**Notes**

**HTML**

The bold and strong tags format the enclosed text to be displayed as bold text. The strong tag also marks the text as important

See the bold tag is action below:

<b>Text goes here</b>

See the strong tag is action below:

<strong>Text goes here</strong>

The italic and emphasis tags format the enclosed text to be displayed as italicized text. The emphasis tag also marks the text as important.

See the italic tag in action below:

<i>Text goes here</i>

See the emphasis tag in action below:

<em>Text goes here</em>

The superscript and subscript tags format the enclosed text to be displayed as superscript or subscript, respectively. Neither tag marks the text as important.

See the superscript and subscript tags in action below:

Superscript: <sup>Text goes here</sup>

Subscript: <sub>Text goes here</sub>

The inserted tag formats the enclosed text as underlined. The deleted tag formats the enclosed text as crossed out. Neither tag marks the text as important.

See the inserted and deleted tags in action below:

Inserted: <ins>Text goes here</ins>

Deleted: <del>Text goes here</del>

The marked tag formats the enclosed text as highlighted.

See the marked tag in action below:

<mark>Text goes here</mark>

The blockquote tag marks the enclosed text as quoted text. Browsers usually display quoted text as being indented with line breaks above and below the indented block.

See the blockquote tag in action below:

Here is an example of some blockquoted text:

<blockquote>

Lorem ipsum dolor sit amet, magna sed quia, elit non

ut corrupti, aenean commodo elit, tincidunt platea

purus quis at dolor a, felis nunc massa pede cras

quis. Lacinia fusce, praesent at pede phasellus, dui

etiam in luctus.

</blockquote>

That was a blockquote!

The code tag displays the enclosed text in a fixed-width font, like you see in many text editors used for programming. It is often used to set apart and display programming code you wish to show your users. This is commonly used on tutorial websites to demonstrate coding concepts.

See the code tag in action below:

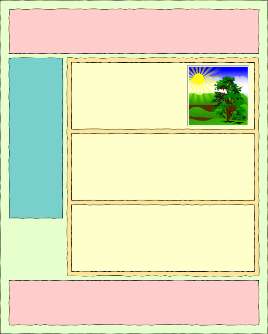
<code>Text goes here</code>

See an HTML comment in action below:

This text will appear on the page.

<!-- This is a comment and will not display -->

This will also appear on the page.



<!DOCTYPE html>

<**html**>

<**head**>

<**title**>Page with Semantic Elements</**title**>

</**head**>

<**body**>

<**div**>

<**div**>

...

</**div**>

<**div**>

...

</**div**>

<**div**>

<**div**>

<**div**>

</**div**>

</**div**>

<**div**>

</**div**>

<**div**>

</**div**>

</**div**>

<**div**>

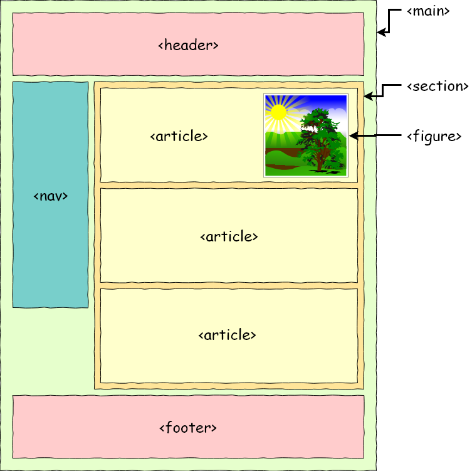
</**div**>

</**div**>

</**body**>

</**html**>

* main: a block that represents the primary section of a page
* header: a block at the top of a page
* footer: a block at the bottom of a page
* nav: a block that includes navigation elements, such as buttons or menus
* article: a block that contains nested blocks related to a single topic, like a newspaper article
* section: a block that identifies a section of content with a specific purpose, like the section of a newspaper
* form: a block that contains the elements of a form
* figure: A block that contains an image and information about that image



<!DOCTYPE html>

<**html**>

<**head**>

<**title**>Page with Semantic Elements</**title**>

</**head**>

<**body**>

<main>

<**header**>

...

</**header**>

<**nav**>

...

</**nav**>

<**section**>

<**article**>

<**figure**>

</**figure**>

</**article**>

<**article**>

</**article**>

<**article**>

</**article**>

</**section**>

<**footer**>

</**footer**>

</main>

</**body**>

</**html**>

<a>Clickable content</a>

<**a** href="https://www.google.com">Google</**a**>.

<**a** href="https://www.google.com">

<**img** src="images/portrait.jpg" />

</**a**>

<**a** href="https://www.google.com" target="\_blank">Google</**a**>

\_blank and \_self. \_blank will open the link in a new browser tab while \_self will open in the same tab, replacing the current page.

Internal Links

<**a** href="about.html">About Us</**a**>

<**a** href="products/desktop-computer.html">Featured Product</**a**>

### Bookmarks

<**body**>

<**h1**>Bookmark Links in HTML</**h1**>

<**p**><**a** href="#target">This is the question.</**a**></**p**>

<**figure**><**img** src="https://cdn.pixabay.com/photo/2018/06/24/20/32/ivy-3495403\_1280.jpg" alt="" /></**figure**>

<**p** id="target">This is the target.</**p**>

<**figure**><**img** src="https://cdn.pixabay.com/photo/2018/06/18/06/43/nature-3481966\_1280.jpg" alt="" /></**figure**>

</**body**>

## **Common Image Attributes**

### SRC Attribute

<**img** src="https://cdn.pixabay.com/photo/2017/06/21/00/22/dog-2425528\_960\_720.jpg" />

### ALT Attribute

<**img** src="https://cdn.pixabay.com/photo/2017/06/21/00/22/dog-2425528\_960\_720.jpg" alt="Photo of a puppy and a kitten." />

### STYLE Attribute

<**img** src="https://cdn.pixabay.com/photo/2017/06/21/00/22/dog-2425528\_960\_720.jpg"

alt="Photo of a puppy and a kitten." style="**border**: thick red solid;" />

<**img** src="https://cdn.pixabay.com/photo/2017/06/21/00/22/dog-2425528\_960\_720.jpg"

alt="Photo of a puppy and a kitten."

style="**border**: thick red solid; **width**: 300px;" />

<**img** src="https://cdn.pixabay.com/photo/2017/06/21/00/22/dog-2425528\_960\_720.jpg"

alt="Photo of a puppy and a kitten."

style="**border**: thick red solid; **width**: 300px; **height**:465px" />

One way to add an image to a website is by specifying the URL of that image in the src attribute. For example:

<**img** src="https://cdn.pixabay.com/photo/2017/06/21/00/22/dog-2425528\_960\_720.jpg" alt="Photo of a puppy and a kitten." />

Using a relative path to a file inside our website then, the img element would look like:

<**img** src="images/portrait.jpg" alt="Portrait of Marci Kitzmiller" />

## **Using Lists**

<**h2**>Ordered Lists</**h2**>

<**ol** type="A">

<**li**>First Item</**li**>

<**li**>Second Item</**li**>

<**li**>Third Item</**li**>

</**ol**>

| **Type** | **Purpose** |
| --- | --- |
| type=”1” | Numbers used to sequence the list (default) |
| type=”A” | Uppercase letters used to sequence the list |
| type=”a” | Lowercase letters used to sequence the list |
| type=”I” | Uppercase Roman Numerals used to sequence the list |
| type=”i” | Lowercase Roman Numerals used to sequence the list |

<**h2**>Unordered Lists</**h2**>

<**ul** style="**list-style-type**: square;">

<**li**>Milk</**li**>

<**li**>Bread</**li**>

<**li**>Cereal</**li**>

</**ul**

| **Type** | **Purpose** |
| --- | --- |
| list-style-type=”disc” | Bullet or Filled Circle indicator (default) |
| list-style-type=”square” | Square indicator |
| list-style-type=”circle” | Hollow circle indicator (similar to disc, but without the inner fill) |
| list-style-type=”none” | No indicator |

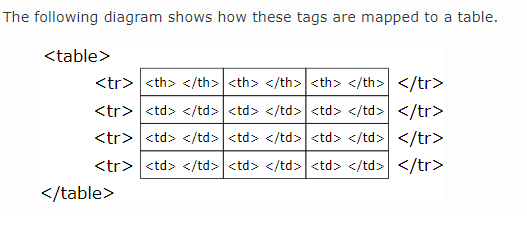
## **Definition Lists**

Although not as common as the other two list types, the definition list provides a useful way to list terms and their associated definition.

There are three elements that make up a definition list:

1. The <dl></dl> tags designate the type of list and serve as a grouping container for all items in the list.
2. The <dt></dt> tags designate the term to be defined
3. The <dd></dd> tags provide the definition/description for the term it is associated with.
4. <**h2**>Definition Lists</**h2**>
5. <**dl**>
6. <**dt**>One</**dt**>
7. <**dd**>The first positive number in our counting system.</**dd**>
8. <**dt**>Two</**dt**>
9. <**dd**>The first whole number greater in value than one.</**dd**>
10. <**dt**>Three</**dt**> <**dd**>The next whole number in increasing value after two.
11. </**dd**>
12. </**dl**>

|  |  |
| --- | --- |
| **Element Name** | **Element Purpose** |
| <table></table> | marks the beginning and end of a table |
| <tr></tr> | table row: marks the beginning and end of a row within a table |
| <th></th> | table header: marks the beginning and end of a cell in a header row |
| <td></td> | table data: marks the beginning and end of a cell in a table row |



### Table with Headings

<!DOCTYPE html>

<**html** lang="en">

<**head**>

<**title**>Tables in HTML</**title**>

</**head**>

<**body**>

<**table** border="1">

<**tr**>

<**th**>First Name</**th**>

<**th**>Last Name</**th**>

<**th**>Phone</**th**>

</**tr**>

<**tr**>

<**td**>John</**td**>

<**td**>Doe</**td**>

<**td**>555-1212</**td**>

</**tr**>

<**tr**>

<**td**>Sally</**td**>

<**td**>Smith</**td**>

<**td**>555-1123</**td**>

</**tr**>

<**tr**>

<**td**>Sam</**td**>

<**td**>Jones</**td**>

<**td**>555-4321</**td**>

</**tr**>

</**table**>

</**body**>

</**html**>

### Table with Caption

<**table** border="1">

<**caption**>Contact Information</**caption**>

<**tr**>

<**th**>First Name</**th**>

<**th**>Last Name</**th**>

<**th**>Phone</**th**>

</**tr**>

<**tr**>

<**td**>John</**td**>

<**td**>Doe</**td**>

<**td**>555-1212</**td**>

</**tr**>

<**tr**>

<**td**>Sally</**td**>

<**td**>Smith</**td**>

<**td**>555-1123</**td**>

</**tr**>

<**tr**>

<**td**>Sam</**td**>

<**td**>Jones</**td**>

<**td**>555-4321</**td**>

</**tr**>

</**table**>

### Table Width

<**table** border="1" width="90%">

<**caption**>Contact Information</**caption**>

<**tr**>

<**th**>First Name</**th**>

### Changing Background Colors

<**table** border="1" width="90%">

<**caption**>Contact Information</**caption**>

<**tr** bgcolor="palegreen">

<**th**>First Name</**th**>

### Cells that Span Multiple Columns

<**table** border="1" width="90%">

<**caption**>Contact Information</**caption**>

<**tr**>

<**th** colspan="2">Name</**th**>

<**th**>Phone</**th**>

</**tr**>

<**tr**>

<**td**>John</**td**>

<**td**>Doe</**td**>

<**td**>555-1212</**td**>

</**tr**>

<**tr**>

<**td**>Sally</**td**>

<**td**>Smith</**td**>

<**td**>555-1123</**td**>

</**tr**>

### Cells That Span Multiple Rows

<**th**>Category</**th**>

<**th** colspan="2">Name</**th**>

<**th**>Phone</**th**>

</**tr**>

<**tr**>

<**td** rowspan="3">Friends:</**td**>

<**td**>John</**td**>

<**td**>Doe</**td**>

<**td**>555-1212</**td**>

</**tr**>

<**tr**>

<**td**>Sally</**td**>

<**td**>Smith</**td**>

<**td**>555-1123</**td**>

</**tr**>

<**tr**>

<**td**>Sam</**td**>

<**td**>Jones</**td**>

<**td**>555-4321</**td**>

| **Form Tag Name** | **Purpose** |
| --- | --- |
| <form> </form> | Marks the beginning and end of the form. |
| <input /> | Represents some type of input element based on its type attribute. Valid type attribute values include: text, radio, submit, password, checkbox, button, and more. |
| <select> </select> | Marks the beginning and end of a drop-down list. |
| <option> </option> | Marks the beginning and end of an option in the drop-down list. |
| <textarea> </textarea> | Marks the beginning and end of a multi-line text input box. |

If the form is to properly submit its data to a server, the <form> tag must have the following two attributes:

1. action – specifies the URL where the form data will be submitted.
2. method – specifies the HTTP method (POST or GET) that should be used to submit the data.
3. <**body**>
4. <**h1**>Form Example</**h1**>
5. <**form** action="fakePage.html" method="POST">
7. </**form**>

## **Input Tag**

<input type="type of input" name="name of input" >

text

<**form** action="fakePage.html" method="POST">

Text Input: <**input** type="text" name="myTextInput" />

</**form**>



radio

<**form** action="fakePage.html" method="POST">

Text Input: <**input** type="text" name="myTextInput" /><**br** />

Radio Buttons:<**br** />

<**input** type="radio" name="answer" value="YES" checked />Yes<**br** />

<**input** type="radio" name="answer" value="NO" />No<**br** />

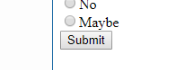
<**input** type="radio" name="answer" value="MAYBE" />Maybe<**br** />

</**form**>



submit

<**input** type="submit"/><**br** />



reset

<**input** type="submit" /><**input** type="reset" /><**br** />



Password

Password Input: <**input** type="password" name="userPassword" /><**br** />



Checkbox

<**input** type="checkbox" name="myChoices" value="Salad" checked/>Salad<**br** />



### Select Tag

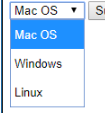
<**select** name="phones">

<**option** value="iPhone6">Mac OS</**option**>

<**option** value="GalaxyS6">Windows</**option**>

<**option** value="Nexus5">Linux</**option**>

</**select**>



### Textarea Tag

<**textarea** name="briefDescription"></**textarea**><**br** />



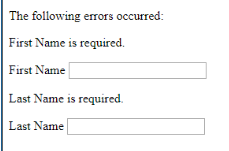
## **Making fields required**

Text Input: <input type="text" name="myTextInput" **required** />

**CSS**

.warning class="warning"

#about id="about"



By default, we do not need these warnings to show, so we can hide them with CSS. When we want to target a class, the selector starts with a dot (.), like .warning. Because the class applies to both div and p elements, we can use a single rule to target all of them.

<!DOCTYPE html>

<**html**>

<**head**>

<**title**>CSS Examples</**title**>

<**style**>

.warning {

**display**: none;

}

</**style**>

</**head**>

<**body**>

<**div** class="warning">

<**p**>The following errors occurred:</**p**>

</**div**>

<**p** class="warning">First Name is required.</**p**>

<**label** for="firstName">First Name</**label**>

<**input** type="textbox" name="firstName"/>

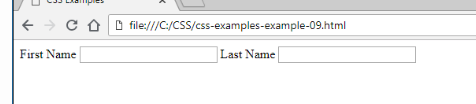
<**p** class="warning">Last Name is required.</**p**>

<**label** for="lastName">Last Name</**label**>

<**input** type="textbox" name="lastName"/>

</**body**>

</**html**>



## **Apply a Rule to More Than One Selector**

<**style**>

#product1, #product3 {

**color**: blue;

}

#product2 {

<**style**>

.warning {

**font-weight**: bold;

**color**: red;

}

**div**.warning {

**font-style**: italic;

}

</**style**>

</**head**>

<**body**>

<**div** class="warning">

<**p**>The following errors occurred:</**p**>

</**div**>

<**p** class="warning">First Name is required.</**p**>

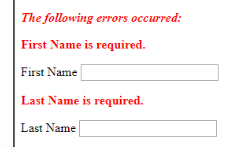
<**label** for="firstName">First Name</**label**>

<**input** type="textbox" name="firstName"/>

<**p** class="warning">Last Name is required.</**p**>

<**label** for="lastName">Last Name</**label**>

<**input** type="textbox" name="lastName"/>



## **Internal Style Sheets**

## **External Style Sheets**

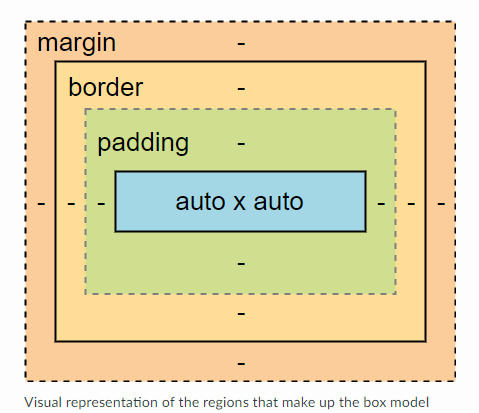
<link rel="stylesheet"

type="text/css"  
 href="sampleStyles.css" />

You can also link to an external style sheet that is already published to the Web, using @import.

@import url("http://mysite.com/styles/sharedStyles.css");

## **The CSS Box Model**



* padding: whitespace around the content, between the content and the border
* border: a line that creates a visual separation between the box's content and the content around it
* margin: whitespace between the border and the content around the box

#### **Tip**

The term whitespace comes from printing layouts and it refers to planned empty space in a page layout. However, it is actually transparent space, so "whitespace" will be whatever color the background uses.

Whitespace is an important part of a page layout because it can help make a page look less cluttered, as well as helping the user identify distinct blocks of content on the page.

Content is what is at the core of the element, which could include images or text. In this particular example, the core is the text "Your Journey Begins Here." The dimensions of the content can be dependent on the content itself, or it can be controlled via the height and width CSS properties. These dimensions can take fixed and relative values. An example of a fixed value is using a specific number of pixels (e.g., 216px). An example of a relative value is to specify a percentage of the parent container – the declaration width=90%; means that the content's width will be displayed at a size equivalent to 90% of its parent container, for example, but the exact width depends on the width of the parent container.

top padding of 8px, a right padding of 16px, a bottom padding of 4px, and a left padding of 4px.

padding: 8px 16px 4px 4px;

If all the sides shared the same padding value

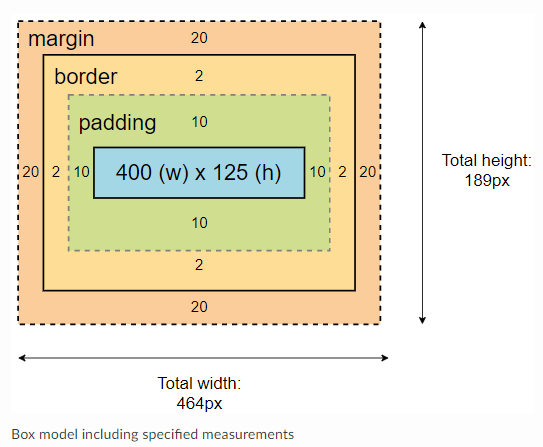
padding: 5px;

The border encloses the padding and the content. The CSS border, border-top, border-left, border-right, and border-bottom shorthand properties give us more flexibility, in that they include:

* Thickness (border-width) – typically in medium, thin, thick, or a fixed length
* Color (border-color) – any CSS-accepted color format, including color names, rgb(a), or hexadecimal.
* Style (border-style) – solid, double, dashed, dotted, groove, ridge, inset, outset, hidden, and none

Use the border shorthand when the style applies to all borders. For example, the following rule would apply a thin, gray, solid border on all sides:

border: thin #888 solid;



### background-image

Specifies the URL of the image to use, using the format background-image:url("image-address.png");.

### background-repeat

* no-repeat: does not repeat
* repeat-x: repeats horizontally
* repeat-y: repeats vertically
* repeat: repeats both horizontally and vertically

### background-attachment

* scroll: scrolls
* fix: anchored/fixed in place

### background-position

Determines the initial position of an image with two values that apply to X (horizontal) and Y (vertical) positioning; values can be percentages, lengths, or keywords.

* X position keywords include:
  + right
  + left
  + center
* Y position keywords include:
  + top
  + bottom
  + center

### Background Shorthand

All of these background properties can be shortened to the format:

background: color image repeat attachment position;

If any of those properties are not specified, they are assumed to be their default values. For example, this:

background-image: url("../images/confetti.gif");

backround-repeat: repeat;

can be simplified to:

background: url("../images/confetti.gif") repeat;

**body** {

**background-image**:url("paper-1914901\_100.png");

**background-color**: #cccccc;

}

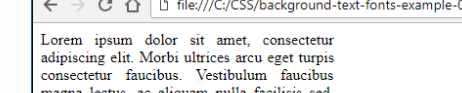
</**style**>

### Alignment

Much like how you can align text in a word processing document, you can do the same in HTML. The CSS attribute text-align allows us to align text to the left of the page, to the right, centered, or justifed. By default, text is aligned to its native position – left if the language's direction is ltr (left-to-right, like English) and right if the direction is rtl (right-to-left, like Arabic or Hebrew).

<**div** style="**text-align**: justify;**width**: 300px;">

Lorem ipsum dolor sit amet, consectetur



### Indentation

<**div** style="**text-indent**: 15%;**width**: 300px;">

Lorem ipsum dolor sit amet, consectetur

### Capitalization

* capitalize – capitalize the first character of each word
* uppercase – convert all characters TO UPPERCASE
* lowercase – CONVERT ALL CHARACTERS to lowercase
* none – no capitalization effects
* <**p** style="**text-transform**: uppercase;">
* Now this is crazy!
* <**span** style="**text-transform**: none; **color**: dodgerblue;">
* Yay!
* </**span**>
* <**span** style="**text-transform**: capitalize;">
* we love CSS!
* </**span**>



<**span** style="**text-decoration**:underline;">

<**span** style="**text-decoration**:overline;">

<**span** style="**text-decoration**:line-through;">



## **Font**

### Style

* italic – italicized font
* oblique – slanted font
* normal – default font style, no slanting

### Size

* Absolute – xx-small, x-small, small, medium, large, x-large, xx-large
* Length – em, ex, px, cm, mm, in, pt, or pc
* Percentage – %
* Relative – smaller, larger

### Weight

The weight of a font is the thickness of the font. The font-weight CSS property controls the strength of the font being styled. Weight values include:

* Numeric values - in increments of 100 between 100 and 900
* normal – default weight, in the middle
* bold – thick characters
* Relative – bolder, lighter

### Family

#### **Generic Families**

* serif – normal fonts with serifs (small lines often added at the end of the lines that make up the letter) such as Garamond, Georgia, New York, Times, and Times New Roman
* sans-serif – normal fonts without serifs such as Arial, Geneva, Helvetica, Lucida Sans, Trebuchet, and Verdana
* monospace – fixed-width fonts such as Courier, Consolas, Lucida Console, and Monaco
* cursive – fonts that emulate handwriting such as Comic Sans MS, Lucida Handwriting, and Zapf Chancery
* fantasy – decorative fonts such as Copperplate, Desdemona, Impact, and Kino
* If you want the browser to use a specific font, you can specify that as well, such as:
* font-family: 'Comic Sans MS';

## **Pseudo-Classes**

So far, we have examined styling HTML elements with CSS rules in their default states. However, sometimes you may need to style some elements in a particular state. Pseudo-classes are used for styling special states of HTML elements. They appear in the CSS selector section of a CSS rule, like this:

selector:pseudo-class {

/\* CSS declarations \*/

}

## **Links**

* link – unvisited link
* visited – visited link (meaning that you have recently opened the target page in your browser)
* hover – mouse over the link
* active – selected link

In this example, our requirements are:

* Unvisited links are green.
* Visited links are red.
* All links are underlined, except when hovering over a link.
* All links are in a monospace font, except when hovering over a link. When hovering over the link, the text is in cursive.
* We want to make sure it is clear which link we are selecting. Active (selected) links are bold.
* a {
* font-family: monospace;
* text-decoration: underline;
* font-weight: none;
* }
* a:link {
* color: green;
* }
* a:visited {
* color: red;
* }
* a:hover {
* text-decoration: none;
* color: #00c;
* font-family: cursive;
* }
* a:active {
* font-weight: bold;
* }

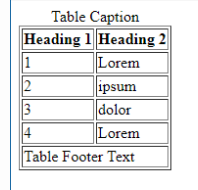
## **Tables**

Tables can be tricky HTML elements to style, in part because there are so many elements that make up a table. Before we get into the CSS side, though, let's look at three more tags that are helpful in table HTML.

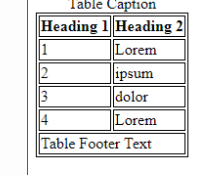
* <thead> </thead> – used for grouping table header content
* <tbody> </tbody> – used for grouping table body content
* <tfoot> </tfoot> – used for grouping table footer content
* <table>
* <caption>Table Caption</caption>
* <thead>
* <tr>
* <th>Heading 1</th>
* <th>Heading 2</th>
* </tr>
* </thead>
* <tbody>
* <tr>
* <td>1</td>
* <td>Lorem</td>
* </tr>
* <tr>
* <td>2</td>
* <td>ipsum</td>
* </tr>
* <tr>
* <td>3</td>
* <td>dolor</td>
* </tr>
* <tr>
* <td>4</td>
* <td>Lorem</td>
* </tr>
* </tbody>
* <tfoot>
* <tr>
* <td colspan="2">Table Footer Text</td>
* </tr>
* </tfoot>
* </table>

### Borders

By default, a table does not display borders. If we add the HTML attribute border="1" to our table tag (<table border="1">),



* Thickness (border-width) – typically in medium/thin/thick or a fixed width
* Color (border-color) – in any CSS-accepted color format
* Style (border-style) – solid, double, dashed, dotted, groove, ridge, inset, outset, hidden, and none
* table, th, td {
* border: 1px solid black;
* }

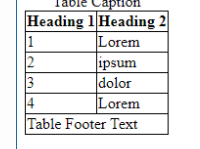


table, th, td {

border: 1px solid black;

border-collapse: collapse;

}



table, th, td {

border: 1px solid black;

border-collapse: collapse;

padding-top: 5px;

padding-right: 0px;

padding-bottom: 5px;

padding-left: 10px;

}

This can also be written with the shorthand syntax:

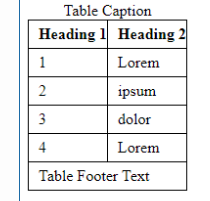
table, th, td {

border: 1px solid black;

border-collapse: collapse;

padding: 5px 0px 5px 10px;

}



### Widths and Heights

table {

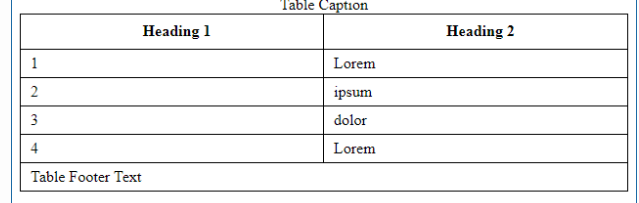
width: 100%;

}

th {

height: 25px;

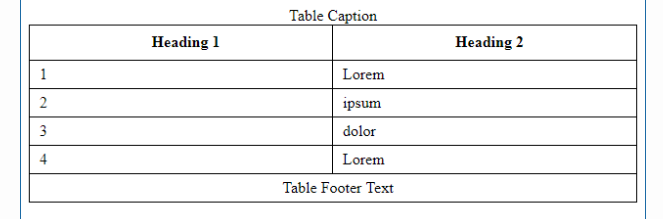
}



## **Text Alignment**

tfoot td {

}



### Styling Table Row Backgrounds

tbody tr:nth-child(even) {

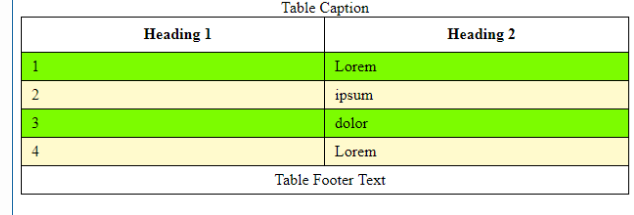
background-color: lemonchiffon;

}

tbody tr:nth-child(odd) {

background-color: lawngreen;

}



## **Final Check**

The complete source code for this table is:

<!DOCTYPE html>

<**html** lang="en">

<**head**>

<**title**>Links and Tables</**title**>

<**style**>

**table**, **th**, **td** {

**border**: 1px solid black;

**border-collapse**: collapse;

**padding**: 5px 0px 5px 10px;

}

**table** {

**width**: 100%;

}

**th** {

**height**: 25px;

}

**tfoot** **td** {

**text-align**: center;

}

**tbody** **tr**:**nth-child**(even) {

**background-color**: lemonchiffon;

}

**tbody** **tr**:**nth-child**(odd) {

**background-color**: lawngreen;

}

</**style**>

</**head**>

<**body**>

<**table**>

<**caption**>Table Caption</**caption**>

<**thead**>

<**tr**>

<**th**>Heading 1</**th**>

<**th**>Heading 2</**th**>

</**tr**>

</**thead**>

<**tbody**>

<**tr**>

<**td**>1</**td**>

<**td**>Lorem</**td**>

</**tr**>

<**tr**>

<**td**>2</**td**>

<**td**>ipsum</**td**>

</**tr**>

<**tr**>

<**td**>3</**td**>

<**td**>dolor</**td**>

</**tr**>

<**tr**>

<**td**>4</**td**>

<**td**>Lorem</**td**>

</**tr**>

</**tbody**>

<**tfoot**>

<**tr**>

<**td** colspan="2">Table Footer Text</**td**>

</**tr**>

</**tfoot**>

</**table**>

</**body**>

</**html**>

## **Bootstrap**

download the CSS and JavaScript files

Linking to a CDN

linking out to the same minified CSS and JavaScript files stored on their website

The abbreviation CDN stands for Content Delivery Network

In order for Bootstrap JavaScript to work, you must add jQuery and popper.js to your website, BEFORE adding Bootstrap.

## **Meta Elements**

<**head**>

<**meta** charset="utf-8">

<**meta** name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

## **Add Bootstrap CSS**

<**title**>Bootstrap Sample Page</**title**>

<!-- Bootstrap CDN copied from https://getbootstrap.com/docs/4.1/getting-started/introduction/ -->

<**link** rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css" integrity="sha384-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPMO" crossorigin="anonymous">

</**head**>

## **Add the JavaScript files**

Now we need to include the files required for the JavaScript content to work. Because Bootstrap is built using the popper.js and jQuery frameworks, the browser must load those frameworks first, before loading the Bootstrap framework, or Bootstrap will not work correctly.

Using the links provided on Bootstrap's Introduction page, add all three frameworks at the bottom of the body element, just above the </body> tag.

<!-- CDN links copied from https://getbootstrap.com/docs/4.1/getting-started/introduction/ -->

<**script** src="https://code.jquery.com/jquery-3.3.1.slim.min.js" integrity="sha384-q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo" crossorigin="anonymous"></**script**>

<**script** src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.3/umd/popper.min.js" integrity="sha384-ZMP7rVo3mIykV+2+9J3UJ46jBk0WLaUAdn689aCwoqbBJiSnjAK/l8WvCWPIPm49" crossorigin="anonymous"></**script**>

<**script** src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/js/bootstrap.min.js" integrity="sha384-ChfqqxuZUCnJSK3+MXmPNIyE6ZbWh2IMqE241rYiqJxyMiZ6OW/JmZQ5stwEULTy" crossorigin="anonymous"></**script**>

</**body**>

</**html**>

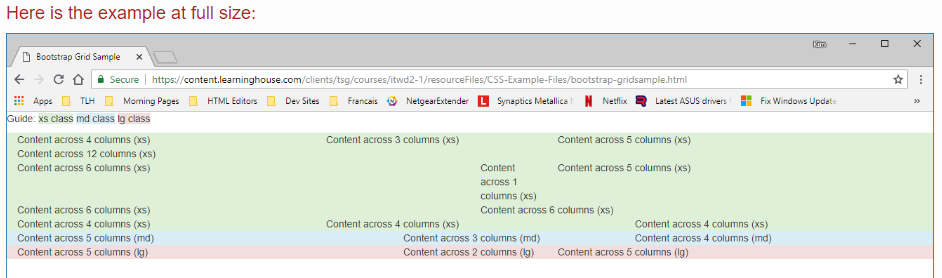
While JavaScript files can be added to the head element with the style sheets, that means the browser will have to read those files before it can display any of the page's content in the body element. To speed up the time required to load the content (and improve user experience), it is common practice to add larger JavaScript files inside the body element, below any content in the page and just above the closing </body> tag. That way the user can see and start to use the HTML content while the browser is loading the JavaScript files in the background

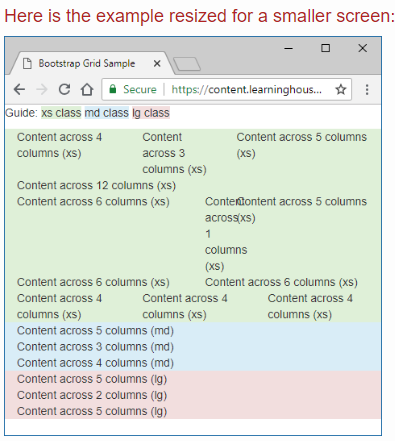
## **Responsive Design with Bootstrap’s Grid System**

Bootstrap’s grid system allows you to lay out your page in a grid that is 12 columns across. It is put together using [flexbox (Links to an external site.)Links to an external site.](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Flexible_Box_Layout/Basic_Concepts_of_Flexbox) which helps automatically size some of our components. Here are some things to note:

* As mentioned above, your content needs to be enclosed in a div with either a ***container*** (fixed-width) or ***container-fluid*** (full-width) class.
* There are CSS classes for the rows and columns of the grid:
  + ***row*** is used to group content into a row.
  + ***col*** classes are there for grouping content across multiple columns. You can specify exactly how many columns the HTML element should span using a notation like **col-4**, which tells Bootstrap that the content should span four grid columns.
  + ***sm***, ***md***, and ***lg*** refer to the devices that they are targeting. If a smaller device style is applied and a larger device style is not present, the smaller device style will apply to the element regardless of the viewing device.
  + In previous versions, **xs** referenced extra small screens. This has been replaced in Bootstrap 4 with just **col** by itself with no size indicator appended.
  + As of Bootstrap 4, If you don't specify a column width for a column (***col*** vs ***col-12***) it will automatically resize the columns in a single row to fit into the 12 grid columns evenly.  For example, three ***col*** columns in a ***row*** will automatically resize to be the equivalent of ***col-4***. Similarly, two ***col*** columns in a ***row*** will automatically resize to be the equivalent of ***col-6*** each.

To better understand the grid, open up [**bootstrap-gridsample.html** (Links to an external site.)Links to an external site.](https://content.learninghouse.com/clients/tsg/courses/itwd2-1/resourceFiles/CSS-Example-Files/bootstrap-gridsample.html). This file displays multiple extra small formatted rows so that you can see how to spread content across 12 columns. There are also medium and large formatted rows.





## **Looking at CSS Media Queries**

Behind the grid system are CSS media queries. These are ways in CSS to apply styles based on certain conditions, including:

* Width and height of the viewport and device
* Orientation – landscape or portrait mode
* Types – including screen, speech (for screen readers), print (for styling output for a printer), and all

Media queries are noted by @media and follow the following format:

@media type and (property:value) {

selector {

property:value;

property:value;

}

selector2{

property:value;

property:value;

}

}

## **Example Media Queries in Bootstrap**

| **Media Query** | **Meaning** |
| --- | --- |
| @media print | For output sent to a printer |
| @media (min-width: 768px) | For a viewport at least 768px wide |
| @media screen and (max-width: 767px) | For viewing with a screen and the max viewing width is 767px |
| @media (max-device-width: 480px) and (orientation: landscape) | For devices that have a max device width of 480px and are displaying the site in landscape mode |
| @media all and (transform-3d), (-webkit-transform-3d) | For all types that support CSS3’s 3D transformations |

You can gain even more control by using Bootstrap's explicit typography styles. These include:

* [Classes for text alignment, wrapping, and effects (Links to an external site.)Links to an external site.](https://getbootstrap.com/docs/4.1/utilities/text/)
* Stand-out display headings:   
    <h1 class="display-1">Important Heading</h1>
* Blockquotes
* Emphasized paragraphs:   
    <p class="lead">This paragraph is more important than the others!</p>

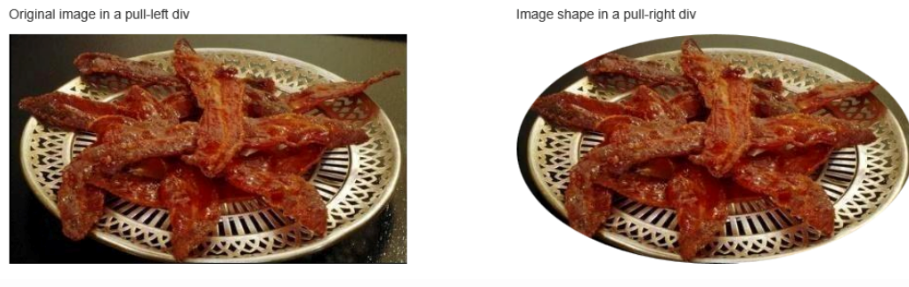
Bootstrap also supports responsive tables. Wrap your <table> in a parent bearing the .table-responsive class and your table will become scrollable if it out-grows the space available in the view.

Forms include inputs, labels, and buttons, but there's more to style than just those. Bootstrap scales form content based on the view width. In addition to HTML controls, Bootstrap offers styles for the following form features:

* Control state:
  + Focused
  + Disabled
  + Read-only
  + Valid or Invalid
* Explicit control sizes
* Help text (instructions or hints)

Image handling is easier with Bootstrap:

* Make images responsive by adding the .img-fluid class. With it, images scale nicely within their parent.
* Add a gently rounded border with .img-thumbnail.
* Add a more pronounced shape effect with the .rounded or .rounded-circle classes.



## **Utilities**

Say you want to tweak some HTML that doesn't quite fit into typography, tables, forms, or images. That's no problem. Mix and match Bootstraps utility classes to get exactly the effect you're after. Utilities include:

* [Borders (Links to an external site.)Links to an external site.](https://getbootstrap.com/docs/4.1/utilities/borders/)
* [Colors (Links to an external site.)Links to an external site.](https://getbootstrap.com/docs/4.1/utilities/colors/)
* Flex and float layouts
* [Position (Links to an external site.)Links to an external site.](https://getbootstrap.com/docs/4.1/utilities/position/)
* [Sizes (Links to an external site.)Links to an external site.](https://getbootstrap.com/docs/4.1/utilities/sizing/)
* And many more

Many of Bootstrap's advanced features are simply extended and combined utility classes!

**JavaScript**

## **Value Types**

* Number
* Boolean
* String
* Undefined
* Null

Whole numbers are known as **integers**, and fractional numbers with decimals are known as **floats**.

12 //integer number

10.90 //float number

 These operations are executed by **operators**. The values used with these operators are considered **operands**. Here is a list of the mathematical operators available in JavaScript.

* Add (+)
* Subtract (-)
* Multiply (\*)
* Divide (/)
* Modulo or Remainder (%)

**Modulo** is similar to division in that it performs a division operation, but it returns only the remainder of the operation. For example, 7 % 2 (pronounced "7 mod 2") returns 1 because 7 / 2 = 3 with a remainder of 1. This operation is most useful for determining whether or not a given value is odd or even: even numbers will always have a % 2 value of 0.

**order of operations** is the phrase Please Excuse My Dear Aunt Sally or PEMDAS. It stands for [Parentheses, Exponents, Multiplication and Division, Addition and Subtraction (Links to an external site.)](https://www.mathsisfun.com/operation-order-pemdas.html)Anything inside parentheses will be evaluated first, then exponents, then multiplication and division, then addition and subtraction. In the expression above, if you want 10 + 2 to be calculated first, you could just put that operation in parentheses.

(10 + 2) \* 3

Doing this would give you an answer of 36.

Boolean values have only two possible values, **true** or **false**

"This is a valid string value."

‘This is a valid string value.’

You can add line breaks to string values using the \n character; the "n" after the backslash tells the computer that you want to start a new line. Using a backslash inside a String value indicates that the character that comes next has special meaning. This is known as **escaping** the character.

console.log("This text will be on line 1. \n This text will be on line 2.");

**Output:**

This text will be on line 1.  
This text will be on line 2.

String values can use the + operator to combine multiple String values into one String value. This is known as **concatenation**.

"Ja" + "va" + "Scri" + "pt"

Undefined means that you have created a variable but not assigned it a value just yet. For null, you've created a variable and its **value**is null.

### Mathematical Operators

### Comparison Operators

* Less than (<)
* Greater than (>)
* Less than or equal to (<=)
* Greater than or equal to (>=)
* Equal to (==)
* Strictly equal to (===)
* Not equal to (!=)
* Not strictly equal to (!===)
* 3 == 3 // true
* 3 == "3" // true
* 3 === 3 //true
* 3 === "3" //false
* 1 > 2 //false

 The == operator checks the **values** of the operands

The === operator checks the **values and types**

### Logical Operators

* And (&&)
* Or (||)
* Not (!)
* The And operator (&&) is a binary operator that will take two Boolean values (true or false). The only way to return a "true" is if **both** values are true.
* true && true //true
* false && true //false
* The Or operator (||) will produce a true value if either of the values is true.
* true || false //true
* false || false //false
* The Not operator (!) will flip a boolean value that is true to false and flip a false value to true.
* !true   //false
* !true //false
* !false //true

# **Intro to the Chrome Developer Tools**

# Clear the developer console by right-clicking in the Console window and clicking **Clear console** in the context menu. You could also clear the console with the following command:

console.clear();

var foo = 20 / 5;

1. Press Enter. The console will return a value of "undefined."
2. console.log(foo);

The output should be:

4

Now let's assign a new value to the foo variable and console.log(foo); to log its newly assigned value.

foo = false;

Press Enter. The output is: false

console.log(foo);

false

Expressions and statements can be compared to sentence fragments and sentences. So far, you have only been working with values. A value by itself can be considered an expression. Multiple values (operands) with operators can also be seen as expressions. **Expressions** are pieces of code that return values - which may be made up of a single value or a combination of operands and operators, functions, and variables.  Expressions can be as simple as a single value or as complex as doing math with values that are returned from function calls.

**Statements**are pieces of code that perform actions or produce behaviors - such as declaring a variable, changing a variable's value, or displaying an alert message.

We have seen how to write expressions that use values and operators to perform operations on those values to create a new value. But how do we save these values? You can assign the results of your expressions to a **variable**, a placeholder used for storing a value in memory.  Saving values in variables is known as saving the **state** of a program. When variable values in a program change,we say that the program’s state has changed.

### Declaring variables

var foo;

var bar;

### Assigning values to variables

We assign a value to a variable using the single equals (=) operator. Unlike the double equals (==) and triple equals (===) operators which are used for comparisons, the single equals operator is used for assigning values to variables.

var foo = 10 \* 2;   //foo is now set to the value "20"

var bar = true;   //bar is now set to the value "true"

Above, we assign expressions to variables, creating entire statements. Variables are called **variables** because their values can vary – they can change from one statement to the next, depending on the needs of the program. If you want to change a variable’s value after it has been declared, you change the assignment without using the var keyword. The var keyword is only needed for declaring a new variable. Once a variable is declared, it doesn't need to be declared again.

var foo = 10;

foo = 20;

* Identifiers can start with a letter, an underscore (\_), or a dollar sign ($).
* Other characters that can be used after the first character include additional letters, numbers, underscores, or dollar signs.
* Hyphens (-) cannot be used in variable names; they are specifically reserved as the subtraction operator.
* JavaScript is case-sensitive: mathFacts and MathFacts are two different names.
* While you can use underscores to represent spaces in variable names, you will also see camelCasing used. This convention states that the first character is lowercase and then the first character of each noun in the name after the first one is capitalized. Examples of this include mathFacts, dayName, documentObject, and totalScore.

JavaScript has two types of operators, **unary** and **binary**. Unary operators perform operations on a single value. Binary operators perform operations on two values, like subtracting one number from another number. Unary operators are special in that they work on variables, not values.

For example, the unary increment operator (++) will increase a value by 1.  The unary decrement operator (--) will decrease a value by 1.

We have covered expressions and statements. Expressions can be thought of as sentence fragments, and statements can be thought of as full sentences. We also covered what a variable is, how to declare them, and how to assign a value to them by using a single equals operator (=) followed by an expression.

Control flow is the order in which a program's lines of code are executed. Generally, a computer will execute JavaScript code one line at a time, from top to bottom. But there are certain control flow mechanisms we can use to interrupt this flow. These control flow mechanisms are core concepts of programming, and they don’t just apply to JavaScript. Other languages use these same concepts we are about to discuss, but they are written differently in different languages. Here, we will look at how they are written in JavaScript.

An if statement will execute code only if a certain condition is met.

if (true) {

  //any code placed here will be executed

}

if (false) {

  //any code placed here will be skipped over

}

var direction = prompt("Should we STOP, SLOW, or GO?");

if (direction == "STOP") {

    console.log("STOP!");

    alert("STOP!");

} else {

    console.log("GO!");

    alert("GO!");

}

var direction = prompt("Should we STOP, SLOW, or GO?");

if (direction == "STOP") {

    console.log("STOP!");

    alert("STOP!");

} else if (direction == "SLOW") {

    console.log("SLOW DOWN!");

    alert("SLOW DOWN!");

} else {

    console.log("GO!");

    alert("GO!");

}

In the above code:

* If (if) the user enters STOP, they will get notifications to STOP.
* If (if) the user enters SLOW, then they will get notifications to SLOW DOWN.
* Otherwise (else), the user will get notifications to GO.

## **Switch**

A switch statement is an alternative to writing a whole bunch of if…else statements. Let's look at how the above example is written as a switch statement

switch (direction) {

    case "STOP":

        console.log("STOP!");

        alert("STOP!");

        break;

    case "SLOW":

        console.log("SLOW DOWN!");

        alert("SLOW DOWN!");

        break;

    default:

        console.log("GO!");

        alert("GO!");

}

A switch statement is created with the switch keyword, followed by an expression in parentheses. The switch expression gets evaluated once, and then is matched to a corresponding case. If a matching case is found, that case's block of code is executed. If no matching case is found, the default block of code is executed. The breakkeyword is used to break out of the switch after a case's block of code is executed. If a case 's block of code is executed and there is no break keyword at the end of its code block, all of the following cases' code blocks will be executed. This is known as "falling through" the switch.

## **Loops**

Loops let you run a block of code multiple times, until a test condition is met. Loops are most commonly run against a data set (arrays or objects). The most common loops you will use in JavaScript are the while loop and the for loop.

### The For Loop

A for loop allows us to run through a process a set number of times, stopping when a counter meets a certain condition.

for (var loopCounter = 0; loopCounter <= 4; loopCounter++) {

    console.log(loopCounter);

}

The above for loop will output the numbers 0-4 to the developer console. The output should be:

0  
1  
2  
3  
4

* The first thing we have in the for declaration is var loopCounter = 0;. This creates a variable named loopCounter and sets its value to 0. This is typically the **counter declaration and initializer.**
* The next part of the for declaration, loopCounter <= 4; simply states that the block of code that follows the condition will execute as long as loopCounter's value is less than or equal to 4. This is the **test condition**.
* The third part of the for declaration, loopCounter++, adds 1 to the current value of loopCounter. Writing a ++after a variable is a short-hand way of writing loopCounter = loopCounter + 1. Terms used for this may include **loop increment** or **loop step**.  Note: while some use the term **increment**, you can also start at a higher number and work lower using the unary decrement (--) operator to decrement the loop counter.
* The first time the loop runs, the value of loopCounter is 0.
* The loop checks to see if the value of loopCounter is less than or equal to 4. Since the value of loopCounter is currently 0, the condition evaluates to true, and the block of code is executed.
* The block of code prints the value of loopCounter to the developer console and then adds 1 to the value of loopCounter, making the value of loopCounter = 1.
* The loop runs again, checking if loopCounter is less than or equal to 4. Since the value of loopCounter is now 1, the condition evaluates to true, and the block of code is executed, printing the current value of loopCounter to the developer console, then adding 1 to the current value of loopCounter. This continues until loopCounter = 5.
* Once the value of loopCounter is greater than 4, the condition loopCounter <= 4; evaluates to false. This stops the for loop from executing the block of code. The loop is finished and will not execute again.

The final result is the numbers 0-4 being printed to the developer console.

### The While Loop

A while loop runs as long as its condition is met. We could perform the same tasks as above with a while loop, as seen in the following code:

var loopCounter = 0;

**while** (loopCounter <= 4) {

console.log(loopCounter);

loopCounter++

}

The above while loop will output the numbers 0-4 to the developer console. The output should be:

0  
1  
2  
3  
4

* We start with a declaration var loopCounter = 0;. This creates a variable named loopCounter and sets its value to 0 **before** entering the loop.
* The loop's condition is in the parentheses after while.
* The loop's counter is modified within the while loop.
* There is a variant of the while loop called a do... while loop. The while loop may never execute, depending on the condition; however, the do... while loop will execute at least once and then continue to repeat as long as the condition is met.
* Suppose we want to get a number from a user between 1 and 10. We need to ask the user for a value at least once, so this would be a good opportunity to look at the do... while loop. The following code will continue to prompt a user for a number between 1 and 10 until the condition is met.
* **do** {
* userInput = prompt("Give me a number between 1 and 10.");
* **if** (userInput <= 0 || userInput > 10) {
* alert("The number must be between 1 and 10");
* }
* } **while** (userInput > 10 || userInput <= 0);

If you enter invalid input such as -1 or 11, you will see an alert box that states to enter a value between 1 and 10.  However, if the input is between 0 and 10 - such as 5, then the loop will exit.  To verify that the loop has run, you can execute the following code and see that userInput is the valid number that you entered between 0 and 10.

console.log(userInput);

The output should be the valid number you entered. In our case, we used 5:

5

var sentence **=** "I love learning software development";

**for** (var i**=**0; i **<** sentence**.**length; i**++**) {

**if**(sentence[i] **!==** " "){

**console.**log(sentence[i]);

}

}

In the for loop, we use the .length property in the loop's condition. Every string in JavaScript has a .lengthproperty, which refers to the number of characters (including spaces) in the string. Here we compare the number of characters in our sentence string (the string's length) to the current value of i. This assures that the loop will only execute as many times as there are characters in our sentence string, preventing an infinite loop.

Then in the for loop's block of code, we use an if statement to check if the current character is not an empty space. If the current character is not an empty space, we print it to the developer console with console.log(sentence[i]);.

Using sentence[i] states that we want the character at a certain index of the string. If we used sentence[0], we would get the first character in the string. If we used sentence[1] we would get the second character in the string, and so on. Since we use sentence[i], we are looking up the character at the index of the current value of i.

In this exercise, we used a for loop and an if statement inside of the for loop's block of code to print all of the letters of a sentence, ignoring the spaces between words.

# **CODE PRACTICE: if Statements Didn’t get**

* In Chrome's Developer Tools Console, solve the following problem:

You are given two numeric variables:

**var** n = 25;

**var** result = 0;

Their values may change when you submit.

Using if and else, make decisions based on the values of n and result with the following rules:

1. When n is even, set the value of result to result's current value plus 10.
2. When n is odd, set the value of result to result's current value minus the value of n.
3. Do not declare new variables.
4. Be sure your solution works for all values of n and result.

If ( n % 2 === 0 ) {

result = result + 10;

console.log(result);

} else {

result = result – n;

console.log(result);

}

(something like this?? Got a Unexpected token { in the console)

# **CODE PRACTICE: for Loops Didn’t Get**

* In Chrome's Developer Tools Console, solve the following problem:

You are given one numeric variable:

**var** n = 25;

You can use 25 as a starting point, but the following steps should work regardless of the value of n.

1. Declare a new variable named "sum" and initialize it to 0. Be sure to use the var keyword.
2. Use a for loop to add every positive integer less than n to sum.
3. Come up with a solution that works for all values of n.

var n = 25;

var sum = 0;

for (var sum = 0; sum <=n; sum++) {

if (sum % 2 = 0){

sum = sum +

console.log(sum);

}

for (var sum = 0; sum % 2<=n; sum++) {

if (sum % 2 = 0

console.log(sum);

}

for (var loopCounter = 0; loopCounter <= 4; loopCounter++) {

    console.log(loopCounter);

}

var sentence **=** "I love learning software development";

**for** (var i**=**0; i **<** sentence**.**length; i**++**) {

**if**(sentence[i] **!==** " "){

**console.**log(sentence[i]);

}

}

var n = 25;

var sum = 0;

for (var i=0; i<n; i++) {

if (i % 2 = 0){

sum = i + sum;}

console.log (i);

console.log (sum);

}

Something like this

## **Functions**

When you use a function, you **call** the function.

console.log(foo);

We can say that foo is an **argument** – a value that gets passed into a function. console.log() takes in an input and writes it out the console. However, it does not return anything because it doesn't create a new value.

### Console functions

console.log()

* console.debug() and console.log() output are similar – typically black text, white background, and no icon.
* console.info() output may start with an icon that has a white lowercase i in a blue circle and may have a light blue background.
* console.warn() output may have an exclamation mark icon and possibly a light yellow background.
* console.error() output may have a red x icon and possibly red text with a light pink background.

If your console gets really cluttered, you can use console.clear() to clear it out.

### Pop-up Functions

* classalert()
* confirm()
* prompt()
* alert()

alert("This is a message.");

ok

confirm("Do you like cake?");

yes or no

prompt('What is your favorite band?');

text field and ok

// Check to see if the user likes cake

var likesCake = confirm("Do you like cake?");

// If the user likes cake, find out what their favorite cake is

if (likesCake) {

var favoriteCake = prompt("What is your favorite cake?");

// Alert them that their favorite cake is whatever they inputted

// from the prompt() function

alert("Your favorite cake is: " + favoriteCake);

} else {

// If the user doesn't like cake, show an alert that the cake is a lie

alert("The cake is a lie anyhow.");

}

### Additional Functions

the Math object has a lot of math-related functions – which are helpful if you find yourself doing mathematical calculations such as square root, exponents, and rounding.

Strings have a variety of functions to make it easier to break strings apart, put strings together, and even see if a character or a string is within another string. substring()

The browser and document objects have functions to make it easy to work with web pages and the browsers that display them.

## **Creating a Function**

we can also create our own functions

function sayHello() {

alert('hi');

}

function sayHello(myName) {

alert('Hello, ' + myName);

}

### Arguments and Parameters

The values that you pass into a function are called **arguments**.

When you call a function, you pass arguments to a function.

The **parameters** are the variables in parentheses when the function is declared.

When you declare a function, you declare the parameters of that function.

Think of parameters as the variables, and arguments are the passed values for those variables.

## **Generating Random Numbers**

| **Code** | **Description** |
| --- | --- |
| Math.random() | Generates a random number between 0 and 1 |

Math.**random**() \* 6;

**for** (**var** i = 0; i < 100; i++) {

console.**log**(Math.**random**() \* 6);

}

he values that you see range from 0 to not‑quite 6. So how do we get these results to be **exactly** 1 – 6?

You likely remember how to round numbers from elementary school math, but the JavaScript Math object has more precise options for rounding float values to integers – ceil(), floor(), and round().

| **Code** | **Value** | **Description** |
| --- | --- | --- |
| Math.ceil(0.9394775930471786) | 1 | ceil is short for **ceiling**. Much like a ceiling is the top of a room, Math.ceil() rounds up to the nearest whole number. |
| Math.floor(0.9394775930471786) | 0 | Much like a floor is the bottom of a room, Math.floor() rounds down to the nearest whole number. |
| Math.round(0.9394775930471786) | 1 | Math.round() rounds up or down to the nearest whole number.   Testing it out, “0.4” with 15 9's after it rounds down to 0.   Math.round(0.4999999999999999) = 0   But if there are more than 15 9's, it rounds up to 1.   Math.round(0.49999999999999999) = 1 |

We want to generate a range of 6 integer values, so we will start by using floor to round each result down to the next integer.

Math.**floor**(Math.**random**() \* 6);

#### **Why not round up (ceil) instead of down (floor)?**

Remember that Math.random can produce a value that is exactly 0. If we simply round up the values Math.random returns, we end up with a possible range of 0 through 6, instead of 1 through 6.

**for** (**var** i = 0; i < 100; i++) {

console.**log**(Math.**floor**(Math.**random**() \* 6));

}

You should see the results as integers varying between 0 and 5. Now all we have to do is add 1 to the results to get values that range from 1 to 6.

**for** (**var** i = 0; i < 100; i++) {

console.**log**(Math.**floor**(Math.**random**() \* 6) + 1);

}

Let's look at the formula in the steps that JavaScript uses to calculate the final result, starting from the inside out.

1. Math.random() : JavaScript generates a random number between 0 and not‑quite 1.
2. Math.random() \* 6: JavaScript multiplies the random number by 6, resulting in a number between 0 and not‑quite 6.
3. Math.floor(Math.random() \* 6) : JavaScript rounds down the current value to the nearest integer, resulting in a value that is 0, 1, 2, 3, 4, or 5.
4. Math.floor(Math.random() \* 6) + 1 : JavaScript adds 1 to the current value, resulting in an integer between 1 and 6, inclusive, which is exactly what we need.

If you run this loop, you should see integers ranging from 1 to 6 in the results.

For dice with different num of sides

**function** rollDice(numSides) {

**return** Math.**floor**(Math.**random**() \* numSides) + 1;

}

# **CODE PRACTICE: Defining Functions Didn’t get**

* In Chrome's Developer Tools Console, solve the following problem:

You are given the following function:

**function** addTwoNumbers(firstNumber, secondNumber){

// your code goes here

}

var newNum

function addTwoNumbers(firstNumber, secondNumber){

*// your code goes here*

firstNumber **+** secondNumber **===** newNum;

**console.**log(newNum)

}

var randomNum **=** 3456

var firstNumber

var secondNumber

**for** (var i**=**0; i**<** randomNun**.**length; i**++**){

**console.**log(randomNum);

randomNum[0] **=** firstNumber;

randomNum[1] **=** secondNumber;

**console.**log(firstNumber);

**console.**log(secondNumber);

}

## **Arrays**

Here is a sample array:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Index** | 0 | 1 | 2 | 3 | 4 | 5 |
| **Element** | "Aneki" | "Quell" | "Clarity" | "Sleepy" | "Roghar" | "DM Crimson" |

var friends = ["Aneki", "Quell", "Clarity", "Sleepy", "Roghar", "DM Crimson"];

friends.length;

friends[0];

"Aneki"

// We need 2 teams - team1 and team2

var team1 = new Array();

var team2 = new Array();

for (var i = 0; i < friends.length; i++) {

    if (i % 2 == 0) {

        // if the index is even, add the person to team 1

    } else {

        // otherwise add them to team 2

    }

}

for (var i = 0; i < friends.length; i++) {

    if (i % 2 == 0) {

        // if the index is even, add the person to team 1

        team1.push(friends[i]);

    } else {

        // otherwise add them to team 2

        team2.push(friends[i]);

    }

}

We can look at the teams via console.log():

console.log(team1);

The output for team 1 should be:

["Aneki", "Clarity", "Roghar"]

Similarly, we can look at team 2:

console.log(team2);

The output for team 2 should be:

["Quell", "Sleepy", "DM Crimson"]

push() and pop() work with the end of an array; two more functions that behave similarly but at the beginning of the array are unshift() and shift(). Thinking of that stack of pancakes, now we want to add pancakes to the bottom of the stack by unshifting them on. We would take pancakes from the bottom by shifting them off.

for (var i = 0; i < friends.length; i++) {

    if (i % 2 == 0) {

        // if the index is even, add the person to team 1

        team1.unshift(friends[i]);

    } else {

        // otherwise add them to team 2

        team2.unshift(friends[i]);

    }

}

Let's look at our teams as they were built with unshift().

console.log(team1);

The output for team 1 should be:

["Roghar", "Clarity", "Aneki"]

Check team 2 as well:

console.log(team2);

Team 2 should be:

["DM Crimson", "Sleepy", "Quell"]

Note that the resulting arrays are backwards from the original arrays that use push(). When we used push(), we added content to the array by stacking the items on top of each other, so items are added to the array from bottom to top (or left to right).

With unshift(), we add each item from the original array to the first position in the new array. If there are already items in the array, the existing items are moved up to later positions in the array to make room for the new first item.

### Populating arrays with a for loop and indexes

We could do the same exercise using the length property as the index to add elements to the end of an array. That would look like this:

var team1 = new Array();

var team2 = new Array();

for (var i = 0; i < friends.length; i++) {

    if (i % 2 == 0) {

        // if the index is even, add the person to team 1

        team1[team1.length] = friends[i];

    } else {

        // otherwise add them to team 2

        team2[team2.length] = friends[i];

    }

}

console.log(team1);

Team 1 should be:

["Aneki", "Clarity", "Roghar"]

Checking on team 2:

console.log(team2);

Team 2 should be:

["Quell", "Sleepy", "DM Crimson"]

Here is another example of working with arrays – this time, with fruits.

var fruits = ['kiwi', 'rambutan', 'mango', 'tomato'];

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Index** | 0 | 1 | 2 | 3 |
| **Element** | "kiwi" | "rambutan" | "mango" | "tomato" |

Now, we pop() to take off the last element:

fruits.pop(); // kiwi, rambutan, mango

fruits should be:

|  |  |  |  |
| --- | --- | --- | --- |
| **Index** | 0 | 1 | 2 |
| **Element** | "kiwi" | "rambutan" | "mango" |

Now, we push() "gooseberry" to the end of the array:

fruits.push('gooseberry'); // kiwi, rambutan, mango, gooseberry

fruits should be:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Index** | 0 |  | 1 | 2 | 3 |
| **Element** | "kiwi" |  | "rambutan" | "mango" | "gooseberry" |

By calling shift(), we take away the first element of the array:

fruits.shift(); // rambutan, mango, gooseberry

fruits should be:

|  |  |  |  |
| --- | --- | --- | --- |
| **Index** | 0 | 1 | 2 |
| **Element** | "rambutan" | "mango" | "gooseberry" |

With unshift() "banana", we set the new value at position 0:

fruits.unshift("banana"); // banana, rambutan, mango, gooseberry

fruits should be:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Index** | 0 | 1 | 2 | 3 |
| **Element** | "banana" | "rambutan" | "mango" | "gooseberry" |

### Slice and Splice

Two other useful functions when working with arrays are slice() and splice().

* slice() will get a number of elements, starting at the starting position. It returns the elements but does not change the array.
* splice() on the other hand will return the elements that are removed and change the array.
* The fruits array is currently at:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Index** | 0 | 1 | 2 | 3 |
| **Fruit** | "banana" | "rambutan" | "mango" | "gooseberry" |

* fruits.slice(0, 2);
* The slice(0,2) returns 2 elements, starting from position 0:

|  |  |  |
| --- | --- | --- |
| **Index** | 0 | 1 |
| **Element** | "banana" | "rambutan" |

console.log(fruits); shows that all four fruits are still in the array, in the same order.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Index** | 0 | 1 | 2 | 3 |
| **Fruit** | "banana" | "rambutan" | "mango" | "gooseberry" |

However, if we use splice(), then fruits will change.

fruits.splice(0, 2);

After the splice(0,2), fruits is now at:

|  |  |  |
| --- | --- | --- |
| **Index** | 0 | 1 |
| **Element** | "mango" | "gooseberry" |

Another way to add an element to the end of an array without having to create a variable to track the position is by using the length property, which adds the new value as a new item at the end of the array.

fruits[fruits.length] = "blueberry"; //mango, gooseberry, blueberry

fruits should now be:

|  |  |  |  |
| --- | --- | --- | --- |
| **Index** | 0 | 1 | 2 |
| **Element** | "mango" | "gooseberry" | "blueberry" |

If you specify a specific index in an assignment expression, the value at that index will get updated with the new value.

fruits[2] = "pineapple"; // mango, gooseberry, pineapple

fruits should be:

|  |  |  |  |
| --- | --- | --- | --- |
| **Index** | 0 | 1 | 2 |
| **Element** | "mango" | "gooseberry" | "pineapple" |

Unlike many other programming languages, JavaScript can shrink and grow its array on its own. You can put an element in a specific index and JavaScript will add undefined spots for empty slots between the filled slots.

Suppose we have to have dragonfruit at the position of index 5.

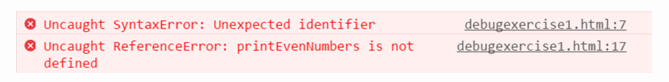
fruits[5] = "dragonfruit"; // mango, gooseberry, pineapple,..., dragonfruit

fruits now looks like this:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Index** | 0 | 1 | 2 | 3 | 4 | 5 |
| **Element** | "mango" | "gooseberry" | "pineapple" | undefined | undefined | "dragonfruit" |

## **Debugger**

debugexercise1.html



Click on the debugexercise1.html:7.

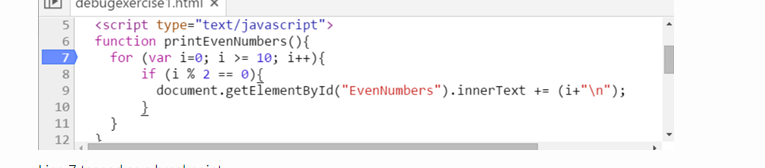
Chrome opened the **Sources** tab and highlighted line 7.

The error message tells us that this problem is a SyntaxError. **Syntax errors** are problems where we are not using the JavaScript syntax properly



change the int to a var,

click on the line number for line 7. This will set a **breakpoint**



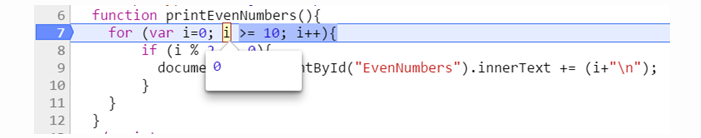
Refresh the page, and click on the **Print Even Numbers** button

debugger is paused, like this:

Paused in the debugger, play button, step over button - this appears when Chrome's tools hit a breakpoint and program execution is paused.

lick the **Step Over** button (Step over button)

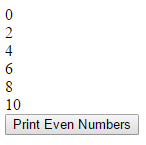
You should see the test condition for the for loop highlighted. If you hover over any instance of i in the test condition on line 7, you should see a popup message that the variable has the value of 0:



Another way to see the value of i is to look at the **Scope** pane of the debugging tools. You will see i in the **Local**scope:

Scope tab, Local variables show i set to 0  0 will never be >= 10,

change the >=to <=. Save the change, refresh the page

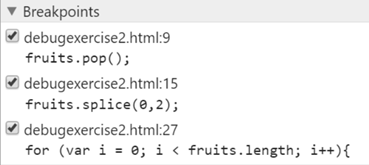


Now add a breakpoint at line 8. Click the **Print Even Numbers** button The Chrome debugging tools can also show the value of variables in line with the code, like this:

Breakpoint paused on if (i % 2 == 0 ){; i=0 appears on the line above

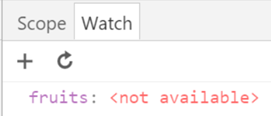
**debugexercise2.html** Set breakpoints on lines 9, 15, and 27.

the right of the Source window, you will find a collapsible pane for **Breakpoints**

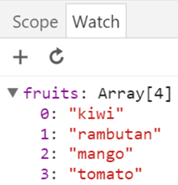
 If you click on any of those snippets of code, the debugging tools will take you to that line in the source code.

****

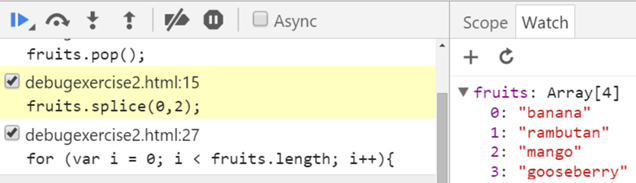
You should also see a **Watch** pane above the **Scope** pane. Expand that pane if necessary and then click the + to add a watch for fruits. This will allow us to look at the fruits variable by itself without being distracted by any other variables that may be in scope. Since we haven't started the program yet, you should see that fruits is not available:



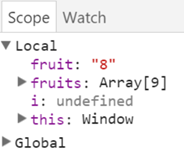
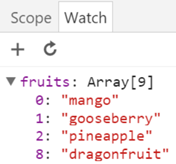
Click on the **Fun With Fruits** button to start the workflow we want to debug. When the program stops at the first breakpoint on line 9, notice that fruits has been populated:

 Click the **play** button or press F8 to advance to the next breakpoint

Also note that the active breakpoint is highlighted.



Click the **play** button or press F8 to advance to the next breakpoint. Notice that the value of fruits has changed. Compared to **Scope**, **Watch** allows us to focus on only the variables we care about:

# **CODE PRACTICE: Find Max Didn’t Get**

# **JavaScript and the Browser**

## **The DOM**

Once an HTML page is loaded in the browser, it is considered a **document object**. The **document object model (DOM)** is a platform-agnostic convention for representing markup objects written in HTML, XML, or XHTML. Each element within the HTML is considered a **node** and is hierarchical. If a larger node contains nested nodes, the larger node is considered a parent node and the directly nested nodes are considered child nodes.

Using this organization of the document object, JavaScript is able to provide us with ways to search for elements in the DOM.

## **DOM manipulation**

The document object may be written as window.document or simply document.

Here are some common ways to search with JavaScript and the DOM:

* document.getElementById("idName") – find the DOM element with the id of idName
* document.getElementsByName("elementName") – find the DOM element(s) with the name of elementName
* document.getElementsByTagName("tagName") – find the DOM element(s) of a particular HTML tag
* document.getElementsByClassName("cssClassName") – find the DOM element(s) with a class attribute of cssClassName

#### **The id attribute must be unique for the HTML elements.**

The other ways to get a collection of HTML elements are the getElementsBy… functions – get elements by their name attribute, HTML tab, or even (CSS) class attribute. These functions return a collection of HTML elements, which can be navigated with a for loop, using the collection's length property as an ending point.

### DOM Collections

The document object also has properties that give us easy access to HTML element collections:

* anchors – hyperlinks
* cookie
* embeds – embedded media
* forms
* images
* links – all hyperlinks and <area> objects with href attributes
* scripts

These collections can be accessed via document.propertyname. For example, let's say we wanted to hide all links. We could create a function to do that using the document.links collection, such as the one listed below:

function hideAllLinks() {

    var allLinks = document.links;

    for (var loopCounter = 0;

        loopCounter < allLinks.length; loopCounter++) {

        allLinks[loopCounter].style.display = "none";

    }

}

## **Events**

An **event**is an activity that can be used to trigger a call to a JavaScript function when the activity happens. There are many events that take place in the web browser. Some types of JavaScript events are:

* Document events – for the DOM objects
* Mouse events – for various mouse states
* Keyboard events – for various keyboard states – such as key presses, key releases
* Form events – for handling <form> actions and also on form controls
* Window events – for the browser window

Keyboard events and mouse events are a couple of ways to act upon input. There are events for dragging-and-dropping, scrolling with a wheel, and even touching (as opposed to mousing). You can find out more about these by doing a search in your search engine for JavaScript DOM events and the input type. For example, you could search for JavaScript's touch events with JavaScript DOM events touch.

You can assign events to objects either through HTML attributes – typically named as oneventname – or in JavaScript.

We are going to look at commonly used events: onload and onclick. As these suggest, we will be tying our HTML to JavaScript via the event attributes on the HTML elements.

### The onload Event

<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">

<head>

    <title>Simple onload Example</title>

</head>

<body onload="alert('Page is loaded');">

    <div>

        <h1>Page Loaded</h1>

        <hr />

        <div>You should be able to see this under the alert.</div>

    </div>

</body>

</html>

Here is a sample of the onloadevent calling a function we created:

<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">

<head>

    <title>Simple onload Example with Function</title>

    <script type="text/javascript">

        function greetTheUser() {

            var userName = prompt("What is your name?");

            document.getElementById("userName").innerText = userName;

        }

    </script>

</head>

<body onload="greetTheUser();">

    <div>

        <h1>Hi, <span id="userName"></span>!</h1>

    </div>

</body>

</html>

### The onclick Event

Besides loading an object, another way to trigger an event can be from a click. In a common use, with HTML <button> and <input type="button"> elements, the onclick event can be used to call a JavaScript function, typically either to modify the DOM or to verify that inputs are properly entered. The onclick event can be triggered by a mouse click, keyboard click (typically with the button selected and activated by the space bar), touch, or other input method. The "click" in this case refers to clicking on the object, regardless of input device.

Here is an example of taking input from the user and then showing a <div> with a greeting:

<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">

<head>

    <title>Simple onclick Example with Function</title>

    <script type="text/javascript">

        function greetTheUser() {

            document.getElementById("userName").innerText = document.getElementById("inputName").value;

            document.getElementById("greeting").style.display = "block";

        }

    </script>

</head>

<body>

    <div>

        Enter your name: <input type="text" id="inputName" placeholder="What's your name?" />

        <button onclick="greetTheUser();">Show Greeting</button>

    </div>

    <div id="greeting" style="display:none;">

        <h1>Hi, <span id="userName"></span>!</h1>

    </div>

</body>

</html>

Notice in the code above that we use the innerText property to set the inside of the span to what we typed in the input box. We can access what was typed in the input box using the value property. We also toggled the visibility of the <div> using the style attribute.