

INDEX

Subject: HTML

S.No.	Topic	P. No.
1.	Introduction to HTML	7
2.	Structure Of HTML	8
3.	Editors	11
4.	Comments	13
5.	Tags <ul style="list-style-type: none"> • Container & Empty tag • Block & Inline Tags 	15
6.	Root Tags	19
7.	Heading Tags	21
8.	List Tags <ul style="list-style-type: none"> • Ordered List • Unordered List • Data list 	23
9.	Formatting Tags	26
10.	Other Commonly Used Tags	28
11.	Table Tag	41
12	Form tag	51

Subject: CSS

S. No.	Topic	P. No.
1.	Introduction	65
2.	Types of Style Sheets <ul style="list-style-type: none"> • Internal Style Sheets • External Style Sheets • Inline Style Sheets 	66
3.	Types of Selectors <ul style="list-style-type: none"> • Class • Id • Element 	70
4.	Comments	76
5.	Colors	78
6.	Box Model <ul style="list-style-type: none"> • Margin • Border • Padding • Content 	81
7.	Outline	89
8.	Font <ul style="list-style-type: none"> • font-family • color • font-size • font-style • font-weight • google fonts 	91
9.	Text	103
10.	Lists	113
11.	Tables	118

	<ul style="list-style-type: none"> • Border • border-collapse • width • height • text-align • vertical-align • padding 	
12.	Dimensions <ul style="list-style-type: none"> • height width • max-height • min-height • max-width • min-width 	134
13.	Display / Visibility	139
14.	Positions <ul style="list-style-type: none"> • Relative • Absolute • Fixed • Static Z-index	145
15.	Overflow	164
16.	Float & clear	171
17	Background Properties	180
18	Transforms	185
19	! Important	187
20	Horizontal & Vertical Align	189
21	Media Queries	170

Subject: Bootstrap

S. No.	Topic	P. No.
1.	Introduction <ul style="list-style-type: none"> • What is Bootstrap? • Bootstrap History • Why Use Bootstrap? • What Does Bootstrap Include? • Downloading Bootstrap • Bootstrap CDN 	202
2.	Typography	205
3.	Colors	206
4.	Tables	207
5.	Images	209
6.	Alerts	210
7.	Buttons	211
8.	Button Group	212
9.	Button Dropdown	213
10.	Badges	214
11.	Progress bars	214
12.	Pagination	216
13.	List Groups	217
14.	Cards	218
15.	Collapse	219

16.	Navbar	220
17.	Forms	223
18.	Carousel	224
19.	Modal	226
20.	Tooltip	228
21.	Popover	230
22.	Utilities	231

VEDA INSTITUTE

HTML

1.HTML Introduction

HTML is a markup language used by the browser to manipulate text, images, and other content, in order to display it in the required format. HTML was created by Tim Berners-Lee in 1991. The first-ever version of HTML was HTML 1.0, but the first standard version was HTML 2.0, published in 1995. Present version is HTML 5 published in 2014.

1.1 What is Hypertext?

Text that is not restricted to a sequential format and that includes links to other text is called Hypertext. The links can connect online pages inside a single or different website.

1.2 What is Markup Language?

Markup Language is a language that is interpreted by the browser and it defines the elements within a document using “tags”. It is human-readable, which means that markup files use common words rather than the complicated syntax of programming languages.

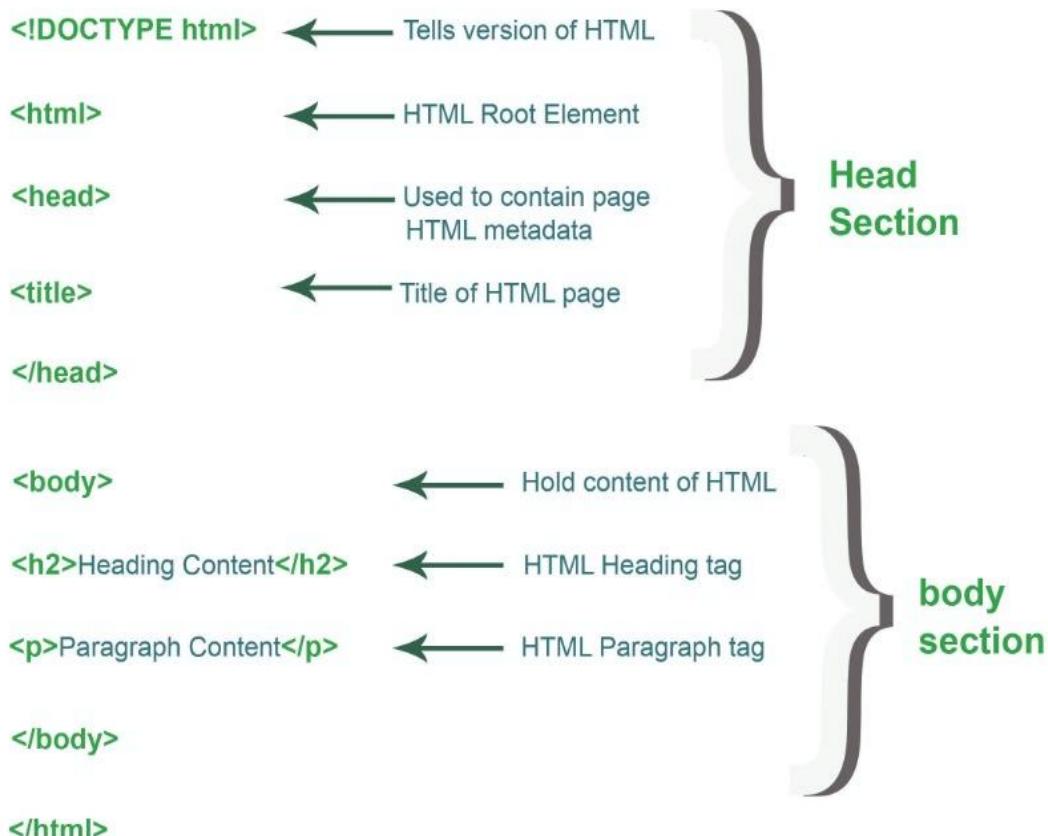
2. HTML Basic Structure:

The basic structure of an HTML page contains the essential building-block elements (i.e. doctype declaration, HTML, head, title, and body elements) upon which all web pages are created.

An HTML Document is mainly divided into two parts:

- **HEAD:** This contains the information about the HTML document. For Example, the Title of the page, version of HTML, Meta Data, etc.
- **BODY:** This contains everything you want to display on the Web Page.

HTML Page Structure

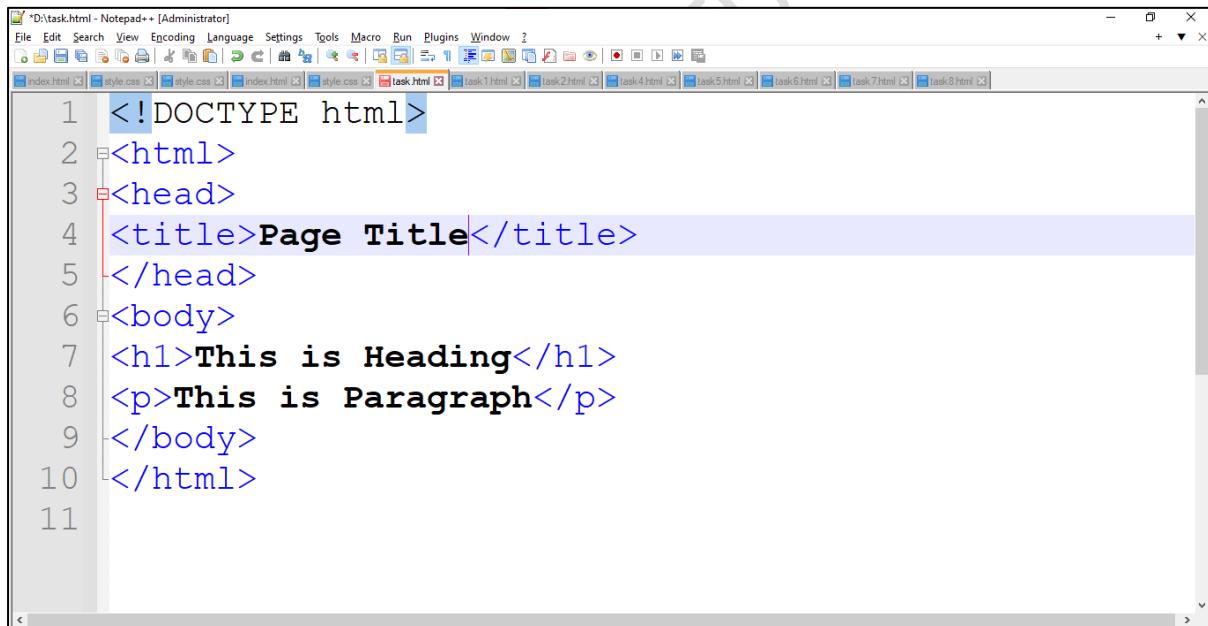


Explanation:

- The **<!DOCTYPE html>** declaration defines that this document is an HTML5 document
- The **<head>** element contains meta information about the HTML page
- The **<title>** element specifies a title for the HTML page (which is shown in the browser's title bar or in the page's tab)
- The **<html>** element is the root element of an HTML page
- The **<body>** element defines the document's body, and is a container for all the visible contents, such as headings, paragraphs, images, hyperlinks, tables, lists, etc.
- The **<h1>** element defines a large heading
- The **<p>** element defines a paragraph

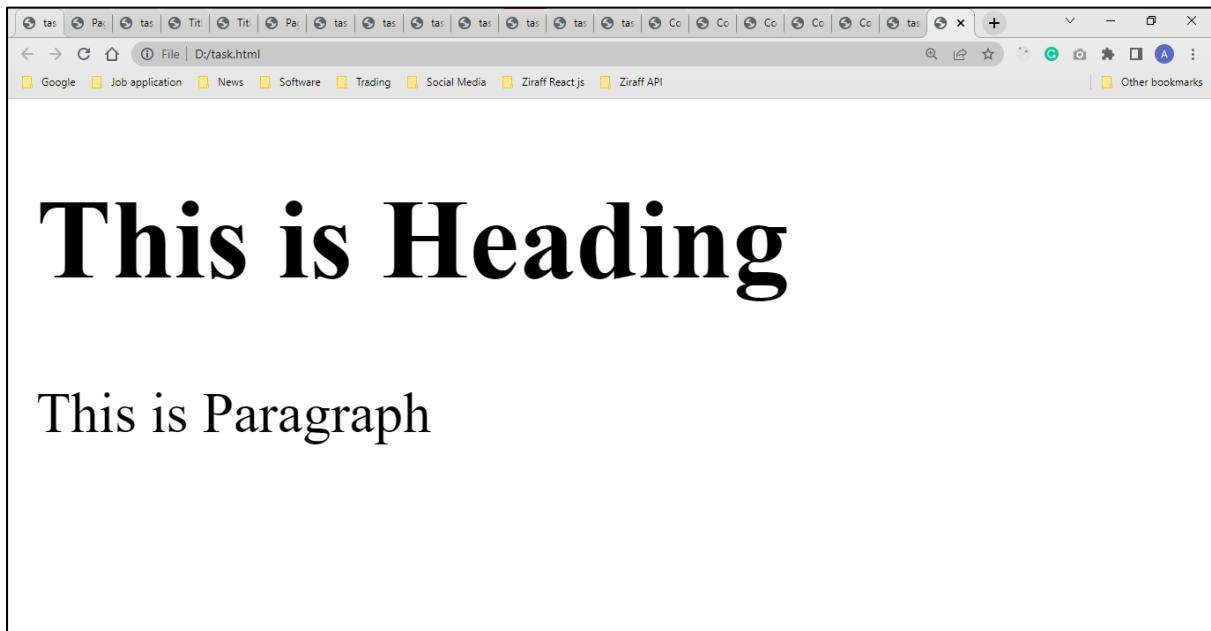
How To write Html Code:

We write code using notepad++ as shown below.



```
*D:\task.html - Notepad++ [Administrator]
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window Z
index.html style.css style.css index.html style.css task.html task1.html task2.html task4.html task5.html task6.html task7.html task8.html
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <title>Page Title</title>
5 </head>
6 <body>
7 <h1>This is Heading</h1>
8 <p>This is Paragraph</p>
9 </body>
10 </html>
11
```

Output:

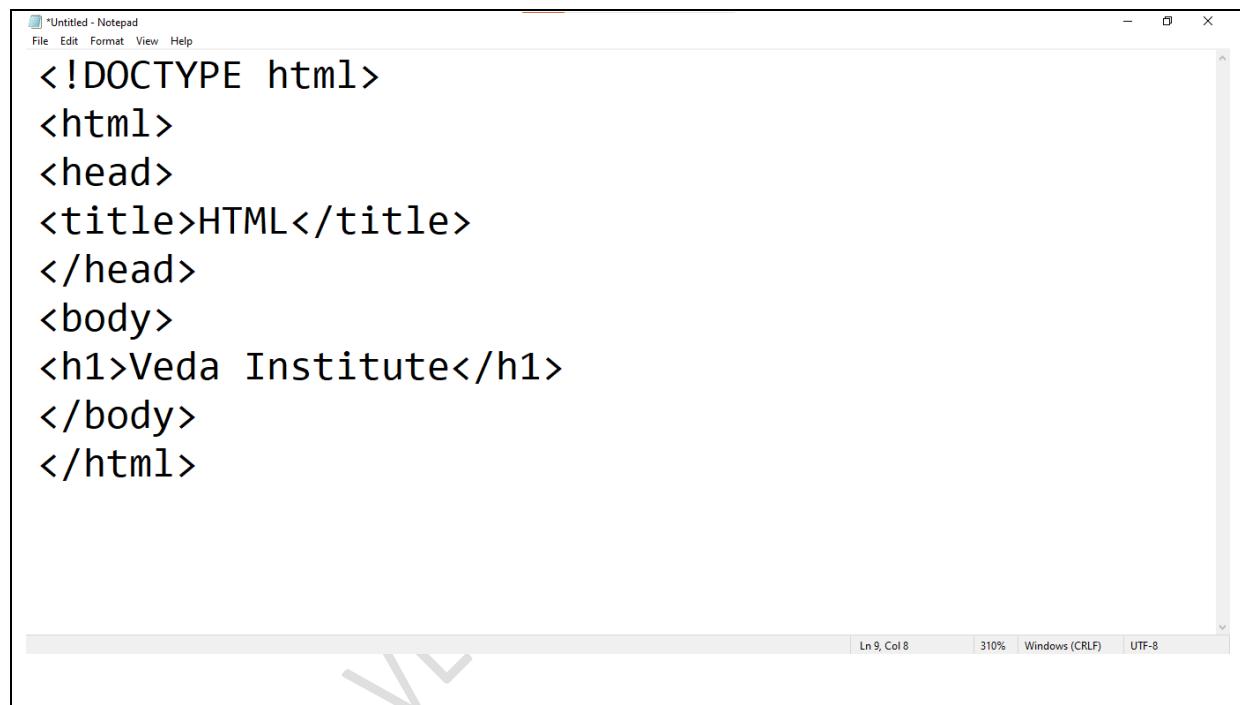


3. Editors:

Write HTML code Using Editors like

1. Notepad
2. Notepad++
3. Visual Studio Code

1. Notepad:

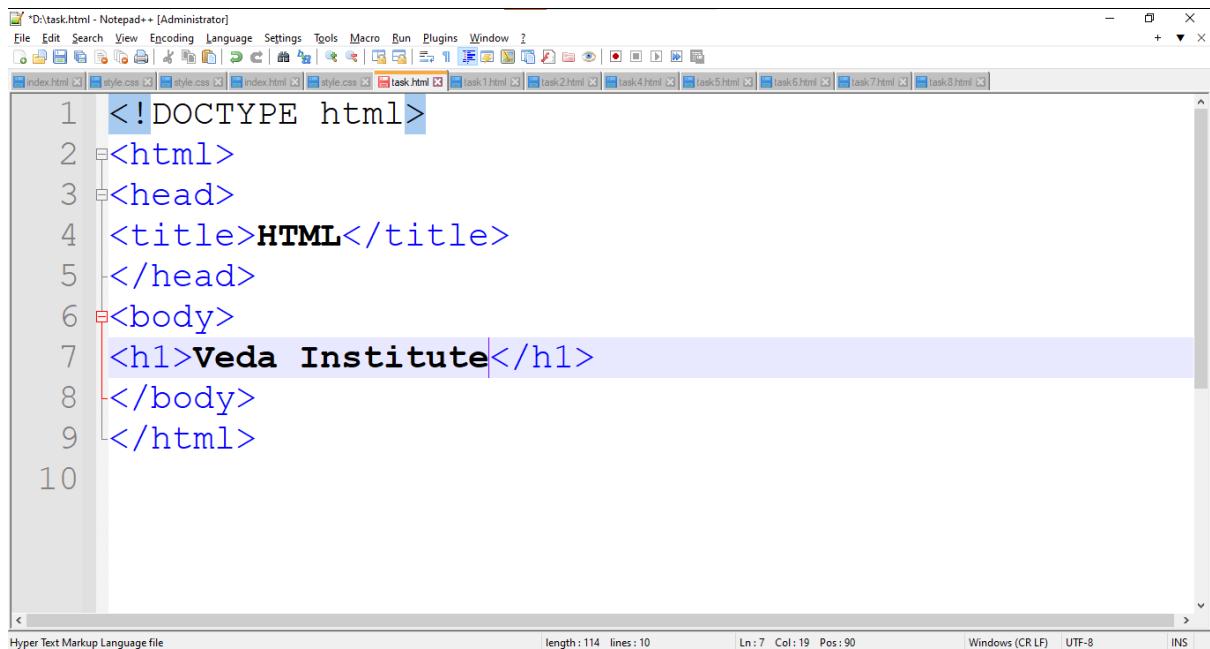


A screenshot of the Windows Notepad application window. The title bar reads "*Untitled - Notepad". The menu bar includes File, Edit, Format, View, and Help. The main text area contains the following HTML code:

```
<!DOCTYPE html>
<html>
<head>
<title>HTML</title>
</head>
<body>
<h1>Veda Institute</h1>
</body>
</html>
```

The status bar at the bottom shows "Ln 9, Col 8" and "Windows (CRLF)".

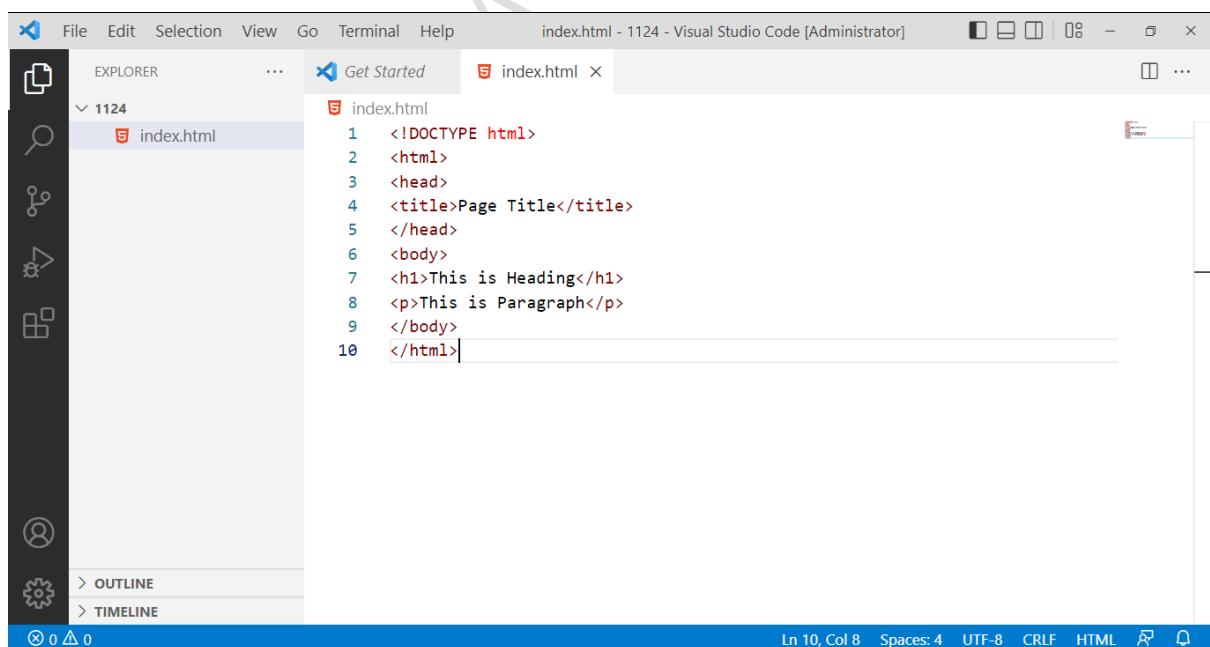
2. Notepad++:



The screenshot shows the Notepad++ application window. The title bar reads "D:\task.html - Notepad++ [Administrator]". The menu bar includes File, Edit, Search, View, Encoding, Language, Settings, Tools, Macro, Run, Plugins, Window, and Help. The toolbar has various icons for file operations. The status bar at the bottom shows "Hyper Text Markup Language file", "length:114 lines:10", "Ln: 7 Col: 19 Pos: 90", "Windows (CR LF)", "UTF-8", and "INS". The main editor area displays the following HTML code:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <title>HTML</title>
5 </head>
6 <body>
7 <h1>Veda Institute</h1>
8 </body>
9 </html>
```

3. Visual Studio Code:



The screenshot shows the Visual Studio Code interface. The top bar includes File, Edit, Selection, View, Go, Terminal, Help, and the current file name "index.html - 1124 - Visual Studio Code [Administrator]". The status bar at the bottom shows "Ln 10, Col 8 Spaces: 4 UTF-8 CRLF HTML ⚡ ⚡". The left sidebar has icons for Explorer, Search, and Timeline, with "1124" expanded to show "index.html". The main editor area displays the following HTML code:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <title>Page Title</title>
5 </head>
6 <body>
7 <h1>This is Heading</h1>
8 <p>This is Paragraph</p>
9 </body>
10 </html>
```

4.Comments:

HTML comments are not displayed in the browser, but they can help document your HTML source code. You can add comments in your HTML file using
`<! -- ... -->` tag.

syntax:

```
<! -- Write your comments here -->
```

4.1 HTML Tags vs HTML Elements vs HTML Attributes:

HTML tags:

Tags are used to mark up the start and end of an HTML element.

Ex:

```
<p> </p>
```

HTML elements:

An element in HTML represents some kind of structure or semantics and generally consists of a start tag, content, and an end tag. The following is a paragraph element:

Ex:

```
<p>This is the content of the paragraph element. </p>
```

HTML Attributes:

An attribute defines a property for an element, consists of an attribute/value pair, and appears within the element's start tag. An element's start tag may contain any number of space separated attribute/value pairs.

Ex:

```

```

In above example **src** and **alt** are attributes. **** tag is dependent on **src** attribute. Similarly, **<a>** and **<link>** tag dependent on **href** attribute.

Example:

```
<!DOCTYPE html>

<html>

<!-- This is Header section -->
<head>

    <!-- Internal CSS -->
    <style>

        body{

            text-align: center;
            background-color: #f0f8ff;
            font-size: 30px;
            color: red;
        }

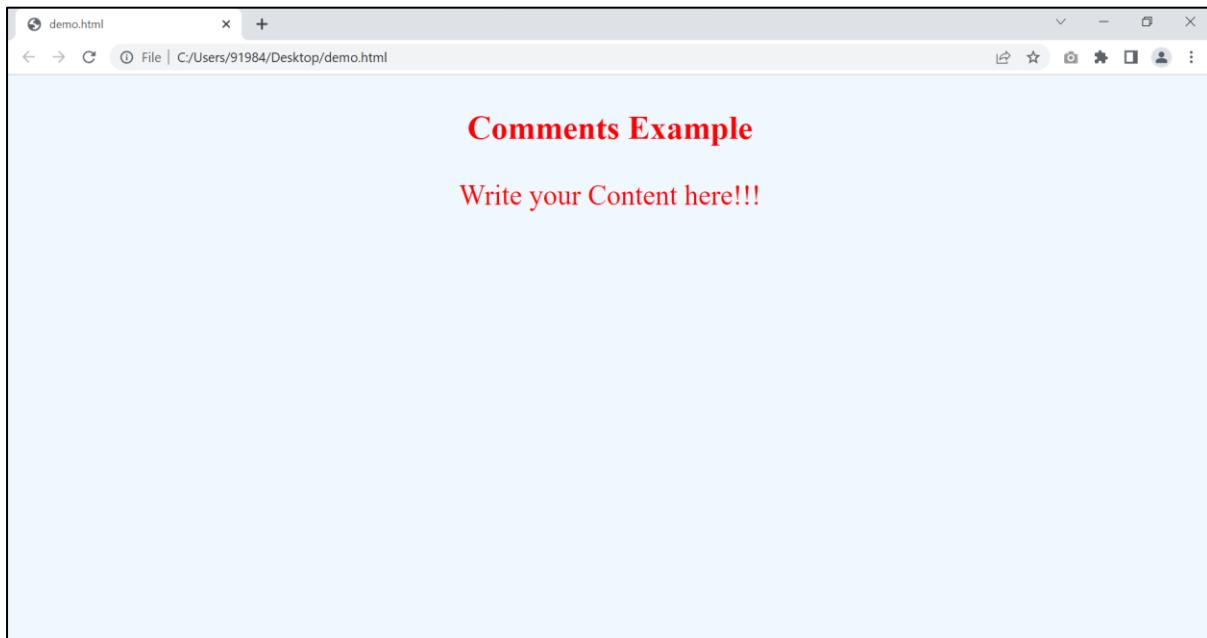
    </style>
</head>

<!-- This is body section, write code here which you want to
display on web-page -->

<body>

    <!-- heading tag -->
    <h2>First Webpage</h2>
    <!-- Paragraph tag -->
    <p>Write your Content here!!!</p>
</body>
</html>
```

Output:



5.Tags:

HTML tags are special keywords that specify how a web browser must format and display the content. Tags are wrapped in brackets < and >. They start with an open angle bracket (<) and close with a closed angle bracket (>). The ending tag has a forward slash(/) before the name of the element. HTML tags can nest (be placed) inside one another.

We have two types of tags. They are

- 5.1. Container and Empty (Non-Container) Tags
- 5.2. Block Level and Inline Tags

5.1. Container and Empty (Non-Container) Tags:

5.1.1 Container tag:

This tag contains 3 parts, namely, the opening tag, content that will be displayed in the browser & the closing tag. If we forget to add the closing tag then the browser will continue to implement the effect of that opening tag on the content inside of that tag. This will deform the structure of the overall webpage.

Syntax:

```
<tag name> Content </tag name>
```

5.1.2 Empty Tags:

Empty tags are the tags that only contain the opening tags, they do not have any closing tag. Hence, they don't affect the webpage if we forget to close any empty tag.

Syntax:

<tag name> or <tag name />

5.2 Block Level and Inline Tags:

Every HTML element has a default display value, depending on what type of element it is. There are two display values: block and inline.

5.2.1 Block level Tag:

A block-level element always starts on a new line, and the browsers automatically add some space (a margin) before and after the element.

A block-level element always takes up the full width available (stretches out to the left and right as far as it can).

5.2.2 Inline Tag:

An inline element does not start on a new line. An inline element only takes up as much width as necessary.

Tag List:

Tag Name	Container Tag	Empty Tag	Block Level Tag	Inline Tag	Description
<html>	Yes	No	Yes	No	Defines the root of an HTML document
<head>	Yes	No	Yes	No	Contains metadata/information for the document
<title>	Yes	No	Yes	No	Defines a title for the document
<body>	Yes	No	Yes	No	Defines the document's body
<h1>-<h6>	Yes	No	Yes	No	Defines HTML headings
<p>	Yes	No	Yes	No	Defines a paragraph
	Yes	No	No	Yes	Defines bold text
	Yes	No	No	Yes	Defines important text
<i>	Yes	No	No	Yes	Defines a part of text in an alternate voice or mood
	Yes	No	Yes	No	Defines an unordered list
	Yes	No	Yes	No	Defines an ordered list
	Yes	No	Yes	No	Defines a list item
<dl>	Yes	No	Yes	No	Defines a description list
<dt>	Yes	No	Yes	No	Defines a term/name in a description list
<dd>	Yes	No	Yes	No	Defines a description/value of a term in a description list
<div>	Yes	No	Yes	No	Defines a section in a document
	Yes	No	No	Yes	Defines a hyperlink
<button>	Yes	No	No	Yes	Defines a clickable button
<nav>	Yes	No	Yes	No	Defines navigation links
<script src="">	Yes	No	No	Yes	Defines a client-side script (java script)

<header>	Yes	No	Yes	No	Defines a header for a document or section
<footer>	Yes	No	Yes	No	Defines a footer for a document or section
<form>	Yes	No	Yes	No	Defines an HTML form for user input
<label>	Yes	No	No	Yes	Defines a label for an <input> element
<option>	Yes	No	Yes	No	Defines an option in a drop-down list
<section>	Yes	No	Yes	No	Defines a section in a document
<select>	Yes	No	No	Yes	Defines a drop-down list
	Yes	No	No	Yes	Defines a part of a document.
<style>	Yes	No	Yes	No	Defines style information for a document
<sub>	Yes	No	No	Yes	Defines subscripted text
<sup>	Yes	No	No	Yes	Defines superscripted text
<textarea>	Yes	No	No	Yes	Defines a multiline input control (text area)
<table>	Yes	No	Yes	No	Defines a table
<tbody>	Yes	No	Yes	No	Groups the body content in a table
<td>	Yes	No	No	Yes	Defines a cell in a table
<tfoot>	Yes	No	Yes	No	Groups the footer content in a table
<th>	Yes	No	No	Yes	Defines a header cell in a table
<thead>	Yes	No	Yes	No	Groups the header content in a table
<tr>	Yes	No	Yes	No	Defines a row in a table
 	No	Yes	Yes	No	Defines a single line break

<hr>	No	Yes	Yes	No	Defines a thematic change in the content
	No	Yes	No	Yes	Defines an image
<input>	No	Yes	No	Yes	Defines an input control
<link href="">	No	Yes	-	-	Defines the relationship between a document and an external resource (most used to link to style sheets)
<meta>	No	Yes	-	-	Defines metadata about an HTML document

6. Root Tags of HTML Document:

These are some root elements of every HTML document, without including these tags in HTML we can not create web pages. They are

- <html></html>
- <head></head>
- <title></title>
- <body></body>

6.1 <html> Tag:

The <html> tag represents the root of an HTML document.

The <html> tag is the container for all other HTML elements (except for the <!DOCTYPE> tag). It is container and block level tag.

6.2 <head> Tag:

The <head> element is a container for metadata (data about data) and is placed between the <html> tag and the <body> tag.

Note: Metadata is data about the HTML document. Metadata is not displayed.

6.3 <title> Tag:

The <title> tag defines the title of the document. The title must be text-only, and it is shown in the browser's title bar or in the page's tab.

The <title> tag is required in HTML documents!

6.4 <body> Tag:

The **<body>** tag defines the document's body.

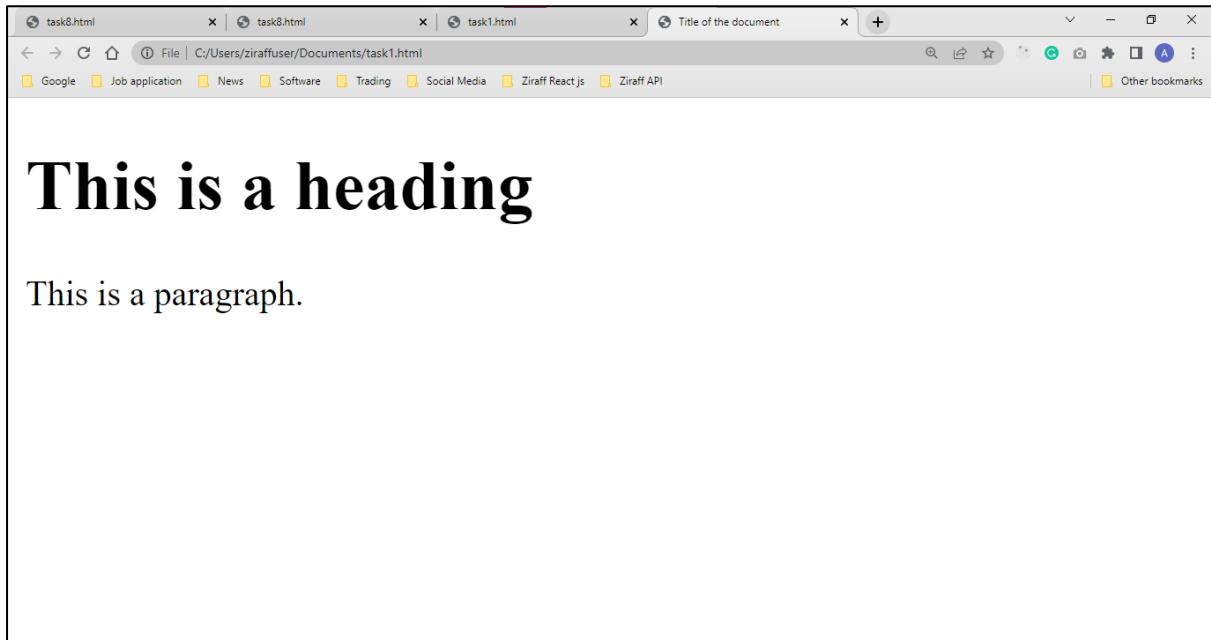
The **<body>** element contains all the contents of an HTML document, such as headings, paragraphs, images, hyperlinks, tables, lists, etc.

Note: There can only be one **<body>** element in an HTML document.

Example:

```
<!DOCTYPE html>
<html>
<head>
  <title>Title of the document</title>
</head>
<body>
  <h1>This is a heading</h1>
  <p>This is a paragraph. </p>
</body>
</html>
```

Output:



7. Heading Tags:

Heading tags are used to provide headings of different text sizes and styles to your web page. Heading tags vary from **<h1>** to **<h6>**.

- **<h1></h1>**
- **<h2></h2>**
- **<h3></h3>**
- **<h4></h4>**
- **<h5></h5>**
- **<h6></h6>**

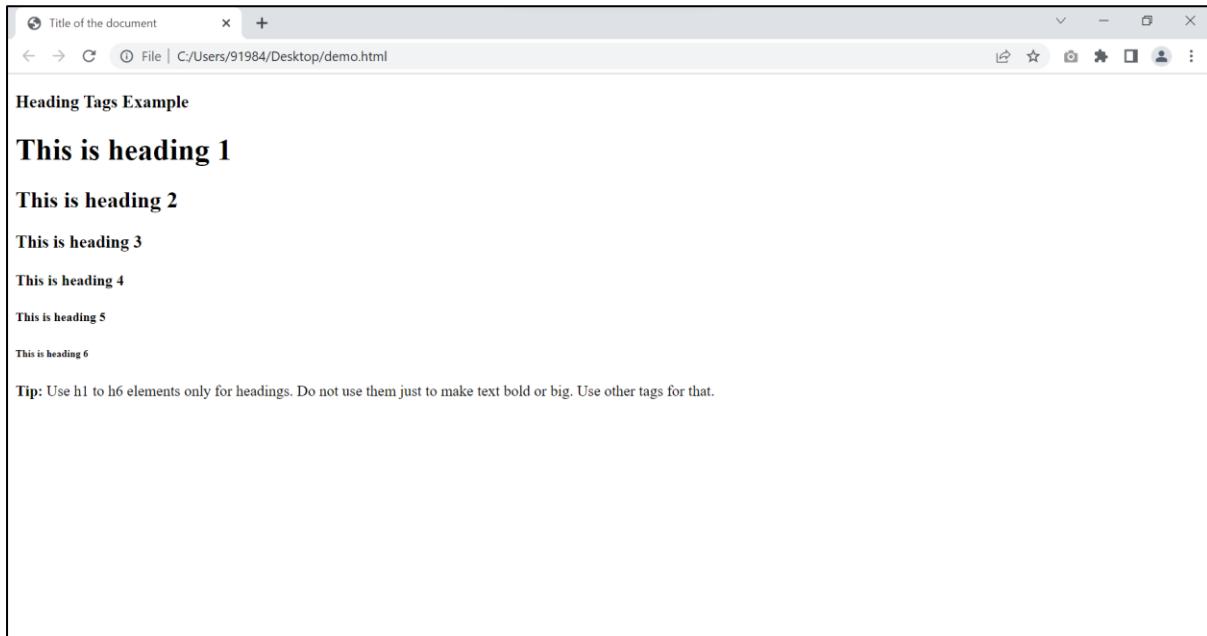
Example:

```
<!DOCTYPE html>

<html >

<head>
    <title>Title of the document</title>
</head>
<body>
    <h3>Heading Tags Example</h3>
    <h1>This is heading 1</h1>
    <h2>This is heading 2</h2>
    <h3>This is heading 3</h3>
    <h4>This is heading 4</h4>
    <h5>This is heading 5</h5>
    <h6>This is heading 6</h6>
    <p><b>Tip:</b> Use h1 to h6 elements only for headings. Do not use them just to make text bold or big. Use other tags for that. </p>
</body>
</html>
```

Output:



8. List Tags:

There are different list tags used for listing different items in a particular way, which is given below:

-
-
-
- <dl></dl>
- <dt></dt>
- <dd></dd>

8.1 Tag:

The tag defines an unordered (bulleted) list. Use the tag together with the tag to create unordered lists. It is container and block level tag.

8.2 Tag:

The tag defines an ordered list. An ordered list can be numerical or alphabetical. It is container and block level tag. The tag is used to define each list item.

8.3 Tag:

The tag defines a list item.

The tag is used inside ordered lists(), unordered lists ()

8.4 <dl> Tag:

The **<dl>** tag defines a description list. The **<dl>** tag is used in conjunction with **<dt>** (defines terms/names) and **<dd>** (describes each term/name).

8.5 <dt> Tag:

The **<dt>** tag defines a term/name in a description list.

The **<dt>** tag is used in conjunction with **<dl>** (defines a description list) and **<dd>** (describes each term/name).

8.6 <dd> Tag:

The **<dd>** tag is used to describe a term/name in a description list.

The **<dd>** tag is used in conjunction with **<dl>** (defines a description list) and **<dt>** (defines terms/names).

Inside a **<dd>** tag you can put paragraphs, line breaks, images, links, lists, et

Example:

```
<!DOCTYPE html>
<html>
<head>
<title>Title of the document</title>
</head>
<body>
<h1>The ul element</h1>
<ul>

<li>Coffee</li>
<li>Tea</li>
<li>Milk</li>

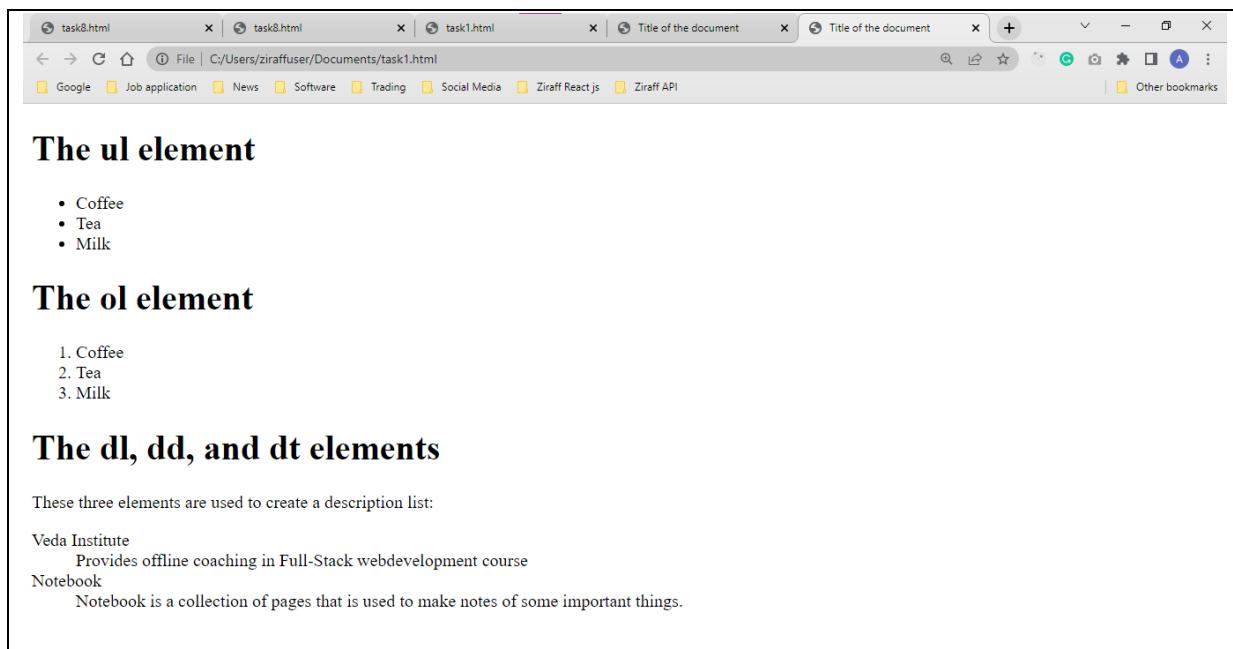
</ul>
<h1>The ol element</h1>
<ol>

<li>Coffee</li>
<li>Tea</li>
<li>Milk</li>

</ol>
<h1>The dl, dd, and dt elements</h1>
<p>These three elements are used to create a description list:</p>
<dl>

<dt>Veda Institute</dt>
<dd>Provides offline coaching in Full-Stack webdevelopment course</dd>
<dt>Notebook</dt>
<dd>Notebook is a collection of pages that is used to make notes of some important things. </dd>
</dl>
</body>
</html>
```

Output:



The screenshot shows a browser window with multiple tabs open. The active tab is titled "Title of the document". Below the tabs, there's a bookmarks bar with items like Google, Job application, News, Software, Trading, Social Media, Ziraff React.js, Ziraff API, and Other bookmarks.

The ul element

- Coffee
- Tea
- Milk

The ol element

1. Coffee
2. Tea
3. Milk

The dl, dd, and dt elements

These three elements are used to create a description list:

Veda Institute
Provides offline coaching in Full-Stack webdevelopment course
Notebook
Notebook is a collection of pages that is used to make notes of some important things.

9.Formatting Tags:

These tags define the way in which text will be shown on the web page.

- <p></p>
-
-
- <i></i>
-

9.1 <p> Tag:

The <p> tag defines a paragraph.

Browsers automatically add a single blank line before and after each <p> element. It is both container and block level tag.

9.2 Tag:

The tag specifies bold text without any extra importance. It is container and inline tag

9.3 Tag:

The tag is used to define text with strong importance. The content inside is typically displayed in **bold**. It is container and inline tag.

9.4 <i> Tag:

The <i> tag defines a part of text in an alternate voice or mood. The content inside is typically displayed in *italic*.

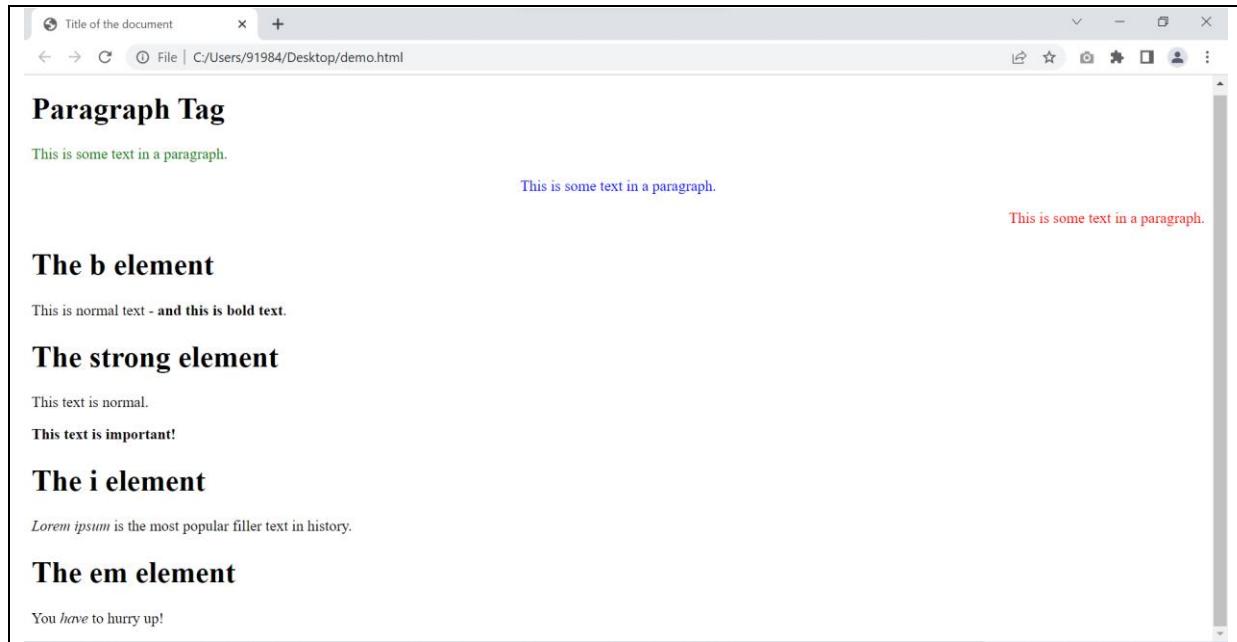
9.5 Tag:

The **** tag is used to define emphasized text. The content inside is typically displayed in *italic*.

Example:

```
<!DOCTYPE html>
<html>
<head>
<title>Title of the document</title>
</head>
<body>
<h1>Paragraph Tag</h1>
<p style="color: green">This is some text in a paragraph. </p>
<p style="text-align: center; color: blue;">This is some text in a paragraph. </p>
<p style="text-align: right; color: red;">This is some text in a paragraph. </p>
<h1>The b element</h1>
<p>This is normal text - <b>and this is bold text. </b> </p>
<h1>The strong element</h1>
<p>This text is normal. </p>
<p><strong>This text is important! </strong></p>
<h1>The i element</h1>
<p><i>Lorem ipsum</i> is the most popular filler text in history. </p>
<h1>The em element</h1>
<p>You <em>have</em> to hurry up! </p>
</body>
</html>
```

Output:



10. Other Commonly used tags:

- <div></div>
-
- <button></button>
- <nav></nav>
- <script src=""></script>

10.1 <div> Tag:

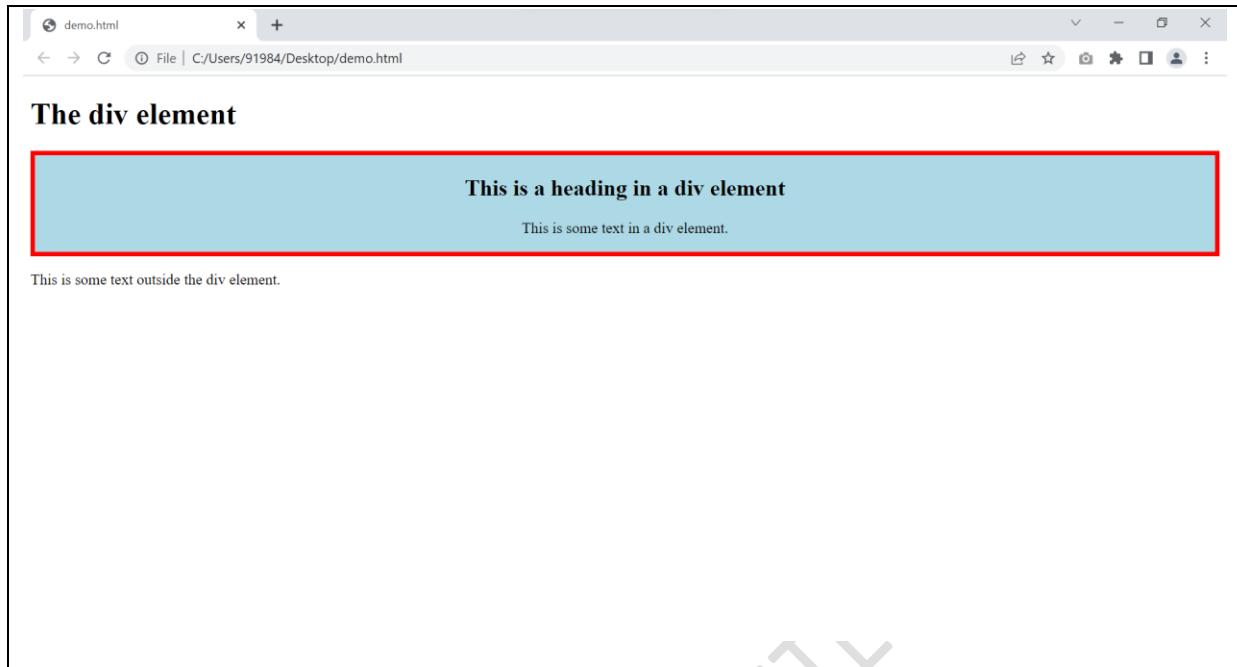
The <div> tag defines a division or a section in an HTML document. The <div> tag is used as a container for HTML elements - which is then styled with CSS or manipulated with JavaScript. The <div> tag is easily styled by using the class or id attribute. Any sort of content can be put inside the <div> tag!

Example:

```
<!DOCTYPE html>

<html >
  <head>
    <style>
      .myDiv {
        border: 5px solid red;
        background-color: lightblue;
        text-align: center;
      }
    </style>
  </head>
  <body>
    <h1>The div element</h1>
    <div class="myDiv">
      <h2>This is a heading in a div element</h2>
      <p>This is some text in a div element. </p>
    </div>
    <p>This is some text outside the div element. </p>
  </body>
</html>
```

Output:



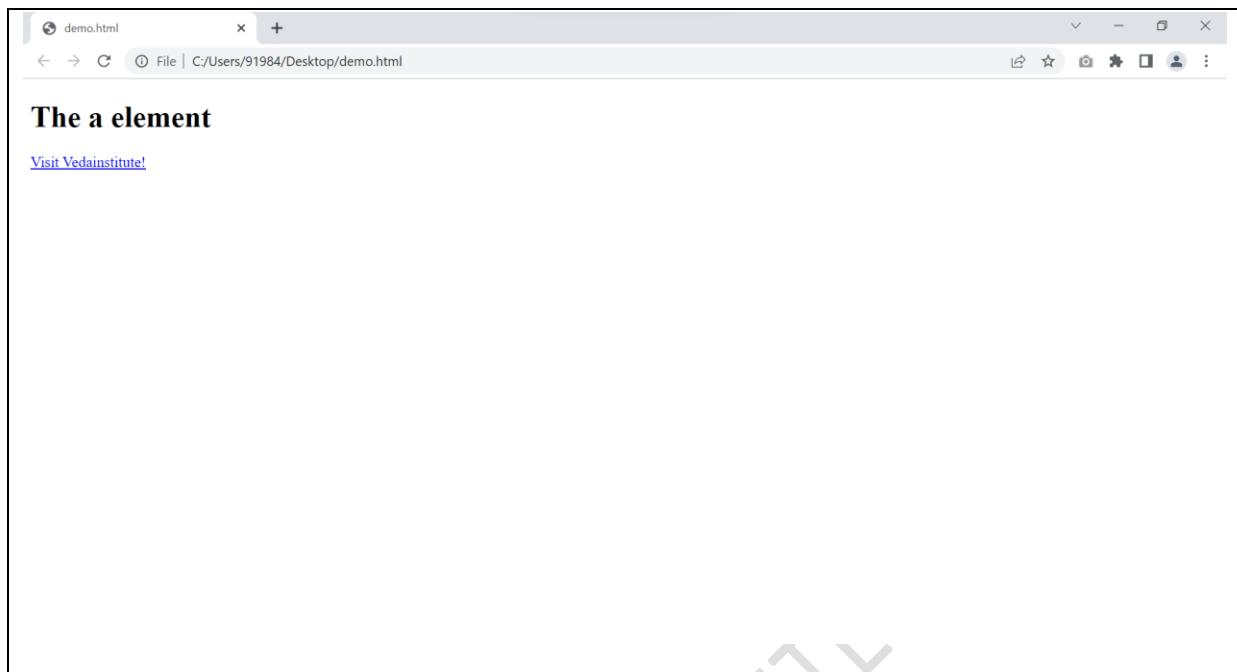
10.2 <a>(anchor) Tag:

The **<a>** tag defines a hyperlink, which is used to link from one page to another. The most important attribute of the **<a>** element is the **href** attribute, which indicates the link's destination.

Example:

```
<!DOCTYPE html>
<html>
<body>
<h1>The a element</h1>
<a href="https://www.vedainstitute.in">Visit Vedainstitute! </a>
</body>
</html>
```

Output:



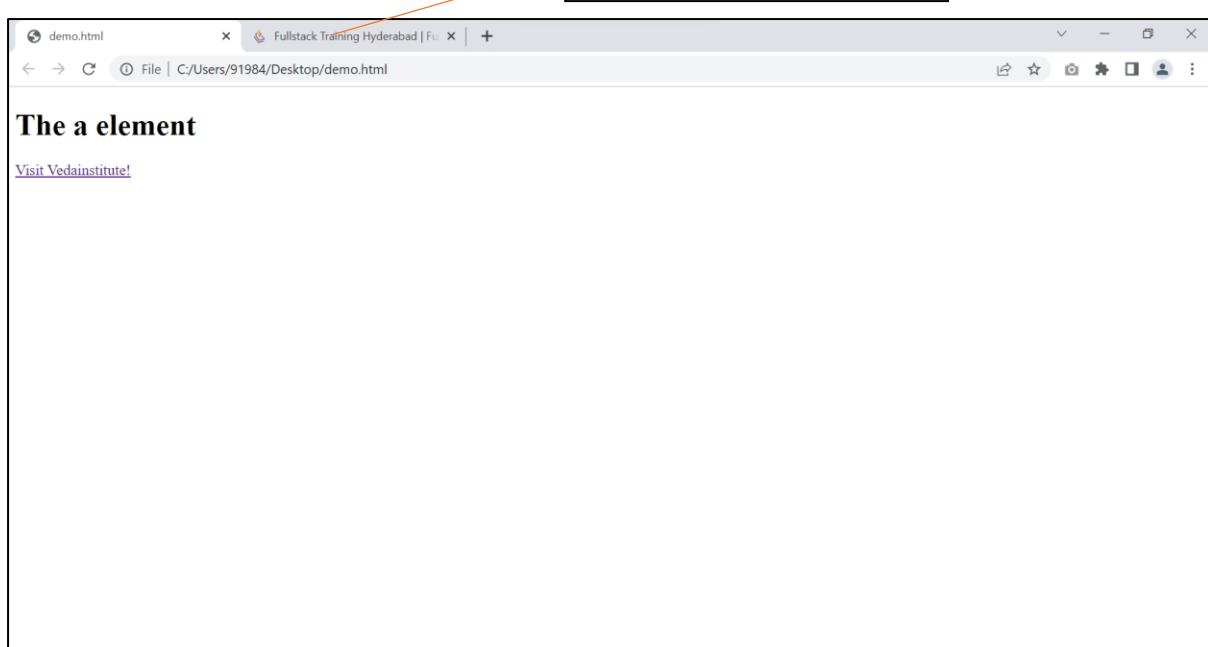
Note: If the **<a>** tag has no **href** attribute, it is only a placeholder for a hyperlink.

Target attribute:

If you set the target attribute to "**_blank**", the link will open in a new browser window or a new tab.

Example:

```
<!DOCTYPE html>
<html>
<body>
<h1>The a element</h1>
<a href="https://www.vedainstitute.in" target="_blank">Visit Veda institute!</a>
</body>
</html>
```

Output:

10.3 <nav>, <section> Tags:

<nav> Tag:

The <nav> tag defines a set of navigation links.

Note: NOT all links of a document should be inside a <nav> element. The <nav> element is intended only for major block of navigation links.

<section> Tag:

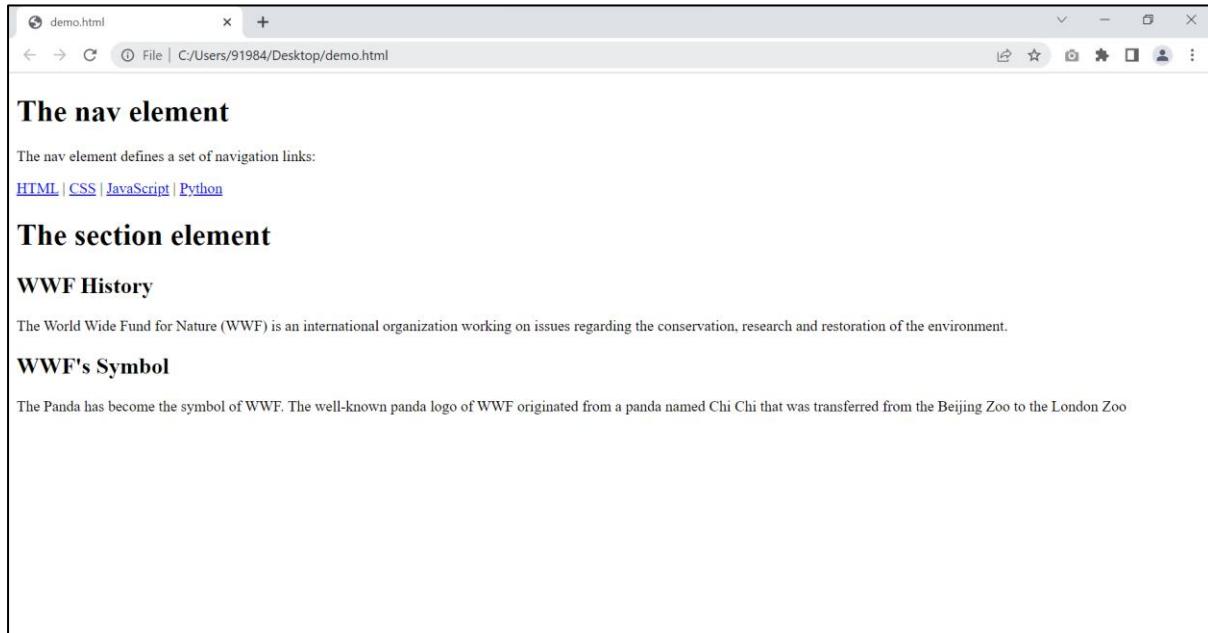
The <section> tag defines a section in a document.

Example:

```
<!DOCTYPE html>

<html>
  <body>
    <h1>The nav element</h1>
    <p>The nav element defines a set of navigation links:</p>
    <nav>
      <a href="/html/">HTML</a> |
      <a href="/css/">CSS</a> |
      <a href="/js/">JavaScript</a> |
      <a href="/python/">Python</a>
    </nav>
    <h1>The section element</h1>
    <section>
      <h2>WWF History</h2>
      <p>The World-Wide Fund for Nature (WWF) is an international organization working on issues regarding the conservation, research and restoration of the environment.</p>
    </section>
    <section>
      <h2>WWF's Symbol</h2>
      <p>The Panda has become the symbol of WWF. The well-known panda logo of WWF originated from a panda named Chi Chi that was transferred from the Beijing Zoo to the London Zoo</p>
    </section>
  </body>
</html>
```

Output:



10.4 Tag:

The **** tag is used to embed an image in an HTML page. The **** tag has two required attributes:

- **src** - Specifies the path to the image
- **alt** - Specifies an alternate text for the image, if the image for some reason cannot be displayed

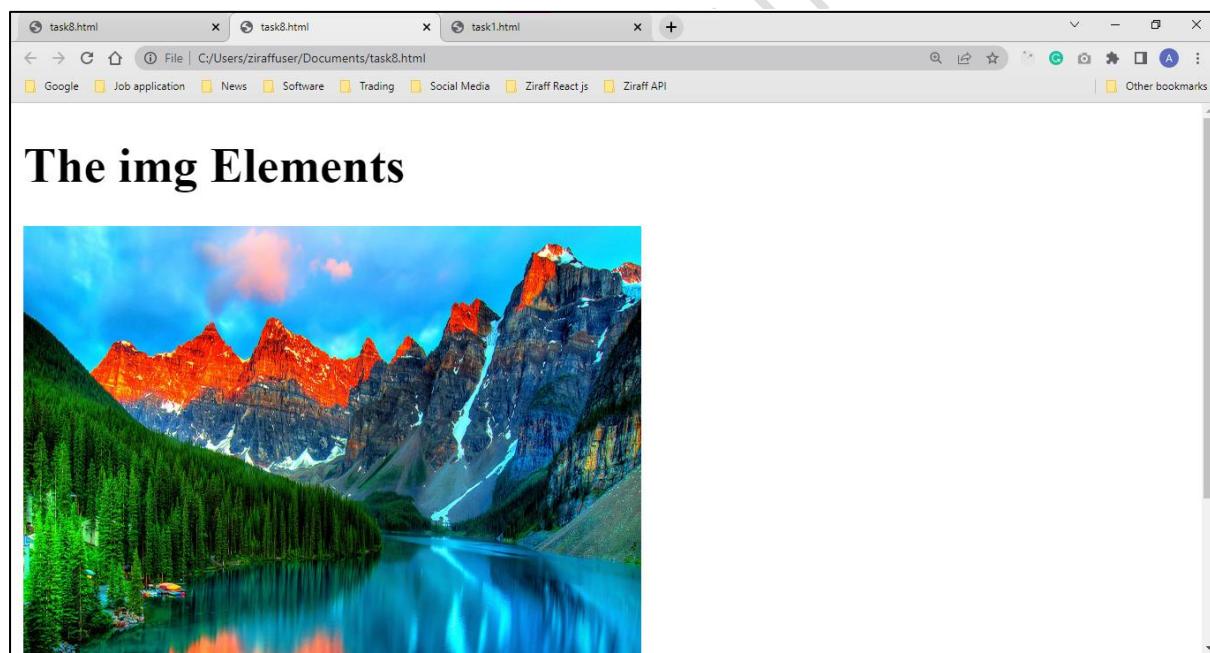
Note: To link an image to another document, simply nest the **** tag inside an **<a>** tag

Example:

```
<!DOCTYPE html>
<html>
<body>
<h1>The img Elements</h1>

</body>
</html>
```

Output:



10.5 ,
, <sup> and <sub> Tags:

 Tag:

The **** tag is an inline container used to mark up a part of a text, or a part of a document.

The **** tag is much like the **<div>** element, but **<div>** is a block-level element and **** is an inline element.

**
 Tag:**

The **
** tag inserts a single line break. The **
** tag is an empty tag which means that it has no end tag.

<sup> Tag:

The **<sup>** tag defines superscript text. Superscript text appears half a character above the normal line, and is sometimes rendered in a smaller font. Superscript text can be used for footnotes, like WWW^[1].

<sub> Tag:

The **<sub>** tag defines subscript text. Subscript text appears half a character below the normal line, and is sometimes rendered in a smaller font. Subscript text can be used for chemical formulas, like H₂O.

Example:

```
<!DOCTYPE html>

<html>

<head>

<title>Page Title</title>

</head>

<body>

<h1>The span Elements</h1>

<p>My mother has <span style="color: blue; font-weight: bold;">Blue</span>eyes and
my father has

<span style="color: darkolivegreen; font-weight: bold">drak green</span>eyes</p>

<h1>The br Elements</h1>

<p>To force<br>line breaks<br> in a text, <br>use the br<br>element</p>

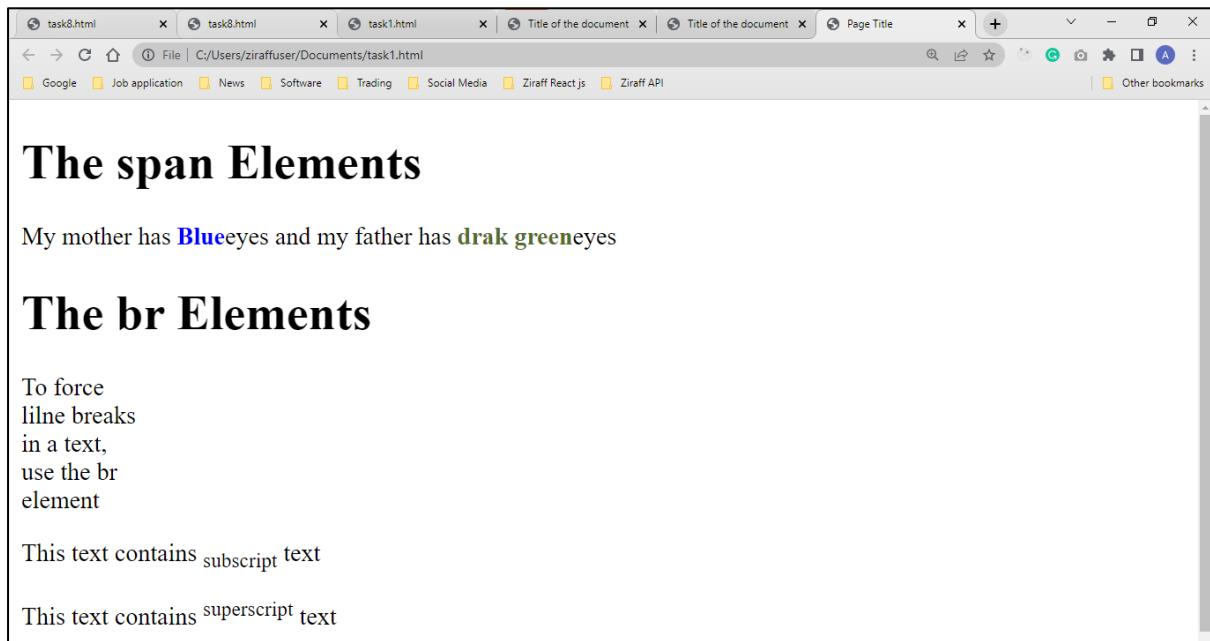
<p>This text contains <sub>subscript</sub> text</p>

<p>This text contains <sup>superscript</sup> text</p>

</body>

</html>
```

Output:



10.6 <script> Tag:

The **<script>** tag is used to embed a client-side script (JavaScript).

The **<script>** element either contains scripting statements, or it points to an external script file through the **src** attribute.

Example:

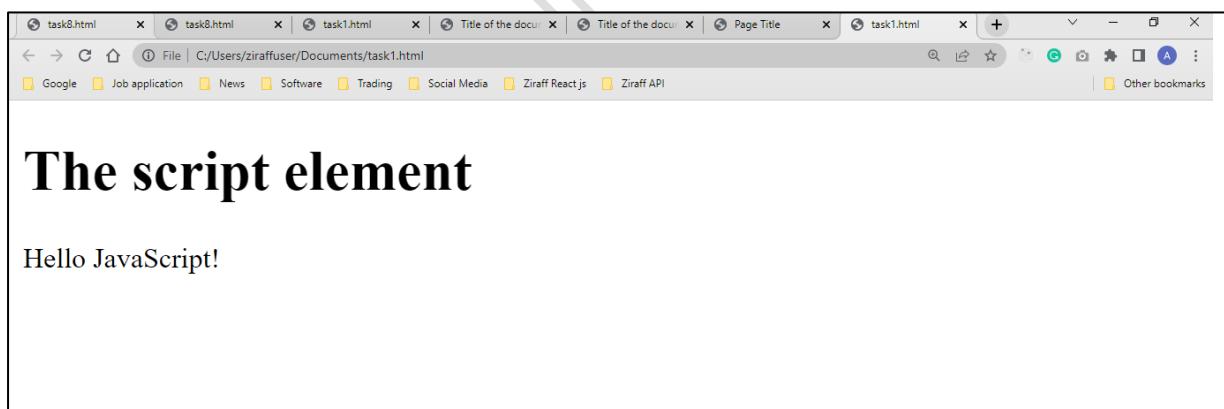
```
<!DOCTYPE html>
<html>
<body>

<h1>The script element</h1>
<p id="demo"></p>

<script>
document.getElementById("demo").innerHTML = "Hello JavaScript!";
</script>

</body>
</html>
```

Output:



10.7 <meta> tag:

The **<meta>** tag defines metadata about an HTML document. Metadata is data (information) about data.

<meta> tags always go inside the **<head>** element, and are typically used to specify character set, page description, keywords, author of the document, and viewport settings.

Metadata will not be displayed on the page, but is machine parsable.

Example:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<meta name="description" content="Free Web tutorials">
<meta name="keywords" content="HTML,CSS,XML,JavaScript">
<meta name="author" content="John Doe">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
</head>
<body>
<p>All meta information goes in the head section...</p>
</body>
</html>
```

<meta name="viewport" content="width=device-width, initial-scale=1.0">

The **width=device-width** part sets the width of the page to follow the screen-width of the device (which will vary depending on the device).

The **initial-scale=1.0** part sets the initial zoom level when the page is first loaded by the browser.

Output:



11.<table> Tag:

HTML tables allow web developers to arrange data into rows and columns.

11.1 Table Cells:

Each table cell is defined by a <td> and a </td> tag.

11.2 Table Rows:

Each table row starts with a <tr> and end with a </tr> tag.

11.3 Table Headers:

Sometimes you want your cells to be headers, in those cases use the <th> tag instead of the <td> tag:

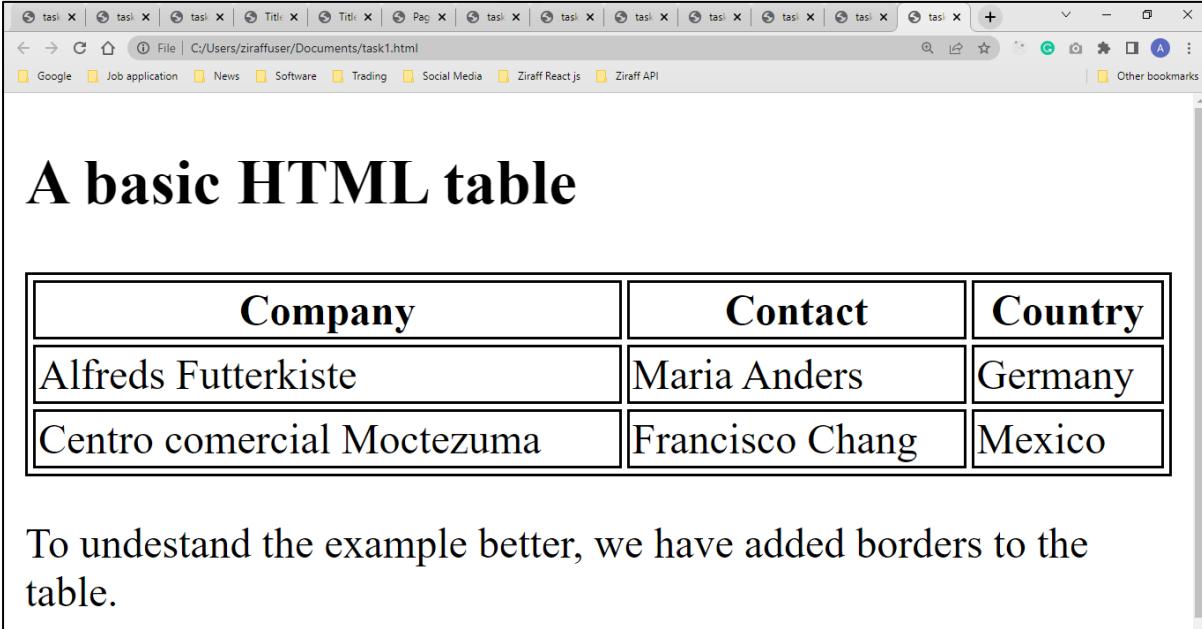
11.4 Table Borders:

To add a **border**, use the CSS border property on **table**, **th**, and **td** elements or add **border attribute** to table

Example:

```
<!DOCTYPE html>
<html>
<style>
table, th, td{
    border:1px solid black;
}
</style>
<body>
<h2>A basic HTML table</h2>
<table style="width:100%">
<tr>
    <th>Company</th>
    <th>Contact</th>
    <th>Country</th>
</tr>
<tr>
    <td>Alfreds Futterkiste</td>
    <td>Maria Anders</td>
    <td>Germany</td>
</tr>
<tr>
    <td>Centro comercial Moctezuma</td>
    <td>Francisco Chang</td>
    <td>Mexico</td>
</tr>
</table>
<p>To understand the example better, we have added borders to the table.</p>
</body>
</html>
```

Output:



Company	Contact	Country
Alfreds Futterkiste	Maria Anders	Germany
Centro comercial Moctezuma	Francisco Chang	Mexico

To understand the example better, we have added borders to the table.

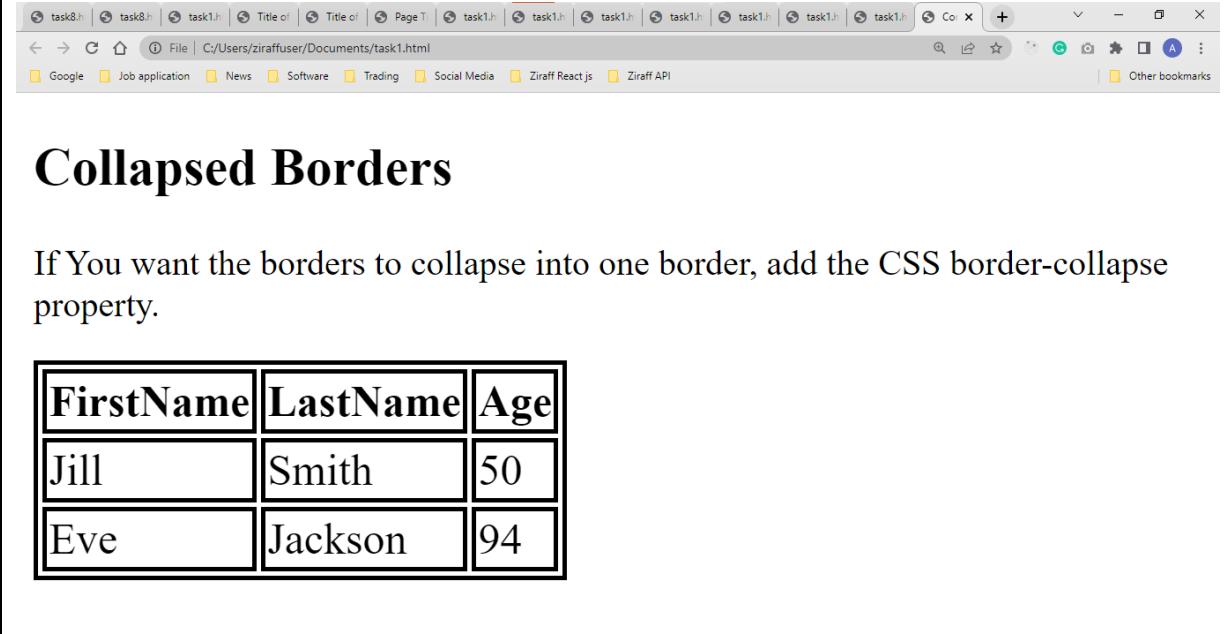
11.5 Collapsed Borders:

To avoid having double borders like in the example above, set the CSS **border-collapse** property to **collapse**.

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Container Tag</title>
<style>
table, th, td {
border: 2px solid black;
border-collapse: collapse;
font-size: 20px;
}
</style>
</head>
<body>
<h2>Collapsed Borders</h2>
<p>If you want the borders to collapse into one border, add the CSS border-collapse property.</p>
<table style="width: 100%">
<tr>
<th>FirstName</th>
<th>LastName</th>
<th>Age</th>
</tr>
<tr>
<td>Jill</td>
<td>Smith</td>
<td>50</td>
</tr>
<tr>
<td>Eve</td>
<td>Jackson</td>
<td>94</td>
</tr>
</table>
</body>
</html>
```

Output:



The screenshot shows a web browser window with the address bar displaying "C:/Users/ziraffuser/Documents/task1.html". Below the address bar is a bookmark bar with several items: Google, Job application, News, Software, Trading, Social Media, Ziraff React.js, and Ziraff API. The main content area of the browser displays a table with three columns: FirstName, LastName, and Age. The table has three rows. The first row contains the column headers. The second row contains the values "Jill" in the FirstName column, "Smith" in the LastName column, and "50" in the Age column. The third row contains the values "Eve" in the FirstName column, "Jackson" in the LastName column, and "94" in the Age column. All cells have a thin black border.

11.6 Table cell Padding & Spacing:

Cell padding:

Cell padding is the space between the cell edges and the cell content.

By default, the padding is set to 0.

Cell spacing:

Cell spacing is the space between each cell.

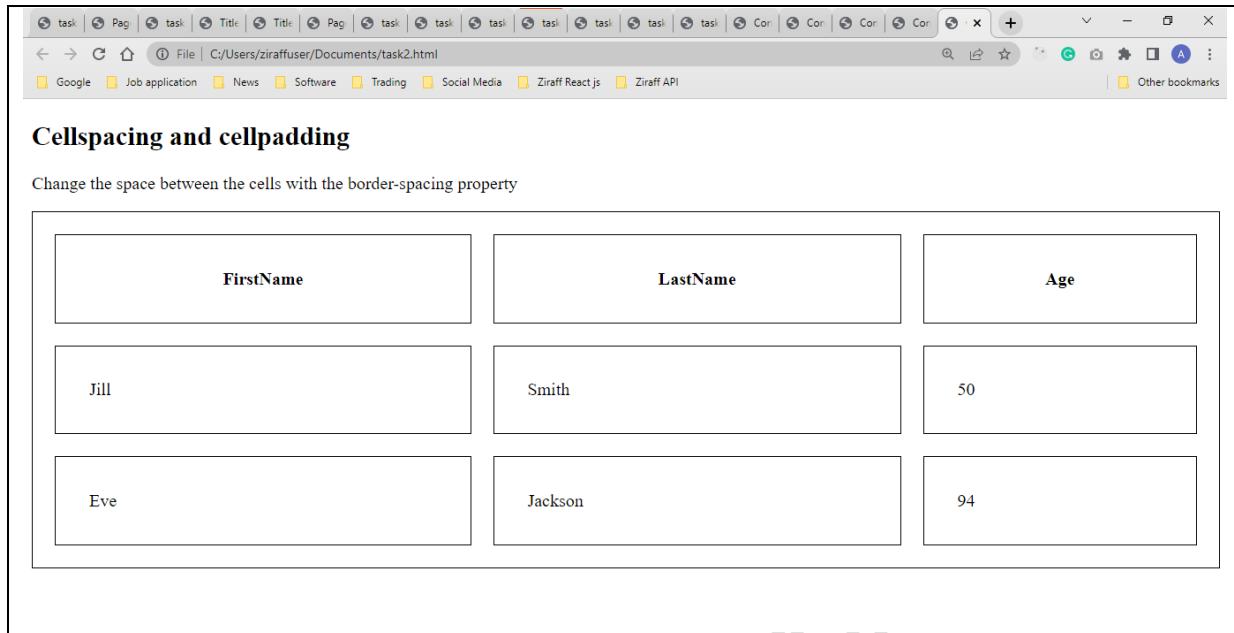
By default, the space is set to 2 pixels.

Example:

```
<!DOCTYPE html>
<html>
<head>
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Container Tag</title>
<style>
table, th, td {
border: 1px solid black;
}
</style>
</head>
<body>
<h2>Cellspacing and cellpadding </h2>
<p>Change the space between the cells with the border-spacing property</p>
<table style="width:100%" Cellspacing="20px" Cellpadding="30px">
<tr>
<th>FirstName</th>
<th>LastName</th>
<th>Age</th>
</tr>
<tr>
<td>Jill</td>
<td>Smith</td>
<td>50</td>
</tr>
<tr>
<td>Eve</td>
<td>Jackson</td>
<td>94</td>
</tr>

</table>
</body>
</html>
```

Output:



11.7 Table Colspan & Rowspan:

Colspan:

To make a cell span over multiple columns, use the **colspan** attribute

Rowspan:

To make a cell span over multiple rows, use the **rowspan** attribute

Example:

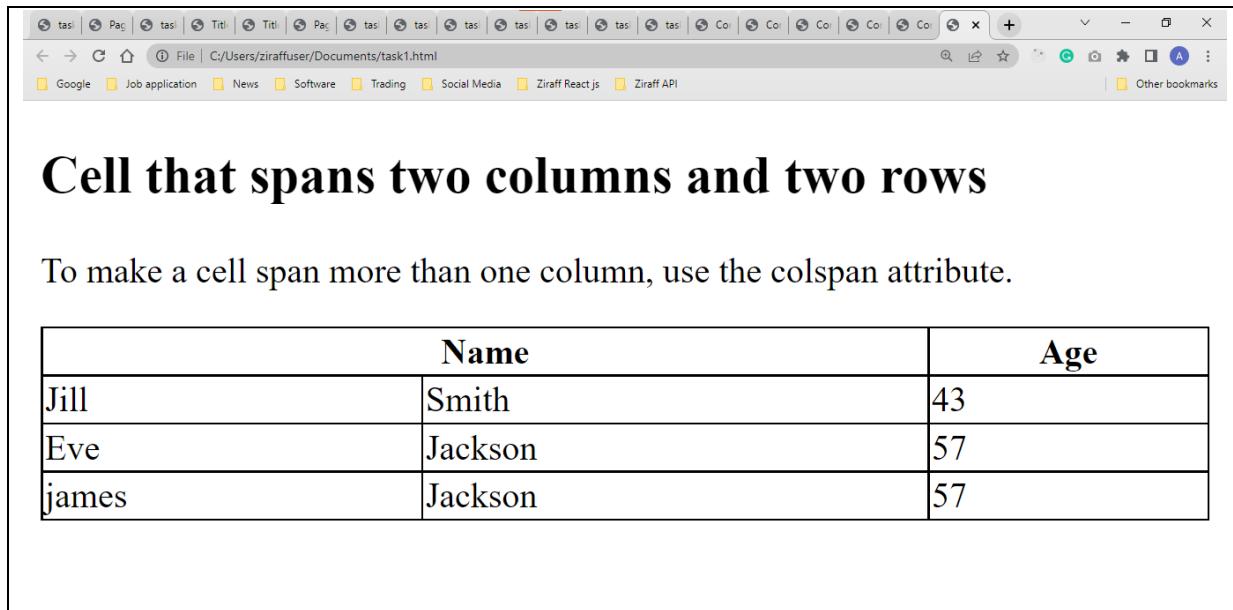
```
<!DOCTYPE html>
<html>
<head>
<style>
table, th, td {
    border: 1px solid black;
    border-collapse: collapse;
}
</style>
</head>
<body>

<h2>Cell that spans two columns and two rows</h2>
<p>To make a cell span more than one column, use the colspan attribute.</p>

<table style="width:100%">
<tr>
    <th colspan="2">Name</th>
    <th>Age</th>
</tr>
<tr>
    <td rowspan="2">Jill</td>
    <td>Smith</td>
    <td>43</td>
</tr>
<tr>
    <td>Eve</td>
    <td>Jackson</td>
    <td>57</td>
</tr>
<tr>
    <td>james</td>
    <td>Jackson</td>
    <td>57</td>
</tr>
</table>

</body>
```

Output:



Name		Age
Jill	Smith	43
Eve	Jackson	57
james	Jackson	57

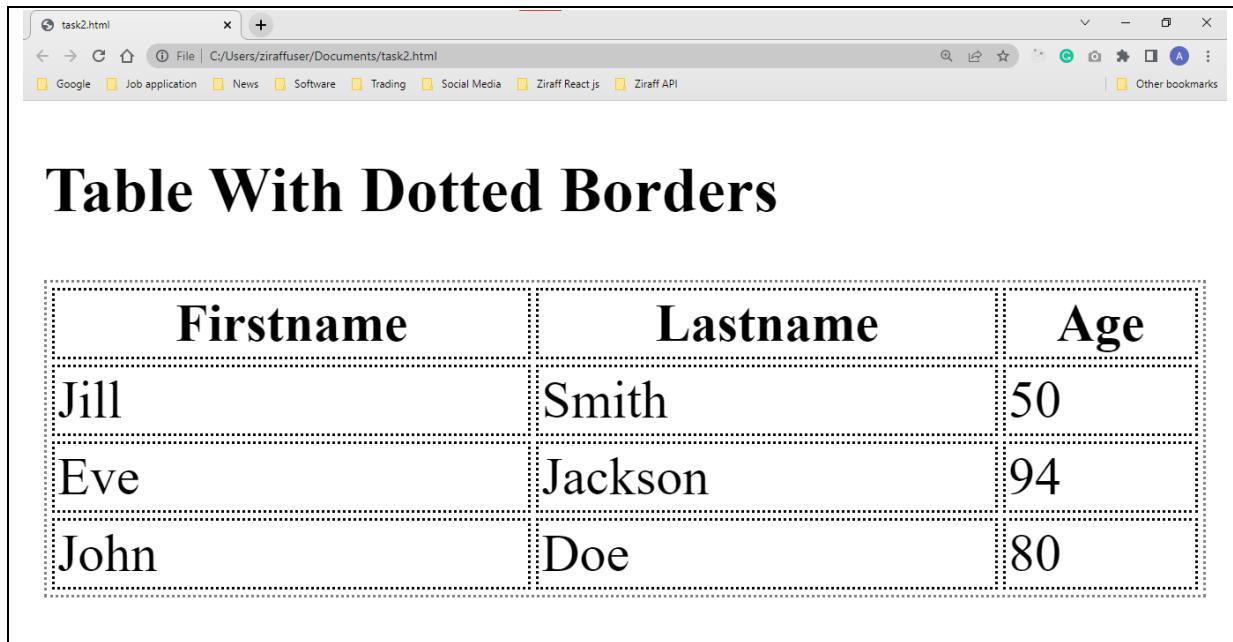
Table Borders:

- Dotted 
- Dashed 
- solid 
- double 
- groove 
- ridge 
- inset 
- outset 
- none
- hidden

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
Table, th, td {
border-style: dotted;
Font-size:20px;
}
</style>
</head>
<body>
<h2>Table With Dotted Borders</h2>
<table style="width:100%">
<tr>
<th>Firstname</th>
<th>Lastname</th>
<th>Age</th>
</tr>
<tr>
<td>Jill</td>
<td>Smith</td>
<td>50</td>
</tr>
<tr>
<td>Eve</td>
<td>Jackson</td>
<td>94</td>
</tr>
<tr>
<td>John</td>
<td>Doe</td>
<td>80</td>
</tr>
</table>
</body>
</html>
```

Output:



A screenshot of a Microsoft Edge browser window titled "task2.html". The address bar shows the file path: C:/Users/ziraffuser/Documents/task2.html. Below the address bar is a bookmarks bar with several items: Google, Job application, News, Software, Trading, Social Media, Ziraff React.js, Ziraff API, and Other bookmarks. The main content area displays a table with three columns: Firstname, Lastname, and Age. The table has four rows of data: Jill Smith 50, Eve Jackson 94, and John Doe 80. The table uses dotted borders for its cells.

Firstname	Lastname	Age
Jill	Smith	50
Eve	Jackson	94
John	Doe	80

12.<form> Tag:

The **<form>** tag is used to create an HTML form for user input. The **<form>** element is a container for different types of input elements, such as: text fields, checkboxes, radio buttons, submit buttons, etc.

<form> Elements:

The HTML **<form>** element can contain one or more of the following form elements:

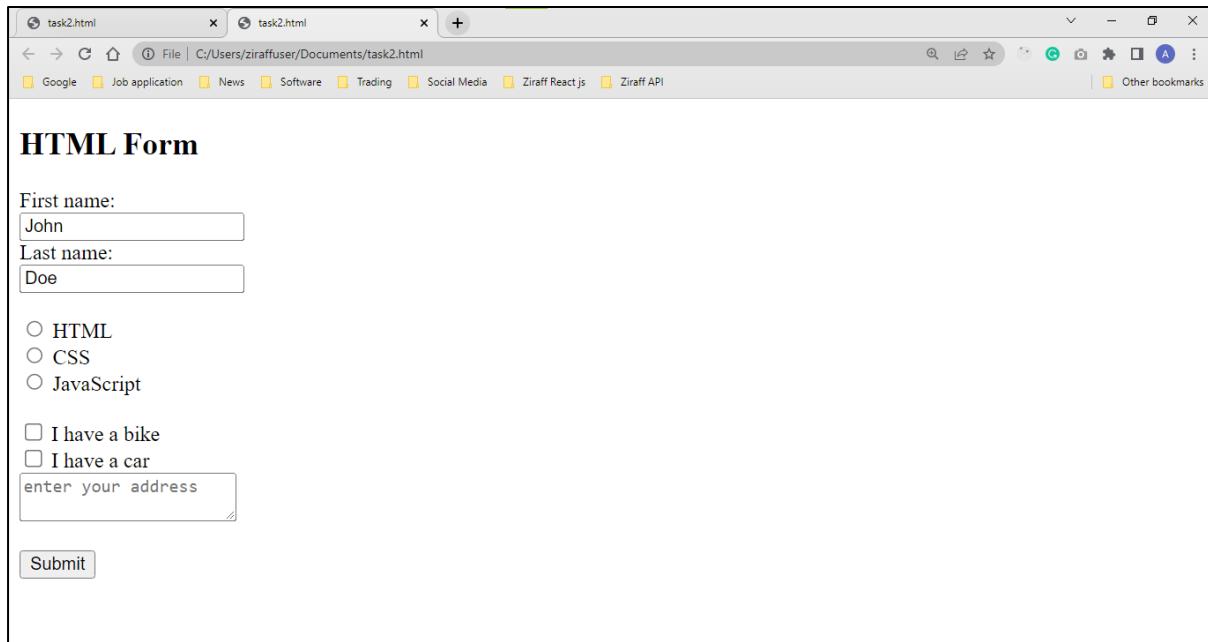
- **<input>**
- **<label>**
- **<select>**
- **<textarea>**
- **<button>**

Example:

```
<!DOCTYPE html>

<html>
<body>
<h2>HTML Form</h2>
<form action="/action_page.php">
<label for="fname">First name:</label><br>
<input type="text" id="fname" name="fname" value="John"><br>
<label for="lname">Last name:</label><br>
<input type="text" id="lname" name="lname" value="Doe"><br><br>
<input type="radio" id="html" name="fav_language" value="HTML">
<label for="html">HTML</label><br>
<input type="radio" id="css" name="fav_language" value="CSS">
<label for="css">CSS</label><br>
<input type="radio" id="javascript" name="fav_language" value="JavaScript">
<label for="javascript">JavaScript</label><br><br>
<input type="checkbox" id="vehicle1" name="vehicle1" value="Bike">
<label for="vehicle1"> I have a bike</label><br>
<input type="checkbox" id="vehicle2" name="vehicle2" value="Car">
<label for="vehicle2"> I have a car</label><br>
<textarea placeholder="enter your address"></textarea><br><br>
<input type="submit" value="Submit">
</form>
</body>
</html>
```

Output:



task2.html

File | C:/Users/ziraffuser/Documents/task2.html

Google Job application News Software Trading Social Media Ziraff React.js Ziraff API Other bookmarks

HTML Form

First name:

Last name:

HTML
 CSS
 JavaScript

I have a bike
 I have a car

enter your address

Submit

12.1 <button> Tag:

The **<button>** tag defines a clickable button. Inside a **<button>** element you can put text (and tags like **<i>**, ****, ****, **
, **, etc.). That is not possible with a button created with the **<input>** element!

Note: Always specify the **type** attribute for a **<button>** element, to tell browsers what type of button it is.

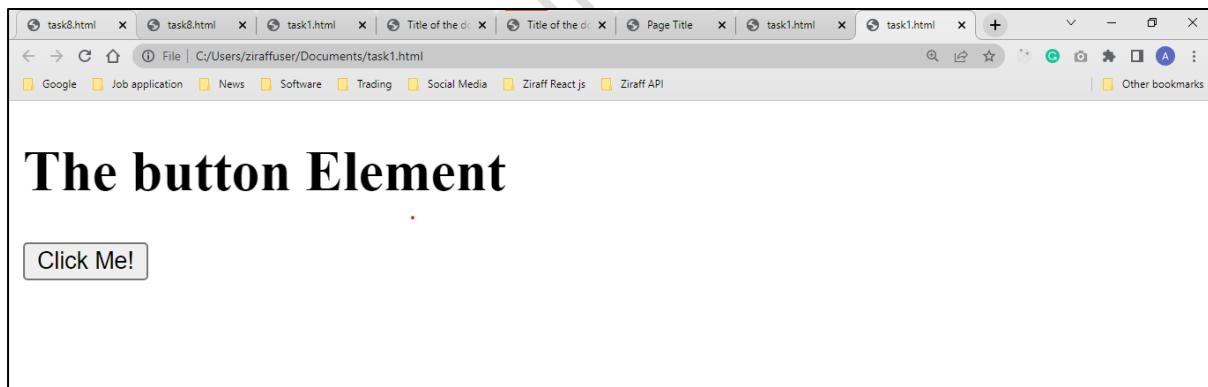
Example:

```
<!DOCTYPE html>
<html>
<body>

<h1>The button Element</h1>

<button type="button" onclick="alert ('Hello world!')">Click Me! </button>
</body>
</html>
```

Output:



12.2 <label> Tag:

The **<label>** tag defines a label for several elements:

Note: The **for** attribute of **<label>** must be equal to the **id** attribute of the related element to bind them together. A label can also be bound to an element by placing the element inside the **<label>** element.

Example:

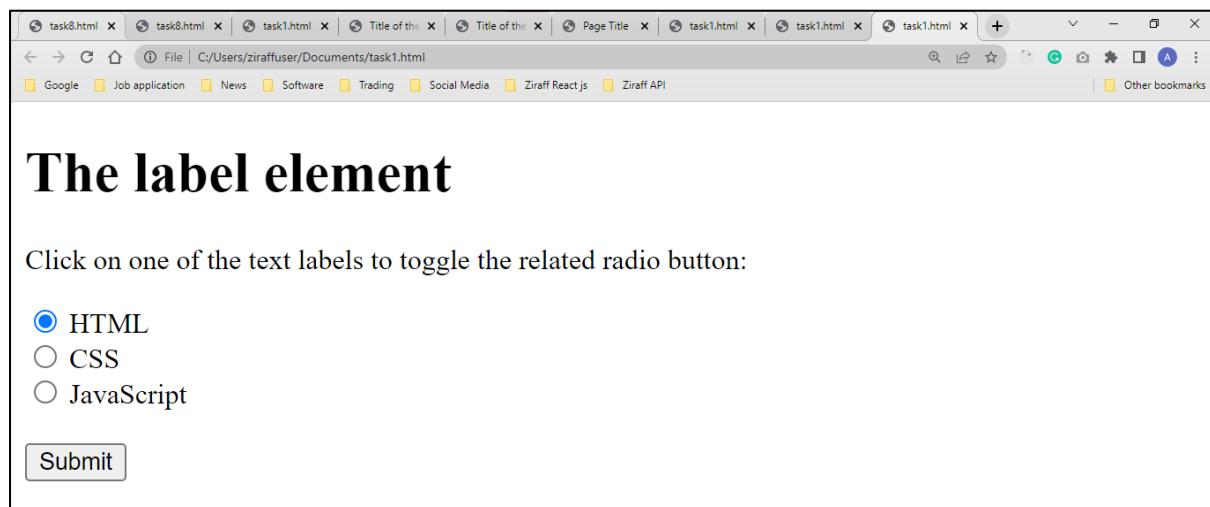
```
<!DOCTYPE html>
<html>
<body>
<h1>The label element</h1>
<p>Click on one of the text labels to toggle the related radio button:</p>
<form action="/action_page.php">
<input type="radio" id="html" name="fav_language" value="HTML">
<label for="html">HTML</label><br>

<input type="radio" id="css" name="fav_language" value="CSS">
<label for="css">CSS</label><br>

<input type="radio" id="javascript" name="fav_language" value="JavaScript">
<label for="javascript">JavaScript</label><br><br>

<input type="submit" value="Submit">
</form>
</body>
</html>
```

Output:



The label element

Click on one of the text labels to toggle the related radio button:

HTML
 CSS
 JavaScript

Submit

12.3 <select> Tag:

The **<select>** element is used to create a drop-down list.

The **<select>** element is most often used in a form, to collect user input.

The **<option>** tags inside the **<select>** element define the available options in the drop-down list.

Attributes:

The **name** attribute is needed to reference the form data after the form is submitted (if you omit the **name** attribute, no data from the drop-down list will be submitted).

The **id** attribute is needed to associate the drop-down list with a label.

Example:

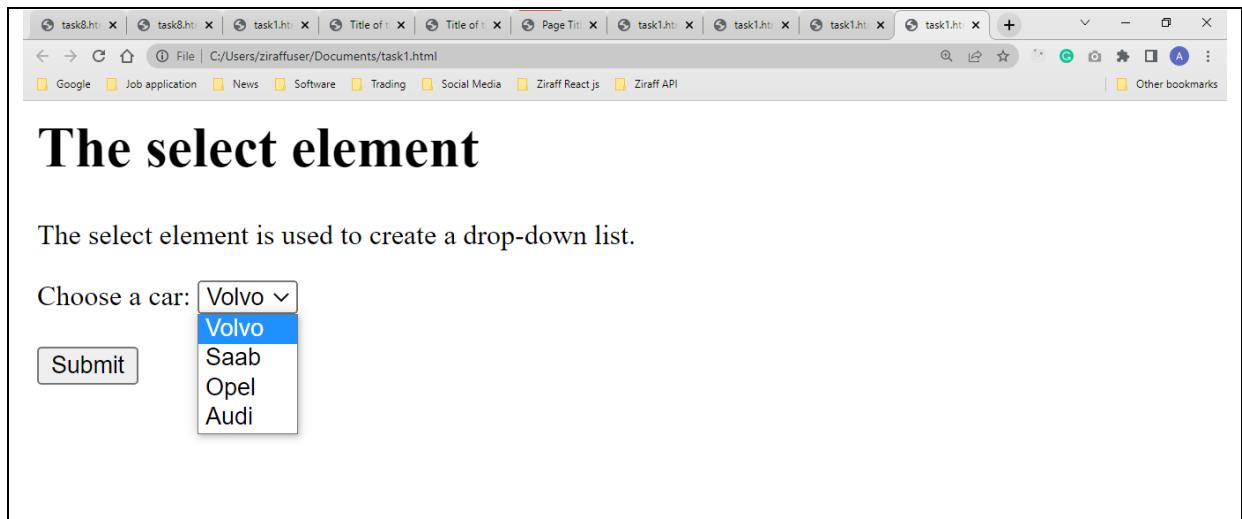
```
<!DOCTYPE html>
<html>
<body>
<h1>The select element</h1>
<p>The select element is used to create a drop-down list. </p>
<form action="/action_page.php">

<label for="cars">Choose a car:</label>

<select name="cars" id="cars">
    <option value="volvo">Volvo</option>
    <option value="saab">Saab</option>
    <option value="opel">Opel</option>
    <option value="audi">Audi</option>
</select>
<br><br>
<input type="submit" value="Submit">

</form>
</body>
</html>
```

Output:



12.4 <textarea> Tag:

The **<textarea>** tag defines a multi-line text input control.

The **<textarea>** element is often used in a form, to collect user inputs like comments or reviews.

The size of a text area is specified by the **<cols>** and **<rows>** attributes

Example:

```
<!DOCTYPE html>

<html>

<body>
<h1>The textarea element</h1>

<form action="/action_page.php">

<p><label for="w3review">Review of Veda</label></p>

<textarea id="w3review" name="w3review" rows="4" cols="50">At Veda you
will learn how to make a website. They are providing all web development
technologies. </textarea>

<br>

<input type="submit" value="Submit">

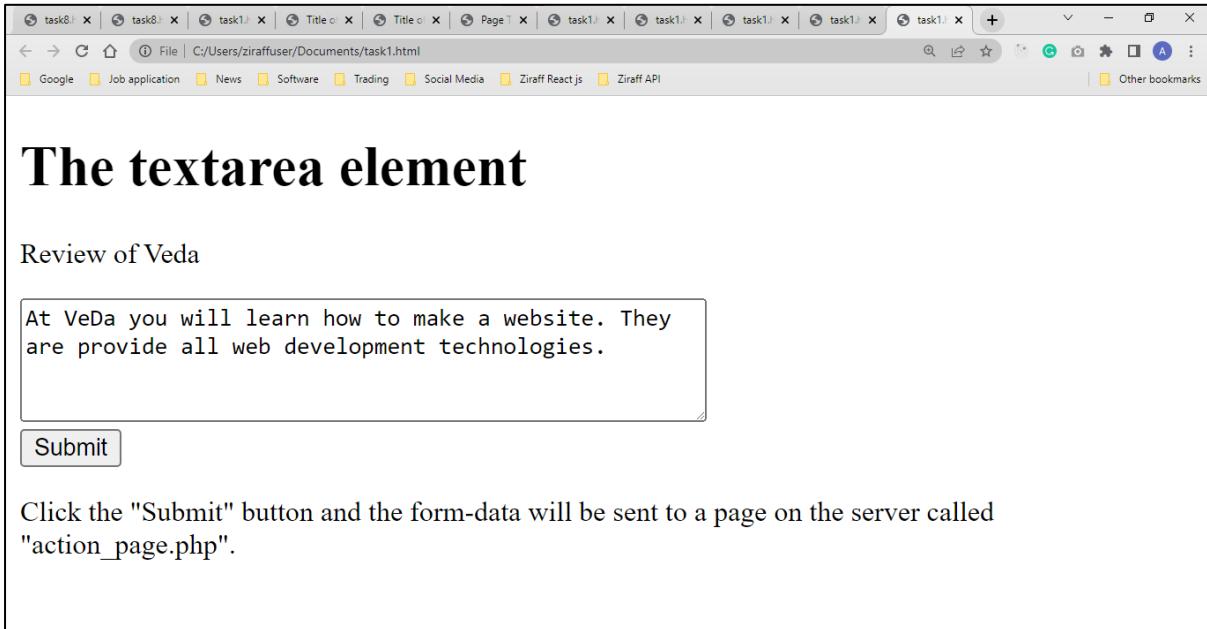
</form>

<p>Click the "Submit" button and the form-data will be sent to a page on the
server called "action_page.php". </p>

</body>

</html>
```

Output:



The screenshot shows a web browser window with multiple tabs open. The active tab displays a form titled "The textarea element". The form contains the following content:

Review of Veda

At VeDa you will learn how to make a website. They are provide all web development technologies.

Click the "Submit" button and the form-data will be sent to a page on the server called "action_page.php".

12.5 <input> Tag:

The **<input>** tag specifies an input field where the user can enter data.
The **<input>** element is the most important form element.
The **<input>** element can be displayed in several ways, depending on the type attribute.

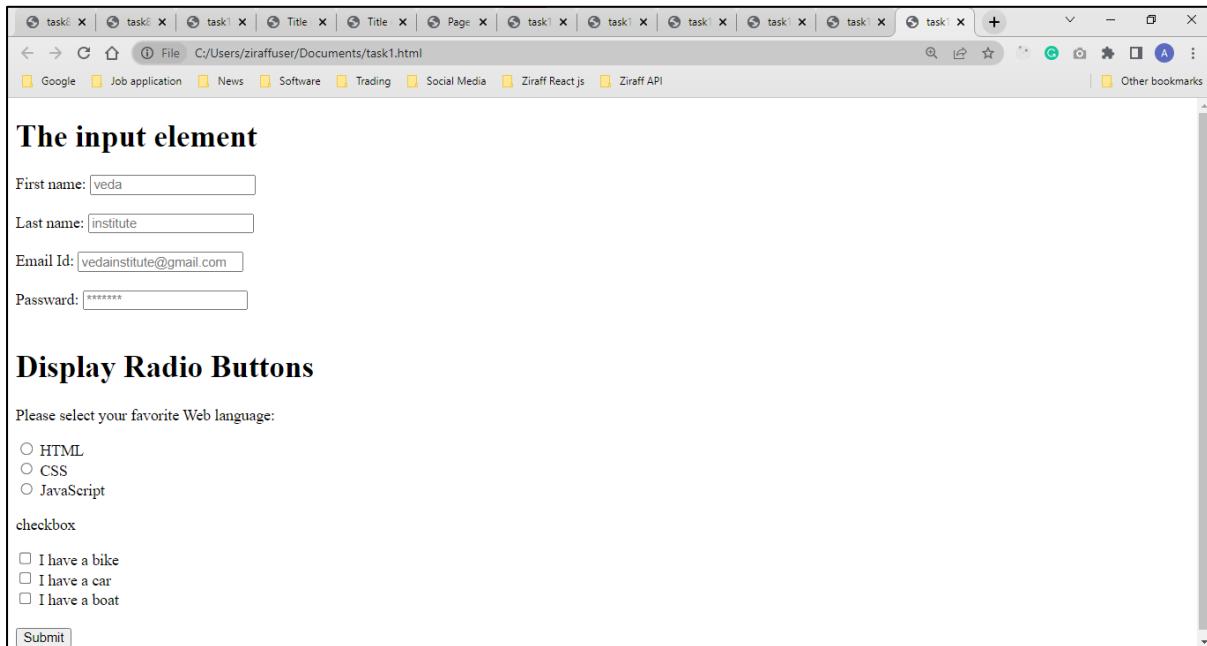
The different input types are as follows:

```
<input type="text">
<input type="button">
<input type="checkbox">
<input type="email">
<input type="number">
<input type="password">
<input type="radio">
```

Example:

```
<!DOCTYPE html>
<html>
<body>
<h1>The input element</h1>
<form action="/action_page.php">
    <label for="fname">First name:</label>
    <input type="text" id="fname" name="fname" placeholder="veda"><br><br>
    <label for="lname">Last name:</label>
    <input type="text" id="lname" name="lname" placeholder="institute"><br><br>
    <label for="lname">Email Id:</label>
    <input
        type="text"
        id="lname"
        name="lname"
        placeholder="vedainstitute@gmail.com"
        ><br>
    <br>
    <label for="lname">Password:</label>
    <input type="text" id="lname" name="lname" placeholder="*****" ><br><br>
<h1>Display Radio Buttons</h1>
<p>Please select your favorite Web language:</p>
    <input type="radio" id="html" name="fav_language" value="HTML">
    <label for="html">HTML</label><br>
    <input type="radio" id="css" name="fav_language" value="CSS">
    <label for="css">CSS</label><br>
    <input type="radio" id="javascript" name="fav_language" value="JavaScript">
    <label for="javascript">JavaScript</label>
<p>checkbox </p>
    <input type="checkbox" id="vehicle1" name="vehicle1" value="Bike">
    <label for="vehicle1"> I have a bike</label><br>
    <input type="checkbox" id="vehicle2" name="vehicle2" value="Car">
    <label for="vehicle2"> I have a car</label><br>
    <input type="checkbox" id="vehicle3" name="vehicle3" value="Boat">
    <label for="vehicle3"> I have a boat</label><br><br>
    <input type="submit" value="Submit">
</form>
</body>
</html>
```

Output:



The screenshot shows a web browser window with multiple tabs open. The active tab displays an HTML form titled "The input element". The form contains four text input fields: "First name: [veda]", "Last name: [institute]", "Email Id: [vedainstitute@gmail.com]", and "Password: [*****]". Below this, a section titled "Display Radio Buttons" asks "Please select your favorite Web language:" and lists three radio button options: "HTML", "CSS", and "JavaScript", all of which are unselected. A checkbox labeled "checkbox" is present, followed by three checkbox options: "I have a bike", "I have a car", and "I have a boat", none of which are checked. At the bottom of the form is a "Submit" button.

12.6 Form Attributes:

This chapter describes the different attributes for the HTML **<form>** element.

Action Attribute:

The **action** attribute defines the action to be performed when the form is submitted.

Target Attribute:

The **target** attribute specifies where to display the response that is received after submitting the form.

Method Attribute:

The **method** attribute specifies the HTTP method to be used when submitting the form data.

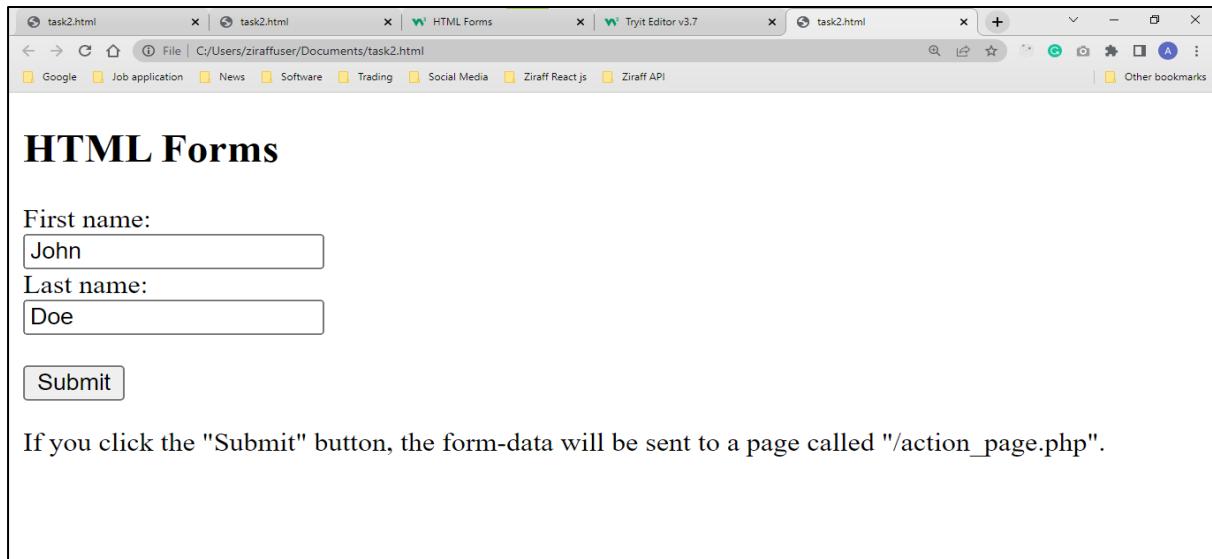
Example:

```
<!DOCTYPE html>

<html>
<body>
<h2>HTML Forms</h2>
<form action="/action_page.php">
<label for="fname">First name:</label><br>
<input type="text" id="fname" name="fname" value="John"><br>
<label for="lname">Last name:</label><br>
<input type="text" id="lname" name="lname" value="Doe"><br><br>
<input type="submit" value="Submit">
</form>

<p>If you click the "Submit" button, the form-data will be sent to a page called "/action_page.php".</p>
</body>
</html>
```

Output:



task2.html task2.html HTML Forms Tryit Editor v3.7 task2.html

File | C:/Users/ziraffuser/Documents/task2.html

Google Job application News Software Trading Social Media Ziraff React.js Ziraff API Other bookmarks

HTML Forms

First name:

Last name:

If you click the "Submit" button, the form-data will be sent to a page called "/action_page.php".

CSS

1.CSS Introduction

CSS is the language we use to style a Web page.

1.1 What is CSS?

- CSS stands for **Cascading Style Sheets**
- CSS describes how HTML elements are to be displayed on screen, paper, or in other media
- CSS saves a lot of work. It can control the layout of multiple web pages all at once
- External stylesheets are stored in CSS files

1.2 Why Use CSS?

CSS is used to define styles for your web pages, including the design, layout and variations in display for different devices and screen sizes.

Syntax:

```
Element {Style="property Name:Property value;"}
```

2.Types of Style Sheets

2.1 Inline Style Sheets:

An inline CSS is used to apply a unique style to a single HTML element.

An inline CSS uses the **style** attribute of an HTML element.

The following example sets the text color of the **<h1>** element to blue, and the text color of the **<p>** element to red:

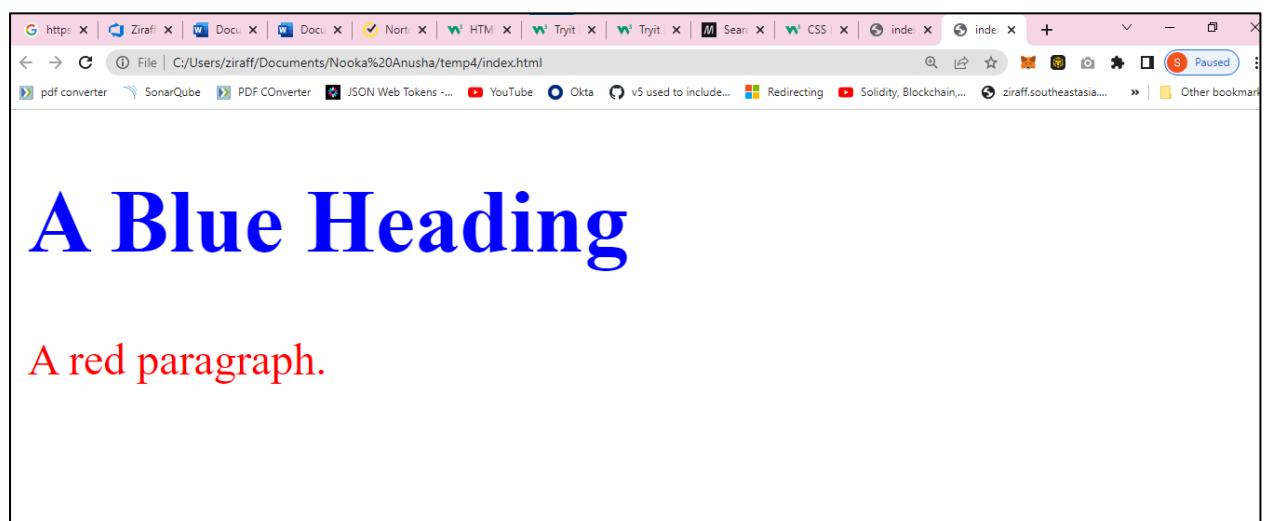
Syntax:

```
<h1 (elements) Style="property Name:Property value;"></h1>
```

Example:

```
<!DOCTYPE html>

<html>
<body>
<h1 style="color: blue;">A Blue Heading</h1>
<p style="color: red;">A red paragraph.</p>
</body>
</html>
```

Output:

2.2 Internal CSS

An internal CSS is used to define a style for a single HTML page.

An internal CSS is defined in the **<head>** section of an HTML page, within a **<style>** element.

The following example sets the text color of ALL the **<h1>** elements (on that page) to blue, and the text color of ALL the **<p>** elements to red. In addition, the page will be displayed with a "powderblue" background color:

Syntax:

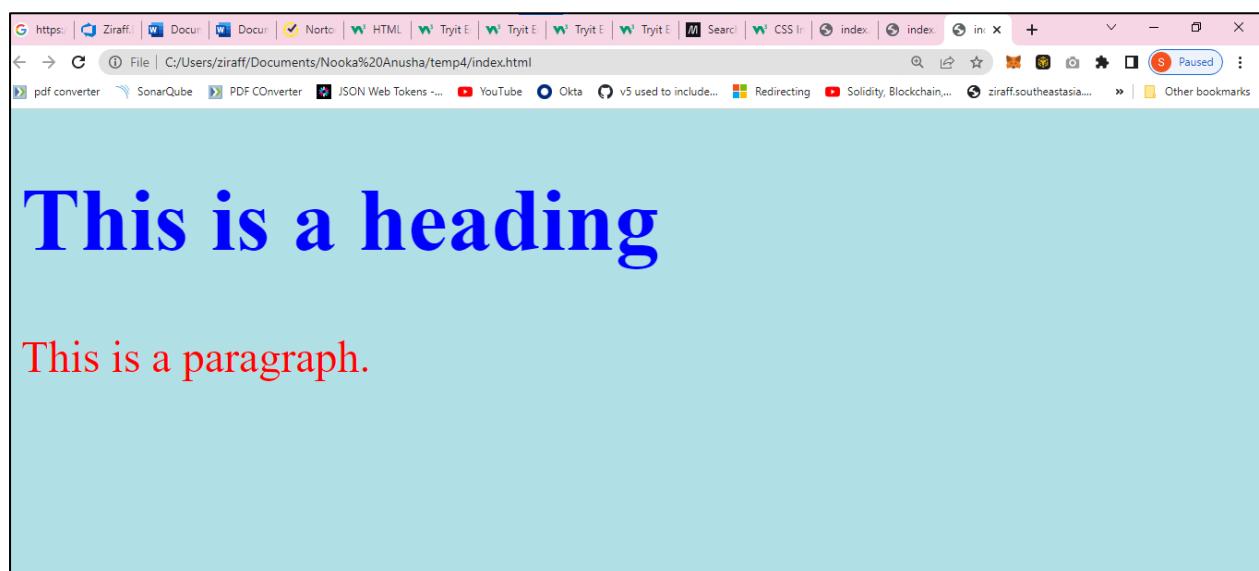
```
<head>
<style>
element {property Name:Property Value;}
</style>
</head>
```

Example:

```
<!DOCTYPE html>

<html>
<head>
<style>
body {background-color: powderblue;}
h1{color: blue;}
p{color: red;}
</style>
</head>
<body>
<h1>This is a heading</h1>
<p>This is a paragraph. </p>
</body>
</html>
```

Output:



The screenshot shows a web browser window displaying the output of the provided HTML code. The browser's address bar shows the file path: C:/Users/ziraff/Documents/Nooka%20Anusha/temp4/index.html. The page content consists of a single heading and a single paragraph. The heading is rendered in blue text, and the paragraph is rendered in red text, both matching the styles defined in the CSS within the original code.

2.3 External Style Sheets

An external style sheet is used to define the style for many HTML pages.

To use an external style sheet, add a link to it in the **<head>** section of each HTML page:

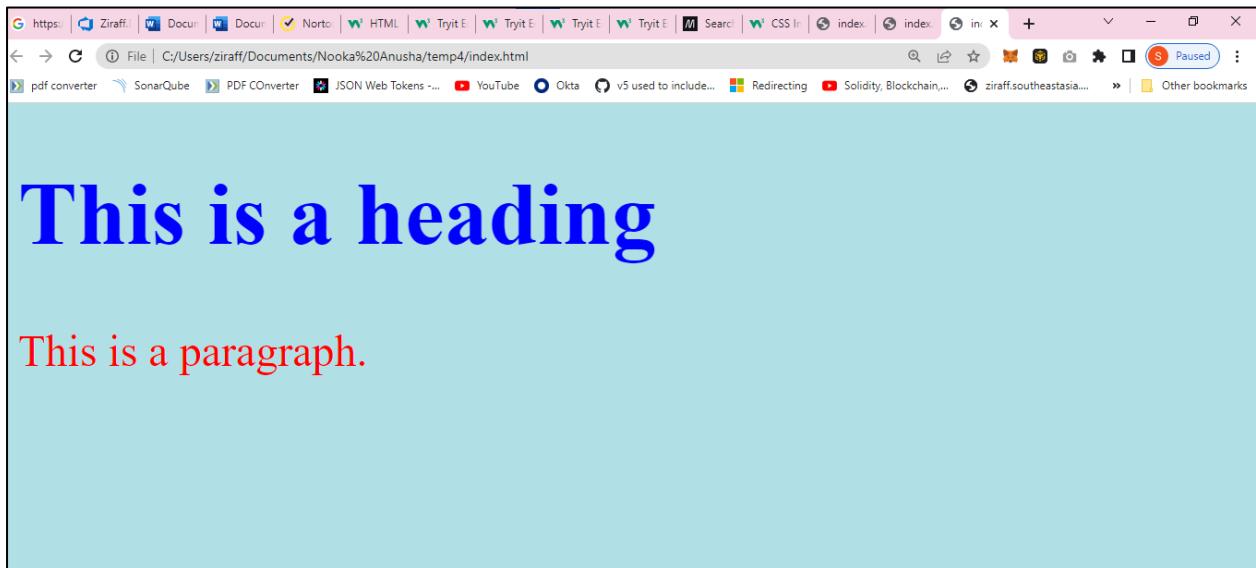
Syntax:

```
<head>
  <link rel = "stylesheet" href = "styles.css">
</head>
```

Example:

```
<!DOCTYPE html>
<html>
<head>
  <link rel = "stylesheet" href = "styles.css">
</head>
<body>
<h1>This is a heading</h1>
<p>This is a paragraph. </p>
</body>
</html>
```

Output:



3.Types of Selectors:

Simple selectors (select elements based on name, id, class)

3.1 Class Selectors:

- The class selector selects HTML elements with a specific class attribute.
- To select elements with a specific class, write a period (.) character, followed by the class name.

Syntax:

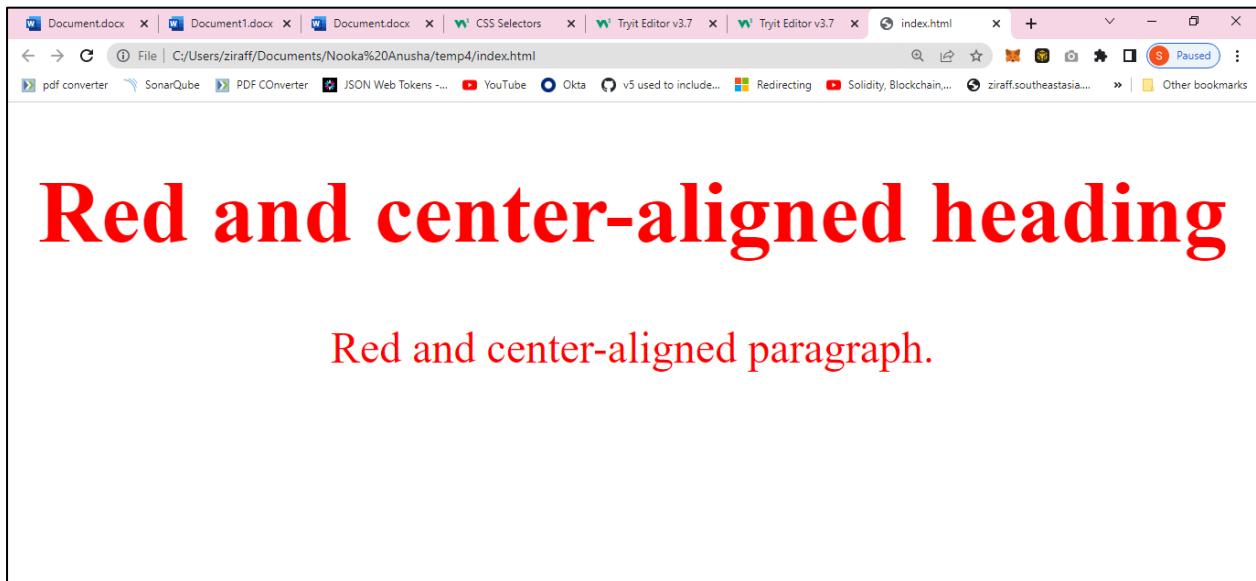
```
.center {text-align: center; color: red;}
```

Example:

```
<!DOCTYPE html>

<html>
<head>
<style>
.center {
    text-align: center;
    color: red;
}
</style>
</head>
<body>
<h1 class="center">Red and center-aligned heading</h1>
<p class="center">Red and center-aligned paragraph.</p>
</body>
</html>
```

Output:



3.2 Id Selectors:

- The id selector uses the id attribute of an HTML element to select a specific element.
- The id of an element is unique within a page, so the id selector is used to select one unique element!
- To select an element with a specific **id**, write a hash (#) character, followed by the id of the element.

Syntax:

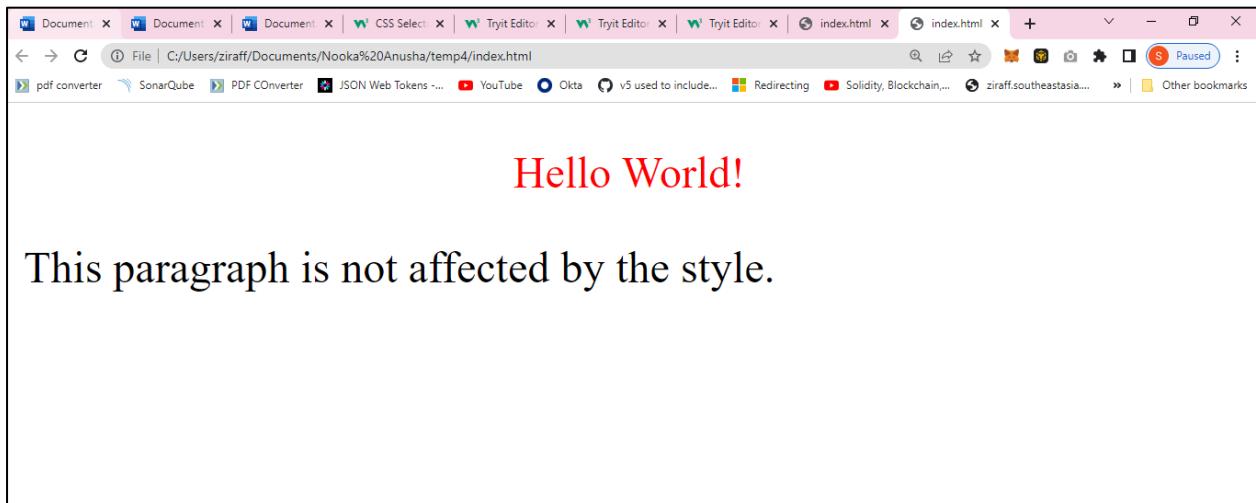
```
#para1 {text-align: center; color: red;}
```

Example:

```
<!DOCTYPE html>

<html>
<head>
<style>
#para1 {
    text-align: center;
    color: red;
}
</style>
</head>
<body>
<p id="para1">Hello World! </p>
<p>This paragraph is not affected by the style. </p>
</body>
</html>
```

Output:



3.3 Element Selectors:

- The element selector selects HTML elements based on the element name.

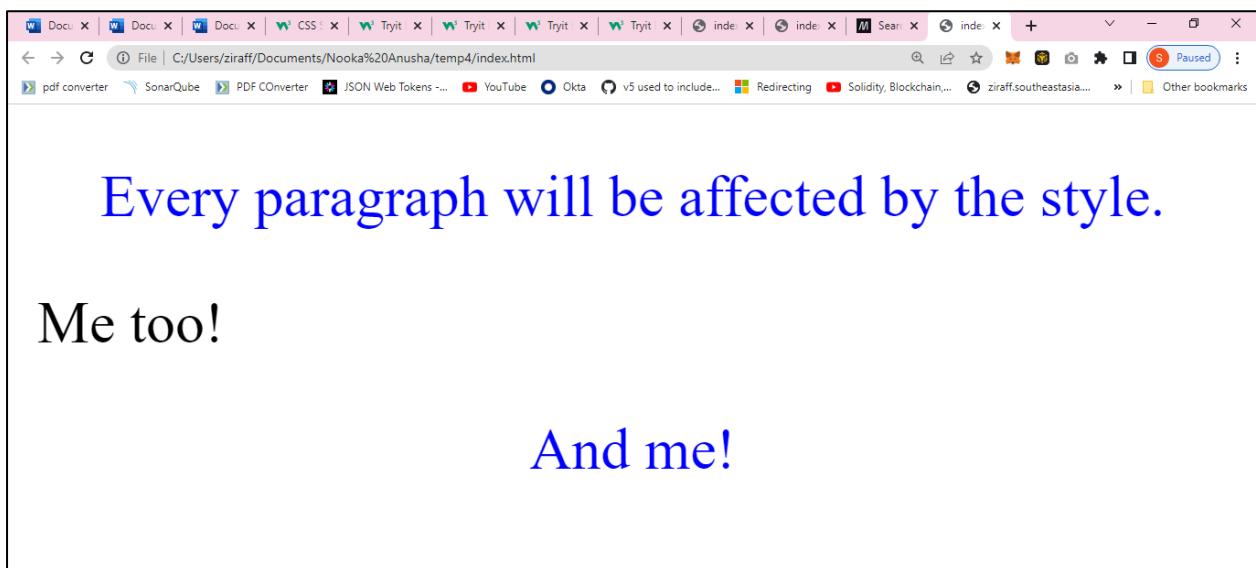
Syntax:

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

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
p {
    text-align: center;
    color: blue;
}
</style>
</head>
<body>
<p>Every paragraph will be affected by the style. </p>
<div>Me too! </div>
<p>And me! </p>
</body>
</html>
```

Output:



4. Comments:

- Comments are used to explain the code, and may help when you edit the source code at a later date.
- Comments are ignored by browsers.
- A CSS comment is placed inside the `<style>` element, and starts with `/*` and ends with `*/`:

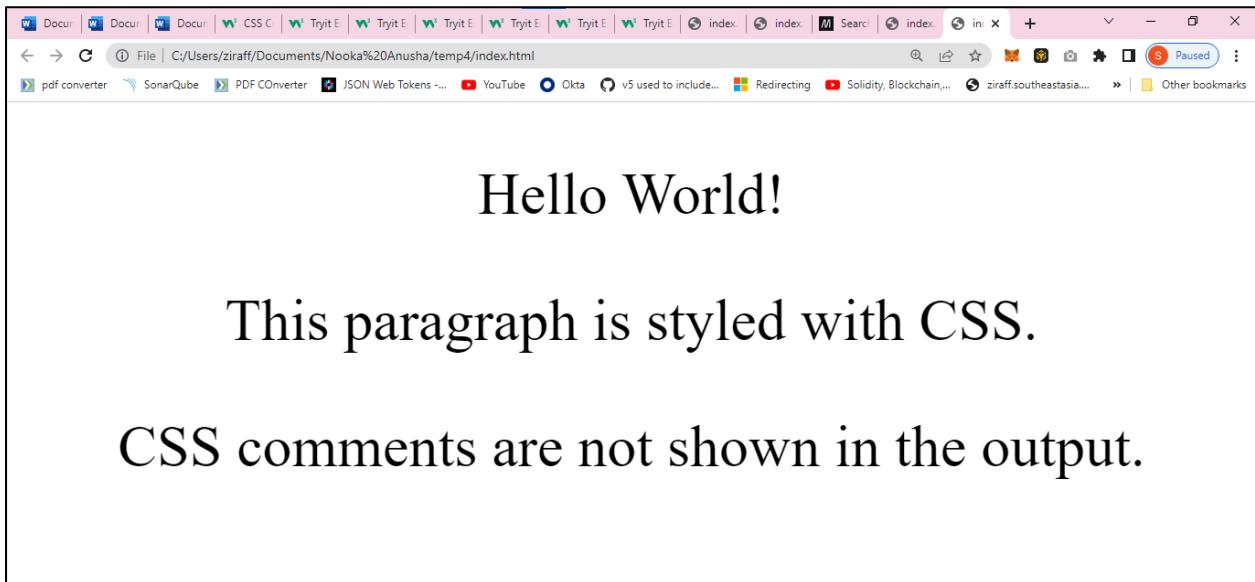
Syntax:

```
/*Veda institute*/
```

Example:

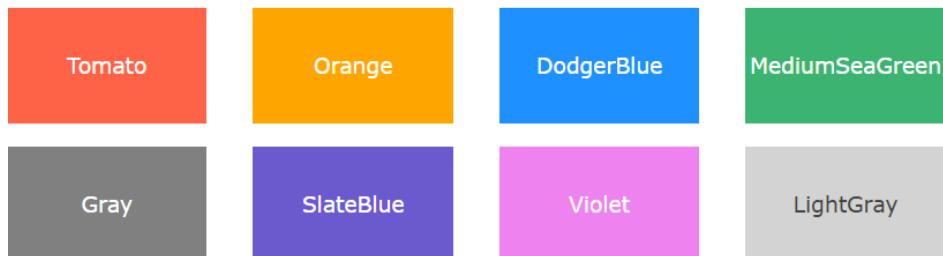
```
<!DOCTYPE html>
<html>
<head>
<style>
p {
/*color: red; */
text-align: center;
}
</style>
</head>
<body>
<p>Hello World! </p>
<p>This paragraph is styled with CSS. </p>
<p>CSS comments are not shown in the output. </p>
</body>
</html>
```

Output:



5. Colors:

In CSS, a color can be specified by using a predefined color name:



5.1 RGB Colors:

An RGB color value represents RED, GREEN, and BLUE light sources. **RGB Value:**

To display black, set all color parameters to 0, like this: `rgb (0, 0, 0)`.

To display white, set all color parameters to 255, like this: `rgb (255, 255, 255)`.

rgb(255, 0, 0)**rgb(0, 0, 255)****rgb(60, 179, 113)****rgb(238, 130, 238)****rgb(255, 165, 0)****rgb(106, 90, 205)**

5.2 HEX Colors:

A hexadecimal color is specified with: #RRGGBB, where the RR (red), GG (green) and BB (blue) hexadecimal integers specify the components of the color.

HEX Value:

#rrggbb

Where rr (red), gg (green) and bb (blue) are hexadecimal values between 00 and ff (same as decimal 0-255).

For example, #ff0000 is displayed as red, because red is set to its highest value (ff) and the others are set to the lowest value (00).

- To display black, set all values to 00, like this: **#000000**
- To display white, set all values to ff, like this: **#ffff**

#ff0000

#0000ff

#3cb371

#ee82ee

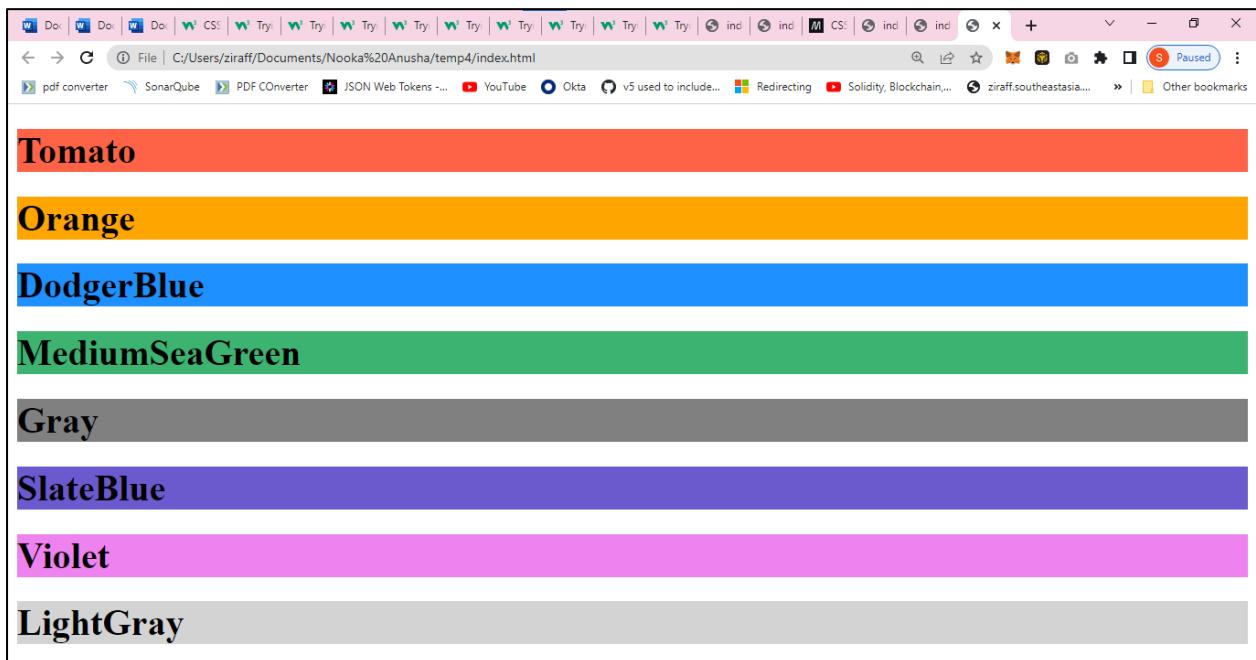
#ffa500

#6a5acd

Example:

```
<!DOCTYPE html>
<html>
<body>
<h1 style="background-color: Tomato;">Tomato</h1>
<h1 style="background-color: Orange;">Orange</h1>
<h1 style="background-color: DodgerBlue;">Dodger Blue</h1>
<h1 style="background-color: MediumSeaGreen; ">MediumSeaGreen</h1>
<h1 style="background-color: Gray;">Gray</h1>
<h1 style="background-color: SlateBlue;">Slate Blue</h1>
<h1 style="background-color: Violet;">Violet</h1>
<h1 style="background-color: Light Gray;">LightGray</h1>
</body>
</html>
```

Output:



6. Box Model

- In CSS, the term "box model" is used when talking about design and layout.
- The CSS box model is essentially a box that wraps around every HTML element. It consists of: margins, borders, padding, and the actual content. The image below illustrates the box model:

Example:



6.1 Margin

The **margin CSS** shorthand property sets the **margin area** on all four sides of an element.

CSS margin Property

- margin-top
- margin-right
- margin-bottom
- margin-left

1. **margin: 10px 5px 15px 20px**

- top margin is 10px
- right margin is 5px
- bottom margin is 15px
- left margin is 20px

2. **margin: 10px 5px 15px**

- top margin is 10px
- right and left margins are 5px
- bottom margin is 15px

3. **margin: 10px 5px**

- top and bottom margins are 10px
- right and left margins are 5px

4. **margin: 35px**

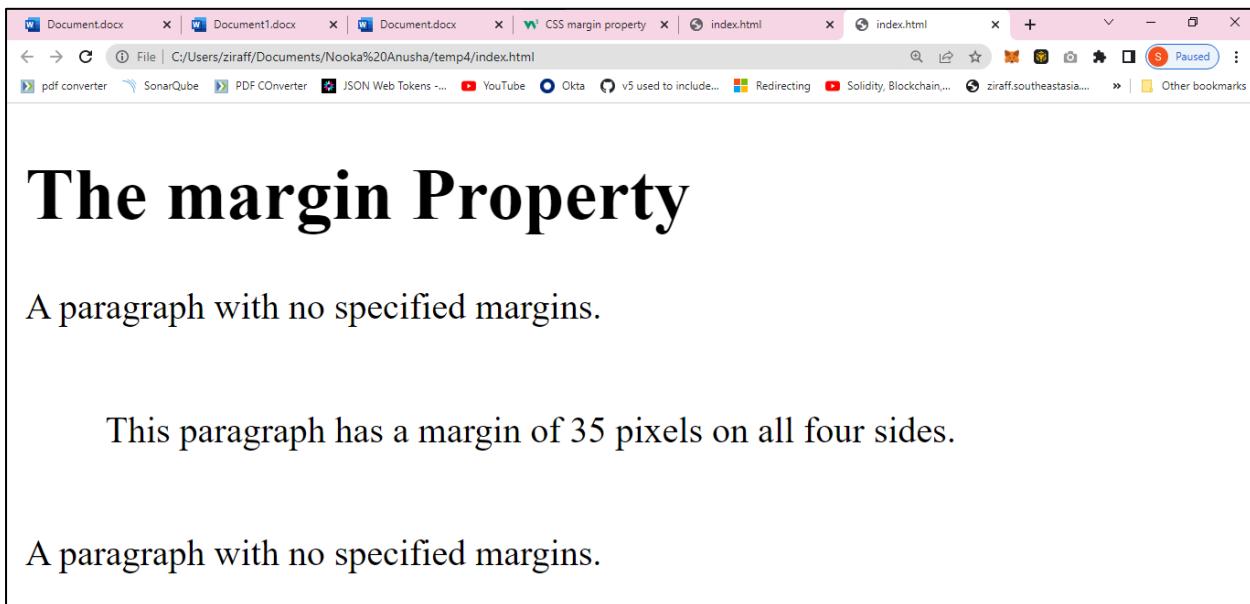
- all four margins are 35px

Example:

```
<!DOCTYPE html>

<html>
<head>
<style>
p.ex1 {
  margin: 35px;
}
</style>
</head>
<body>
<h1>The margin Property</h1>
<p>A paragraph with no specified margins. </p>
<p class="ex1">This paragraph has a margin of 35 pixels on all four sides. </p>
<p>A paragraph with no specified margins. </p>
</body>
</html>
```

Output:



The margin Property

A paragraph with no specified margins.

This paragraph has a margin of 35 pixels on all four sides.

A paragraph with no specified margins.

6.2 Border

The CSS border properties allow you to specify the style, width, and color of an element's border.

Border Style: The border style have property values dotted, dashed, solid, double, groove, ridge, inset, outset, none, hidden

Ex: border-style: solid;

Border Width:

The **border-width** property specifies the width of the four borders.

The width can be set as a specific size (in px, pt, cm, em, etc) or by using one of the three pre-defined values: thin, medium, or thick

Ex: border-width: 5px;

Border Color:

The **border-color** property is used to set the color of the four borders.

- name - specify a color name, like "red"
- HEX - specify a HEX value, like "#ff0000"
- RGB - specify a RGB value, like "rgb (255,0,0)"
- HSL - specify a HSL value, like "hsl(0, 100%, 50%)"
- Transparent.

Border - Shorthand Property:

```
border-top: 6px solid red;  
border-right: 6px solid red;  
border-bottom: 6px solid red;  
border-left: 6px solid red;
```

Rounded Borders

The **border-radius** property is used to add rounded borders to an element

Ex: border: 2px solid red;
border-radius: 5px;



border-radius: 50px;

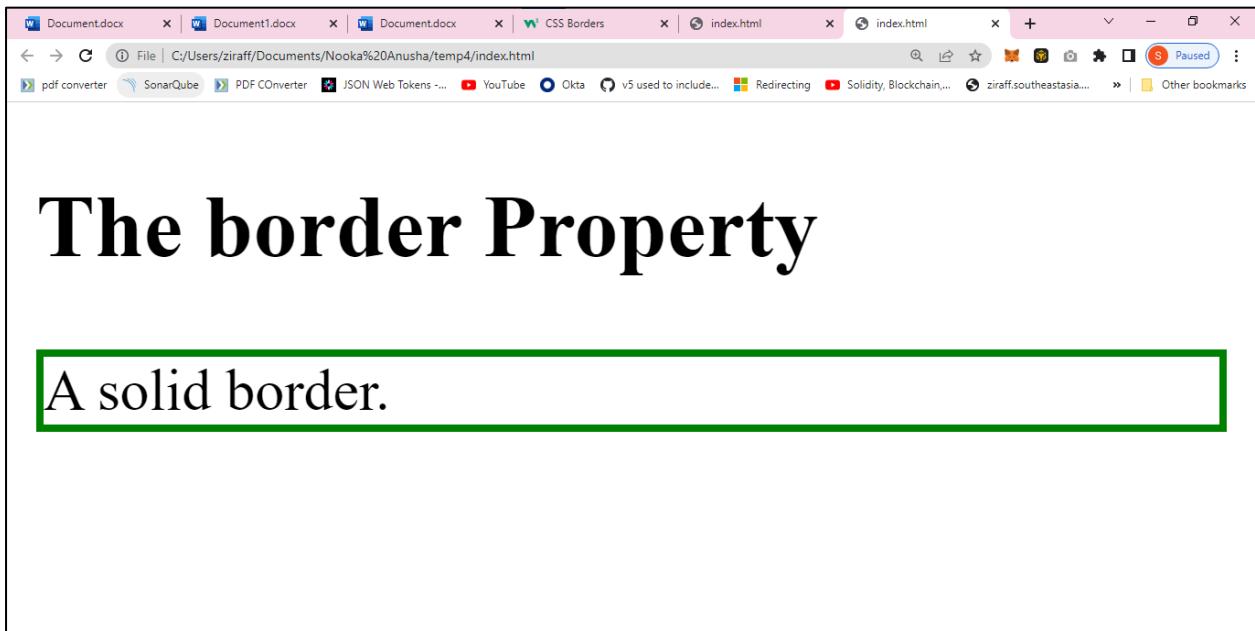


Example:

```
<!DOCTYPE html>

<html>
<head>
<style>
p. solid {border:2px solid green;}
</style>
</head>
<body>
<h2>The border Property</h2>
<p class="solid">A solid border.</p>
</body>
</html>
```

Output:



6.3 Padding

Padding is used to create space around an element's content, inside of any defined borders.

CSS Padding Property

- padding-top
- padding-right
- padding-bottom
- padding-left

1.padding: 25px 50px 75px 100px

- top padding is 25px
- right padding is 50px
- bottom padding is 75px
- left padding is 100px

2.padding: 25px 50px 75px

- top padding is 25px
- right and left paddings are 50px
- bottom padding is 75px

3.padding: 25px 50px

- top and bottom paddings are 25px
- right and left paddings are 50px

4.padding: 25px

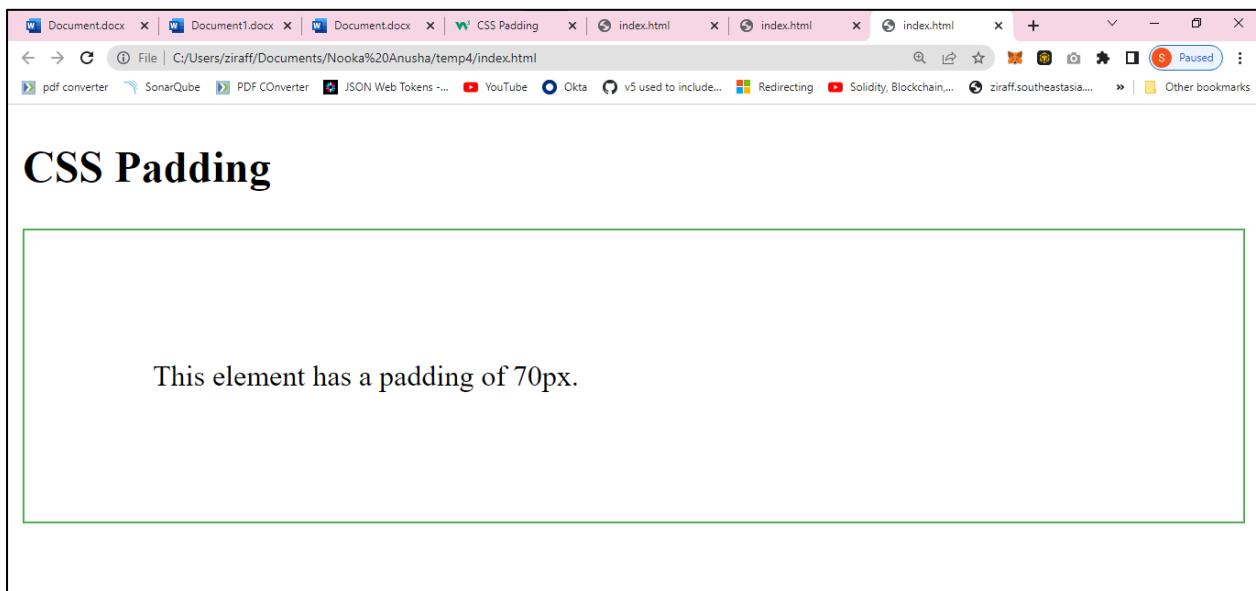
- all four paddings are 25px

Example:

```
<!DOCTYPE html>

<html>
<head>
<style>
div {
  padding: 70px;
  border: 1px solid #4CAF50;
}
</style>
</head>
<body>
<h2>CSS Padding</h2>
<div>This element has a padding of 70px. </div>
</body>
</html>
```

Output:

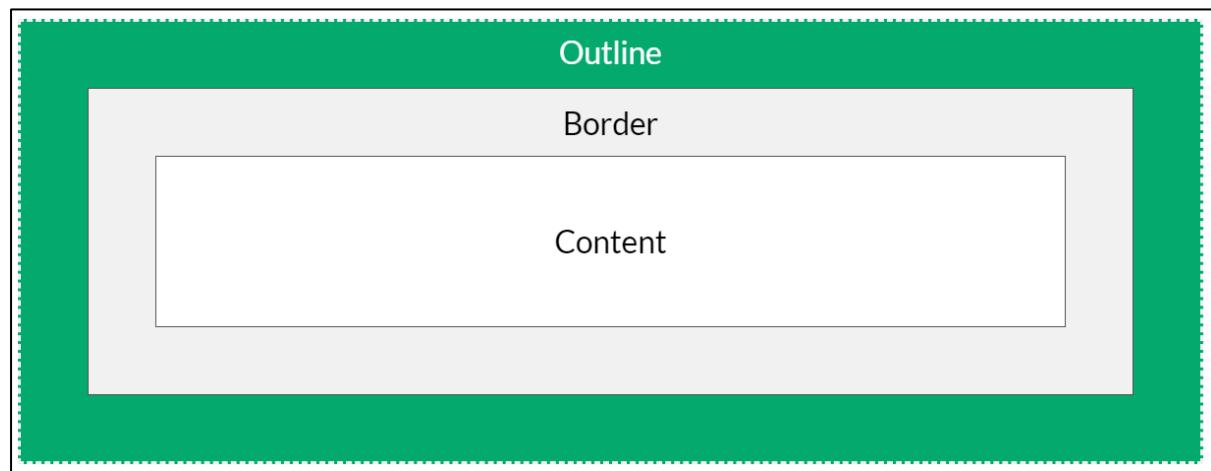


6.4 Content

The content of the box, where text and images appear.

7. Outline:

An outline is a line that is drawn around elements, OUTSIDE the borders, to make the element "stand out".



CSS has the following outline properties:

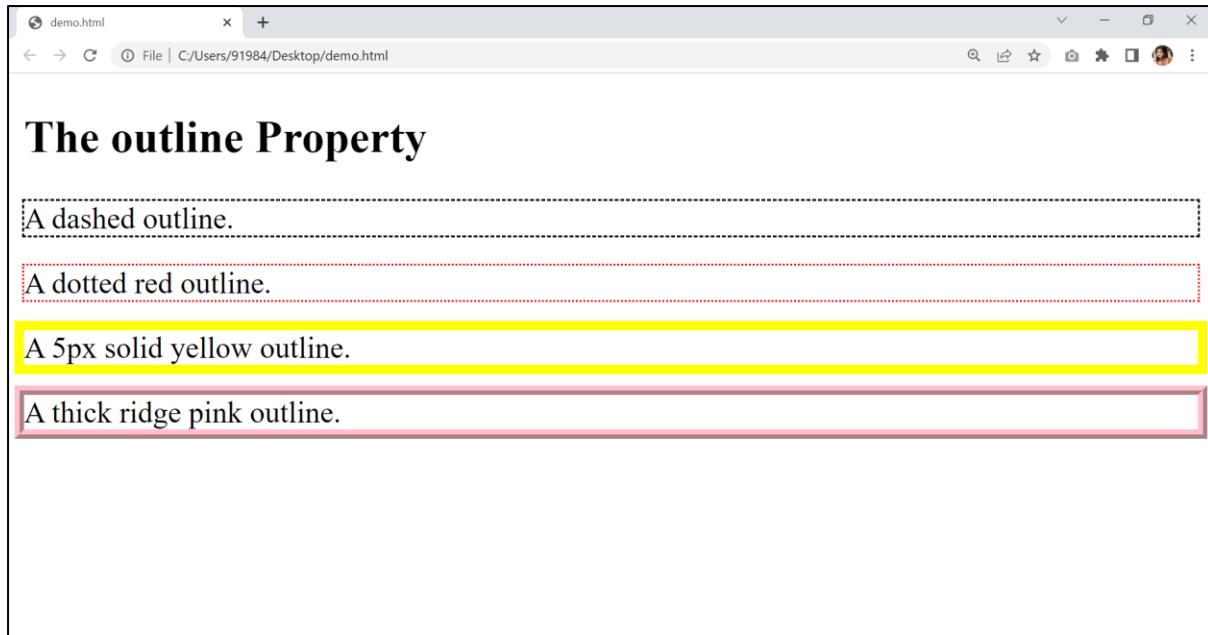
- outline-style
- outline-color
- outline-width
- outline-offset
- outline

Note: Outline differs from borders! Unlike border, the outline is drawn outside the element's border, and may overlap other content. Also, the outline is NOT a part of the element's dimensions; the element's total width and height is not affected by the width of the outline.

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
p.ex1 {outline: dashed;}
p.ex2 {outline: dotted red;}
p.ex3 {outline: 5px solid yellow;}
p.ex4 {outline: thick ridge pink;}
</style>
</head>
<body>
<h2>The outline Property</h2>
<p class="ex1">A dashed outline.</p>
<p class="ex2">A dotted red outline.</p>
<p class="ex3">A 5px solid yellow outline.</p>
<p class="ex4">A thick ridge pink outline.</p>
</body>
</html>
```

Output:



Note: None of the other outline properties (which you will learn more about in the next chapters) will have ANY effect unless the outline-style property is set!

8. Fonts

- Choosing the right font has a huge impact on how the readers experience a website.
- The right font can create a strong identity for your brand.
- Using a font that is easy to read is important. The font adds value to your text. It is also important to choose the correct color and text size for the font.

8.1 font-family:

1. **Serif** fonts have a small stroke at the edges of each letter. They create a sense of formality and elegance.
2. **Sans-serif** fonts have clean lines (no small strokes attached). They create a modern and minimalistic look.
3. **Monospace** fonts - here all the letters have the same fixed width. They create a mechanical look.
4. **Cursive** fonts imitate human handwriting.
5. **Fantasy** fonts are decorative/playful fonts.

Ex: body {font-family: Calibri, sans-serif;}

8.2 color:

The **color** CSS property sets the foreground **color value** of an element's text and **text decorations**, and sets the **currentcolor** value. Current color may be used as an indirect value on *other* properties and is the default for other color properties, such as **border-color**.

Ex: `p{color:red;}`

8.3 Font-Size:

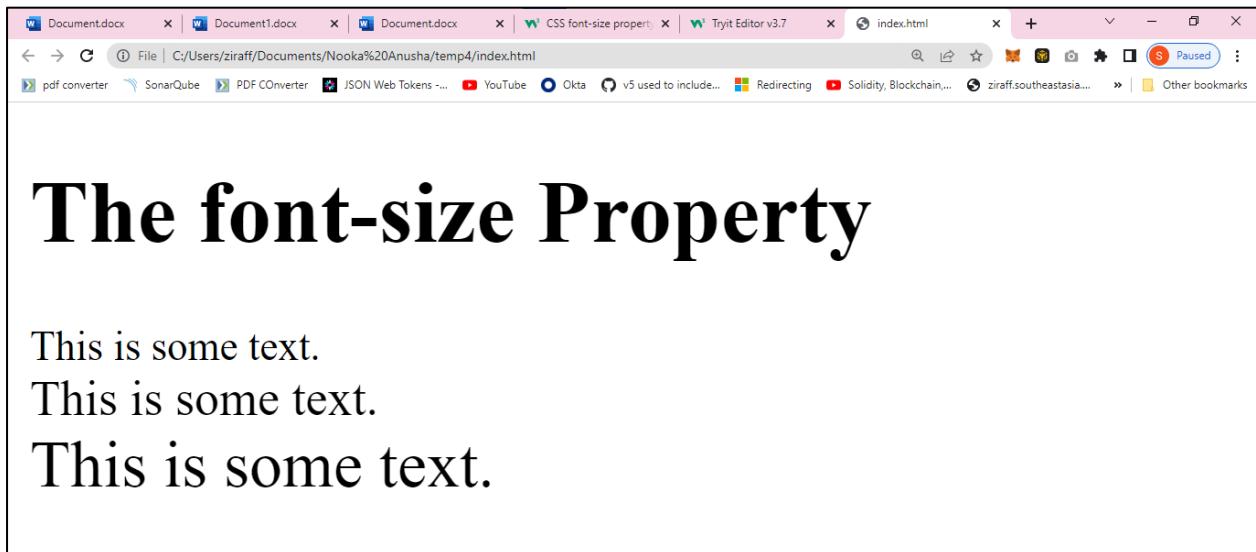
- The **font-size** property sets the size of a font.

Example:

```
<!DOCTYPE html>

<html>
<head>
<style>
div.a {
    font-size: 15px;
}
div.b {
    font-size: large;
}
div.c {
    font-size: 150%;
}
</style>
</head>
<body>
<h1>The font-size Property</h1>
<div class="a">This is some text.</div>
<div class="b">This is some text.</div>
<div class="c">This is some text.</div>
</body>
</html>
```

Output:



8.4 Font-Style:

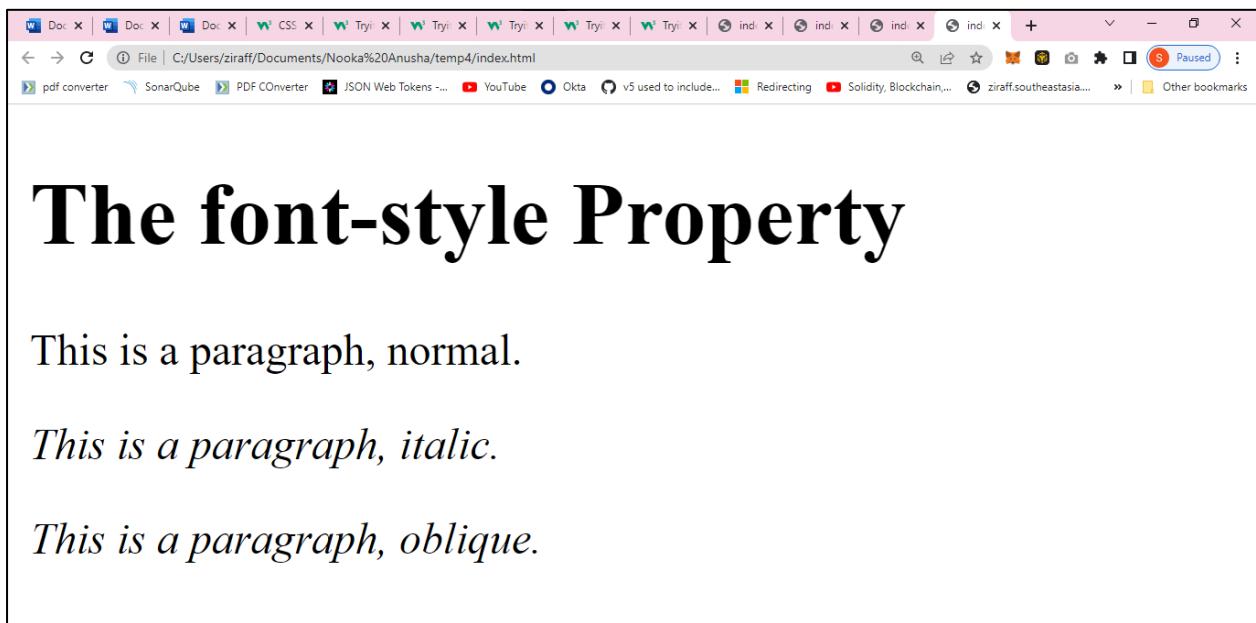
- Set different font styles like normal, italic, oblique.

Example:

```
<!DOCTYPE html>

<html>
<head>
<style>
p.a {
    font-style: normal;
}
p.b {
    font-style: italic;
}
p.c {
    font-style: oblique;
}
</style>
</head>
<body>
<h1>The font-style Property</h1>
<p class="a">This is a paragraph, normal . </p>
<p class="b">This is a paragraph, italic. </p>
<p class="c">This is a paragraph, oblique. </p>
</body>
</html>
```

Output:



The screenshot shows a browser window with multiple tabs open, all related to "Try" and "ind". The main content area displays three paragraphs demonstrating the `font-style` property:

- A large, bold heading: **The font-style Property**.
- A regular paragraph:

This is a paragraph, normal.
- An italicized paragraph:

This is a paragraph, italic.
- An oblique paragraph:

This is a paragraph, oblique.

8.5 font-weight

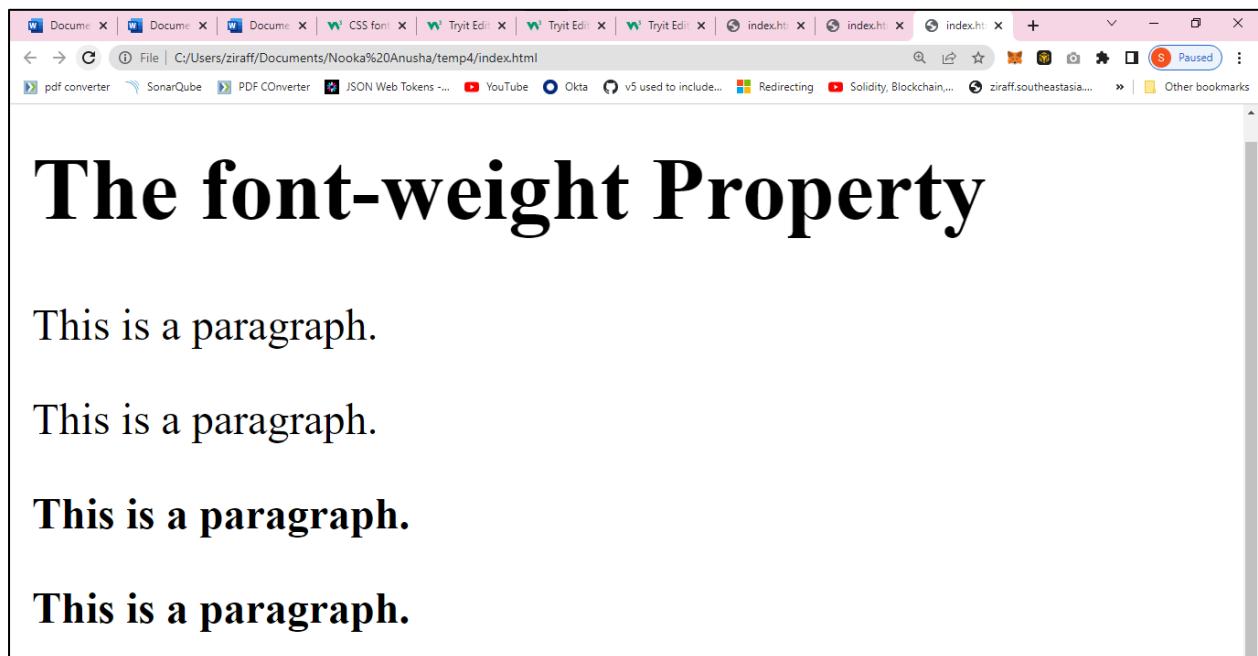
- Set different font weights like bold, bolder, normal and from 100 to 900.

Example:

```
<!DOCTYPE html>

<html>
<head>
<style>
p.normal {
    font-weight: normal;
}
p. light {
    font-weight: lighter;}
p. thick {
    font-weight: bold;}
p. thicker {
    font-weight: 900;}
</style>
</head>
<body>
<h1>The font-weight Property </h1>
<p class="normal">This is a paragraph. </p>
<p class="light">This is a paragraph. </p>
<p class="thick">This is a paragraph. </p>
<p class="thicker">This is a paragraph. </p>
</body>
</html>
```

Output:



8.6 Google Fonts:

If you do not want to use any of the standard fonts in HTML, you can use Google Fonts. Google Fonts are free to use, and have more than 1000 fonts to choose from.

8.6.1 How to Use Google Fonts:

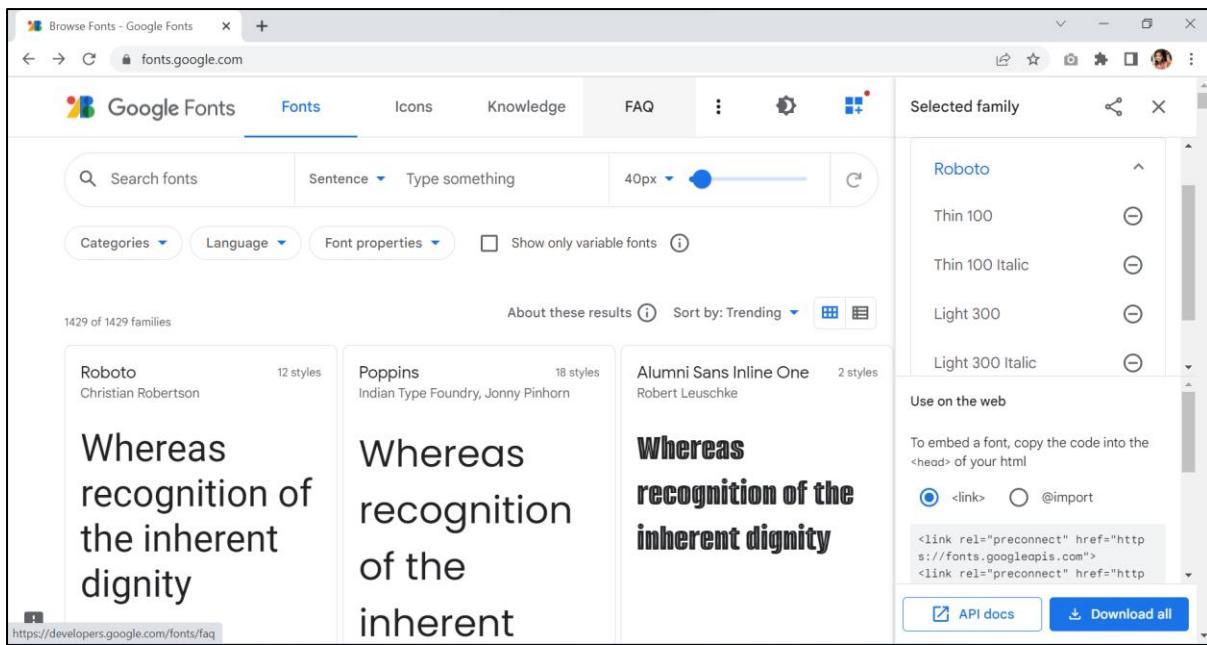
Just add a special style sheet **link** in the <head> section or **import** style from google fonts into external css file and then refer to the font in the CSS.

Ex:

```
<link  
href="https://fonts.googleapis.com/css2?family=Roboto:ital,wght@0,100;0,300;1,100;1,3  
00&display=swap" rel="stylesheet">
```

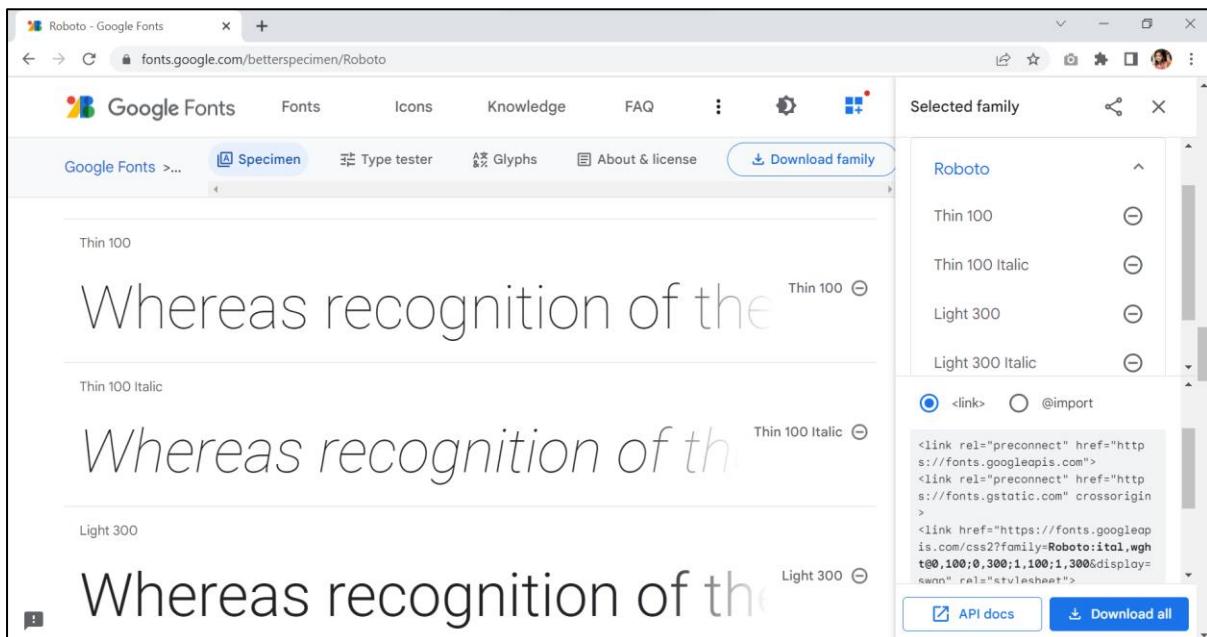
The above link can be get from google font website. Process for get above link as follows

1. Go to browser and type google fonts then click on Google fonts(<https://fonts.google.com>)
2. It shows different font style like Roboto, Poppins, Alumni Sans Inline One etc...



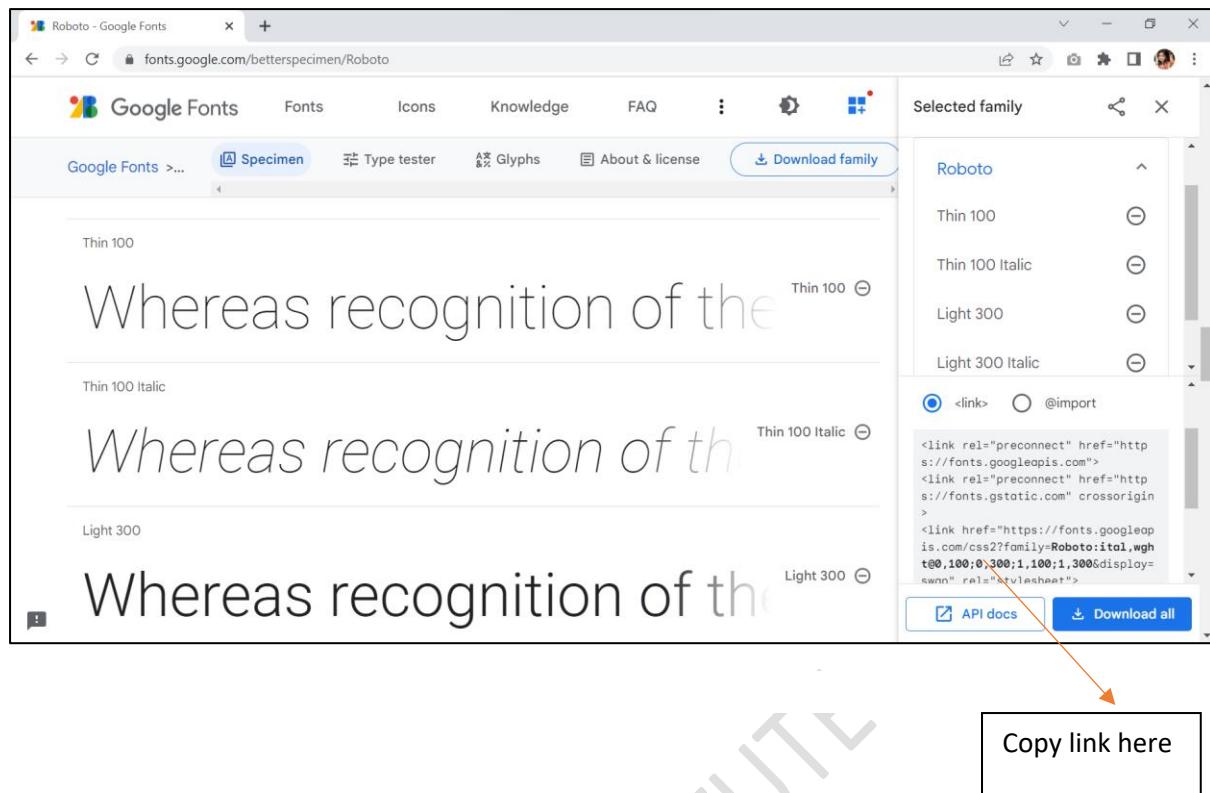
The screenshot shows the Google Fonts website with the Roboto font family selected. The left sidebar lists categories like 'Google Fonts', 'Fonts', 'Icons', 'Knowledge', 'FAQ', and a settings gear icon. The main content area shows a search bar with 'Search fonts' and a dropdown set to '40px'. Below it are buttons for 'Categories', 'Language', 'Font properties', and 'Show only variable fonts'. A preview section displays the text 'Whereas recognition of the inherent dignity' in different font weights: 'Thin 100' (thin), 'Thin 100 Italic' (thin italic), 'Light 300' (light), and 'Light 300 Italic' (light italic). The right sidebar is titled 'Selected family' and lists the Roboto font family with its sub-weights. It also includes sections for 'Use on the web' with code snippets for <link> and @import, and buttons for 'API docs' and 'Download all'.

3. We select whatever font we required. Here we select Roboto font family with different font-weights and font-styles like **Thin 100, Thin 100 italic etc.....**



The screenshot shows the Roboto font specimen page on Google Fonts. The top navigation bar includes 'Google Fonts', 'Fonts', 'Icons', 'Knowledge', 'FAQ', and a settings gear icon. The 'Specimen' tab is selected. The main content area displays the text 'Whereas recognition of the inherent dignity' in three different styles: 'Thin 100' (thin), 'Thin 100 Italic' (thin italic), and 'Light 300' (light). The 'Download family' button is located at the top right of the main content area. The right sidebar is identical to the one in the previous screenshot, showing the 'Selected family' list for Roboto and options for embedding the font.

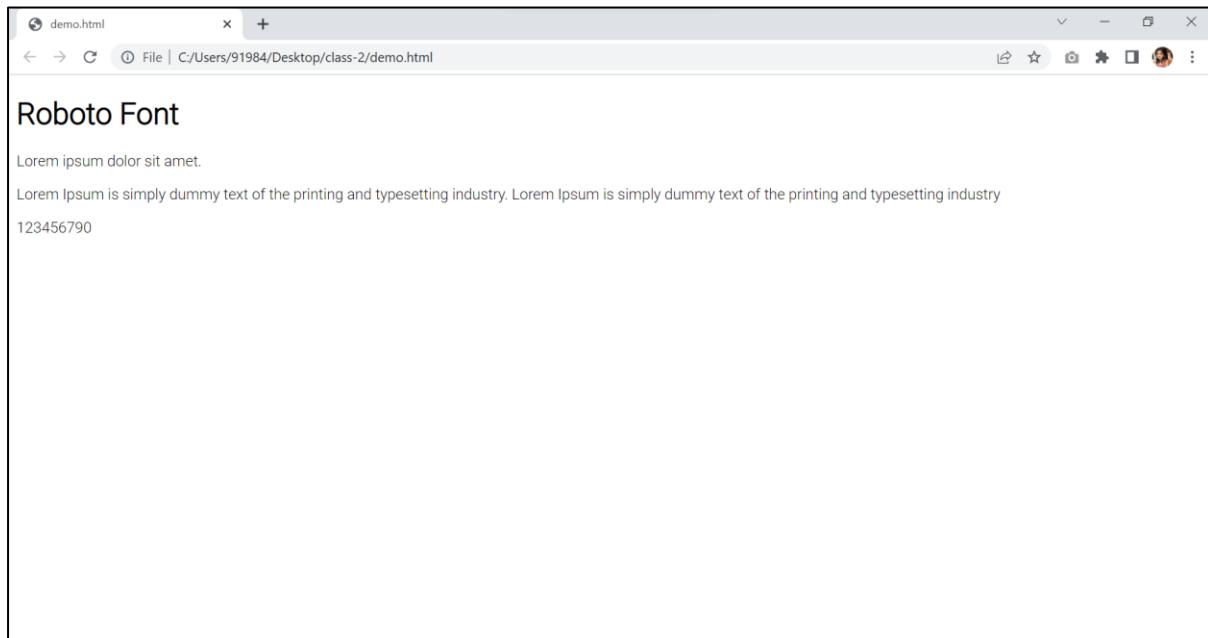
4. We can use link tag to link google font to our html file in header section. Copy the link tag as shown in below



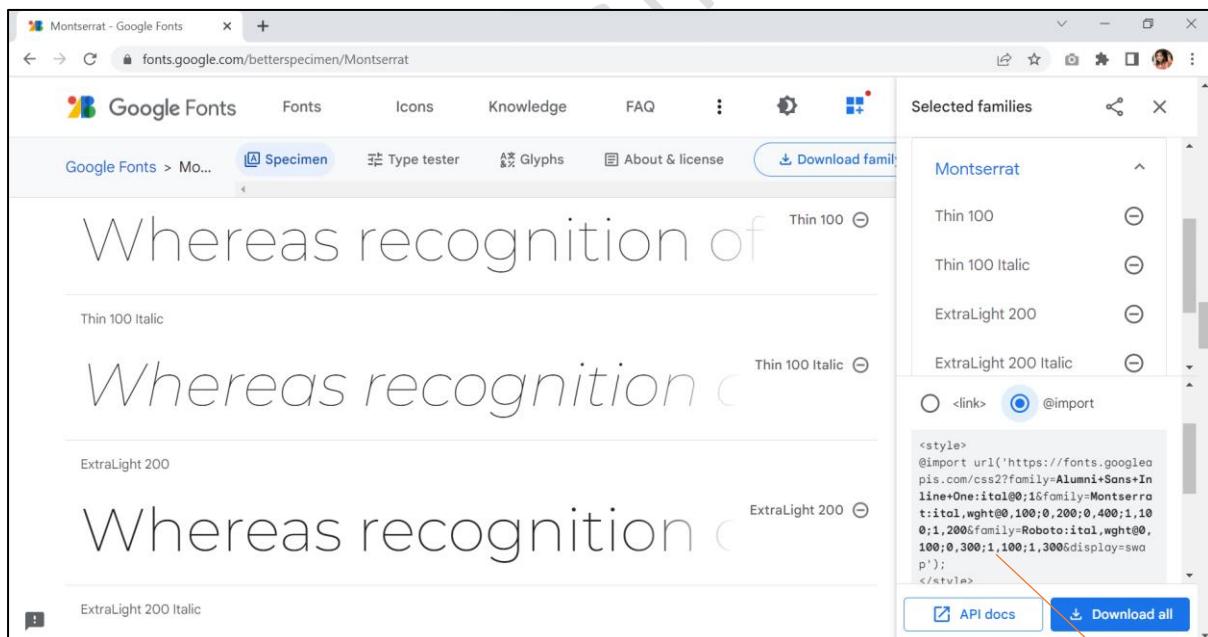
Example:

```
<!DOCTYPE html>
<html>
<head>
<link
  href="https://fonts.googleapis.com/css2?family=Roboto:ital,wght@0,100
;0,300;1,100;1,300&display=swap" rel="stylesheet">
<style>
body {
  font-family: "Roboto", sans-serif;
}
</style>
</head>
<body>
<h1>Roboto Font</h1>
<p>Lorem ipsum dolor sit amet.</p>
<p>123456790</p>
</body>
</html>
```

Output:



6. We can also apply font family to our html file using **@import** into our external css or copy **@import** link into **<style>** tag in head section.



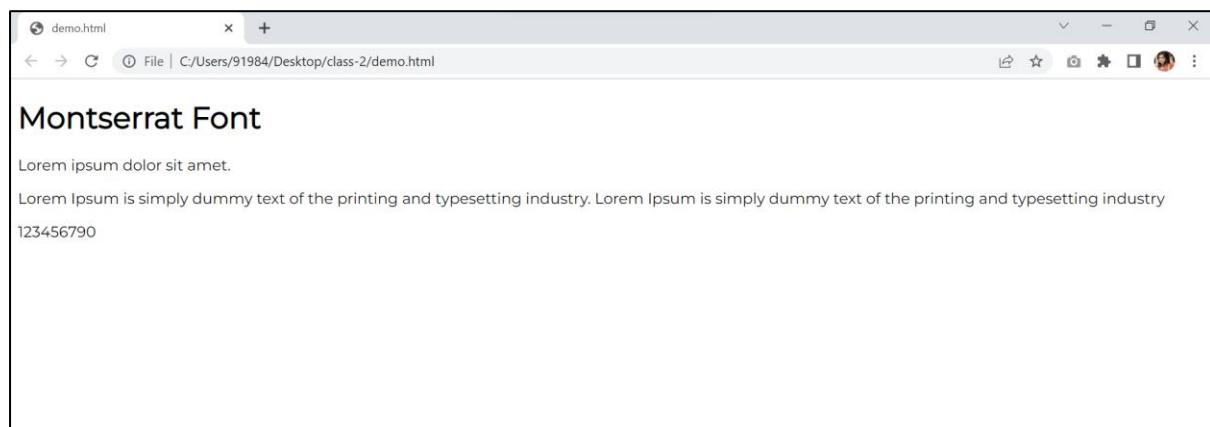
The screenshot shows the Google Fonts website for the Montserrat font. On the left, there's a specimen view of the font in various styles: Thin 100, Thin 100 Italic, ExtraLight 200, and ExtraLight 200 Italic. On the right, a sidebar titled "Selected families" lists the Montserrat font with its sub-variants. Below the sidebar, there are two radio button options: "<link>" (unchecked) and "@import" (checked). A code snippet for an @import rule is displayed, which includes the URL for the font's CSS file and the font-family name. At the bottom right of the sidebar, there are "API docs" and "Download all" buttons. An orange arrow points from a callout box at the bottom right towards the "@import" radio button.

Copy above @import link

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
@import
url('https://fonts.googleapis.com/css2?family=Roboto:ital,wght@0,100;0,
300;1,100;1,300&display=swap');
body {
font-family: " Montserrat ", sans-serif;
}
</style>
</head>
<body>
<h1>Montserrat Font</h1>
<p>Lorem ipsum dolor sit amet. </p>
<p>Lorem Ipsum is simply dummy text of the printing and typesetting
industry. Lorem Ipsum is simply dummy text of the printing and
typesetting industry</p>
<p>123456790</p>
</body>
</html>
```

Output:



9.Text:

CSS has a lot of properties for formatting text. They are

- Text Color
- Text Alignment
- Text Decoration
- Text Transformation
- Text Spacing
- Text Shadow

9.1 Text Color:

The **color** property is used to set the color of the text. The color is specified by:

- a color name - like "red"
- a HEX value - like "#ff0000"
- an RGB value - like "rgb (255,0,0)"

The default text color for a page is defined in the body selector.

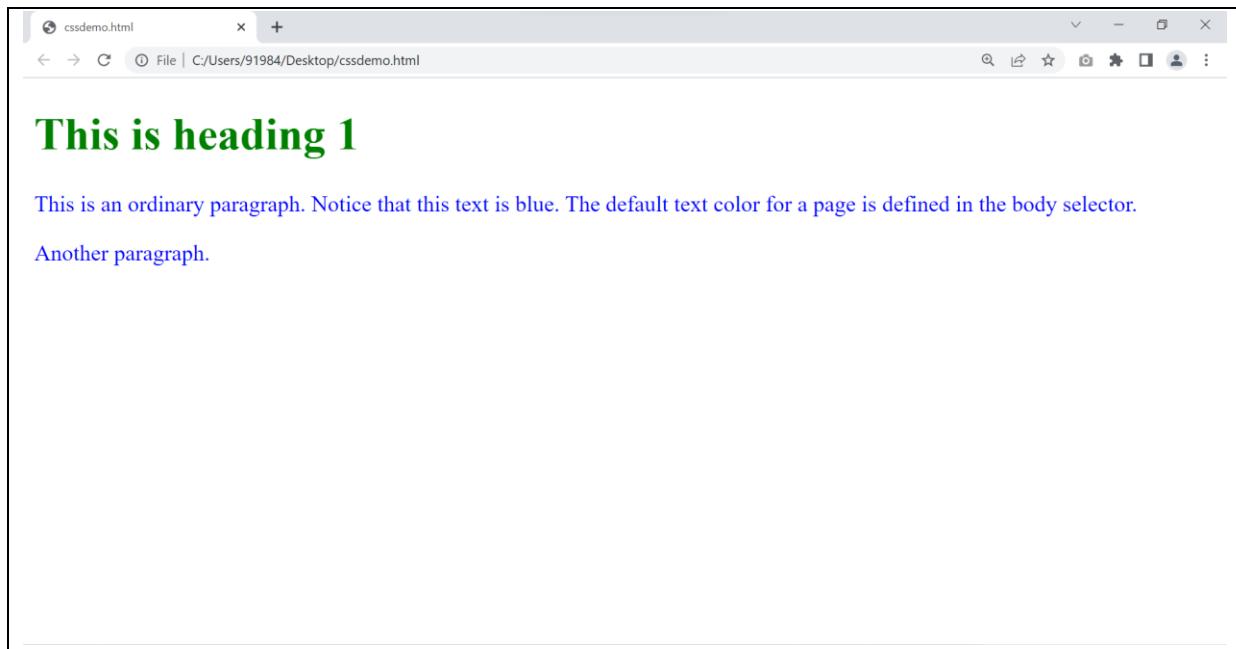
Example:

```
<!DOCTYPE html>

<html>

<head>
<style>
body {
    color: blue;
}
h1 {
    color: green;
}
</style>
</head>
<body>
<h1>This is heading 1</h1>
<p>This is an ordinary paragraph. Notice that this text is blue.
The default text color for a page is defined in the body selector.</p>
<p>Another paragraph. </p>
</body>
</html>
```

Output:



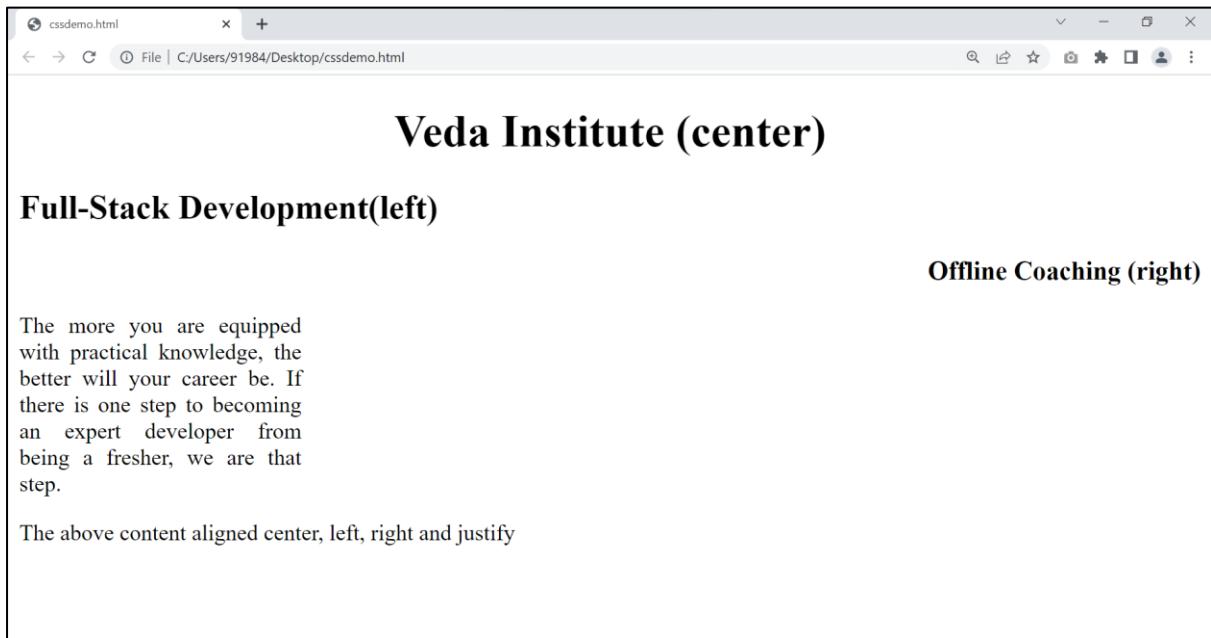
9.2 Text Alignment:

The text-align property is used to set the horizontal alignment of a text. A text can be left or right aligned, centered, or justified.

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
h1 {
text-align: center;
}
h2 {
text-align: left;
}
h3 {
text-align: right;
}
div{width:200px;text-align:justify;}
</style>
</head>
<body>
<h1>Veda Institute (center)</h1>
<h2>Full-Stack Development(left)</h2>
<h3>Offline Coaching (right)</h3>
<div>The more you are equipped with practical knowledge, the better will your
career be. If there is one step to becoming an expert developer from being a
fresher, we are that step. </div>
<p>The above content aligned center, left, right and justify</p>
</body>
</html>
```

Output:



9.3 Text Decoration:

The text decoration have following properties:

- text-decoration-line
- text-decoration-color
- text-decoration-style
- text-decoration-thickness
- text-decoration

text-decoration-line: The text-decoration-line property is used to add a decoration line to text. It has property values like **overline**, **line-through** and **underline**.

text-decoration-color: The text-decoration-color property is used to set the color of the decoration line. The property value represents with color name(red) and hexa code(#ffffff)

text-decoration-style: The text-decoration-style property is used to set the style of the decoration line. Style property values are solid, dotted, dashed, etc....

text-decoration-thickness: The text-decoration-thickness property is used to set the thickness of the decoration line. The text-decoration-thickness property values are auto, measured with **px**, **%** like 5px and 25%.

Note: For **<a>** tag, by default text decoration is underline. If you don't want underline for that text we use **text-decoration: none** property

Example:

```
<!DOCTYPE html>

<html>

<head>

<style>

h1{ text-decoration-line: underline; }

h2{text-decoration-line: line-through; text-decoration-color: red; }

h3{text-decoration-line: line-through; text-decoration-style: dashed; }

h4{text-decoration-line: underline; text-decoration-thickness: 5px; }

.note{

    text-decoration-line: underline;

    text-decoration-color: blue;

    text-decoration-style: solid;

    text-decoration-thickness: 5px;

}

.underline{text-decoration: none; }

</style>

</head>

<body>

<h1>Heading 1</h1>

<h2>Heading 2</h2>

<h3>Heading 3</h3>

<h4>Heading 4</h4>

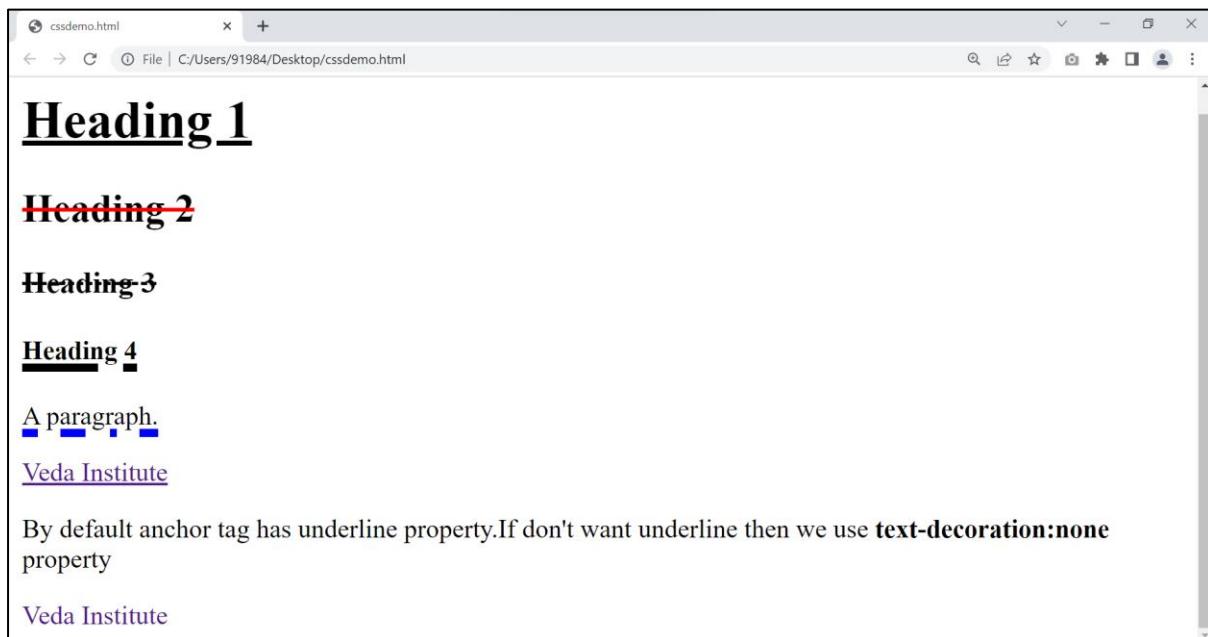
<p class="note">A paragraph. </p>

<a href="https://vedainstitute.in/">Veda Institute</a>

<p>By default anchor tag has underline property. If don't want underline then we use
<b>text-decoration: none</b> property</p>

<a href="https://vedainstitute.in/" class="underline">Veda Institute</a>

</body>
```

Output:

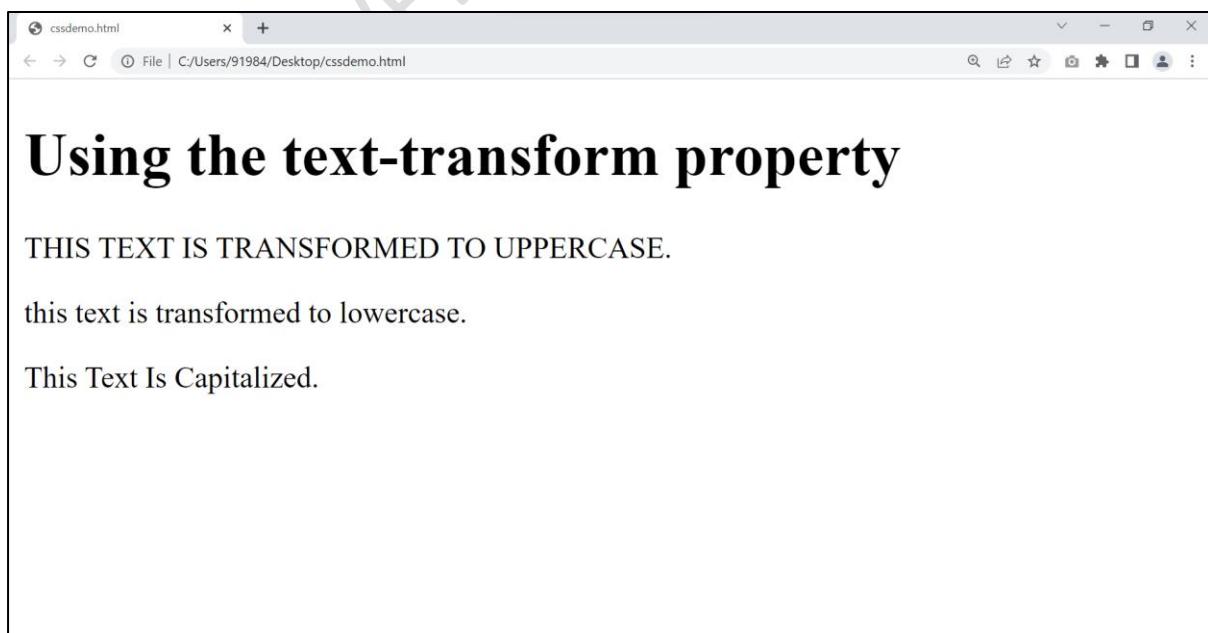
9.4 Text Transformation:

The **text-transform** property is used to specify uppercase and lowercase letters in a text. It has properties like uppercase, lowercase and capitalize.

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
p.uppercase {
text-transform: uppercase;
}
p.lowercase {
text-transform: lowercase;
}
p.capitalize {
text-transform: capitalize;
}</style>
</head>
<body>
<h1>Using the text-transform property</h1>
<p class="uppercase">This text is transformed to uppercase.</p>
<p class="lowercase">This text is transformed to lowercase.</p>
<p class="capitalize">This text is capitalized.</p>
</body>
</html>
```

Output:

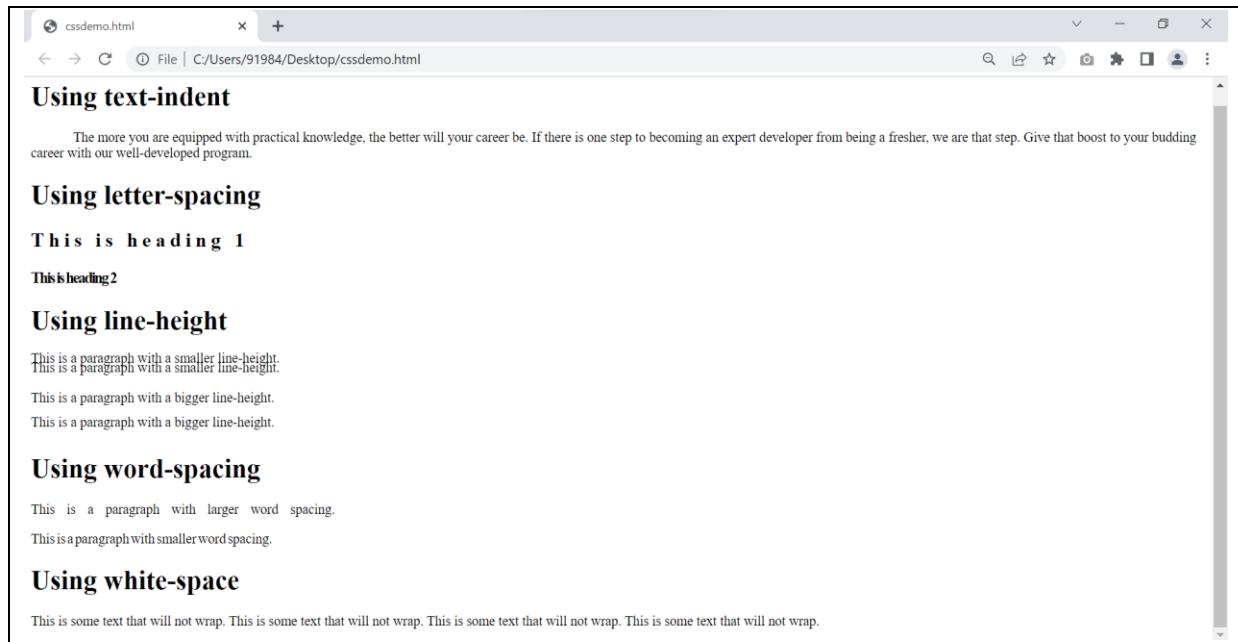


9.5 Text-spacing: In text-spacing we have following properties. They are

Property	Description
letter-spacing	Specifies the space between characters in a text
line-height	Specifies the line height
text-indent	Specifies the indentation of the first line in a text-block
white-space	Specifies how to handle white-space inside an element
word-spacing	Specifies the space between words in a text

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
p.indent {text-indent: 50px;}
h2 {letter-spacing: 5px;}
h3 {letter-spacing: -2px;}
p.small {line-height: 0.7;}
p.big {line-height: 1.8;}
p.one {word-spacing: 10px;}
p.two {word-spacing: -2px;}
p.white-space {white-space: nowrap;}
</style>
</head>
<body>
<h1>Using text-indent</h1>
<p class="indent">The more you are equipped with practical knowledge, the better will your career be. If there is one step to becoming an expert developer from being a fresher, we are that step. Give that boost to your budding career with our well-developed program. </p>
<h1>Using letter-spacing</h1>
<h2>This is heading 1</h2>
<h3>This is heading 2</h3>
<h1>Using line-height</h1>
<p class="small">This is a paragraph with a smaller line-height.<br>
This is a paragraph with a smaller line-height.<br></p>
<p class="big">
This is a paragraph with a bigger line-height.<br>
This is a paragraph with a bigger line-height.<br>
</p>
<h1>Using word-spacing</h1>
<p class="one">This is a paragraph with larger word spacing.</p>
<p class="two">This is a paragraph with smaller word spacing.</p>
<h1>Using white-space</h1>
<p class="white-space">
This is some text that will not wrap.
This is some text that will not wrap.
</p>
</body>
</html>
```



The more you are equipped with practical knowledge, the better will your career be. If there is one step to becoming an expert developer from being a fresher, we are that step. Give that boost to your budding career with our well-developed program.

Using text-indent

This is a paragraph with a smaller line-height.
This is a paragraph with a smaller line-height.

Using letter-spacing

This is heading 1
This is heading 2

Using line-height

This is a paragraph with a bigger line-height.
This is a paragraph with a bigger line-height.

Using word-spacing

This is a paragraph with larger word spacing.
This is a paragraph with smaller word spacing.

Using white-space

This is some text that will not wrap. This is some text that will not wrap. This is some text that will not wrap. This is some text that will not wrap.

10. Lists:

The CSS list properties allow you to:

- Set different list item markers for ordered lists
- Set different list item markers for unordered lists
- Set an image as the list item marker
- Add background colors to lists and list items

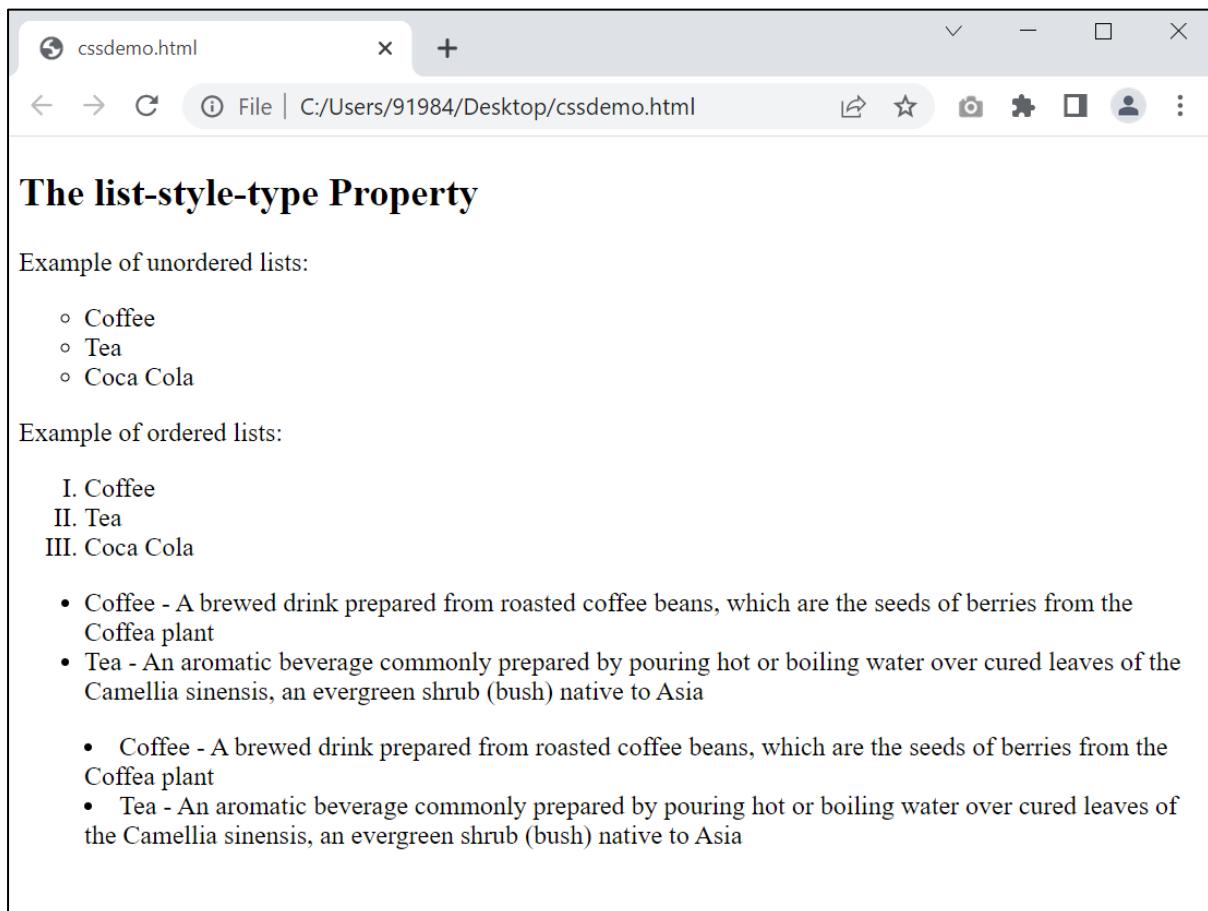
Note: For ul element the default list style is disc style. If we don't want list style then we use **list-style: none** property.

10.1 CSS List Properties: In lists we have following properties. They are

Property	Description
list-style	Sets all the properties for a list in one declaration
list-style-image	Specifies an image as the list-item marker
list-style-position	Specifies the position of the list-item markers (bullet points)
list-style-type	Specifies the type of list-item marker

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
ul.a {
list-style-type: circle;
}
ol.c {
list-style-type: upper-roman;
}
</style>
</head>
<body>
<h2>The list-style-type Property</h2>
<p>Example of unordered lists:</p>
<ul class="a">
<li>Coffee</li>
<li>Tea</li>
<li>Coca Cola</li>
</ul>
<p>Example of ordered lists:</p>
<ol class="c">
<li>Coffee</li>
<li>Tea</li>
<li>Coca Cola</li>
</ol>
<ul class="outside">
<li>Coffee - A brewed drink prepared from roasted coffee beans, which are the seeds of berries from the Coffea plant</li>
<li>Tea - An aromatic beverage commonly prepared by pouring hot or boiling water over cured leaves of the Camellia sinensis, an evergreen shrub (bush) native to Asia</li>
</ul>
<ul class="inside">
<li>Coffee - A brewed drink prepared from roasted coffee beans, which are the seeds of berries from the Coffea plant</li>
<li>Tea - An aromatic beverage commonly prepared by pouring hot or boiling water over cured leaves of the Camellia sinensis, an evergreen shrub (bush) native to Asia</li>
</ul>
</body>
</html>
```

Output:

The list-style-type Property

Example of unordered lists:

- Coffee
- Tea
- Coca Cola

Example of ordered lists:

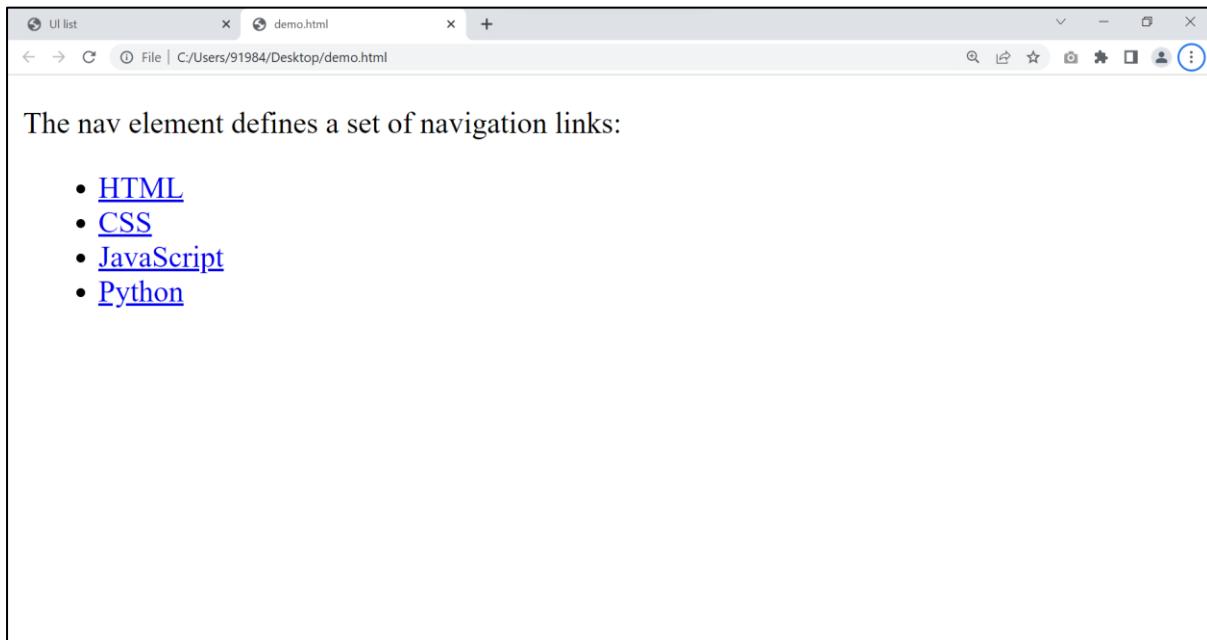
- I. Coffee
- II. Tea
- III. Coca Cola

- Coffee - A brewed drink prepared from roasted coffee beans, which are the seeds of berries from the Coffea plant
- Tea - An aromatic beverage commonly prepared by pouring hot or boiling water over cured leaves of the Camellia sinensis, an evergreen shrub (bush) native to Asia
 - Coffee - A brewed drink prepared from roasted coffee beans, which are the seeds of berries from the Coffea plant
 - Tea - An aromatic beverage commonly prepared by pouring hot or boiling water over cured leaves of the Camellia sinensis, an evergreen shrub (bush) native to Asia

Note: For ul element the default list style is disc style. If we don't want list style then we use **list-style: none** property.

Without applying **list-style: none** property the output is

Output:



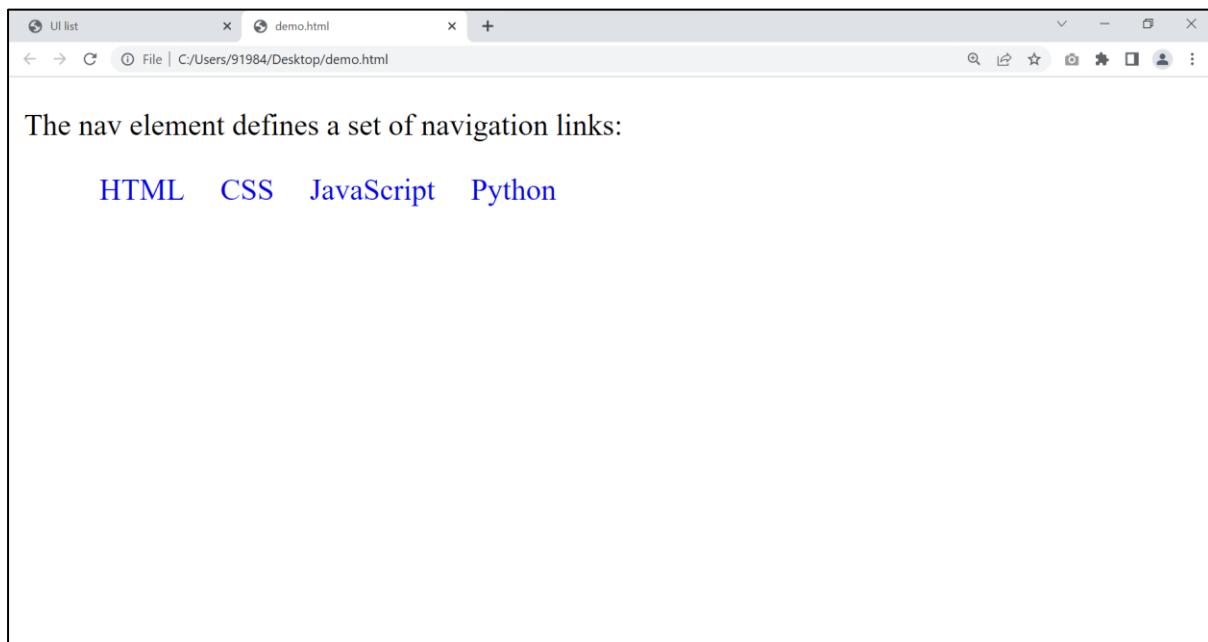
We use `` tag for list items in header. We use **list-style: none** property for menu links.

Example:

```
<!DOCTYPE html>

<head>
<style>
    ul {list-style: none;}
    ul li {display: inline-block;}
    ul li a {text-decoration: none; margin-right:15px;}
</style>
</head>
<html>
<body>
<p>The nav element defines a set of navigation links:</p>
<nav>
<ul>
<li><a href="/html/">HTML</a></li>
<li><a href="/css/">CSS</a></li>
<li><a href="/js/">JavaScript</a></li>
<li><a href="/python/">Python</a></li>
</nav>
</body>
</html>
```

Output:



11.Tables:

The look of an HTML table can be greatly improved with CSS

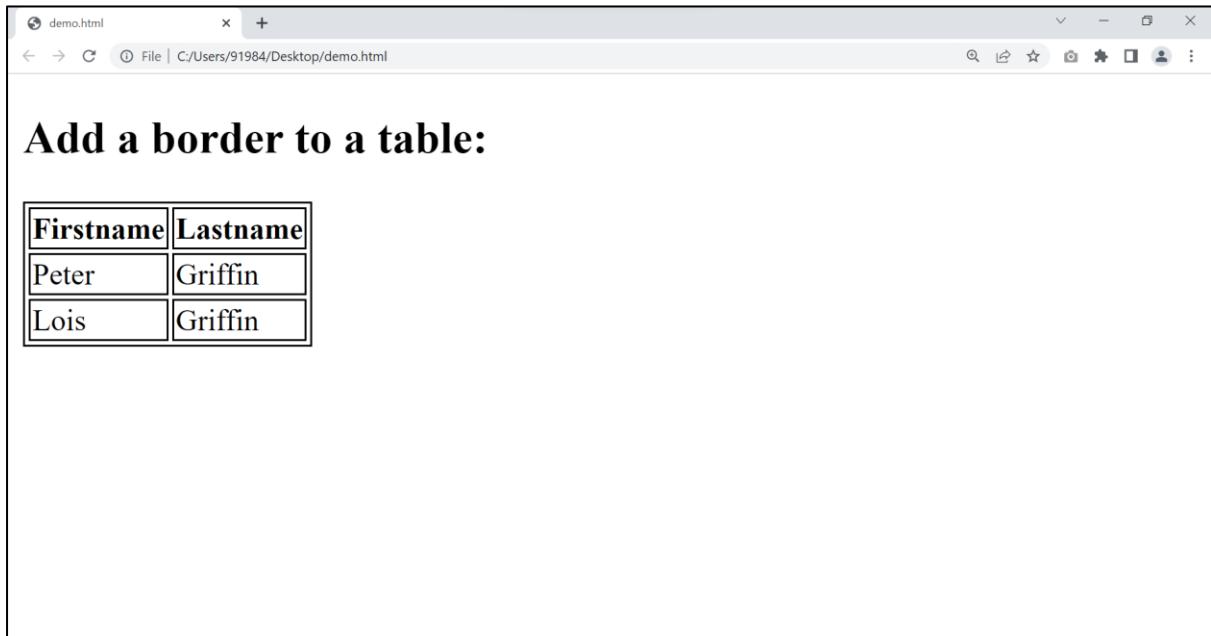
11.1 Borders:

To specify table borders in CSS, use the **border** property. The example below specifies a solid border for `<table>`, `<th>`, and `<td>` elements:

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
table, th, td {
border: 1px solid;
}
</style>
</head>
<body>
<h2>Add a border to a table:</h2>
<table>
<tr>
<th>Firstname</th>
<th>Lastname</th>
</tr>
<tr>
<td>Peter</td>
<td>Griffin</td>
</tr>
<tr>
<td>Lois</td>
<td>Griffin</td>
</tr>
</table>
</body>
</html>
```

Output:



A screenshot of a web browser window titled "demo.html". The address bar shows "File | C:/Users/91984/Desktop/demo.html". The main content area displays the text "Add a border to a table:" followed by a table with two rows and two columns. The table has a border and contains the following data:

Firstname	Lastname
Peter	Griffin
Lois	Griffin

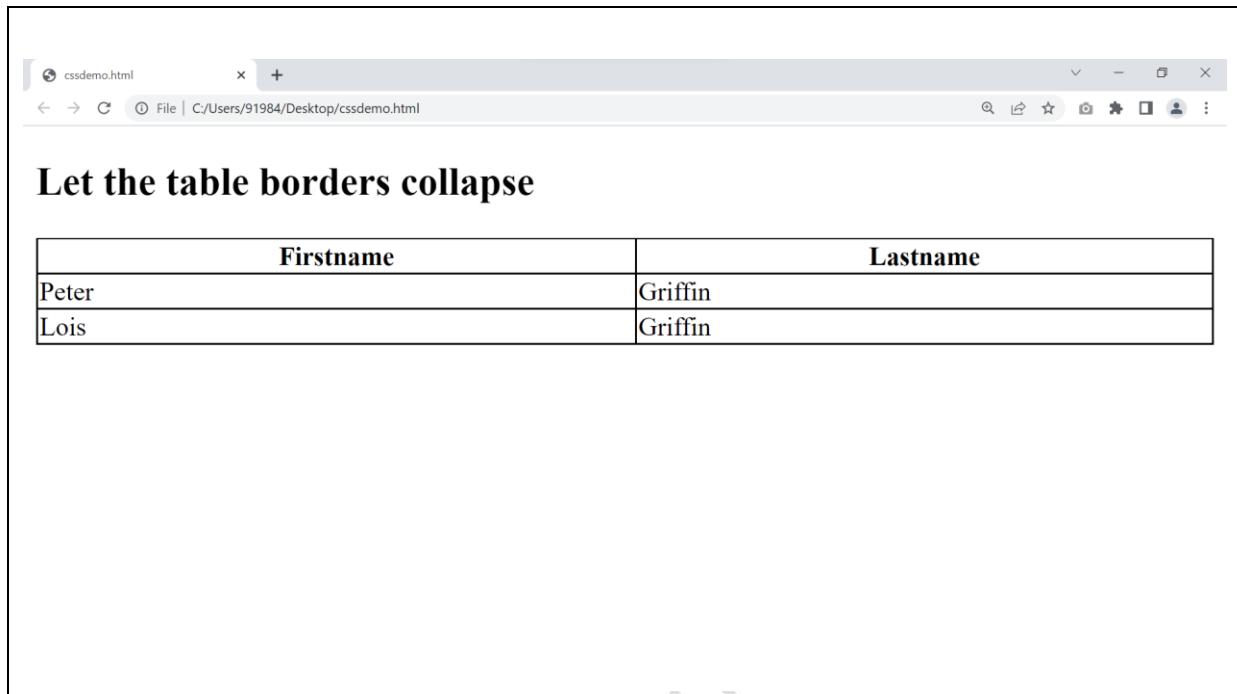
11.2 Collapse Table Borders:

The border-collapse property sets whether the table borders should be collapsed into a single border:

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
table, td, th {
border: 1px solid;
}
table {
width: 100%;
border-collapse: collapse;
}
</style>
</head>
<body>
<h2>Let the table borders collapse</h2>
<table>
<tr>
<th>Firstname</th>
<th>Lastname</th>
</tr>
<tr>
<td>Peter</td>
<td>Griffin</td>
</tr>
<tr>
<td>Lois</td>
<td>Griffin</td>
</tr>
</table>
```

Output:



11.3 Table Size:

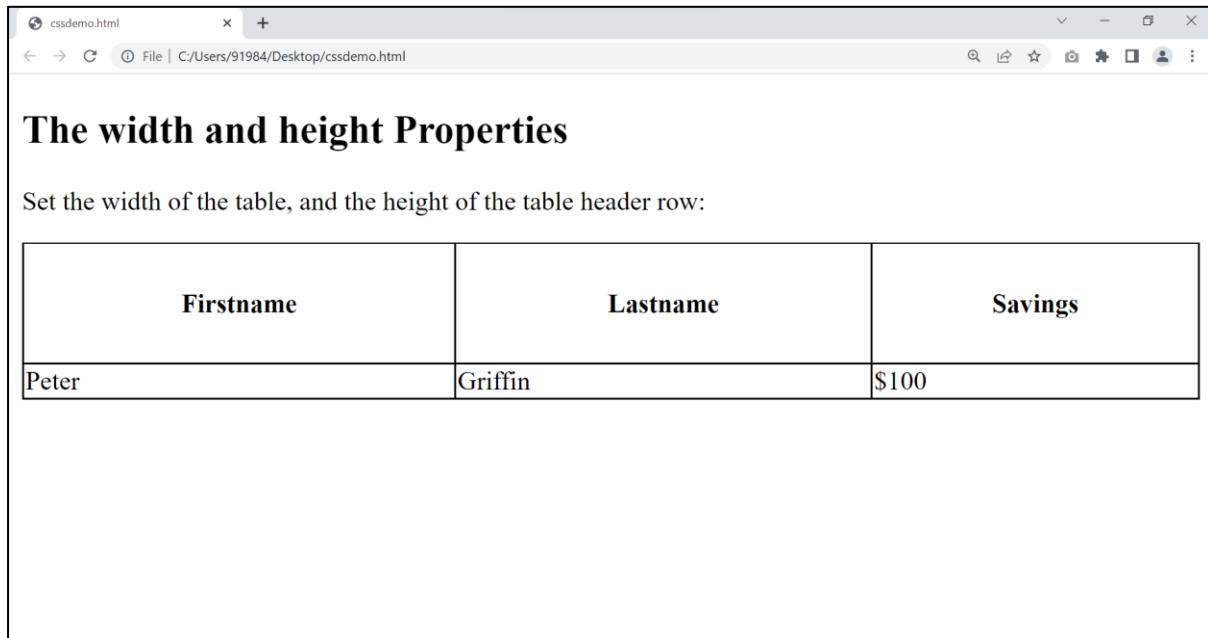
Table Width and Height: The width and height of a table are defined by the width and height properties.

The example below sets the width of the table to 100%, and the height of the <th> elements to 70px:

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
table, td, th {
border: 1px solid black;
}table {
border-collapse: collapse;
width: 100%;
}
th {
height: 70px;
}
</style>
</head>
<body>
<h2>The width and height Properties</h2>
<p>Set the width of the table, and the height of the table header row:</p>
<table>
<tr>
<th>Firstname</th>
<th>Lastname</th>
<th>Savings</th>
</tr>
<tr>
<td>Peter</td>
<td>Griffin</td>
<td>$100</td>
</tr>
</table>
</body>
<html>
```

Output:



Firstname	Lastname	Savings
Peter	Griffin	\$100

11.4 Table Alignment:

11.4.1 Horizontal Alignment:

The **text-align** property sets the horizontal alignment (like left, right, or center) of the content in **<th>** or **<td>**.

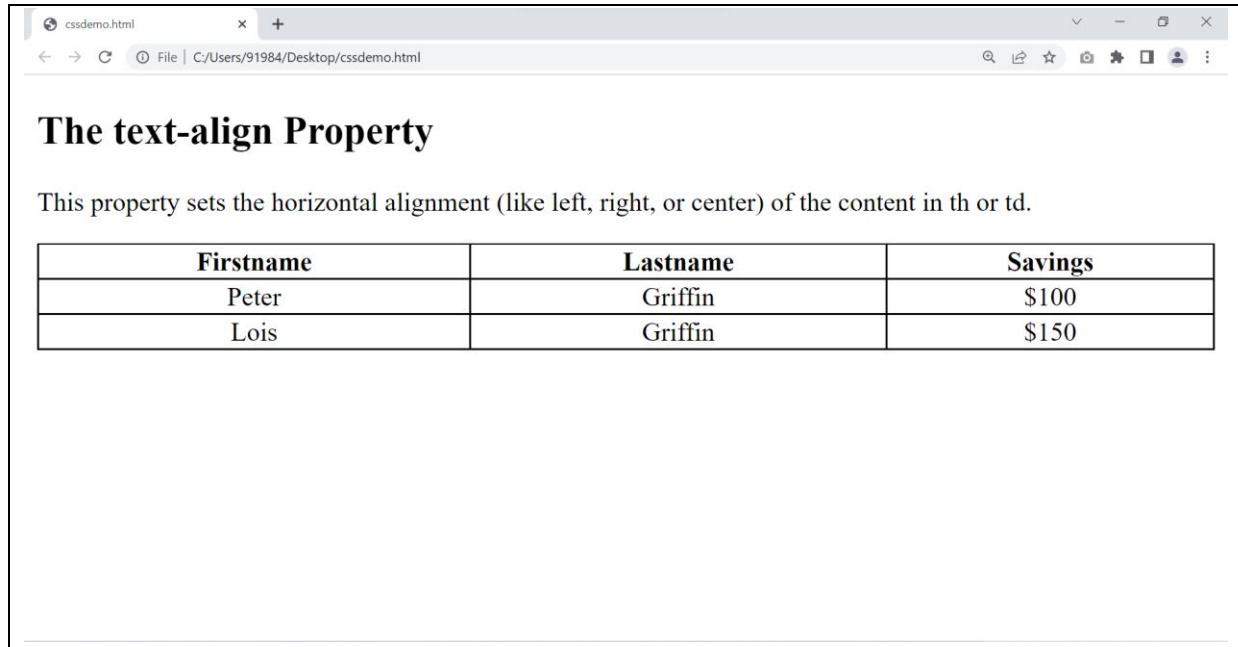
By default, the content of **<th>** elements are center-aligned and the content of **<td>** elements are left-aligned.

To center-align the content of **<td>** elements as well, use **text-align: center**

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
table, td, th {
    border: 1px solid black;
}
table {
    border-collapse: collapse;
    width: 100%;
}
td {
    text-align: center;
}
</style>
</head>
<body>
<h2>The text-align Property</h2>
<p>This property sets the horizontal alignment (like left, right, or center) of the content in th or td.</p>
<table>
<tr>
<th>Firstname</th>
<th>Lastname</th>
<th>Savings</th>
</tr>
<tr>
<td>Peter</td>
<td>Griffin</td>
<td>$100</td>
</tr>
<tr>
<td>Lois</td>
<td>Griffin</td>
<td>$150</td>
</tr>
</table>
</body>
</html>
```

Output:



The screenshot shows a Microsoft Edge browser window with the title "cssdemo.html". The address bar shows "File | C:/Users/91984/Desktop/cssdemo.html". The main content area displays the heading "The text-align Property" and a table with three columns: "Firstname", "Lastname", and "Savings". The table has three rows. The first row contains the column headers. The second row contains the values "Peter" and "Griffin" in the first two columns, and "\$100" in the third. The third row contains the values "Lois" and "Griffin" in the first two columns, and "\$150" in the third.

Firstname	Lastname	Savings
Peter	Griffin	\$100
Lois	Griffin	\$150

11.5 Vertical Alignment

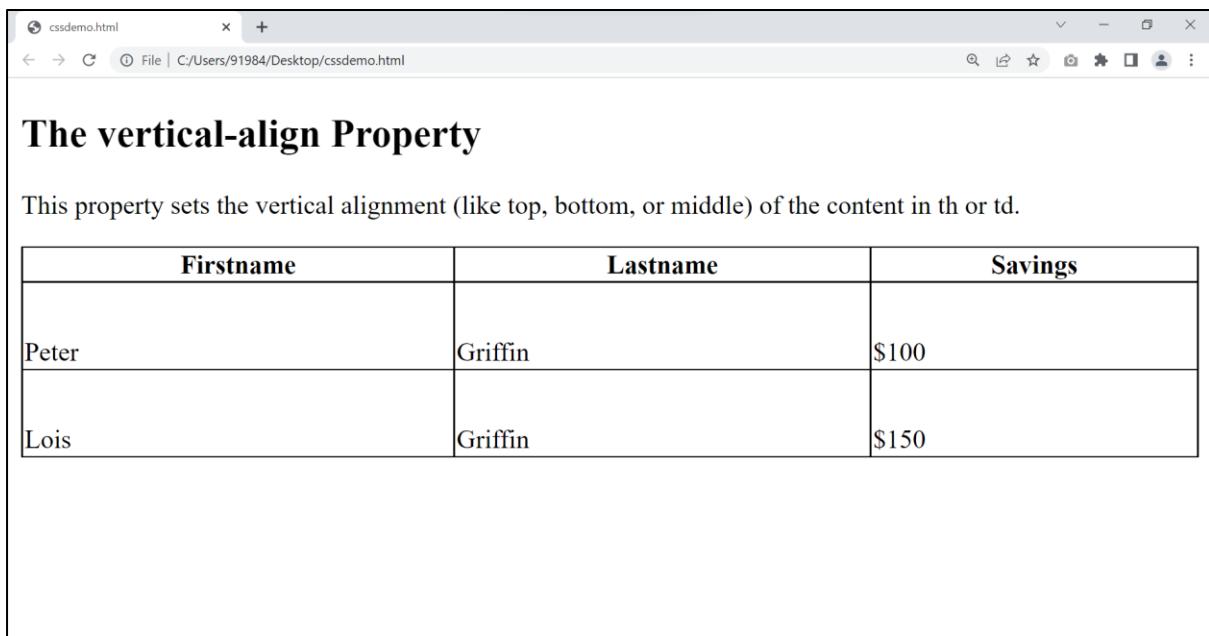
The vertical-align property sets the vertical alignment (like top, bottom, or middle) of the content in <th> or <td>.

By default, the vertical alignment of the content in a table is middle (for both <th> and <td> elements).

The following example sets the vertical text alignment to bottom for <td> elements:

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
table, td, th {
    border: 1px solid black;
}
table {
    border-collapse: collapse;
    width: 100%;
}
td {
    height: 50px;
    vertical-align: bottom;
}
</style>
</head>
<body>
<h2>The vertical-align Property</h2>
<p>This property sets the vertical alignment (like top, bottom, or middle) of the content in th or td.</p>
<table>
<tr>
<th>Firstname</th>
<th>Lastname</th>
<th>Savings</th>
</tr>
<tr>
<td>Peter</td>
<td>Griffin</td>
<td>$100</td>
</tr>
<tr>
<td>Lois</td>
<td>Griffin</td>
<td>$150</td>
</tr>
</table>
</body>
</html>
```

Output:

The screenshot shows a browser window with the title "cssdemo.html". The page content is titled "The vertical-align Property". A text block explains that this property sets the vertical alignment (like top, bottom, or middle) of the content in th or td. Below the text is a table with three columns: "Firstname", "Lastname", and "Savings". The table has two rows. The first row contains "Peter" in the Firstname column and "Griffin" in the Lastname column, with a value of "\$100" in the Savings column. The second row contains "Lois" in the Firstname column and "Griffin" in the Lastname column, with a value of "\$150" in the Savings column.

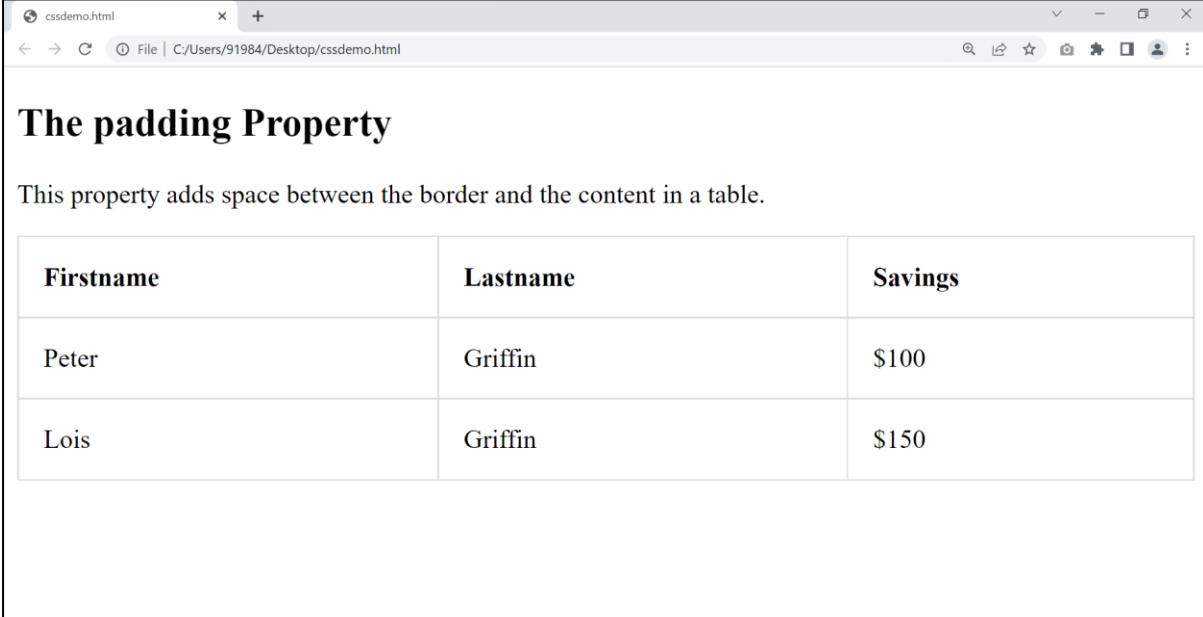
Firstname	Lastname	Savings
Peter	Griffin	\$100
Lois	Griffin	\$150

11.6 Table Style:**11.6.1 Table Padding:**

To control the space between the border and the content in a table, use the padding property on <td> and <th> elements

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
table, td, th {
border: 1px solid #ddd;
text-align: left;
}
table {
border-collapse: collapse;
width: 100%;
}
th, td {
padding: 15px;
}
</style>
</head>
<body>
<h2>The padding Property</h2>
<p>This property adds space between the border and the content in a table.</p>
<table>
<tr>
<th>Firstname</th>
<th>Lastname</th>
<th>Savings</th>
</tr>
<tr>
<td>Peter</td>
<td>Griffin</td>
<td>$100</td>
</tr>
<tr>
<td>Lois</td>
<td>Griffin</td>
<td>$150</td>
</tr>
</table>
```

Output:

The padding Property

This property adds space between the border and the content in a table.

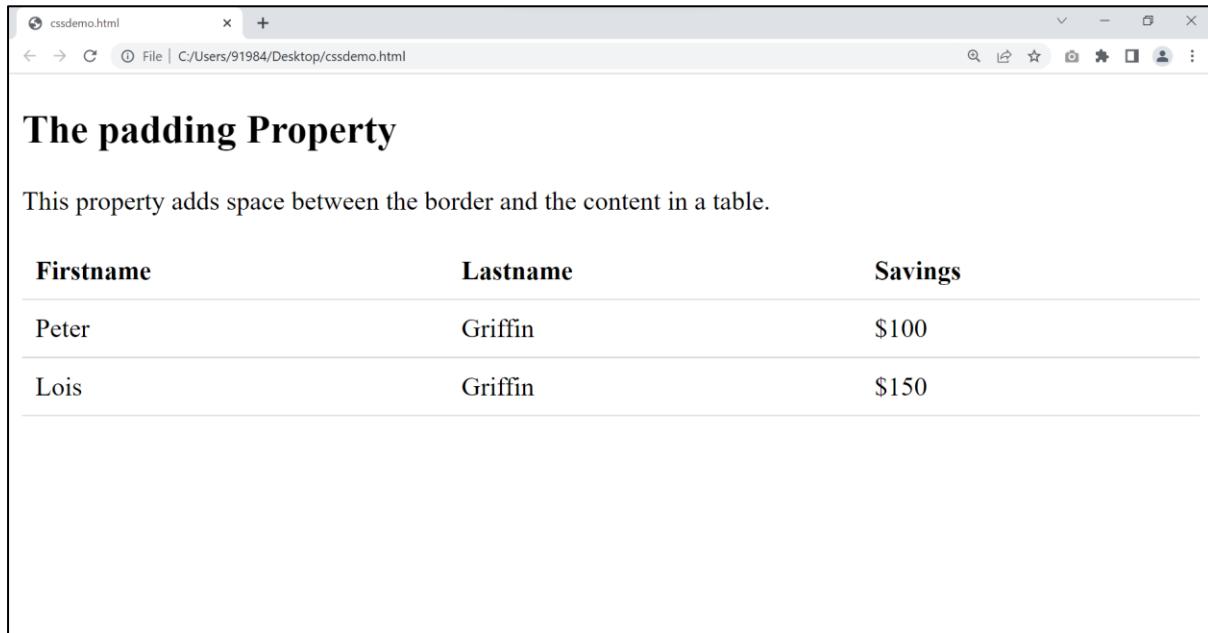
Firstname	Lastname	Savings
Peter	Griffin	\$100
Lois	Griffin	\$150

11.7 Horizontal Dividers:

Add the border-bottom property to <th> and <td> for horizontal dividers:

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
table {
border-collapse: collapse;
width: 100%;
}
th, td {
padding: 8px;
text-align: left;
border-bottom: 1px solid #ddd;
}
</style>
</head>
<body>
<h2>Bordered Table Dividers</h2>
<p>Add the border-bottom property to th and td for horizontal dividers:</p>
<table>
<tr>
<th>Firstname</th>
<th>Lastname</th>
<th>Savings</th>
</tr>
<tr>
<td>Peter</td>
<td>Griffin</td>
<td>$100</td>
</tr>
<tr>
<td>Lois</td>
<td>Griffin</td>
<td>$150</td>
</tr>
</table>
```

Output:

Firstname	Lastname	Savings
Peter	Griffin	\$100
Lois	Griffin	\$150

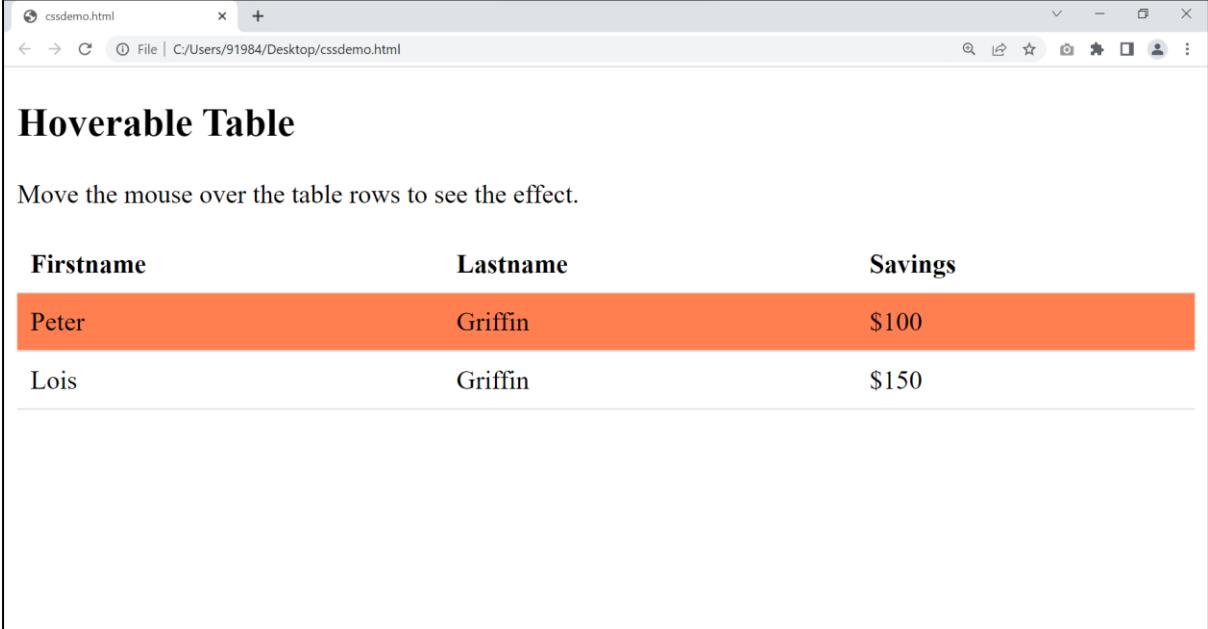
11.8 Hoverable Table:

Use the: **hover** selector on <tr> to highlight table rows on mouse over:

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
table {
border-collapse: collapse;
width: 100%;
}
th, td {
padding: 8px;
text-align: left;
border-bottom: 1px solid #ddd;
}
tr:hover {background-color: coral;}
</style>
</head>
<body>
<h2>Hoverable Table</h2>
<p>Move the mouse over the table rows to see the effect.</p><tr>
<th>Firstname</th>
<th>Lastname</th>
<th>Savings</th>
</tr>
<tr>
<td>Peter</td>
<td>Griffin</td>
<td>$100</td>
</tr>
<tr>
<td>Lois</td>
<td>Griffin</td>
<td>$150</td>
</tr>
<tr>
```

Output:



The screenshot shows a web browser window with the title "cssdemo.html". The page content is titled "Hoverable Table" and includes a sub-instruction: "Move the mouse over the table rows to see the effect.". Below this is a table with three columns: "Firstname", "Lastname", and "Savings". There are two rows of data: one for "Peter Griffin" with \$100 savings, and another for "Lois Griffin" with \$150 savings. The second row is highlighted with a red background, demonstrating the CSS hover effect.

Firstname	Lastname	Savings
Peter	Griffin	\$100
Lois	Griffin	\$150

12. Dimensions:

The dimensions in css are height, width, min-width , max-width, min-height and max-height.

12.1 Height, Width and Max-width:

The CSS height and width properties are used to set the height and width of an element. The CSS max-width property is used to set the maximum width of an element.

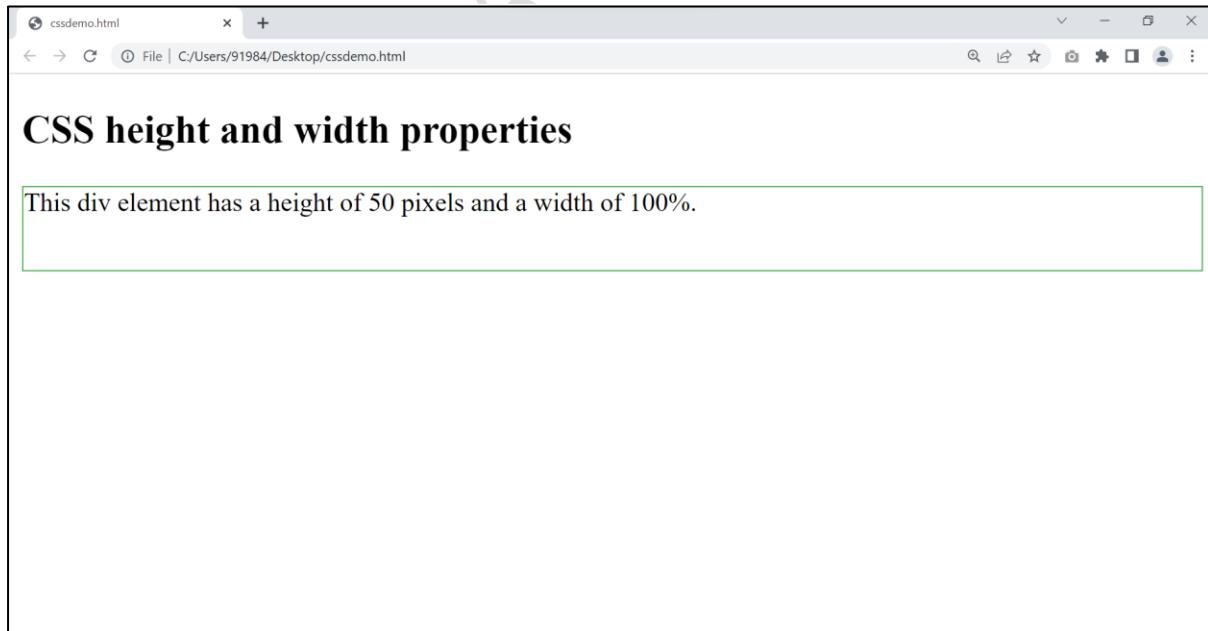
The height and width properties may have the following values:

- **auto** - This is default. The browser calculates the height and width
- **length** - Defines the height/width in px, cm etc.
- **%** - Defines the height/width in percent of the containing block
- **initial** - Sets the height/width to its default value
- **inherit** - The height/width will be inherited from its parent value

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
div {
height: 50px;
width: 100%;
border: 1px solid #4CAF50;
}
</style>
</head>
<body>
<h2>CSS height and width properties</h2>
<div>This div element has a height of 50 pixels and a width of 100%.</div>
</body>
</html>
```

Output:



12.1.1 max-width:

The **max-width** property is used to set the maximum width of an element. The **max-width** can be specified in *length values*, like px, cm, etc., or in percent (%) of the containing block, or set to none (this is default. Means that there is no maximum width).

The problem with the `<div>` above occurs when the browser window is smaller than the width of the element (500px). The browser then adds a horizontal scrollbar to the page.

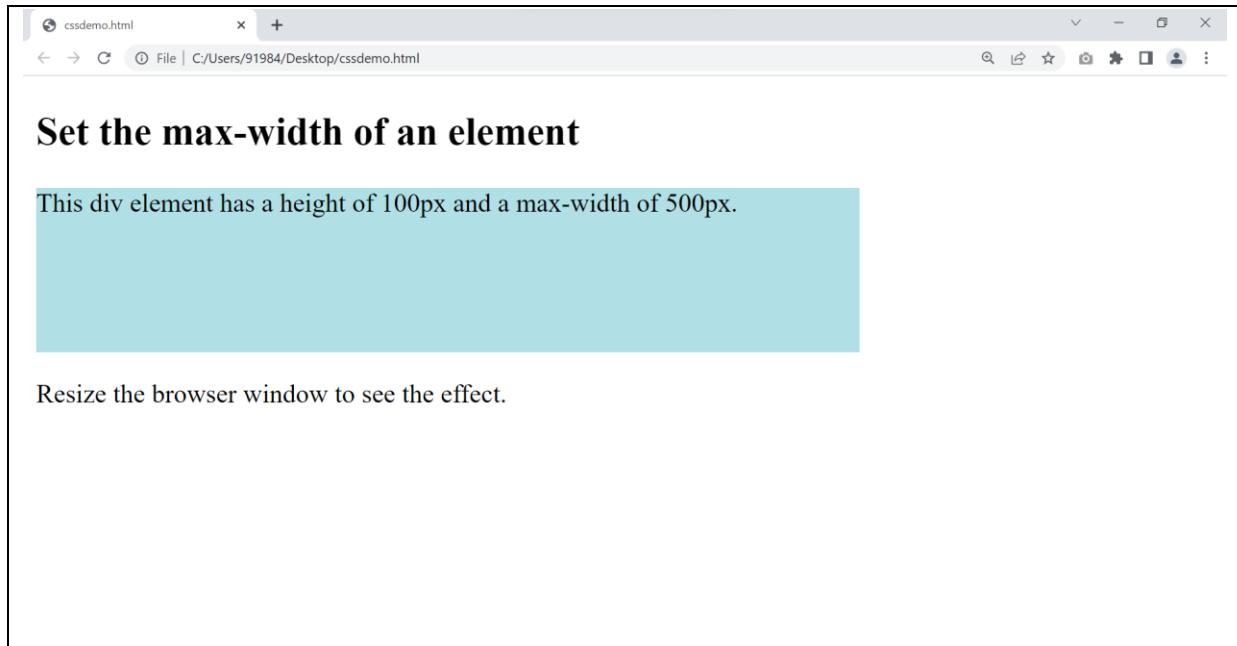
Using **max-width** instead, in this situation, will improve the browser's handling of small windows.

Note: If you for some reason use both the **width** property and the **max-width** property on the same element, and the value of the **width** property is larger than the **max-width** property; the **max-width** property will be used (and the **width** property will be ignored).

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
div {
  max-width: 500px;
  height: 100px;
  background-color: powderblue;
}
</style>
</head>
<body>
<h2>Set the max-width of an element</h2>
<div>This div element has a height of 100px and a max-width of 500px.</div>
<p>Resize the browser window to see the effect.</p>
</body>
</html>
```

Output:



12.2 Max-height and Min-height:

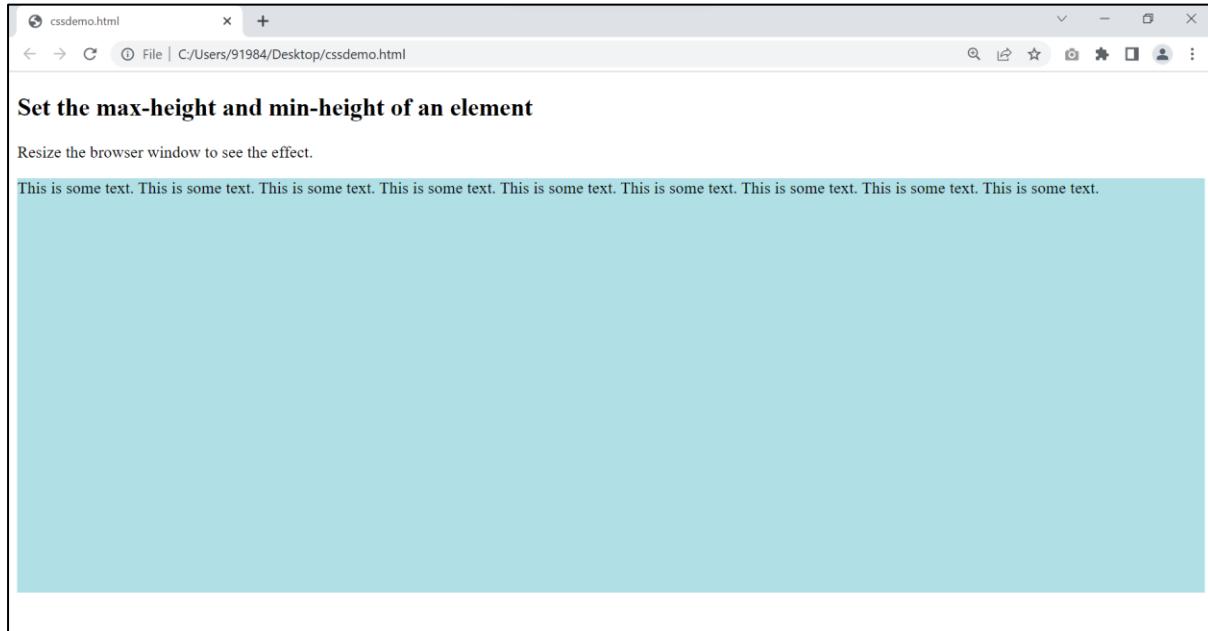
12.2.1 Max-height: The **max-height** CSS property sets the maximum height of an element. It prevents the used value of the height property from becoming larger than the value specified for **max-height**.

12.2.2 Min-height: The **min-height** CSS property sets the minimum height of an element. It prevents the used value of the height property from becoming smaller than the value specified for **min-height**.

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
div {
  max-height: 600px;
  min-height: 400px;
  background-color: powderblue;
}
</style>
</head>
<body>
<h2>Set the max-height and min-height of an element</h2>
<p>Resize the browser window to see the effect.</p>
<div>This is some text. This is some text. This is some text.
  This is some text. This is some text. This is some text.
  This is some text. This is some text. This is some text.</div>
</body>
</html>
```

Output:



13. Display

The **display** property is the most important CSS property for controlling layout. The display property specifies if/how an element is displayed. Every HTML element has a default display value depending on what type of element it is. The default display value for most elements is **block** or **inline**.

- Display-none
- Display-inline
- Display-block
- Display-inlineblock
- Display-flex

13.1 Display-none:

display: none; is commonly used with JavaScript to hide and show elements without deleting and recreating them. Take a look at our example on this page if you want to know how this can be achieved.

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
h1.hidden {
    display: none;
}</style>
</head>
<body>

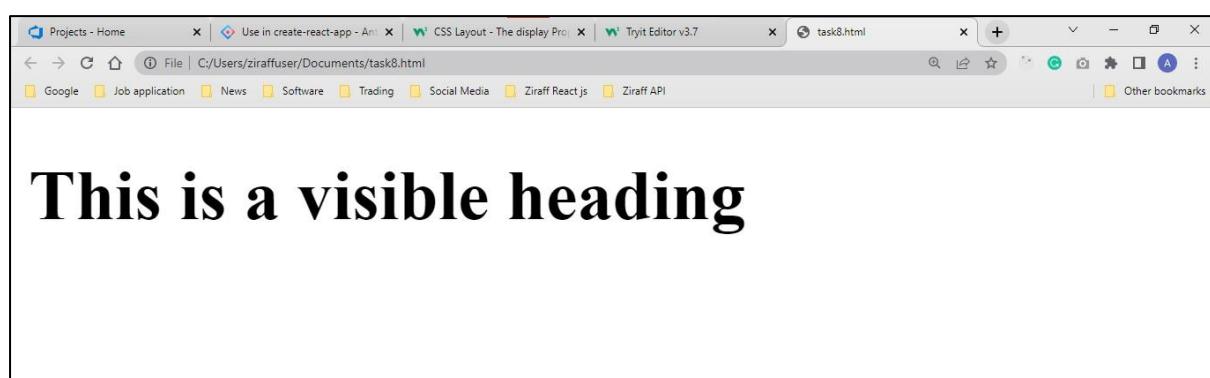
<h1>This is a visible heading</h1>

<h1 class="hidden">This is a hidden heading</h1>

<p>Notice that the h1 element with display: none; does not take up any space. </p>

</body>
</html>
```

Output:



13.2 Display-inline:

As mentioned, every element has a default display value. However, you can override this. Changing an inline element to a block element, or vice versa, can be useful for making the page look a specific way, and still follow the web standards.

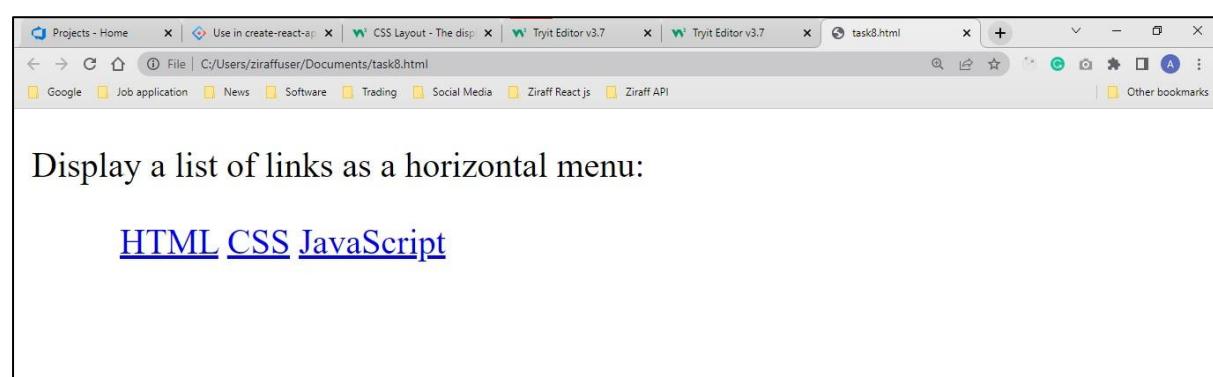
A common example is making inline elements for horizontal menus:

Example:

```
<!DOCTYPE html>

<html>
<head>
<style>
li {display: inline;}
</style>
</head>
<body>
<p>Display a list of links as a horizontal menu:</p>
<ul>
<li><a href="/html/default.asp" target="_blank">HTML</a></li>
<li><a href="/css/default.asp" target="_blank">CSS</a></li>
<li><a href="/js/default.asp" target="_blank">JavaScript</a></li>
</ul>
</body>
</html>
```

Output:



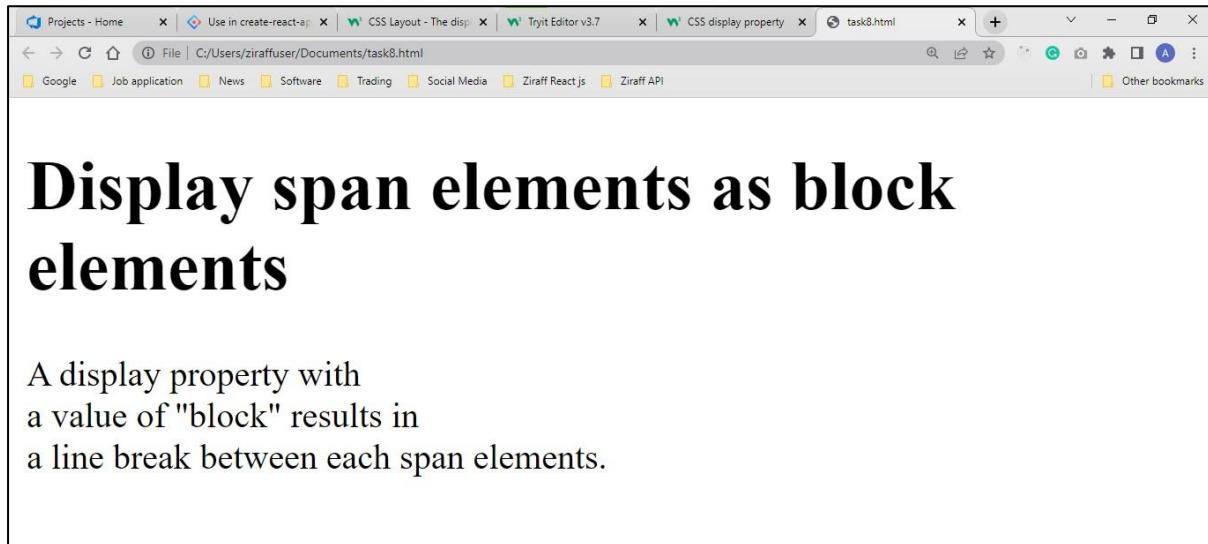
13.3 Display-Block:

Displays an element as a block element (like <p>). It starts on a new line, and takes up the whole width

Example:

```
<!DOCTYPE html>

<html>
<head>
<style>
span {display: block;}
</style>
</head>
<body>
<h1>Display span elements as block elements</h1>
<span>A display property with</span>
<span>a value of "block" results in</span> <span>a line break between each span elements. </span>
</body>
</html>
```

Output:**13.4 Display-inlineBlock:**

Displays an element as an inline-level block container. The element itself is formatted as an inline element, but you can apply height and width values.

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
span {display: block;}
</style>
</head>
<body>

<h2>display: inline-block</h2>
<div>Lorem ipsum dolor sit amet, consectetur adipescent elite.

Vestibulum consequent scalarise elit sit amet consequent. Aliquam erat volutpat.

<span class="b">Aliquam</span> <span class="b">venenatis</span>

gravida nisl sit amet facilisis. Nullam cursus fermentum velit sed laoreet. </div>

</body>

</html>
```

Output:



13.5 Display-Flex:

An area of a document laid out using flexbox is called a **flex container**. To create a flex container, we set the value of the area's container's **display** property to **flex**. As soon as we do this the direct children of that container become **flex items**. The usual properties of flex-container are:

- Justify-content
 - align-items

13.5.1 Justify-content:

The **justify-content** property is used to align the flex items. The values used for justify-content property are **space-between**, **space-around**, **space-evenly**, **center**, **flex-start**, **flex-end**.

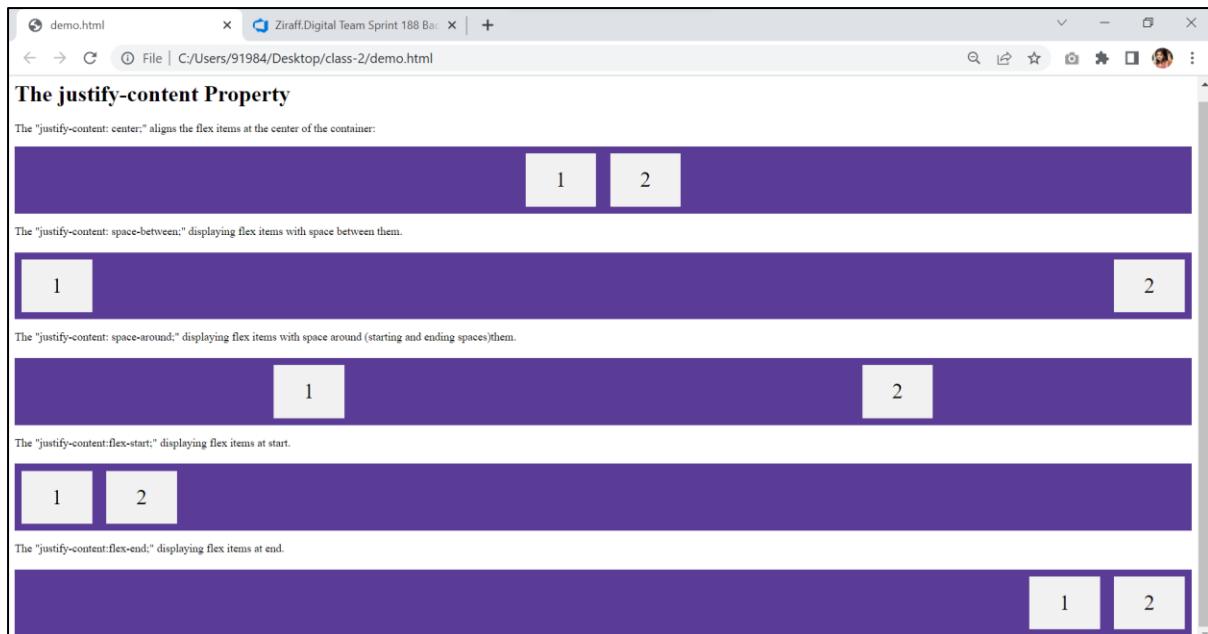
Example: html file

```
<!DOCTYPE html>
<html>
<head>
<link rel="Stylesheet" href="style.css">
</head>
<body>
<h1>The justify-content Property</h1>
<p>The "justify-content: center;" aligns the flex items at the center of the container:</p>
<div class="flex-container">
<div>1</div>
<div>2</div>
</div>
<p>The "justify-content: space-between;" displaying flex items with space between them.</p>
<div class="space-between">
<div>1</div>
<div>2</div>
</div>
<p>The "justify-content: space-around;" displaying flex items with space around (starting and ending spaces)them.</p>
<div class="space-around">
<div>1</div>
<div>2</div>
</div>
<p>The "justify-content: flex-start;" displaying flex items at start.</p>
<div class="flex-start">
<div>1</div>
<div>2</div>
</div>
<p>The "justify-content: flex-end;" displaying flex items at end.</p>
<div class="flex-end">
<div>1</div>
<div>2</div>
</div>
</body>
</html>
```

Example: Css file

```
.flex-container {  
    display: flex;  
    justify-content: center;  
    background-color: #5A3B96;  
}  
.space-between {  
    display: flex;  
    justify-content: space-between;  
    background-color: #5A3B96;  
    margin-top:20px;  
}  
.space-around {  
    display: flex;  
    justify-content: space-around;  
    background-color: #5A3B96;  
    margin-top:20px;  
}  
.flex-start {  
    display: flex;  
    justify-content: flex-start;  
    background-color: #5A3B96;  
    margin-top:20px;  
}  
.flex-end {display: flex;  
justify-content: flex-end;  
background-color: #5A3B96;  
margin-top:20px;  
}  
.flex-container > div {  
    background-color: #f1f1f1;  
    width: 100px;  
    margin: 10px;  
    text-align: center;  
    line-height: 75px;  
    font-size: 30px;  
}  
.space-around > div, .flex-start > div, .space-between > div, .flex-end > div {  
    background-color: #f1f1f1;  
    width: 100px;  
    margin: 10px;  
    text-align: center;  
    line-height: 75px;  
    font-size: 30px;
```

Output:



13.5.2 align-items: The **align-items** property is used to align the flex items. The values of align-items property are **center, flex-start, flex-end etc.**

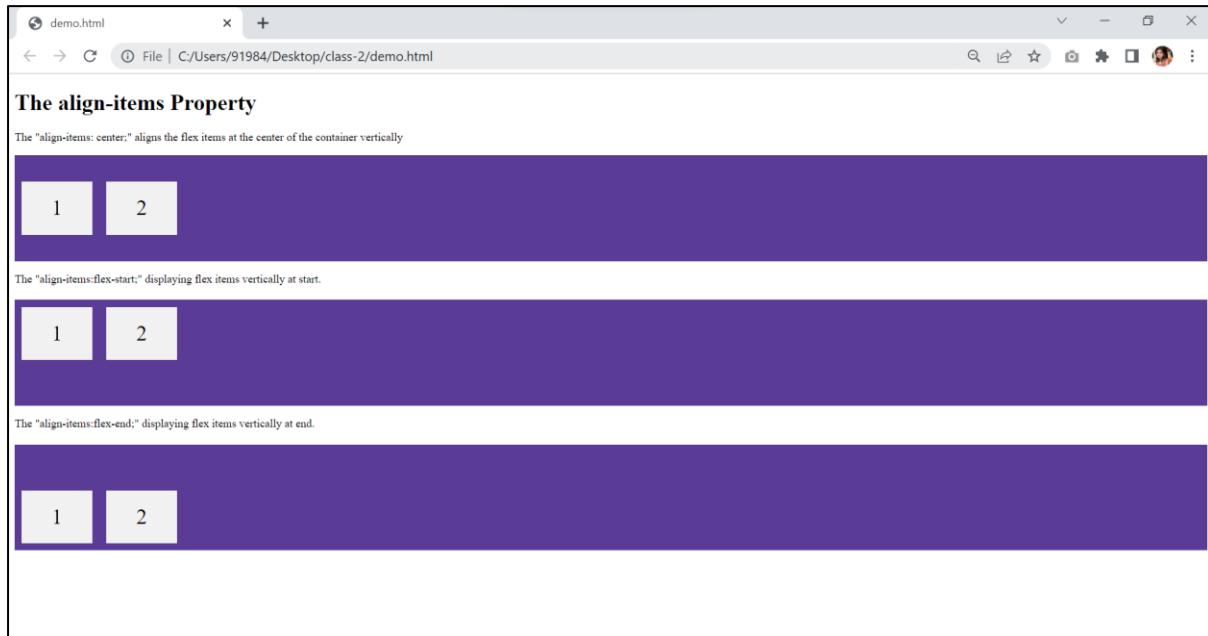
Example: html file

```
<!DOCTYPE html>
<html>
<head>
<link rel="stylesheet" href="style.css">
</head>
<body>
<h1>The align-items Property</h1>
<p>The "align-items: center;" aligns the flex items at the center of the container vertically</p>
<div class="flex-container">
<div>1</div>
<div>2</div>
</div>
<p>The "align-items:flex-start;" displaying flex items vertically at start.</p>
<div class="flex-start">
<div>1</div>
<div>2</div>
</div>
<p>The "align-items:flex-end;" displaying flex items vertically at end.</p>
<div class="flex-end">
<div>1</div>
<div>2</div>
</div>
</body>
</html>
```

Css file:

```
.flex-container {  
    display: flex;  
    height:150px;  
    align-items: center;  
    background-color: #5A3B96;  
}  
.flex-start {  
    display: flex;  
    height:150px;  
    align-items: flex-start;  
    background-color: #5A3B96;  
    margin-top:20px;  
}  
.flex-end {  
    display: flex;  
    height:150px;  
    align-items: flex-end;  
    background-color: #5A3B96;  
    margin-top:20px;  
}  
.flex-container > div {  
    background-color: #f1f1f1;  
    width: 100px;  
    margin: 10px;  
    text-align: center;  
    line-height: 75px;  
    font-size: 30px;  
}  
.flex-start > div, .baseline > div, .flex-end > div {  
    background-color: #f1f1f1;  
    width: 100px;  
    margin: 10px;  
    text-align: center;  
    line-height: 75px;  
    font-size: 30px;  
}
```

Output:



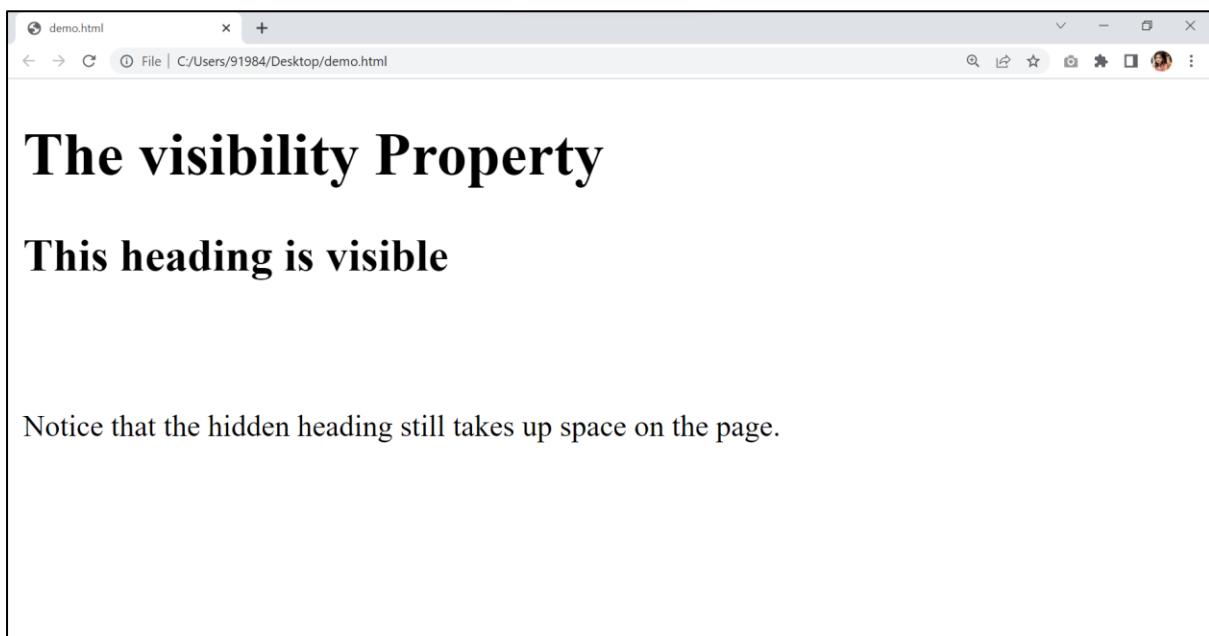
13.6.Visibility:

The **visibility** property specifies whether or not an element is visible.

Note: Hidden elements take up space on the page. Use the **display** property to both hide and remove an element from the document layout!

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
h2.a {
  visibility: visible
}
h2.b {
  visibility: hidden;
}
</style>
</head>
<body>
<h1>The visibility Property</h1>
<h2 class="a">This heading is visible</h2>
<h2 class="b">This heading is hidden</h2>
<p>Notice that the hidden heading still takes up space on the page.</p>
</body>
</html>
```

Output:

The visibility Property

This heading is visible

Notice that the hidden heading still takes up space on the page.

14. Positions

The **position** property specifies the type of positioning method used for an element (static, relative, fixed, absolute or sticky).

- **Position-Relative**
- **Position-absolute**
- **Position-static**
- **Position-fixed**
- **Position-sticky**

14.1 Position-Relative:

An element with **position: relative;** is positioned relative to its normal position. Setting the top, right, bottom, and left properties of a relatively-positioned element will cause it to be adjusted away from its normal position. Other content will not be adjusted to fit into any gap left by the element.

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
div.relative {
    position: relative;
    left: 30px;
    border: 3px solid #73AD21;
}
</style>
</head>
<body>

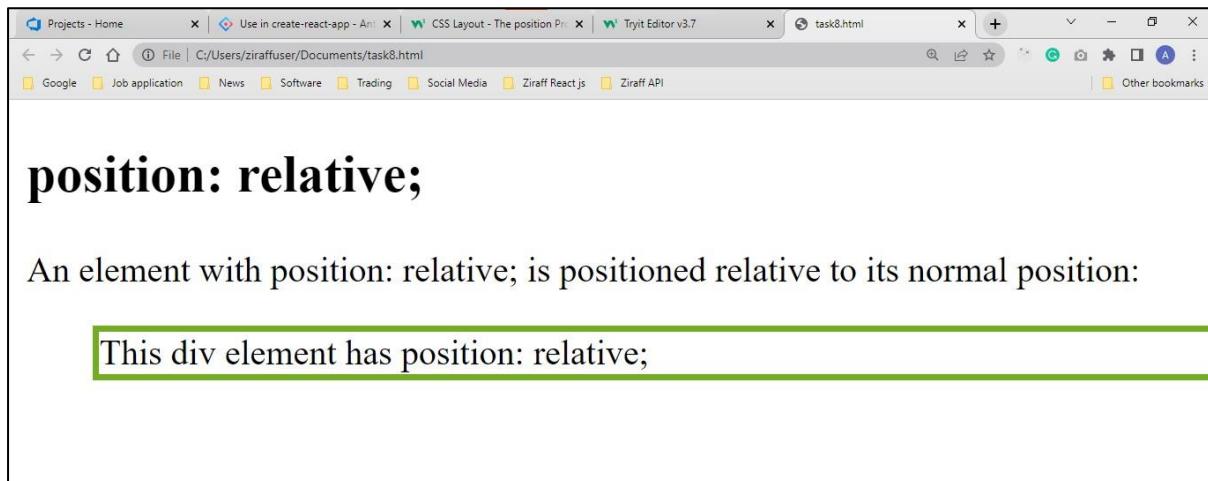
<h2>position: relative;</h2>

<p>An element with position: relative; is positioned relative to its normal position:</p>

<div class="relative">
This div element has position: relative;
</div>

</body>
</html>
```

Output:



14.2 Position-Absolute:

An element with **position: absolute;** is positioned relative to the nearest positioned ancestor (instead of positioned relative to the viewport, like fixed).

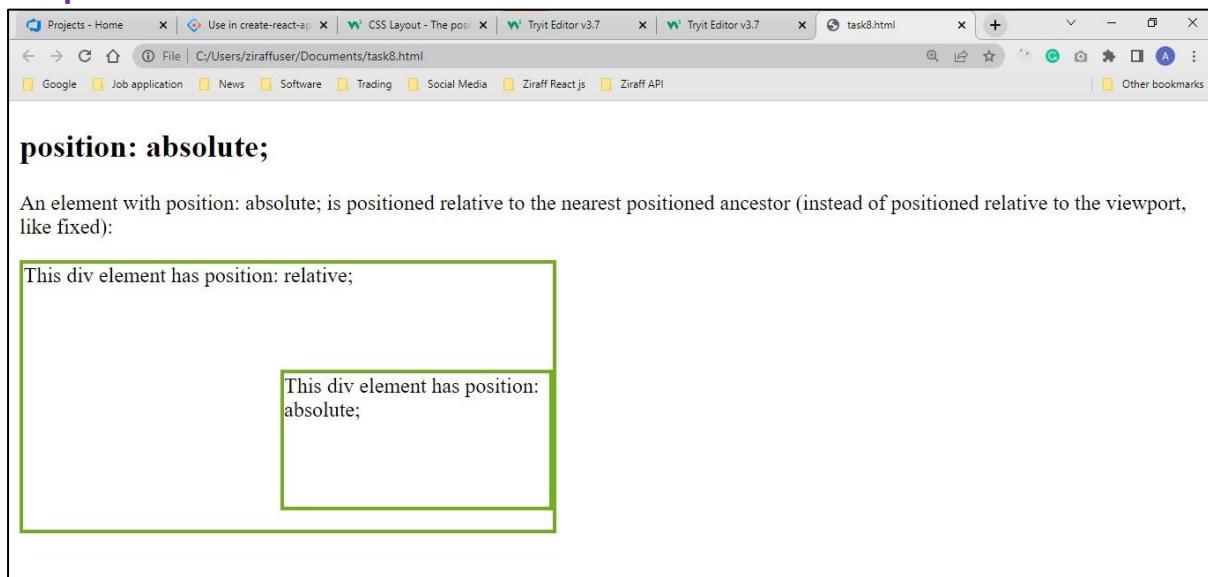
However; if an absolute positioned element has no positioned ancestors, it uses the document body, and moves along with page scrolling.

Note: Absolute positioned elements are removed from the normal flow, and can overlap elements.

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
div. relative {
position: relative;
width: 400px;
height: 200px;
border: 3px solid #73AD21;
}
div. absolute {
position: absolute;
top: 80px;
right: 0;
width: 200px;
height: 100px;
border: 3px solid #73AD21;
}
</style>
</head>
<body>
<h2>position: absolute;</h2><p>An element with position: absolute; is
positioned relative to the nearest positioned ancestor (instead of positioned
relative to the viewport, like fixed):</p>
<div class="relative">This div element has position: relative;
<div class="absolute">This div element has position: absolute;</div>
</div>
</body>
</html>
```

Output:



14.3 Position-Fixed:

An element with **position: fixed;** is positioned relative to the viewport, which means it always stays in the same place even if the page is scrolled. The top, right, bottom, and left properties are used to position the element.

A fixed element does not leave a gap in the page where it would normally have been located.

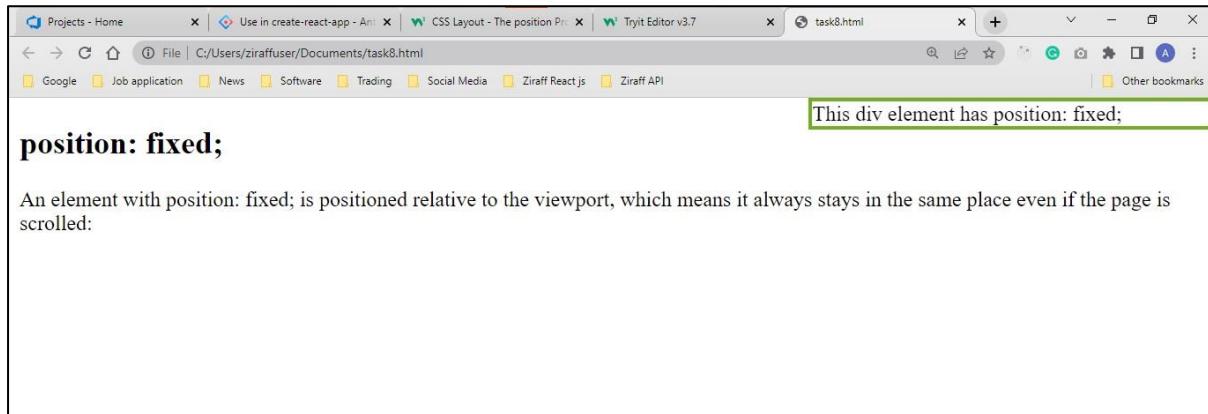
Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
div.fixed {
position: fixed;
top: 0;
right: 0;
width: 300px;
border: 3px solid #73AD21;
}
</style>
</head>
<body>
<h2>position: fixed;</h2>

<p>An element with position: fixed; is positioned relative to the viewport, which means it always stays in the same place even if the page is scrolled:</p>

<div class="fixed">
This div element has position: fixed;
</div>
</body>
</html>
```

Output:



14.4 Position-Static:

HTML elements are positioned static by default.

Static positioned elements are not affected by the top, bottom, left, and right properties.

An element with **position: static;** is not positioned in any special way; it is always positioned according to the normal flow of the page

Example:

```
<!DOCTYPE html>

<html>
<head>
<style>
div. static {
    position: static;
    border: 3px solid #73AD21;
}
</style>
</head>
<body>

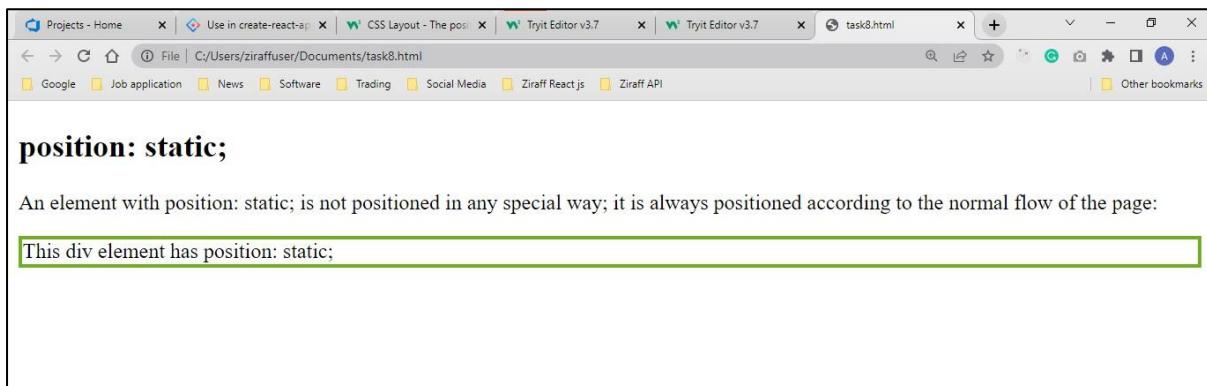
<h2>position: static;</h2>

<p>An element with position: static; is not positioned in any special way; it is always
positioned according to the normal flow of the page:</p>

<div class="static">
This div element has position: static;
</div>

</body>
</html>
```

Output:



14.5 Position-Sticky:

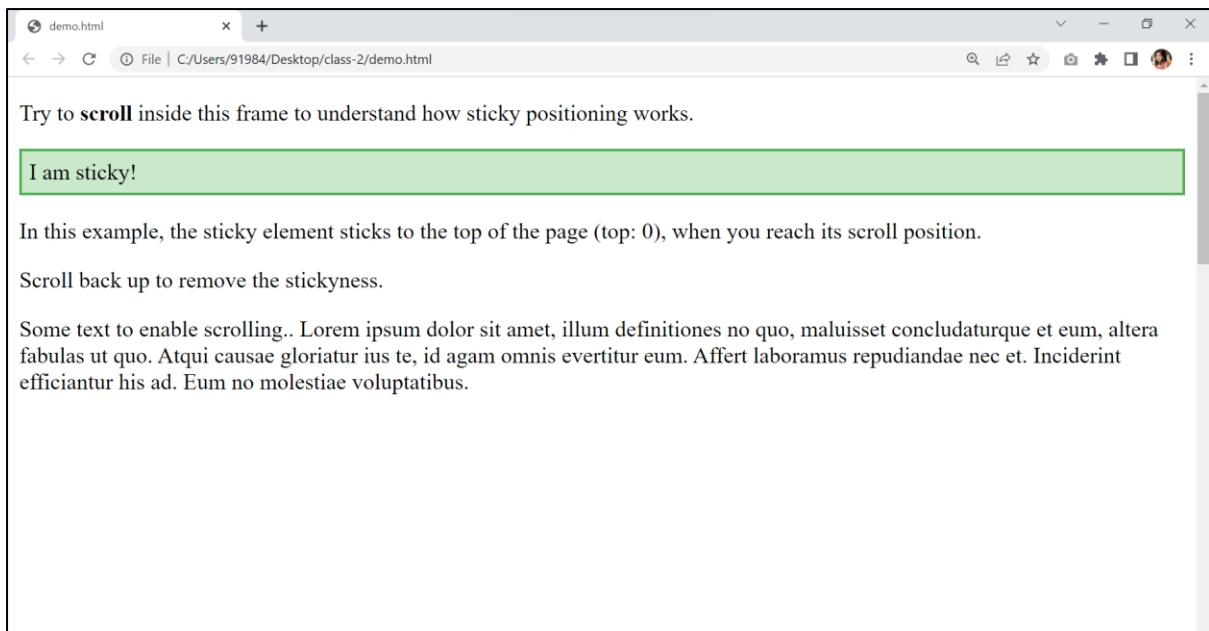
An element with **position: sticky;** is positioned based on the user's scroll position.

A sticky element toggles between **relative** and **fixed**, depending on the scroll position. It is positioned relative until a given offset position is met in the viewport - then it "sticks" in place (like position: fixed).

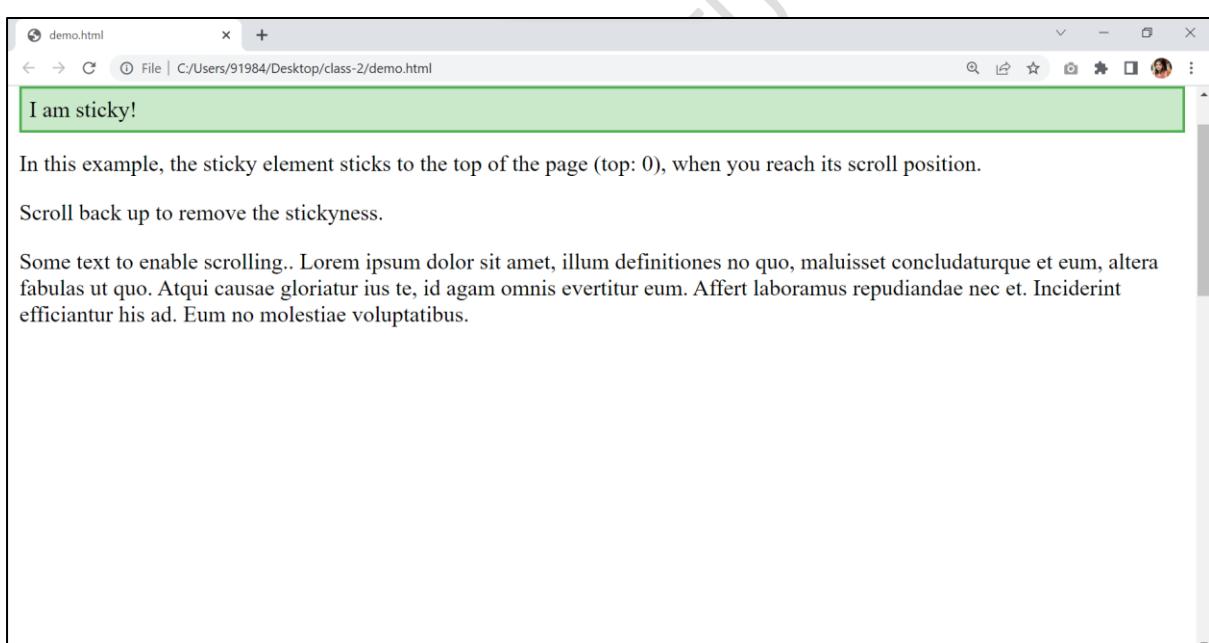
Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
div. sticky {
position:- webkit -sticky;
position: sticky;
top: 0;
padding: 5px;
background-color: #cae8ca;
border: 2px solid #4CAF50;
}
</style>
</head>
<body>
<p>Try to <b>scroll</b> inside this frame to understand how sticky positioning works.</p>
<div class="sticky">I am sticky!</div>
<div style="padding-bottom:1000px">
<p>In this example, the sticky element sticks to the top of the page (top: 0), when you reach its scroll position.</p>
<p>Scroll back up to remove the stickiness.</p>
<p>Some text to enable scrolling.. Lorem ipsum dolor sit amet, illum definitions no quo, maluisset concludaturque et eum, altera fabulas ut quo. Atqui causae gloriatur ius te, id agam omnis evertitur eum. Affert laboramus repudiandae nec et. Inciderint efficiantur his ad. Eum no molestiae voluptatibus. </p>
</div>
</body>
</html>
```

Output:



After scrolling the content in browser "**I am sticky**" content div stick at top observe below output.



14.6 Z-index:

The **z-index** property specifies the stack order of an element.

An element with greater stack order is always in front of an element with a lower stack order.

Note: z-index only works on positioned elements (**position: absolute**, **position: relative**, **position: fixed**, or **position: sticky**) and flex items (elements that are direct children of **display: flex** elements).

Note: If two positioned elements overlap without a z-index specified, the element positioned last in the HTML code will be shown on top.

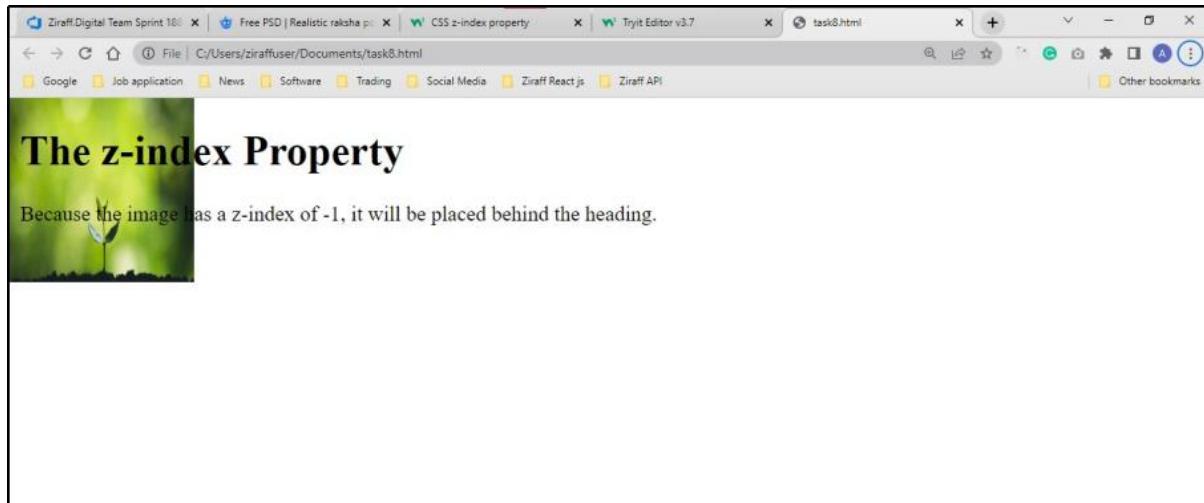
Example:

```
<!DOCTYPE html>

<html>
<head>
<style>
img {
    position: absolute;
    left: 0px;
    top: 0px;
    z-index: -1;
}
</style>
</head>
<body>
<h1>The z-index Property</h1>

<p>Because the image has a z-index of -1, it will be placed behind the heading.</p>
</body>
</html>
```

Output:



15.Overflow:

The **overflow** property specifies whether to clip the content or to add scrollbars when the content of an element is too big to fit in the specified area.

The **overflow** property has the following values:

- **visible** - Default. The overflow is not clipped. The content renders outside the element's box
- **hidden** - The overflow is clipped, and the rest of the content will be invisible
- **scroll** - The overflow is clipped, and a scrollbar is added to see the rest of the content
- **auto** - Similar to scroll, but it adds scrollbars only when necessary

Note: The overflow property only works for block elements with a specified height.

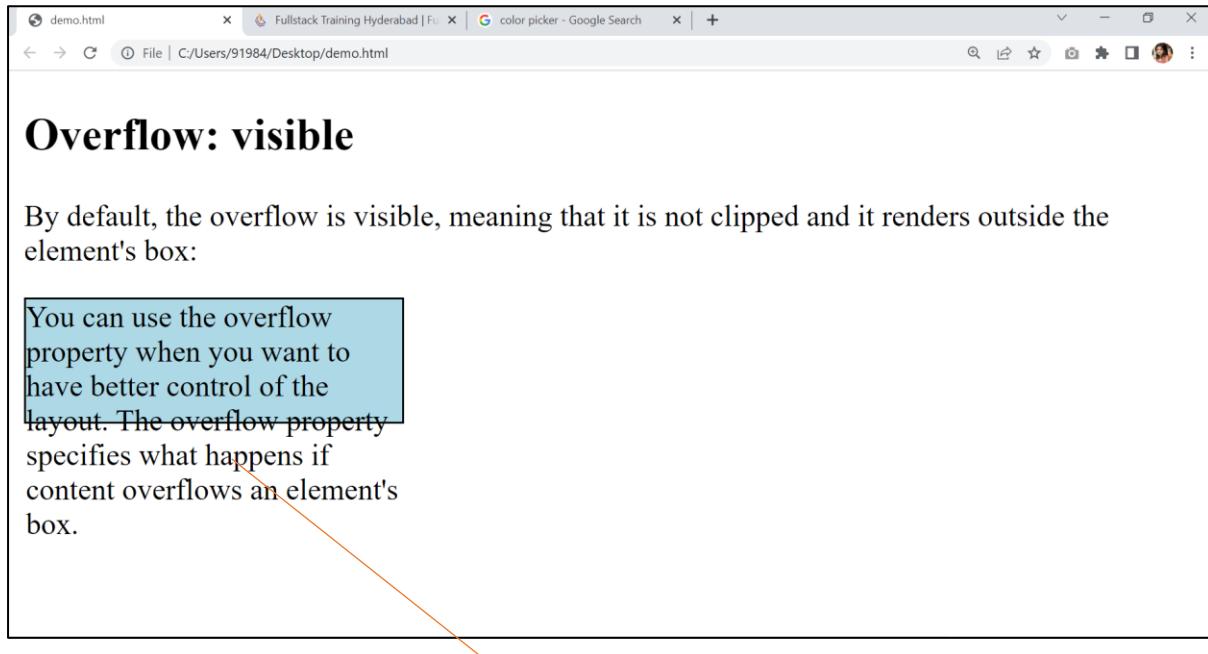
15.1 Overflow: visible

By default, the overflow is **visible**, meaning that it is not clipped and it renders outside the element's box:

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
div {
background-color: coral;
width: 200px;
height: 65px;order: 1px solid;
overflow: visible;}
</style>
</head>
<body>
<h2>Overflow: visible</h2>
<p>By default, the overflow is visible, meaning that it is not clipped and it renders outside the element's box:</p>
<div>You can use the overflow property when you want to have better control of the layout. The overflow property specifies what happens if content overflows an element's box.
</div>
</body>
</html>
```

Output:



The screenshot shows a browser window with three tabs: 'demo.html', 'Fullstack Training Hyderabad | Fu', and 'color picker - Google Search'. The main content area displays the heading 'Overflow: visible' and a paragraph of text. A red arrow points from the word 'content' in the text paragraph to a callout box labeled 'Overflow content'.

Overflow: visible

By default, the overflow is visible, meaning that it is not clipped and it renders outside the element's box:

You can use the overflow property when you want to have better control of the layout. The **overflow** property specifies what happens if content overflows an element's box.

Overflow content

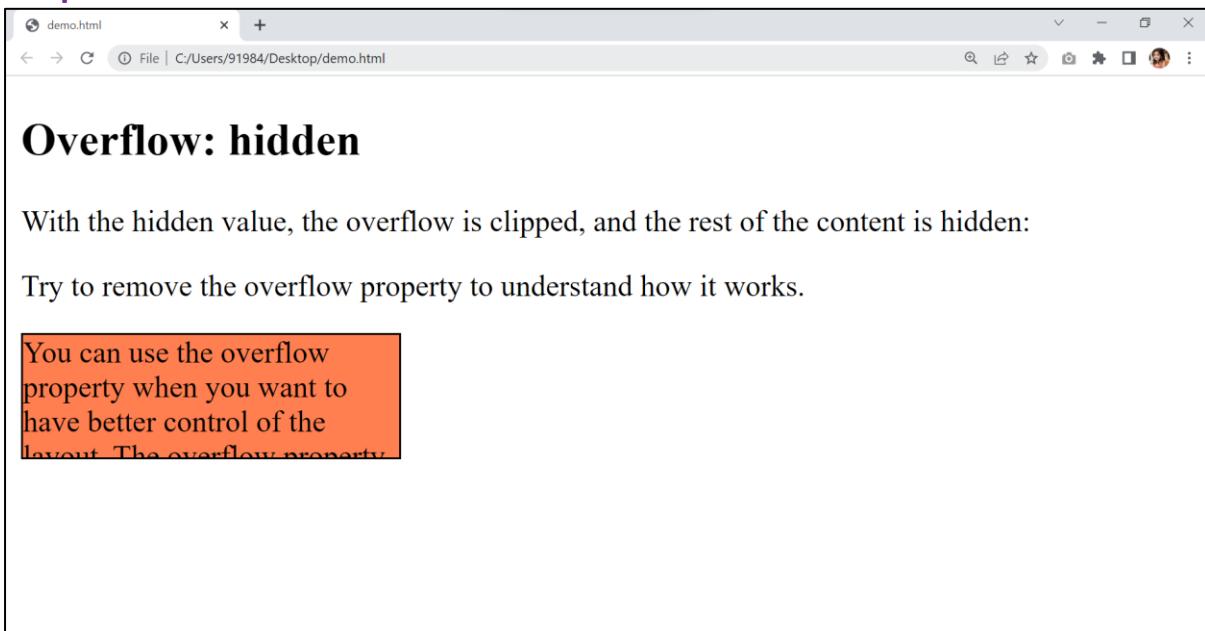
15.1 overflow: hidden:

With the **hidden** value, the overflow is clipped, and the rest of the content is hidden

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
div {
background-color: coral;
width: 200px;
height: 65px;order: 1px solid;
overflow: hidden;}
</style>
</head>
<body>
<h2>Overflow: hidden</h2>
<p> With the hidden value, the overflow is clipped, and the rest of the content is hidden:</p>
<p>Try to remove the overflow property to understand how it works.</p>
<div>You can use the overflow property when you want to have better control of the layout. The overflow property specifies what happens if content overflows an element's box.
</div>
</body>
</html>
```

Output:



demo.html

File | C:/Users/91984/Desktop/demo.html

Overflow: hidden

With the hidden value, the overflow is clipped, and the rest of the content is hidden:

Try to remove the overflow property to understand how it works.

You can use the overflow property when you want to have better control of the layout. The overflow property

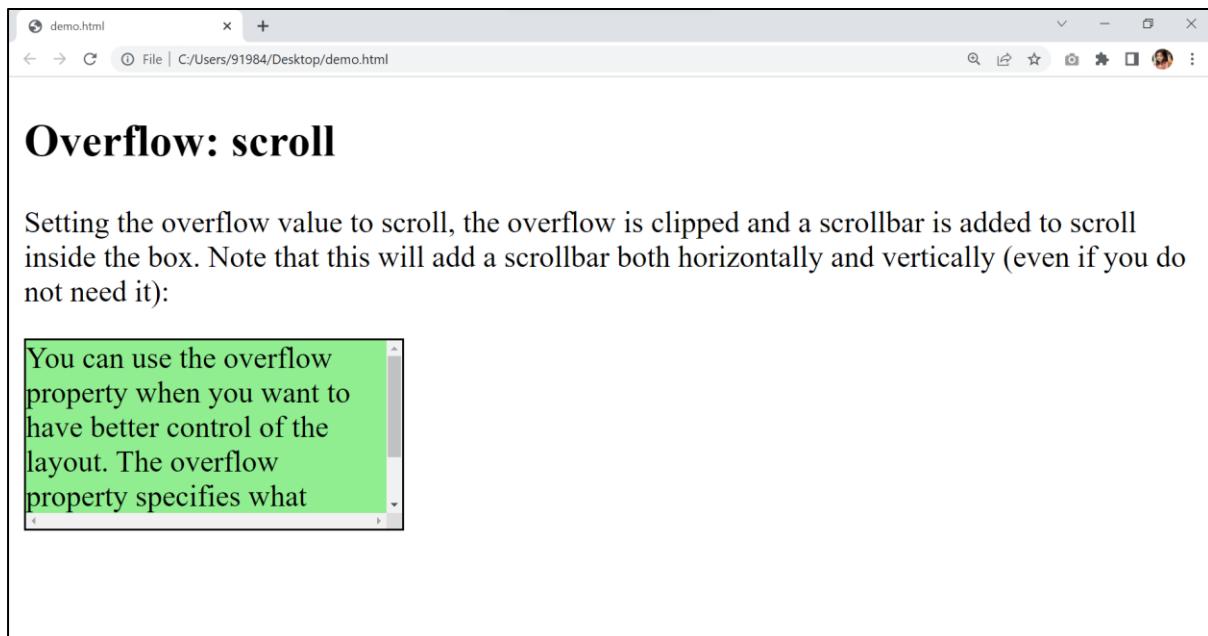
15.3 overflow: scroll

Setting the value to **scroll**, the overflow is clipped and a scrollbar is added to scroll inside the box. Note that this will add a scrollbar both horizontally and vertically (even if you do not need it).

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
div {
    background-color: lightgreen;
    width: 200px;
    height: 100px;
    border: 1px solid black;
    overflow: scroll;
}
</style>
</head>
<body>
<h2>Overflow: scroll</h2>
<p>Setting the overflow value to scroll, the overflow is clipped and a scrollbar is added to scroll inside the box. Note that this will add a scrollbar both horizontally and vertically (even if you do not need it):</p>
<div>You can use the overflow property when you want to have better control of the layout. The overflow property specifies what happens if content overflows an element's box.</div>
</body>
</html>
```

Output:

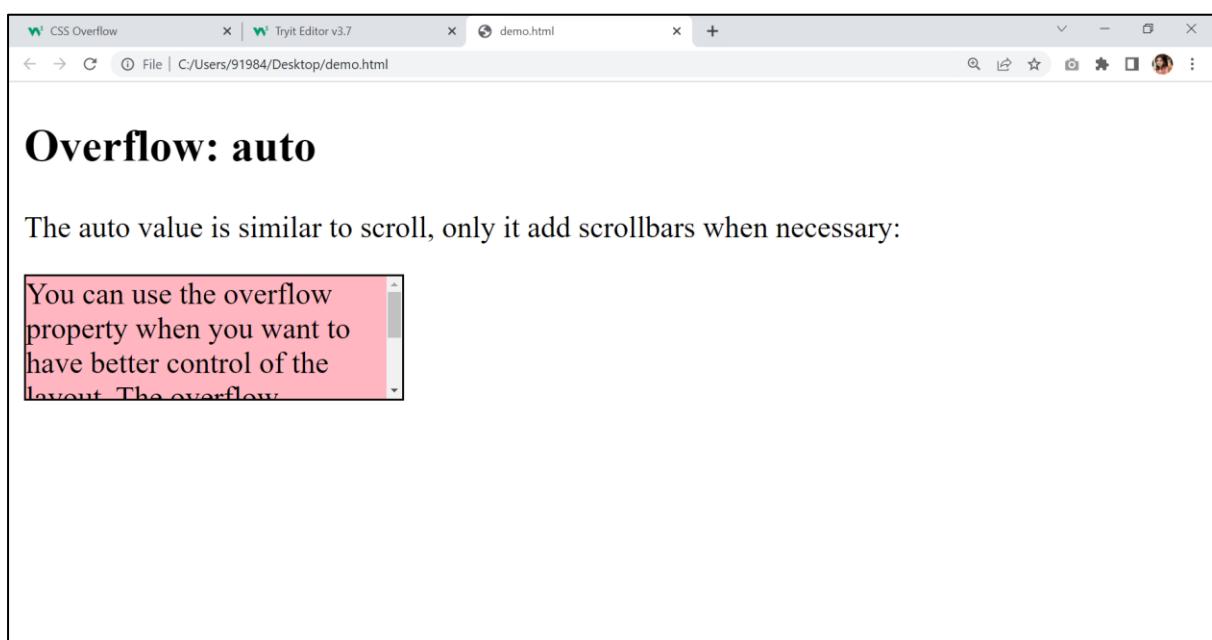


15.4 Overflow: auto

The **auto** value is similar to **scroll**, but it adds scrollbars only when necessary.

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
div {
    background-color: lightpink;
    width: 200px;
    height: 65px;
    border: 1px solid black;
    overflow: auto;
}
</style>
</head>
<body>
<h2>Overflow: auto</h2>
<p>The auto value is similar to scroll, only it add scrollbars when necessary:</p>
<div>You can use the overflow property when you want to have better control of the layout. The overflow property specifies what happens if content overflows an element's box.</div>
</body>
</html>
```

Output:

The auto value is similar to scroll, only it add scrollbars when necessary:

You can use the overflow property when you want to have better control of the layout. The overflow

16.float and clear:

The CSS **float** property specifies how an element should float.

The CSS **clear** property specifies what elements can float beside the cleared element and on which side.



16.1 float Property

- left
- right
- none

float: left:

The element floats to the left of its container

Example:

```
<!DOCTYPE html>

<html>
<head>
<style>
img {
    float: left;
}
</style>
</head>
<body>

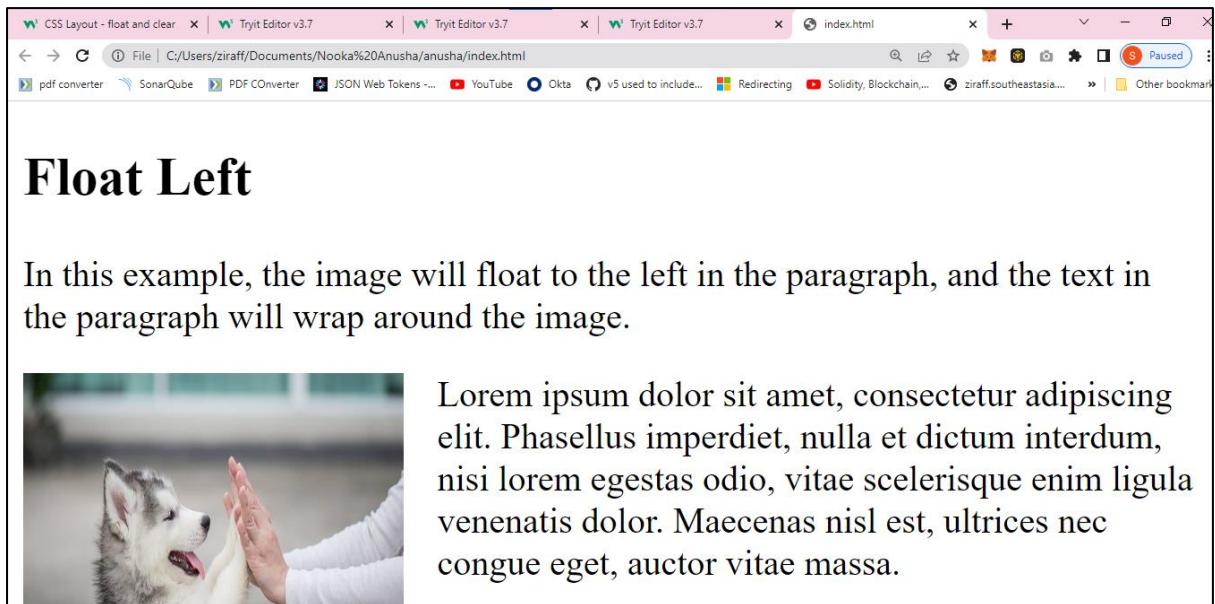
<h2>Float Left</h2>

<p>In this example, the image will float to the left in the paragraph, and the text in the paragraph will wrap around the image. </p>

<p>
    Lorem ipsum dolor sit amet, consectetur adipiscing elit. Phasellus imperdiet, nulla et
    dictum interdum, nisi lorem egestas odio, vitae scelerisque enim ligula venenatis dolor.
    Maecenas nisl est, ultrices nec congue eget, auctor vitae massa. </p>

</body>
</html>
```

Output:



Float Left

In this example, the image will float to the left in the paragraph, and the text in the paragraph will wrap around the image.



Lorem ipsum dolor sit amet, consectetur adipiscing elit. Phasellus imperdiet, nulla et dictum interdum, nisi lorem egestas odio, vitae scelerisque enim ligula venenatis dolor. Maecenas nisl est, ultrices nec congue eget, auctor vitae massa.

16.2 float: right:

The element floats to the right of its container

Example:

```
<!DOCTYPE html>

<html>
<head>
<style>
img {
    float: right;
}
</style>
</head>
<body>

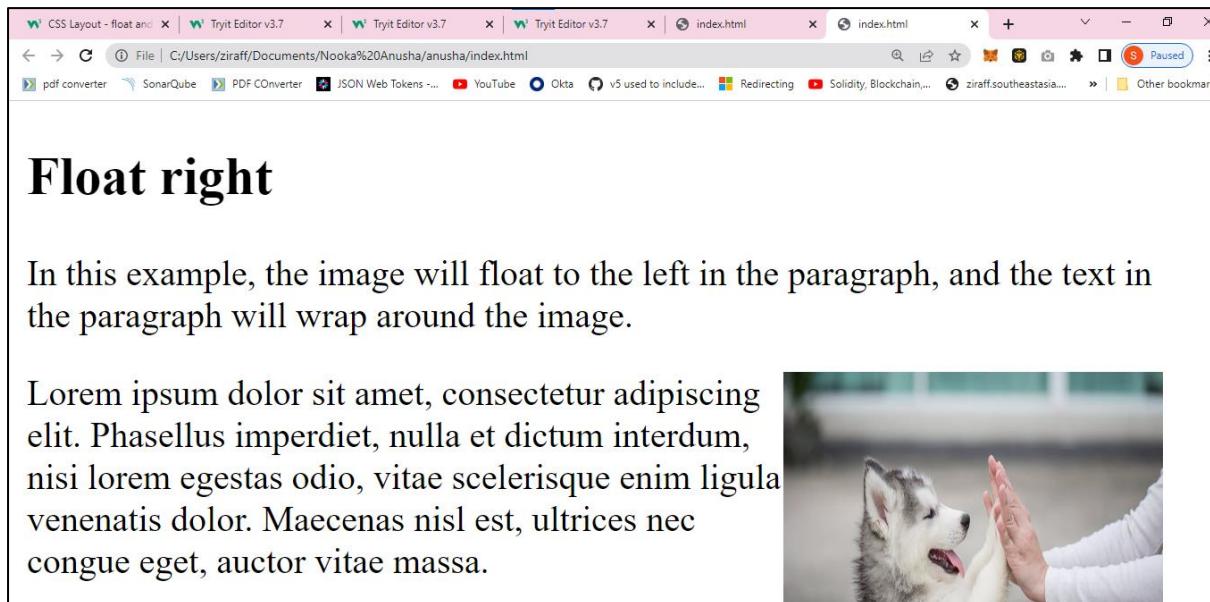
<h2>Float right</h2>

<p>In this example, the image will float to the left in the paragraph, and the text in the paragraph will wrap around the image. </p>

<p>
    Lorem ipsum dolor sit amet, consectetur adipiscing elit. Phasellus imperdiet, nulla et
    dictum interdum, nisi lorem egestas odio, vitae scelerisque enim ligula venenatis dolor.
    Maecenas nisl est, ultrices nec congue eget, auctor vitae massa. </p>

</body>
</html>
```

Output:



16.3 Float: none:

The element does not float (will be displayed just where it occurs in the text). This is default

Example:

```
<!DOCTYPE html>

<html>
<head>
<style>
img {
    float: none;
}
</style>
</head>
<body>

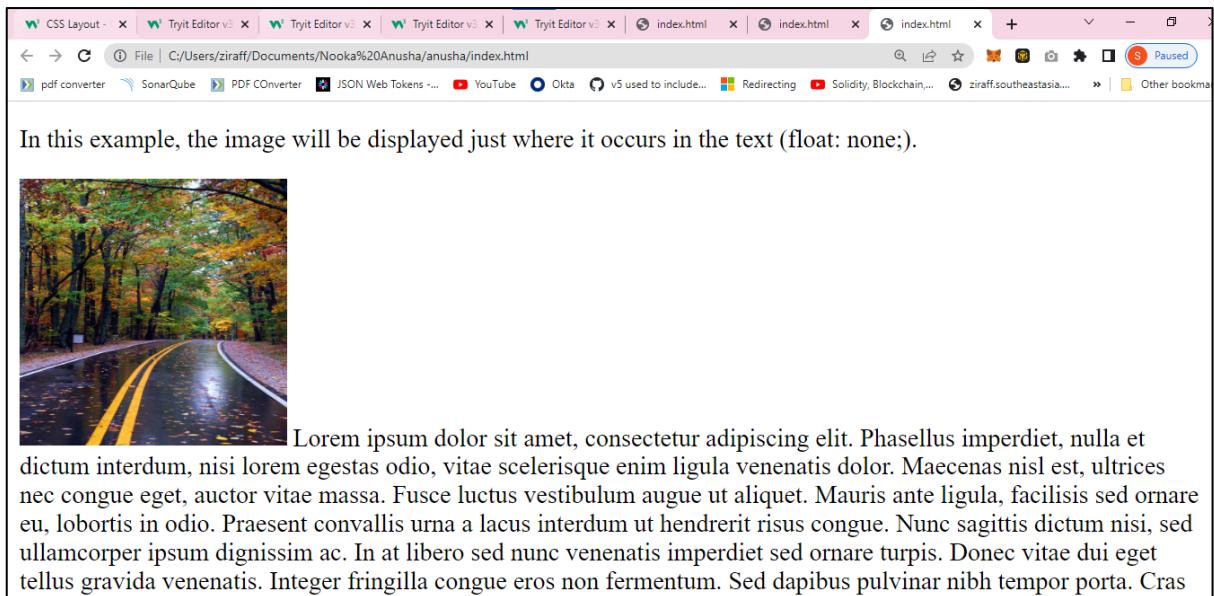
<h2>Float None</h2>

<p>In this example, the image will be displayed just where it occurs in the text (float: none;).</p>

<p>
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Phasellus imperdiet, nulla et
dictum interdum, nisi lorem egestas odio, vitae scelerisque enim ligula venenatis dolor.
Maecenas nisl est, ultrices nec congue eget, auctor vitae massa. Fusce luctus
vestibulum augue ut aliquet. Mauris ante ligula, facilisis sed ornare eu, lobortis in odio.
Praesent convallis urna a lacus interdum ut hendrerit risus congue. Nunc sagittis dictum
nisi, sed ullamcorper ipsum dignissim ac. In at libero sed nunc venenatis imperdiet sed
ornare turpis. Donec vitae dui eget tellus gravida venenatis. Integer fringilla congue
eros non fermentum. Sed dapibus pulvinar nibh tempor porta. Cras ac leo purus.
Mauris quis diam velit.</p>

</body>
</html>
```

Output:



16.4 Clear:

When we use the **float** property, and we want the next element below (not on right or left), we will have to use the **clear** property.

The **clear** property specifies what should happen with the element that is next to a floating element.

The **clear** property can have one of the following values:

- **none** - The element is not pushed below left or right floated elements. This is default
 - **left** - The element is pushed below left floated elements
 - **right** - The element is pushed below right floated elements
 - **both** - The element is pushed below both left and right floated elements

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
div {
    border: 3px solid #4CAF50;
    padding: 5px;
}
.img1 {
    float: right;
}
.img2 {
    float: right;
}
.clearfix {
    overflow: auto;
}
</style>
</head>
<body>
<h2>Without Clearfix</h2>
<p>This image is floated to the right. It is also taller than the element containing it, so it overflows outside of its container:</p>
<div>
    
    Lorem ipsum dolor sit amet, consectetur adipiscing elit. Phasellus imperdiet...
</div>

<h2 style="clear: right">With Clearfix</h2>
<p>We can fix this by adding a clearfix class with overflow: auto; to the containing element:</p>

<div class="clearfix">
    
    Lorem ipsum dolor sit amet, consectetur adipiscing elit. Phasellus imperdiet...
</div>

</body>
</html>
```

Output:

Without Clearfix

This image is floated to the right. It is also taller than the element containing it, so it overflows outside of its container:

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Phasellus imperdiet...



With Clearfix

We can fix this by adding a clearfix class with overflow: auto; to the containing element:

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Phasellus imperdiet...



17. Backgrounds

The CSS background properties are used to add background effects for elements.

- **background-color**
- **background-image**
- **background-repeat**
- **background-position**

17.1 background-color:

The **background-color** property specifies the background color of an element.

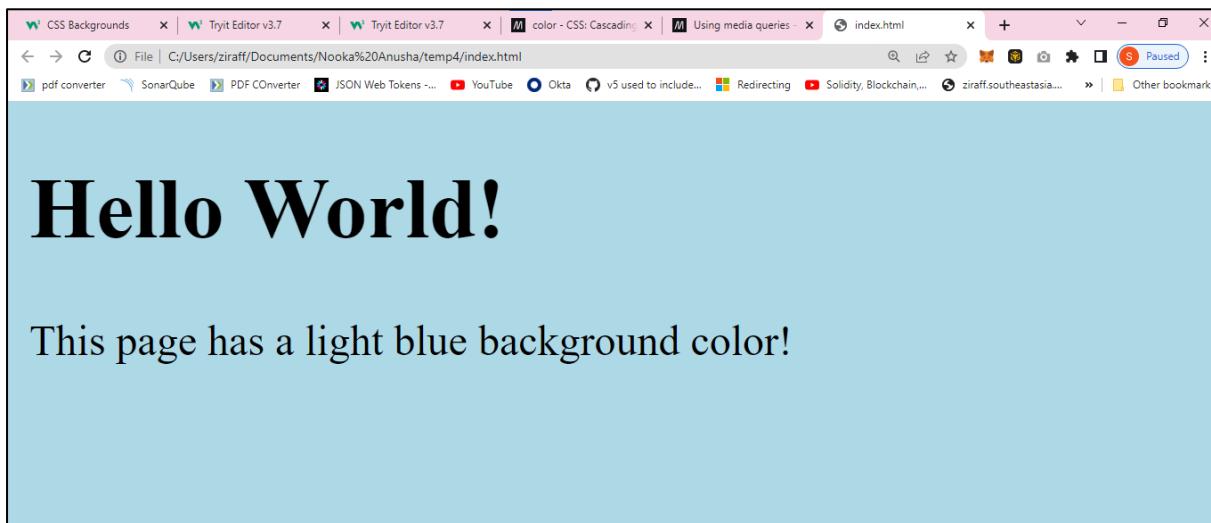
- a valid color name - like "red"
- a HEX value - like "#ff0000"
- an RGB value - like "rgb(255,0,0)"

Example:

```
<!DOCTYPE html>

<html>
<head>
<style>
body {
    background-color: lightblue;
}
</style>
</head>
<body>
<h1>Hello World! </h1>
<p>This page has a light blue background color! </p>
</body>
</html>
```

Output:



17.2 Background Image:

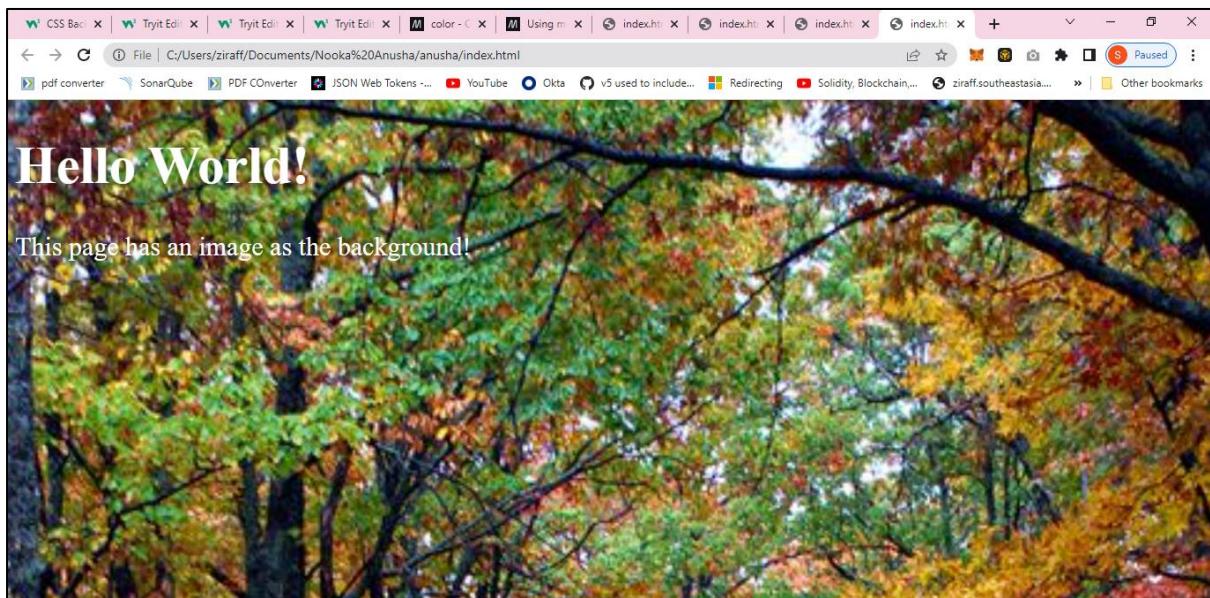
The **background-image** property specifies an image to use as the background of an element. By default, the image is repeated so it covers the entire element.

Example:

```
<!DOCTYPE html>

<html>
  <head>
    <style>
      body {background-image: url("image.jpg");
            color:white;
            font-size:30px;
          }
    </style>
  </head>
  <body>
    <h1>Hello World!</h1>
    <p>This page has an image as the background!</p>
  </body>
</html>
```

Output:



17.3 Background Image Repeat:

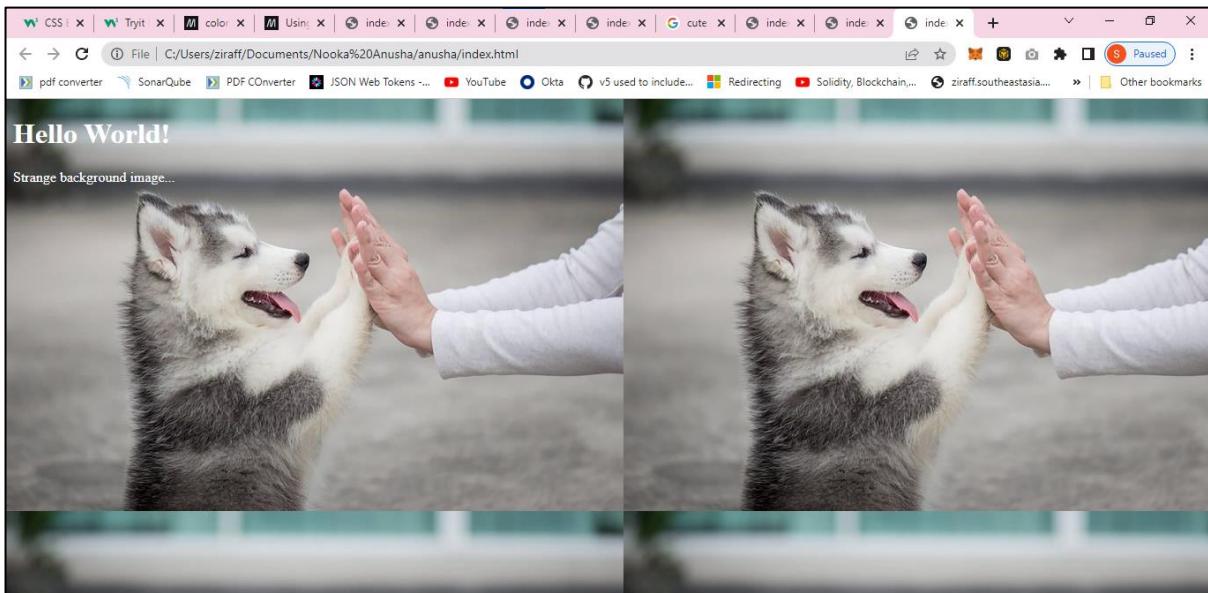
By default, the **background-image** property repeats an image both horizontally and vertically.

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
body {
background-image: url("image2.jpg");
color: white;

}
</style>
</head>
<body>
<h1>Hello World! </h1>
<p>Strange background image...</p>
</body>
</html>
```

Output:



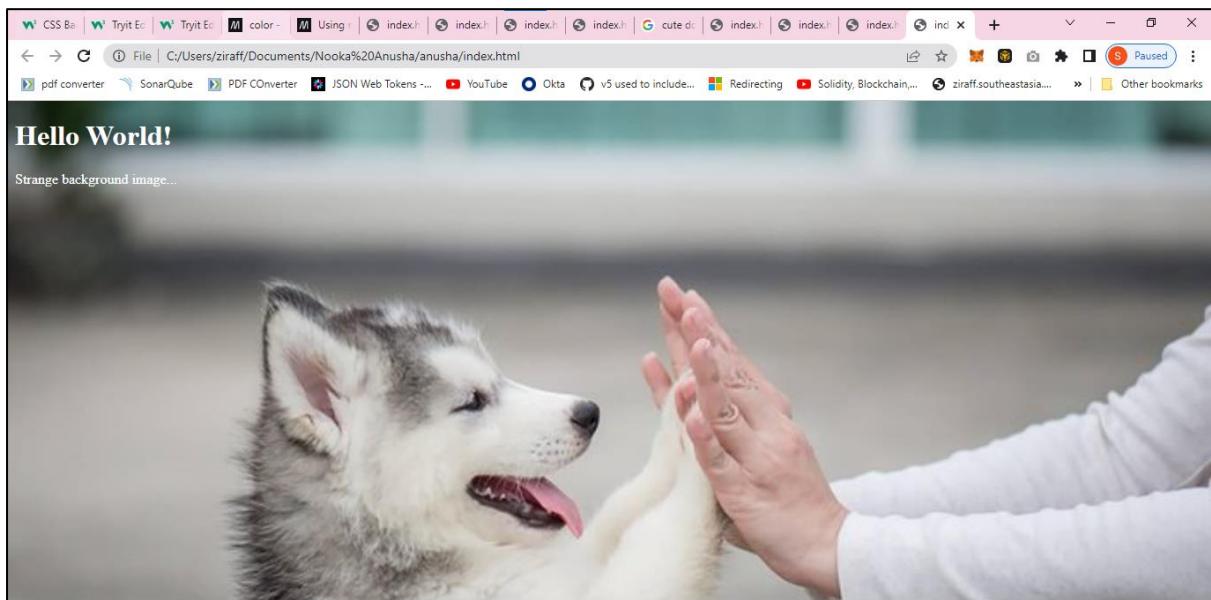
17.4 background-repeat: no-repeat:

Showing the background image only once is also specified by the **background-repeat** property

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
body {
background-image: url("image2.jpg")
background-repeat: no-repeat;
background-size: cover;
color:white;
}
</style>
</head>
<body>
<h1>Hello World! </h1>
<p>Strange background image...</p>
</body>
</html>
```

Output:



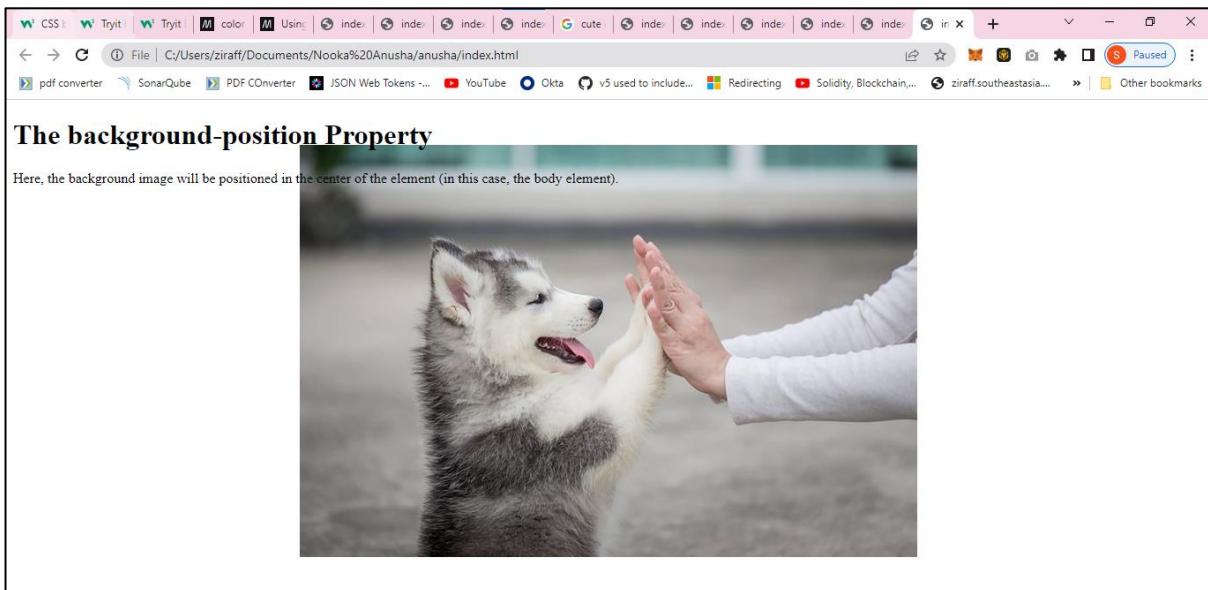
17.4 background-position:

The **background-position** property sets the starting position of a background image.

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
body {
background-image: url(image2.jpg);
background-repeat: no-repeat;
background-attachment: fixed;
background-position: center;
color:white;
}
</style>
</head>
<body>
<h1>Hello World! </h1>
<p>Strange background image...</p>
</body>
</html>
```

Output:



18.Transforms:

The **transform** property applies a 2D or 3D transformation to an element. This property allows you to rotate, scale, move, skew, etc., elements.

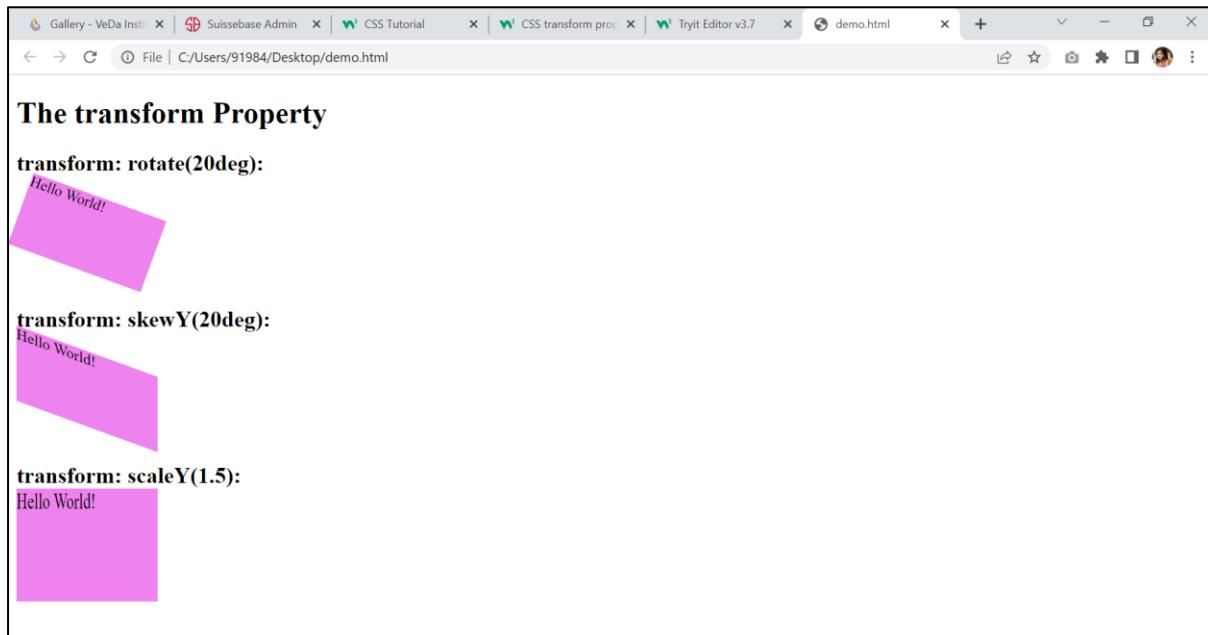
- translate ()
- rotate ()
- scaleX ()
- scaleY ()
- scale ()
- skewX ()
- skewY ()
- skew ()
- matrix ()

Mainly we use rotate () and scale () methods.

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
div.a {
    width: 150px;
    height: 80px;
    background-color: violet ;
    -ms-transform: rotate(20deg); /* IE 9 */
    transform: rotate(20deg);}
div.b {
    width: 150px;
    height: 80px;
    background-color: violet ;
    -ms-transform: skewY(20deg); /* IE 9 */
    transform: skewY(20deg);
}
div.c {
    width: 150px;
    height: 80px;
    background-color: violet ;
    -ms-transform: scaleY(1.5); /* IE 9 */
    transform: scaleY(1.5);
}
</style>
</head>
<body>
<h1>The transform Property</h1>
<h2>rotate(20deg):</h2>
<div class="a">Hello World!</div>
<br>
<h2>transform: skewY(20deg):</h2>
<div class="b">Hello World!</div>
<br>
<h2>transform: scaleY(1.5):</h2>
<div class="c">Hello World!</div>
</body>
</html>
```

Output:



19.! Important:

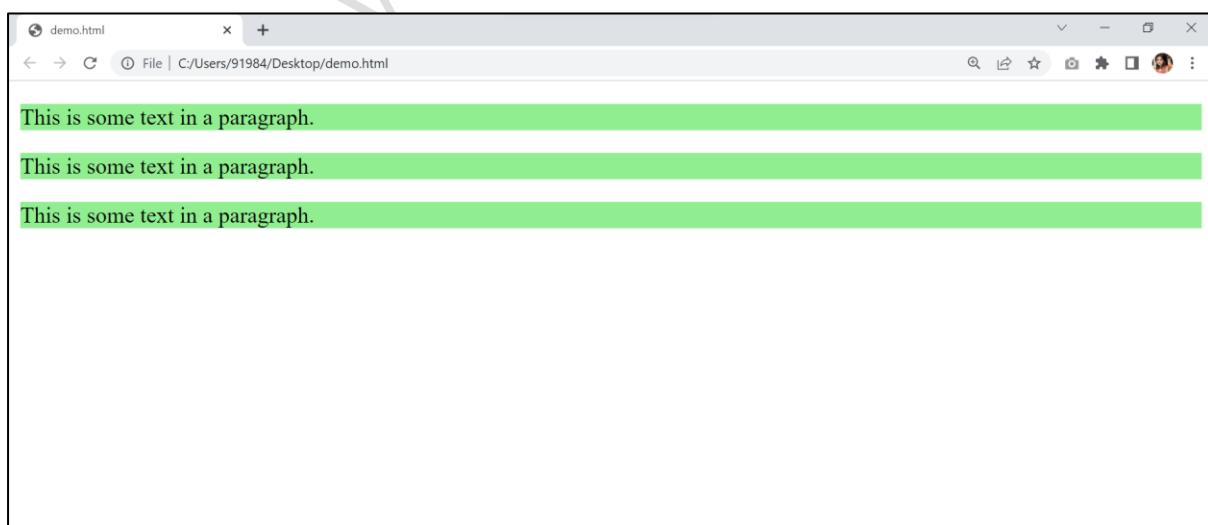
The **!important** rule in CSS is used to add more importance to a property/value than normal.

In fact, if you use the **!important** rule, it will override ALL previous styling rules for that specific property on that element!

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>#myid {
background-color: blue;
}
.myclass {
background-color: gray;
}
p {
background-color: lightgreen !important;
}
</style>
</head>
<body>
<p>This is some text in a paragraph.</p>
<p class="myclass">This is some text in a paragraph.</p>
<p id="myid">This is some text in a paragraph.</p>
</body>
</html>
```

Output:



20.Horizontal & Vertical Align:

We can arrange elements and text horizontally and vertically by using different properties like **margin**, **text-align**, **padding** etc....

20.1 Center Align Elements:

To horizontally center a block element (like `<div>`), use **margin: auto;**

Setting the width of the element will prevent it from stretching out to the edges of its container.

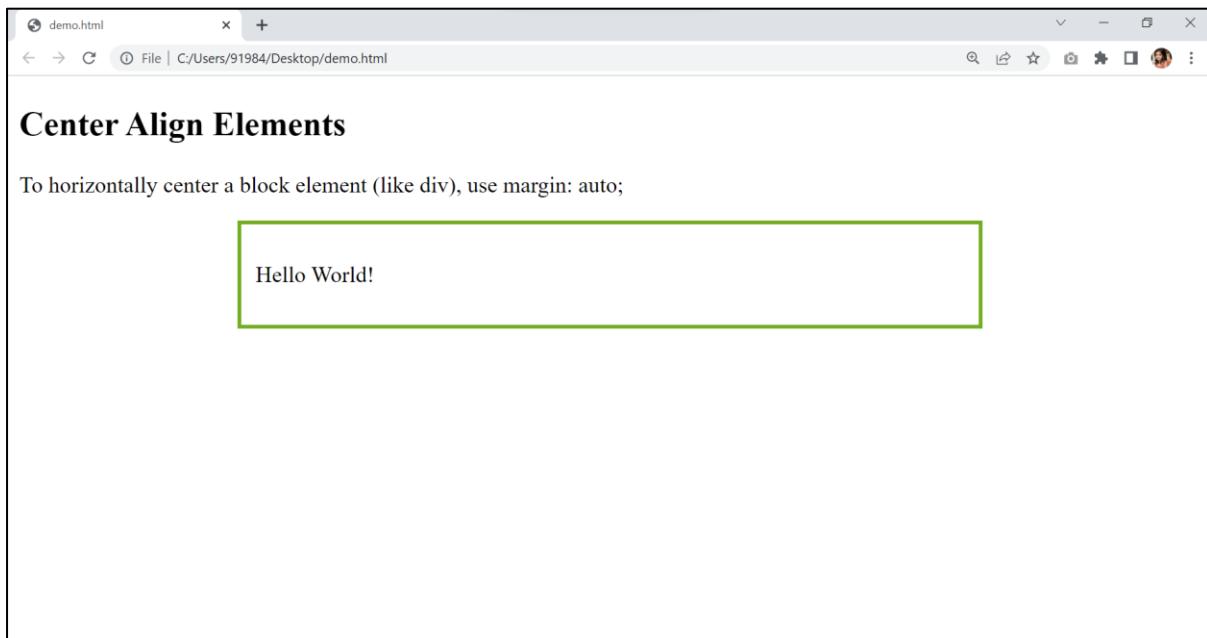
The element will then take up the specified width, and the remaining space will be split equally between the two margins:

Note: Center aligning has no effect if the **width** property is not set (or set to 100%).

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
.center {
margin: auto;
width: 60%;
border: 3px solid #73AD21;
padding: 10px;
}
</style>
</head>
<body>
<h2>Center Align Elements</h2>
<p>To horizontally center a block element (like div), use margin: auto;</p>
<div class="center">
<p>Hello World!</p>
</div>
</body>
```

Output:

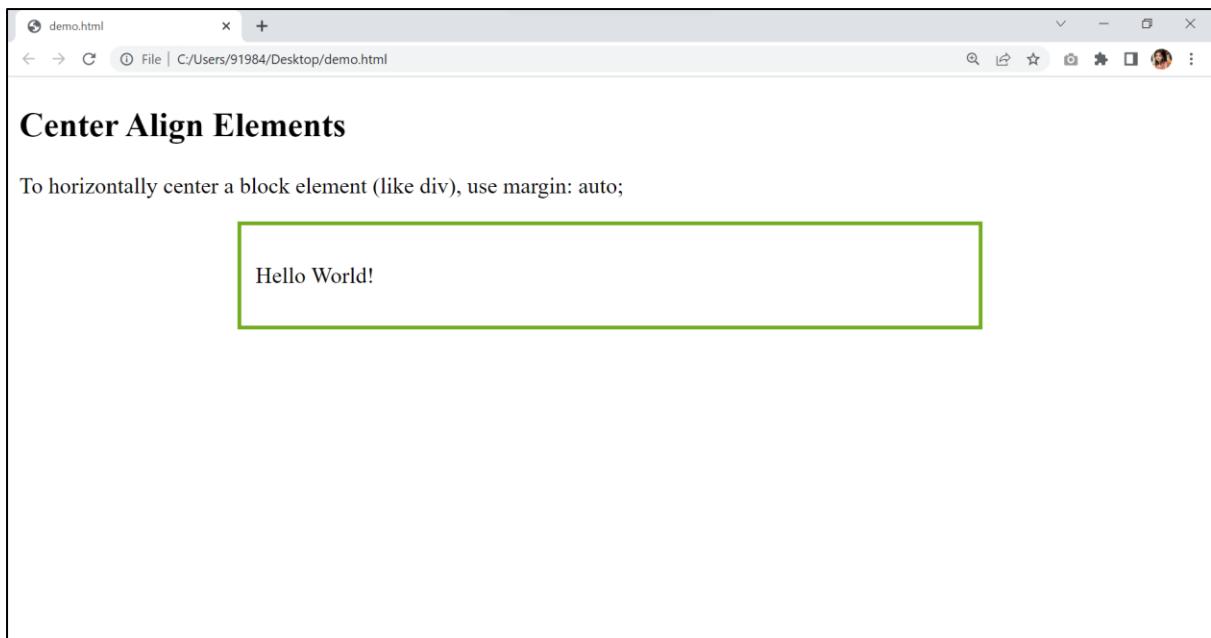


20.2 Center Align Text:

To just center the text inside an element, use **text-align: center;**

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
.center {
    text-align: center;
    border: 3px solid green;
}
</style>
</head>
<body>
<h2>Center Text</h2>
<div class="center">
<p>This text is centered.</p>
</div>
</body>
</html>
```



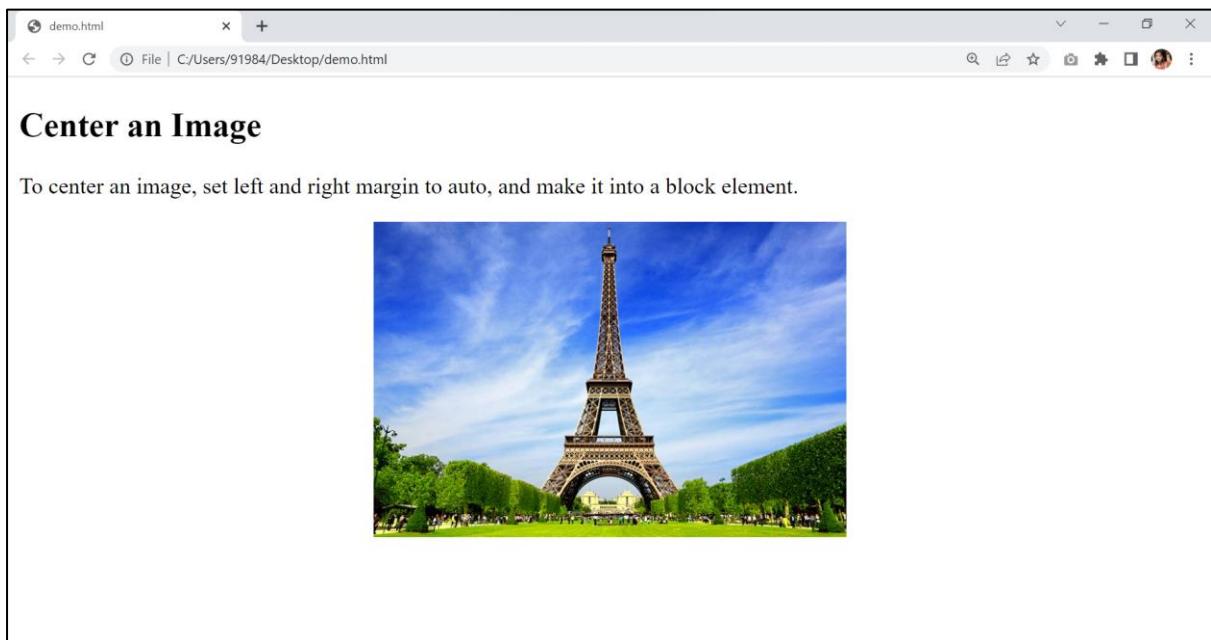
20.3 Center an Image:

To center an image, set left and right margin to **auto** and make it into a **block** element:

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
{
  display: block;
  margin-left: auto;
  margin-right: auto;
}
</style>
</head>
<body>
<h2>Center an Image</h2>
<p>To center an image, set left and right margin to auto, and make it into a block element.</p>

</body>
</html>
```

Output:

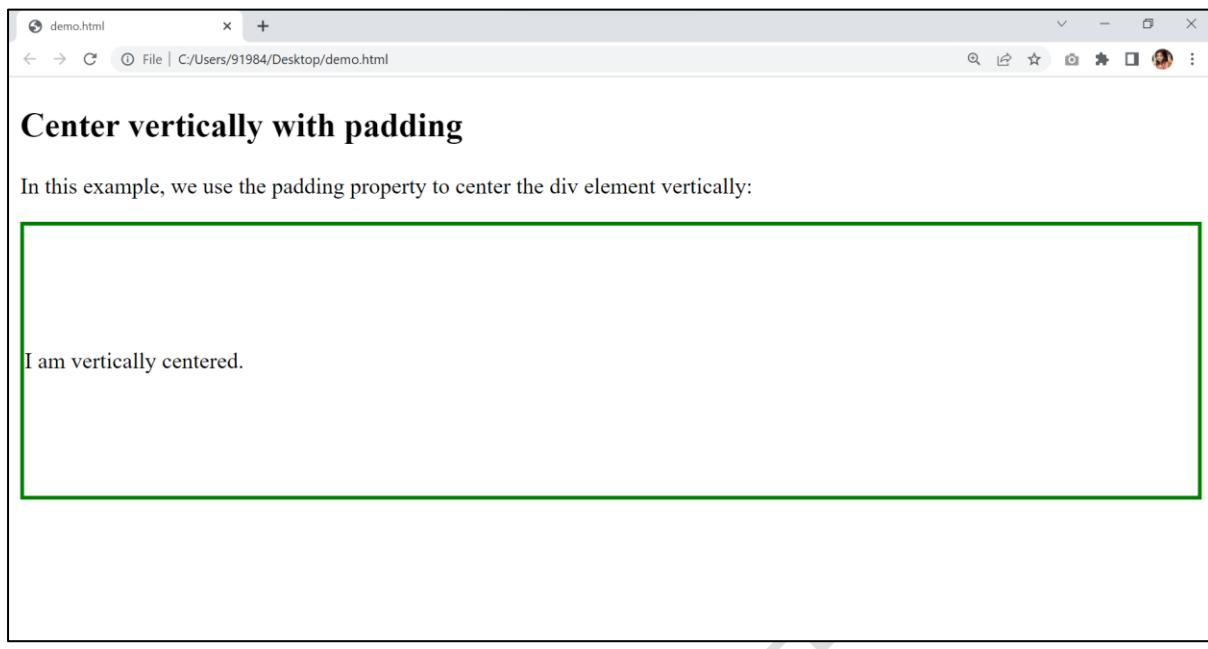
20.4 Center Vertically - Using padding:

There are many ways to center an element vertically in CSS. A simple solution is to use top and bottom **padding**:

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
.center {
  padding: 70px 0;
  border: 3px solid green;
}
</style>
</head>
<body>
<h2>Center vertically with padding</h2>
<p>In this example, we use the padding property to center the div element vertically:</p>
<div class="center">
  <p>I am vertically centered.</p>
</div>
</body>
</html>
```

Output:

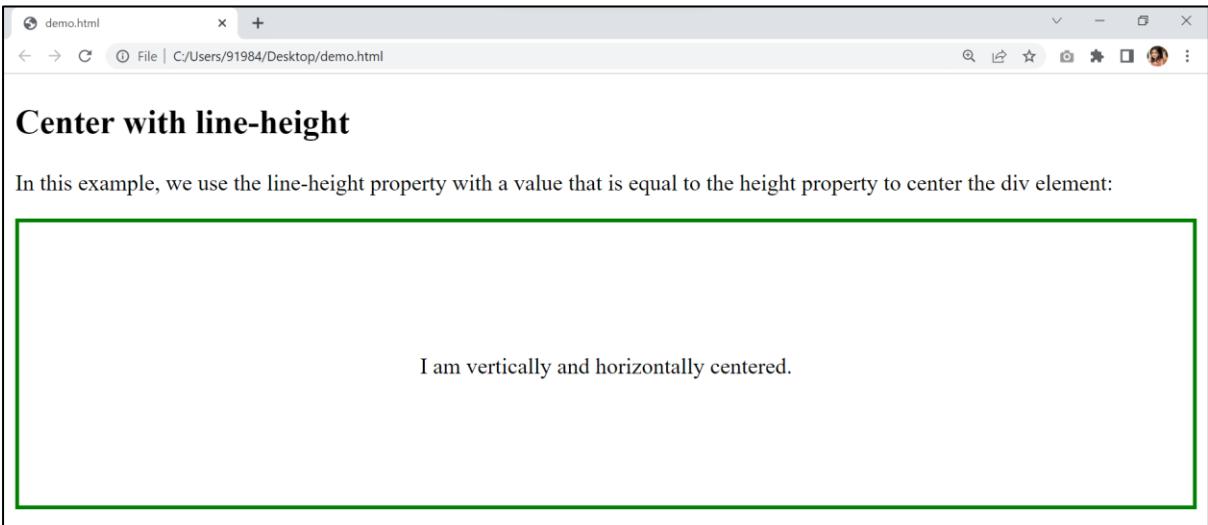


20.5 Center Vertically - Using line-height:

Another trick is to use the **line-height** property with a value that is equal to the **height** property:

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
.center {
    line-height: 200px;
    height: 200px;
    border: 3px solid green;
    text-align: center;
}
.center p {
    line-height: 1.5;
    display: inline-block;
    vertical-align: middle;
}
</style>
</head>
<body>
<h2>Center with line-height</h2>
<p>In this example, we use the line-height property with a value that is equal to the height property to center the div element:</p>
<div class="center">
    <p>I am vertically and horizontally centered.</p>
</div>
</body>
</html>
```

Output:

demo.html

In this example, we use the line-height property with a value that is equal to the height property to center the div element:

I am vertically and horizontally centered.

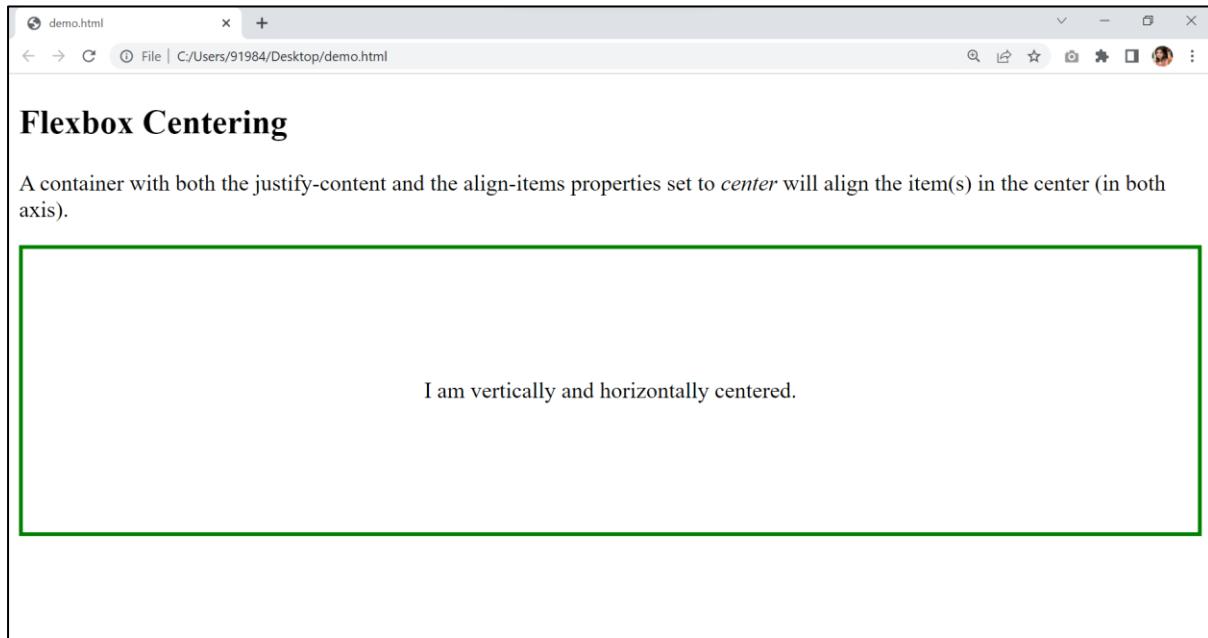
20.6 Center Vertically - Using Flexbox:

You can also use flexbox to center things. Just note that flexbox is not supported in IE10 and earlier versions:

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
.center {
  display: flex;
  justify-content: center;
  align-items: center;
  height: 200px;
  border: 3px solid green;
}
</style>
</head>
<body>
<h2>Flexbox Centering</h2>
<p>A container with both the justify-content and the align-items properties set to <em>center</em> will align the item(s) in the center (in both axis).</p>
<div class="center">
<p>I am vertically and horizontally centered.</p>
</div>
</body>
</html>
```

Output:



21. Media Queries

Media queries in CSS3 extended the CSS2 media types idea: Instead of looking for a type of device, they look at the capability of the device.

Media queries can be used to check many things, such as:

- width and height of the viewport
- width and height of the device
- orientation (is the tablet/phone in landscape or portrait mode?)
- resolution

Using media queries are a popular technique for delivering a tailored style sheet to desktops, laptops, tablets, and mobile phones (such as iPhone and Android phones).

21.1 Media Types:

The **@media** rule, introduced in CSS2, made it possible to define different style rules for different media types.

Examples: You could have one set of style rules for computer screens, one for printers, one for handheld devices, one for television-type devices, and so on.

Unless you use the **not** or **only** operators, the media type is optional and the **all** type will be implied.

Value	Description
All	Used for all media type devices
Print	Used for printers
screen	Used for computer screens, tablets, smart-phones etc.
speech	Used for screen readers that "reads" the page out loud

21.2. Media Query Syntax

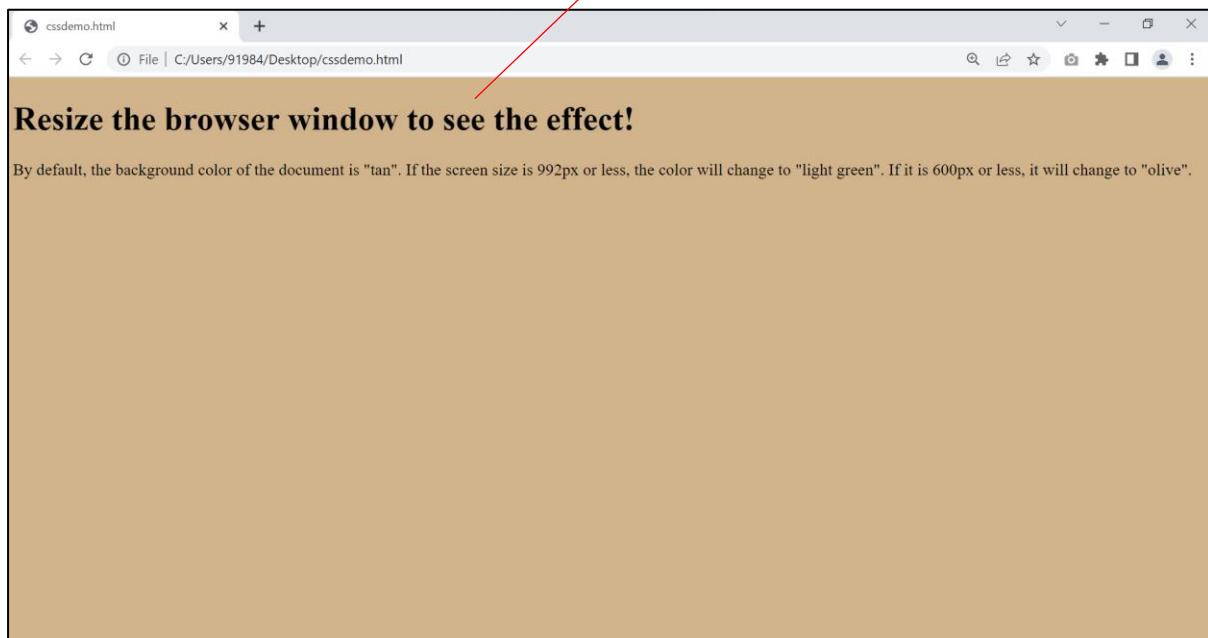
```
@media not|only mediatype and (expressions) {
    CSS-Code;
}
```

Example:

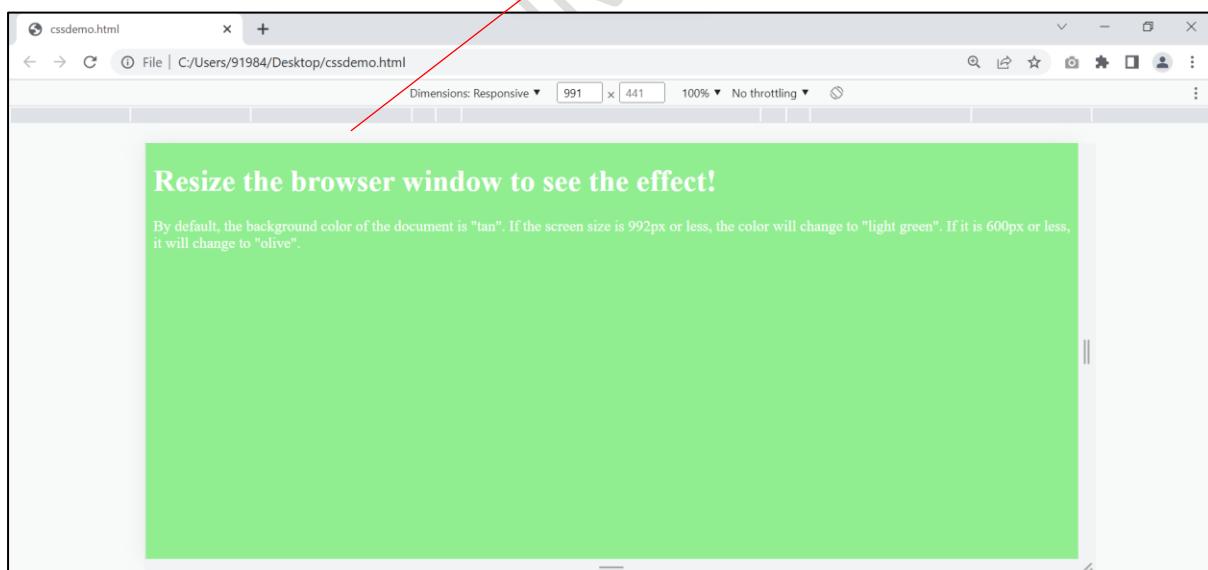
```
<!DOCTYPE html>
<html>
<head>
<style>
body {
    background-color: tan;
    color: black;
}
/* On screens that are 992px wide or less, the background color is light green */
@media screen and (max-width: 992px) {
body {
    background-color: lightgreen;
    color: white;
}
}
/* On screens that are 600px wide or less, the background color is olive */
@media screen and (max-width: 600px) {
body {
background-color: olive;
color: white;
}
}
</style>
</head>
<body>
<h1>Resize the browser window to see the effect! </h1>
<p>By default, the background color of the document is "tan". If the screen size is 992px or less, the color will change to "lightgreen". If it is 600px or less, it will change to "olive".</p>
</body>
</html>
```

Output:

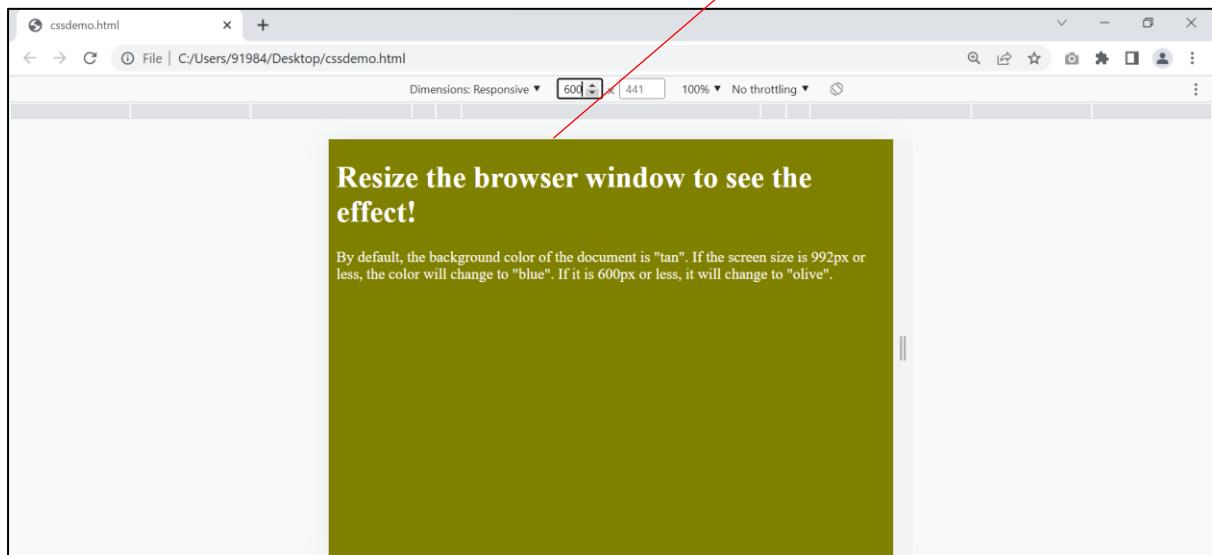
Screen size is more than 992px "tan" color applied.



Screen size is less than 992px and more than 600px then light green color applied.



Screen size from 600px Olive color can be applied.



BOOTSTRAP

1.BOOTSTRAP Introduction

Bootstrap 5 is the newest version of **Bootstrap**, which is the most popular HTML, CSS, and JavaScript framework for creating responsive, mobile-first websites.

Bootstrap 5 is completely free to download and use!

1.1 What is Bootstrap?

- Bootstrap is a free front-end framework for faster and easier web development
- Bootstrap includes HTML and CSS based design templates for typography, forms, buttons, tables, navigation, modals, image carousels and many other, as well as optional JavaScript plugins
- Bootstrap also gives you the ability to easily create responsive designs

1.2 Bootstrap History

Bootstrap was **created at Twitter in mid-2010 by @mdo and @fat**. Prior to being an open-sourced framework, Bootstrap was known as Twitter Blueprint. A few months into development, Twitter held its first Hack Week and the project exploded as developers of all skill levels jumped in without any external guidance.

1.3 Why Use Bootstrap ?

Advantages of Bootstrap:

- **Easy to use:** Anybody with just basic knowledge of HTML and CSS can start using Bootstrap
- **Responsive features:** Bootstrap's responsive CSS adjusts to phones, tablets, and desktops
- **Mobile-first approach:** In Bootstrap, mobile-first styles are part of the core framework
- **Browser compatibility:** Bootstrap 5 is compatible with all modern browsers (Chrome, Firefox, Edge, Safari, and Opera). **Note** that if you need support for IE11 and down, you must use either BS4 or BS3.

1.4 What Does Bootstrap Include ?

These are the HTML elements for which styles are provided: Typography, Code, Tables, Forms, Buttons, and many more. Whether you need to add drop-down menus, pagination, tooltips, or alert boxes, Bootstrap has got you covered.

The pre-styled components are:

- Dropdowns
- Button Groups
- Navigation Bar
- Breadcrumbs
- Labels & Badges
- Alerts
- Progress Bar
- Cards
- Carousel
- Modal
- And many others.

1.5 Downloading Bootstrap

Download Bootstrap to get the compiled CSS and JavaScript, source code, or include it with your favorite package managers like npm, RubyGems, and more.

Compiled CSS and JS

Download ready-to-use compiled code for **Bootstrap v5.2.0** to easily drop into your project, which includes:

- Compiled and minified CSS bundles (see [CSS files comparison](#))
- Compiled and minified JavaScript plugins (see [JS files comparison](#))

This doesn't include documentation, source files, or any optional JavaScript dependencies like Popper.

Download

Source files

Compile Bootstrap with your own asset pipeline by downloading our source Sass, JavaScript, and documentation files. This option requires some additional tooling:

- **Sass compiler** for compiling Sass source files into CSS files
- **Autoprefixer** for CSS vendor prefixing

Should you require our full set of **build tools**, they are included for developing Bootstrap and its docs, but they're likely unsuitable for your own purposes.

Download source

Examples

If you want to download and examine our **examples**, you can grab the already built examples:

1.6 Bootstrap CDN

Skip the download with **jsDelivr** to deliver cached version of Bootstrap's compiled CSS and JS to your project.

```
<link
  href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.0/dist/css/bootstrap.min.css" rel="stylesheet">

<script
  src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.0/dist/js/bootstrap.bundle.min.js"></script>
```

If you're using our compiled JavaScript and prefer to include Popper separately, add Popper before our JS, via a CDN preferably.

```
<script
  src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.11.5/dist/umd/popper.min.js"></script>

<script
  src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.0/dist/js/bootstrap.min.js"
></script>
```

2. Typography

Bootstrap 5 uses a default **font-size** of 1rem (16px by default), and its **line-height** is 1.5. In addition, all **<p>** elements have **margin-top: 0** and **margin-bottom: 1rem** (16px by default).

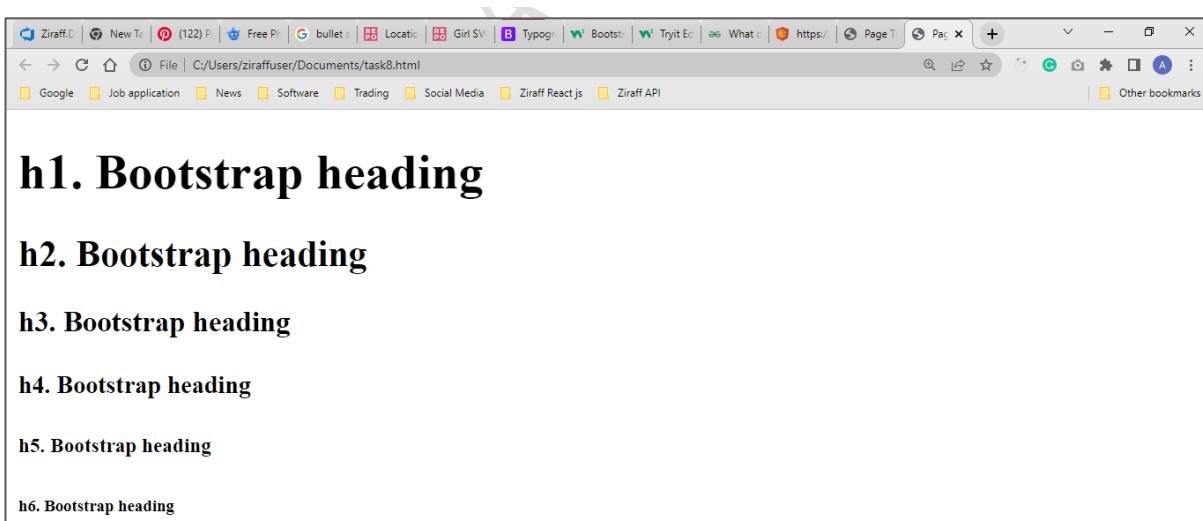
<h1> - <h6>

Bootstrap 5 styles HTML headings (**<h1> to <h6>**) with a bolder font-weight and a responsive font-size.

Example:

```
<h1>h1. Bootstrap heading</h1>
<h2>h2. Bootstrap heading</h2>
<h3>h3. Bootstrap heading</h3>
<h4>h4. Bootstrap heading</h4>
<h5>h5. Bootstrap heading</h5>
<h6>h6. Bootstrap heading</h6>
```

Output:



3. Colors:

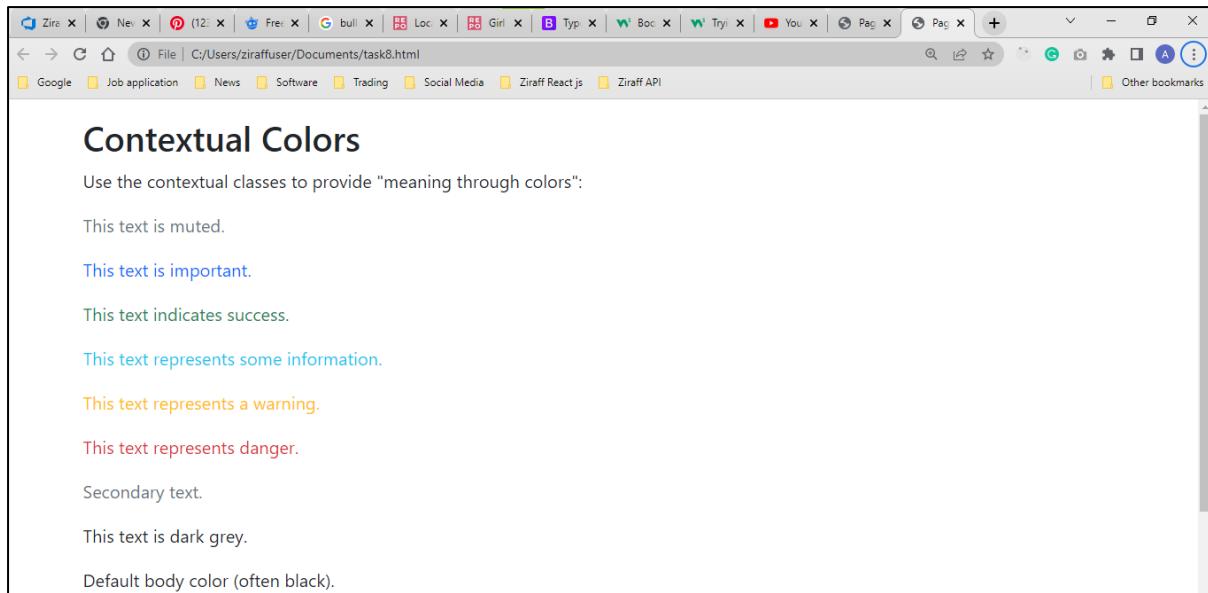
Bootstrap 5 has some contextual classes that can be used to provide "meaning through colors".

The classes for text colors are: **.text-muted**, **.text-primary**, **.text-success**, **.text-info**, **.text-warning**, **.text-danger**, **.text-secondary**, **.text-white**, **.text-dark**, **.text-body** (default body color/often black) and **.text-light**:

Example:

```
<div class="container mt-3">
  <h2>Contextual Colors</h2>
  <p>Use the contextual classes to provide "meaning through colors":</p>
  <p class="text-muted">This text is muted. </p>
  <p class="text-primary">This text is important. </p>
  <p class="text-success">This text indicates success. </p>
  <p class="text-info">This text represents some information. </p>
  <p class="text-warning">This text represents a warning. </p>
  <p class="text-danger">This text represents danger. </p>
  <p class="text-secondary">Secondary text. </p>
  <p class="text-dark">This text is dark grey. </p>
  <p class="text-body">Default body color (often black). </p>
  <p class="text-light">This text is light grey (on white background). </p>
  <p class="text-white">This text is white (on white background). </p>
</div>
```

Output:



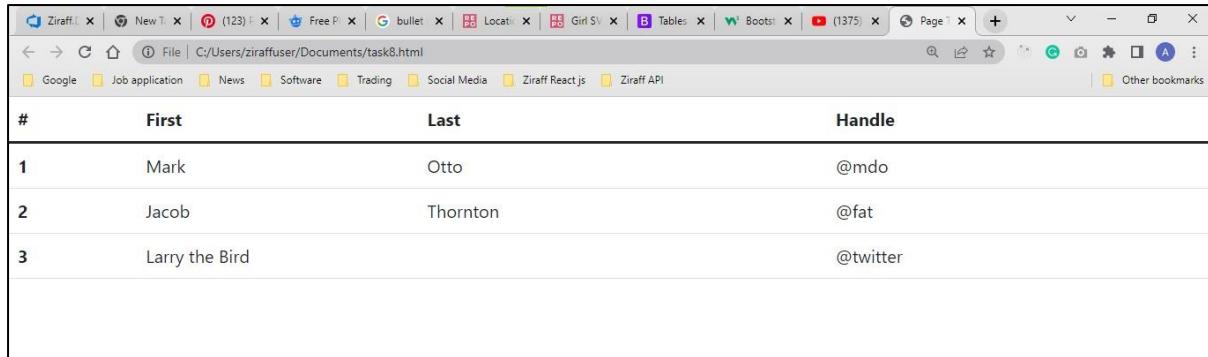
4. Tables:

Due to the widespread use of `<table>` elements across third-party widgets like calendars and date pickers, Bootstrap's tables are **opt-in**. Add the base **class .table** to any `<table>`, then extend with our optional modifier classes or custom styles. All table styles are not inherited in Bootstrap, meaning any nested tables can be styled independent from the parent. Using the most basic table markup, here's how .table-based tables look in Bootstrap.

Example:

```
<table class="table">
<thead>
<tr>
<th scope="col">#</th>
<th scope="col">First</th>
<th scope="col">Last</th>
<th scope="col">Handle</th>
</tr>
</thead>
<tbody>
<tr>
<th scope="row">1</th>
<td>Mark</td>
<td>Otto</td>
<td>@mdo</td>
</tr>
<tr>
<th scope="row">2</th>
<td>Jacob</td>
<td>Thornton</td>
<td>@fat</td>
</tr>
<tr>
<th scope="row">3</th>
<td colspan="2">Larry the Bird</td>
<td>@twitter</td>
</tr>
</tbody>
</table>
```

Output:



#	First	Last	Handle
1	Mark	Otto	@mdo
2	Jacob	Thornton	@fat
3	Larry	the Bird	@twitter

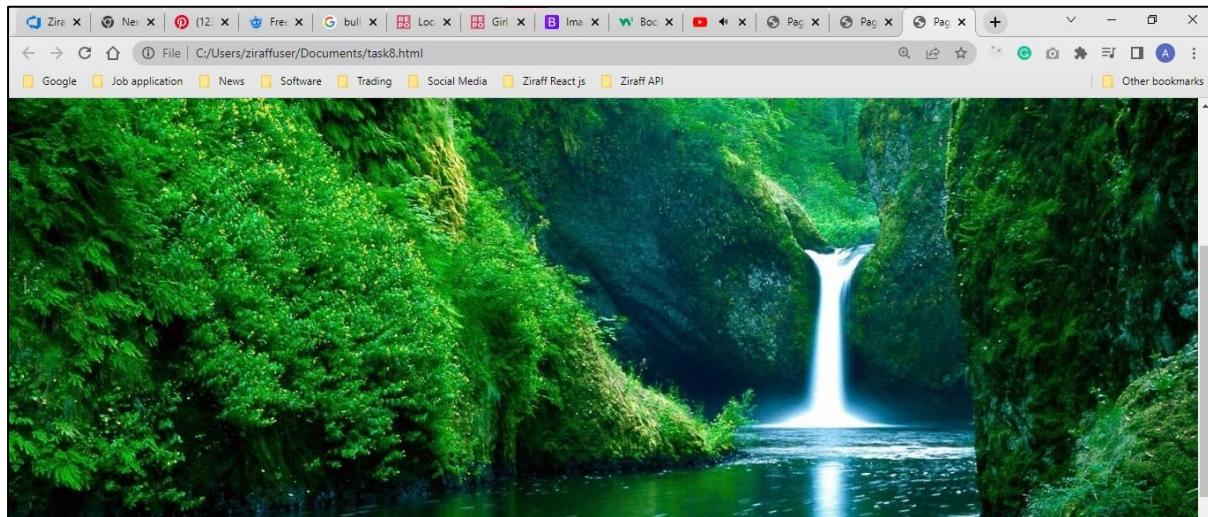
5. Images:

Images in Bootstrap are made responsive with **.img-fluid**. This applies **max-width: 100%**; and **height: auto;** to the image so that it scales with the parent width.

Example:

```
>
```

Output:



6. Alerts:

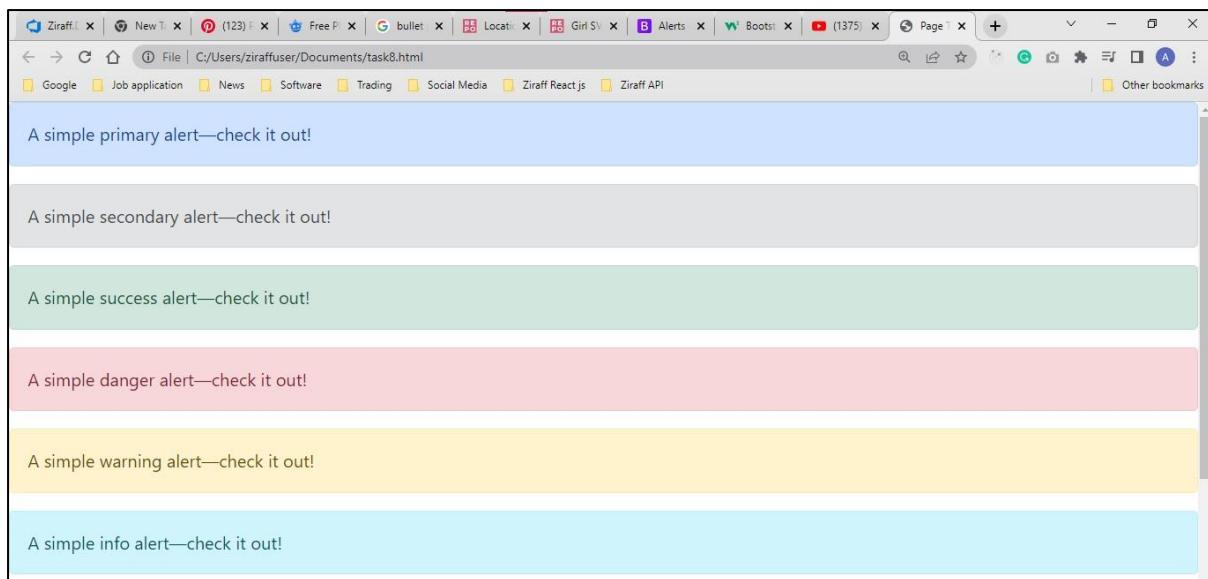
Provide contextual feedback messages for typical user actions with the handful of available and flexible alert messages.

Alerts are available for any length of text, as well as an optional close button. For proper styling, use one of the eight **required** contextual classes (e.g.,. alert-success). For inline dismissal, use the **alerts JavaScript plugin**.

Example:

```
<div class="alert alert-primary" role="alert">  
A simple primary alert—check it out!  
</div>  
<div class="alert alert-secondary" role="alert">  
A simple secondary alert—check it out!  
</div>  
<div class="alert alert-success" role="alert">  
A simple success alert—check it out!  
</div>  
<div class="alert alert-danger" role="alert">  
A simple danger alert—check it out!  
</div>  
<div class="alert alert-warning" role="alert">  
A simple warning alert—check it out!  
</div>  
<div class="alert alert-info" role="alert">  
A simple info alert—check it out!  
</div>  
<div class="alert alert-light" role="alert">  
A simple light alert—check it out!  
</div>  
<div class="alert alert-dark" role="alert">  
A simple dark alert—check it out!  
</div>
```

Output:



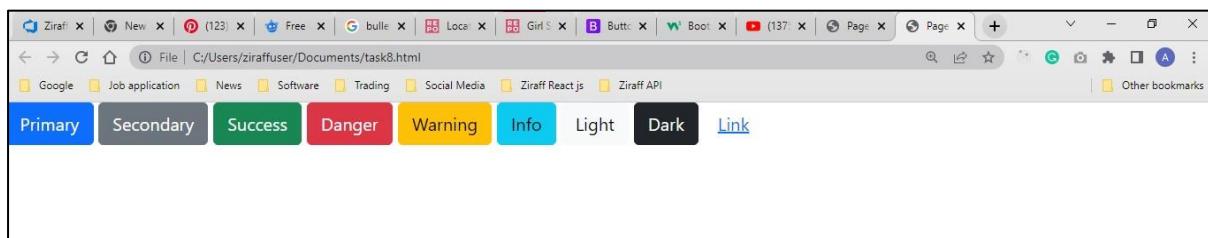
7. Buttons:

Use Bootstrap's custom button styles for actions in forms, dialogs, and more with support for multiple sizes, states, and more. Bootstrap includes several predefined button styles, each serving its own semantic purpose, with a few extras thrown in for more control.

Example:

```
<button type="button" class="btn btn-primary">Primary</button>
<button type="button" class="btn btn-secondary">Secondary</button>
<button type="button" class="btn btn-success">Success</button>
<button type="button" class="btn btn-danger">Danger</button>
<button type="button" class="btn btn-warning">Warning</button>
<button type="button" class="btn btn-info">Info</button>
<button type="button" class="btn btn-light">Light</button>
<button type="button" class="btn btn-dark">Dark</button>
<button type="button" class="btn btn-link">Link</button>
```

Output:

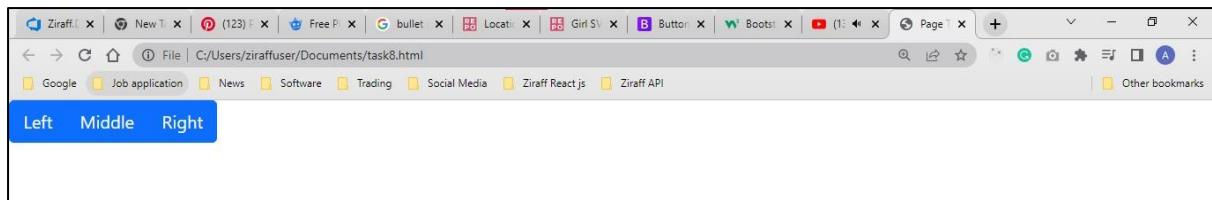


8. Button Groups:

Group a series of buttons together on a single line or stack them in a vertical column.
Wrap a series of buttons with **.btn** in **.btn-group**

Example:

```
<div class="btn-group" role="group" aria-label="Basic example">
  <button type="button" class="btn btn-primary">Left</button>
  <button type="button" class="btn btn-primary">Middle</button>
  <button type="button" class="btn btn-primary">Right</button>
</div>
```

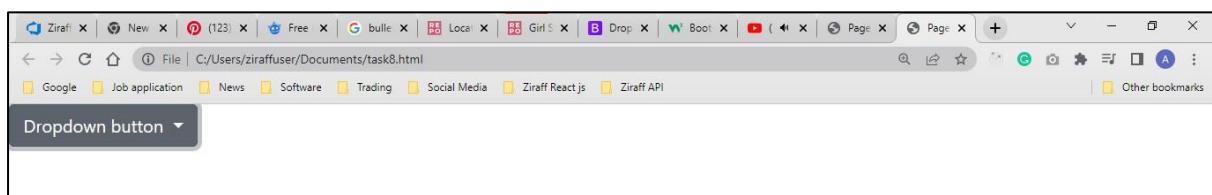
Output:

9. Button DropDown:

Toggle contextual overlays for displaying lists of links and more with the Bootstrap dropdown plugin.

Example:

```
<div class="dropdown">
  <button class="btn btn-secondary dropdown-toggle" type="button" data-bs-
  toggle="dropdown" aria-expanded="false">
    Dropdown button
  </button>
  <ul class="dropdown-menu">
    <li><a class="dropdown-item" href="#">Action</a></li>
    <li><a class="dropdown-item" href="#">Another action</a></li>
    <li><a class="dropdown-item" href="#">Something else here</a></li>
  </ul>
</div>
```

Output:

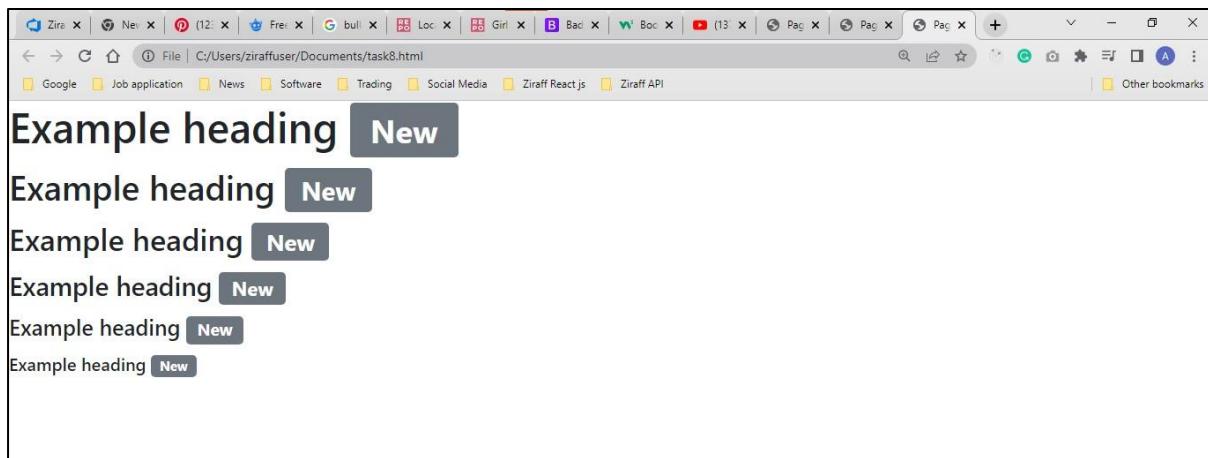
10. Badges:

Documentation and examples for badges, our small count and labeling component. Badges scale to match the size of the immediate parent element by using relative font sizing and em units. As of v5, badges no longer have focus or hover styles for links.

Example:

```
<h1>Example heading <span class="badge bg-secondary">New</span></h1>
<h2>Example heading <span class="badge bg-secondary">New</span></h2>
<h3>Example heading <span class="badge bg-secondary">New</span></h3>
<h4>Example heading <span class="badge bg-secondary">New</span></h4>
<h5>Example heading <span class="badge bg-secondary">New</span></h5>
<h6>Example heading <span class="badge bg-secondary">New</span></h6>
```

Output:



11. Progress Bars:

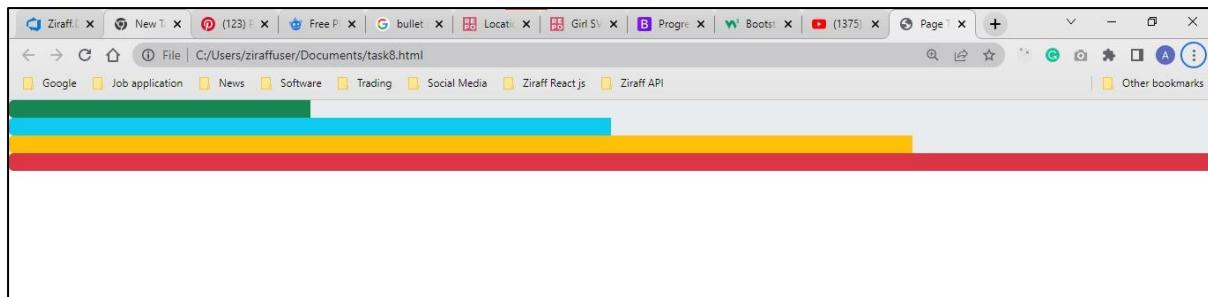
Documentation and examples for using Bootstrap custom progress bars featuring support for stacked bars, animated backgrounds, and text labels.

- We use the .progress as a wrapper to indicate the max value of the progress bar.
- We use the inner .progress-bar to indicate the progress so far.
- The .progress-bar requires an inline style, utility class, or custom CSS to set their width.
- The .progress-bar also requires some role and aria attributes to make it accessible, including an accessible name (using aria-label, aria-labelledby, or similar).

Example:

```
<div class="progress">
  <div class="progress-bar" role="progressbar" aria-label="Basic example" aria-
  valuenow="0" aria-valuemin="0" aria-valuemax="100"></div>
</div>
<div class="progress">
  <div class="progress-bar" role="progressbar" aria-label="Basic example" style="width: 25%" aria-valuenow="25" aria-valuemin="0" aria-
  valuemax="100"></div>
</div>
<div class="progress">
  <div class="progress-bar" role="progressbar" aria-label="Basic example" style="width: 50%" aria-valuenow="50" aria-valuemin="0" aria-
  valuemax="100"></div>
</div>
<div class="progress">
  <div class="progress-bar" role="progressbar" aria-label="Basic example" style="width: 75%" aria-valuenow="75" aria-valuemin="0" aria-
  valuemax="100"></div>
</div>
<div class="progress">
  <div class="progress-bar" role="progressbar" aria-label="Basic example" style="width: 100%" aria-valuenow="100" aria-valuemin="0" aria-
  valuemax="100"></div>
</div>
```

Output:



12. Pagination

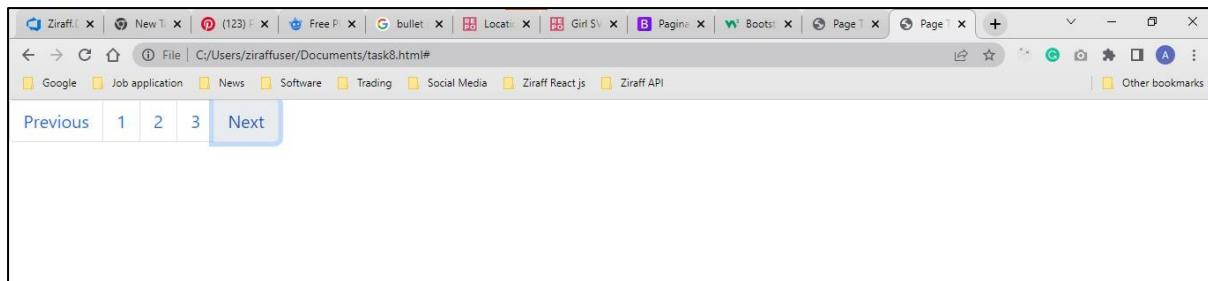
Documentation and examples for showing pagination to indicate a series of related content exists across multiple pages.

In addition, as pages likely have more than one such navigation section, it's advisable to provide a descriptive aria-label for the `<nav>` to reflect its purpose. For example, if the pagination component is used to navigate between a set of search results, an appropriate label could be `aria-label="Search results pages"`.

Example:

```
<nav aria-label="Page navigation example">
  <ul class="pagination">
    <li class="page-item"><a class="page-link" href="#">Previous</a></li>
    <li class="page-item"><a class="page-link" href="#">1</a></li>
    <li class="page-item"><a class="page-link" href="#">2</a></li>
    <li class="page-item"><a class="page-link" href="#">3</a></li>
    <li class="page-item"><a class="page-link" href="#">Next</a></li>
  </ul>
</nav>
```

Output:



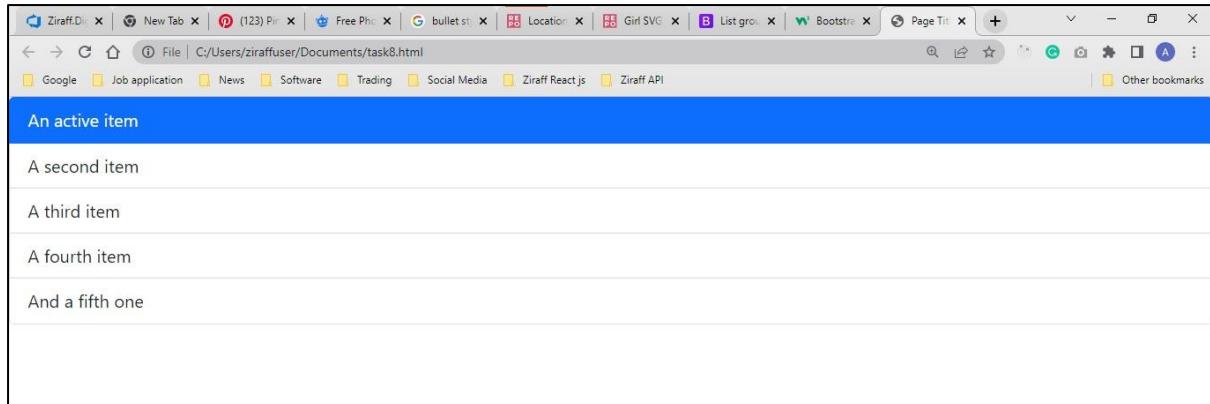
13. List Groups

List groups are a flexible and powerful component for displaying a series of content. Modify and extend them to support just about any content within.

Example:

```
<ul class="list-group">
  <li class="list-group-item active" aria-current="true">An active item</li>
  <li class="list-group-item">A second item</li>
  <li class="list-group-item">A third item</li>
  <li class="list-group-item">A fourth item</li>
  <li class="list-group-item">And a fifth one</li>
</ul>
```

Output:



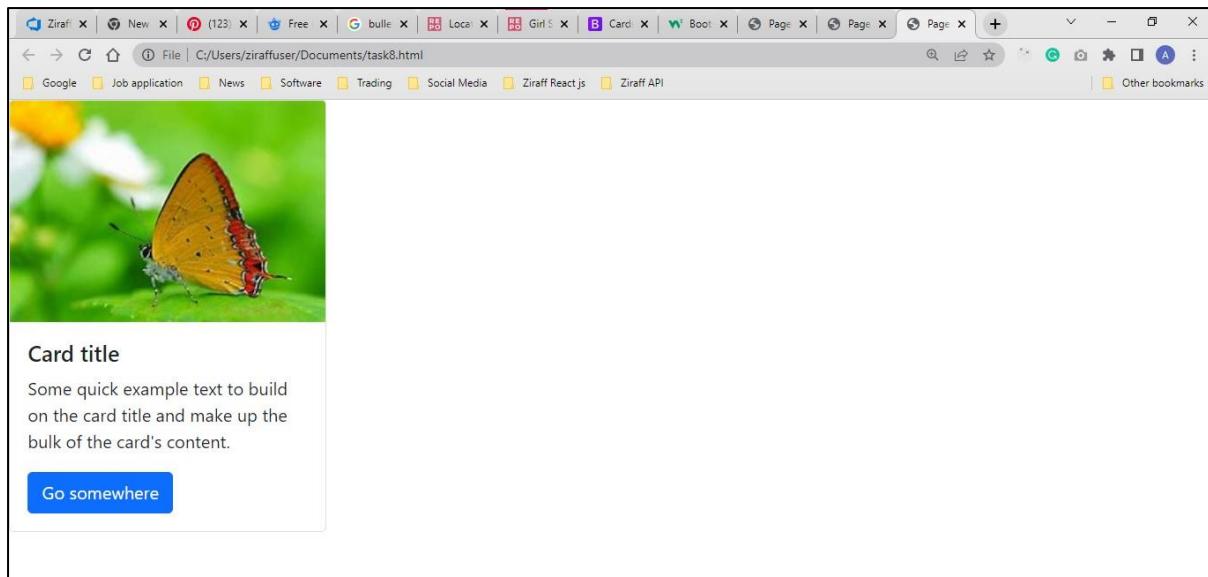
14.Cards

A **card** is a flexible and extensible content container. It includes options for headers and footers, a wide variety of content, contextual background colors, and powerful display options. If you're familiar with Bootstrap 3, cards replace our old panels, wells, and thumbnails. Similar functionality to those components is available as modifier classes for cards.

Example:

```
<div class="card" style="width: 18rem;">
  
  <div class="card-body">
    <h5 class="card-title">Card title</h5>
    <p class="card-text">Some quick example text to build on the card title and
make up the bulk of the card's content.</p>
    <a href="#" class="btn btn-primary">Go somewhere</a>
  </div>
</div>
```

Output:



15.Collapse

Toggle the visibility of content across your project with a few classes and our JavaScript plugins. The collapse JavaScript plugin is used to show and hide content. Buttons or anchors are used as triggers that are mapped to specific elements you toggle. Collapsing an element will animate the **height** from its current value to 0. Given how CSS handles animations, you cannot use padding on a **.collapse** element. Instead, use the class as an independent wrapping element.

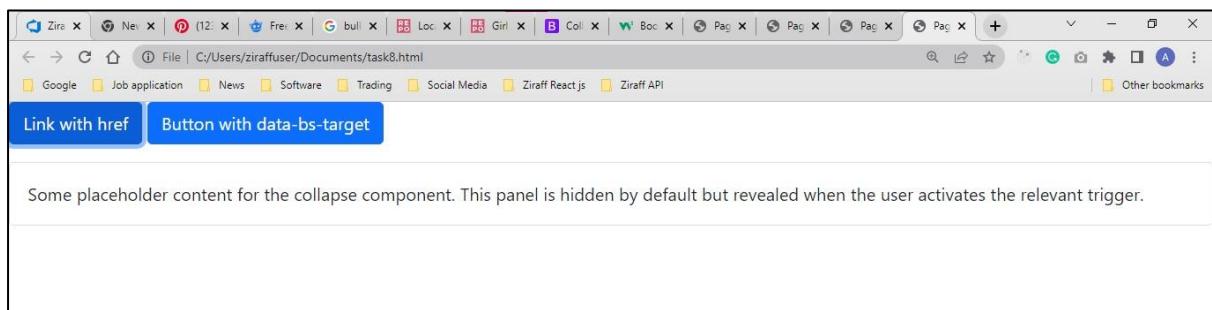
Example:

```

<p>
  <a class="btn btn-primary" data-bs-toggle="collapse" href="#collapseExample"
  role="button" aria-expanded="false" aria-controls="collapseExample">
    Link with href
  </a>
  <button class="btn btn-primary" type="button" data-bs-toggle="collapse" data-
  bs-target="#collapseExample" aria-expanded="false" aria-
  controls="collapseExample">
    Button with data-bs-target
  </button>
</p>
<div class="collapse" id="collapseExample">
  <div class="card card-body">
    Some placeholder content for the collapse component. This panel is hidden by
    default but revealed when the user activates the relevant trigger.
  </div>
</div>

```

Output:



16. Navbar

Documentation and examples for Bootstrap's powerful, responsive navigation header, the navbar. Includes support for branding, navigation, and more, including support for our collapse plugin.

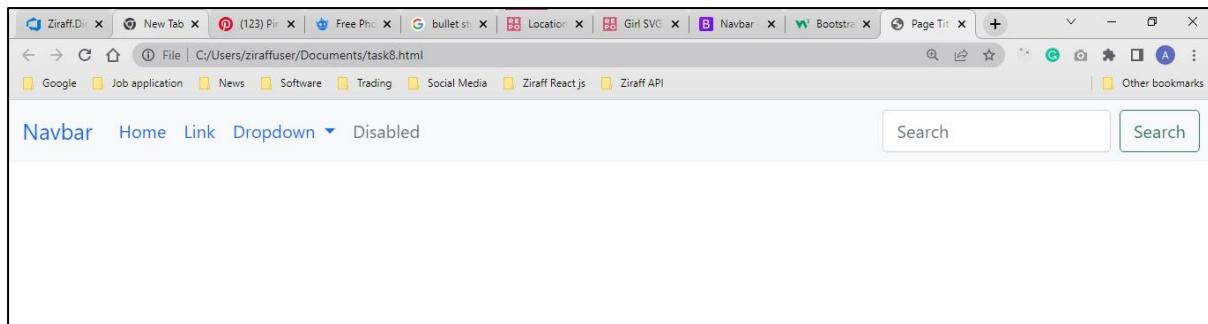
- Navbars require a **wrapping .navbar** with **.navbar-expand{-sm|-md|-lg|-xl|-xxl}** for responsive collapsing and **color scheme** classes.
- Navbars and their contents are fluid by default. Change the **container** to limit their horizontal width in different ways.

- Use our **spacing** and **flex** utility classes for controlling spacing and alignment within navbars.
- Navbars are responsive by default, but you can easily modify them to change that. Responsive behavior depends on our Collapse JavaScript plugin.
- Ensure accessibility by using a `<nav>` element or, if using a more generic element such as a `<div>`, add a **role="navigation"** to every navbar to explicitly identify it as a landmark region for users of assistive technologies.
- Indicate the current item by using **aria-current="page"** for the current page or **aria-current="true"** for the current item in a set.
- **New in v5.2.0:** Navbars can be themed with CSS variables that are scoped to the **.navbar** base class. **.navbar-light** has been deprecated and **.navbar-dark** has been rewritten to override CSS variables instead of adding additional styles.
- **.navbar-brand** for your company, product, or project name.
- **.navbar-nav** for a full-height and lightweight navigation (including support for dropdowns).
- **.navbar-toggler** for use with our collapse plugin and other **navigation toggling** behaviors.
- Flex and spacing utilities for any form controls and actions.
- **.navbar-text** for adding vertically centered strings of text.
- **.collapse .navbar-collapse** for grouping and hiding navbar contents by a parent breakpoint.
- Add an optional **.navbar-scroll** to set a max-height and **scroll expanded navbar content**.

Example:

```
<nav class="navbar navbar-expand-lg bg-light">
  <div class="container-fluid">
    <a class="navbar-brand" href="#">Navbar</a>
    <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarSupportedContent" aria-controls="navbarSupportedContent" aria-expanded="false" aria-label="Toggle navigation">
      <span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse" id="navbarSupportedContent">
      <ul class="navbar-nav me-auto mb-2 mb-lg-0">
        <li class="nav-item">
          <a class="nav-link active" aria-current="page" href="#">Home</a>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="#">Link</a>
        </li>
        <li class="nav-item dropdown">
          <a class="nav-link dropdown-toggle" href="#" role="button" data-bs-toggle="dropdown" aria-expanded="false">
            Dropdown
          </a>
          <ul class="dropdown-menu">
            <li><a class="dropdown-item" href="#">Action</a></li>
            <li><a class="dropdown-item" href="#">Another action</a></li>
            <li><hr class="dropdown-divider"></li>
            <li><a class="dropdown-item" href="#">Something else here</a></li>
          </ul>
        </li>
        <li class="nav-item">
          <a class="nav-link disabled">Disabled</a>
        </li>
      </ul>
      <form class="d-flex" role="search">
        <input class="form-control me-2" type="search" placeholder="Search" aria-label="Search">
        <button class="btn btn-outline-success" type="submit">Search</button>
      </form>
    </div>
  </div>
</nav>
```

Output:



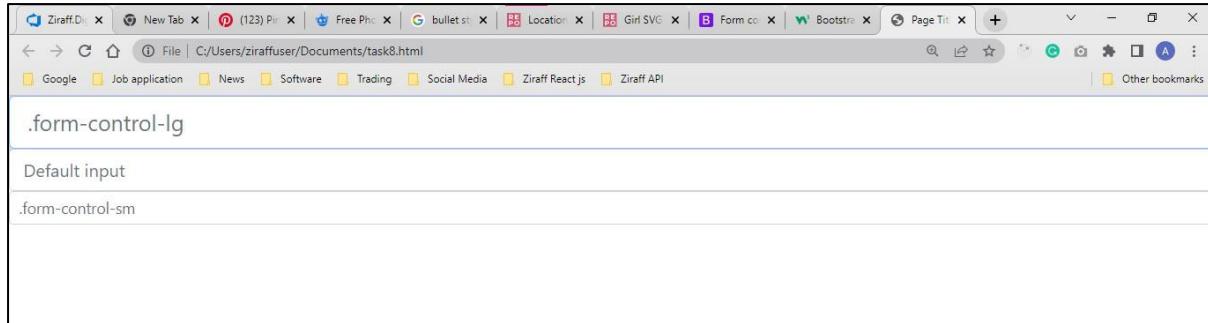
17.Forms

Give textual form controls like `<input>`s and `<textarea>`s an upgrade with custom styles, sizing, focus states, and more.

Example:

```
<input class="form-control form-control-lg" type="text" placeholder=". form-control-lg" aria-label=". form-control-lg example">
<input class="form-control" type="text" placeholder="Default input" aria-label="default input example">
<input class="form-control form-control-sm" type="text" placeholder=". form-control-sm" aria-label=".form-control-sm example">
```

Output:



18.Carousel

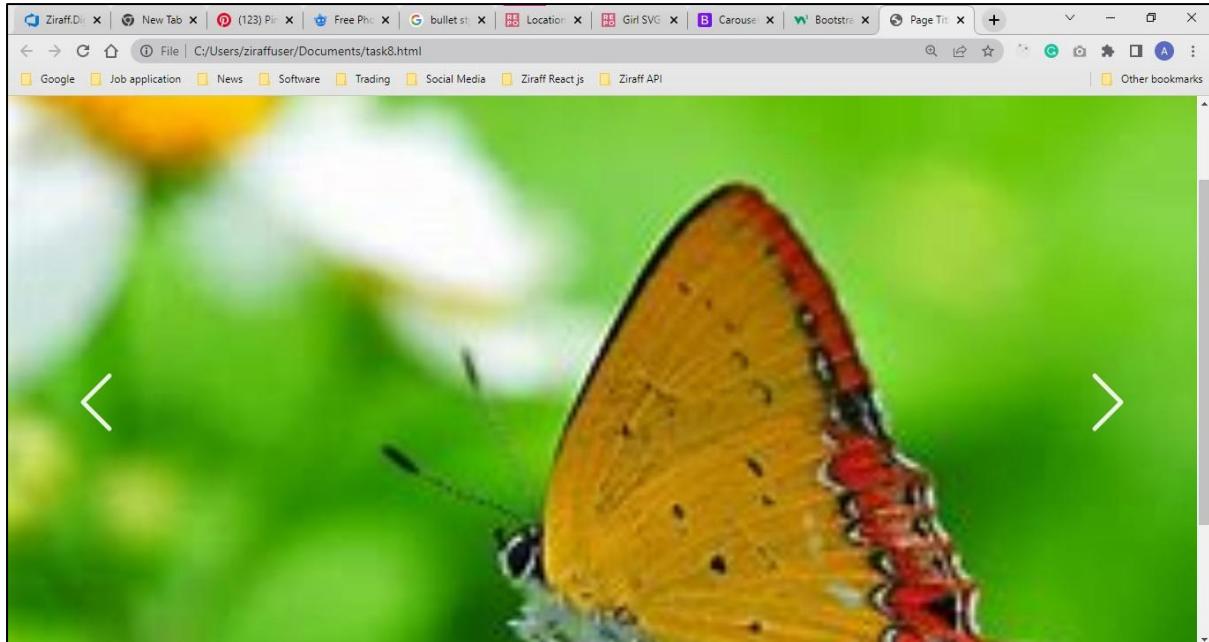
A slideshow component for cycling through elements—images or slides of text—like a carousel.

The. active class needs to be added to one of the slides otherwise the carousel will not be visible. Also be sure to set a unique id on the **.carousel** for optional controls, especially if you're using multiple carousels on a single page. Control and indicator elements must have a **data-bs-target** attribute (or href for links) that matches the id of the. **carousel** element.

Example:

```
<div id="carouselExampleControls" class="carousel slide" data-bs-ride="carousel">
  <div class="carousel-inner">
    <div class="carousel-item active">
      
    </div>
    <div class="carousel-item">
      
    </div>
    <div class="carousel-item">
      
    </div>
  </div>
  <button class="carousel-control-prev" type="button" data-bs-target="#carouselExampleControls" data-bs-slide="prev">
    <span class="carousel-control-prev-icon" aria-hidden="true"></span>
    <span class="visually-hidden">Previous</span>
  </button>
  <button class="carousel-control-next" type="button" data-bs-target="#carouselExampleControls" data-bs-slide="next">
    <span class="carousel-control-next-icon" aria-hidden="true"></span>
    <span class="visually-hidden">Next</span>
  </button>
</div>
```

Output:



19.Modal

Use Bootstrap's JavaScript modal plugin to add dialogs to your site for lightboxes, user notifications, or completely custom content.

Modal components

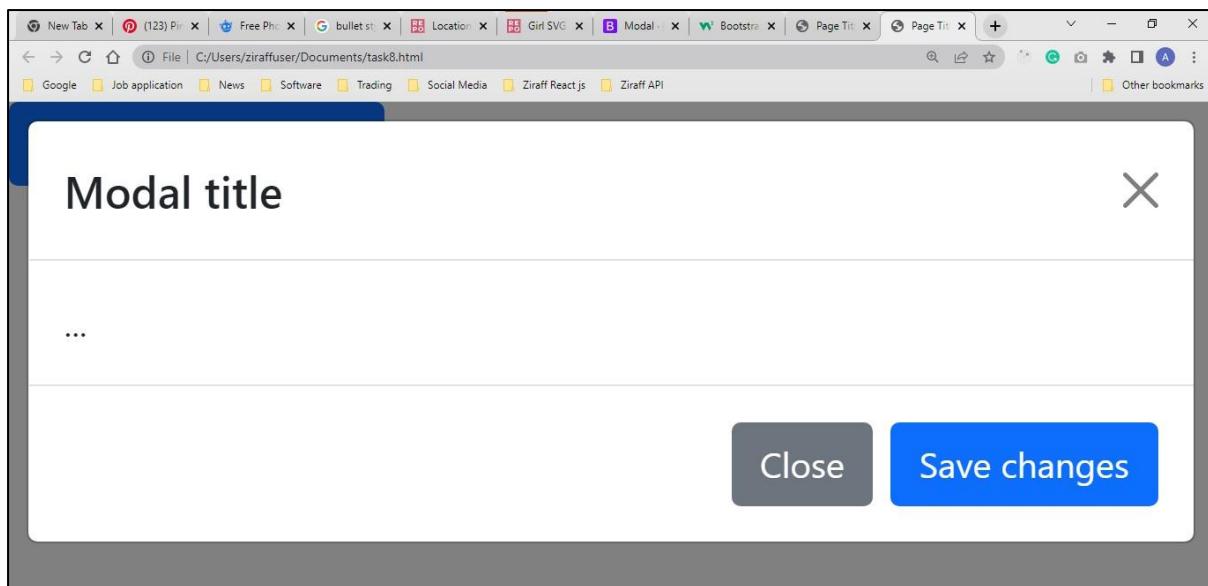
Below is a *static* modal example (meaning its position and display have been overridden). Included are the modal header, modal body (required for padding), and modal footer (optional). We ask that you include modal headers with dismiss actions whenever possible, or provide another explicit dismiss action.

Example:

```
<! -- Button trigger modal -->
<button type="button" class="btn btn-primary" data-bs-toggle="modal" data-bs-target="#exampleModal">
  Launch demo modal
</button>

<! -- Modal -->
<div class="modal fade" id="exampleModal" tabindex="-1" aria-labelledby="exampleModalLabel" aria-hidden="true">
  <div class="modal-dialog">
    <div class="modal-content">
      <div class="modal-header">
        <h5 class="modal-title" id="exampleModalLabel">Modal title</h5>
        <button type="button" class="btn-close" data-bs-dismiss="modal" aria-label="Close"></button>
      </div>
      <div class="modal-body">
        ...
      </div>
      <div class="modal-footer">
        <button type="button" class="btn btn-secondary" data-bs-dismiss="modal">Close</button>
        <button type="button" class="btn btn-primary">Save changes</button>
      </div>
    </div>
  </div>
</div>
```

Output:



20. ToolTip

- Tooltips rely on the third-party library **Popper** for positioning. You must include **popper.min.js** before bootstrap.js, or use one bootstrap.bundle.min.js which contains Popper.
- Tooltips are opt-in for performance reasons, so **you must initialize them yourself**.
- Tooltips with zero-length titles are never displayed.
- Specify **container: 'body'** to avoid rendering problems in more complex components (like our input groups, button groups, etc).
- Triggering tooltips on hidden elements will not work.
- Tooltips for **.disabled** or disabled elements must be triggered on a wrapper element.
- When triggered from hyperlinks that span multiple lines, tooltips will be centered. Use **white-space: nowrap;** on your <a>s to avoid this behavior.
- Tooltips must be hidden before their corresponding elements have been removed from the DOM.
- Tooltips can be triggered thanks to an element inside a shadow DOM.

Got all that? Great, let's see how they work with some examples.

Example:

```
<div class="container mt-3">
<h3>Tooltip Example</h3>
<button type="button" class="btn btn-primary" data-bs-toggle="tooltip"
title="Hooray!"> Hover over me! </button>
</div>
<script>
// Initialize tooltips
var tooltipTriggerList = [].slice. call(document.querySelectorAll('[data-bs-
toggle="tooltip"]'))
var tooltipList = tooltipTriggerList.map(function (tooltipTriggerEl) {
return new bootstrap.Tooltip(tooltipTriggerEl)
})
</script>
```

Output:

Whenever we hover on button
the tooltip will be visible



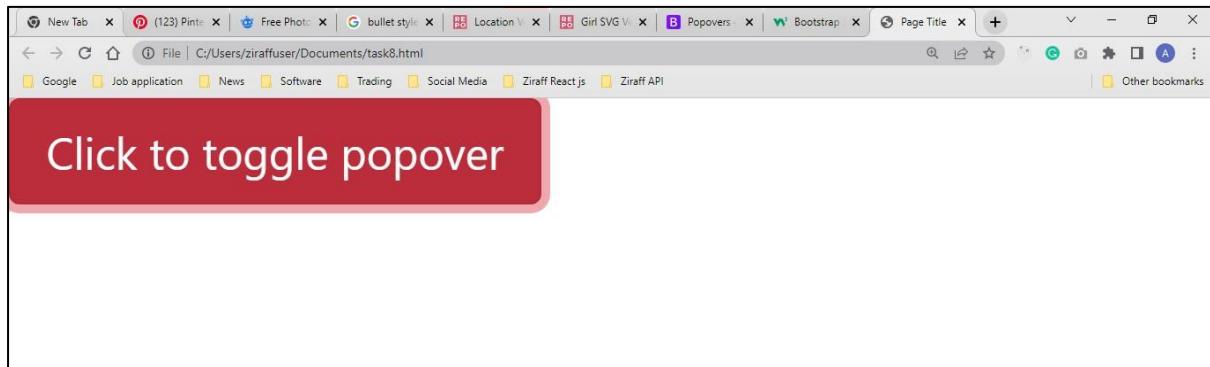
21. Popover

- Popovers rely on the third party library **Popper** for positioning. You must include **popper.min.js** before bootstrap.js, or use one bootstrap.bundle.min.js which contains Popper.
- Popovers require the **popover plugin** as a dependency.
- Popovers are opt-in for performance reasons, so **you must initialize them yourself**.
- Zero-length title and content values will never show a popover.
- Specify container: 'body' to avoid rendering problems in more complex components (like our input groups, button groups, etc).
- Triggering popovers on hidden elements will not work.
- Popovers for **.disabled** or disabled elements must be triggered on a wrapper element.
- When triggered from anchors that wrap across multiple lines, popovers will be centered between the anchors' overall width. Use **.text nowrap** on your <a>s to avoid this behavior.
- Popovers must be hidden before their corresponding elements have been removed from the DOM.
- Popovers can be triggered thanks to an element inside a shadow DOM.

Example:

```
<button type="button" class="btn btn-lg btn-danger" data-bs-toggle="popover" data-bs-title="Popover title" data-bs-content="And here's some amazing content. It's very engaging. Right?">Click to toggle popover</button>
```

Output:



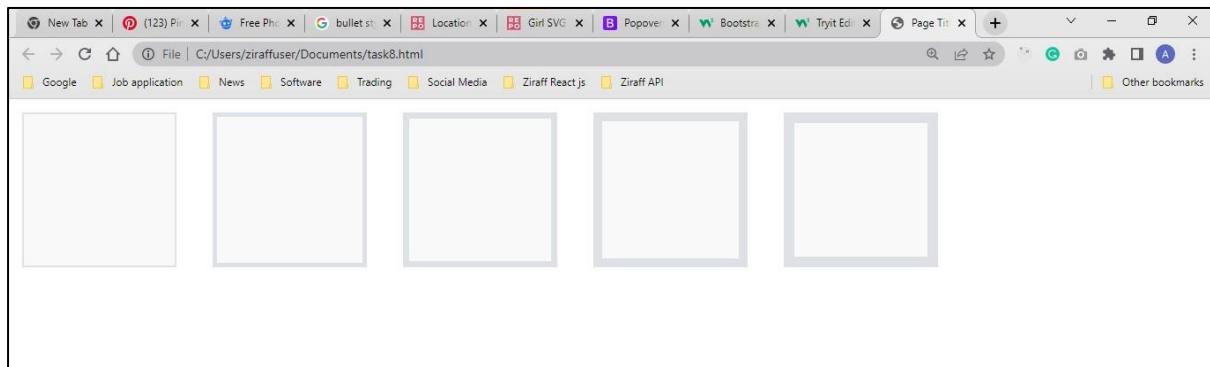
22.Utilities

Bootstrap 5 has a lot of utility/helper classes to quickly style elements without using any CSS code.

Example:

```
<span class="border border-1"></span>
<span class="border border-2"></span>
<span class="border border-3"></span>
<span class="border border-4"></span>
<span class="border border-5"></span>
```

Output:



JAVASCRIPT

What is JavaScript?

JavaScript is a cross-platform, object-oriented scripting language used to make webpages interactive (e.g., having complex animations, clickable buttons, popup menus, etc.).

History of JavaScript

JavaScript was **invented by Brendan Eich in 1995**. It was developed for Netscape 2, and became the ECMA-262 standard in 1997. After Netscape handed JavaScript over to ECMA, the Mozilla foundation continued to develop JavaScript for the Firefox browser.

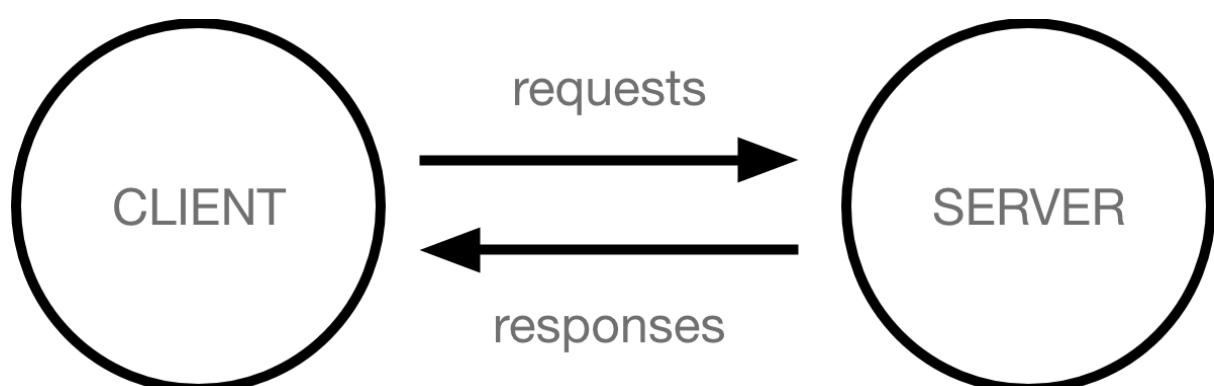
Installing basic software's

Required tools:

1. Text Editor
2. Modern Web Browsers

How the web works

Computers connected to internet are called Clients & Server, The Below Diagram show how they interact



How to write JavaScript

We can write JavaScript in Two ways

1. Internal
2. External

Internal JavaScript:

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <title>Document</title>
8  </head>
9  <body>
10
11      <script>
12          console.log("Welcome to Veda Institute of Technologies");
13      </script>
14  </body>
15 </html>
```

We use **script** tag to write internal JavaScript.

We write the JavaScript at bottom of the body only.

External JavaScript

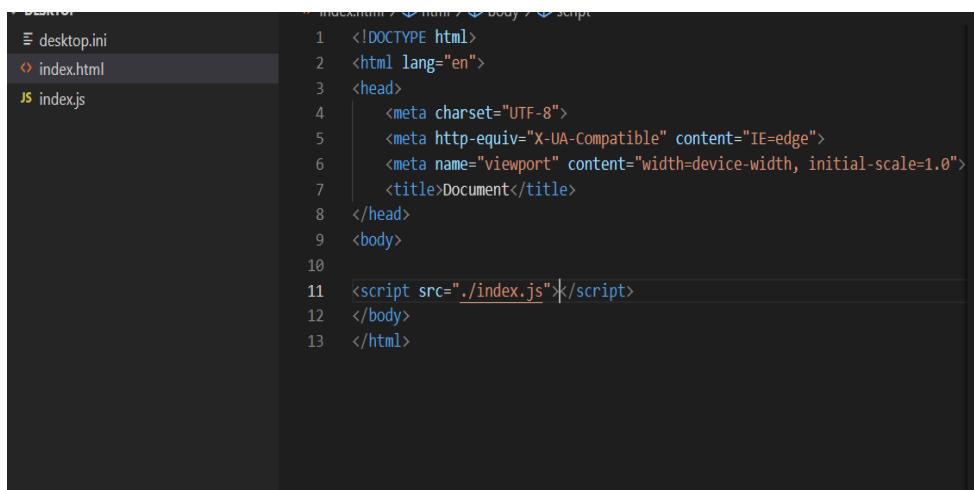
index.js

```
1  console.log("Welcome to Veda Institute of technologies")
```

1. Create a new file with **.js** extension
2. Then write JavaScript code in created file
3. Reference the JS file in Web page

Referencing JS

```
<script src="[path of the js file]"></script>
```



```
index.html
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta http-equiv="X-UA-Compatible" content="IE=edge">
6   <meta name="viewport" content="width=device-width, initial-scale=1.0">
7   <title>Document</title>
8 </head>
9 <body>
10
11 <script src="./index.js"></script>
12 </body>
13 </html>
```

Note: We always reference JS file at the bot

tom of the body only

Variables

Variables are containers that store values. You start by declaring a variable with the **let** keyword, followed by the name you give to the variable:

```
let my Variable;
```

Semicolon Indicates the end of Statement

After declaring a variable, you can give it a value:

```
my Variable = 'Bob';
```

Also, you can do both these operations on the same line:

```
let my Variable = 'Bob';
```

You retrieve the value by calling the variable name

```
console.log(my Variable);
```

After assigning a value to a variable, you can change it later in the code

```
my Variable = "Hello";
```

If you want a general rule: always declare variables with const.

If you think the value of the variable can change, use let.

Note that Variable may store the values that have different data types

why do we need variables?

Variables are necessary to do anything interesting in programming. If values couldn't change, then you couldn't do anything dynamic

Variable	Explanation	Example
<u>String</u>	This is a sequence of text known as a string. To signify that the value is a string, let my Variable = 'Bob'; enclose it in single quote marks.	
<u>Number</u>	This is a number. Numbers don't have quotes around them.	let my Variable = 10;
<u>Boolean</u>	This is a True/False value. The words true and false are special keywords that don't need quote marks.	let my Variable = true;
<u>Array</u>	This is a structure that allows you to store multiple values in a single reference.	let my Variable = [1,'Bob','Steve',10]; Refer to each member of the array like this: my Variable [0], my Variable [1], etc.

Variable	Explanation	Example
Object	This can be anything. Everything in JavaScript is an object and can be stored in document.querySelector('h1'); a variable. Keep this in mind as you learn. All of the above examples too.	let my Variable =

Operators

An Operator is mathematical symbol that produces a result based on two values or variables

Operator	Explanation	Symbol (s)	Example
Addition	Add two numbers together or combine two strings.	+	<code>6 + 9;</code> <code>'Hello ' + 'world!';</code>
Subtraction, Multiplication, Division	These do what you'd expect them to do in basic math.	-, *, /	<code>9 - 3;</code> <code>8 * 2; // multiply in JS is an asterisk</code> <code>9 / 3;</code>
Assignment	As you've seen already: this assigns a value to a variable.	=	<code>let my Variable = 'Bob';</code>
Strict equality	This performs a test to see if two values are equal. It returns a true/false (Boolean) result.	====	<code>let my Variable = 3;</code> <code>my Variable === 4;</code>

Operator	Explanation	Symbol (s)	Example
Not, Does-not-equal	This returns the logically opposite value of what it precedes. It turns a true into a false, etc. When it is used alongside the Equality operator, the negation operator tests whether two values are <i>not</i> equal.	!, !=	For "Not", the basic expression is true, but the comparison returns false because we negate it: let my Variable = 3; !(my Variable === 3); "Does-not-equal" gives basically the same result with different syntax. Here we are testing "is my Variable NOT equal to 3". This returns false because my Variable IS equal to 3: let my Variable = 3; my Variable! == 3;



Conditions

Conditionals are code structures used to test if an expression returns true or not. A very common form of conditionals is the if...else statement. For example:

```
JS index.js > ...
1 let car = "Crysta";
2
3 if(car === "Crysta"){
4     console.log("I love Crysta")
5 }
6 else{
7     console.log("I Love Crysta")
8 }
```

The Expression inside the if () is the test This uses the strict equality operator to compare the variable car with the string Crista

1. If the both values same the expression returns true. then first block will be executed
2. If the expression returns false (if both values are not same) second block will be executed (false)

Syntax:

```
if (condition1)
    statement1
else if (condition2)
    statement2
else if (condition3)
    statement3
else
    statementN
```

Functions

Functions are one of the fundamental building blocks in JavaScript. A function in JavaScript is a set of statements to perform specific task.

To use a function, you must define it somewhere in the scope from which you wish to call it.

Functions allow you to store a piece of code that does a single task inside a defined block, and then call that code whenever you need it using a single short command — rather than having to type out the same code multiple times

Defining Functions

A **function definition** (also called a **function declaration**, or **function statement**) consists of the **function** keyword, followed by:

- The name of the function.
- A list of parameters to the function, enclosed in parentheses and separated by commas.

- The JavaScript statements that define the function, enclosed in curly brackets, { /* ... */ }.

Syntax:

```
function name (parameter1, parameter2, parameter3) {  
    // code to be executed  
}
```

Example:

```
function sum (value1, value2) {  
    return value1+value2;  
}
```

The above function takes two parameters value1 and value2 & it performs the sum between two values and returns the result to calling area

```
let result = sum (2,4);  
console.log(result) // here result gets the value of 6
```

Calling Functions

Defining function does not execute it. Defining It names the function and specifies that what to do when the function called

Calling the function will perform the action with indicated parameters

Ex: sum (5,5);

The above function calls the function with Two arguments 5 & 5. the function will execute the statements and returns the result value

Functions must be *in scope* when they are called, but the function declaration can be hoisted (function hoisting) (appear below the call in the code),

```
console.log(multiply (2,2));  
function multiply (value1, value2) {  
    return value1 * value2;  
}
```

Function Scope: Variables defined inside a function cannot be accessed from anywhere outside the function, because the variable is defined only in the scope of the function.

However, a function can access all variables and functions defined inside the scope in which it is defined.

In other words, a function defined in the global scope can access all variables defined in the global scope. A function defined inside another function can also access all variables defined in its parent function, and any other variables to which the parent function has access.

```
// The following variables are defined in the global scope
const num1 = 20;
const num2 = 3;
const name = 'Chamakh';

// This function is defined in the global scope
function multiply() {
    return num1 * num2;
}

multiply(); // Returns 60

// A nested function example
function getScore() {
    const num1 = 2;
    const num2 = 3;

    function add() {
        return `${name} scored ${num1 + num2}`;
    }

    return add();
}

getScore(); // Returns "Chamakh scored 5"
```

Types of Functions

1. Named Functions
2. Anonymous Functions
3. Auto / Immediate Invoked Functions

Named Functions:

Named functions in JavaScript is just a fancy way to refer to a function that uses the `function` keyword and then a name you can use as a reference to that function

Ex `functions myFun () {`

`console.log ("Named Function");`

`}`

EX: <script>

```
function test () {
    console.log ('This is a named function! `);
};

</script>
```

Anonymous Function:

An anonymous function is a function without a name.

Ex : let myFun = `function () {`

`// stmnts`

`}`

Ex: <script>

```
var test = function () {
    console.log("This is an anonymous function!");
};  
test();
</script>
```

Auto / Immediate Invoked Function:

An Immediate Invoked function is a nameless (anonymous) function that is invoked immediately after its definition.

Ex : `(function () {`

`// stats`

`})());`

Strings:

The **String** object is used to represent and manipulate a sequence of characters.

Ex: let str = “Veda Institute of Technologies”;

String Methods:

1. Length (Prop)
2. indexOf
3. lastIndexOf
4. replace
5. substring
6. substr
7. Charat
8. split

9. to Uppercase
10. to Lowercase

length: is a read-only property it contains the length of the string.

```
Ex : console.log(str.length); // Expected Output : 30
```

indexOf : it returns the index of the first occurrence of specified substring

```
Ex: console.log(str.indexOf("Institute")); // Output : 5
```

lastIndexOf : it returns the index of the last occurrence of specified substring

```
Ex: console.log(str.indexOf("Institute")); // Output : 5
```

replace: The **replace()** method returns a new string with one, some, or all matches of a pattern replaced by a replacement. The pattern can be a string or a RegExp, and the replacement can be a string or a function called for each match. If pattern is a string, only the first occurrence will be replaced. The original string is left unchanged.

```
Ex : console.log(str.replace("Institute","Tech"));
```

```
// Out put : Vedat Tech of Technologies;
```

substring :

The **substring()** method returns the part of the string between the start and end indexes, or to the end of the string.

```
Ex : console.log(str.substring(0,4)); // Output : Veda
```

substr (deprecated): The **substr()** method returns a portion of the string, starting at the specified index and extending for a given number of characters afterwards.

charAt : **charAt()** method returns the character at specified index position

```
Ex: str.charAt(0) // Output : V
```

split : The **split()** method takes a pattern and divides a String into an ordered list of substrings by searching for the pattern, puts these substrings into an array, and returns the array.

```
const str = 'The quick brown fox jumps over the lazy dog.';  
const words = str.split(' ');  
console.log(words[3]);
```

```
// expected output: "fox"
const chars = str.split("");
console.log(chars[8]);
// expected output: "k"
const strCopy = str.split();
console.log(strCopy);
// expected output: Array ["The quick brown fox jumps over the lazy dog."]
```

toUpperCase() : The **toUpperCase()** method returns the calling string value converted to uppercase (the value will be converted to a string if it isn't one).

```
const sentence = 'The quick brown fox jumps over the lazy dog.';
console.log(sentence.toUpperCase());
// expected output: "THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG."
```

to Lowercase (): The **toLowerCase ()** method returns the calling string value converted to lower case.

```
const sentence = 'The quick brown fox jumps over the lazy dog.';
console.log(sentence.toLowerCase());
// expected output: "the quick brown fox jumps over the lazy dog."
```

Events

Events are actions or occurrences that happen in the system you are programming, which the system tells you about so your code can react to them.

For example, if the user clicks a button on a webpage, you might want to react to that action by displaying an information box. In this article, we discuss some important concepts surrounding events, and look at how they work in browsers.

Let's look at a simple example of what we mean here. In the following example, we have a single `<button>`, which when pressed, makes the background change to a random colour:

```
<button>Change color</button>
```



```
const btn = document.querySelector('button');

function random(number) {
    return Math.floor(Math.random() * (number+1));
}

btn.addEventListener('click', () => {
    const rndCol = `rgb(${random(255)}, ${random(255)}, ${random(255)})`;
    document.body.style.backgroundColor = rndCol;
});
```

Other Events:

1. onblur
2. onmouseover
3. onmouseleave
4. onfocus
5. ondblclick
6. onkeypress
7. onkeyup



Keyboard Event

```
<input id="textBox" type="text" />
<div id="output"></div>
```

```
const textBox = document.querySelector("#textBox");
const output = document.querySelector("#output");
textBox.addEventListener('keydown', (event) => output.textContent = `You pressed
"${event.key}"`);
```

Removing Event

If you've added an event handler using `addEventListener()`, you can remove it again using the [`removeEventListener\(\)`](#) method. For example, this would remove the `changeBackground()` event handler:

```
const btn = document.querySelector('button');

function random(number) {
    return Math.floor(Math.random() * (number+1));
}

function changeBackground() {
    const rndCol = `rgb(${random(255)}, ${random(255)}, ${random(255)})`;
    document.body.style.backgroundColor = rndCol;
}

btn.addEventListener('click', changeBackground);
```

```
btn.removeEventListener('click', changeBackground);
```



Arrays:

a neat way of storing a list of data items under a single variable name

Arrays are generally described as "list-like objects"; they are basically single objects that contain multiple values stored in a list. Array objects can be stored in variables and dealt with in much the same way as any other type of value, the difference being that we can access each value inside the list individually, and do super useful and efficient things with the list, like loop through it and do the same thing to every value.

Creating array

Arrays consist of square brackets and items that are separated by commas.

```
const shopping = ['bread', 'milk', 'cheese', 'hummus', 'noodles'];
console.log(shopping);
```



In the above example, each item is a string, but in an array we can store various data types — strings, numbers, objects, and even other arrays. We can also mix data types in a single array we do not have to limit ourselves to storing only numbers in one array, and in another only strings. For example:

```
const sequence = [1, 1, 2, 3, 5, 8, 13];
const random = ['tree', 795, [0, 1, 2]];
```



Length

You can find out the length of an array (how many items are in it) in exactly the same way as you find out the length (in characters) of a string — by using the [length](#) property. Try the following:

```
const shopping = ['bread', 'milk', 'cheese', 'hummus', 'noodles'];
console.log(shopping.length); // 5
```



Accessing and modifying array items:

Items in an array are numbered, starting from zero. This number is called the item's *index*. So the first item has index 0, the second has index 1, and so on. You can access individual items in the array using bracket notation and supplying the item's index

```
const shopping = ['bread', 'milk', 'cheese', 'hummus', 'noodles'];
console.log(shopping[0]);
// returns "bread"
```

We can modify an item in an array by giving a single array item a new value

```
const shopping = ['bread', 'milk', 'cheese', 'hummus', 'noodles'];
shopping[0] = 'tahini';
console.log(shopping);
// shopping will now return [ "tahini", "milk", "cheese", "hummus",
"noodles" ]
```

IndexOf :

You can find the index of a particular item using the [indexOf\(\)](#) method. This takes an item as an argument and returns the index, or -1 if the item was not found in the array:

```
const birds = ['Parrot', 'Falcon', 'Owl'];
console.log(birds.indexOf('Owl')); // 2
console.log(birds.indexOf('Rabbit'));
```

Push :

To add one or more items to the end of an array we can use [push\(\)](#). Note that you need to include one or more items that you want to add to the end of your array.

```
const cities = ['Manchester', 'Liverpool'];
cities.push('Cardiff');
console.log(cities); // [ "Manchester", "Liverpool", "Cardiff" ]
cities.push('Bradford', 'Brighton');
console.log(cities); // [ "Manchester", "Liverpool", "Cardiff", "Bradford",
"Brighton" ]
```

To add an item to the start of the array, use [unshift\(\)](#):

```
const cities = ['Manchester', 'Liverpool'];
cities.unshift('Edinburgh');
console.log(cities); // [ "Edinburgh", "Manchester", "Liverpool" ]
```

Pop

To remove the last item from the array, use [pop\(\)](#).

```
const cities = ['Manchester', 'Liverpool'];
cities.pop();
console.log(cities); // [ "Manchester" ]
```

The pop() method returns the item that was removed. To save that item in a new variable, you could do this:

```
const cities = ['Manchester', 'Liverpool'];
const removedCity = cities.pop();
console.log(removedCity); // "Liverpool"
```

Splice

If you know the index of an item, you can remove it from the array using [splice\(\)](#):

```
const cities = ['Manchester', 'Liverpool', 'Edinburgh', 'Carlisle'];
const index = cities.indexOf('Liverpool');
if (index !== -1) {
  cities.splice(index, 1);
}
console.log(cities); // [ "Manchester", "Edinburgh", "Carlisle" ]
```

In this call to splice(), the first argument says where to start removing items, and the second argument says how many items should be removed. So you can remove more than one item:

```
const cities = ['Manchester', 'Liverpool', 'Edinburgh', 'Carlisle'];
const index = cities.indexOf('Liverpool');
if (index !== -1) {
  cities.splice(index, 2);
}
console.log(cities); // [ "Manchester", "Carlisle" ]
```

Map

Sometimes you will want to do the same thing to each item in an array, leaving you with an array containing the changed items. You can do this using [map\(\)](#). The code below takes an array of numbers and doubles each number:

```
function double(number) {
  return number * 2;
}
const numbers = [5, 2, 7, 6];
const doubled = numbers.map(double);
console.log(doubled); // [ 10, 4, 14, 12 ]
```

Filter

Sometimes you'll want to create a new array containing only the items in the original array that match some test. You can do that using [filter\(\)](#). The code below takes an array of strings and returns an array containing just the strings that are greater than 8 characters long:

```
function isLong(city) {
  return city.length > 8;
}
const cities = ['London', 'Liverpool', 'Totnes', 'Edinburgh'];
const longer = cities.filter(isLong);
console.log(longer); // [ "Liverpool", "Edinburgh" ]
```

Split

The [split\(\)](#) method takes a pattern and divides a String into an ordered list of substrings by searching for the pattern, puts these substrings into an array, and returns the array.

```
const str = 'The quick brown fox jumps over the lazy dog.';

const words = str.split(' ');
console.log(words[3]);
// expected output: "fox"

const chars = str.split('');
console.log(chars[8]);
// expected output: "k"

const strCopy = str.split();
console.log(strCopy);
// expected output: Array ["The quick brown fox jumps over the lazy dog."]
```

Join

The **join()** method creates and returns a new string by concatenating all of the elements in an array separated by commas or a specified separator string. If the array has only one item, then that item will be returned without using the separator.

```
const elements = ['Fire', 'Air', 'Water'];

console.log(elements.join());
// expected output: "Fire,Air,Water"

console.log(elements.join(''));
// expected output: "FireAirWater"

console.log(elements.join('-'));
// expected output: "Fire-Air-Water"
```

Object

The **Object** type represents one of [JavaScript's data types](#). It is used to store various keyed collections and more complex entities. Objects can be created using the [Object\(\)](#) constructor or the [object initializer / literal syntax](#).

The **Object constructor** turns the input into an object. Its behavior depends on the input's type.

- If the value is [null](#) or [undefined](#), it will create and return an empty object.
- Otherwise, it will return an object of a Type that corresponds to the given value.
- If the value is an object already, it will return the value.

```
new Object(value)
Object(value)
```

```
const o = new Object();
o.foo = 42;

console.log(o);
// Object { foo: 42 }
```

The following examples store an empty Object object in o

```
const o = new Object();
```

```
const o = new Object(undefined);
```

```
const o = new Object(null);
```

Object Static methods

Object.assign()

Copies the values of all enumerable own properties from one or more source objects to a target object.

Object.create()

Creates a new object with the specified prototype object and properties.

Object.freeze()

Freezes an object. Other code cannot delete or change its properties.

Object.keys()

Returns an array containing the names of all of the given object's **own** enumerable string properties.

For More Details [Click here](#)

Object initializer:

An object initializer is a comma-delimited list of zero or more pairs of property names and associated values of an object, enclosed in curly braces ({}).

```
const object1 = { a: 'foo', b: 42, c: {} };

console.log(object1.a);
// expected output: "foo"

const a = 'foo';
const b = 42;
const c = {};
const object2 = { a: a, b: b, c: c };

console.log(object2.b);
// expected output: 42

const object3 = { a, b, c };

console.log(object3.a);
// expected output: "foo"
```

Arrays

Arrays are used to store collection of multiple values under single variable name.

JavaScript arrays are resizable and can contain a mix of different data types.

JavaScript arrays are zero-indexed: the first element of an array is at index 0, the second is at index 1, and so on, and the last element is at the value of the array's [length](#) property minus 1.

Accessing an array item by index:

```
const fruits = ['Apple', 'Banana'];

// The index of an array's first element is always 0.
fruits[0]; // Apple

// The index of an array's second element is always 1.
fruits[1]; // Banana

// The index of an array's last element is always one
// less than the length of the array.
fruits[fruits.length - 1]; // Banana

// Using a index number larger than the array's length
// returns 'undefined'.
fruits[99]; // undefined
```

Methods

concat():

The **concat()** method is used to merge two or more arrays. This method does not change the existing arrays, but instead returns a new array.

```
const array1 = ['a', 'b', 'c'];
const array2 = ['d', 'e', 'f'];
const array3 = array1.concat(array2);

console.log(array3);
// expected output: Array ["a", "b", "c", "d", "e", "f"]
```

filter ():

Returns a new array containing all elements of the calling array for which the provided filtering function returns true.

```
const words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];

const result = words.filter(word => word.length > 6);

console.log(result);
// expected output: Array ["exuberant", "destruction", "present"]
```

find()

Returns the value of the first element in the array that satisfies the provided testing function, or undefined if no appropriate element is found.

Example:

```
<script>
cons ages = [3, 10, 18, 20];

document.getElementById("demo").innerHTML = ages.find(checkage);

function checkage(age) {
  return age > 18;
}
</script>
```

find Index ()

Returns the index of the first element in the array that satisfies the provided testing function, or -1 if no appropriate element was found.

Example:

```
<script>
cons ages = [3, 10, 18, 20];

document.getElementById("demo").innerHTML = ages.findIndex(checkage);

function checkage(age) {
  return age > 18;
}
</script>
```

find Last ()

Returns the value of the last element in the array that satisfies the provided testing function, or undefined if no appropriate element is found.

Example:

```
cons array1 = [5, 12, 50, 130, 44];

cons found = array1.findLast((element) => element > 45);

console.log(found);
// expected output: 130
```

findLastIndex():

Returns the index of the last element in the array that satisfies the provided testing function, or -1 if no appropriate element was found.

Example:

```
cons array1 = [5, 12, 50, 130, 44];  
  
cons isLargeNumber = (element) => element > 45;  
  
console.log (array1.findIndex(isLargeNumber));  
// expected output: 3 (of element with value: 130)  
for Each ()
```

Calls a function for each element in the calling array.

Example:

```
<script>  
let text = "";  
cons fruits = ["apple", "orange", "cherry"];  
fruits.forEach(my Function);  
  
document.getElementById("demo").inner HTML = text;
```

```
function my Function (item, index) {  
  text += index + ": " + item + "<br>";  
}  
</script>
```

o/p:
0: apple
1: orange
2: cherry

includes()

Determines whether the calling array contains a value, returning true or false as appropriate.

Example:

```
<script>  
cons fruits = ["Banana", "Orange", "Apple", "Mango"];  
document.getElementById("demo").inner HTML = fruits. Includes("Mango");  
</script>
```

indexOf()

Returns the first (least) index at which a given element can be found in the calling array.

Example:

```
<script>
```

```
cons fruits = ["Banana", "Orange", "Apple", "Mango"];
let index = fruits.indexOf("Apple");

document.getElementById("demo").innerHTML = index;
</script>
```

o/p:

2

join()

Joins all elements of an array into a string.

Example:

```
<script>
cons fruits = ["Banana", "Orange", "Apple", "Mango"];
let text = fruits.join();
document.getElementById("demo").innerHTML = text;
</script>
```

o/p:

Banana,Orange,Apple,Mango

lastIndexOf()

Returns the last (greatest) index at which a given element can be found in the calling array, or -1 if none is found.

Example:

```
<script>
cons fruits = ["Apple", "Orange", "Apple", "Mango"];
let index = fruits.lastIndexOf("Apple");

document.getElementById("demo").innerHTML = index;
</script>
```

o/p:

2

map()

Returns a new array containing the results of invoking a function on every element in the calling array.

Example:

```
<script>  
const numbers = [4, 9, 16, 25];  
  
document.getElementById("demo").innerHTML = numbers.map(Math.sqrt);  
</script>
```

o/p:

2,3,4,5

pop()

Removes the last element from an array and returns that element.

Example:

```
<script>  
const fruits = ["Banana", "Orange", "Apple", "Mango"];  
  
fruits.pop();  
  
document.getElementById("demo").innerHTML = fruits;  
</script>
```

o/p:

Banana,Orange,Apple

push()

Adds one or more elements to the end of an array, and returns the new length of the array.

Example:

```
<script>
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.push("Kiwi");
```

```
document.getElementById("demo").innerHTML = fruits;
</script>
```

o/p:

Banana,Orange,Apple,Mango,Kiwi

reduce()

Executes a user-supplied "reducer" callback function on each element of the array (from left to right), to reduce it to a single value.

Example:

```
<script>
const numbers = [175, 50, 25];
document.getElementById("demo").innerHTML = numbers.reduce(myFunc);
```

```
function myFunc(total, num) {
    return total - num;
}
</script>
```

o/p:

100

reduceRight()

Executes a user-supplied "reducer" callback function on each element of the array (from right to left), to reduce it to a single value.

Example:

```
<script>  
const numbers = [2, 45, 30, 100];  
  
document.getElementById("demo").innerHTML = numbers.reduceRight(getSum);
```

```
function getSum(total, num) {  
    return total - num;  
}  
</script>
```

o/p:

23

reverse()

Reverses the order of the elements of an array *in place*. (First becomes the last, last becomes first.)

Example:

```
<script>  
const fruits = ["Banana", "Orange", "Apple", "Mango"];  
  
document.getElementById("demo").innerHTML = fruits.reverse();
```

```
</script>
```

o/p:

Mango,Apple,Orange,Banana

shift()

Removes the first element from an array and returns that element.

Example:

```
<script>  
const fruits = ["Banana", "Orange", "Apple", "Mango"];  
  
fruits.shift();
```

```
document.getElementById("demo").innerHTML = fruits;
```

```
</script>
```

o/p:

Orange,Apple,Mango

slice()

Extracts a section of the calling array and returns a new array.

Example:

```
<script>
```

```
const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];  
  
const citrus = fruits.slice(1, 3);
```

```
document.getElementById("demo").innerHTML = citrus;
```

</script>

o/p:

Orange,Lemon

some()

Returns true if at least one element in the calling array satisfies the provided testing function.

Example:

```
<script>
```

```
const ages = [3, 10, 18, 20];
```

```
document.getElementById("demo").innerHTML = ages.some(checkAdult);
```

```
function checkAdult(age) {  
    return age > 18;  
}  
</script>
```

o/p:

true

sort()

Sorts the elements of an array in place and returns the array.

Example:

```
<script>
```

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
```

```
document.getElementById("demo").innerHTML = fruits.sort();
```

</script>

o/p:

Apple,Banana,Mango,Orange

splice()

Adds and/or removes elements from an array.

Example:

<script>

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
```

```
// At position 2, add 2 elements:
```

```
fruits.splice(2, 0, "Lemon", "Kiwi");
```

```
document.getElementById("demo").innerHTML = fruits;
```

```
</script>
```

o/p:

Banana,Orange,Lemon,Kiwi,Apple,Mango

unshift()

Adds one or more elements to the front of an array, and returns the new length of the array.

Example:

<script>

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
```

```
fruits.unshift("Lemon", "Pineapple");
```

```
document.getElementById("demo").innerHTML = fruits;  
</script>
```

o/p:

Lemon,Pineapple,Banana,Orange,Apple,Mango

Loops and iteration

Loops offer a quick and easy way to do something repeatedly.

There are many different kinds of loops, but they all essentially do the same thing: they repeat an action some number of times. (Note that it's possible that number could be zero!)

The various loop mechanisms offer different ways to determine the start and end points of the loop. There are various situations that are more easily served by one type of loop over the others.

for statement:

A for loop repeats until a specified condition evaluates to false

```
for ([initialExpression]; [conditionExpression]; [incrementExpression])  
  
<form name="selectForm">  
  <label for="musicTypes">Choose some music types, then click the button below:</label>  
  <select id="musicTypes" name="musicTypes" multiple>  
    <option selected>R&B</option>  
    <option>Jazz</option>  
    <option>Blues</option>  
    <option>New Age</option>  
    <option>Classical</option>  
    <option>Opera</option>  
  </select>  
  <button id="btn" type="button">How many are selected?</button>  
</form>
```

```
function howMany(selectObject) {  
    let numberSelected = 0;  
    for (let i = 0; i < selectObject.options.length; i++) {  
        if (selectObject.options[i].selected) {  
            numberSelected++;  
        }  
    }  
    return numberSelected;  
}  
  
const btn = document.getElementById('btn');  
  
btn.addEventListener('click', () => {  
    const musicTypes = document.selectForm.musicTypes;  
    console.log(`You have selected ${howMany(musicTypes)} option(s).`);  
});
```

do...while statement:

The **do...while statement** creates a loop that executes a specified statement until the test condition evaluates too false. The condition is evaluated after executing the statement, resulting in the specified statement executing at least once.

```
let result = '';  
let i = 0;  
  
do {  
    i = i + 1;  
    result = result + i;  
} while (i < 5);  
  
console.log(result);  
// expected result: "12345"
```

while statement

The **while statement** creates a loop that executes a specified statement as long as the test condition evaluates to true. The condition is evaluated before executing the statement.

```
let n = 0;  
  
while (n < 3) {  
    n++;  
}  
  
console.log(n);  
// expected output: 3
```

Exception:

An **exception** is a condition that interrupts normal code execution. In JavaScript syntax errors are a very common source of exceptions.

throw statement

Use the throw statement to throw an exception. A throw statement specifies the value to be thrown:

```
throw expression;
```

You may throw any expression, not just expressions of a specific type. The following code throws several exceptions of varying types:

```
throw 'Error2'; // String type  
throw 42; // Number type  
throw true; // Boolean type  
throw {toString() { return "I'm an object!"; } };
```

Handling Exceptions

try...catch statement:

The try...catch statement marks a block of statements to try, and specifies one or more responses should an exception be thrown. If an exception is thrown, the try...catch statement catches it.

The try...catch statement consists of a try block, which contains one or more statements, and a catch block, containing statements that specify what to do if an exception is thrown in the try block.

```

function getMonthName(mo) {
    mo--; // Adjust month number for array index (so that 0 = Jan, 11 = Dec)
    const months = [
        'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
        'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec',
    ];
    if (months[mo]) {
        return months[mo];
    } else {
        throw new Error('InvalidMonthNo'); // throw keyword is used here
    }
}

try { // statements to try
    monthName = getMonthName(myMonth); // function could throw exception
} catch (e) {
    monthName = 'unknown';
    logMyErrors(e); // pass exception object to error handler (i.e. your own
function)
}

```

finally block:

The finally block contains statements to be executed *after* the try and catch blocks execute. Additionally, the finally block executes *before* the code that follows the try...catch...finally statement.

It is also important to note that the finally block will execute *whether or not* an exception is thrown. If an exception is thrown, however, the statements in the finally block execute even if no catch block handles the exception that was thrown.

You can use the finally block to make your script fail gracefully when an exception occurs. For example, you may need to release a resource that your script has tied up.

The following example opens a file and then executes statements that use the file. (Server-side JavaScript allows you to access files.) If an exception is thrown while the file is open, the finally block closes the file before the script fails. Using finally here *ensures* that the file is never left open, even if an error occurs.

```
openMyFile();
try {
  writeMyFile(theData); // This may throw an error
} catch (e) {
  handleError(e); // If an error occurred, handle it
} finally {
  closeMyFile(); // Always close the resource
}
```

If the finally block returns a value, this value becomes the return value of the entire try...catch...finally production, regardless of any return statements in the try and catch blocks:

```
function f() {
  try {
    console.log(0);
    throw 'bogus';
  } catch (e) {
    console.log(1);
    return true;      // this return statement is suspended
                     // until finally block has completed
    console.log(2); // not reachable
  } finally {
    console.log(3);
    return false;    // overwrites the previous "return"
    console.log(4); // not reachable
  }
  // "return false" is executed now
  console.log(5); // not reachable
}
console.log(f()); // 0, 1, 3, false
```

Errors

Range Error:

The **RangeError** object indicates an error when a value is not in the set or range of allowed values.

ReferenceError :

The **ReferenceError** object represents an error when a variable that doesn't exist (or hasn't yet been initialized) in the current scope is referenced.

```
try {
  let a = undefinedVariable
} catch (e) {
  console.log(e instanceof ReferenceError) // true
  console.log(e.message) // "undefinedVariable is not
defined"
  console.log(e.name) // "ReferenceError"
  console.log(e.fileName) // "Scratchpad/1"
  console.log(e.lineNumber) // 2
  console.log(e.columnNumber) // 6
  console.log(e.stack) // "@Scratchpad/2:2:7\n"
}
```



SyntaxError :

The **SyntaxError** object represents an error when trying to interpret syntactically invalid code. It is thrown when the JavaScript engine encounters tokens or token order that does not conform to the syntax of the language when parsing code.

```
try {
  eval('hoo bar');
} catch (e) {
  console.error(e instanceof SyntaxError);
  console.error(e.message);
  console.error(e.name);
  console.error(e.fileName);
  console.error(e.lineNumber);
  console.error(e.columnNumber);
  console.error(e.stack);
}
```

[Click here to see more error types](#)

Promises:

The **Promise** object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

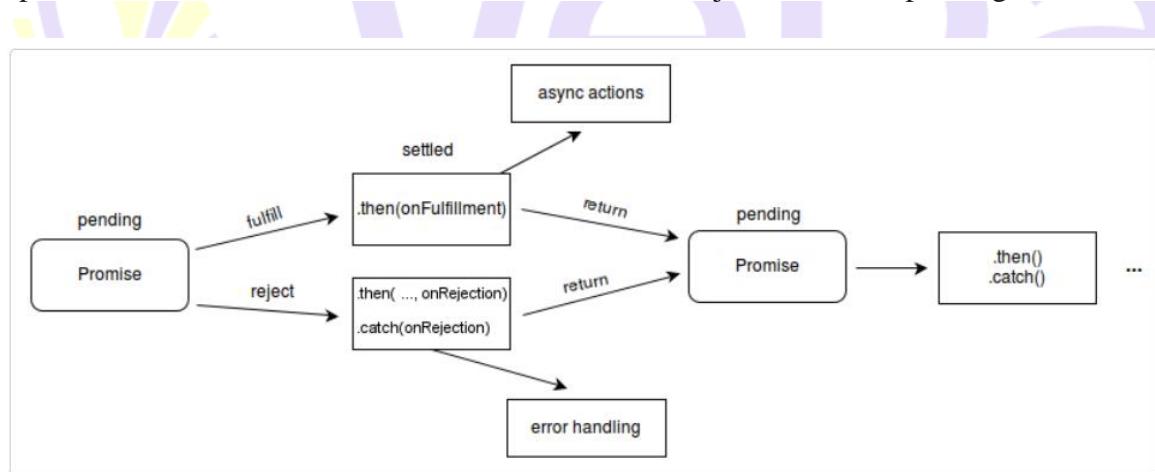
A **Promise** is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a *promise* to supply the value at some point in the future.

A Promise is in one of these states:

- *pending*: initial state, neither fulfilled nor rejected.
- *fulfilled*: meaning that the operation was completed successfully.
- *rejected*: meaning that the operation failed.

The *eventual state* of a pending promise can either be *fulfilled* with a value or *rejected* with a reason (error). When either of these options occur, the associated handlers queued up by a promise's `then` method are called. If the promise has already been fulfilled or rejected when a corresponding handler is attached, the handler will be called, so there is no race condition between an asynchronous operation completing and its handlers being attached.

A promise is said to be *settled* if it is either fulfilled or rejected, but not pending.



`.then()`

The `then()` method returns a Promise. It takes up to two arguments: call back functions for the success and failure cases of the Promise.

```
const promise1 = new Promise((resolve, reject) => {
  resolve('Success!');
});

promise1.then((value) => {
  console.log(value);
  // expected output: "Success!"
});
```

.catch()

The **catch()** method returns a Promise and deals with rejected cases only. It behaves the same as calling `Promise.prototype.then(undefined, onRejected)`

```
const promise1 = new Promise((resolve, reject) => {
  throw 'Uh-oh!';
});

promise1.catch((error) => {
  console.error(error);
});
// expected output: Uh-oh!
```



Date

JavaScript **Date** objects represent a single moment in time in a platform-independent format. Date objects contain a Number that represents milliseconds since 1 January 1970 UTC.

Date () constructor

The **Date ()** constructor can create a Date instance or return a string representing the current time

```
const date1 = new Date('December 17, 1995 03:24:00');
// Sun Dec 17 1995 03:24:00 GMT...

const date2 = new Date('1995-12-17T03:24:00');
// Sun Dec 17 1995 03:24:00 GMT...

console.log(date1 === date2);
// expected output: false;

console.log(date1 - date2);
// expected output: 0
```

[Click here](#) to see All date methods.

Predefined Methods:

ParseInt:

The `parseInt` method parses a value as a string and returns the first integer.

Ex:`parseInt("40 years")` is 40

parse Float ():

Parses a string argument and returns a floating-point number

Ex: `parse Float ("10.33")` is 10.33

escape ():

Returns the hexadecimal encoding of an argument

unescaped ():

Returns the ASCII string for the specified value

React Index

1 Code Editors

1.1 Setting up VS Code

2 Server environments

2.1 Installing Nodejs

3 Fundamentals

3.1 What is ReactJS

3.2 Why react instead of JavaScript

3.3 Building SPA's with react

3.4 Briefing Datatypes

3.5 Arrow function briefing

3.6 Exports and imports

3.7 Classes properties and methods

3.8 How to use react in a website

3.9 Create a new react app

3.10 Basic starting program

3.11 JSX intro

3.12 Create Basic Component

3.12.1 Class

3.12.2 Functional

3.13 Building a custom component

3.14 Props

3.15 State Intro

3.16 Events

3.17 List rendering

3.18 Conditional renderings

3.19 Basic CSS styling

3.20 Splitting components into multiple components

4 Advanced

4.1 Forms

4.2 State management

4.2.1 Working with state

4.2.2 useState hook usage

4.2.3 Working with multiple states

4.2.4 Child to parent communication

4.3 Life cycles

4.3.1 Component did mount

4.3.2 Component will unmount

4.4 React Hooks

4.4.1 Introducing hook

4.4.2 State hook

4.4.3 Effect hook

React Index

4.4.4 Hooks rules

4.5 Routing

- 4.5.1 Install react-router
- 4.5.2 Routing intro
- 4.5.3 Link usage
- 4.5.4 Parameters
- 4.5.5 Custom navigation usage
- 4.5.6 Nested routing

4.6 HTTP requests

4.7 Redux

- 4.7.1 Store
- 4.7.2 Action
- 4.7.3 Reducers
- 4.7.4 Redux api
- 4.7.5 Provider component

5 Framework

5.1 Integrating with other libraries

- 5.1.1 Installation
- 5.1.2 Process and rules
- 5.1.3 Usage

5.2 Identity

- 5.2.1 Login
- 5.2.2 Logout
- 5.2.3 Configuration Setup

5.3 Antd

- 5.3.1 Usage of Antd
- 5.3.2 Form
- 5.3.3 Validations
- 5.3.4 Upload
- 5.3.5 Model popup

5.4 Kendo grid

5.5 Https requests Uage

- 5.5.1 Authorization
- 5.5.2 Authentication

5.6 Redux usage in application

5.7 Sample Project.

React Fundamental's

1. Code Editors

1.1. Setting up VS Code

2. Server environments

2.1. Installing Nodejs

3. Fundamentals

- 3.1. What is reactjs
- 3.2. Why react instead of JavaScript
- 3.3. Building SPA's with react
- 3.4. Briefing Datatypes
- 3.5. Arrow function briefing
- 3.6. Create a new react app
- 3.7. Basic starting program
- 3.8. JSX intro
- 3.9. Create Basic Components (Class and functional)
- 3.10. Exports and imports
- 3.11. Building a custom component
- 3.12. Props
- 3.13. State Intro
- 3.14. Events
- 3.15. List rendering
- 3.16. Conditional renderings
- 3.17. Basic CSS styling

1. Code Editors

Visual Studio Code is a **code** editor redefined and optimized for building and debugging modern web and cloud applications. **Visual Studio Code** is free and available on your favorite platform.

To download Visual studio code, use the link below

<https://code.visualstudio.com/Download>

1.1. Setting up VS Code

Step1: open above link and download base on platform OS.

Step2: Run download file and continue to install.

Step3: Accept All teams and conditions and continue till end of installation.

2. Server environments

Nodejs:

Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine.

As an asynchronous event-driven JavaScript runtime, Node.js is designed to build scalable network applications. In the following "hello world" example, many connections can be handled concurrently. Upon each connection, the callback is fired, but if there is no work to be done, Node.js will sleep

React Fundamental's

To download Nodejs use below link.

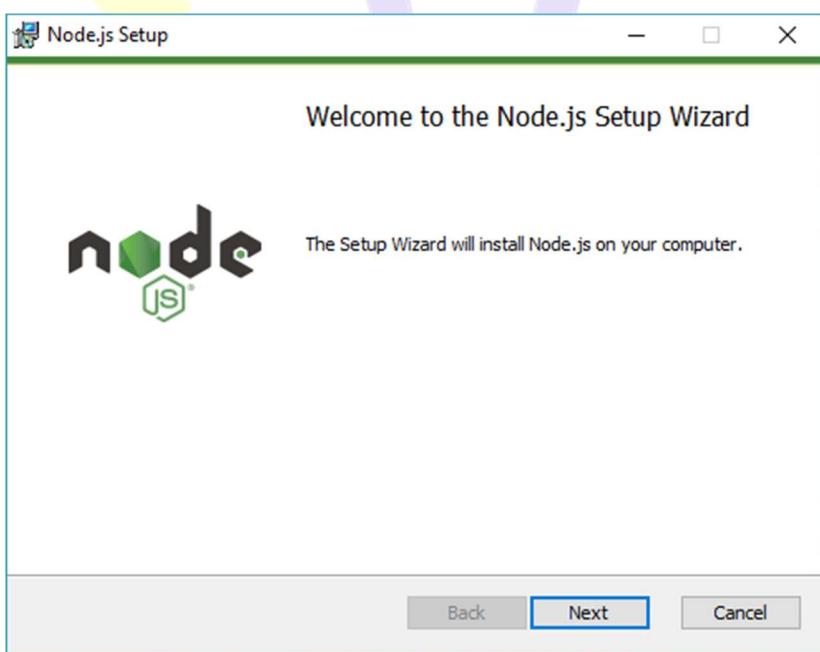
<https://nodejs.org/en/download/>

2.1. Installation

Running the Node.js installer.

Now you need to install the node.js installer on your PC. You need to follow the following steps for the Node.js to be installed.

- Double click on the .msi installer.
The Node.js Setup wizard will open.
- Welcome To Node.js Setup Wizard.



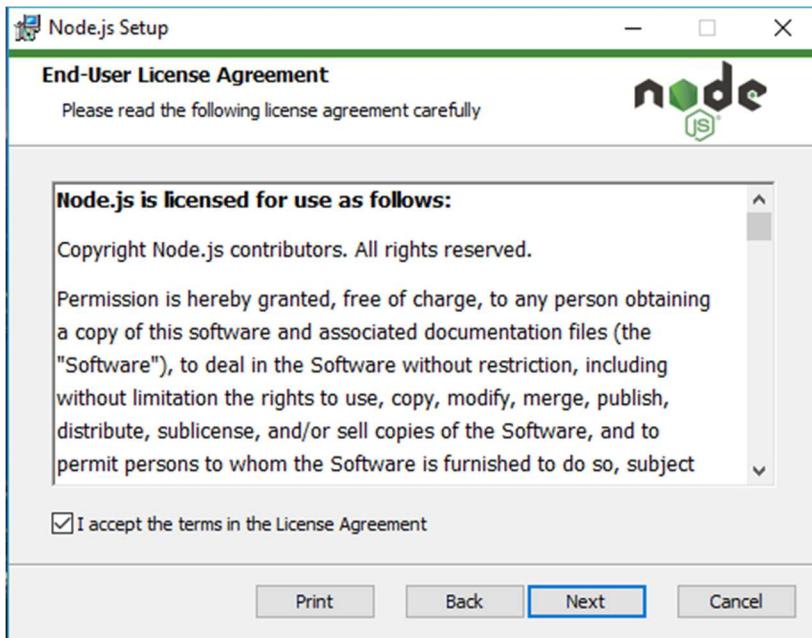
Select “Next”

- After clicking “Next”, End-User License Agreement (EULA) will open.

React Fundamental's

Check "I accept the terms in the License Agreement"

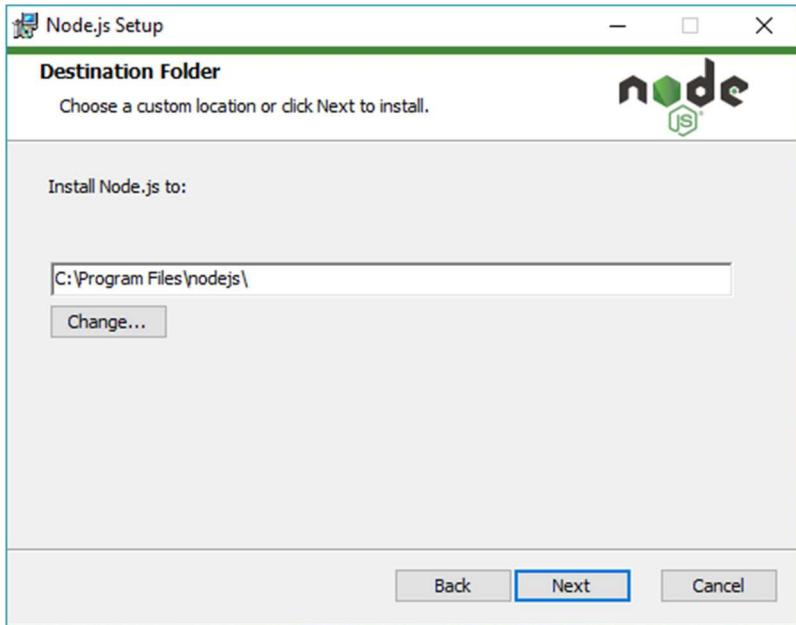
Select "Next"



- Destination Folder

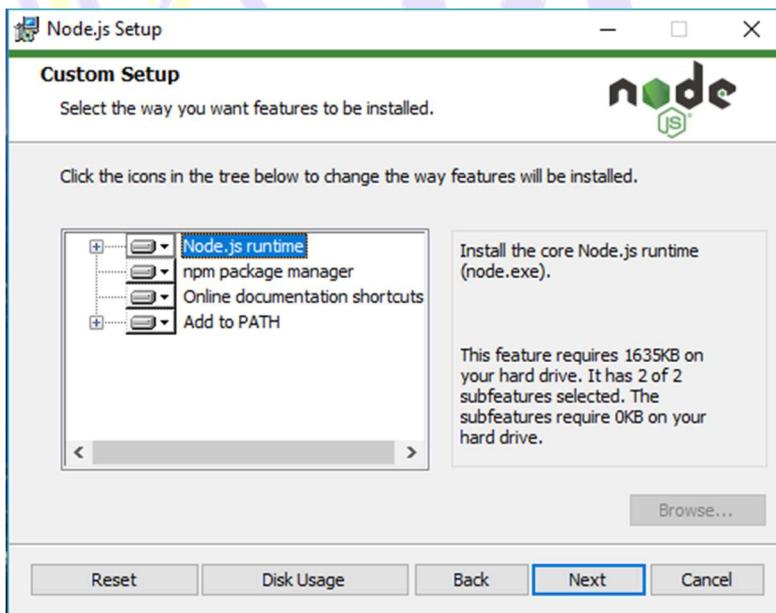
Set the Destination Folder where you want to install Node.js & **Select "Next"**

React Fundamental's



- Custom Setup

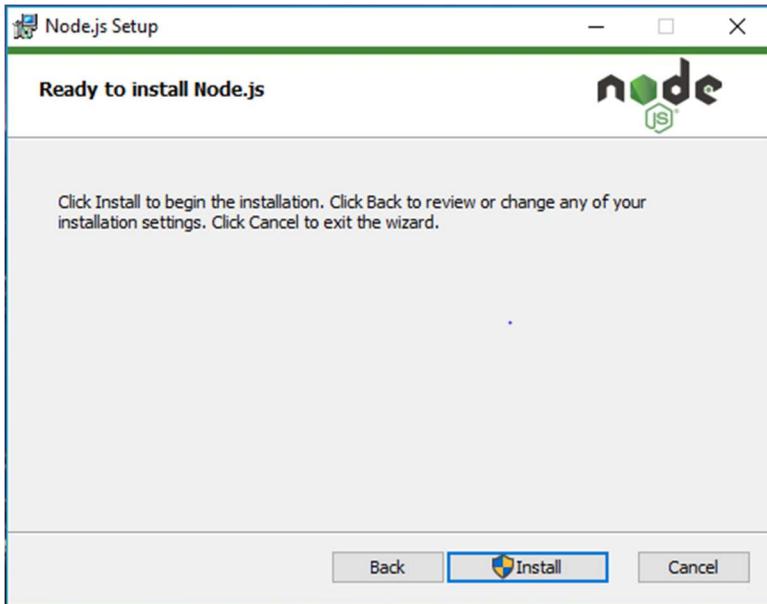
Select “Next”



- Ready to Install Node.js.

Select “Install”

React Fundamental's



NOTE :

A prompt saying – “This step requires administrative privileges” will appear.

Authenticate the prompt as an “Administrator”

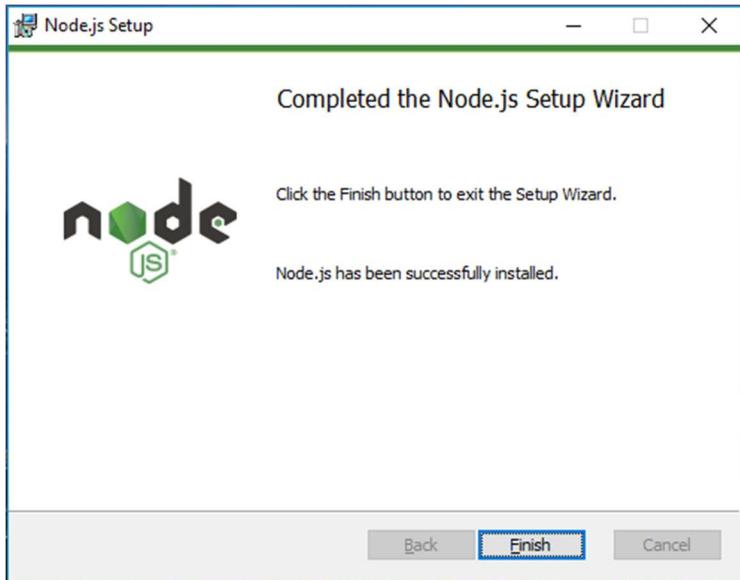
- Installing Node.js.

Do not close or cancel the installer until the install is complete

- Complete the Node.js Setup Wizard.

Click “Finish”

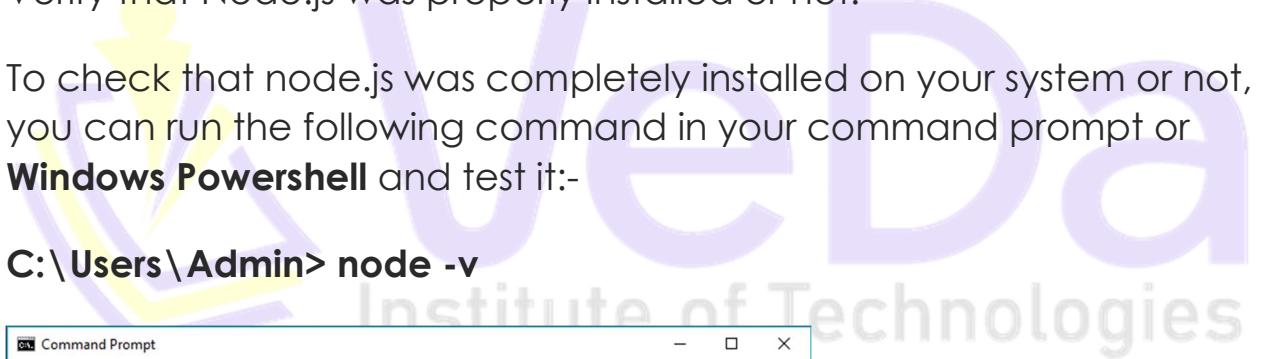
React Fundamental's



Verify that Node.js was properly installed or not.

To check that node.js was completely installed on your system or not, you can run the following command in your command prompt or **Windows Powershell** and test it:-

C:\Users\Admin> node -v

A screenshot of a Windows Command Prompt window. The title bar says "Command Prompt". The window displays the following text:

```
Microsoft Windows [Version 10.0.16299.547]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\Admin>node -v
v10.15.3

C:\Users\Admin>
```

3. Fundamentals

3.1. What is reactjs

React.js was released by a software engineer working for Facebook – Jordane Walke in 2011,

React is a JavaScript library focused on creating declarative, efficient, and flexible user interfaces (UIs). It lets you compose complex UIs from small and isolated pieces of code called “components”. It’s used for handling the view layer and can be used for web and mobile apps. React’s main goal is to be extensive, fast, declarative, flexible, and simple.

React is not a framework, it is specifically a library. The explanation for this is that React only deals with rendering UIs and reserves many things at the discretion of individual projects. The standard set of tools for creating an application using ReactJS is frequently called the stack.

The initial version, 0.3.0 of React is released on May, 2013 and the latest version, 18.2.0 is released on June, 2022.

3.2. Why react instead of JavaScript

We can do everything with JavaScript when it comes to web development without any doubt.

After all these libraries/frameworks (React.JS, Angular, jQuery, and Vue) are built on top of the JavaScript or TypeScript (which is again a superset of JavaScript). So, under the hood, it's all JavaScript which runs.

We all know that there is a difference in hard work and smart work and we also know that smart work can only be achieved by putting in a lot of hard work.

When it comes to web application development, think of JavaScript as a Hard work and React.Js as Smart Work because there is a wide range of developers/communities who have made all the hard work by writing JavaScript codes and exporting it as a library called “React.Js” so, that we as a web developers can do some smart work.

3.3. Building SPA's with react

SPA stands for Single Page Application. It is a very common way of programming websites these days. The idea is that the website loads all the HTML/JS the first time you visit. When you then navigate, the browser will only re-render the content without refreshing the website.

This is used to make the user experience feel a lot smoother. You can tell when it's a SPA or multi-page application when navigating between menus often because a multi-page application will reload, making the whole UI blink fast depending on the content. This is due to it refreshing the website. A SPA will instead feel smooth in the transaction as it simply shows other content without refreshing.

First, create the application template using create-react-app.

```
create-react-app app
```

React Fundamental's

Enter the newly created app folder and install the React Router, which we will use to create the SPA.

```
npm i react-router-dom --save
```

We will begin by modifying the created application, open up src/App.js and replace everything in it with the following:

```
import
'./App.css';

import Main from './layouts/main';

function App() {
  return (
    <Main></Main>
  );
}

export default App;
```

App.js removing the original splash screen

Notice that we are returning the Main component? Let's create it so the application doesn't crash. Create a folder called layouts and a file in it called layouts/main.js. This file will hold the base layout of the application.

```
mkdir layouts
```

React Fundamental's

touch layouts/main.js

In the main.js, we will create a navbar that looks very bad, but we are here to learn React Router and not CSS. This will have navigation links to an about page and a contact page. Below the navbar, we will have a content div that will be used to render the selected link. This is a very common SPA setup where you have a static layout with navigation and a card that changes the content. Here's the code:

```
import Navbar from
'../components/navbar/navbar'

function Main() {
  return (
    <div>
      <Navbar></Navbar>
      <div className="content">
        </div>
      </div>
    )
}

export default Main;
```

Main.js – The applications default layout

Let's create the navbar and then visit the page. Create a folder in the src called components which will hold the components of the application. Then create components/navbar/navbar.js .

mkdir -p components/navbar/

Veda Institute of technologies

React Fundamental's

touch components/navbar/navbar.js

```
function
Navbar()
{
    return (
        <div className="navbar">
            <h1>Navbar</h1>
            <ul>
                <li><a href="/">Home</a></li>
                <li><a href="/about">about</a></li>
                <li><a href="/contact">Contact</a></li>
            </ul>
        </div>
    )
}

export default Navbar;
```

Navbar.js – A simple list of links

Finally, open src/index.css, and paste in the following CSS snippet to render the navbar as a horizontal list instead:

```
.navbar
ul {
    list-style-type: none;
    margin: 0;
    padding: 0;
    overflow: hidden;
}

.navbar li {
    float: left;
```

```
}

.navbar li a {
  display: block;
  padding: 8px;
  background-color: #dddddd;
}
```

index.css – Adding horizontal navbar

Start the application and visit localhost:3000 to view the resulting application.

npm start

3.4. Briefing Datatypes

JavaScript provides different data types to hold different types of values. There are two types of data types in JavaScript.

- ✓ Primitive data type
- ✓ Non-primitive (reference) data type

JavaScript primitive data types

Data Type	Description
String	represents sequence of characters e.g. "hello"
Number	represents numeric values e.g. 100
Boolean	represents boolean value either false or true

React Fundamental's

Undefined	represents undefined value
Null	represents null i.e. no value at all

JavaScript non-primitive data types

Data Type	Description
Object	represents instance through which we can access members
Array	represents group of similar values
RegExp	represents regular expression

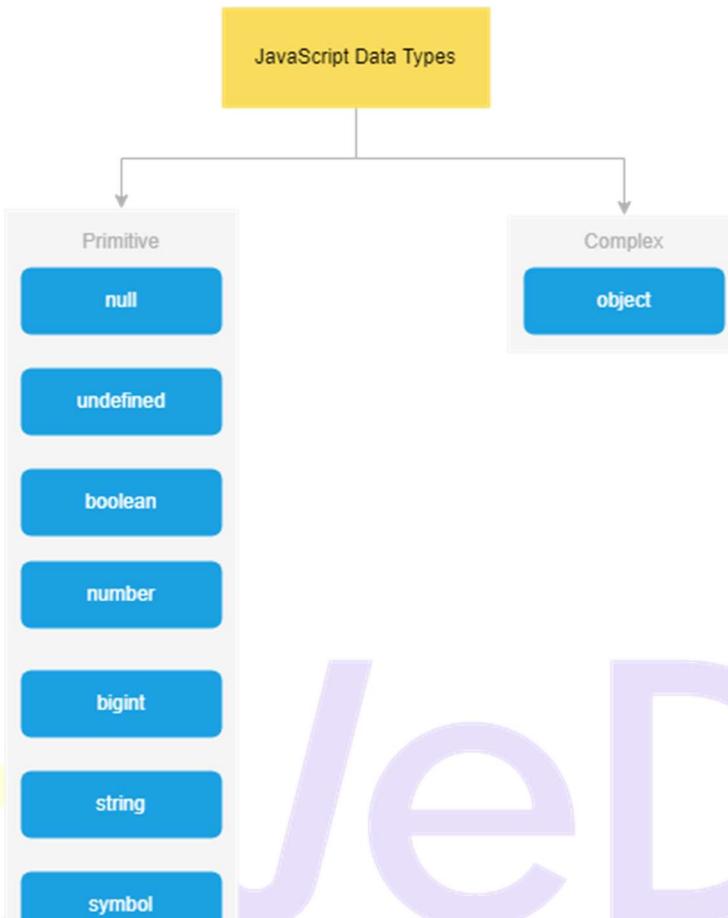
String

In JavaScript, a string is a sequence of zero or more characters. A string literal begins and ends with either a single quote(') or a double quote (").

```
let greeting = 'Hi'; //string

let message = "Bye"; //string
```

React Fundamental's



JeDa

institute of Technologies

Undefined

The undefined type is a primitive type that has only one value undefined. By default, when a variable is declared but not initialized, it is assigned the value of undefined.

```
let counter;  
console.log(counter); // undefined  
console.log(typeof counter); // undefined
```

Boolean

The boolean type has two literal values: true and false in lowercase. The following example declares two variables that hold the boolean values.

```
let inProgress = true;
let completed = false;

console.log(typeof completed); // boolean

console.log(Boolean('Hi')) // true console.log(Boolean('')) // false
console.log(Boolean(20)); // true console.log(Boolean(Infinity)); // true
console.log(Boolean(0)); // false console.log(Boolean({foo: 100})); // true on
non-empty object

console.log(Boolean(null)); // false
```

Number

JavaScript uses the number type to represent both integer and floating-point numbers.

```
let x = 16;

let y = 16.2;

console.log(typeof(x)) //number

console.log(typeof(y)) //number
```

NaN

NaN stands for Not a Number. It is a special numeric value that indicates an invalid number. For example, the division of a string by a number returns NaN

```
console.log('a'/2); // NaN  
console.log(NaN/2); // NaN  
console.log(NaN == NaN); // false
```

Bigint – available from ES2020

The bigint type represents the whole numbers that are larger than $2^{53} - 1$. To form a bigint literal number, you append the letter n at the end of the number

```
let pageView = 9007199254740991n;  
console.log(typeof(pageView)); // 'bigint'
```

object

In JavaScript, an object is a collection of properties, where each property is defined as a key-value pair

```
let emptyObject = {};  
  
let person = {  
    firstName: 'John',  
  
    lastName: 'Doe'  
};
```

3.5. Arrow function briefing

ES6 arrow functions provide you with an alternative way to write a shorter syntax compared to the function expression.

The following example defines a function expression that returns the sum of two numbers:

```
let add = function (x, y) {  
    return x + y;  
};  
console.log(add(10, 20)); // 30
```

add() function expression but use an arrow function

instead:

```
let add = (x, y) => x + y;  
console.log (add (10, 20)); // 30;  
  
let add = (x, y) => {return x + y;};  
  
console.log(typeof add); //function
```

3.6. Create a new react app

If you have npx and Node.js installed, you can create a React application by using **create-react-app**

If you've previously installed **create-react-app** globally, it is recommended that you uninstall the package to ensure npx always uses the latest version of **create-react-app**.

To uninstall, run this command: **npm uninstall -g create-react-app**

Run this command to create a React application named **my-react-app**

```
npx create-react-app my-react-app
```

The **create-react-app** will set up everything you need to run a React application.

Run the React Application

Now you are ready to run your first *real* React application!

Run this command to move to the **my-react-app** directory

```
cd my-react-app
```

Run this command to run the React application **my-react-app**

```
npm start
```

3.7. Basic starting program

Look in the **my-react-app** directory, and you will find a **src** folder. Inside the **src** folder there is a file called **App.js**, open it and it will look like this

```
/myReactApp/src/App.js
```

```
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
```

React Fundamental's

```
<img src={logo} className="App-logo" alt="logo" />

<p>
  Edit <code>src/App.js</code> and save to reload.
</p>

<a
  className="App-link"
  href="https://reactjs.org"
  target="_blank"
  rel="noopener noreferrer"
>
  Learn React
</a>
</header>

</div>

};

}

export default App;
```

Example

Replace all the content inside the **<div className="App">** with a **<h1>** element.

See the changes in the browser when you click Save.

```
function App() {  
  return (  
    <div className="App">  
      <h1>Hello World!</h1>  
    </div>  
  );  
}  
  
export default App;
```

3.8. JSX intro

JSX stands for JavaScript XML.

JSX allows us to write HTML in React.

JSX makes it easier to write and add HTML in React.

React Fundamental's

Coding JSX

JSX allows us to write HTML elements in JavaScript and place them in the DOM without any **createElement()** and/or **appendChild()** methods.

JSX converts HTML tags into react elements.

You are not required to use JSX, but JSX makes it easier to write

Example

JSX:

```
const myElement = <h1>I Love JSX!</h1>;  
  
const root =  
ReactDOM.createRoot(document.getElementById('root'));  
  
root.render(myElement);
```

Without JSX

```
const myElement = React.createElement('h1', {}, 'I do not use JSX!');  
  
const root = ReactDOM.createRoot(document.getElementById('root'));  
root.render(myElement);
```

Expressions in JSX

With JSX you can write expressions inside curly **braces { }**.

The expression can be a React variable, or property, or any other valid JavaScript expression. JSX will execute the expression and return the result

```
const myElement = <h1>React is {5 + 5} times better with JSX</h1>;
```

Inserting a Large Block of HTML

To write HTML on multiple lines, put the HTML inside parentheses

```
const myElement = (  
  <ul>  
    <li>Apples</li>  
    <li>Bananas</li>  
    <li>Cherries</li>  
  </ul>  
);
```

One Top Level Element

The HTML code must be wrapped in **ONE** top level element.

So if you like to write two paragraphs, you must put them inside a parent element, like a div element.

```
const myElement = (
  <div>
    <p>I am a paragraph.</p>
    <p>I am a paragraph too.</p>
  </div>
);
```

Attribute **class** = **className**

The class attribute is a much used attribute in HTML, but since JSX is rendered as JavaScript, and the class keyword is a reserved word in JavaScript, you are not allowed to use it in JSX.

Use attribute **className** instead.

```
const myElement = <h1 className="myclass">Hello World</h1>;
```

Conditions - if statements

React supports if statements, but not inside JSX.

To be able to use conditional statements in JSX, you should put the if statements outside of the JSX, or you could use a ternary expression instead:

Option 1:

Write if statements outside of the JSX code:

Example

React Fundamental's

Write "Hello" if x is less than 10, otherwise "Goodbye":

```
const x = 5;  
  
let text = "Goodbye";  
  
if (x < 10) {  
  
    text = "Hello";  
  
}  
  
const myElement = <h1>{text}</h1>;
```

Option 2:

Use ternary expressions instead:

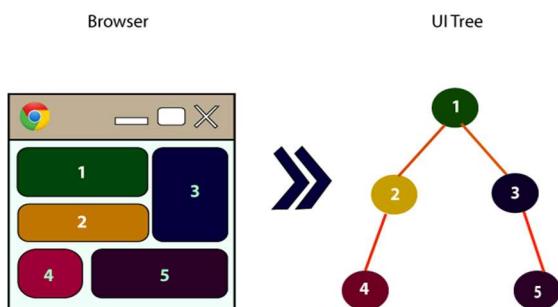
Example

Write "Hello" if x is less than 10, otherwise "Goodbye":

```
const x = 5;  
  
const myElement = <h1>{(x) < 10 ? "Hello" : "Goodbye"}</h1>;
```

3.9. Building a custom component

Components are independent and reusable bits of code. They serve the same purpose as JavaScript functions, but work in isolation and return HTML.



Class Components

A class component must include the `extends React.Component` statement. This statement creates an inheritance to `React.Component` and gives your component access to `React.Component`'s functions.

The component also requires a `render()` method, this method returns HTML.

Class components are more complex than functional components.

```
class MyComponent extends React.Component {  
  render() {
```

React Fundamental's

```
return (  
    <div>This is main component.</div>  
);  
}  
}
```

Example:

```
import React, { Component } from 'react';  
  
class App extends React.Component {  
  constructor() {  
    super();  
    this.state = {  
      data:  
      [  
        {  
          "name": "Abhishek"  
        },  
        {  
          "name": "Saharsh"  
        },  
        {  
          "name": "Ajay"  
        }  
      ]  
    };  
  }  
}
```

React Fundamental's

```
        }
    ]
}
}

render() {
    return (
        <div>
            <StudentName/>
            <ul>
                {this.state.data.map((item) => <List data = {item} />)}
            </ul>
        </div>
    );
}

class StudentName extends React.Component {
    render() {
        return (
            <div>
                <h1>Student Name Detail</h1>
            </div>
        )
    }
}
```

```
);

}

}

class List extends React.Component {

render() {

return (

<ul>

<li>{this.props.data.name}</li>

</ul>
);

}

export default App;
```



Functional Components

A Function component also returns HTML, and behaves much the same way as a Class component, but Function components can be written using much less code, are easier to understand, and will be preferred in this tutorial.

Example

```
function Car() {  
  return <h2>I am a Car!</h2>;  
}  
  
function Garage() {  
  return (  
    <>  
      <h1>Who lives in my Garage?</h1>  
      <Car />  
    </>  
  );  
}  
  
const root =  
  ReactDOM.createRoot(document.getElementById('root'));  
root.render(<Garage />)
```

3.10. Exports and imports

we can create modules in JavaScript. In a module, there can be classes, functions, variables, and objects as well. To make all these available in another file, we can use **export** and **import**. The **export** and **import** are the keywords used for exporting and importing one or more members in a module.

3.10.1. Export

You can export a variable using the **export** keyword in front of that variable declaration. You can also export a function and a class by doing the same.

Syntax for variable

```
export let variable_name;
```

Syntax for function

```
export function function_name() {  
    // Statements  
}
```

React Fundamental's

Syntax for class

```
export class Class_Name {  
    constructor() {  
        // Statements  
    }  
}
```

Example 1

```
export let num_set = [1, 2, 3, 4, 5];  
  
export default function hello() {  
    console.log("Hello World!");  
}  
  
export class Greeting {  
    constructor(name) {  
        this.greeting = "Hello, " + name;  
    }  
}
```

Example 2:

```
let num_set = [1, 2, 3, 4, 5];

export default function hello() {
    console.log("Hello World!");
}

class Greeting {
    constructor(name) {
        this.greeting = "Hello, " + name;
    }
}
export { num_set, Greeting as Greet };
```

3.10.2. **Import:**

You can import a variable using import keyword. You can specify one of all the members that you want to import from a JavaScript file.

```
import member_to_import from "path_to_js_file";

// You can also use an alias while importing a member.
import Greeting as Greet from "./export.js";

// If you want to import all the members but don't
// want to Specify them all then you can do that using
// a '*' star symbol.
```

```
import * as exp from "./export.js";
```

3.11. Props

Props stand for "Properties." They are read-only components. It is an object which stores the value of attributes of a tag and work similar to the HTML attributes. It gives a way to pass data from one component to other components. It is similar to function arguments. Props are passed to the component in the same way as arguments passed in a function.

Props are immutable so we cannot modify the props from inside the component. Inside the components, we can add attributes called props. These attributes are available in the component as this.props and can be used to render dynamic data in our render method.

```
const myElement = <Car brand="Ford" />;
```

```
-----  
function Car(props) {  
  return <h2>I am a { props.brand }!</h2>;  
}
```

Props are also how you pass data from one component to another, as parameters.

Example:

```
function Car(props) {  
  return <h2>I am a { props.brand }!</h2>;  
}  
  
function Garage() {  
  const carName = "Ford";  
  return (  
    <>  
      <h1>Who lives in my garage?</h1>  
      <Car brand={ carName } />  
    </>  
  );  
}
```

3.12. State Intro

The state is an updatable structure that is used to contain data or information about the component. The state in a component can change over time. The change in state over time can happen as a response to user action or system event. A component with the state is known as stateful components. It is the heart of the react component which determines the behavior of the component and

React Fundamental's

how it will render. They are also responsible for making a component dynamic and interactive.

```
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {brand: "Ford"};
  }
  render() {
    return (
      <div>
        <h1>My Car</h1>
      </div>
    );
  }
}
```

Changing the state Object

To change a value in the state object, use the `this.setState()` method.

When a value in the state object changes, the component will re-render, meaning that the output will change according to the new value(s).

Example:

```
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
      color: "red",
    };
  }
  // ...
}
```

```
    year: 1964
  );
}
changeColor = () => {
  this.setState({color: "blue"});
}
render() {
  return (
    <div>
      <h1>My {this.state.brand}</h1>
      <p>
        It is a {this.state.color}
        {this.state.model}
        from {this.state.year}.
      </p>
      <button
        type="button"
        onClick={this.changeColor}
      >Change color</button>
    </div>
  );
}
```

Functional component State Usage:

The React `useState` Hook allows us to track state in a function component.

To use the `useState` Hook, we first need to `import` it into our component.

```
import { useState } from "react";
```

Initialize useState

We initialize our state by calling `useState` in our function component.

useState accepts an initial state and returns two values:

- The current state.
- A function that updates the state.

```
import { useState } from "react";
import ReactDOM from "react-dom/client";

function FavoriteColor() {
  const [color, setColor] = useState("red");

  return <h1>My favorite color is {color}</h1>
}
```

React Fundamental's

Difference between State and Props

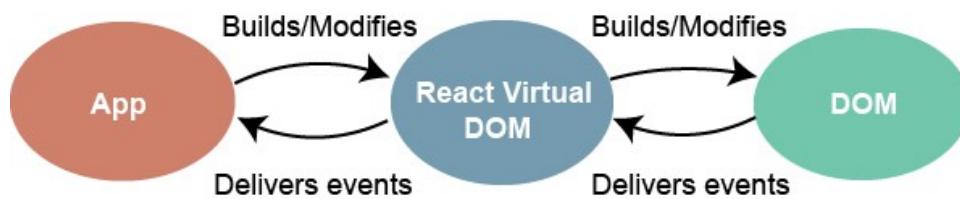
Props	State
Props are read-only.	State changes can be asynchronous.
Props are immutable.	State is mutable.
Props allow you to pass data from one component to other components as an argument.	State holds information about the components.
Props can be accessed by the child component.	State cannot be accessed by child components.
Props are used to communicate between components. Stateless component can have Props. Props make components reusable. Props are external and controlled by whatever renders the component.	States can be used for rendering dynamic changes with the component. Stateless components cannot have State. State cannot make components reusable. The State is internal and controlled by the React Component itself.

3.13. Events

An event is an action that could be triggered as a result of the user action or system generated event. For example, a mouse click, loading of a web page, pressing a key, window resizes, and other interactions are called events.

React has its own event handling system which is very similar to handling events on DOM elements. The react event handling system is known as Synthetic Events. The synthetic event is a cross-browser wrapper of the browser's native event.

1. React events are named as **camelCase** instead of **lowercase**.
2. With JSX, a function is passed as the **event handler** instead of a **string**



Handling events with react have some syntactic differences from handling events on DOM. These are:

React:

```
<button onClick={shoot}>Take the Shot!</button>
```

HTML:

```
<button onclick="shoot()">Take the Shot!</button>
```

Example:

```
function Football() {  
  const shoot = () => {  
    alert("Great Shot!");  
  }  
  
  return (  
    <button onClick={shoot}>Take the shot!</button>  
  );  
}  
  
const root =  
ReactDOM.createRoot(document.getElementById('root'));  
root.render(<Football />);
```

Passing Arguments

To pass an argument to an event handler, use an arrow function.

Example:

```
function Football() {  
  const shoot = (a) => {  
    alert(a);  
  }  
  
  return (  
    <button onClick={() => shoot("Goal!")}>Take the  
shot!</button>  
  );  
}
```

```
function Football() {  
  const shoot = (a, b) => {  
    alert(b.type);  
    /*  
    'b' represents the React event that triggered the function,  
    in this case the 'click' event  
    */  
  }  
  
  return (  
    <button onClick={(event) => shoot("Goal!", event)}>Take  
the shot!</button>  
  );  
}
```

3.14. Conditional rendering

In React, we can create multiple components which encapsulate behavior that we need. After that, we can render them depending on some conditions or the state of our application. In other words, based on one or several conditions, a component decides which elements it will return. In React, conditional rendering works the same way as the conditions work in JavaScript. We use JavaScript operators to create elements representing the current state, and then React Component update the UI to match them.

- if
- ternary operator
- logical && operator
- switch case operator
- Conditional Rendering with enums

```
function Goal(props) {  
  const isGoal = props.isGoal;  
  if (isGoal) {  
    return <MadeGoal/>;  
  }  
  return <MissedGoal/>;  
}
```

```
function Garage(props) {  
  const cars = props.cars;  
  return (
```

React Fundamental's

```
<>
<h1>Garage</h1>
{cars.length > 0 &&
<h2>
  You have {cars.length} cars in your garage.
</h2>
}
</>
);
}
```

```
function Garage(props) {
  const cars = props.cars;
  return (
    <>
      <h1>Garage</h1>
      {cars.length > 0 &&
        <h2>
          You have {cars.length} cars in your garage.
        </h2>
      }
    </>
  );
}
```

3.15. List rendering

In React, you will render lists with some type of loop.

The JavaScript `map()` array method is generally the preferred method.

```
function Car(props) {  
  return <li>I am a { props.brand }</li>;  
}  
  
function Garage() {  
  const cars = ['Ford', 'BMW', 'Audi'];  
  return (  
    <>  
      <h1>Who lives in my garage?</h1>  
      <ul>  
        {cars.map((car) => <Car brand={car} />)}  
      </ul>  
    </>  
  );  
}
```

```
function Car(props) {  
  return <li>I am a { props.brand }</li>;  
}  
  
function Garage() {  
  const cars = ['Ford', 'BMW', 'Audi'];  
  return (  
    <>  
      <h1>Who lives in my garage?</h1>
```

```
<ul>
  {cars.map((car) => <Car brand={car} />)}
</ul>
</>
);
```

3.16. Basic CSS styling

There are many ways to style React with CSS, this tutorial will take a closer look at three common ways:

- Inline styling
- CSS stylesheets
- CSS Modules

Examples:

```
const Header = () => {
  return (
    <>
      <h1 style={{color: "red"}}>Hello Style!</h1>
      <p>Add a little style!</p>
    </>
  );
}
```

```
const Header = () => {
  const myStyle = {
    color: "white",
    backgroundColor: "DodgerBlue",
    padding: "10px",
    fontFamily: "Sans-Serif"
  };
  return (
    <>
      <h1 style={myStyle}>Hello Style!</h1>
      <p>Add a little style!</p>
    </>
  );
}
```



3.17. Array Functions

Not really next-gen JavaScript, but also important: JavaScript array functions like `map()` , `filter()` , `reduce()` etc.

You'll see me use them quite a bit since a lot of React concepts rely on working with arrays (in immutable ways).

The following page gives a good overview over the various methods you can use on the array prototype - feel free to click through them and refresh your knowledge as required:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

React Fundamental's

Particularly important in this course are:

Map:

The map() method creates a new array populated with the results of calling a provided function on every element in the calling array.

Syntax:

```
// Arrow function  
  
map((element) => { /* ... */ })  
  
map((element, index) => { /* ... */ })  
  
map((element, index, array) => { /* ... */ })  
  
// Callback function  
  
map(callbackFn)  
  
map(callbackFn, thisArg)  
  
  
// Inline callback function  
  
map(function(element) { /* ... */ })  
  
map(function(element, index) { /* ... */ })  
  
map(function(element, index, array){ /* ... */ })  
  
map(function(element, index, array) { /* ... */ }, thisArg)
```

map calls a provided callbackFn function once for each element in an array, in order, and constructs a new array from the results. callbackFn is invoked only for indexes of the array which have assigned values (including undefined).

Example:

```
const array1 = [1, 4, 9, 16];
// pass a function to map
const map1 = array1.map(x => x * 2);
console.log(map1);
// expected output: Array [2, 8, 18, 32]
```

find():

The find() method returns the first element in the provided array that satisfies the provided testing function. If no values satisfy the testing function, undefined is returned.

```
const array1 = [5, 12, 8, 130, 44];
const found = array1.find(element => element > 10);
console.log(found);
```

Syntax

```
find((element) => { /* ... */ } )
find((element, index) => { /* ... */ } )
find((element, index, array) => { /* ... */ } )

// Callback function
find(callbackFn)
find(callbackFn, thisArg)

// Inline callback function
find(function(element) { /* ... */ })
find(function(element, index) { /* ... */ })
find(function(element, index, array){ /* ... */ })
find(function(element, index, array) { /* ... */ }, thisArg)
```



findIndex()

The `findIndex()` method returns the index of the first element in an array that satisfies the provided testing function. If no elements satisfy the testing function, -1 is returned.

Syntax:

```
findIndex((element) => { /* ... */ } )
findIndex((element, index) => { /* ... */ } )
findIndex((element, index, array) => { /* ... */ } )

// Callback function
```

```
findIndex(callbackFn)
findIndex(callbackFn, thisArg)

// Inline callback function
findIndex(function(element) { /* ... */ })
findIndex(function(element, index) { /* ... */ })
findIndex(function(element, index, array){ /* ... */ })
findIndex(function(element, index, array) { /* ... */ }, thisArg)
```

Example

```
const array1 = [5, 12, 8, 130, 44];

const isLargeNumber = (element) => element > 13;

console.log(array1.findIndex(isLargeNumber));

// expected output: 3
```

filter()

The filter() method creates a shallow copy of a portion of a given array, filtered down to just the elements from the given array that pass the test implemented by the provided function.

```
// Arrow function
filter((element) => { /* ... */ })
filter((element, index) => { /* ... */ })
filter((element, index, array) => { /* ... */ })

// Callback function
```

```
filter(callbackFn)
filter(callbackFn, thisArg)

// Inline callback function
filter(function(element) { /* ... */ })
filter(function(element, index) { /* ... */ })
filter(function(element, index, array){ /* ... */ })
filter(function(element, index, array) { /* ... */ }, thisArg)
```

Example:

```
const words = ['spray', 'limit', 'elite', 'exuberant', 'destruction',
'present'];

const result = words.filter(word => word.length > 6);

console.log(result);

// expected output: Array ["exuberant", "destruction", "present"]
```

concat()

The concat() method is used to merge two or more arrays. This method does not change the existing arrays, but instead returns a new array.

```
concat()
concat(value0)
concat(value0, value1)
concat(value0, value1, /* ... */ valueN)
```

```
const array1 = ['a', 'b', 'c'];
const array2 = ['d', 'e', 'f'];
const array3 = array1.concat(array2);
console.log(array3);
// expected output: Array ["a", "b", "c", "d", "e", "f"]
```

Splice()

The `splice()` method changes the contents of an array by removing or replacing existing elements and/or adding new elements in place. To access part of an array without modifying it, see

Syntax:

```
splice(start)
splice(start, deleteCount)
splice(start, deleteCount, item1)
splice(start, deleteCount, item1, item2, itemN)
```

Example:

React Fundamental's

```
const months = ['Jan', 'March', 'April', 'June'];

months.splice(1, 0, 'Feb');

// inserts at index 1

console.log(months);

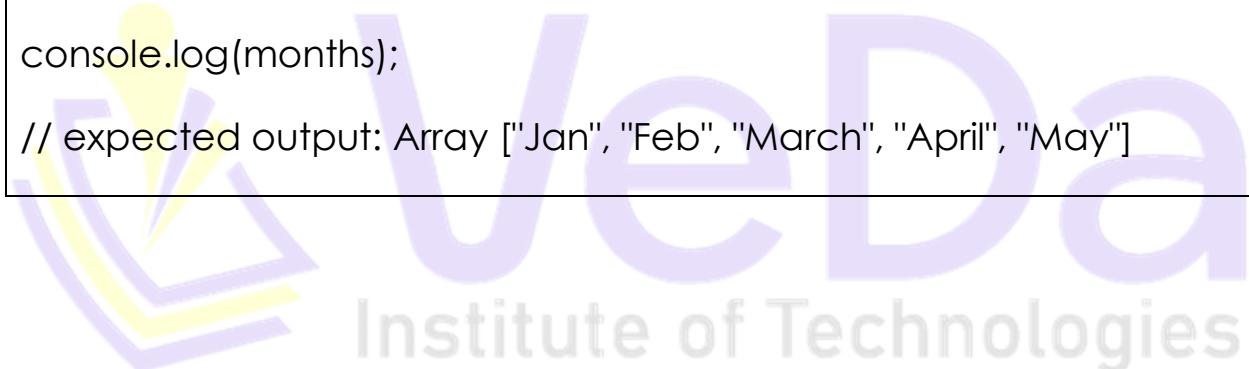
// expected output: Array ["Jan", "Feb", "March", "April", "June"]
```

```
months.splice(4, 1, 'May');

// replaces 1 element at index 4

console.log(months);

// expected output: Array ["Jan", "Feb", "March", "April", "May"]
```



ReactJS Advanced Material

4 Advanced

4.1 State management

- 4.1.1 Working with state
- 4.1.2 useState hook usage
- 4.1.3 Working with multiple states
- 4.1.4 Child to parent communication

4.2 Life cycles

- 4.2.1 Component did mount
- 4.2.2 Component will unmount

4.3 React Hooks

- 4.3.1 Introducing hook
- 4.3.2 State hook
- 4.3.3 Effect hook
- 4.3.4 Hooks rules

4.4 Routing

- 4.4.1 Install react-router
- 4.4.2 Routing intro
- 4.4.3 Link usage
- 4.4.4 Parameters
- 4.4.5 Custom navigation usage
- 4.4.6 Nested routing

4.5 Forms

4.6 HTTP requests

4.7 Redux

- 4.7.1 Store
- 4.7.2 Action
- 4.7.3 Reducers
- 4.7.4 Redux api
- 4.7.5 Provider component

ReactJS Advanced Material

4.1 State management

4.1.1 Working with state

Introduction to State in React:

State allows us to manage changing data in an application. It's defined as an object where we define key-value pairs specifying various data we want to track in the application

In React, all the code we write is defined inside a component.

There are mainly two ways of creating a component in React:

- class-based component
- functional component

First, we will discuss class-based component

When creating a React component, the component's name must start with an upper-case letter.

The component also requires a `render()` method, this method returns HTML.

Import React from 'react';

Class Car extends React.Component{

 render (){

 Return(<h2>Hai , i am a Car! </h2>

 }

 export default Car;

ReactJS Advanced Material

Component Constructor

If there is a `constructor ()` function in your component, this function will be called when the component gets initiated.

The constructor function is where you initiate the component's properties.

In React, component properties should be kept in an object called `state`.

You will learn more about `state` later in this tutorial.

Import React from 'react';

```
class Car extends React.Component {  
  constructor () {  
    super ();  
    this.state = { color : "red" };  
  }  
  render () {  
    return <h2>I am a Car! </h2>;  
  }  
}  
export default Car;
```

Props

Another way of handling component properties is by using `props`.

Props are like function arguments, and you send them into the component as attributes.

```
class Car extends React.Component {  
  render () {
```

ReactJS Advanced Material

```
return
(<h2> I am a { props.brand }! </h2>)
}
export default Car;

class Garage extends React.Component{

render () {
return (>

Who lives in my garage?
<Car brand=" Ford"/>

</>
);
}

export default Garage;
```

React Class Component State

React Class components have a built-in **state** object.

The **state** object is where you store property values that belongs to the component.

When the **state** object changes, the component re-renders.

the **state** object anywhere in the component by using the **this.state.propertyname** syntax: **this.state.brand**

```
class Car extends React.Component {

constructor(props) {
super(props);

this.state = {brand: "Ford"}; //Creating the state object
```

ReactJS Advanced Material

```
}

render() {
  return (
    <div>
      <h1>My Car brand {this.state.brand}</h1>
    </div>
  );
}

export default Car;
```

Changing the **state** Object

To change a value in the state object, use the `this.setState()` method.

Example:

Add a button with an `onClick` event that will change the color property:

```
import React from "react";
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
      color: "red",
      year: 1964
    };
  }
  changeColor() {
    const newColor = this.state.color === "red" ? "blue" : "red";
    this.setState({ color: newColor });
  }
  render() {
    return (
      <div>
        <h1>My Car brand {this.state.brand}</h1>
        <p>Model: {this.state.model}</p>
        <p>Color: {this.state.color}</p>
        <p>Year: {this.state.year}</p>
        <button onClick={this.changeColor}>Change Color</button>
      </div>
    );
  }
}
```

ReactJS Advanced Material

```
};

}

changeColor = () => {
    this.setState({color: "blue"});
}

render() {
    return (
        <div>
            <h1>My {this.state.brand}</h1>
            <p>
                It is a {this.state.color}{" "}
                {this.state.model}{" "}
                from {this.state.year}.
            </p>
            <button
                type="button"
                onClick={this.changeColor}
            >Change color</button>
        </div>
    );
}

export default Car
```

4.1.2 useState hook usage

ReactJS Advanced Material

What is a React Hook?

A hook is a special function that lets you "hook into" various React features. Imagine a function that returns an array with two values:

To use the useState Hook, we first need to import it into our component.

At the top of your component, import the useState Hook.

```
import { useState } from "react";  
const [name, setName] = useState("React")
```

- **The first value:** a variable with the state.
- **The second value:** a variable with an handler (a function to change the current state).

Example:

```
import React,{useState} from "react";  
function FavoriteColor() {  
  const [color, setColor] = useState("red");  
  
  return <h1>My favorite color is {color}!</h1>  
}  
  
export default FavoriteColor;
```

- To update our state, we use our state updater function.

Example

```
import React,{useState} from "react";  
function FavoriteColor() {  
  const [color, setColor] = useState("red");  
  
  return <>My favorite color is {color}!
```

ReactJS Advanced Material

```
<br/>
<button onClick={()=>setColor("blue")}>Change Color</button>
</>
}
export default FavoriteColor;
```

What Can State Hold

The `useState` Hook can be used to keep track of strings, numbers, booleans, arrays, objects, and any combination of these!

We could create multiple state Hooks to track individual values.

```
function FavoriteColor() {
  const [brand, setBrand] = useState("Ford");
  const [model, setModel] = useState("Mustang");
  const [year, setYear] = useState("1964");
  const [color, setColor] = useState("red");
  return (
    <>
      <h1>My {brand}</h1>
      <p>
        It is a {color} {model} from {year}.
      </p>
    </>
  )
}

export default FavoriteColor;

import React,{useState} from "react";

function FavoriteColor() {
  const [car, setCar] = useState({
    brand: "Ford",
  })
```

ReactJS Advanced Material

```
model: "Mustang",
year: "1964",
color: "red"

});

const updateColor = () => {
  setCar(previousState => {
    return { ...previousState, color: "blue" }
  });
}

return (
  <>
  <h1>My {car.brand}</h1>
  <p>
    It is a {car.color} {car.model} from {car.year}.
  </p>
  <button
    type="button"
    onClick={updateColor}
    >Blue</button>
  </>
)
}

export default FavoriteColor;
```

ReactJS Advanced Material

Child to parent communication:

To pass data from child to parent component in React:

- Pass a function as a prop to the **Child** component.
- Call the function in the **Child** component and pass the data as arguments.
- Access the data in the function in the **Parent**.

```
import React,{useState} from "react";

export default function Parent() {
  const [count, setCount] = useState(0);

  const handleClick = (num) => {
    // ↪ take parameter passed from Child component
    setCount(current => current + num);
  };

  return (
    <div>
      <Child handleClick={handleClick} />

      <h2>Count: {count}</h2>
    </div>
  );
}

function Child({handleClick}) {
  return (
    <div>
```

ReactJS Advanced Material

```
<button onClick={event => handleClick(100)}>Click</button>

</div>
);

}

export default Child;
```

Working with multiple states

The state object of a component may contain multiple attributes and React allows to use `setState()` function to update only a subset of those attributes as well as using multiple `setState()` methods to update each attribute value independently.

```
this.state = {
name: "Javascript",
age: 12
};

This.setState({...this.state, name:"React js", age:30});
```

Child to parent communication

To pass data from child to parent component in React:

- Pass a function as a prop to the **Child** component.
- Call the function in the **Child** component and pass the data as arguments.
- Access the data in the function in the **Parent**.

Parent.js

```
import {useState} from 'react';

import Child from './child1';

function Parent() {

  const [name, setName] = useState("Java script");

  return (
    <div>
      <Child name={name} />
    </div>
  );
}
```

ReactJS Advanced Material

```
}

export default Parent
```

Child.js

```
function Child(props) {

  debugger

  return (
    <div>
      {props.name}
    </div>
  );
}

export default Child;
```

4.2 Life cycles

Each component in React has a lifecycle which you can monitor and manipulate during its these are main phases.

Component did mount:(Mounting)

Mounting means putting elements into the DOM.

React has four built-in methods that gets called, in this order, when mounting a component:

constructor(): The constructor is a method used to initialize an object's state in a class. It automatically called during the creation of an object in a class.

getDerivedStateFromProps(): method is called right before the render method:

render(): method is required, and is the method that actually outputs the HTML to the DOM.

componentDidMount(): method is called after the component is rendered.

Examples(**constructor**)

```
import React from "react";

class Header extends React.Component {
```

ReactJS Advanced Material

```
constructor(props) {  
    super(props);  
    this.state = {favoritecolor: "red"};  
}  
  
//render():this method that actually outputs the HTML to the DOM  
  
render() {  
    return (  
        <h1>My Favorite Color is {this.state.favoritecolor}</h1>  
    );  
}  
  
}  
  
export default Header;
```

componentDidMount

The `componentDidMount()` method is called after the component is rendered.

```
import React from 'react';  
  
class Header extends React.Component {  
    constructor(props) {  
        super(props);  
        // Initializing the state  
        this.state = { color: 'lightgreen' };  
    }  
  
    componentDidMount() {  
        // Changing the state after 2 sec  
        // from the time when the component  
        // is rendered  
        setTimeout(() => {  
            this.setState({ color: 'red' });  
        }, 4000);  
    }  
}
```

ReactJS Advanced Material

```
render() {  
    return (  
        <div>  
            <p  
                style={{  
                    color: this.state.color,  
                }}>  
                >  
                GeeksForGeeks  
            </p>  
  
        </div>  
    );  
}  
export default Header;
```

Component will unmount

The componentWillUnmount() method allows us to execute the React code when the component gets destroyed or unmounted from the DOM (Document Object Model). This method is called during the Unmounting phase of the React Life-cycle i.e before the component gets unmounted.

React Hooks:

Hooks allow function components to have access to state and other React features. Because of this, class components are generally no longer needed.

Hooks generally replace class components, there are no plans to remove classes from React.

ReactJS Advanced Material

Introducing hook:

- You must `import Hooks from react`.
- Here we are using the `useState` Hook to keep track of the application state.
- State generally refers to application data or properties that need to be tracked.

Hook Rules: There are 3 rules for hooks:

- Hooks can only be called inside React function components.
- Hooks can only be called at the top level of a component.
- Hooks cannot be conditional

Note: Hooks will not work in React class components.

Example:

```
import React, { useState } from "react";

function FavoriteColor() {
  const [color, setColor] = useState("red");

  return (<>
    <h1>My favorite color is {color}!</h1>
    <button
      type="button"
      onClick={() => setColor("blue")}
    >Blue</button>
  </>)
}

export default FavoriteColor;
```

State hook:

1. The React `useState` Hook allows us to track state in a function component.
2. State generally refers to data or properties that need to be tracking in an application.
3. To use the `useState` Hook, we first need to `import` it into our component.
4. At the top of your component, `import` the `useState` Hook.

ReactJS Advanced Material

Initialize useState

We initialize our state by calling `useState` in our function component.

`useState` accepts an initial state and returns two values:

- The current state.
- A function that updates the state.

```
import { useState } from "react";
```

```
function FavoriteColor() {
  const [color, setColor] = useState("");
}
```

- The first value, `color`, is our current state.
- The second value, `setColor`, is the function that is used to update our state.

```
import { useState } from "react";
```

```
function FavoriteColor() {
  const [color, setColor] = useState(" red ");
  return <h1>My favorite color is {color}!</h1>
}
export default FavoriteColor;
```

Update State

To update our state, we use our state updater function.

```
function FavoriteColor() {
  const [color, setColor] = useState("red");

  return (
    <>
      <h1>My favorite color is {color}!</h1>
      <button
        type="button"
        onClick={() => setColor("blue")}
      >Blue</button>
    </>
  )
}

export default FavoriteColor;
```

ReactJS Advanced Material

useState Hook can be used to keep track of strings, numbers, booleans, arrays, objects, and any combination of these.

For example:

```
import { useState } from "react";

function Car() {
  const [brand, setBrand] = useState("Ford");
  const [model, setModel] = useState("Mustang");
  const [year, setYear] = useState("1964");
  const [color, setColor] = useState("red");

  return (
    <>
      <h1>My {brand}</h1>
      <p>
        It is a {color} {model} from {year}.
      </p>
    </>
  )
};

export default Car;
```

Effect hook

Some examples of side effects are: fetching data, directly updating the DOM, and timers.

useEffect accepts two arguments. The second argument is optional.

```
useEffect(<function>, <dependency>)
```

No dependency passed:

```
useEffect(() => {
  //Runs on every render
});
```

An empty array:

```
useEffect(() => {
  //Runs only on the first render
}, []);
```

Props or state values:

```
useEffect(() => {
  //Runs on the first render
```

ReactJS Advanced Material

```
//And any time any dependency value changes
}, [prop, state]);
```

Example:

```
import { useState, useEffect } from "react";

function Timer() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    setTimeout(() => {
      setCount((count) => count + 1);
    }, 1000);
  });

  return <h1>I've rendered {count} times!</h1>;
}

export default Timer;
```

Hook Rules

There are 3 rules for hooks:

- Hooks can only be called inside React function components.
- Hooks can only be called at the top level of a component.
- Hooks cannot be conditional

Note: Hooks will not work in React class components.

React Routing

1. Install react-router-dom npm package:

React is a JavaScript library, not a framework hence many of the functionalities including routing aren't built-in and need to be added through npm packages.

We will install the “**react-router-dom**” package to add routing functionality to the project. With the terminal command below:

```
npm i react-router-dom
```

```
// For projects configured with yarn
```

ReactJS Advanced Material

yarn add react-router-dom

Once the package is successfully installed, we can verify the dependencies from the **package.json** of the project.

```
"main": "src/index.js",
"dependencies": {
  "react": "17.0.2",
  "react-dom": "17.0.2",
  "react-router-dom": "5.3.0",
  "react-scripts": "4.0.0"
},
```

Routing intro

All present-day websites are single page websites which are actually pretend to be multiple pages

Routing is a mechanism to redirect the users from one page to another page, and provides the navigation from one page to another page.

Begin with installation

To install the React routing, use NPM:

```
npm install react-router-dom
```

Using the react-router-dom package, you will also need to import BrowserRouter, Route, and Switch. Here is how:-

```
import React, { Component } from 'react';
import { BrowserRouter, Route, Switch } from 'react-router-dom';
```

Setting the app

So, any application that you need to use with React first needs to be set up with the React Router. All the features that you need to render in your SPA will be put in the <BrowserRouter> element.

ReactJS Advanced Material

begin by wrapping your app in it. <BrowserRouter> element or component will do all the logic that helps display the various components that will be added into your app.

```
index.js
ReactDOM.render(
  <BrowserRouter>
    <App />
  </BrowserRouter>,
  document.getElementById('root')
)
```

```
App.js
```

```
function App() {
  return (
    <main>
      <Switch>
        </Switch>
      </main>
  )
}
```

Now add the Route tags

You will here just need to add the name of the component you will want to redirect to by using the component attribute and a path attribute. Here 'home' is the component. The <Route> tags are placed in the <Switch> tags.

```
<Route path='/' component={Home} />
```

Exact tag

The exact tag is an easy solution to deal with the issues when the '/' component might redirect to any of the URL components that deal with '/' and not lead to the specific route asked.

```
function App() {
  return (
    <main>
      <Switch>
```

The logo for Veda Institute of Technologies features the word "Veda" in a large, stylized purple font. Below it, "Institute of Technologies" is written in a smaller, gray font. To the left of the text, there is a graphic element consisting of overlapping yellow and purple curved shapes.

ReactJS Advanced Material

```
<Route path="/" component={Home} exact />
<Route path="/about" component={About} />
<Route path="/contact" component={Contact} />
</Switch>
</main>
)
}
```

Add the components

All you need to do now is add the components to the app

Here is an example:

```
import Home from './components/Home';
import About from './components/About';
import Contact from './components/Contact';
```

Link usage

To make your site accessible by using clickable links and not just typing in the URL manually, use a code to set up a Navbar for navigation and use the Link component. Here, add the components to the app.

NavBar.js

```
function Navbar() {
  return (
    <div>
      <Link to="/">Home </Link>
      <Link to="/about">About Us </Link>
      <Link to="/contact">Contact Now </Link>
    </div>
  );
}

export default Navbar;
```

Example for routing:

ReactJS Advanced Material

App.js

```
import {BrowserRouter, Switch, Route, Link} from "react-router-dom";
import Home from './route/home';
import About from './route/about';
import Contact from './route/contact';
import NavBar from './route/nav';

function App() {
  return (
    <div className="App">
      <NavBar />
      <BrowserRouter>
        <Switch>
          <Route path="/" component={Home} exact/>
          <Route path="/about" component={About}>/>
          <Route path="/contact" component={Contact}>/>
        </Switch>
      </BrowserRouter>
    </div>
  )
}

export default App;
```

NavBar.js

```
import {Link} from "react-router-dom";
const NavBar=() =>{
  return (
    <div>
      <Link to="/">Home </Link>
      <Link to="/about">About Us </Link>
      <Link to="/contact">Contact Now </Link>
    </div>
  )
}
```

ReactJS Advanced Material

```
</div>  
);  
};  
  
export default NavBar;
```

Home.js

```
const Home=()=>{  
  
return(<>  
  
<div style={{marginBottom:"10px",  
width: "100px",  
height: "30px",  
background: "saddlebrown",  
color: "greenyellow",  
marginTop: "78px",  
marginLeft: "600px"}>  
  
Home Page  
</div>  
</>)  
}  
  
export default Home;
```

About.js

```
const About=()=>{  
  
return(<>  
  
<div style={{marginBottom:"10px",  
width: "100px",  
height: "30px",  
background: "saddlebrown",  
color: "greenyellow",  
marginTop: "78px",  
marginLeft: "600px"}>
```


ReactJS Advanced Material

```
About Page</div></>)  
}  
  
export default About;  
  
Contact.js  
  
const Contact=()=>{  
  return(<>  
    <div style={{marginBottom:"10px",  
width: "100px",  
height: "30px",  
background: "saddlebrown",  
color: "greenyellow",  
marginTop: "78px",  
marginLeft: "600px"}}>  
      Contact Page  
    </div>  
  </>)  
}  
  
export default Contact;
```

Parameters

What is parameter in React?

React parameters are **used in React routing**, where we have parameters we need to access in the route.

For example, if we had a route such as <Route path="/:id" /> we could access that particular string or value in the route by calling the useParams hook. let { id } = useParams();

App.js

```
import {BrowserRouter,Switch,Route,Link} from "react-router-dom";  
  
import Home from './route/home';
```

ReactJS Advanced Material

```
import NavBar from './route/nav';

function App() {
  return (
    <div className="App">
      <NavBar />
      <BrowserRouter>
        <Switch>
          <Route path="/" component={Home} exact/>
        </Switch>
      </BrowserRouter>
    </div>
  )
}

export default App;

navBar.js
import {Link} from "react-router-dom";
const Navbar=() =>{
  return (
    <div>
      <Link to={`/home/${"Dhana"}`}>Home </Link>
    </div>
  );
}

export default Navbar;
```

home.js

```
import { useParams } from "react-router-dom";
const Home=()=>{
  let { name } = useParams();
```

ReactJS Advanced Material

```
return(<>
<div style={{marginBottom:"10px",
width: "200px",
height: "40px",
background: "saddlebrown",
color: "greenyellow",
marginTop: "78px",
marginLeft: "530px"}}>
    Welcome To Home Page {name}
</div>
</>)
}

export default Home;
```

Nested routing

In my own words, a nested route is **a region within a page layout that responds to route changes**. For example, in a single-page application, when navigating from one URL to another, you do not need to render the entire page, but only those regions within the page that are dependent on that URL change.

App.js

```
import {BrowserRouter,Switch,Route,Link} from "react-router-dom";
import Nested from './route/nested';
import Student from './route/student'

const App=() =>{
    return (
        <div>
            <Link to="/">Home </Link>
            <Link to="/studentDetails">StudentDetails </Link>
        </div>
    )
}

export default App;
```

ReactJS Advanced Material

```
</div>

< BrowserRouter>

<Switch>

<Route path="/" component={Home} exact/>

<Route path="/Student" component={Student}>/>

</Switch>

</BrowserRouter>

);

};

export default App;

students.js
```

```
import { Switch, Route, Link, useRouteMatch } from "react-router-dom";

import StudentDetails from './StudentDetails'

const Student=()=>{
    let { path, url } = useRouteMatch();
    return(<>
        <div style={{marginBottom:"10px",
        width: "250px",
        height: "100px",
        border: "2px solid black",
        color: "greenyellow",
        marginTop: "78px",
        marginLeft: "600px"}}>
            Welcome To Student Details Page
            <ul>
                <li>
                    <Link to={`${url}/Dhana/1`}>Dhana</Link>
                </li>
            </ul>
        </div>
    </>)
}
```

ReactJS Advanced Material

```
</li>
<li>
  <Link to={`${url}/lakshmi/2`}>lakshmi</Link>
</li>
<li>
  <Link to={`${url}/raj/3`}>raj</Link>
</li>
</ul>
<Switch>
  <Route exact path={path}>
    <h3>Please select a topic.</h3>
  </Route>
  <Route path={`${path}/:name/:id`}>
    <StudentDetails />
  </Route>
</Switch>
</div>
</>
}
export default Student;
studentDetails.js
import { useParams } from "react-router-dom";
const StudentDetails =()=>{
  let { name,id } = useParams();
  return (
    <div>
      <h3>Student Name:{name}</h3>
      <h3> Student Id:{id}</h3>
    </div>
  )
}
```

ReactJS Advanced Material

```
);  
}  
  
export default StudentDetails;
```

Forms

React uses forms to allow users to interact with the web page.

Handling Forms

Handling forms is about how you handle the data when it changes value or gets submitted.

In HTML, form data is usually handled by the DOM.

In React, form data is usually handled by the components.

You can control changes by adding event handlers in the `onChange` attribute.

Example:

Use the `useState` Hook to manage the input:

```
import React,{useState} from "react";  
const MyForm=() =>{  
  const [name, setName] = useState("");  
  const handleSubmit = (event) => {  
    event.preventDefault();  
    alert(`The name you entered was: ${name}`)  
  }  
  return (  
    <>  
    <form onSubmit={handleSubmit}>  
      <label>Enter your Form Name:  
      <input  
        type="text"
```

ReactJS Advanced Material

```
value={name}  
placeholder={"enter name"}  
onChange={(e) => setName(e.target.value)}  
/>  
</label>  
  
<br/><br/>  
  
<input type="submit" />  
  
</form>  
  
</>  
)  
}  
  
export default MyForm;
```

HTTP requests

we are going to learn how to send and receive Http Responses in a React application.

To send or receive data, we don't need to use third-party packages, rather we can use the **fetch()** method which is now supported by all the modern browsers.

Sending GET request

<https://jsonplaceholder.typicode.com/todos/1>

" **Jsonplaceholder** is a fake API which is used to learn the process of sending requests."

```
import React, { useEffect, useState } from 'react';  
const App = () => {  
  const [data, setData] = useState(null);  
  const [fetchData, setFetch] = useState(false);  
  
  useEffect(() => {  
    if (fetchData) {  
      fetch('https://jsonplaceholder.typicode.com/todos/1')  
        .then((response) => response.json())  
        .then((data) => setData(data.title));  
    }  
  }, [fetchData]);  
  
  return (  
    <div>  
      <h1>{data}</h1>  
      <button onClick={() => setFetch(!fetchData)}>Get Data</button>  
    </div>  
  );  
};  
  
export default App;
```

ReactJS Advanced Material

```
        }
    }, [fetchData]);
return (
    <>
        <h1>{data}</h1>
        <button onClick={() => setFetch(true)}>Fetch Data</button>
    </>
);
};

export default App;
```

In the above example, we are sending the GET request to the **jsonplaceholder** and accessing the data which is going to be inserted in the state as soon as the response is received.

Sending POST request:

<https://jsonplaceholder.typicode.com/todos/1>

Jsonplaceholder is a fake API which is used to learn the process of sending requests.

```
import React, { useEffect, useState } from 'react';
import Input from './Input';
import './App.css';
const App = () => {
    const [data, setData] = useState(null);
    const [val, setVal] = useState('');
    const [fetchData, setFetch] = useState(false);

    useEffect(() => {
        if (fetchData) {
            const payload = {
                method: 'POST',
                body: JSON.stringify({ title: val }),
            };
            fetch('https://jsonplaceholder.typicode.com/posts', payload)
                .then((res) => res.json())
                .then((data) => setData(data.id));
        }
    }, [fetchData]);
    return (
        <>
            {data && <h1>Your data is saved with Id: {data}</h1>}
            <input
```

ReactJS Advanced Material

```
placeholder="Title of Post"
value={val}
onChange={(e) => setVal(e.target.value)}
/>
<button onClick={() => setFetch(true)}>Save Data</button>
</>
);
};

export default App;
```

In the above example, we are sending the POST request to the **jsonplaceholder** with the input field value in the body and displaying the response accordingly.

Sending PUT request:

'<https://jsonplaceholder.typicode.com/posts/1>'

Jsonplaceholder is a fake API which is used to learn the process of sending requests.

```
useEffect(() => {
  if (fetchData) {
    fetch('https://jsonplaceholder.typicode.com/posts/1', {
      method: 'PUT',
      body: JSON.stringify({
        id: 1,
        title: 'React js',
        body: 'react body',
        userId: 1,
      }),
      headers: {
        'Content-type': 'application/json; charset=UTF-8',
      },
    })
      .then((response) => response.json())
      .then((json) => console.log(json));
}
```

ReactJS Advanced Material

```
}, [fetchData]);
```

React Redux

React Redux is the official React UI bindings layer for Redux. It lets your React components read data from a Redux store, and dispatch actions to the store to update state.

Redux is a state management framework for use in JavaScript-based front-end web apps.

Installation

If you use npm:

- npm install react-redux

Or if you use Yarn:

- yarn add react-redux

Run the below command in your command prompt to install Redux dev-tools.

- npm install --save-dev redux-devtools

Store

A store is an immutable object tree in Redux.

A store is a state container which holds the application's state

Redux can have only a single store in your application.

Whenever a store is created in Redux, you need to specify the reducer.

Let us see how we can create a store using the **createStore** method from Redux.

need to import the createStore package from the Redux library.

Example:

```
import { createStore } from 'redux';
```

A createStore function can have three arguments. The following is the syntax –

ReactJS Advanced Material

```
createStore(reducer, [preloadedState], [enhancer])
```

reducer : A reducer is a function that returns the next state of app.

preloaded State : A preloadedState is an optional argument and is the initial state of your app.

Enhancer : An enhancer is also an optional argument. It will help you enhance store with third-party capabilities.

A store has three important methods as given below –

getState:

It helps you retrieve the current state of your Redux store.

```
store.getState()
```

dispatch:

It allows you to dispatch an action to change a state in your application.

```
store.dispatch({type: 'ITEMS_REQUEST'})
```

subscribe

It helps you register a callback that Redux store will call when an action has been dispatched. As soon as the Redux state has been updated, the view will re-render automatically.

```
store.subscribe(()=>{ console.log(store.getState());})
```

Note that subscribe function returns a function for unsubscribing the listener. To unsubscribe the listener, we can use the below code –

```
const unsubscribe = store.subscribe(()=>{console.log(store.getState());});  
unsubscribe();
```

Action:

ReactJS Advanced Material

Actions are the only source of information for the store as per Redux official documentation. It carries a payload of information from your application to store.

As discussed earlier, actions are plain JavaScript object that must have a type attribute to indicate the type of action performed. It tells us what had happened. Types should be defined as string constants in your application as given below –

```
const ITEMS_REQUEST = 'ITEMS_REQUEST';
```

Apart from this type attribute, the structure of an action object is totally up to the developer.

It is recommended to keep your action object as light as possible and pass only the necessary information.

To cause any change in the store, you need to dispatch an action first by using store.dispatch() function

```
{ type: GET_ORDER_STATUS , payload: {orderId,userId} }  
{ type: GET_WISHLIST_ITEMS, payload: userId }
```

Actions Creators

- Action creators are the functions that encapsulate the process of creation of an action object.
- These functions simply return a plain Js object which is an action.
- It promotes writing clean code and helps to achieve reusability.

```
const ITEMS_REQUEST = 'ITEMS_REQUEST' ;  
const ITEMS_REQUEST_SUCCESS = 'ITEMS_REQUEST_SUCCESS' ;  
export function itemsRequest(bool,startIndex,endIndex) {  
    let payload = {  
        isLoading: bool,  
        startIndex,  
        endIndex  
    }  
    return {  
        type: ITEMS_REQUEST,  
        payload  
    }  
}  
export function itemsRequestSuccess(bool) {
```

ReactJS Advanced Material

```
return {
  type: ITEMS_REQUEST_SUCCESS,
  isLoading: bool,
}
}

dispatch(itemsRequest(true,1, 20));
dispatch(itemsRequestSuccess(false));
```

Reducers

- Reducers are a pure function in Redux.
- Pure functions are predictable.
- Reducers are the only way to change states in Redux.
- It is the only place where you can write logic and calculations.
- Reducer function will accept the previous state of app and action being dispatched, calculate the next state and returns the new object.

syntax of a reducer:

```
(state,action) => newState
```

- Let us see below how to write its reducer

```
const initialState = {
  isLoading: false,
  items: []
};

const reducer = (state = initialState, action) => {
  switch (action.type) {
    case 'ITEMS_REQUEST':
      return Object.assign({}, state, {
        isLoading: action.payload.isLoading
      })
    case 'ITEMS_REQUEST_SUCCESS':
      return Object.assign({}, state, {
        items: state.items.concat(action.items),
        isLoading: action.isLoading
      })
    default:
      return state;
}
```

ReactJS Advanced Material

```
}

export default reducer;

orderStatusReducer.js

import { GET_ORDER_STATUS } from '../constants/appConstant';
export default function (state = {}, action) {
  switch(action.type) {
    case GET_ORDER_STATUS:
      return { ...state, orderStatusData: action.payload.orderStatus };
    default:
      return state;
  }
}

GetWishlistDataReducer.js
```

```
import { GET_WISHLIST_ITEMS } from '../constants/appConstant';
export default function (state = {}, action) {
  switch(action.type) {
    case GET_WISHLIST_ITEMS:
      return { ...state, wishlistData: action.payload.wishlistData };
    default:
      return state;
  }
}
```

```
Index.js
```

```
import { combineReducers } from 'redux';

import OrderStatusReducer from './orderStatusReducer';
import GetWishlistDataReducer from './getWishlistDataReducer';

const rootReducer = combineReducers ({
  orderStatusReducer: OrderStatusReducer,
  getWishlistDataReducer: GetWishlistDataReducer
});

export default rootReducer;

const store = createStore(rootReducer);
```

Veda Institute of Technologies

ReactJS Advanced Material

Redux api

Provider component

Provider is a **component** given to us to use from the **react-redux node package**.

We use Provider in order to pass **the store** as an attribute. By passing the store as an attribute in the Provider component, we are avoiding having to store **the store** as props.



```
import { Provider } from "react-redux"
ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider >,
  document.getElementById('root')
)
```

Node Js

Node.js is an open source server environment.

Node.js allows you to run JavaScript on the server.

Node.js has a set of built-in modules.

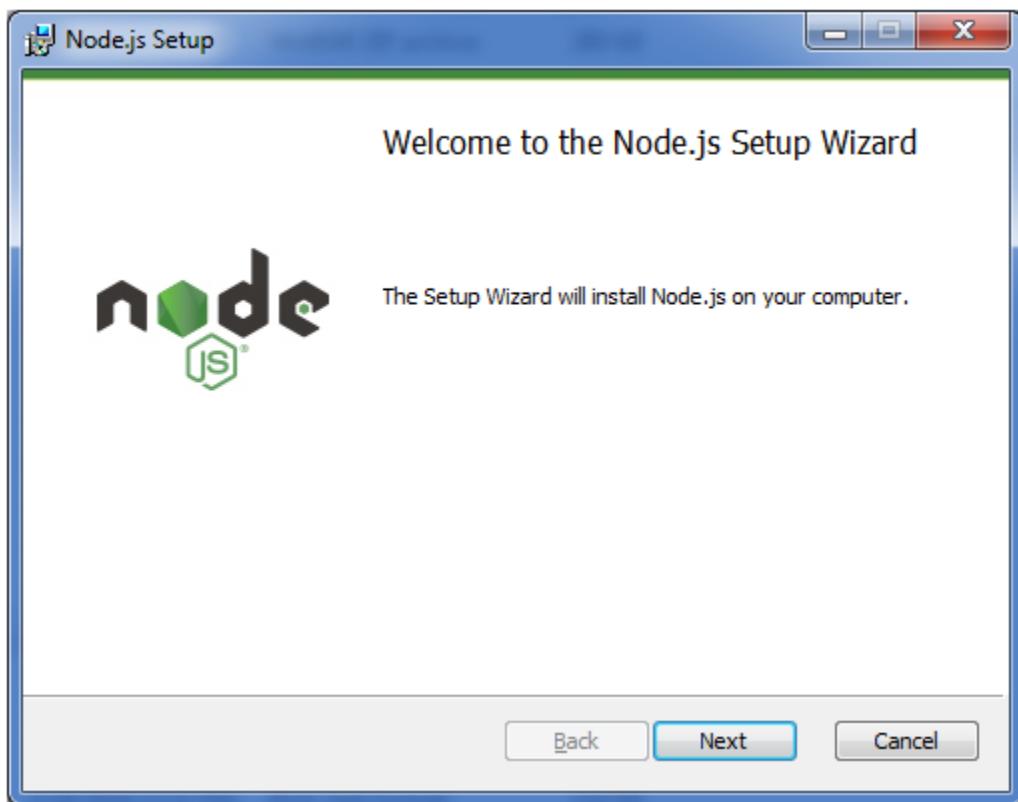
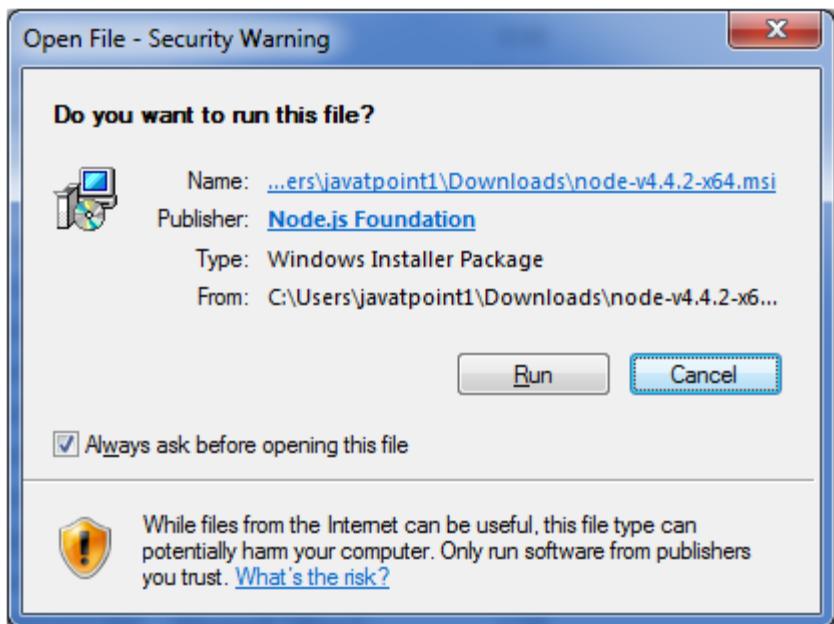
The Node.js Runtime:

The source code written in source file is simply JavaScript. It is interpreted and executed by the Node.js interpreter.

How to download Node.js:

You can download the latest version of Node.js installable archive file from <https://nodejs.org/en/>

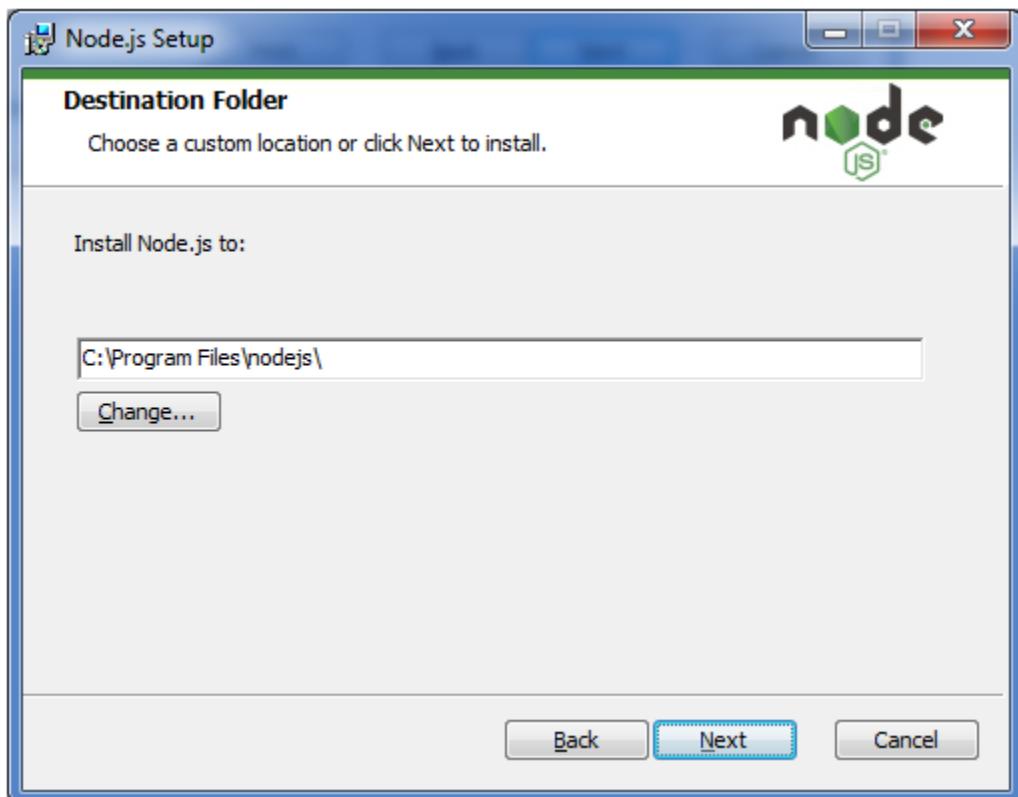


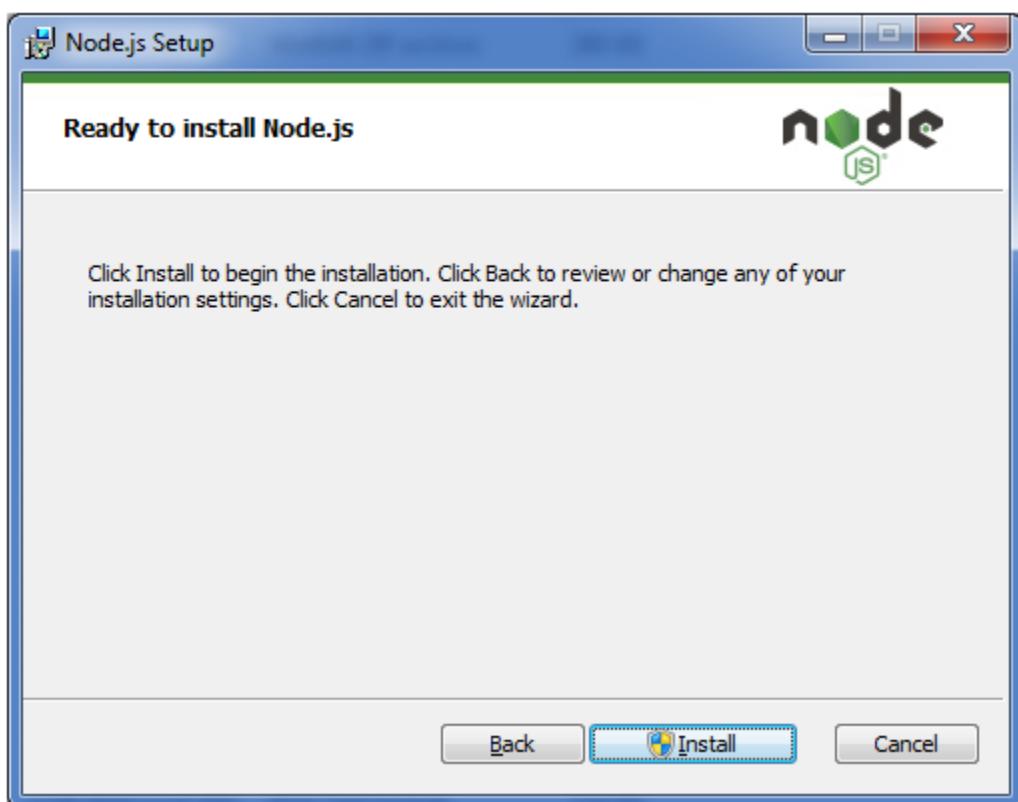
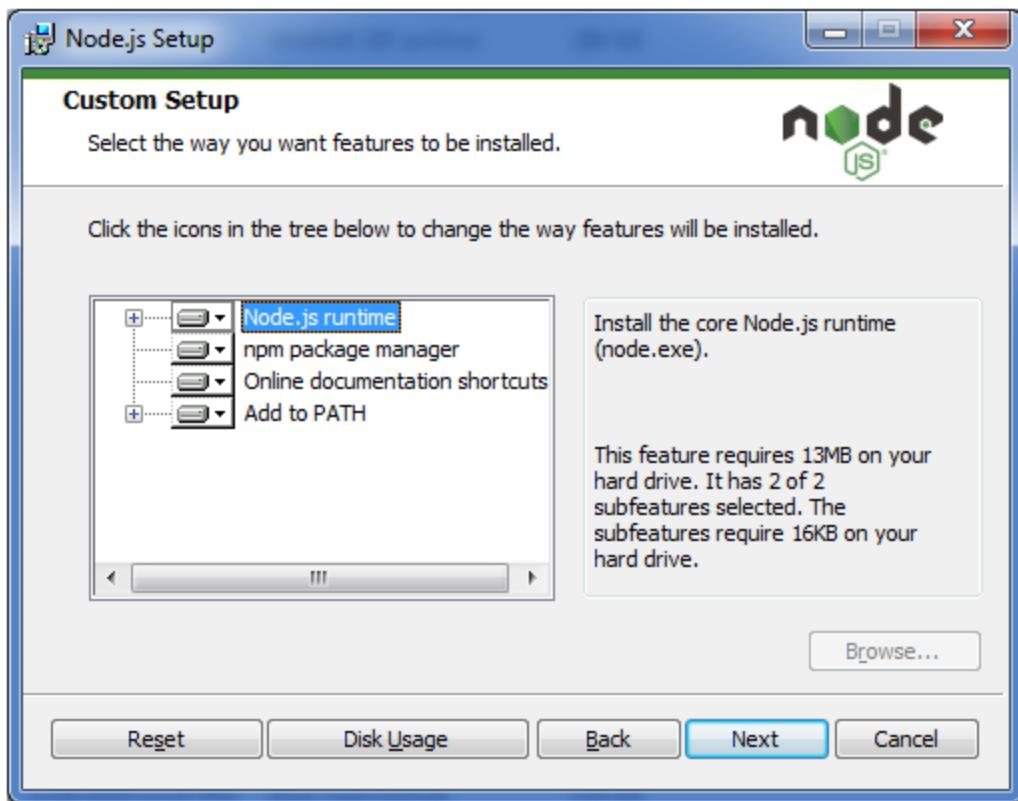


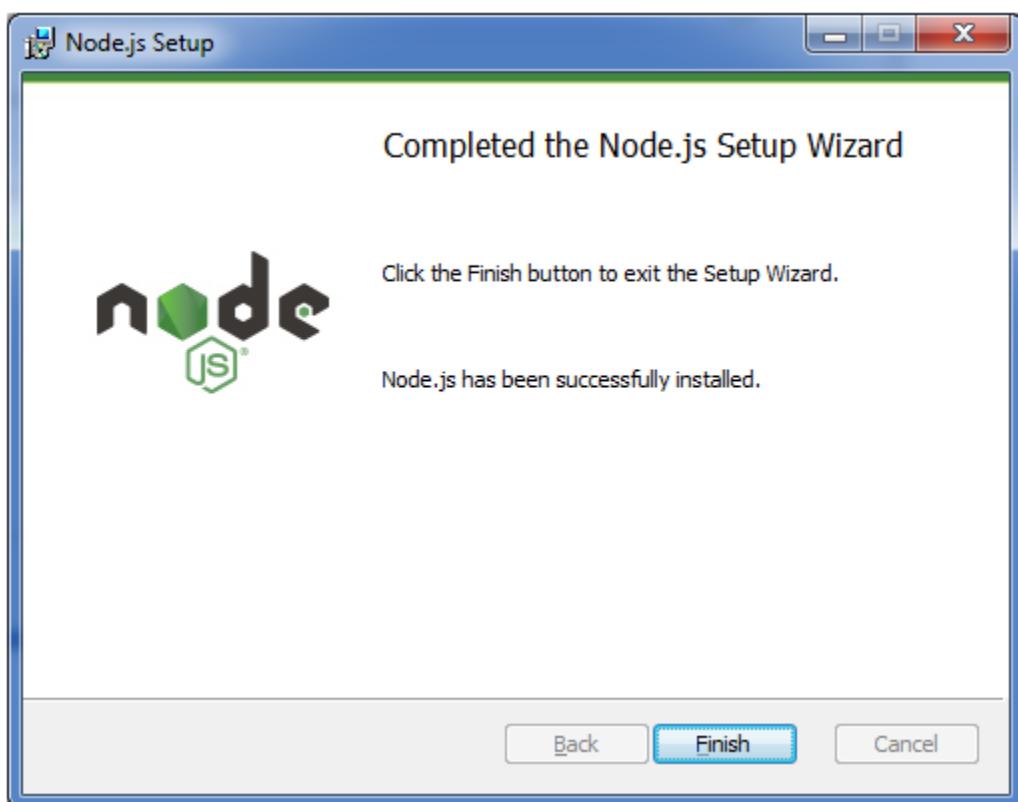
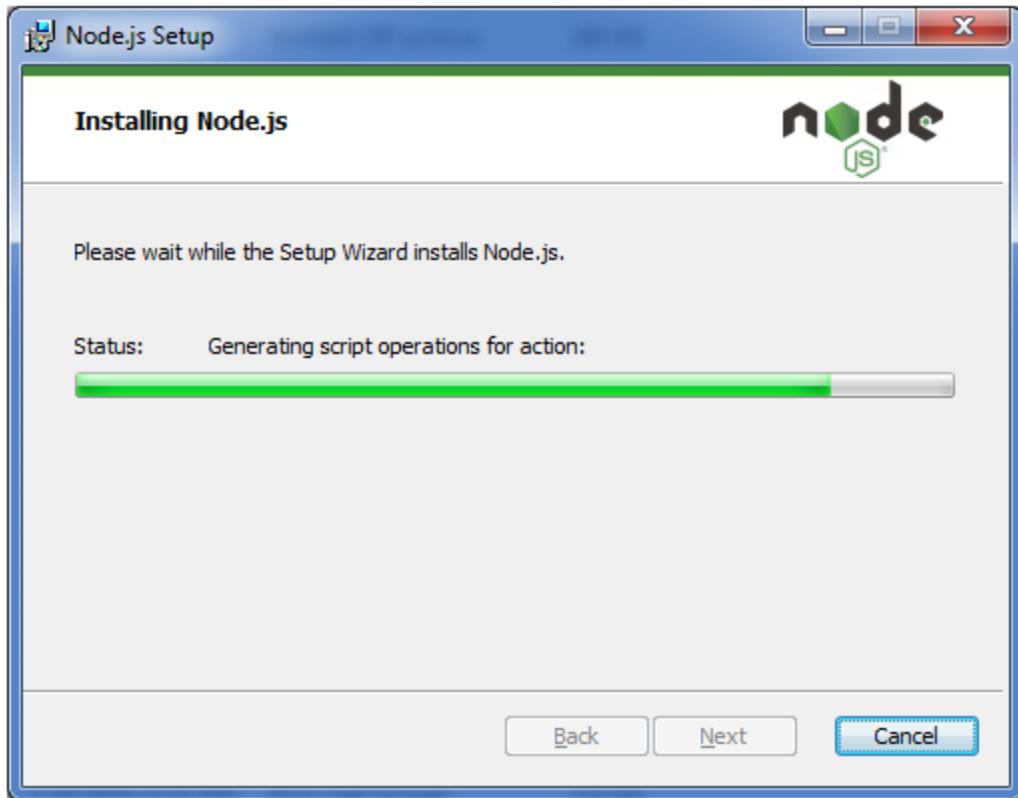
Accept the terms of license agreement.



Choose the location where you want to install.







What is Node Js?

- **Nodejs is a JavaScript runtime**
- Node.js is an open source server environment
- Node.js is free
- Node.js runs on various platforms (Windows, Linux, Unix, Mac OS X, etc.)
- Node.js uses JavaScript on the server
- Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.
- Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally,

Why Node.js?

Node.js uses asynchronous programming!

Here is how PHP or ASP handles a file request:

1. Sends the task to the computer's file system.
2. Waits while the file system opens and reads the file.
3. Returns the content to the client.
4. Ready to handle the next request.

Here is how Node.js handles a file request:

1. Sends the task to the computer's file system.
2. Ready to handle the next request.
3. When the file system has opened and read the file, the server returns the content to the client.

Node.js eliminates the waiting, and simply continues with the next request.

Node.js runs single-threaded, non-blocking, asynchronous programming, which is very memory efficient.

What Can Node.js Do?

- Node.js can generate dynamic page content
- Node.js can create, open, read, write, delete, and close files on the server

- Node.js can collect form data
- Node.js can add, delete, modify data in your database

What is a Node.js File?

- Node.js files contain tasks that will be executed on certain events
- A typical event is someone trying to access a port on the server
- Node.js files must be initiated on the server before having any effect
- Node.js files have extension ".js"

Features of Node.js

Following is a list of some important features of Node.js that makes it the first choice of software architects.

1. **Extremely fast:** Node.js is built on Google Chrome's V8 JavaScript Engine, so its library is very fast in code execution.
2. **I/O is Asynchronous and Event Driven:** All APIs of Node.js library are asynchronous i.e. non-blocking. So a Node.js based server never waits for an API to return data. The server moves to the next API after calling it and a notification mechanism of Events of Node.js helps the server to get a response from the previous API call. It is also a reason that it is very fast.
3. **Single threaded:** Node.js follows a single threaded model with event looping.
4. **Highly Scalable:** Node.js is highly scalable because event mechanism helps the server to respond in a non-blocking way.
5. **No buffering:** Node.js cuts down the overall processing time while uploading audio and video files. Node.js applications never buffer any data. These applications simply output the data in chunks.
6. **Open source:** Node.js has an open source community which has produced many excellent modules to add additional capabilities to Node.js applications.
7. **License:** Node.js is released under the MIT license.

First node js example:

result will be displaying in terminal

File save in .js extension

Run in :

Src is my main folder

Index.js is my file name

`node src/index.js`

Index.js

```
console.log(" Hello World ")
```

output: Hello World

Node.js REPL

The term **REPL** stands for **Read Eval Print and Loop**. It specifies a computer environment like a window console or a Unix/Linux shell where you can enter the commands and the system responds with an output in an interactive mode.

REPL Environment

The Node.js or node come bundled with REPL environment. Each part of the REPL environment has a specific work.

Read: It reads user's input; parse the input into JavaScript data-structure and stores in memory.

Eval: It takes and evaluates the data structure.

How to start REPL

You can start REPL by simply running "node" on the command prompt



After starting REPL node command prompt put any mathematical expression:



Using variable

Variables are used to store values and print later. If you don't use **var** keyword then value is stored in the variable and printed whereas if **var** keyword is used then value is stored but not printed. You can print variables using `console.log()`.

Ln 1, Col 25 Spaces: 4 UTF-8 CRLF {} JavaScript ⚡ Go Live ⚡ 🔍

Ln 1, Col 25 Spaces: 4 UTF-8 CRLF {} JavaScript ⚡ Go Live ⚡ 🔍

Hello World HTTP server

Index.js

```
const http = require('http'); // Loads the http module

http.createServer((request, response) => {

    // 1. Tell the browser everything is OK (Status code 200), and the data is in
    plain text

    response.writeHead(200, {
        'Content-Type': 'text/plain'
    });

    // 2. Write the announced text to the body of the page
    response.write('Hello, World!\n');

    // 3. Tell the server that all of the response headers and body have been sent
    response.end();

}).listen(1337); // 4. Tells the server what port to be on
```

Run the application by: node index.js

Hello World with Express:

Install express

npm install --save express

```
// Import the top-level function of express
const express = require('express');

// Creates an Express application using the top-level function
const app = express();

// Define port number as 3000
const port = 3000;

// Routes HTTP GET requests to the specified path "/" with the specified callback
function

app.get('/', function(request, response) {
```

```
response.send('Hello, World!');

});

// Make the app listen on port 3000

app.listen(port, function() {
  console.log('Server listening on http://localhost: ' + port);
});
```

What is a Module in Node.js?

Built-in Modules

Node.js has a set of built-in modules which you can use without any further installation.

To include a module, use the `require()` function with the name of the module:

```
var http = require('http');
```

Now your application has access to the HTTP module, and is able to create a server:

Create Your Own Modules

You can create your own modules, and easily include them in your applications.

The following example creates a module that returns a date and time object:

App.js

```
function sum(x,y){
  return x+y;
}

module.exports.sum=sum;
```

Use the `exports` keyword to make properties and methods available outside the module file.

Include Your Own Module

Now you can include and use the module in any of your Node.js files.

Index.js:

```
Const calculator= require("./app");
console.log(calculator.sum(2,4))
```

Node.js HTTP Module

The Built-in HTTP Module

Node.js has a built-in module called HTTP, which allows Node.js to transfer data over the Hyper Text Transfer Protocol (HTTP).

To include the HTTP module, use the `require()` method:

```
var http = require('http');
```

Node.js as a Web Server

The HTTP module can create an HTTP server that listens to server ports and gives a response back to the client.

Use the `createServer()` method to create an HTTP server:

```
var http = require('http');

//create a server object:

http.createServer(function (req, res) {
  res.write('Hello World!'); //write a response to the client
  res.end(); //end the response
}).listen(8080); //the server object listens on port 8080
```

Output:

operating system (os):

OS is a node module used to provide information about the computer
operating system

Advantages:

- It provides functions to interact with the operating system.
- It provides the hostname of the operating system and returns the amount of free system memory in bytes.

```
const os=require("node:os")
console.log("system freeSpace :",os.freemem())
//Returns the amount of free system memory in bytes as an integer.
console.log("operating system name :",os.type())
//Returns the operating system name as returned by
```



Node.js - Express Framework

Express Overview

Express is a minimal and flexible Node.js web application framework that provides a robust set of features to develop web and mobile applications. It facilitates the rapid development of Node based Web applications. Following are some of the core features of Express framework –

- Allows to set up middlewares to respond to HTTP Requests.
- Defines a routing table which is used to perform different actions based on HTTP Method and URL.
- Allows to dynamically render HTML Pages based on passing arguments to templates

Installing Express

Firstly, install the Express framework globally using NPM so that it can be used to create a web application using node terminal.

```
npm install express -save
```

The above command saves the installation locally in the **node_modules** directory and creates a directory **express** inside **node_modules**. You should install the following important modules along with express –

- **body-parser** – This is a node.js middleware for handling JSON, Raw, Text and URL encoded form data.

```
npm install body-parser -save
```

Hello world Example

Following is a very basic Express app which starts a server and listens on port 8081 for connection.

```
var express = require('express');
var app = express();
app.get('/', function (req, res) {
  res.send('Hello World');
})
var server = app.listen(3000, function () {
  var port = server.address().port
  console.log("Example app listening at", port)
})
```

Request & Response

Express application uses a callback function whose parameters are **request** and **response** objects

```
app.get('/', function (req, res) {  
  Response_Object  
    // --  
})
```

- Request Object – The request object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, and so on.
- Response Object – The response object represents the HTTP response that an Express app sends when it gets an HTTP request.

Basic Routing

We have seen a basic application which serves HTTP request for the homepage. Routing refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on).

We will extend our Hello World program to handle more types of HTTP requests.

```
var express = require('express');  
  
var app = express();  
  
// This responds with "Hello World" on the homepage  
  
app.get('/', function (req, res) {  
  console.log("Got a GET request for the homepage");  
  res.send('Hello GET');  
})  
  
// This responds a POST request for the homepage  
  
app.post('/', function (req, res) {  
  console.log("Got a POST request for the homepage");  
  res.send('Hello POST');  
})
```

```

// This responds a DELETE request for the /del_user page.

app.delete('/del_user', function (req, res) {
  console.log("Got a DELETE request for /del_user");
  res.send('Hello DELETE');
})

// This responds a GET request for the /list_user page.

app.get('/list_user', function (req, res) {
  console.log("Got a GET request for /list_user");
  res.send('Page Listing');
})

// This responds a GET request for abcd, abxcd, ab123cd, and so on

app.get('/ab*cd', function(req, res) {
  console.log("Got a GET request for /ab*cd");
  res.send('Page Pattern Match');
})

var server = app.listen(3000, function () {
  var port = server.address().port

  console.log("Example app listening at ", host, port)
})

```

GET Method:

Here is a simple example which passes two values using HTML FORM GET method. We are going to use **process_get** router inside server.js to handle this input.

