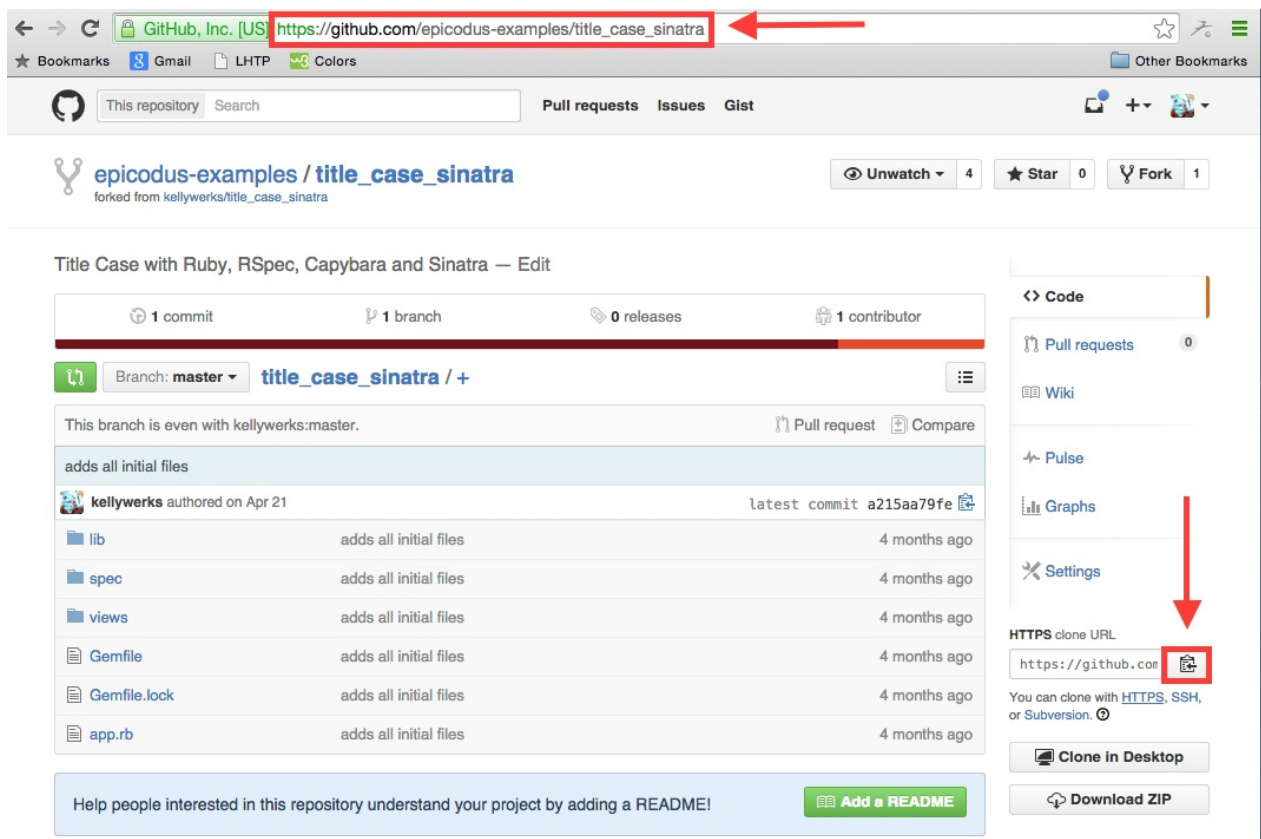
Some other README examples include:

Triangle Tracker - small Epicodus JavaScript app simple\_form - large Ruby app capybara-webkit - really large ruby webkit app

# Git clone

Often, we'll want a local copy of a public repository on our local devices. This is called a clone in Git terminology.

To create a local clone, find the GitHub repository URL and copy it - either from the address bar or with the copy to clipboard button (see red boxes below).



In the terminal, navigate to the directory to store the cloned repository and run git clone with the chosen repo's URL:

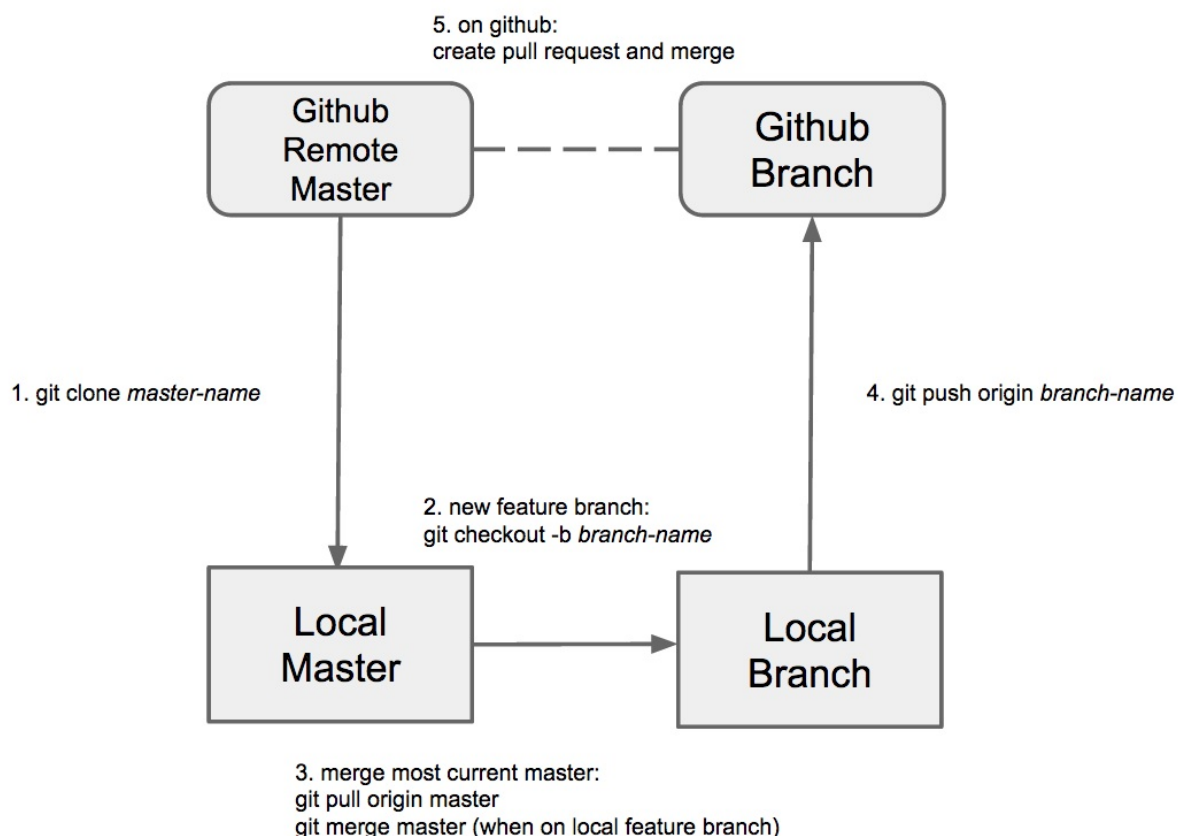
```
$ git clone https://github.com/epicodus_examples/title_case_sinatra
Cloning into 'title_case'...
remote: Counting objects: 14, done.
remote: Total 14 (delta 0), reused 0 (delta 0), pack-reused 14
Unpacking objects: 100% (14/14), done.
Checking connectivity... done.
```

The clone command creates a clone of the remote repository in the current directory, including the initialized .git repository so you can immediately begin working and making commits.

# Git with Teams

To work on a team project, we need to take advantage of some Git functionality that we haven't needed to use previously on our pair projects: branching and merging.

Below is a broad overview of how your changes can be added to the shared repository for your team project.



The workflow for working in a team follows these steps.

1. Build a repo and make all team members collaborators.
2. Clone master to each pair's desktop creating a local master branch.
3. Create a feature branch locally to work on a new feature.
4. Complete work on feature branch.
5. Pull any changes from remote master to local master. If there are changes, merge master into the feature branch and resolve any conflicts.
6. Push updated, completed branch (all tests pass, functionality in place) to remote master.

7. Navigate to branch and create a pull request.
8. Collaborators review new branch code.
9. Merge.
10. Delete feature branch.

## Git Commands Reference

- **git checkout -b branch-name**: creates and checks out a new branch named branch-name
- **git branch**: lists current branches with an asterisk by the current branch
- **git branch -D branch-name**: delete the branch named branch-name
- **git merge branch-name**: merge branch-name into the current branch - **git clone remote-name**: creates a local repository from a remote repository
- **git remote**: lists currently associated remote repositories
- **git remote add remote-name url**: adds a new remote repository with the name remote-name using the URL of the remote repo on github
- **git push remote-name branch-name**: pushes your branch (branch-name) to a remote repository (remote-name)
- **git pull remote-name branch-name**: pull the latest changes from a remote repository to your repository (e.g. git pull origin master)

## Summary

git checkout `-b _branch-name_` : creates and checks out a new branch named branch-name  
git branch: lists current branches with an asterisk to indicate the one you're currently on  
git branch -D `_branch-name_` : delete the branch named branch-name  
git merge `_branch-name_` : merge branch-name into the current branch  
git clone `_remote-name_` : creates a local repository from a remote repository  
git remote: lists currently associated remote repositories  
git remote add `_remote-name_ _url_` : adds a new remote repository with the name remote-name using the URL of the remote repo on github  
git push remote-name `_branch-name_` : pushes your branch (branch-name) to a remote repository (remote-name)  
git pull remote-name `_branch-name_` : pull the latest changes from a remote repository to your repository (e.g. git pull origin master)



## 25. GitHub Pages

**Embedded Video:** <https://www.youtube.com/watch?v=vXXfGddXCnE?rel=0>

It's more fun to make web pages when we can show them off to our friends. There are a lot of ways to put our pages online; GitHub Pages is one of the easiest I've found.

First, make sure to have a GitHub account set up. The awesome thing about GitHub pages is that you get one main site per account and as many project pages as you would like. That means that when you are ready to create a portfolio of all of your coding work, you can have a main page all about you and a page for each of your coding projects that you do here at Epicodus or at home.

A quick note about structuring projects before we begin. When we start a new project, it is a good idea to create a folder to hold the HTML file and the css and img folders. So far, we have been using a descriptive name for our HTML file, such as favorite-things.html or my-first-webpage.html. Often, though, you'll want to give the descriptive name for your folder, and name the web page's main HTML file index.html. As we start to add pages to our web site, the index.html file will serve as the home page of our site.

For this lesson, we will be putting our favorite-things.html file on GitHub Pages. I've made a folder called favorite-things and we'll start in that folder in the Terminal. Since we want favorite-things.html to be the home page of our site, we need to rename it index.html. GitHub Pages won't recognize it as the main page if we don't.

```
$ mv favorite-things.html index.html
```

The mv command in the Terminal moves the content of the favorite-things.html file into a new file called index.html.

Since we have been using Git to track our page's development, let's commit this new change and push it to GitHub.

```
$ git status
$ git add .
$ git commit -m "Change name of HTML file to index.html."
$ git push origin master
```

One of the most important features of a version control system like Git is the ability to create a "branch". We will spend more time with branching and merging later when we begin working in larger groups, but for now, think of creating a branch like making an exact copy of

everything in the project folder. We can experiment with the code - add, delete, move, etc - and it won't change anything in our original "master" branch. In order for GitHub Pages to work, we need to create a branch called gh-pages.

First we will create the new branch and switch into it by running the commands:

```
$ git branch gh-pages  
$ git checkout gh-pages
```

Now that we are in the gh-pages branch, let's push this new branch to GitHub:

```
$ git push origin gh-pages
```

Now our project should be available for viewing at [my-github-username.github.io/repository-name](https://my-github-username.github.io/repository-name). So for this project we will go to [epicodus-lessons.github.io/favorite-things](https://epicodus-lessons.github.io/favorite-things). Let's check it out!

You can also set up the GitHub Pages webpage for your project in the browser by going to your project's repository, clicking the branch drop down on the left, typing `gh-pages` and selecting `Create branch: gh-pages`. Done!

When we start to work in larger groups, we will cover merging our branches with our master branch to keep everything in sync, but until we do that, we'll just use our `gh-pages` branch to make any changes.

If you would like to learn how to use GitHub Pages to create a main page for your personal portfolio, you can check out GitHub's instructions and follow the instructions for the "account/organization" page.

# Git cheat sheet

Basic Git workflow: Create a repository and make your first commit.

Go to the top level of your project folder.

```
cd ~/path/to/your/folder
```

Now that you're inside your project folder, initialize a local repository.

```
git init
```

Configure user settings.

When working with a pair, add each person individually, then add yourselves as a pair using your initials:

```
git pair add mkn Michael Kaiser-Nyman
```

```
git pair add yn Your Name
```

```
git pair mkn yn
```

When soloing:

```
git config user.name "Your Name"
```

```
git config user.email your@email.com
```

Add all your files to the repository so that git knows to keep track of them.

```
git add .
```

Or add just a couple files:

```
git add lib/kitten.rb spec/kitten_spec.rb
```

Commit your changes.

```
git commit -m "Add initial files."
```

Create a remote repository on your github account. If you are pair programming, each person will have to do

this separately so that the project is saved on both people's github accounts.

On the github site, select the "Repositories" tab and then click the green "New" button.

screenshot of button for creating a new repository

Then copy the path that it gives you at the top.

screenshot of path for a repository

Add the remote repository so that your computer knows where to find each person's remote repository by using the git remote add command with the path you just copied from your github repository. The example below adds a remote repository at <https://github.com/dianedouglas/test.git> and names it "diane", and another one at <https://github.com/keebz/test.git> and names that one "lee".

```
git remote add diane https://github.com/dianedouglas/test.git
```

```
git remote add lee https://github.com/keebz/test.git
```

If you are soloing, the standard name to use for a remote repository is "origin".

```
git remote add origin https://github.com/dianedouglas/test.git
```

Push your commit up to the remote repository to be stored on github using the git push command. Enter your github username and password if you are prompted to do so after this command. After the words git push use the name you picked for your remote repository in the previous step, followed by the word master. master is the name for your local repository.

When pairing:

```
git push diane master
```

```
git push lee master
```

Or when soloing:

```
git push origin master
```

That's your first commit on a new repository! Now you do some work. You write your first spec, then make it pass, then you do a little cleanup and refactoring as needed. This would be a good place to make your second commit. Think of it like a save point in a video game.

Let's see how to continue making commits as we work:

Add the changes you have made to your local repository on your computer using the same git add command as when you were setting up.

```
git add .
```

Commit them to the local repository. This is also the same command you used above during setup.

```
git commit -m "Add first spec and make it pass."
```

Push your new commit to the remote repository. This is the same as the last setup step and you should use the same remote name. When pairing:

`git push diane master`

`git push lee master`

Or when soloing:

`git push origin master`

You can also push several commits at once to the remote repository. Repeat steps 1 and 2 above as often as you like to save your changes on your local repository. Any time that you have done something that you would be sad to lose. Then when you are at a good milestone, use step 3 to push your changes up to the remote repository on your Github account.

# JavaScript basics

Embedded Video: <https://www.youtube.com/watch?v=Ptq5bf0TrUg?rel=0>

At this point, you're probably in one of two camps: designer or developer.

If you're a designer, you're incredibly excited by all of this CSS stuff. You have begun to appreciate the power of CSS and can't wait to continue to unravel the mystery of how to make web pages that don't look awful. If that's the case, congratulations! You should continue on the path to be a front-end designer, but unfortunately, the rest of this course is much more heavily weighted towards programming, not design. I'd still recommend you learn programming (especially JavaScript), because it will make you a better designer and help you collaborate better with back-end programmers. But you might want to take a break from this site and continue to work on your HTML and CSS skills.

If you're a developer, you are sick and tired of all of this design stuff. Sure, you like websites to look nice, but you really don't want to spend your time fiddling with colors or fonts - you just want to build something. Well, the good news is, we're about to move on!

If you fall into both camps, that's pretty awesome. You're unusual and your talents are in high demand. I'll continue to teach you the programming side of things, and afterwards, you should make it your business to get good at the design side as well.

Whether you want to focus on programming or design, a great way to continue to learn HTML and CSS is to copy other people. If you see something you like, check it out in the Chrome inspector, and try implementing it yourself. One caveat to this is that there is a lot of bad code out there, so make sure you understand what it is doing, rather than just blindly copying and pasting. Bootstrap is a fantastic framework, so it's a great place to learn from.

Sometimes, you just need a reference on how a certain CSS property or HTML tag should work (such as when you're trying to understand a style that you want to copy!). The Mozilla Developer Network is one of the best sources; it has references on CSS and HTML elements and HTML attributes. Be wary of other sources, as there is a lot of bad information out there.

# JavaScript basics objectives

In this section, we will be learning the basics of JavaScript, the programming language of the web. JavaScript allows us to interact with the webpage's user. When learning any language, we must start with the basic building blocks and for JavaScript these include:

- Arithmetic
- Variables
- Methods
- Strings
- Functions

We will be working almost exclusively in the JavaScript Console this week. Make sure you are familiar with how to access it.

## 02.Arithmetic

Embedded Video: <https://www.youtube.com/watch?v=fDiZAeu7sls?rel=0>

Now that you have the basics of HTML and CSS under your belt, we're going to start learning our first programming language: JavaScript. Remember, web browsers can only do three things: display content with HTML, style it with CSS, and change what's displayed with JavaScript. Because all browsers support JavaScript, it's the de facto programming language of the web.

We're going to take a detour away from the aesthetically pleasing and obviously useful world of web pages, so that we can learn some programming fundamentals. But don't worry! We'll be back in no time.

Like earlier, I'm going to assume you're using Google Chrome for these lessons. Chrome includes a nice little console that lets you type in JavaScript code. To access it, click on the ☰ button, go to the Tools menu, and click JavaScript Console. As a shortcut, you can also press `Cmd + Alt + J`.

Here at the `>` prompt, you can type JavaScript and see what it does. Try typing `1 + 2`; and pressing Enter.

Congratulations! You just ran your first JavaScript code.

You should see something like:

```
1 + 2;
```

```
3
```

JavaScript took your code - `1 + 2` - ran it, and returned the result - `3`.

The semicolon at the end of the line tells JavaScript to run everything before it. You'd think JavaScript would know on its own, wouldn't you?

Now try these:



```
4 - 3;  
  
5 * 6;  
  
10 / 2;  
  
9 / 2;  
  
7 + 8 * 9;  
  
(7 + 8) * 9;
```

Try some other arithmetic yourself. Play around with the `%` operator, called **modulo**. It will give you the remainder of dividing two numbers. Don't be fooled into thinking it has something to do with percentages!

`+`, `-`, `*`, and `/` are called **operators**. An operator is a special character (or characters) that indicates an action to be performed.

Try dividing 0 by 0. The result, `NaN`, stands for not a number. In JavaScript, `NaN` is actually considered a type of number (bizarre as that may seem).

Try dividing any other number by 0. The result, `Infinity`, is also a number in JavaScript.

## Summary

Cmd + Alt + J opens the JavaScript console in Chrome.

Basic arithmetic works just like you'd expect:

```
1 + 2;  
4 - 3;  
5 * 6;  
9 / 2;  
7 + 8 * 9;  
(7 + 8) * 9;
```

`+`, `-`, `*`, and `/` are called operators. An operator is a special character (or characters) that indicates an action to be performed.

`9 % 2` ; returns the remainder of 9 divided by 2. `%` is called modulo.

`0 / 0` ; returns `NaN`, which stands for not a number. `NaN` is a type of number.

`Infinity` is also a number.



## 03.Arithmetic practice

In the Arithmetic video, we learned:

- How to access the JavaScript Console
- Addition, subtraction, multiplication, and division using JavaScript
- How to use the `%` operator
- `NaN` means not a number but is actually considered a type of number

Here are some exercises for you to practice arithmetic in JavaScript:

- Add:
  - 77 to 99
  - 34 to 1233
  - -53 to 99
  - 9,092 to -12312
  - 943,456,575 to 39,087,092,348,570
- Subtract:
  - 99 from 665
  - 555 from 23
  - 7,912 from 88888
  - 6,348,709,234 from 87,023,984,709,871,234.
- Multiply:
  - 89 and 54
  - 932 and 1,900
  - -34 and 8
  - 25 and 700
- Divide:
  - 1008 by 7
  - 23423 by 75
  - 90 by 3
  - 9,870,834,205,987 by 324
- Find the remainder of dividing the following numbers:
  - 1008 by 7
  - 23423 by 75
  - 90 by 3
  - 9,870,834,205,987 by 324
- Divide a number by 0.
- Divide 0 by 0.

## 04.Variables

Embedded Video: <https://www.youtube.com/watch?v=G3kiWW8BBdk?rel=0>

Try this:

```
var myNumber = 45;

myNumber;
```

`myNumber` is a **variable** that we set equal to 45. On the second line, when we type `myNumber` and press Enter, JavaScript returns the value of the variable. In JavaScript, it's conventional to give variables names that start with a lowercase letter, and if they are more than one word, to capitalize the first letter of subsequent words. This is called lower camelcase, because the capitalized letters look like a camel's humps... or something.

Sometimes you'll see a variable initially set without the `var` keyword, like this:

```
myNumber = 45;
```

This works... most of the time, but it can cause really bad bugs down the road. Get in the habit of using the `var` keyword whenever you create a new variable, and you'll save yourself some massive headaches later.

You can change the value of a variable:

```
var myNumber = 45;

myNumber;

myNumber = 50;

myNumber;
```

You can do arithmetic with your variables:

```
var favoriteNumber = 13;

favoriteNumber * 4;
```

Does the variable change when you perform arithmetic on it?

```
var favoriteNumber = 7;

favoriteNumber + 1;

favoriteNumber;
```

Nope. But what if we do:

```
var favoriteNumber = 6;

favoriteNumber = favoriteNumber + 1;

favoriteNumber;
```

Here's a shortcut: `favoriteNumber += 1` .

You can use more than one variable at a time, too:

```
var num1 = 5;

var num2 = 6;

var num3 = num1 + num2;

num3;
```

## Summary

Set a variable equal to a number:

```
var myNumber = 45;
```

Use lower camelcase for variable names.

Use a variable without modifying its value:

```
favoriteNumber * 4;
```

Use a variable and modify it:

```
favoriteNumber = favoriteNumber * 4;
```

Shortcut:

```
favoriteNumber *= 4;
```

Use multiple variables:

```
var myNumber = 45;  
var otherNumber = 12;  
myNumber + otherNumber;
```

## 05.Variables practice

In the Variables video, we learned:

- Variables, in JavaScript, are written in lower camelcase
- Variables are set using the var keyword
- Variables can be used in arithmetic

Here are a few exercises for you to practice using variables:

- Set a variable called **someName** equal to your name. Put your name in quotation marks.
- Display the value of **someName** in the console.
- Change the value of **someName**.
- Set a variable called **favoriteNumber** equal to your favorite number.
- Calculate what your favorite number divided by 2 is.
- Set another variable called **michaelsFavorite** equal to **13**.
- Change the value of **michaelsFavorite** to **7**.
- Subtract your favorite number from mine.
- Change the value of my favorite number to be 26 times its current value.

## 06.Methods

**Embedded Video:** <https://www.youtube.com/watch?v=fBZvoT9WfMs?rel=0>

Now that you've got the basics of numbers down, let's learn how to manipulate them a bit. First, let's change a number into exponential notation.

If you're not familiar with exponential notation, here's how it works. Take the number 78.5. In exponential notation, we write it *7.85 10*. Or take 356.97; in exponential notation, that's *3.5697 10<sup>2</sup>*. You can also write it as 3.5697e+2.

Exponential notation makes it easy to write very large or very small numbers. For example, 1,000,000,000 becomes 1e+9, and 0.00000002 becomes 2e-8.

We can use JavaScript to easily change numbers into exponential notation:

```
>48432.78.toExponential();  
  
"4.843278e+4"
```

`toExponential()` is called a method. You can think of a method as a message that you send to a number, and the result that JavaScript gives you as the number's response to that message.

You can also go in the other direction, and convert out of exponential notation:

```
>4.587e2.toFixed();  
  
"459"
```

The `toFixed()` method will round to the nearest whole number. Here's how we can tell it how many decimal places to use:

```
46.1.toFixed(2);  
  
46.10
```

The **2** in the parentheses is an **argument** to the `toFixed()` method. Arguments provide a bit more information to methods to help them know what they're supposed to do. In this case, the argument is optional. When a method doesn't take an argument, or when the argument is optional and you aren't using it, you still need the parentheses on the end - so you have write `1.05e3.toFixed()`, not `1.05e3.toFixed`.



Here's another method that takes an argument:

```
>8.12345.toPrecision(4);  
  
8.123
```

You can also call a method on a variable, since the variable is just standing in for the number it represents:

```
>var myNumber = 8.12345;  
  
>myNumber.toPrecision(4);
```

## Summary

A method is a message you send, and the result is the response.

Call the `toExponential()` method on a number to convert it to exponential notation:

```
48432.78.toExponential();
```

Call `toFixed()` to convert from exponential:

```
4.587e2.toFixed();
```

Put arguments in the parentheses after a method:

```
46.1.toFixed(2);  
8.12345.toPrecision(4);
```

Remember to put parentheses after every method, even if you are not putting an argument in them.

You can call methods on variables:

```
var myNumber = 8.12345;  
myNumber.toPrecision(4);
```

## 07.Methods practice

In the Methods video, we learned:

- How to manipulate numbers with methods
- Exponential notation
- The argument to the method is located within the parentheses
- Methods, in JavaScript, are always followed by parentheses

Practice using the methods you've learned about:

- Convert the following to exponential notation:
  - 9238.479
  - 98370923874.32
  - 324.64322
- Convert them back.
- Specify the number of decimal places each number should have.
- Repeat all of these using a variable set equal to any number you choose.

## 08.Strings

**Embedded Video:** <https://www.youtube.com/watch?v=fO2HLxkzISg?rel=0>

We've done a lot with numbers, but there's more to this world than just math (thank goodness). Let's look at words.

```
"Hello world!";
```

The stuff inside the quotes is called a string. Strings can include letters, punctuation, and even numbers. These are all strings:

```
"5674";  
"!?!";  
"Strings are crazy! ;)";
```

What happens if we don't surround our string with quotes?

```
hello;
```

This doesn't work because JavaScript doesn't know what hello means when it's not a string. It looks to see if it's a variable or something similar, and then doesn't find it defined anywhere.

We can set variables equal to strings:

```
var myString = "Strings can contain characters like @, $, and %.";  
  
myString
```

You can call methods on strings, just like you can with numbers:

```
"supercalifragilisticexpialidocious".toUpperCase();
```

Or even call the method on a variable assigned to a string:

```
var word = "foo";  
  
word.concat("bar");
```

Methods can be **chained** like this:

```
"foo".concat("bar").toUpperCase();
```

The `concat()` method (which concatenates, or combines, two strings) returns a string, which then has `toUpperCase()` called on it. Then, `toUpperCase()` returns the final result.

By the way, here's a nice shortcut for the `concat()` method:

```
"I love" + " " + "Opteamize";
```

Back to arguments. String methods can take numbers as arguments, too:

```
"caterpillar".charAt(5);
```

Did you notice something funny about this example? What's the fifth character of the word "caterpillar"? Well, with the way we normally count, "c" is the first letter, and "r" is the fifth. But JavaScript says that the character at the 5th position is "p". That's because computers start counting at zero. So "c" is the zeroth letter, and "r" is the fifth.

If you want to put a quote inside a string, you have two options. Here's the first:

```
"Quoth the raven, \"Nevermore.\"";
```

The `\\` is called an **escape**. It tells JavaScript the the " that comes right after it is not the end of the string, but just a character inside the string.

You can also use single quotes:

```
'"Programming is fun!", she exclaimed.';
```

JavaScript generally doesn't care if you use single or double quotes to indicate a string. If you have a lot of double quotes within the string, using single quotes to indicate the string saves you from having to escape all of the double quotes inside.

I could describe many more of the methods available to strings, but I'm instead going to refer you the Mozilla Developer Network. The MDN is one of the best sources of documentation on JavaScript, and you should start to get familiar with it. Take a look at their documentation of string methods.

By the way, MDN is awesome, but you should be wary of other resources you find online. There is a lot of bad information out there, especially on JavaScript, so you should always look to MDN first.

Stack Overflow is another good resource; it's a site for asking and answering questions about programming. You do need to be careful about what you find there, as there is plenty of bad information, but it's a great resource. Just make sure you search to see if somebody has already answered your question before posting. And give back by answering questions for other people!

Finally, if you're stuck on something, chances are, somebody has gotten stuck on it, too - try a search engine. To make sure you get up-to-date results, may find it helpful to limit your results to the past year (in Google, click Search tools in the bar above your search results, and then change All time to Past year).

## Summary

```
"Strings can contain letters, numbers like 1, 14.5, and 5784329, and characters like @, $
```

You can call methods on strings, or variables assigned to strings:

```
"supercalifragilisticexpialidocious".toUpperCase();  
var word = "foo";  
word.concat("bar");
```

Methods can be chained (for strings, numbers, or anything else):

```
"foo".concat("bar").toUpperCase();
```

A few useful string methods:

- `charAt()`;
- `toUpperCase()`;
- `toLowerCase()`;
- `concat()`;

Combine strings with `+` instead of `concat()` :

```
"I love" + " " + "Opteamize";
```

Include quotes in strings by either escaping them with `\"` , or by using single quotes outside:

```
"Quoth the raven, \"Nevermore.\"";  
'"Programming is fun!", she exclaimed.';
```

For reference, use Mozilla's documentation of string methods.

Ask and answer questions on Stack Overflow.

Search for other people who had your same problem. Limit results to the past year: in Google, click Search tools in the bar above your search results, and then change All time to Past year.

## 09.String practice

In the Strings video, we learned:

- Strings can include letters, punctuation and numbers
- Strings are surrounded with quotation marks
- Variables can be set equal to strings
- Methods can be called on strings
- Methods can be chained
- To escape characters use the `\` character

Practice using strings:

- Type a greeting to your best friend. Set a variable equal to a string.
- Call a method on that variable.
- Set another variable with a string in all uppercase.
- Use the `concat` method with your two variables.
- Now concatenate them with the shortcut.
- Use the `toUpperCase` method on your first variable.
- Now use the `toLowerCase` method on your second variable.
- Find the character at the 3rd position either of your variables.
- Make sure you're clear on the difference between `"9"` with quotes and `9` without.
- Check out the MDN's documentation of string methods and try out at least five new methods for yourself. If some of them don't make sense, don't worry about it - just try the ones that do. (Stick to the section labelled Methods unrelated to HTML. Click on the method itself to see how to use it in your code.)

# 10.Functions

Embedded Video: <https://www.youtube.com/watch?v=Esb3XRJwu5w?rel=0>

So far, none of the JavaScript we've learned has let us do anything with a web page. Let's try something a little more interesting:

```
alert("Hello world!");
```

Wow! When you press enter, the page pops open a dialog box that says "Hello world!". Exciting stuff!

`alert()` is a **function**. A function is something that performs an action. Just like a method, a function can take an argument. The `alert()` function pops up a dialog box with the string that you pass in as an argument.

Here's another JavaScript function:

```
prompt("What is the air-speed velocity of an unladen swallow?"); // I'm going to type "Af  
"African or European?"
```

In JavaScript, everything after the `//` is a **comment**, and is ignored. Comments are a convenient way to leave notes in your code for yourself or other programmers.

This dialog box lets you type in a response, and then that response is returned from the function. One cool thing we can do here is to set a variable equal to the response, like this:

```
var favoriteColor = prompt("What is your favorite color?"); // I'm going to type "green"  
  
favoriteColor;  
  
"green"
```

Just like we could chain methods to each other, we can also chain methods to a function:

```
prompt("Type something in lowercase:").toUpperCase(); // I'm going to type "lowercase"  
  
"LOWERCASE"
```



Try the `confirm()` function yourself, now.

You should have seen that `confirm()` returns one of two values: `true` or `false`. Notice that there are no quotes around these values. `true` and `false` aren't strings - they're called **booleans**. They simply represent being true or false.

And on that note, you might have noticed that `alert()` returned `undefined`, also without quotes. `undefined` simply represents that nothing has been returned from the function, or that a variable hasn't been assigned a value.

Having now used numbers, strings, booleans, and `undefined`, you've come across four of the five basic JavaScript data types, or **primitives**. The last one is `null`, which represents nothingness. Don't worry about `null` for now - we'll learn more about it down the road.

## Summary

A function is something that performs an action.

A boolean is `true` or `false`.

- `alert()` opens a dialog box and returns `undefined`.
- `confirm()` opens a dialog box and returns a boolean.
- `prompt()` opens a dialog box and returns a string. The five JavaScript primitives are numbers, strings, booleans, `undefined`, and `null`.

Everything after `//` is a comment and ignored by JavaScript.

# 11.Functions practice

In the Functions video, we learned:

- Functions perform actions in the browser
- Examples of JavaScript functions are `alert()` , `prompt()` , and `confirm()`
- `//` precedes comments in JavaScript
- Methods can be chained onto functions
- `true` and `false` are booleans, not strings
- `undefined` means that nothing has been returned from the function or a variable hasn't been given a value
- Data types in JavaScript are called primitives. Practice using JavaScript functions:
- Use `alert()` to pop up a dialog box with a warning for the user.
- Use `confirm()` to ask a yes or no question.
- Use `prompt()` to ask a question.
- Save the response to the `prompt()` as a variable, and then run at least 3 string methods of your choice on it.
- Ask a new question and save the response as a new variable and run 3 different string methods on it.

## 12. Writing functions

Embedded Video: <https://www.youtube.com/watch?v=EE4MarV56IU?rel=0>

Now that we know how to use a couple JavaScript functions, let's write some ourselves.

We're going to write a very simple function to start:

```
var sayHi = function() { alert('Hello from Epicodus!'); };  
  
sayHi();
```

`sayHi()` simply pops up a dialog box with the text Hello from Epicodus!. This isn't terribly useful, so let's write a slightly more interesting function:

```
var saySomething = function(whatToSay) { alert(whatToSay); };  
  
saySomething("hello");
```

As you know, we call `"hello!"` an **argument** to the function `saySomething()`. In the `saySomething()` function, that argument is assigned to the variable `whatToSay` - we call that variable a **parameter**. If you're confused about the difference between arguments and parameters, just remember that the argument is the information you pass in, and the parameter is the variable that receives the argument. In this case, `"hello!"` is the argument, and `whatToSay` is the parameter. The parameter can then be used just like any other variable.

Okay, on to another, slightly more complex function:

```
var add = function(number1, number2) { return number1 + number2; };  
  
add(3, 5);
```

The `return` keyword tells JavaScript to return the result from that line of code. Without a `return`, JavaScript will return `undefined` from the function, which is JavaScript's way of saying something does not have a value.

Let's step through exactly what happens if we call `add(3, 5)`:

1. We call the `add` function with the arguments `(3, 5)`.
2. The `add` function is run. The parameter `number1` is set equal to 3, and the parameter `number2` is set equal to 5.

3.  $3 + 5$  is run.
4. The result of  $3 + 5$  is returned.
5. The add function ends.

Notice our variables names: `number1` and `number2`. We could have called them `n1` and `n2`, and it would have taken less typing. But the name `number1` very clearly expresses what the variable is for, whereas `n1` will require another programmer (or your future self, when you come back to your code in a few months and don't remember exactly how it works) to figure it out from context. In this example, it would be pretty obvious what `n1` is for. But if you practice choosing descriptive names now and resisting the temptation to abbreviate, you will be in the habit of doing it when you're writing more complex code where it matters more.

## Summary

Example function:

```
var add = function(number1, number2) { return number1 + number2; };  
add(1, 2);
```

An **argument** is what you pass into a function or method; a **parameter** is a variable that's assigned to the argument. Above, `number1` and `number2` are parameters, and `1` and `2` are arguments.

Use descriptive variable names so that your code is easily readable. Don't be tempted to abbreviate.

## 13.Practice writing functions

In the Writing functions video, we learned:

- Arguments to functions can be assigned to variables called parameters
- The `return` keyword tells JavaScript to return the result from the line of code
- Don't abbreviate variable names because it can get confusing after time has passed

Write some functions of your own:

- Take somebody's name and say a greeting to them;
- Write a function to subtract two numbers;
- Now one to multiply two numbers. Then create a new function called `threeTimes` to multiply three numbers together.
- Now write one to divide two numbers. Then write a new function called `remainder` to find the remainder of two numbers.
- Prompt the user to enter their age, another to enter their name and another to enter their favorite food. Then return an alert with a sentence that combines all of this information.
- Write a function that calculates BMI using two arguments. Then write another function that prompts the user for their height and weight, uses the BMI function and alerts the user of their BMI.

# 01.JavaScript Branching and Looping Objectives

In the JavaScript branching and looping section, we will learn how to enable our webpages to make decisions based on the current state of the page or on user input. At the end of the section, we will learn how to debug to help us figure out what is going on when our JavaScript isn't working.

## Overview of the week

- **Branching** - allows JavaScript to perform different actions based on those different states or inputs
- **Looping** - allows JavaScript to repeat an action until some condition is met
- **Arrays** - lists of data or information

## 02.Branching

Embedded Video: <https://www.youtube.com/watch?v=xoOOnfrlb08?rel=0>

Now that we've learned the basics of how to manipulate web pages with jQuery, let's add some logic to our pages with branching. Here's an example:

### drinks.html

```
<!DOCTYPE html>
<html>
  <head>
    <link href="css/bootstrap.css" rel="stylesheet" type="text/css">
    <link href="css/styles.css" rel="stylesheet" type="text/css">
    <script src="js/jquery-1.10.2.js"></script>
    <script src="js/scripts.js"></script>
    <title>Drinks</title>
  </head>
  <body>
    <div class="container">
      <div id="drinks">
        <h1>Drink menu</h1>

        <h2>Beer</h2>
        <ul>
          <li>The King of Beer</li>
          <li>The Queen of Beer</li>
          <li>Real cold beer</li>
        </ul>

        <h2>Wine</h2>
        <ul>
          <li>Red wine</li>
          <li>White wine</li>
          <li>Blue wine</li>
        </ul>
      </div>

      <div id="under-21">
        <h1>Sorry, we can't serve you.</h1>

        <p>It's illegal in the US. Try Mexico or Europe.</p>
      </div>
    </div>
  </body>
</html>
```

### styles.css

```
#drinks, #under-21 {  
  display: none;  
}
```

### scripts.js

```
$(document).ready(function() {  
  var age = parseInt(prompt("How old are you?"));  
  
  if (age >= 21) {  
    $('#drinks').show();  
  } else {  
    $('#under-21').show();  
  }  
});
```

Since `prompt()` returns a string, we have to change it into a number using the `parseInt()` function. (The Int bit is short for integer, which means it's a whole number. If we wanted to convert something into a number with a decimal, we'd use `parseFloat` - floating point numbers are numbers with decimals.)

The new part of the JavaScript is called an `if...else` statement, or a **branch**. If the condition in the parentheses is true, then the first statement is run. If not, the statement following the `else` keyword is run.

The curly braces after the `if` statement don't end with a semicolon. This is because an `if` statement isn't a statement itself; rather, it's a collection of statements. Or something like that. Just know that it's a rule: no semicolons after the curly braces for `if` statements.

Also, once again, note the indentation: everything that inside the `if` and `else` gets indented in two spaces.

You can even make the logic a little more complex:

### scripts.js



```
$(document).ready(function() {  
    var age = parseInt(prompt("How old are you?"));  
  
    if (age > 21) {  
        $('#drinks').show();  
    } else if (age === 21) {  
        alert("Now don't go crazy!");  
        $('#drinks').show();  
    } else {  
        $('#under-21').show();  
    }  
});
```

Notice the triple equals operator. When we're asking whether something is equal, we use `===` (3 equal signs). When we're setting a variable equal to something, we use `=`. Mixing these up is one of the easiest syntax errors to make.

JavaScript also has an operator with 2 equal signs, but it is almost never used, and you should generally avoid it. It does things like return `true` for `"2" == 2`, but many of its rules are confusing, inconsistent, and hard to remember.

When JavaScript tries to figure out if the condition is true, it's looking for a boolean. Remember booleans - `true` and `false`? They were returned from the `confirm()` function. Check out what's going on here in the JavaScript console:

```
var age = 22;  
  
age > 21;
```

We're getting a boolean, just like with `confirm()`. We could write the JavaScript for our drinks page like this:

### scripts.js

```
$(document).ready(function() {  
    var over21 = confirm("Are you over 21? Click OK for yes or Cancel for no.");  
  
    if (over21) {  
        $('#drinks').show();  
    } else {  
        $('#under-21').show();  
    }  
});
```

In other words, when I earlier said "If the condition in the parentheses is true, then the first statement is run," it's more precise to say "If the condition in the parentheses evaluates to `true`, then the first statement is run." (Note: this isn't 100% correct, as we'll see in the next lesson, but don't worry about that for now.)

## Summary

One branch:

```
if (age > 21) {  
  $('#drinks').show();  
}
```

Two branches:

```
if (age > 21) {  
  $('#drinks').show();  
} else {  
  $('#under-21').show();  
}
```

Three branches:

```
if (age > 21) {  
  $('#drinks').show();  
} else if (age === 21) {  
  $('#drinks').show();  
} else {  
  $('#under-21').show();  
}
```

Branching can use a variable whose value is a boolean:

```
var over21 = true;  
if (over21) {  
  $('#drinks').show();  
}
```

Comparison operators: `===`, `>`, `<`, `>=`, `<=`.

Comparison operators return booleans:

```
3 > 2;  
// returns true
```

`=` sets a variable; `===` compares two things. Don't use `==` .

## 03.Branching Practice

In the Branching video, we learned:

- How to use an `if..else` statement to add logic to our webpages.
- The difference between `===` and `=`

Now, choose **two** of the following webpages to make that incorporates branching:

- A page with information about voting that displays different information to minors. Add in links to websites where young voters can go to learn about the voting process.
- A page with information about turtles, snakes, and insects that asks you which animal you'd like to learn about. When a user chooses which animal to learn about, take them to a page with information (including pictures) about that animal.
- A page for an amusement park that only shows you information on rides that you're tall enough to go on. Then try showing a list of all of the rides in the park and just highlighting (with CSS) the ones that they can ride on. Make sure there are some rides that an adult can't ride on because they are too tall.

## 04. More Branching

Embedded Video: <https://www.youtube.com/watch?v=aVHS8RSI54Y?rel=0>

Sometimes, our branching logic can get more complicated. Let's build a website to give car insurance quotes. We'll keep it simple to start:

### car-insurance.html

```
<!DOCTYPE html>
<html>
  <head>
    <link href="css/bootstrap.css" rel="stylesheet" type="text/css">
    <link href="css/styles.css" rel="stylesheet" type="text/css">
    <script src="js/jquery-1.10.2.js"></script>
    <script src="js/scripts.js"></script>
    <title>Car insurance</title>
  </head>
  <body>
    <div class="container">
      <h1>Car insurance</h1>

      <p>To give you a quote, we need to know a little about you:</p>

      <form id="insurance">
        <label for="age">Age</label>
        <input id="age" type="text">
        <label for="gender">Gender</label>
        <select id="gender">
          <option value="female">Female</option>
          <option value="male">Male</option>
        </select>

        <button type="submit" class="btn">Calculate my quote</button>
      </form>

      <div id="quote">
        <p>You will have to pay $<span id="rate"></span> per month to get car insurance.
      </div>
    </div>
  </body>
</html>
```

### styles.css

```
.btn {  
  display: block;  
}  
  
#quote {  
  display: none;  
}
```

### scripts.js

```
$(document).ready(function() {  
  $("form#insurance").submit(function(event) {  
    var age = parseInt($("#input#age").val());  
    var gender = $("#select#gender").val();  
  
    var quote = (100 - age) * 3;  
  
    $("#rate").empty().append(quote);  
    $("#quote").show();  
  
    event.preventDefault();  
  });  
});
```

Hopefully it's pretty clear how this works. Now, insurance companies generally charge more to men under 26. Let's add \$50 to the quote for this demographic:

### scripts.js

```
$(document).ready(function() {  
  $("form#insurance").submit(function(event) {  
    var age = parseInt($("#input#age").val());  
    var gender = $("#select#gender").val();  
  
    var quote = (100 - age) * 3;  
    if (gender === 'male' && age < 26) {  
      quote += 50;  
    }  
  
    $("#rate").text(quote);  
    $("#quote").show();  
  
    event.preventDefault();  
  });  
});
```

Here, we've used the `&&` (**and**) operator. The expression `gender === 'male' && age < 26` only evaluates to `true` if both conditions evaluate to `true`. You can try it out in the JavaScript console to see for yourself.

If we decide that we actually want to charge the premium to all men, and to anyone who's under 26, we can use the `||` (**or**) operator:

### scripts.js

```
if (gender === 'male' || age < 26) {  
  quote += 50;  
}
```

The expression `gender === 'male' || age < 26` evaluates to `true` if either condition evaluates to `true`.

One thing I don't like about our page is that it doesn't give an error if the user forgets to enter their age - it just treats it like their age is 0. Let's give an error if they leave the age field blank:

### scripts.js

```
$(document).ready(function() {  
  $("form#insurance").submit(function(event) {  
    var age = parseInt($("#input#age").val());  
    var gender = $("#select#gender").val();  
  
    if (age) {  
      var quote = (100 - age) * 3;  
      if (gender === 'male' && age < 26) {  
        quote += 50;  
      }  
  
      $("#rate").empty().append(quote);  
      $("#quote").show();  
    } else {  
      alert('Please enter your age.');    }  
  
    event.preventDefault();  
  });  
});
```

Does this strike you as a little funny? Before now, all of the conditions for our `if` statements have evaluated to booleans. This time, we're looking at a string. If the user doesn't input a value, jQuery will return an empty string (with the car insurance page open,

type `$("input#age").val();` in the JavaScript console to see for yourself). An empty string isn't `true` or `false`.

What's happening here is that JavaScript has concepts called `truthy` and `falsey`. Falsey values include empty strings, the number `0`, the number `NaN`, `undefined`, `null` (which we still haven't seen), and, of course, `false` itself. If JavaScript sees any of these as a branching condition, it will treat them as false. Everything else is truthy.

Here's another operator you should know about: if you want to test that something is not true, you can use the `!` (**not**) operator like this:

```
if (!under18) {  
  // do something only adults can do  
}
```

And finally, here's a trick you'll commonly see. This code:

```
if (something > 0) {  
  return true;  
} else {  
  return false;  
}
```

can be written simply as:

```
return something > 0;
```

That's because the condition `( something > 0 )` evaluates to a boolean - either true or false. So we can just return that Boolean.

## Summary

Logical operators:

- `&&` means and: `gender === 'male' && age < 26`
- `||` means or: `gender === 'male' || age < 26`
- `!` means not: `!under18` Empty strings, the number `0`, the number `NaN`, `undefined`, `null`, and `false` itself are **falsey**. If JavaScript sees any of these as a branching condition, it will treat them as false. Everything else is truthy.



```
if (something > 0) {  
  return true;  
} else {  
  return false;  
}
```

can be written as:

```
return something > 0;
```

## 05. More Branching Practice

In the More branching video, we learned:

- The `&&` and the `||` operator
- The JavaScript values of `truthy` and `falsey`
- When to return the boolean of the condition rather than writing an `if...else` statement

Build the following websites to practice your branching skills:

- A celebrity dating webpage, where you enter information about yourself, and the page suggests which celebrity you should date.
- A political beliefs meter that asks you questions about your values and tells you how liberal or conservative you are.

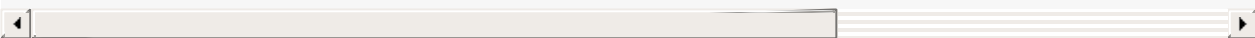
Make sure you code to check for fields left blank on these pages. If you're checking multiple fields, don't just pop up a dialog box for each one - that would be annoying. Instead, add some text by the field that needs to be fixed. Bootstrap has some nice styles for this - check out their form documentation, in the last section titled `Validation states`.

## 06.Arrays

Embedded Video: <https://www.youtube.com/watch?v=2PQ9vDuKjsA?rel=0>

Until now, we've always dealt with one piece of information at a time: one number, one string, one element on a page. But often, we need to group things together. For example, what if we wanted to have a list of the months of the year? We'd use an **array**, which is just a list of things grouped together. An array looks like this:

```
var months = ["january", "february", "march", "april", "may", "june", "july", "august", "
```



Here are some more arrays for you to try in the JavaScript console:

```
[2, 5, 7, 3423, 85, 65]
```

```
["e", "p", "i", "c", "o", "d", "u", "s"]
```

```
["word", 45, "blah", 123]
```

You can put variables and expressions in your arrays, or even other arrays:

```
var variable = "i'm a variable";
```

```
[variable, "i'm not a variable"]
```

```
[62, 62 / 2]
```

```
["string", 123, ["another string", 456], 321, "yet another string"]
```

Just like numbers and strings, arrays have methods. Here are a couple for you to start with:

```
["apple", "orange", "kiwi"].pop();
```

```
"kiwi"
```

```
[1, 2, 3].reverse();
```

```
[3, 2, 1]
```

You can also add elements to an array, or combine two arrays:

```
var greetings = [];  
  
greetings.push('hi');  
  
greetings.push('hello');  
  
greetings.concat(['hola', 'buenos dias']);
```

Note that `concat()` doesn't actually change the original array, but rather returns a new array that combines the two.

If you want to access an element from an array, the syntax is a bit different from anything we've seen before:

```
var letters = ['e', 'p', 'i', 'c', 'o', 'd', 'u', 's'];  
  
letters[0]  
  
letters[4]
```

Just like with strings, we count array elements starting with zero. So the zeroth element of the array is `'e'`, and the fourth element is `'o'`.

## Summary

Arrays can have just about anything as elements:

```
["string", 123, true, ["another string", 456], 321, 52 / 3]
```

Common array methods:

- `pop()` : remove the last element
- `push()` : add an element to the end
- `concat()` : combine with another array
- `reverse()` Access elements using square brackets:

```
var letters = ['a', 'b', 'c'];  
letters[0] // returns 'a'
```

Start counting elements at 0.

## 07.Arrays Practice

In the Arrays video, we learned:

- Arrays can contain numbers, strings, variables, expressions and other arrays
- Arrays have methods like `pop()` , `reverse()` , and `push()`
- How to access an element from an array

Play around with arrays of your own:

- Make an array of your siblings' names or your favorite movie characters' names.
- Make an array of your parents' names.
- Combine these two arrays using `concat()` .
- Add your pets' names using `push()` .
- Reverse the order of the array.
- Access one of your parents' names using the square bracket notation.
- Add the name of your hometown using the square bracket notation.
- Check out methods from the MDN (stick to the accessor and mutator methods sections). Try at least five of them.

# JavaScript Reserved Words

In JavaScript, some identifiers are reserved words and cannot be used as variables or function names.

## JavaScript Standards

ECMAScript 3 (ES3) was released in December 1999.

ECMAScript 4 (ES4) was abandoned.

ECMAScript 5 (ES5) was released in December 2009.

ECMAScript 6 (ES6) was released in June 2015, and is the latest official version of JavaScript.

Time passes, and we are now beginning to see complete support for ES5/ES6 in all modern browsers.

## JavaScript Reserved Words

In JavaScript you cannot use these reserved words as variables, labels, or function names:

<b>abstract</b>	<b>arguments</b>	<b>boolean</b>	<b>break</b>	<b>byte</b>
case	catch	char	class*	const
continue	debugger	default	delete	do
double	else	enum*	eval	export*
extends*	false	final	finally	float
for	function	goto	if	implements
import*	in	instanceof	int	interface
let	long	native	new	null
package	private	protected	public	return
short	static	super*	switch	synchronized
this	throw	throws	transient	true
try	typeof	var	void	volatile
while	with	yield		

Words marked with\* are new in ECMAScript5

## 08.Looping

Embedded Video: <https://www.youtube.com/watch?v=cN5nXKjGIqc?rel=0>

Now that you've got the basics of arrays under your belt, let's turn to a more advanced concept: **looping**. Here's a very simple loop:

```
var languages = ['HTML', 'CSS', 'JavaScript'];
languages.forEach(function(language) {
  alert('I love ' + language + '!');
});
```

To type multiple-line statements into the JavaScript console, you can hold down Alt while you press Enter, so that it knows to create a new line (instead of running the line, as it does by default when you press Enter by itself). You can also use JSFiddle, a site that lets you easily play around with writing and running JavaScript.

Let's step through how this works:

1. Create an array of strings.
2. Call the `forEach` method on the array.
3. Pass in a callback function to `forEach`, with a parameter called `language`.
4. Take the first element in the array, 'HTML' and assign it to `language`.
5. Pop up an alert that says you love HTML.
6. Repeat 4 and 5 for the other elements in `languages`.

We can use loops for more interesting problems, too. For example, we've written code to add two numbers, but what if we wanted to add an arbitrary amount of numbers?

```
var total = 0;

summands = [1, 2, 3, 4, 5];

summands.forEach(function(summand) {
  total += summand;
});

total
```

It's conventional that an array name is plural, and that the parameter to the function passed into `forEach` have a singular name (e.g., `summands` for the array and `summand` for the argument to the function). But for all JavaScript cares, the array could be called `summands`



and the argument to the function could be called `sofa` .

Let's go back to our Mad Libs JavaScript. One thing I didn't like about our code was all the duplication:

### scripts.js

```
$(document).ready(function() {  
  $("#blanks form").submit(function(event) {  
    var person1Input = $("input#person1").val()  
    var person2Input = $("input#person2").val()  
    var animalInput= $("input#animal").val()  
    var exclamationInput = $("input#exclamation").val()  
    var verbInput = $("input#verb").val()  
    var nounInput = $("input#noun").val()  
  
    $(".person1").text(person1Input);  
    $(".person2").text(person2Input);  
    $(".animal").text(animalInput);  
    $(".exclamation").text(exclamationInput);  
    $(".verb").text(verbInput);  
    $(".noun").text(nounInput);  
  
    $("#story").show();  
  
    event.preventDefault();  
  });  
});
```

Let's refactor this with arrays and looping:

### scripts.js

```
$(document).ready(function() {  
  $("#blanks form").submit(function(event) {  
    var blanks = ["person1", "person2", "animal", "exclamation", "verb", "noun"];  
  
    blanks.forEach(function(blank) {  
      var userInput = $("input#" + blank).val();  
      $("." + blank).text(userInput);  
    });  
  
    $("#story").show();  
  
    event.preventDefault();  
  });  
});
```

You might be getting tired of me pointing out the indentation, but I'm going to do it again, because it's something worth taking your time to do right: everything inside the `forEach()` method is indented in two spaces, because it's code that's inside the callback function, and everything in a function gets indented so that it's obvious what's in the function and what's outside it.

I'm going to end this lesson with a short lecture. One of the best definitions of "good code" I've heard comes from Corey Haines, a developer who helps train other developers. The one thing that's certain about software, according to Corey, is that it changes. There will always be new features to add and bugs to fix. Often, the people working on the software will change, and even if they don't, the original developers will need to modify code they wrote months or years ago, code that they sometimes can't even remember writing.

One of things I really like about the new, looping version of the Madlibs JavaScript is that there's no repetition in the code, which makes it easy to change. Before, we did the same thing 6 times: get the value of the input, then put it in the appropriate blank space. If we want to change something - like, I don't know, capitalize all the text before inserting it onto the page (obviously this is contrived, but hopefully you get the idea) - then we have to change all 6 places. In this small example, the only room for error is maybe forgetting to update one line, although that would be embarrassing. But as your programs get more complex, you'll see that, if you have repetitive code, it becomes very difficult to maintain, and you will regularly update one place and fail to update another - or worse yet, update them in different ways that cause incompatibilities down the road.

There's a principle in software development called Don't Repeat Yourself, or DRY. DRY code is easy to change, because you only have to make any change in one place. One way we DRY up our code is by taking repetitive bits of code and extracting them into a function. Another way is by taking something we do several times and by turning it into a loop. Whenever you finish writing some code, you should always look back to see if there is any way you can DRY it up.

Also, remember how I harped on about using descriptive variable names? "Easy to change" doesn't just mean clean code - if you can't figure out what a variable is for or what a function does based on its name, then it's harder to change it down the road, when you might not remember exactly how your code works (or when somebody else is trying to change your code).

## Summary

Simple loop with `forEach()` :

```
languages.forEach(function(language) {  
    alert('I love ' + language + '!');  
});
```

More advanced loop to add numbers:

```
var total = 0;  
summands = [1, 2, 3, 4, 5];  
summands.forEach(function(summand) {  
    total += summand;  
});
```

Loop through and manipulate DOM elements:

```
var classes = ["some-class", "some-other-class", "another-class"];  
classes.forEach(function(class) {  
    $(". " + class).hide();  
});
```

## 09.Looping Practice

In the Looping video, we learned:

- How to use a `forEach()` loop
- Naming conventions for arrays and the parameter in a loop
- The importance of DRY code

Looping can be tricky, so you should practice it until you feel comfortable:

- Make an array of your friends' names, then loop through it and `alert()` a hello to each of them.
- Make a web page that lists out your favorite ice cream flavors. Rather than writing the flavors in the HTML, use a JavaScript loop to insert the flavors into the page from an array.
- Write a loop to multiply an arbitrary amount of numbers. This should look very similar to the code to sum numbers from the lesson.
- Refactor your Mad Libs page to use a loop.
- Go back to your page where it would pop up an alert every time you clicked an element and say what kind of tag it was. Refactor your JavaScript for this page to use a loop.

**Remember:** When choosing a name for the variables in your loop, remember that it is good practice to use a plural for the array and the singular form of that word for the loop.

```
> var languages = ['HTML', 'CSS', 'JavaScript'];
> languages.forEach(function(language) {
  alert('I love ' + language + '!');
});
```

The array is named `languages` and the parameter is the singular `language` .

# 10.Looping With For

Embedded Video: <https://www.youtube.com/watch?v=STC9qbB95Rc?rel=0>

`forEach()` is a nice, easy-to-read, easy-to-understand method, but it's actually a pretty new addition to JavaScript. Here's how loops traditionally have looked:

```
for (var index = 1; index <= 3; index += 1) {  
  alert(index);  
}
```

Here's how the `for` loop works:

1. The `for` statement takes three parameters: initialization, condition, and final expression.
2. The initialization parameter lets you initialize a variable. In this case, we've initialized a variable called `index` that starts at 1.
3. The condition parameter tells the `for` loop when to stop running. In this case, we've said to stop looping when `index` reaches 3.
4. The final expression parameter says to add 1 to the `index` parameter after each time through the loop.
5. On each loop through the array, we simply pop up an alert with the number of the pass we're on.

Here's an example we saw in the last lesson re-written with a `for` loop:

```
var languages = ['HTML', 'CSS', 'Javascript'];  
for (var index = 0; index < languages.length; index += 1) {  
  alert('I love ' + languages[index] + '!');  
}
```

You might have noticed I snuck something new in here: `languages.length` isn't a method - it's missing the parentheses at the end. It's a property, which we'll learn more about later. Strings also have this property, e.g. `'foobar'.length`.

The `forEach()` method is much cleaner than the `for` loop, but sometimes you'll find yourself in a situation where you need to run a loop a certain number of times, rather than looping over an array.

## Summary

Example `for` loop:

```
for (var index = 1; index <= 3; index += 1) {  
  alert(index);  
}
```

Example of using a `for` loop with an array:

```
var languages = ['HTML', 'CSS', 'Javascript'];  
for (var index = 0; index < languages.length; index += 1) {  
  alert('I love ' + languages[index] + '!');  
}
```

# 11.For Loop Practice

In the Looping with for video, we learned:

- The three parameters of a `for` loop
- The `length` property in JavaScript

Here are a couple exercises for you to get some practice with `for` loops:

- Go back through the looping practice problems, and redo them all with `for` loops.
- Make a web page that prints out the lyrics to "99 bottles of beer on the wall".

## 12. Debugging In JavaScript

Embedded Video: [https://www.youtube.com/watch?v=qYky\\_O7yvwI?rel=0](https://www.youtube.com/watch?v=qYky_O7yvwI?rel=0)

As you start writing more and more complex JavaScript, you'll run into increasingly difficult bugs and problems. Here are some approaches to debugging when something goes wrong.

As an example, let's use our Madlibs page, but with some errors I've introduced for us to debug. Here's the code:

**madlibs.html**



```
<!DOCTYPE html>
<html>
  <head>
    <link href="css/bootstrap.css" rel="stylesheet" type="text/css">
    <link href="css/styles.css" rel="stylesheet" type="text/css">
    <script src="js/jquery-1.10.2.js"></script>
    <script src="js/scripts.js"></script>
    <title>A fantastical adventure</title>
  </head>
  <body>
    <div class="container">
      <h1>Fill in the blanks to write your story!</h1>
      <div id="blanks">
        <form>
          <div class="form-group">
            <label for="person1">A name</label>
            <input id="person1" class="form-control" type="text">
          </div>
          <div class="form-group">
            <label for="person2">Another name</label>
            <input id="person2" class="form-control" type="text">
          </div>
          <div class="form-group">
            <label for="animal">An animal</label>
            <input id="animal" class="form-control" type="text">
          </div>
          <div class="form-group">
            <label for="exclamation">An exclamation</label>
            <input id="exclamation" class="form-control" type="text">
          </div>
          <div class="form-group">
            <label for="verb">A past tense verb</label>
            <input id="verb" class="form-control" type="text">
          </div>
          <div class="form-group">
            <label for="noun">A noun</label>
            <input id="noun" class="form-control" type="text">
          </div>

          <button type="submit" class="btn">Show me the story!</button>
        </form>
      </div>

      <div id="story">
        <h1>A fantastical adventure</h1>
        <p>One day, <span class="person1"></span> and <span class="person2"></span> were
      </div>
    </div>
  </body>
</html>
```

## scripts.js

```
$(document).ready(function() {  
  $("#blanks form").submit(function() {  
    var blanks = ["person1", "person2", "animal", "exclamation", "verb", "noun"];  
  
    blanks.forEach(function(blank) {  
      var userInput = $("input." + blank).val();  
      $("." + blank).text(userInput).val();  
    });  
  
    $("#story").sho();  
  });  
});
```

When I load up this page, fill out the form, and click the submit button, nothing happens. Time to debug!

The first step of debugging is to look for error messages. If we open up the JavaScript console, sure enough, there's an error: `GET`

`file:///Users/michael/epicodus/debug/js/scripts.js index.html:7` . This error tells us that the browser tried to get the file `scripts.js` but couldn't find it. And hey, that's our problem - we left the `r` out of `scripts.js` . I can't tell you how many times I've seen students spend a long time debugging, just to find out they mis-spelled a filename. Let's fix that typo and reload the page.

Great, no more errors in the JavaScript console. But when we submit our form, again, we get nothing. Let's use another tactic: pausing on exceptions. If we open our JavaScript console and switch to the tab on the top that says Sources, there's a button on the top right that looks like an octagon with a pause button. If we click it twice, it turns purple. This will cause JavaScript to stop running whenever there's an error. If we submit our form, sure enough, it pauses on and highlights the offending line. I've mis-typed the method name: `$("#story").sho()` ; should be `$("#story").show()` ;.

Let's fix this and move forward. Now, we don't have any errors, but we still aren't seeing the story. Let's try another tactic: checking to see what code is executed. I do that by adding an `alert()` to a couple points in my code:

## scripts.js

```
$(document).ready(function() {  
  $("#blanks form").submit(function() {  
    alert('Got to beginning of form submit!');  
    var blanks = ["person1", "person2", "animal", "exclamation", "verb", "noun"];  
  
    blanks.forEach(function(blank) {  
      var userInput = $("input." + blank).val();  
      $("." + blank).text(userInput).val();  
    });  
  
    $("#story").show();  
    alert('Got to end of form submit!');  
  });  
});
```

Now, when I submit my form, I can see if all my code is run, or if the form breaks at some point in the middle. In this case, both dialog boxes open up. So there's nothing actually breaking or not getting run in my code. That should be a clue to step back and review the documentation for the tools I'm using to make sure I haven't left anything out. If I look back at the lesson on forms with jQuery, I can see that I forgot to include `event.preventDefault()`. Let's add that:

### scripts.js

```
$(document).ready(function() {  
  $("#blanks form").submit(function(event) {  
    var blanks = ["person1", "person2", "animal", "exclamation", "verb", "noun"];  
  
    blanks.forEach(function(blank) {  
      var userInput = $("input." + blank).val();  
      $("." + blank).text(userInput).val();  
    });  
  
    $("#story").show();  
  
    event.preventDefault();  
  });  
});
```

And now our paragraph with the story finally shows!

But, there's still a problem: none of what we type actually gets put into the story. Something is going wrong, but it's hard to tell what by just looking at our code. Wouldn't it be nice if we could drop in and run just one line of our code at a time, to see what's going wrong? Chrome has a handy tool called the `debugger` keyword to do just that. Here's how to use it:

### scripts.js

```
$(document).ready(function() {  
  $("#blanks form").submit(function(event) {  
    var blanks = ["person1", "person2", "animal", "exclamation", "verb", "noun"];  
  
    blanks.forEach(function(blank) {  
      debugger;  
      var userInput = $("input." + blank).val();  
      $("." + blank).text(userInput).val();  
    });  
  
    $("#story").show();  
  
    event.preventDefault();  
  });  
});
```

Now, whenever Chrome JavaScript engine hits the `debugger` keyword, it will pause execution and let us run whatever code we please. This only happens when the JavaScript console window is open, so let's make sure that it is. Now, when we submit the form, we get the same kind of screen we got when we paused on the exception. Down in the bottom left, there's a little arrow and some lines. Clicking that will open up a JavaScript console below our code. Here, we can run JavaScript as if we were at the exact point where the debugger keyword is.

Let's copy and paste the next line of code: `var userInput = $("input." + blank).val();`. Hm, it returns `undefined`. I would have expected it to return the first person's name. Let's type `blank` so that we can see what the value of that variable is. Sure enough, it's `person1`. Let's just run `$("input." + blank);` to make sure that we're selecting the correct element. Oh, this is the problem - we're getting back an empty array. That's not what we wanted. If we look back at our HTML, we can see that our inputs have IDs, not classes, so we should be using `#` s, not `.` s in our jQuery selector. Let's try changing that in the debugger console: sure enough, `$("input#" + blank);` returns the proper element. If we go back to our code, remove the `debugger` and update our selector, our page now works.

One last helpful JavaScript debugging tool I'll tell you about is `console.log()`. Let's go back to before we used `debugger`, and try `console.log()` instead:

**scripts.js**

```
$(document).ready(function() {  
  $("#blanks form").submit(function(event) {  
    var blanks = ["person1", "person2", "animal", "exclamation", "verb", "noun"];  
  
    blanks.forEach(function(blank) {  
      var userInput = $("input." + blank).val();  
      console.log(userInput);  
      $("." + blank).text(userInput).val();  
    });  
  
    $("#story").show();  
  
    event.preventDefault();  
  });  
});
```

When we submit our form, the JavaScript console now says `6 undefined scripts.js:7` . In other words, `undefined` was logged 6 times from line 7 of `scripts.js` . If we fix our code but leave the `console.log()` in, the proper values get logged to the console.

Generally, I use `console.log()` when there's a little piece of information I need to know that will help me debug, and `debugger` when I need to explore my code to know what's going wrong.

Just as a reminder, a useful debugging technique we explored before is changing the background color of an element to make sure you correctly selected what you were trying to, or checking in the Chrome developer tools under Event Listeners.

And those are the most important tricks of the trade for debugging.

## Summary

- Check for errors in the console.
- Read all errors carefully.
- Break on errors.
- Stick in an `alert()` or two and see if you get there.
- `console.log()` relevant information.
- Add `debugger` and explore your live code.
- Read the documentation.
- Add green background to make sure right thing is selected.
- Check Event Listeners in the Chrome DevTools

# 13.The Ping Pong Test Code Review

## The Ping-Pong test

The code review for this week is a programming test that makes sure you understand how to use loops, conditionals, and variables. The challenge is to simply make a web page where the user is prompted to enter a number, and then the page displays every number up to that number. But, for multiples of three, the page prints "ping" instead of the number, and for multiples of five, the page prints "pong". For numbers that are multiples of three and five, the page prints "ping-pong". Here's an example:

Take the test and make the page! Here is a hint: To tell if one number is divisible by another number, use the `%` operator. For example, if a variable called `number` is divisible by 7, then `number % 7` returns `0`.

**Due:** Friday, October 16th at midnight

Below are the objectives the instructor will use to review your code.

## This week's code review objectives

- All previous standards (last code review's objectives) are in place
- Logic is easy to understand
- JavaScript function(s) process user input
- jQuery updates the DOM after user input

## Last review's objectives

- Variable names are descriptive of what they represent
- Web page is styled using Bootstrap and CSS
- Code has proper indentation and spacing
- Commits are made regularly with clear messages that finish the phrase "It will..."
- Project README that includes:
  - author name
  - program name
  - description
  - **setup instructions with link to project on github.io**
  - copyright and license information

# JavaScript JSON

## What is JSON?

- JSON stands for JavaScript Object Notation
- JSON is lightweight data interchange format
- JSON is language independent \*
- JSON is "self-describing" and easy to understand

## JSON Example

This JSON syntax defines an employees object: an array of 3 employee records (objects):

### JSON Example

```
{
  "employees": [
    {"firstName": "John", "lastName": "Doe"},
    {"firstName": "Anna", "lastName": "Smith"},
    {"firstName": "Peter", "lastName": "Jones"}
  ]
}
```

## The JSON Format Evaluates to JavaScript Objects

The JSON format is syntactically identical to the code for creating JavaScript objects.

Because of this similarity, a JavaScript program can easily convert JSON data into native JavaScript objects.

## JSON Syntax Rules

- Data is in name/value pairs
- Data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays

## JSON Data - A Name and a Value

JSON data is written as name/value pairs, just like JavaScript object properties.

A name/value pair consists of a field name (in double quotes), followed by a colon, followed by a value:

```
"firstName":"John"
```

## JSON Objects

JSON objects are written inside curly braces.

Just like in JavaScript, objects can contain multiple name/value pairs:

```
{"firstName":"John", "lastName":"Doe"}
```

## JSON Arrays

JSON arrays are written inside square brackets.

Just like in JavaScript, an array can contain objects:

```
"employees":[
  {"firstName":"John", "lastName":"Doe"},
  {"firstName":"Anna", "lastName":"Smith"},
  {"firstName":"Peter", "lastName":"Jones"}
]
```

In the example above, the object "employees" is an array. It contains three objects.

Each object is a record of a person (with a first name and a last name).

## Converting a JSON Text to a JavaScript Object

A common use of JSON is to read data from a web server, and display the data in a web page.

For simplicity, this can be demonstrated using a string as input (or read more in our JSON tutorial):



First, create a JavaScript string containing JSON syntax:

```
var text = '{ "employees" : [' +
  '{ "firstName":"John" , "lastName":"Doe" },' +
  '{ "firstName":"Anna" , "lastName":"Smith" },' +
  '{ "firstName":"Peter" , "lastName":"Jones" } ]}';
```

Then, use the JavaScript built-in function `JSON.parse()` to convert the string into a JavaScript object:

```
var obj = JSON.parse(text);
```

Finally, use the new JavaScript object in your page:

## Example

```
<!DOCTYPE html>
<html>
<body>

<h2>Create Object from JSON String</h2>

<p id="demo"></p>

<script>
var text = '{"employees":["{
  "firstName":"John", "lastName":"Doe" },
  "firstName":"Anna", "lastName":"Smith" },
  "firstName":"Peter", "lastName":"Jones" ]}';

obj = JSON.parse(text);
document.getElementById("demo").innerHTML =
obj.employees[1].firstName + " " + obj.employees[1].lastName;
</script>

</body>
</html>
```

# jQuery

<http://ejohn.org/apps/workshop/intro/#1>

# 01.jQuery Objectives

In this section, we will learn how to use jQuery, an extensive JavaScript library to help make our webpages interactive. By using jQuery, our webpages can change when a user clicks on a button or hovers over a section of text and that is just a small part of the functionality of jQuery!

## Overview of the section

- Learn key jQuery concepts like **callbacks** and **event handlers**
- Show, hide and toggle HTML elements
- Understand the **Document Object Model (DOM)** and how JavaScript changes the displayed page, not the HTML
- Traverse and manipulate the DOM
- Create forms to gather input from the user
- Manipulate the attributes of HTML elements

## 02.Introducing jQuery

Embedded Video: <https://www.youtube.com/watch?v=5dGgLPDPPOk?rel=0>

Now that we know a bit of JavaScript, let's learn jQuery, a JavaScript library that makes it easy to make web pages interactive. Let's start by learning how to pop up dialogue boxes when you click certain parts of the page.

First, we need to download jQuery. Choose the uncompressed, development version from the jQuery 1.x section (jQuery 2.x is faster and smaller, but it doesn't work on Internet Explorer 6, 7, or 8); if the file just opens in your browser, save it into your js folder. Let's also make a file called `scripts.js` to store our own JavaScript code, and then begin developing our page with this HTML:

### html-help.html

```
<!DOCTYPE html>
<html>
  <head>
    <link href="css/bootstrap.css" rel="stylesheet" type="text/css">
    <title>HTML help</title>
  </head>
  <body>
    <div class="container">
      <h1>HTML help</h1>

      <p>This is a very special page. If you click somewhere, it will tell you what type

      <p>Look at this cute walrus!</p>
      
    </div>

    <script src="js/jquery-1.9.1.js"></script>
    <script src="js/scripts.js"></script>
  </body>
</html>
```

It's important that our own `scripts.js` file goes after jQuery, as it will rely on jQuery functionality that must be loaded first.

Okay, we're finally ready to use jQuery! Here's the code to type in `scripts.js` :

### scripts.js

```
jQuery("h1").click(function() {  
    alert("This is a header.");  
});  
  
jQuery("p").click(function() {  
    alert("This is a paragraph.");  
});  
  
jQuery("img").click(function() {  
    alert("This is an image.");  
});
```

Before we actually talk about the JavaScript, I just want to point out the `function()` s we're using here are just like the ones we practiced before. We haven't assigned them to variables, though, and we've split them onto multiple lines to be easier to read. Also, please notice the indentation! The `alert()` s are two spaces in, because they're inside the `function()` S.

If we load up the page and click on different parts, we'll get dialog boxes popping up and telling us what they are. Hooray! Now, let's look more closely at how this works.

Here we have a new function: the `jQuery()` function! One thing that the jQuery function does is to select HTML elements on the page, based on the argument you pass in to it. So the code `jQuery("p")` selects all of the paragraphs on the page.

After we select the elements, we use the `.click()` method to attach an **event handler** to them. The event handler "listens" to the elements and responds when they're clicked.

Next, we need to tell jQuery what to do when the user clicks that element. We pass in a function as an argument, and that function contains the `alert()` function. The reason why jQuery requires that we pass in a function that contains `alert()` , rather than just passing in `alert()` itself, is so that if we want, we can pass in more than one function. We could actually pop up two alerts, like this:

```
jQuery("h1").click(function() {  
    alert("This is a header.");  
    alert("I told you, THIS IS A HEADER!");  
});
```

When you pass in a function as an argument to another function, the function being passed in is called a **callback**.

Phew! That's a lot of work just to make a few silly pop-ups. Fortunately, even though this might feel a bit overwhelming at first, most of the jQuery we'll write will look an awful lot like this, so you'll get the hang of it pretty quickly.

Let's do a couple things to clean up our code. First, it feels a bit wrong to put our `<script>` tags in the document body, where the content of the page lives. Let's move them up into the `<head>` , which is where we put information about the page that isn't displayed:

### html-help.html

```
<head>
  <link href="css/bootstrap.css" rel="stylesheet" type="text/css">
  <script src="js/jquery-1.9.1.js"></script>
  <script src="js/scripts.js"></script>
  <title>HTML help</title>
</head>
```

Oh dear. Now, if we reload the page in our browser, nothing happens when we click anything. Let's review our JavaScript to see why this broke it. Remember that I said that `jQuery('p')` will look for all of the `<p>` tags on the page? The web browser loads your page from the top of your HTML document to the bottom. So when we put our JavaScript in the `<head>` , rather than at the end of the document, there are no `<p>` tags yet, and so there's nothing for `.click()` to attach to. What we need to do is tell our JavaScript not to run until the document has finished loading. Fortunately, jQuery provides an easy way to do this:

### scripts.js

```
jQuery(document).ready(function() {
  jQuery("h1").click(function() {
    alert("This is a header.");
  });

  jQuery("p").click(function() {
    alert("This is a paragraph.");
  });

  jQuery("img").click(function() {
    alert("This is an image.");
  });
});
```

Here, we tell jQuery to look at the HTML document, and then we call the `.ready()` method on it. When the document has finished loading, jQuery will run the callback function that is passed into `.ready()` . And that function will run our code. Generally, it's a good idea to wrap your JavaScript in a function passed to `.ready()` , so that it's executed after the page loads and you don't run into the problem we just had.

Check out our page - it works again.

There's a handy shortcut that most JavaScript developers use: instead of writing `jQuery()` all of the time, we can simply use `$()` :

### scripts.js

```
$(document).ready(function() {  
  $("h1").click(function() {  
    alert("This is a header.");  
  });  
  
  $("p").click(function() {  
    alert("This is a paragraph.");  
  });  
  
  $("img").click(function() {  
    alert("This is an image.");  
  });  
});
```

This isn't really necessary, but it saves us a little bit of repetitive typing, and is how most people write the jQuery function.

Here's one final trick I want to show you. If something on your page isn't working right, you can check in Chrome to make sure that you've attached the event listener correctly. Right-click the element you want to check, go to Inspect Element, and then, in the upper right corner of the box on the bottom, click Event Listeners. If you've attached the listener correctly, you should see it listed there.

## Summary

Link it before your scripts that use it:

```
<head>  
  <script src="js/jquery-1.9.1.js"></script>  
  <script src="js/scripts.js"></script>  
</head>
```

A **callback** is a function passed as an argument to another function.

Select a tag and bind an event handler to it:

```
$("h1").click(function() {  
  alert("This is a header.");  
});
```

The part `function() { alert("This is a header."); }` is considered the callback because it is being passed as an argument into the method `.click()` .

Wrap all your jQuery code in callback passed to `$(document).ready()` so that it is run after the page loads:

```
$(document).ready(function() {  
    $("h1").click(function() {  
        alert("This is a header.");  
    });  
});
```

To check if an event handler is attached properly, right-click the element you want to check, Inspect Element, and click Event Listeners.[aa](#)



## 03.jQuery Practice

In the jQuery video, we learned:

- jQuery is a library of JavaScript that makes web pages interactive
- Link to the scripts.js file after the `jquery.js` file in the tag
- The `jQuery()` function selects HTML elements
- An event handler "listens" to the elements and responds when they are manipulated
- A function passed into another function as an argument is called a callback
- Wrap your JavaScript in a function passed to `.ready()` as a callback

Now it's your turn to practice some basics of jQuery.

- Follow along with the video from the last lesson and re-create the page demonstrated.
- Add an `<h2>` and bind an event handler to it with `.click()`. Remember an event is an action by the user that can be detected by your webpage, so in this instance it would be a mouse click on a particular HTML element.
- Pass in a callback to the `.click()` function that contains an `alert()`. Remember a callback is a function that is passed as an argument to another function.
- Do the same for a `<ul>`.
- Swap out the `.click()` method for `.dblclick()` and `.hover()` in a couple places.

## 04.Simple Effects

Embedded Video: <https://www.youtube.com/watch?v=1O4VzVJc2Do?rel=0>

Popping up dialogue boxes isn't very interesting, so let's move on to actually manipulating our web page by showing and hiding elements. Here's the HTML for the page we'll work on:

### peek-a-boo.html

```
<!DOCTYPE html>
<html>
  <head>
    <link href="css/bootstrap.css" rel="stylesheet" type="text/css">
    <link href="css/styles.css" rel="stylesheet" type="text/css">
    <script src="js/jquery-1.9.1.js"></script>
    <script src="js/scripts.js"></script>
    <title>Peek-a-boo</title>
  </head>
  <body>
    <div class="container">
      <h1>Peek-a-boo</h1>

      <p>Let's play peek-a-boo. Click here to see the surprise!</p>

      
    </div>
  </body>
</html>
```

Obviously, this doesn't work yet. The first thing we should do is to hide the walrus with a bit of CSS. As usual, we'll make a file called `styles.css` in our `css` folder. Then, we'll add a CSS rule to hide the image:

### styles.css

```
img {
  display: none;
}
```

Now, we can use jQuery to show the walrus when you click the paragraph above it:

### scripts.js

```
$(document).ready(function() {  
  $("p").click(function() {  
    $("img").show();  
  });  
});
```

Very nice! Here, we've used another jQuery function called `.show()` that will, obviously, show a hidden element.

But wouldn't it be nicer if the text of the paragraph changed, so after you clicked it, it said "What a cute walrus! Click here to hide her again." Let's do that now. First, we need to make a couple changes to our HTML:

### **peek-a-boo.html**

```
<body>  
  <div class="container">  
    <h1>Peek-a-boo</h1>  
  
    <div class="walrus-hidden">  
      <p>Let's play peek-a-boo. Click here to see the surprise!</p>  
    </div>  
  
    <div class="walrus-showing">  
      <p>What a cute walrus! Click here to hide her again.</p>  
        
    </div>  
  </div>  
</body>
```

Now we've wrapped our page in two `<div>` s: one for when the walrus is hidden, and one for when she's showing. Let's update our CSS to hide the walrus at first:

### **styles.css**

```
.walrus-showing {  
  display: none;  
}
```

And now, we'll update our JavaScript:

### **scripts.js**

```
$(document).ready(function() {  
  $("p").click(function() {  
    $(".walrus-showing").show();  
    $(".walrus-hidden").hide();  
  });  
});
```

One of the cool things about jQuery is that it uses the exact same selectors as CSS.

Now, our code successfully changes the text that is shown above the walrus! Of course, it doesn't switch back and hide her again, so let's update our code once more to take care of that:

### **scripts.js**

```
$(document).ready(function() {  
  $("p").click(function() {  
    $(".walrus-showing").toggle();  
    $(".walrus-hidden").toggle();  
  });  
});
```

Nice. Now everything works as planned.

There's something I don't like about this page, though. Most users expect that when an element is clickable, it's a different color, and that when they hover their mouse over it, the cursor will change to a hand and the text will be underlined. Fortunately, this is easy to implement with CSS:

### **styles.css**

```
p {  
  cursor: pointer;  
  color: #0088cc;  
}  
  
p:hover {  
  text-decoration: underline;  
}
```

Now, this works, and it looks much better. But I'm still not satisfied. Here's why. Let's add another paragraph to this page:

### **peek-a-boo.html**

```
<body>
  <div class="container">
    <h1>Peek-a-boo</h1>
    <p><strong>Peek-a-boo</strong> is an ancient game riddled in mystery and deceit. Thou

    <div class="walrus-hidden">
      <p>Let's play peek-a-boo. Click here to see the surprise!</p>
    </div>

    <div class="walrus-showing">
      <p>What a cute walrus! Click here to hide her again.</p>
      
    </div>
  </div>
</body>
```

Gross! Our styling has run amok and is applied to the intro paragraph, which really shouldn't be clickable. And even worse: if you click it, it toggles the walrus! We really need to limit the scope of our CSS and JavaScript. Let's add some `<span>` tags to clean this up:

### peek-a-boo.html

```
<body>
  <div class="container">
    <h1>Peek-a-boo</h1>
    <p><strong>Peek-a-boo</strong> is an ancient game riddled in mystery and deceit. Th

    <div class="walrus-hidden">
      <p>Let's play peek-a-boo. <span class="clickable">Click here</span> to see the surp
    </div>

    <div class="walrus-showing">
      <p>What a cute walrus! <span class="clickable">Click here</span> to hide her again.
      
    </div>
  </div>
</body>
```

Now, we can update our CSS:

### styles.css

```
.walrus-showing {
  display: none;
}

.clickable {
  cursor: pointer;
  color: #0088cc;
}

.clickable:hover {
  text-decoration: underline;
}
```

And JavaScript:

### scripts.js

```
$(document).ready(function() {
  $(".clickable").click(function() {
    $(".walrus-showing").toggle();
    $(".walrus-hidden").toggle();
  });
});
```

Ah, that's much better now. When you're writing JavaScript (and CSS), it's a good idea to use classes to be specific about the elements you want to select. But keep in mind, if you have more than one thing on the page that can change, you might want to use one class for your CSS and another class for each of the actions. For example, if your page had one place that let you show and hide a walrus, and another place that let you show and hide an emu, you might have both clickable `<span>` s styled with the `clickable` class, but one `<span>` with the `toggle-walrus` , and the other with a `toggle-emu` class.

Speaking of being specific, it feels wrong to me to use the classes `walrus-hidden` and `walrus-showing` . We're supposed to use classes to identify similar parts of our pages that should look and act the same. For our `clickable` `span`, using a class makes perfect sense: we have two places on the page where we use it, and I can imagine that our website could potentially use it in many, many more places. But `walrus-hidden` and `walrus-showing` really couldn't be re-used anywhere else on this page - these sections should be unique. Often, this is the case with JavaScript: we want to identify a unique part of the page that should be changed when a unique event happens. For this case, we actually have another tool we can use: an `ID` . Check it out:

### peek-a-boo.html

```

<body>
  <div class="container">
    <h1>Peek-a-boo</h1>
    <p><strong>Peek-a-boo</strong> is an ancient game riddled in mystery and deceit. Th

    <div id="walrus-hidden">
      <p>Let's play peek-a-boo. <span class="clickable">Click here</span> to see the surp
    </div>

    <div id="walrus-showing">
      <p>What a cute walrus! <span class="clickable">Click here</span> to hide her again.
      
    </div>
  </div>
</body>

```

Now, we change our CSS like this:

### styles.css

```

#walrus-showing {
  display: none;
}

```

And our JavaScript like this:

### scripts.js

```

$(document).ready(function() {
  $(".clickable").click(function() {
    $("#walrus-showing").toggle();
    $("#walrus-hidden").toggle();
  });
});

```

So, what's the difference between a class and an ID, except that we use a `.` to select a class and a `#` to select an ID? Nothing, really, except that you can only use an ID once on a page, and you can use a class as many times as you want. But by using an ID, you can be clear that you're referring to one particular thing, rather than potentially referring to several.

We could actually go one step further in thinking about how to make our code re-usable. Instead of calling the IDs `walrus-showing` and `walrus-hidden`, we could call them `initially-hidden` and `initially-showing`. That way, we could re-use this code on other that have the same kind of toggling functionality.

Now, a quick lecture before you move on to practicing all that you've just learned. When we use JavaScript to manipulate the page, we haven't changed the source code. No matter how much we change the page with JavaScript, if you hit the Refresh button in your browser, it will go back to the initial state. What JavaScript is doing is manipulating the **Document Object Model**, or **DOM**. The DOM is your browser's interpretation of the HTML it reads. If you inspect an element of the page, you're actually seeing the DOM, not the HTML. Try inspecting the page we've been working on, and then click the click here part that changes the page. You can see in the inspector that the DOM changes.

## Summary

Show, hide, and toggle elements:

```
$(".my-class").show();
$(".my-class").hide();
$(".my-class").toggle();
```

Make a class look like a link:

```
.clickable {
  cursor: pointer;
  color: #0088cc;
}

.clickable:hover {
  text-decoration: underline;
}
```

Use classes (e.g. `.my-class` ) for things that are on the page multiple times. Use IDs (e.g. `#my-id` ) for something that is unique on the page.

The `Document Object Model` , or `DOM` , is your browser's interpretation of the HTML it reads. When JavaScript changes the page, it updates the DOM, not the HTML.



## 05.jQuery Effects Practice

In the Simple effects video, we learned:

- How to show and hide elements using `.show()` and `.hide()`
- How to toggle with `.toggle()`
- How to style a link with CSS
- How to use `<span>` tags to limit the scope of CSS and JavaScript
- Classes are used for elements that show up in multiple places on our web page
- IDs are used for an element that shows up only once on our web page
- When we use JavaScript to manipulate a web page, we are really manipulating the DOM, not the HTML.

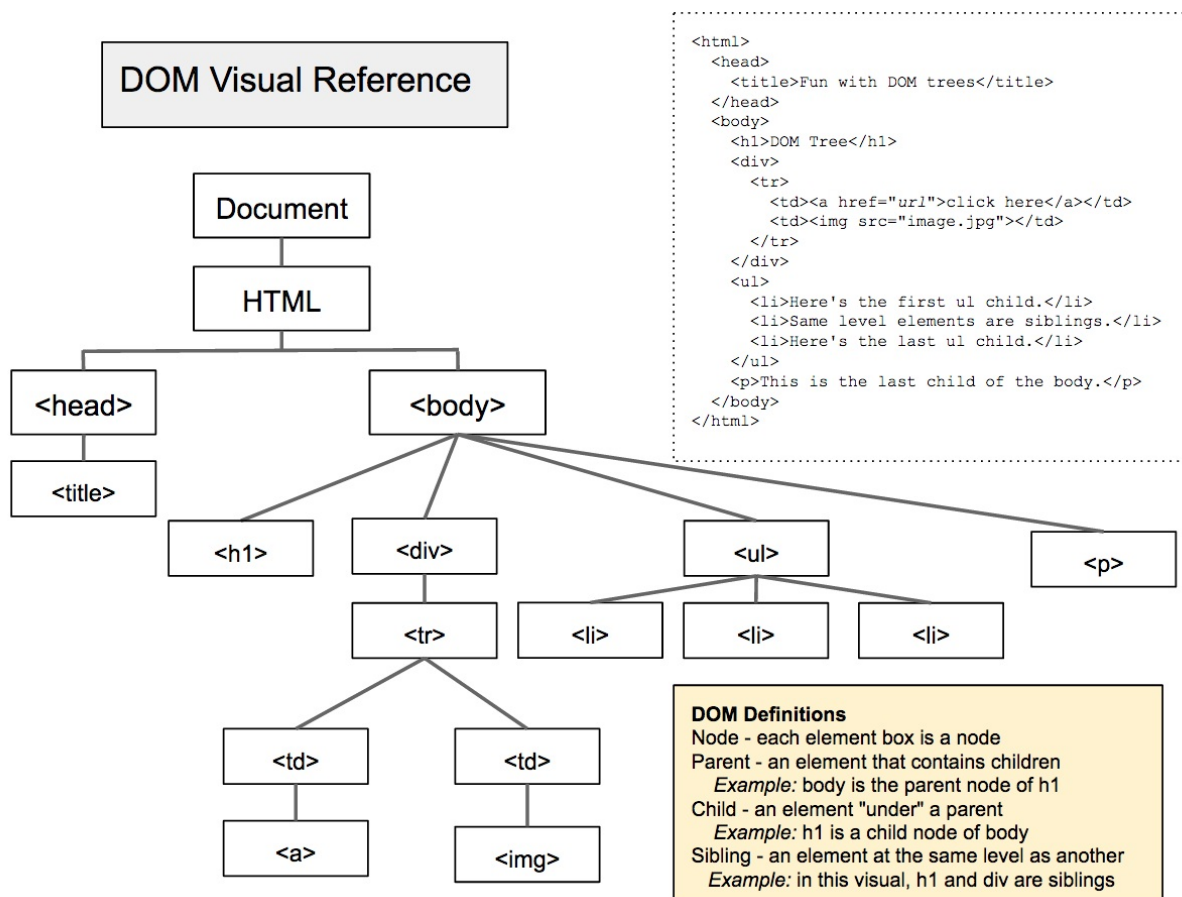
Let's get some practice using jQuery effects.

- On the walrus page, add some clickable text that alternates between saying "Hide/show images", and toggles the images appropriately.
- On either the walrus page or the webpage from the previous exercise, try fading and sliding elements with `.fadeIn()` , `.fadeOut()` , `.fadeToggle()` , `.slideDown()` , `.slideUp()` , and `.slideToggle()` .
- Make a webpage with a few different sections, where a few of the sections are hidden by default, with a bit of teaser text that expands when you click on it (e.g., "Click to learn about the giant sandcastles of this fabulous country!"). Be creative!

## 06.DOM Manipulation and Traversal

Embedded Video: <https://www.youtube.com/watch?v=q0EZq-T1Lac?rel=0>

In this lesson, we are going to explore manipulating DOM elements by inserting new text with the jQuery `prepend()` method. Then we'll look at how to traverse the DOM in search of a specific element to remove it. Let's take a look at a visual representation of the DOM that you can use as reference for manipulation and traversal. Each element in the DOM, represented in each box below, can be a parent, child or sibling to other elements. Understanding the position of elements in the DOM will help you insert, locate and remove elements, as needed.



In the last lesson, we explored how to simply show and hide elements of the DOM. Now, let's get a little more advanced and insert content into the DOM. Here's some HTML we'll start with:

**talk.html**

```
<!DOCTYPE html>
<html>
  <head>
    <link href="css/bootstrap.css" rel="stylesheet" type="text/css">
    <link href="css/styles.css" rel="stylesheet" type="text/css">
    <script src="js/jquery-1.10.2.js"></script>
    <script src="js/scripts.js"></script>
    <title>Talk to the web page</title>
  </head>
  <body>
    <div class="container">
      <h1>Talk to the web page</h1>
      <p>Click a button to say something to the web page. See what it says back!</p>

      <button class="btn btn-primary" id="hello">Say "hello"</button>
      <button class="btn btn-inverse" id="goodbye">Say "goodbye"</button>
      <button class="btn btn-danger" id="stop">Say "stop copying me!"</button>

      <div class="row">
        <div class="col-md-6">
          <h2>You said:</h2>
          <ul class="unstyled">

            </ul>
        </div>

        <div class="col-md-6">
          <h2>The web page said back:</h2>
          <ul class="unstyled">

            </ul>
        </div>
      </div>
    </div>
  </body>
</html>
```

Now, here's the JavaScript to make the buttons work:

### **scripts.js**

```
$(document).ready(function() {  
  $("button#hello").click(function() {  
    $("ul").prepend("<li>Hello!</li>");  
  });  
  
  $("button#goodbye").click(function() {  
    $("ul").prepend("<li>Goodbye!</li>");  
  });  
  
  $("button#stop").click(function() {  
    $("ul").prepend("<li>Stop copying me!</li>");  
  });  
});
```

The `.prepend()` method will insert the text of the argument it's given into to the top of `<ul>` as the first child of the `<ul>`. (As you might guess, there's also a `.append()` method that would insert at the bottom as the last child of the `<ul>`; there are also `.before()` and `.after()` methods that would add the argument before or after the `<ul>` tag as siblings, rather than within it as children.)

This is nice, but not a very fun conversation. Let's have the webpage say something different back to the user:

### scripts.js

```
$(document).ready(function() {  
  $("button#hello").click(function() {  
    $("ul#user").prepend("<li>Hello!</li>");  
    $("ul#webpage").prepend("<li>Why hello there!</li>");  
  });  
  
  $("button#goodbye").click(function() {  
    $("ul#user").prepend("<li>Goodbye!</li>");  
    $("ul#webpage").prepend("<li>Goodbye, dear user!</li>");  
  });  
  
  $("button#stop").click(function() {  
    $("ul#user").prepend("<li>Stop copying me!</li>");  
    $("ul#webpage").prepend("<li>Pardon me. I meant no offense.</li>");  
  });  
});
```

Of course, we need to change the `<ul>` tags to have IDs for `user` and `webpage`. For the sake of brevity, I won't bother showing the updated HTML here. Go ahead and do that yourself.

## THE CODE BELOW IS TRICKY. WE JUST WANT YOU TO KNOW THE STUFF ABOVE THIS POINT, BUT WATCH THE WHOLE VIDEO.

Now that we know how to add to the DOM, let's learn how to remove from it. Let's make it so that when a user clicks a message, it disappears.

First, we need to select each message. When I'm working on some tricky DOM manipulation, I usually start by selecting the element and changing its background color to green, just so that I know I have my selector working properly. You might think this bit of jQuery would work:

### scripts.js

```
$(document).ready(function() {  
  // previous code...  
  $('li').css('background-color', 'green');  
});
```

But it doesn't. This code is run right after the web page finishes loading

( `$(document).ready()` ). And after the page finishes loading, there are no elements with class `delete` . The elements are only added after we click a button. So we need to move our new code to within the callback passed to `click()` . Let's start by just adding it to one of the `click()` methods:

```
$("#button#hello").click(function() {  
  $("#ul#user").prepend("<li>Hello! <span class='clickable delete'>x</span></li>");  
  $("#ul#webpage").prepend("<li>Why hello there! <span class='clickable delete'>x</span></li>");  
  $('li').css('background-color', 'green');  
});
```

Now, if we click Say "hello", the messages are green.

Let's move to the next step - attaching an event handler in place of changing the background color. Again, in the spirit of taking one step at a time, we're not going to actually try to get the click to delete the element - instead, we're going to simply open a dialog box:

```
$("#button#hello").click(function() {  
  $("#ul#user").prepend("<li>Hello! <span class='clickable delete'>x</span></li>");  
  $("#ul#webpage").prepend("<li>Why hello there! <span class='clickable delete'>x</span></li>");  
  $('li').click(function() {  
    alert('hi');  
  });  
});
```

Now, if we click Say "hello" and then the message, we get our alert! But there's a problem: if we click the button twice, clicking the last message will open the alert twice. And if we click the button again, the last message will open the alert three times, and the middle message will open it twice. What's happening is that the first time we click the button, `$('.delete')` only finds a single message in each list, and attaches an event handler. But the next time we click the button, that first message is already on the page, and a second event handler is attached to it. And each time we click the button again, another event handler is attached to every message on the page.

What we want is to only attach handlers to the message we most recently inserted. Since we're inserting them at the top of each list, we can select one of the `<ul>` s, look through its child elements (the `<li>` s), and select the first one of them:

```
$("#ul#user").children("li").first().click(function() {
    alert('hi');
});
$("#ul#webpage").children("li").first().click(function() {
    alert('hi');
});
```

Now, each of the messages only opens a single dialog box when clicked.

Finally, we should replace our callback with the actual code to delete the message:

```
$("#ul#user").children("li").first().click(function() {
    $(this).remove();
});
$("#ul#webpage").children("li").first().click(function() {
    $(this).remove();
});
```

`remove()` is pretty straightforward, but what is `this` ? `this` is a bit of a tricky concept in JavaScript, and I'm not going to give it a full explanation here. For now, you can think of it as referring to whatever was clicked on.

We're done with this rather long lesson. Congrats on making it through!

## Summary

Insert within a DOM element at the beginning or end:

```
$('#ul').prepend('<li>First list item</li>');
$('#p').append('New last sentence of the paragraph.');
```

Insert before or after a DOM element:

```
$('#p').before('<h3>Title for paragraph</h3>');  
$('#h1').after('<h2>New subheading</h2>');
```

Remove a DOM element:

```
$('#id-to-remove').remove();
```

Select the children of a DOM element:

```
$('.some-class').children(); //select all children  
$('ul').children('li'); // selects just the <li>s
```

Select only the first or last child element for a node:

```
$('ul').children('li').first();  
$('ul').children('li').last();
```

Select the element that was clicked on with this:

```
$('.some-class').click(function() {  
    $(this);  
});
```

Test that you've selected the correct DOM element:

```
$('.element-to-select').css('background-color', 'green');
```

Test that you've properly attached an event handler:

```
$('.element-to-select').click(function() {  
    alert('hi');  
});
```

## 07.DOM Manipulation and Traversal Practice

In the DOM manipulation video, we learned:

- The visual representation of the DOM's parent, child and sibling relationships
- How a child element will be inserted at the top of a `<ul>` tag with `.prepend()` and at the end with `.append()`
- How to correctly select DOM elements and delete them

Okay, your turn. Practice adding some interactivity to your web pages:

- Follow along with the video and build a page that "talks" to the user.
- Practice selecting certain HTML elements and changing the background color to green.
- Now practice removing those elements, just like in the video.
- Make a "cat vs dog" page - if you click a button for the cat to meow, the dog should bark back, and vice versa. Use Bootstrap to style your pages!
- In addition to `.prepend()` and `.append()`, you can add content before or after the selected tags (rather than within them) with `.before()` and `.after()`. Make a page where if a user clicks on an element some sort of image gets inserted into the page before or after that element. Allow the new image to be removed by a click as well.



## 08.Forms

Embedded Video: <https://www.youtube.com/watch?v=eGbTDP6phh4?rel=0>

So far, the only way we've been able to capture user input is by using `confirm()` and `alert()`. Let's learn about forms so that we can build more interesting pages.

Have you ever played Mad Libs? You're prompted to fill out a list of nouns, verbs, adjectives, etc., and then copy them onto another piece of paper that contains a story, missing those crucial words that you are now providing. The idea is to pick bizarre words without knowing what the story is, and then when you fill them in, the results can be hilarious.

Let's make a page that mimics the Mad Libs format:

**mad-lib.html**

```
<!DOCTYPE html>
<html>
  <head>
    <link href="css/bootstrap.css" rel="stylesheet" type="text/css">
    <link href="css/styles.css" rel="stylesheet" type="text/css">
    <script src="js/jquery-1.10.2.js"></script>
    <script src="js/scripts.js"></script>
    <title>A fantastical adventure</title>
  </head>
  <body>
    <div class="container">
      <h1>Fill in the blanks to write your story!</h1>
      <div id="blanks">
        <form>
          <div class="form-group">
            <label for="person1">A name</label>
            <input id="person1" class="form-control" type="text">
          </div>
          <div class="form-group">
            <label for="person2">Another name</label>
            <input id="person2" class="form-control" type="text">
          </div>
          <div class="form-group">
            <label for="animal">An animal</label>
            <input id="animal" class="form-control" type="text">
          </div>
          <div class="form-group">
            <label for="exclamation">An exclamation</label>
            <input id="exclamation" class="form-control" type="text">
          </div>
          <div class="form-group">
            <label for="verb">A past tense verb</label>
            <input id="verb" class="form-control" type="text">
          </div>
          <div class="form-group">
            <label for="noun">A noun</label>
            <input id="noun" class="form-control" type="text">
          </div>

          <button type="submit" class="btn">Show me the story!</button>
        </form>
      </div>

      <div id="story">
        <h1>A fantastical adventure</h1>
        <p>One day, <span class="person1"></span> and <span class="person2"></span> were
      </div>
    </div>
  </body>
</html>
```

Here, I've used some Bootstrap classes to make the form look nice; you can read more about them in the [Bootstrap form documentation](#).

Let's make the story hidden to start:

### **styles.css**

```
#story {  
  display: none;  
}
```

Now, our JavaScript needs to get the value from the form inputs, insert them into the `<span>`s where the information should go, and then un-hide the story itself. We know how to insert text into our pages, so let's get that working before we try to get the data from the form:

### **scripts.js**

```
$(document).ready(function() {  
  $("#blanks form").submit(function() {  
    $(".person1").append("blah blah");  
    $(".person2").append("blah blah");  
    $(".animal").append("blah blah");  
    $(".exclamation").append("blah blah");  
    $(".verb").append("blah blah");  
    $(".noun").append("blah blah");  
  
    $("#story").show();  
  });  
});
```

The `submit()` function attaches an event listener for when a form is submitted. It's different from the `click()` function because a form can be submitted by clicking the submit button, of course, but it can also be submitted by pressing Enter while a form field is selected.

But when we submit the form, the story briefly flashes and then disappears. What's happening is that, by default, forms try to send the information somewhere, and since we haven't told it where to submit to, the page just refreshes and erases our hard work. If you look in the URL bar, you can see that there's a `?` at the end of the address now. This is your clue that the form has been submitted to nowhere and the page refreshed. We need to prevent the default action for the form:

### **scripts.js**

```
$(document).ready(function() {  
  $("#blanks form").submit(function(event) {  
    $(".person1").append("blah blah");  
    $(".person2").append("blah blah");  
    $(".animal").append("blah blah");  
    $(".exclamation").append("blah blah");  
    $(".verb").append("blah blah");  
    $(".noun").append("blah blah");  
  
    $("#story").show();  
  
    event.preventDefault();  
  });  
});
```

Notice that we've added a parameter `event` to the callback function we passed to `submit()` method. When the form is submitted, jQuery runs the callback function and passes in something as an argument. We don't know what this "something" is - we just know that it somehow represents the event of the form being submitted, and so we give the parameter the name `event`. And we know that if we call the method `preventDefault()` on this `event` thing, it will stop the form from submitting.

This is a pretty confusing concept. If you aren't at least 70% clear on the previous paragraph, stop and re-read it, slowly and carefully. If you are at least 70% clear, come back and re-read it a couple more times after you finish this lesson. You might also re-visit the lesson on Writing functions, which includes an explanation of arguments and parameters.

Now, we need to actually get the values from the form. If we open up the JavaScript console on the page and run:

```
$("#input#person1").val();
```

We can see that this jQuery method returns the value from the input as a string. Since the `append()` method takes a string as an argument, we can update our code like this:

**scripts.js**

```
$(document).ready(function() {  
  $("#blanks form").submit(function(event) {  
    var person1Input = $("input#person1").val();  
    var person2Input = $("input#person2").val();  
    var animalInput= $("input#animal").val();  
    var exclamationInput = $("input#exclamation").val();  
    var verbInput = $("input#verb").val();  
    var nounInput = $("input#noun").val();  
  
    $(".person1").append(person1Input);  
    $(".person2").append(person2Input);  
    $(".animal").append(animalInput);  
    $(".exclamation").append(exclamationInput);  
    $(".verb").append(verbInput);  
    $(".noun").append(nounInput);  
  
    $("#story").show();  
  
    event.preventDefault();  
  });  
});
```

Now our page works. Hooray!

There's one last thing to fix, though. If you don't refresh the page and you change the value of one of the inputs, it just adds it after the first value, instead of replacing it. We need to replace the existing text rather than just appending to it:

**scripts.js**

```
$(document).ready(function() {  
    $("#blanks form").submit(function(event) {  
        var person1Input = $("input#person1").val();  
        var person2Input = $("input#person2").val();  
        var animalInput= $("input#animal").val();  
        var exclamationInput = $("input#exclamation").val();  
        var verbInput = $("input#verb").val();  
        var nounInput = $("input#noun").val();  
  
        $(".person1").text(person1Input);  
        $(".person2").text(person2Input);  
        $(".animal").text(animalInput);  
        $(".exclamation").text(exclamationInput);  
        $(".verb").text(verbInput);  
        $(".noun").text(nounInput);  
  
        $("#story").show();  
  
        event.preventDefault();  
    });  
});
```

## NEW STUFF STARTS HERE

Before we move on, I want to introduce you to the idea of **variable scope**. A variable is only available within the function in which it was defined. Here's an example:

### variable-scope-example.html

```
<!DOCTYPE html>  
<html>  
  <head>  
    <script src="js/jquery-1.10.2.js"></script>  
    <script src="js/scripts.js"></script>  
    <title>An adventure in variable scope</title>  
  </head>  
  <body>  
    <div id="click-one">click me first</div>  
    <div id="click-two">click me second</div>  
  </body>  
</html>
```

### scripts.js

```
$(document).ready(function() {  
  
    $("div#click-one").click(function(event) {  
        var whatToSay = "Hello!";  
        alert(whatToSay);  
    });  
  
    $("div#click-two").click(function(event) {  
        alert(whatToSay);  
    });  
});
```

If we click click me first, a dialog box will pop open that says "Hello!". If we then click click me second, what will happen? Nothing, and the JavaScript console will have the error `Uncaught ReferenceError: whatToSay is not defined`. That's because the `whatToSay` variable is **scoped** to the function in which it was defined.

Now, I'm going to show you something you should never do, just so that you understand what's going on if you see it elsewhere:

### scripts.js

```
$(document).ready(function() {  
  
    $("div#click-one").click(function(event) {  
        whatToSay = "Hello!";  
        alert(whatToSay);  
    });  
  
    $("div#click-two").click(function(event) {  
        alert(whatToSay);  
    });  
});
```

I've removed the `var` keyword from before defining `whatToSay`. Now, if we click click me first and then click click me second, the dialog box will open up both times. Using the `var` keyword is what scopes your variables to the function in which they are defined. Without `var`, your variables become **global variables** available anywhere.

This code is very small, so you may not see the problem with using a global variable. But imagine a code base that's thousands or tens of thousands of lines long. If you see a function that looks like:

```
function(event) {  
    alert(whatToSay);  
}
```

it would be incredibly difficult to figure out where `whatToSay` was defined. And if `whatToSay` was used and changed in multiple places, it would be next to impossible to figure out where it's value was last set. Using scoped variables lets us easily follow the flow of our application, by explicitly passing variables as arguments to other functions when needed.

## Summary

Capture input when a form is submitted:

```
<form id="some-form">
  <label for="some-input">Your input:</label>
  <input id="some-input" type="text">

  <button type="submit" class="btn">Submit!</button>
</form>
```

```
$("#form#some-form").submit(function(event) {
  var someInput = $("#input#some-input").val()

  event.preventDefault();
});
```

If you submit your form and then there's a `?` at the end of the address bar, you forgot to put `event.preventDefault();` , or you attached your event listener to the wrong form.

Replace text:

```
$(".some-class").text("New text");
```

Variables defined with `var` are scoped to the function in which they are defined. Omitting `var` creates a global variable and is a bad practice.



## 09.Forms Practice

In the Forms video, we learned:

- The HTML structure of a form
- The `.submit()` function
- `event.preventDefault()`
- Using `.val()` to gather the input values from the form
- Variable scope

Okay, now it's your turn.

- Follow along with the video and make a Mad Libs of your own.
- Make a form letter where the user inputs their name, and they get back a letter addressed to them. Feel free to use lorem ipsum for the text of the letter. Use Bootstrap and your knowledge of CSS to make the letter look like a letter.
- Create a page that asks the user to type something into a field, and when they submit the form, have the page say it back to them in all uppercase.
- Create a website with a form on it to take the user's full name and address as if they were ordering a product from you. After they submit the form you should show them a receipt. Thank them for purchasing whatever it is you are selling, and be sure to use their full name. Then reprint their address as a confirmation of shipping details.
- Create a website for booking appointments with you. The user should enter their name, the date they would like for their appointment, as well as the start and end times. Again, after they submit the form you should show them a confirmation page telling them that the booking was successful, and be sure to reprint all the information they entered into your form for confirmation.
- Use a form to let the user fill out a survey. Ask them to fill in their name, their favorite foods, their favorite music, and any other information you feel like. Explore some of the other input types. You can read about them here. For example, you can have them pick a favorite color with `<input id="color" name="color" class = form-control type="color">` .

# 10.Attributes

Embedded Video: <https://www.youtube.com/watch?v=8-lfOO7fDd0?rel=0>

We're almost done learning the basics of jQuery. There's one more especially useful tool I want to show you. Let's start with this HTML:

## colors.html

```
<!DOCTYPE html>
<html>
  <head>
    <link href="css/bootstrap.css" rel="stylesheet" type="text/css">
    <link href="css/styles.css" rel="stylesheet" type="text/css">
    <script src="js/jquery-1.9.1.js"></script>
    <script src="js/scripts.js"></script>
    <title>Colors</title>
  </head>
  <body>
    <div class="container">
      <h1>What's your favorite color?</h1>
      <p>Click a button to choose what color you'd like this page to be:</p>

      <button class="btn btn-success" id="green">Green</button>
      <button class="btn btn-warning" id="yellow">Yellow</button>
      <button class="btn btn-danger" id="red">Red</button>
    </div>
  </body>
</html>
```

Now, some JavaScript:

## scripts.js

```
$(document).ready(function() {  
  $("#button#green").click(function() {  
    $("body").addClass("green-background");  
  });  
  
  $("#button#yellow").click(function() {  
    $("body").addClass("yellow-background");  
  });  
  
  $("#button#red").click(function() {  
    $("body").addClass("red-background");  
  });  
});
```

When we click the green button, for example, jQuery will add the green-background class to the `<body>` . Now, let's create the CSS classes to actually give the page the background color we want:

### **styles.css**

```
.green-background {  
  background-color: green;  
}  
  
.yellow-background {  
  background-color: yellow;  
}  
  
.red-background {  
  background-color: red;  
}
```

And our page works! But there's a problem. If we click any one button, everything is fine. But if we then click a button of a lighter color, it won't change the background. So if we click Yellow and then Red, it works. But if we click then click Green after Red, nothing happens. We need to remove the other classes before applying the new class. Fortunately, that's easy:

### **scripts.js**

```
$(document).ready(function() {  
  $("#button#green").click(function() {  
    $("body").removeClass();  
    $("body").addClass("green-background");  
  });  
  
  $("#button#yellow").click(function() {  
    $("body").removeClass();  
    $("body").addClass("yellow-background");  
  });  
  
  $("#button#red").click(function() {  
    $("body").removeClass();  
    $("body").addClass("red-background");  
  });  
});
```

And now everything works perfectly!

`removeClass()` without an argument removes all classes from the selected element; if you'd like to just remove a specific class, you can pass it in as an argument (or pass in multiple classes separated by spaces).

There is another way to change the CSS of an element with jQuery:

```
$("#button#green").click(function() {  
  $("body").css("background-color", "green");  
});
```

You've seen this before, in my trick for making sure the correct element is selected. However, this is a bad approach for real code, for two reasons. First, it's mixing our concerns. JavaScript is responsible for how the page behaves; CSS is responsible for how the page looks. Here, we've put something about how the page looks in our JavaScript. This makes it difficult for other programmers who might need to change this page in the future to know where to look for the appropriate code. Second, you lose one of the biggest powers of CSS: the ability to create a class and re-use that style across elements and pages. Here, when we've put a style in our JavaScript, it can't be re-used. So, when you want to change the style of your page, stick to adding and removing classes.

Before we move on from our intro to jQuery, you should take a look at the jQuery documentation. It will probably feel a bit overwhelming at first, and there is a lot in there we haven't even begun to cover. But check out the sections on Effects, Mouse Events, Form Events, Manipulation, and Traversing. There's a lot in there you'll be able to understand. And in the future, when you're trying to figure out how to do something with jQuery, the documentation is a great place to look.

# Summary

Add a CSS class to an element:

```
$( "body" ).addClass( "my-class" );
```

Remove a CSS class from an element:

```
$( "body" ).removeClass();
```

Use the [jQuery documentation](#) as a reference.

# 11.jQuery Attributes Practice

In the Attributes video, we learned:

- `.removeClass()`
- `.addClass()`
- Why using `.css()` should not be used for anything other than verifying if an element is selected correctly.

Here are some exercises for you to practice using jQuery:

- Most people like dark text on a light background, but some people prefer light text on a dark background. Update your dog blog or cat homepage to include a button that lets your users switch to light on dark.
- Now, update this page so that users can switch back to the original color scheme.
- Update your boring lecture page so that when somebody clicks on a paragraph, it highlights it and adds a box around the edges.

## 13.jQuery Code Review

### Favorite things list

Create a webpage (using HTML, CSS, JavaScript and JQuery) where a user can create a list of their favorite things.

When you have reviewed your code for all of the objectives, submit your code to the JQuery Code Review on Epicenter.

**Due:** Friday, October 9th at midnight

Below are the objectives the instructor will use to review your code.

### This week's code review objectives

- All previous standards (last review's objectives) are in place
- Accurately links to scripts.js in the `<head>` tag
- List of favorite things displays on page when button is clicked using `.show()`
- Variable names are descriptive of what they represent
- Web page is styled using Bootstrap and CSS

### Last review's objectives

- Code has proper indentation and spacing
- Commits are made regularly with clear messages that finish the phrase "It will..."
- Project README that includes:
  - author name
  - program name
  - description
  - **setup instructions with link to project on github.io**
  - copyright and license information

## 12. More jQuery Practice

<http://jqexercise.droppages.com/>



## 47.Entry to Level 2 Test

**Embedded Video:** <https://www.youtube.com/watch?v=IUXjViTZJ3s?rel=0>

For acceptance into Level 2 courses at Opteamize, an IBM Coding School, each student is expected to be independently proficient in these areas of knowledge:

- Markdown
- HTML and CSS
- Bootstrap
- Command line
- Git
- GitHub Pages
- Behavior Driven Development
- JavaScript
- jQuery

The Level 2 Placement Test provides students an opportunity to demonstrate the necessary proficiency to begin Level 2 coursework. You may use documentation resources for the languages and concepts you use, but you should not get help from other people or look for solutions others have come up with before.

If you feel you are not yet able to independently complete all of the areas in the challenge, we encourage you to rework through the Opteamize Level 1 coursework.

To proceed with the test, carefully read all of the requirements for coding and submission. You will only have the opportunity to submit your work for placement once.

### Placement Test Instructions

Create a web application that takes a positive number from a user and returns a range of numbers from 1 to the chosen number with the following exceptions:

- Numbers divisible by 3 are replaced with "ping"
- Numbers divisible by 5 are replaced with "pong"
- Numbers divisible by 15 are replaced with "pingpong"

Watch the video to see a live version of the expected application. Your final work should look like the examples below including the following information in the sidebar:

- your full name
- a link to your GitHub repository for this test

- the name of the course you want to be placed into

Your page and README should mirror these examples. Slight variances in color and font will be accepted but should be matched as closely as possible.

### Layout and styling example

The screenshot shows a web application titled "Ping-Pong" with a light orange header. The main content area is light gray and divided into two columns. The left column has a green background and contains the following text: "About Ping Pong", "Created by: Kit Fisto", "GitHub Username/URL: greensaber", "Level 2 Test for: Java". The right column contains the heading "Let's play!", the label "Rules:", and the text "Ping Pong will count up to your number from 1 with the following exceptions:". Below this are three numbered rules: "1. Numbers divisible by 3 become 'ping'", "2. Numbers divisible by 5 become 'pong'", and "3. Numbers divisible by 3 and 5 become 'pingpong'". A form with the label "Enter your number here:" and a text input field containing "17" is followed by a "Ping Pong!" button. Below the button is a list of results: "1", "2", "ping", "4", "pong", "ping", "7", "8", "ping", "pong", "11", "ping", "13", "14", "pingpong", "16", and "17".

# Ping-Pong

## About Ping Pong

Created by:  
Kit Fisto  
GitHub Username/URL:  
greensaber  
Level 2 Test for:  
Java

## Let's play!

Rules:

Ping Pong will count up to your number from 1 with the following exceptions:

1. Numbers divisible by 3 become "ping"
2. Numbers divisible by 5 become "pong"
3. Numbers divisible by 3 and 5 become "pingpong"

Enter your number here:  Ping Pong!

- 1
- 2
- ping
- 4
- pong
- ping
- 7
- 8
- ping
- pong
- 11
- ping
- 13
- 14
- pingpong
- 16
- 17

## README example

# ping-pong

**By: Epicodus Staff**

Ping Pong is a sample JavaScript application for demonstrating basic proficiency in JavaScript, jQuery, Git, Markdown, HTML, Bootstrap, CSS, and BDD.

A user enters a number and is shown a range of numbers from 1 to the number entered with the following exceptions:

- Numbers divisible by 3 are replaced with "ping"
- Numbers divisible by 5 are replaced with "pong"
- Numbers divisible by 15 are replaced with "pingpong"

## Installation

Install ping-pong by cloning this repository:

```
https://github.com/sampleRepo/ping-pong
```

## License

MIT License. Copyright 2015 Epicodus

## Requirements

Please do not add additional features or user interface functionality beyond the scope of what is requested in the instructions. Before submission, review your code to ensure it meets the following objectives. If you are unclear what is expected in an objective, follow the link to the lessons that cover that content.

- Code has proper indentation and spacing
- Bootstrap, HTML and CSS are used to re-create the layout and styling example
- JavaScript variable names are clear, descriptive, and follow naming conventions
- JavaScript specs have been written using the Mocha spec framework and Chai assertion library to cover all user input possibilities
- All JavaScript specs pass when run
- JavaScript function(s) have been written to process user input
- jQuery is used to update the DOM after user input is processed
- Working code is stored in a GitHub repository with regular commits (Commits should

follow the standard tense completing the sentence "This commit will...")

- Markdown is used in a README file with styles and sections like the README example
- The site is successfully deployed using GitHub's GitHub Pages option

Opteamize Coding School instructors will review your code and provide one of the following responses for each of the above objectives:

- Code meets the standard ALL of the time.
- Code meets the standard MOST of the time.
- Code does NOT meet the standard.

IMPORTANT NOTE: All objectives must meet the standard ALL or MOST of the time to enroll in a Level 2 course. If an objective is NOT met, the student is enrolled in our Intro to Programming course. Additional feedback regarding performance on the Code Challenge is not provided. Because our teachers have limited time to grade these tests, no resubmissions will be allowed.

## Submission

When you are confident that your code meets the objectives of the challenge, you can submit on [edu.opteamize.in](https://edu.opteamize.in) after navigating to the Coding/Grading tab.

## Results

Your code will be reviewed within one week of submission. You will be notified by email with your results.