WEB DEVELOPMENT NOTES

**HTML-hypertext markup language**

\*) foundation for any website

\*) can't create website without html

------------------------------------------------------------------------------------------------

**USEFUL TAGS**

**HEADINGS IN DESCENDING ORDER**

<h1>this is a level 1 heading created by using h1 tag</h1>

<h1> (starting tag) Heading 1(content)</h1>(ending tag)

<h2>Heading 2</h2>

<h3>Heading 3</h3>

<h4>Heading 4</h4>

<h5>Heading 5</h5>

<h6>Heading 6</h6>

------------------------------------------------------------------------------------------------

**LINE BREAK (new line)**

<br/> - at the end of the line.

\*) self closing tag.

------------------------------------------------------------------------------------------------

**HORIZONTAL RULE(line)**

<hr /> - self closing tag for inserting lines

<hr size="3" noshade> (line with 3px in size and no shading)

-----------------------------------------------------------------------------------------------

**HTML BOILERPLATE(template**)

<!doctype html> - indicates that the below code is HTML5 type.

<html lang ="en" dir="ltr"> - start of the HTML code block.

<head> - start of the head content.

<meta charset="utf-8"> -Unicode Encoding System contains all the international symbol.

<title>topic</title> - title which display in the browser tab.

................

</head> - end of head content.

<body> - start of body content which contains the block used to display in the web pages.

..................

</body> - end of the body content.

</html> - end of the html code block.

------------------------------------------------------------------------------------------------

**PARAGRAPH**

<p> content </p> - used to insert paragraph in the web pages.

------------------------------------------------------------------------------------------------

**ITALICS**

<i> content </i>

-----------------------------------------------------------------------------------------------

**EMPHASIS**

<em> content </em> - styling the content.

---------------------------------------------------------------------------------------------

**BOLD**

<bold> content </bold> - highlighting the content.

<strong> content </strong> - highlighting the content.

------------------------------------------------------------------------------------------------

**HTML LIST**

**UNORDERED LIST** - bullets **ORDERED LIST** - numbering

<ul> <ol> or <ol type="i"> - for roman numerals

<li> content </li> <li> content </li>

</ul> </ol>

------------------------------------------------------------------------------------------------

**HTML IMAGE ELEMENTS**

<img src="gukan.png" alt="profile picture">

src - source that can be URL or file directory.

alt - alternative text to the picture in case any fault occurs.

------------------------------------------------------------------------------------------------

**HTML LINKS**

<a href="https://google.com">content </a> - anchor tags which is used for hyperlink.

href - can be link address or HTML file name.

------------------------------------------------------------------------------------------------

**HTML TABLES**

<table> - table tag used for inserting table into web (OR) <table border="1"> - table with border one pixel

<thead> - table header block

<tr> - table row used for inserting rows into table

<th>content</th> - head content that highlighted

<th>content</th>

</tr> - end of row

</thead> - end of head block in table.

<tbody> - body block of table

<tr>

<td>content</td> - table data used for inserting data into table

<td>content</td>

</tr>

</tbody> - end of the body block

<tfoot> - footer block of table

<tr><th>content</th><td>content</td></tr>

</tfoot> - end of footer block

</table> - end of the table

-----------------------------------------------------------------------------------------------

**HTML FORMS**

<form action="mailto:gukan08111997intelligent@gmail.com" method="post" enctype="text/plain"> <! --when press the submit button, the below info is posted in above mail address and encoding type is plain text-->

<label>your Name:</label>

<input type="text" name="yourName" value=""><br> - get the text input from the user

<label>your Email:</label>

<input type="email" name="yourEmail" value="" ><br> - get the email-id(auto check the email format) from the user

<label>your Message:</label><br>

<textarea name="yourMessage" rows="10" cols="30"><!--for size adjustment box in the message area--></textarea><br>

<input type="submit" name="">

</form>

------------------------------------------------------------------------------------------------

**CSS- CASCADING STYLE SHEETS**

\*) it is not a stand-alone programming language but it is used for html styling purpose

**INLINE CSS** - changing the style of HTML element within the HTML tag itself

<body style="background-color:#EAF6F6;"> (OR) <body style="background-color: blue;"> - changing the background color of the body by specifying either colour name or colour-code(hexa decimal)

NOTE: changing the style in every tag is inconvenient and error prone so we can use internal CSS.

------------------------------------------------------------------------------------------------

**INTERNAL CSS** - provide style for all related HTML tags in head block.

<head>

..........................

<style>

body { background-color: blue; }

hr { background-color: white;

border-style: none;

border-width: 5px;

border-color: grey;

border-top-style:dotted;

height:1px;

width:30%;}

img { height :20px;

}

</style>

</head>

------------------------------------------------------------------------------------------------

**EXTERNAL CSS**

The above code is stored separately in folder: css/styles.css

<!DOCTYPE html>

<html lang="en" dir="ltr"> CSS/STYLES.CSS

<head> body {

<meta charset="utf-8"> background-color: #EAF6F6;

<title>🥰 Gukan's personal site</title>

**<link rel="stylesheet" href="css/styles.css">** hr {

</head> border-style:dotted none none ;

border-width:5px;

border-color: grey;

width: 5%;

}

h1{

color:#66BFBF;

}

h3{

color:#66BFBF;

}

NOTE: The priority of the style is given to inline css---->internal css----->external css.

------------------------------------------------------------------------------------------------

**THE ANATOMY OF CSS SYNTAX**

selector { property: value;}

selector - (who?) - h1, img, body, p, table, etc.

property - (what?) - background-color, border, cell-spacing, height, etc.

value - (how?) - red, 1px, etc.

------------------------------------------------------------------------------------------------

**CSS SELECTORS**

NOTE: if we use the tag selectors then the property we change will affect all related tag selector hence to overcome this we use class selectors.

EXMP: <img class="bacon" src="https://emojipedia-us.s3.amazonaws.com/thumbs/240/apple/118/bacon\_1f953.png" alt="bacon-img">

<img class="broccoli" src="https://emojipedia-us.s3.dualstack.us-west-1.amazonaws.com/thumbs/120/apple/325/broccoli\_1f966.png" alt="broccoli-img">

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*class-selectors\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

.bacon{

background-color: green;

}

.broccoli{

background-color: red;

}

------------------------------------------------------------------------------------------------

**CLASS Vs ID**

NOTE: Use classes when you want to apply the same style to a group of related items and use the id to apply a specific style to a single element on your web page. One or more selectors can have same class name and can have more than one class name but id should be unique.

Similarity is that both the priority of the class and id have precedence over tag selectors.

-------------------------------------------------

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>Bacon Fansite</title>

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

</head>

<body>

<h1 id="heading">I Love Bacon</h1>

<p>bacon, bacon, bacon, bacon, bacon, bacon</p>

<p>bacon, bacon, bacon, bacon, bacon, bacon</p>

<p>bacon, bacon, bacon, bacon, bacon, bacon</p>

<img class="bacon" src="https://emojipedia-us.s3.amazonaws.com/thumbs/240/apple/118/bacon\_1f953.png" alt="bacon-img">

<img class="broccoli circular" src="https://emojipedia-us.s3.dualstack.us-west-1.amazonaws.com/thumbs/120/apple/325/broccoli\_1f966.png" alt="broccoli-img">

</body>

</html>

--------------------------------------------------

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*class-selectors\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

.bacon{

background-color: green;

}

.broccoli{

background-color: red;

}

.circular{

border-radius: 100%;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*id-selectors\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#heading {

color: black;

}

------------------------------------------------------------------------------------------------

**PSEUDO CLASS**

img:hover {

background-color: gold;

} - :hover is pseudo class by which we can change colour when you hover over particular selector.

------------------------------------------------------------------------------------------------

**INTERMEDIATE CSS**

**FAVICON** - favourite icon

<link rel="icon" href="favicon.ico"> - favicon.ico --> icon image file name

------------------------------------------------------------------------------------------------

**HTML DIV** - content division element

It basically allows you to split up or divide your content into separate containers or boxes so that you can affect the layout of each box separately.

<body> div{

<div class=""> background-color: #AAC4FF;

<h1>i'm gukan</h1> margin-top: 0;

<p>a web developer</p> }

</div>

</body>

------------------------------------------------------------------------------------------------

**BOX MODEL OF CSS STYLING**

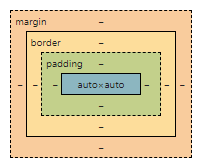
All contents in web page are considered as the boxes(margin, border, padding, height of the box, width of the box),

By manipulating the box dimensions, we can manipulate the content position, looking, etc. This is called Box model of CSS styling.

**BORDER** - border, border-(position)-color, border-(positon)-width, border-(positon)-style, border-(position)-radius where position - top, bottom, left, right.

**MARGIN** - margin-(position): 0px; margin:0px 10px 20px 30px(circular from top to left clockwise)

**PADDING** - padding-(position): 0px;(note: if it is 0 pixel, it is ok to omit px)



------------------------------------------------------------------------------------------------

**CSS DISPLAY PROPERTY**

It has four values - Block, Inline, Inline-Block, none.

\*)**Block** elements are those that take up essentially the whole width of the screen on a web page, so effectively blocking out any other elements from sitting next to it on the left or on the right.

EXMP - Paragraph(<p>), Headings(<h1> to <h6>), Divisions(<div>), List and list items(<li>,<ol>,<ul>), Forms(<form>).

\*)An **inline** display element only takes up as much space as it needs to in the height and in its width. Here we can't able to change the width of element.

EXMP - Spans(<span>), Images(<img>), Anchors(<a>).

\*)An **inline-block** elements are those in which we can change the width of that element and also we can add other element sitting adjacent to it.

EXMP- image elements(<img>).

\*)**None** means as if it is deleted(gone from web page along with space occupied by it).

**Display:none** – gone and affect the flow of the web page.

**Visibility:hidden** – gone and left the space behind which occupied by it

<p>gukan</p> p{

<p class="para">devan</p> background-color: red;

<span>hello</span> display: inline-block;

<span>world</span> width: 100px;

}

.para{

background-color: yellow;

}

span{

background-color: blue;

display: inline-block;

width: 100px;

}

-----------------------------------------------------------------------------------------------

**CSS POSITIONS**

NOTE: HTML Predefined rules for how the contents display on the web page.

i) Content is everything - small-font content takes smaller space and big-font content takes bigger space.

ii) Orders come from HTML code.

iii) Children sit on parents - <body><h1><span>web</span>developer</h1></body> --> order of parent to child - BODY-->H1-->SPAN.

CSS position property has four values - Static, Relative, Absolute, Fixed.

**POSITION: STATIC**

All HTML elements are static in their position by default, and static just means go along with the HTML rules and keep to the default HTML flow.

**POSITION: RELATIVE**

This allows us to position the element that we select relative to how it would have been positioned in static.

div.blue{

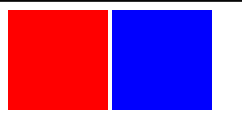
width:100px;

height:100px;

background-color:blue;

display:inline-block;

}

****

div.red{

width:100px;

height:100px;

background-color:red;

display:inline-block;

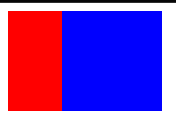
}

<div class="red">

</div>

<div class="blue">

</div>

****

div.blue{

width:100px;

height:100px;

background-color:blue;

display:inline-block; **position:relative;**

**right:50px;**

}

**POSITION: ABSOLUTE**

The elements are positioned with respect to its parent.

div.blue{

width:100px;

height:100px;

background-color:blue;

display:inline-block; **position:absolute;**

**right:50px;**

}



Here the parent for **blue div** is entire **body** so it positioned 50px right with respect to body.

**NOTE:** when we use **relative positioning** where we left behind some sort of ghost element where it still kept the positions of everything else the same, but when we use **absolute positioning**, you're actually taking the element out of the flow of the document and it's no longer considered a part of the natural flow of the document.

div.container {

height:150px;

width:300px;

background-color:grey;

**position:relative;**

}

div.red {

width:100px;

height:100px;

background-color:red;

display:inline-block;

**position:absolute;**

right:50px;

}

<body>

<div class="container">

<div class="red">

</div>

</div>

</body>



Here the parent for **div.red** is **div.container** hence div.red positioned itself right by 50px with respect to div.container.

**POSITION: FIXED**

If we make the position of any element fixed then we scroll down the web page, it will stay at same position all the time.

**CENTERING ELEMENTS WITH CSS**

h1{margin-top: 0;}

body{

margin: 0;

**text-align:center;**

}



h1{ margin-top: 0;

**width:10%;}**

body{

margin: 0;

text-align:center;

}

->here changing the width of the heading will bring it to initial position nonetheless (text-align:center;), to overcome this we use (**margin:0 auto;)**

h1{

 margin-top: 0;

width:10%;

**margin:0 auto;**

}

body{

margin: 0;

text-align:center;

}

------------------------------------------------------------------------------------------------

**FONT STYLING IN CSS**

<link rel="preconnect" href="https://fonts.googleapis.com">

<link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>

<link href="https://fonts.googleapis.com/css2?family=Merriweather&family=Montserrat&family=Sacramento&display=swap" rel="stylesheet">

Here, at first it goes to link provided in first two lines and then it fetches the font info from it.

**font-family**: 'Sacramento', cursive;

**font-family**: 'Montserrat', sans-serif;

here if any problem occurs in first font style then it uses the second font style which is web-safe fonts.

**CSS FONT SIZING**

**Font-size**:100%=16px=1em🡪where “em” for dynamic font size.

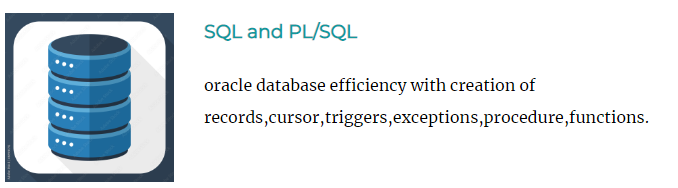
NOTE: When we use **%, px, em** in font-size the problem (the font-size of both parent and child applied together to make font bigger or smaller) occurs.

To overcome this, we use **rem🡪root em** which remove font-size of parent and apply only the current font-size of the child.

For 90px🡪 90/16 = 5.625 🡪5.625rem.

**CSS FLOAT**

Wrapping of text over an image either on left or on right.

.sk1{

width:25%;

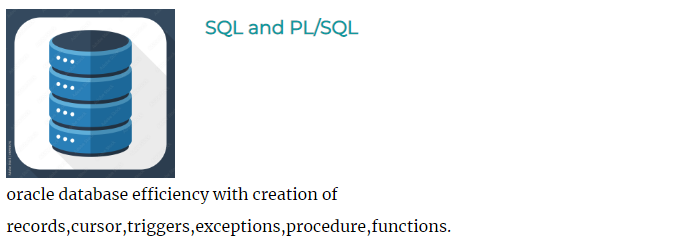
**float:left;**

margin-right:30px;

}

**CSS CLEAR**

The main function of clear is to eliminate any wrapping of images either on left or on right to itself.



.sk1{

width:25%;

float:left;

margin-right:30px;

}

.para1{

**clear:left;**

}

**BOOTSTRAP – FRONTEND LIBRARY**

 Bootstrap is very firmly on the front-end.

It's a bunch of code that was written to be reusable that you can drop into your site, and it'll make designing the website much, much easier for you.

Now, Bootstrap was originally developed by Twitter in order to try and get more consistency across websites that they design.

Bootstrap makes the layout of website adaptable to every devices (laptop, mobile, tablet) – i.e Responsive.

Bootstrap contains built-in css code which when added as class then it will style the selectors in HTML elements

<div class="**navbar bg-dark**">

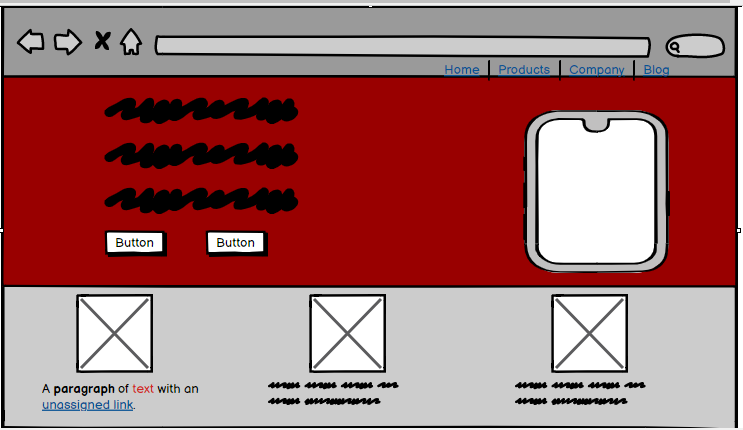
<button class="**btn btn-primary**">hello world</button>

</div>



**WEB DESIGN**

**WIRE FRAMING –** are the low fidelity representation of your work using pen and a paper.



**MOCK-UPS –** are the high-fidelity representation of app and website’s design.

More likely the finished product.

**NAVBAR –** navigation bar

<nav class="**navbar navbar-expand-lg navbar-dark bg-dark**">

<a class="**navbar-brand**" href="">tingdog</a>

<button class="**navbar-toggler**" type="button" data-bs-toggle="collapse" data-bs-target="**#navbarTogglerDemo02**" aria-controls="navbarTogglerDemo02" aria-expanded="false" aria-label="Toggle navigation">

<span class="navbar-toggler-icon"></span>

</button>

<div class="**collapse navbar-collapse**" id="**navbarTogglerDemo02**">

<ul class="**navbar-nav ms-auto**">

<li class="**nav-item**">

<a class="**nav-link**" href="">Content</a>

</li>

<li class="**nav-item**">

<a class="**nav-link**" href="">pricing</a>

</li>

<li class="**nav-item**">

<a class="**nav-link**" href="">Download</a>

</li>

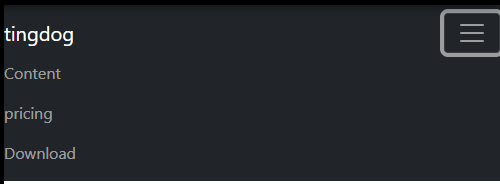
</ul>

</div>

</nav>



When the screen shrinks either by (converting laptop mode to mobile mode) or by (reducing the browser page size) then it automatically collapse into the dropdown button.



**BOOTSTRAP GRID LAYOUT SYSTEM**

<div class="**row**">

<div class="**col**" style="background-color:red;border: 1px solid;">

col

</div>

<div class="**col**" style="background-color:red;border: 1px solid;">

col

</div>

</div>

<div class="**row**">

<div class="**col-6**" style="background-color:green; border:1px solid">col-6</div>

<div class="**col-3**" style="background-color:green; border:1px solid">col-3</div>

<div class="**col-2**" style="background-color:green; border:1px solid">col-2</div>

<div class="**col-1**" style="background-color:green; border:1px solid">col-1</div>

</div>

<div class="row">

<div class="**col-lg-3 col-md-4 col-sm-4 col-6**" style="background-color:yellow; border:1px solid;">col-lg-3 col-md-4 col-sm-6</div>

<div class="**col-lg-3 col-md-4 col-sm-4 col-6**" style="background-color:yellow; border:1px solid;">col-lg-3 col-md-4 col-sm-6</div>

<div class="**col-lg-3 col-md-4 col-sm-4 col-6**" style="background-color:yellow; border:1px solid;">col-lg-3 col-md-4 col-sm-6</div>

<div class="**col-lg-3 col-md-4 col-sm-4 col-6**" style="background-color:yellow; border:1px solid;">col-lg-3 col-md-4 col-sm-6</div>

</div>

<div class="row">

<div class="**col-lg-2 col-md-3 col-12**" style="background-color:grey; border:1px solid;">col-lg-2 col-md-3 col-sm-12</div>

<div class="**col-lg-2 col-md-3 col-12**" style="background-color:grey; border:1px solid;">col-lg-2 col-md-3 col-sm-12</div>

<div class="**col-lg-2 col-md-3 col-12**" style="background-color:grey; border:1px solid;">col-lg-2 col-md-3 col-sm-12</div>

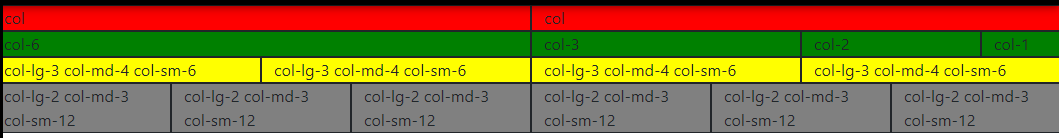
<div class="**col-lg-2 col-md-3 col-12**" style="background-color:grey; border:1px solid;">col-lg-2 col-md-3 col-sm-12</div>

<div class="**col-lg-2 col-md-3 col-12**" style="background-color:grey; border:1px solid;">col-lg-2 col-md-3 col-sm-12</div>

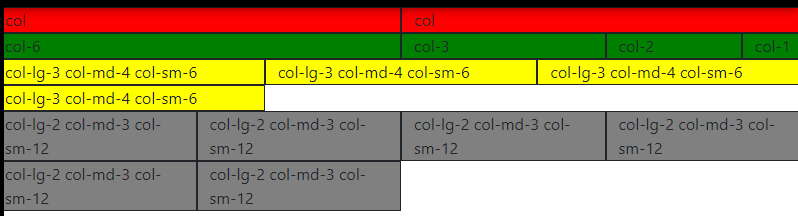
<div class="**col-lg-2 col-md-3 col-12**" style="background-color:grey; border:1px solid;">col-lg-2 col-md-3 col-sm-12</div>

</div>

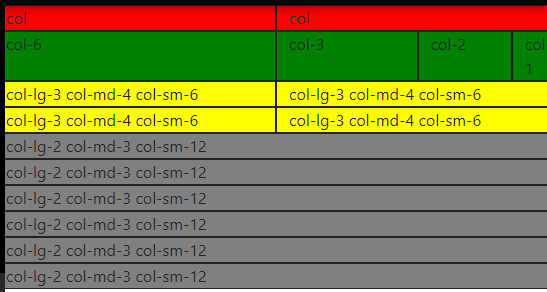
Here when laptop mode (lg-large) – 6 div with 2 units – grey



When tablet mode (md-medium) – 4 div with 3 units - grey



When mobile mode (sm-small) – 1 div with 12 units - grey



<div class="row">

**<div class="col-lg-6"> <!--when large mode(laptop) – 2divs with 6 units**

**When medium and small it occupies 100%-->**

<h1 class="heading1">Meet new and interesting dogs nearby.</h1>

<button type="button">Download</button>

<button type="button">Download</button>

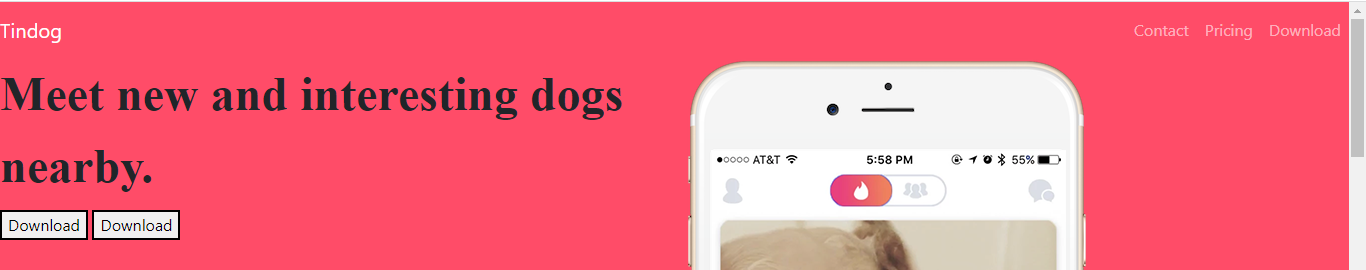
</div>

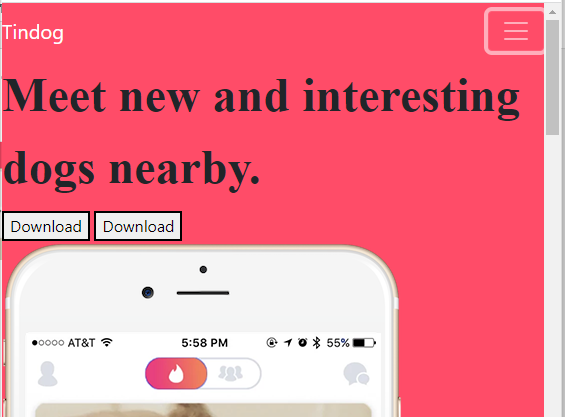
**<div class="col-lg-6">**

<img src="images/iphone6.png" alt="iphone-mockup">

</div>

</div>





**CONTAINER**

Containers are a fundamental building block of Bootstrap that contain, pad, and align your content within a given device or viewport.

Bootstrap comes with three different containers:

* **.container**, which sets a max-width at each responsive breakpoint
* .container-{breakpoint}, which is width: 100% until the specified breakpoint
* .**container-fluid**, which is width: 100% at all breakpoints.

<**div class="container"** style="background-color:green;">

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Id consectetur purus ut faucibus pulvinar.

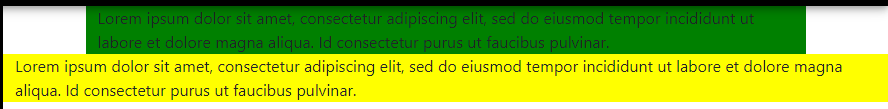
</div>

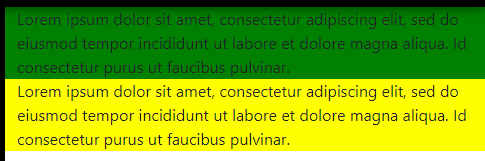
<**div class="container-fluid"** style="background-color:yellow;">

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Id consectetur purus ut faucibus pulvinar.

</div>

In laptop mode



In mobile or tablet mode

<div class="**container-fluid**">

<div class="row">

<div class="col-lg-6">

<h1 class="heading1">Meet new and interesting dogs nearby.</h1>

<button type="button">Download</button>

<button type="button">Download</button>

</div>

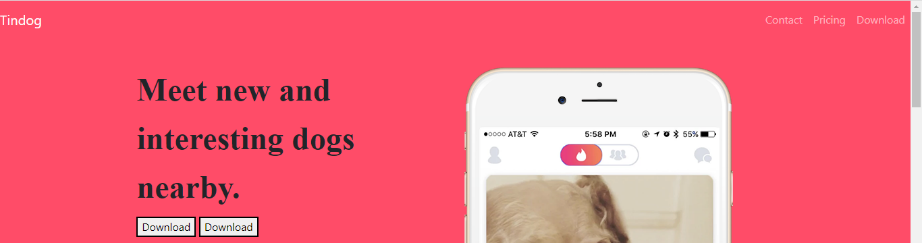
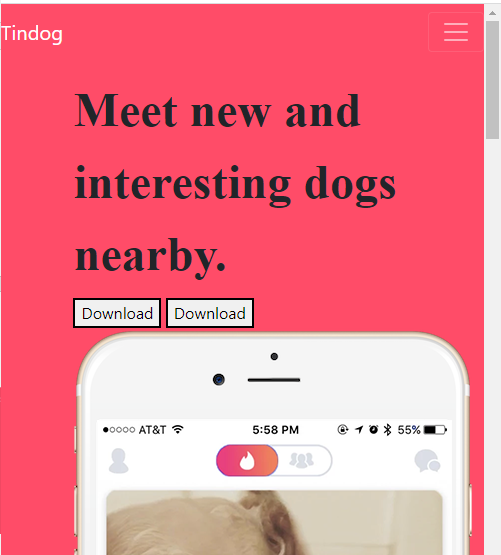
<div class="col-lg-6">

<img src="images/iphone6.png" alt="iphone-mockup">

</div>

</div>

In laptop mode In mobile mode



**BOOTSTRAP BUTTON AND FONT AWESOME**

Use lots of icons free for web projects by simply copy and paste the CDN in the **head** section of HTML code and use it in body section.

<!-- Font Awesome -->

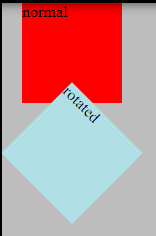
<scriptsrc="https://kit.fontawesome.com/3a6d1ac116.js"crossorigin="anonymous"></script>

<**button** type="button" class="btn btn-dark btn-lg">**<i class="fa-brands fa-apple"></i>** Download</button>

<**button** type="button" class="btn btn-outline-light btn-lg">**<i class="fa-brands fa-google-play"></i>** Download</button>



**IMAGE ROTATION**

<div class="normal">normal</div>

<div class="rotate">rotated</div>

.normal{

background-color:red;

height:100px;

width:100px;

margin-left:20px;

}

.rotate{

background-color:powderblue;

height:100px;

width:100px;

margin-left:20px;

**transform:rotate(45deg);+ for clockwise, - for anticlockwise.**

}

**ICON SIZING**

<i class="icon fa-solid fa-circle-check **fa-4x**"></i>

For different **x (**2x, 3x, 4x, etc**)** the size gets varied.

**CAROUSEL**

Slideshow for website

[**https://getbootstrap.com/docs/5.2/components/carousel/**](https://getbootstrap.com/docs/5.2/components/carousel/)

**CARDS**

Cards with header, body, footer

<https://getbootstrap.com/docs/5.2/components/card/>

**Z-INDEX**

**Position:Relative/Absolute** must be declared for z-index to work. Default value of z-index for all elements is 0.

<div class="head">Header</div>

<div class="body">Body</div>

<div class="foot">Footer</div>

.head{

height:100px;

width:100px;

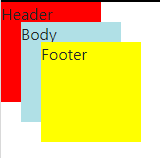
background-color:red;

}

.body{

height:100px;

width:100px;

 background-color:powderblue;

position:absolute;

top:20px;left:20px;**z-index:1;**

}

.foot{

height:100px;

width:100px;

background-color:yellow;

position:absolute;

top:40px;left:40px;**z-index:2;**

}

**MEDIA QUERY BREAKPOINTS**

**Three ways of creating websites**

1. Do nothing – create only large-sized website for laptop.
2. Create a separate website for mobile size.
3. Create a responsive website – preferrable.

In order to create responsive websites, we use media query breakpoints.

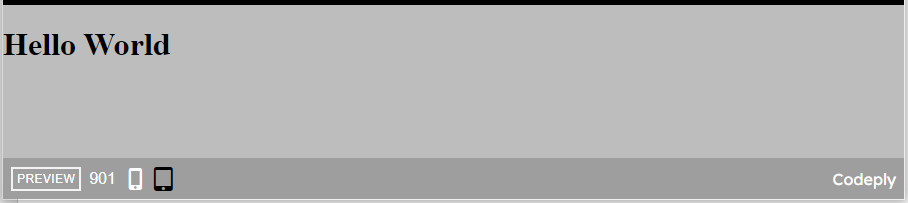
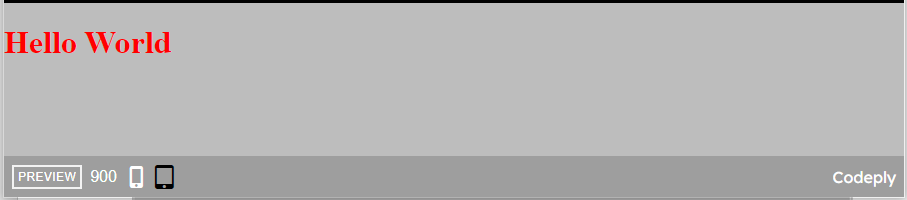
**@media <type> <feature>**

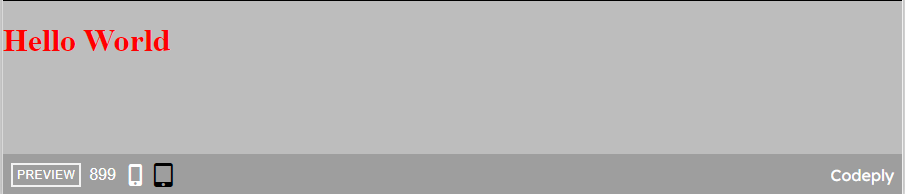
@media screen (min-width:900px)

{ //change something

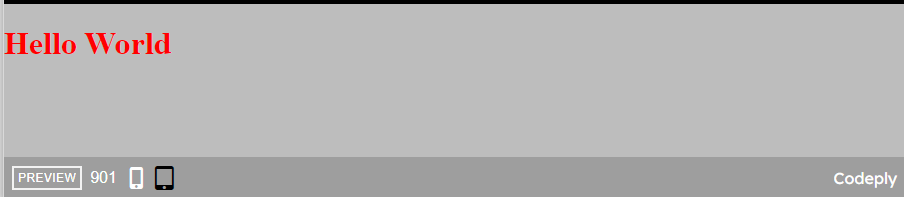
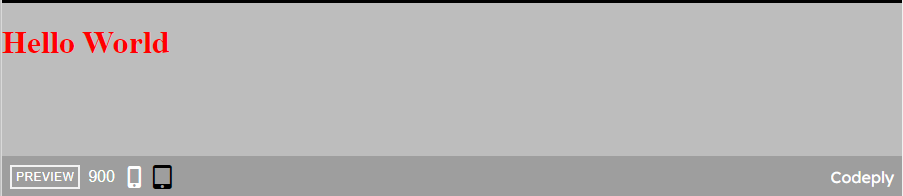
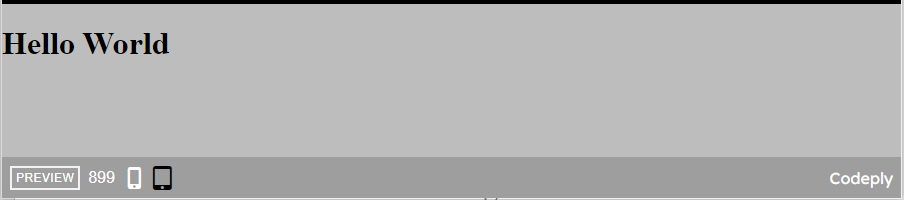
}

**<h1>Hello World</h1> @media (max-width:900px){ h1{ color:red;} }** – here the colour of heading h1 turns from black to red when the screen size attains 900px and remain in red there from lesser than 900px onwards.





**<h1>Hello World</h1> @media (min-width:900px){ h1{ color:red;} }**



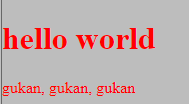
**CODE REFACTORING**

Four principles for code refactoring (importance from top to bottom)

1. Readability – using **comments** and write code in proper sequence.
2. Modularity – narrow down your code with several **sections**.
3. Efficiency – **fast** execution.
4. Length – **length** of the code.

**COMBINING SELECTORS**

**MULTIPLE SELECTORS**

**selector1, selector2 { …….. }**

h1, h2, h3, h4, h5, h6{ font-family:”montserrat”;}

<h1>hello world</h1> h1,p{

<p>gukan, gukan, gukan</p> color:red;

}

**HIERARCHICAL SELECTORS**

**selector1 selector2 {……..}** where selector1 – parent and selector2 – child**(space must)**

* Read from right to left – selector2 which is inside the selector1 should have this style.

<div class="container1"> **div h1**{

<h1 class="title1">hello</h1> color:red;

</div> }

<div class="container2">

<h1 class="title2">world</h1>

</div>

Here all the h1 element inside all the div element will have the styles applied**(broaden).**

**.container1 .title1**{

color:red;

}

**.container2 .title2**{

color:blue;

}

Here element with the **class title1** which inside the element with **class container1** will have **red** colour text and element with the **class title2** which inside the element with **class container2** will have **blue** colour text**(specific).**

**COMBINED SELECTORS**

**selector1selector2{………}–**selector1 is HTML element and selector2 is either class or id(**no space between selectors)**

<h1 class="**title**">hello</h1> **h1.title**{ color:red;}

<h1 id="**heading**">world</h1> **h1#heading**{ color:blue;}

Here the **h1 with class title** and **h1 with id heading** will have the style specified



<div class="**container1**"> **.container1 h1.title**{ color:red;}

<h1 class="**title**">hello</h1> **.container2 h1.title**{ color:blue;}

</div>

<div class="**container2**">

<h1 class="**title**">world</h1>

</div>

Here **the h1 with class title which is inside the element with class container1** will have red colour text.

**WEB DESIGN**

**FOUR PRINCIPLES**

1. Colour theory
2. Typography
3. User interface design (UI)
4. User experience design (UX)

**COLOUR THEORY**

* Love, Energy, Intensity
* Joy, Intellect, Attention
* Freshness, Safety, Growth
* Stability, Trust, Serinity
* Royalty, Wealth, Femininity

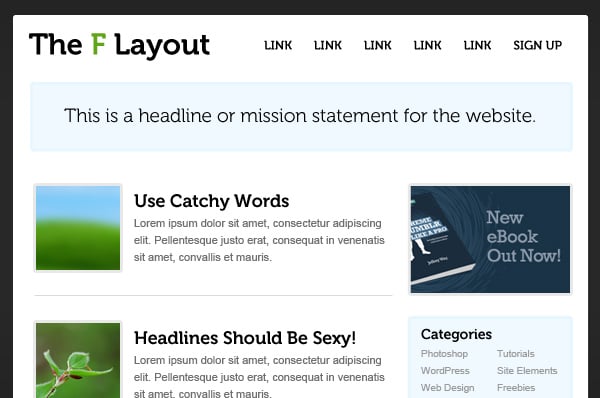
**TYPOGRAPHY**

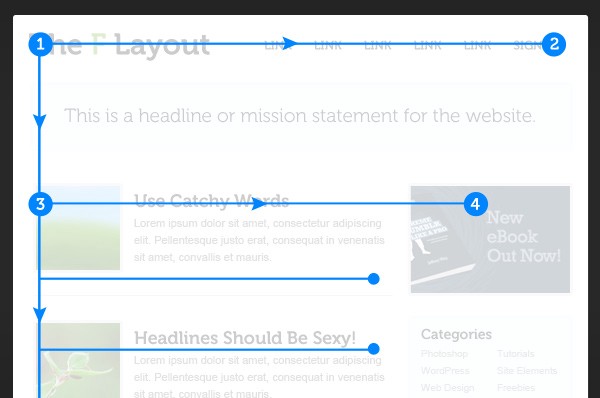
****

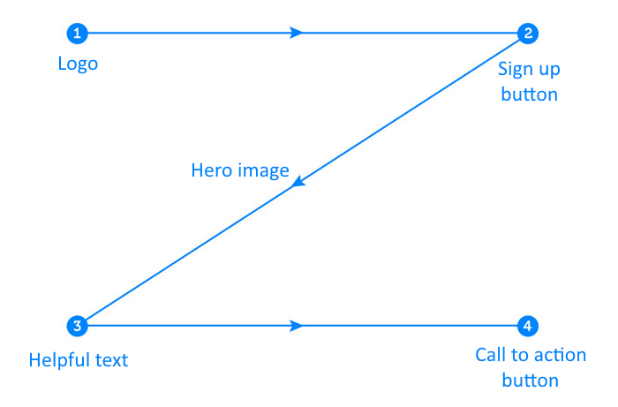
**USER INTERFACE DESIGN (UI)**

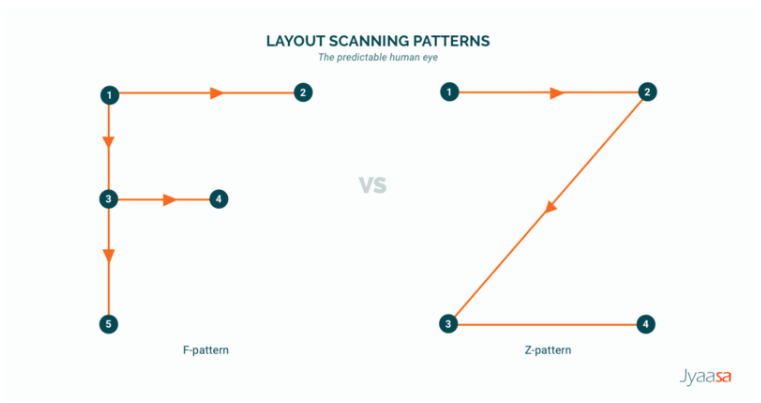
1. Hierarchy – colours (primary to secondary), font-size (large to small), etc.
2. Layout – page with pictures, videos are separated by containers.
3. Alignment – left, right and centre.
4. White space – space around attentive objects.
5. Audience – depending on moods, ages (children, middle-aged, old).

**USER EXPERIENCE DESIGN (UX)**

1. Simplicity – keep things simple as possible.
2. Consistency – in design and functionality (older version to upgrade version).
3. Reading patterns







1. All platform design – Responsive (desktop to mobile).
2. Don’t use your power for evil.

**JAVASCRIPT**

JavaScript is an interpreted programming language.

slow and they had to execute all the instructions line by line.

Use to add functionalities and user interaction to our websites.

**alert(“hello”); -** pop-up the message inside the double quotes.

**alert** – keyword/function **hello** – message **;** - end.

**DATATYPES**

String, Number, Boolean

**typeof(“hello”); -** gives the datatype which is inside the parenthesis **-** string

**typeof(23); -** number

**typeof(true);** - Boolean.

**VARIABLES -**  containers

**var myName = “gukan”; -** here gukan is stored inside the myName.

**var –** keyword for variable **myName –** variable name **gukan –** value.

**var yourName = prompt(“what is your name?”); -**  here the prompt get the name and store it inside the yourName.

**NAMING CONVENTIONS**

1. There is no spaces between variable name – **my name( invalid).**
2. Variable name should not start with number or symbol – **1my, $my( invalid).**
3. Reserved keywords should not be used – **var, alert, prompt, etc(invalid).**
4. Should follow camel casing – **myName, yourName.**
5. Variable name can contain **letters, numbers, $, \_.**

**CONCATENATION**

“hello” **+** “ ” **+** “world” – hello world.

**STRING LENGTH**

**var written = prompt ("write a tweet");**

**alert ("you have written " + written.length + " words," + " you have " + 180-written.length + " characters left.");**

written.length – gives the length of the string.

**STRING SLICE**

var name = “gukan”;

name.**slice**(0,1); - g

name.**slice**(0,3); - guk

name**.slice**(2,3); - k

**alert( prompt("write a tweet").slice(0,140)); -** first gets a tweet from a user and then slice the first 140 characters and make a alert to user.

**EXAMPLE**

**var name = prompt("what is your name?");**

**var firstLetter = name.slice(0,1).toUpperCase();**

**var remLetter = name.slice(1,name.length).toLowerCase();**

**alert("Hello, "+ firstLetter + remLetter);**

**BASIC ARITHEMETIC**

**var dogAge = prompt("what is your dog's age?");**

**var humanAge = ((dogAge-2)\*4)+21; -** BODMAS rule.

**alert("dog's human age is " + humanAge);**

**INCREMENT AND DECREMENT OPERATOR**

x = x+1 => **x++**

x = x-1 => **x--**

x = x(+,-,\*,/) any number => **x(+,-,\*,/)=that number**

x = x\*2 => x\*=2

**FUNCTIONS**

**CREATING A FUNCTION:**

**Function function-name () {** code to be executed **}**

**CALLING A FUNCTION:**

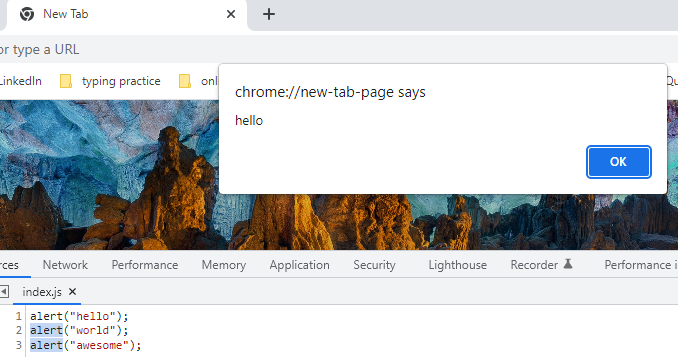
**Function-name ();**

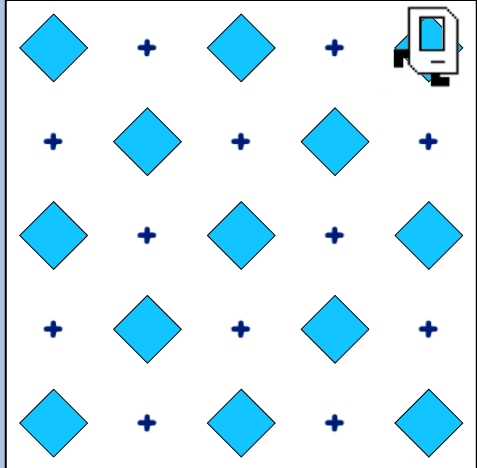
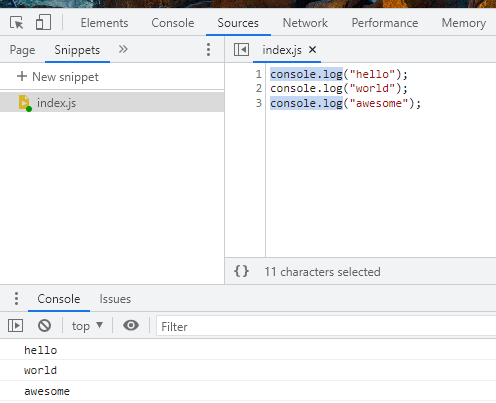
**ALERT AND CONSOLE LOG**

Important difference between the console log and the alert is that the alert is something that the user can see. So if you had **an alert** on your web site, then **any visitor will be able to see the alert**, but the **console logs** are only for the developer.

They **only show up in the console**, which is not intended for the user.

It's meant for the developer to debug your code, to find out if there were any problems, and to essentially print out parts of the code into the console.



****

function main(){

movePutBeeper();

movePutBeeper();

moveLeft();

movePutBeeper();

moveRight();

movePutBeeper();

movePutBeeper();

moveLeft();

movePutBeeper();

moveRight();

movePutBeeper();

movePutBeeper();

putBeeper();

}

function movePutBeeper(){

putBeeper();

move();

move();

}

function moveLeft(){

putBeeper();

turnLeft();

move();

turnLeft();

move();

}

function moveRight(){

putBeeper();

move();

turnRight();

move();

turnRight();

}

**FUNCTIONS WITH PARAMETER AND ARGUMENTS**

function getMilk(**money**) {

console.log("buy " + **Math.floor(money/1.5)** + " bottles of milk"); - **here Math.floor is to round-off**

}

getMilk(10);

function lifeInWeeks(**age**) {

var x =((90\*365)-(**age**\*365));

var y =((90\*52)-(**age**\*52));

var z =((90\*12)-(**age**\*12));

console.log("You have "+ x + " days, " + y + " weeks, and " + z + " months left." );

}

lifeInWeeks(25);

**FUNCTIONS WITH OUTPUT AND RETURN VALUES**

function getMilk(**money**) {

console.log("buy " + **Math.floor(money/1.5)** + " bottles of milk");

**return** money % 1.5;

}

Var change = getMilk(10);

Console.log(change);

**BMI CHALLENGE**

function **bmiCalculator** (weight, height) {

var bmi1 = **Math.round**(weight/**Math.pow**(height,2));

return bmi1;

}

Var bmi = bmiCalculator(65,1.8);

**RANDOM GENERATION**

**Math.random() -**  static method returns a floating-point, pseudo-random number that's greater than or equal to 0 and less than 1 (0 – 0.9999999999999999(16 decimal places)).

**Math.floor() -** The **Math.floor()** static method always rounds down and returns the largest integer less than or equal to a given number.

console.log(Math.floor(5.95));

// expected output: 5

console.log(Math.floor(5.05));

// expected output: 5

console.log(Math.floor(5));

// expected output: 5

console.log(Math.floor(-5.05));

// expected output: -6

**LOVE CALCULATOR CHALLENGE**

var firstName = prompt("first person Name");

var secondName = prompt("second person Name");

var n= **Math.random();**

n = n\*100;

n = Math.floor(n)+1;

alert("love percentage is :" + n + " %");

**CONTROL STATEMENTS**

**IF - ELSE STATEMENTS**

If (n === 3) {code to executed}

Else {code to be executed }

**COMPARATORS**

**=== -** equal to.

!== - not equal to.

>= - greater than or equal to. > - greater than.

<= - lesser than or equal to. < - lesser than.

Var a = 1;

Var b =”1”;

If (a ===/== b) { console.log(“yes”);} –> **here == only checks the value whereas === checks the value and also its datatypes are matching.**

Else { console.log(“no”);}

**COMBINING COMPARATORS**

&& - **AND || - OR ! – NOT**

**if (n > 70)** {

alert("love percentage is :" + n + " %" + " love like flesh and skin");

}

**if (n > 30 && n <= 70)** {

alert("love percentage is :" + n + " %");

}

**if (n <=30)** {

alert("love percentage is :" + n + " %" + "love like oil and water");

}

**BMI CALCULATOR ADVANCED**

function bmiCalculator (weight, height) {

var bmi = weight/(Math.pow(height,2));

var interpretation;

if (bmi<18.5){

interpretation = "Your BMI is " + bmi +", so you are underweight."

}

if (bmi>=18.5 && bmi<=24.9){

interpretation = "Your BMI is " + bmi + ", so you have a normal weight."

}

if (bmi>24.9){

interpretation = "Your BMI is " + bmi +", so you are overweight."

}

return interpretation;

}

**LEAP YEAR CHALLENGE**

function isLeap(year) {

var statement;

if (year%4 ===0) {

if (year%100 === 0) {

if (year%400 === 0){

statement = "Leap year.";

}

else{

statement = "Not leap year.";

}

}

else {

statement = "Leap year.";

}

}

else {

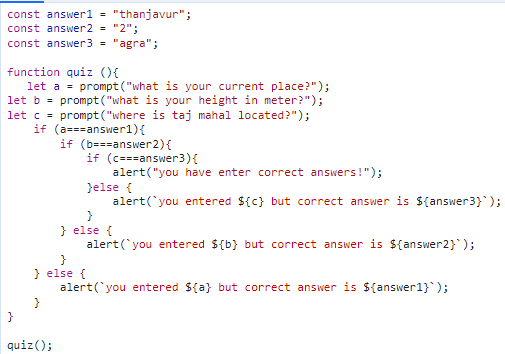
statement = "Not leap year.";

}

return statement;

}

**Quiz question**

****

**ARRAYS**

It’s a collection of items that are related, and they can be stored together into the same container or the same variable.

**var name = [“gukan”, “vasu”, “devan”]; - initialization.**

var guestList = ["Angela", "Jack", "Pam", "James", "Lara", "Jason"];

var name = prompt("what is your name?");

if (guestList**.includes**(name)){

alert("Welcome!");

}

else {

alert("Sorry, next time please");

}

Here **guestList.includes(name) –** checks whether the name included in the guestlist array or not.

**FIZZBUZZ CHALLENGE**

var output = [];

var count = 1;

function fizzBuzz(){

if (count%3 === 0 && count%5 ===0){

output**.push**("FizzBuzz");  **-> here output.push() pushes the value into the array.**

} **similarly, output.pop() removes the last element of array.**

else if (count%3 === 0){

output.push("fizz");

}

else if (count%5 === 0){

output.push("Buzz");

}

else{

output.push(count);

}

count++;

console.log(output);

}

**Who buying the lunch challenge**

Var names = [“gukan”, “vasu”, “devan”, “mala”, “devi”];

function whosPaying(names) {

var n = Math.random();

n = Math.floor(n \* names.length );

return names[n] + " is going to buy lunch today!";

}

**CONTROL STATEMENTS**

**WHILE LOOPS**

**While ( something is true ) {**

**// do something**

**}**

**99 BOTTLES CHALLENGE**

var count = 99;

function beer() {

while (count >= 1) {

var less = count-1;

console.log(count + " bottles of beer on the wall, " + count + " bottles of beer. Take 1 down, pass it around, " + less + " bottles of beer on the wall.");

count--;

}

console.log("No more bottles of beer on the wall, no more bottles of beer.");

}

**FOR LOOP**

**For (i=1; i<2; i++) {code to be executed }**

var output = [];

function fizzBuzz(){

**for (count = 1; count<=100; count++){**

if (count%3 === 0 && count%5 ===0){

output.push("FizzBuzz");

}

else if (count%3 === 0){

output.push("fizz");

}

else if (count%5 === 0){

output.push("Buzz");

}

else{

output.push(count);

}

}

console.log(output);

}

**FIBONACCI SERIES**

function fibonacciGenerator (n) {

var output = [];

var a,b,c,x,y,z;

if (n===1) {

output.push(0);

return output;

}

else if (n===2) {

output.push(0,1);

return output;

}

else if (n>2) {

a = 0;

b = 1;

output.push(a,b);

for (var i=2; i<n; i++) {

z = a+b;

output.push(z);

c = a;

a = b;

b = c+a;

}

return output;

}

}

**OR**

function fibonacciGenerator (n) {

var output = [];

if (n===1) {

output.push(0);

}

else if (n===2) {

output.push(0,1);

}

else if (n>2) {

output.push(0,1);

for (var i=2; i<n; i++) {

output.push(output[output.length-2]+output[output.length-1]);

}

}

return output;

}

**ADDING JAVASCRIPT TO WEBSITES**

Similar to css styles, we can add javascript in three ways

**INLINE JAVASCRIPT**

<html lang="en" dir="ltr">

<head>

<meta charset="utf-8">

<title>My Website</title>

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

</head>

**<body onload="alert('Hello');"> - here onload will execute the javascript codes inside**

<h1>Hello</h1> **the quotes when body loads up.**

</body>

</html>

**INTERNAL JAVASCRIPT**

Here we use **script tag** for internal, everything in between the script tag are consider as javascript codes.

<!DOCTYPE html>

<html lang="en" dir="ltr">

<head>

<meta charset="utf-8">

<title>My Website</title>

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

</head>

<body>

<h1>Hello</h1>

**<script type="text/javascript">**

**alert("hello");**

**</script>**

</body>

</html>

**EXTERNAL JAVASCRIPT**

Here we use **script tag** that links source file**(index.js).**

<!DOCTYPE html>

<html lang="en" dir="ltr"> **index.js**

<head> alert("hello");

<meta charset="utf-8">

<title>My Website</title>

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

</head>

<body>

<h1>Hello</h1>

**<script src="index.js" charset="utf-8"></script>**

</body>

</html>

Unlike css, javascript has different usage, because inside our script we tried to change an HTML element that does not yet exist then it will fail. So **always try to add external script at the bottom of body element to make sure the content loads first and then script.**

**DOCUMENT OBJECT MODEL (DOM)**

catalogues the web page into individual objects **that we can select and manipulate in live state.**

when you load up the web page, the browser turns each of these elements and their associated data into a **tree structure** with a whole bunch of objects that you can select and manipulate.

Objects inside the DOM can have

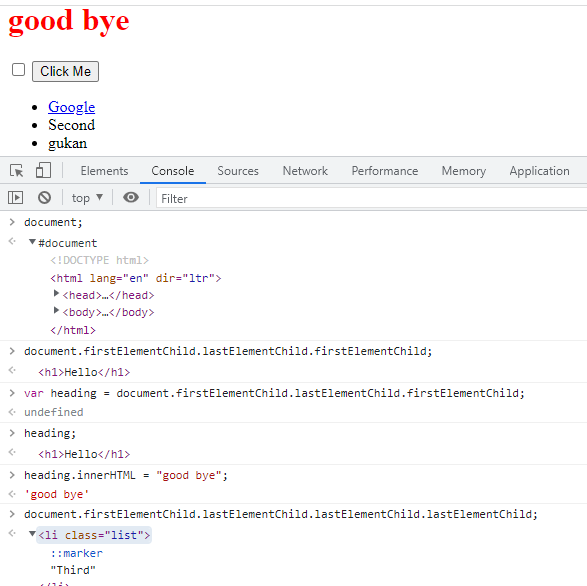
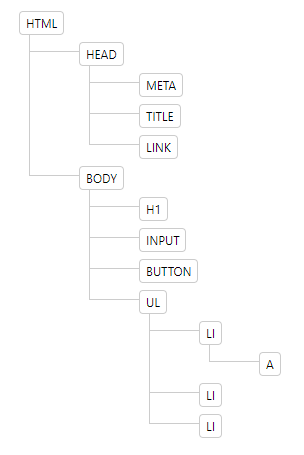
* **Properties(describe something about the object)**

Get property – **heading.colour;**

Set property – **heading.colour = “red”;**

* **Methods(something that object can do).**

**SELECTING HTML ELEMENT USING DOM**



**getElementsByTagName(“name of the tag”); -** gives the output in form of **array.**

used to select the **tag** mentioned for manipulation.

**getElementsByClassName(“name of the class”);** - gives the o/p in form of **array**.

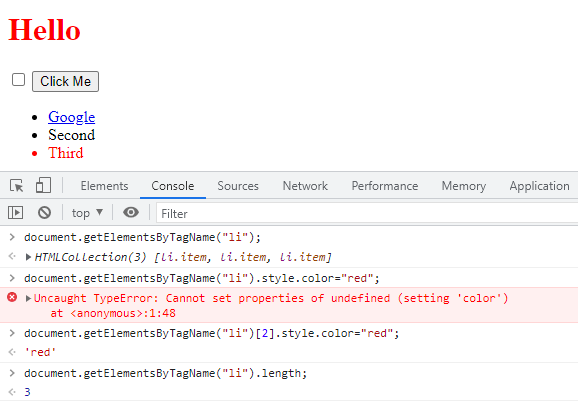
Used to select the **class** mentioned for manipulation.

**getElementById(“id name”);** - gives the **single** element.

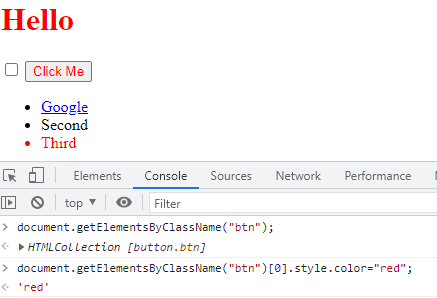
Used to select the **id** mentioned for manipulation.

**querySelector(“tag/.class/#id/combined selectors”); -**gives only single element which is first.

**querySelectorAll(“tag/.class/#id/combined selectors”); -** gives the all elements in **array.**



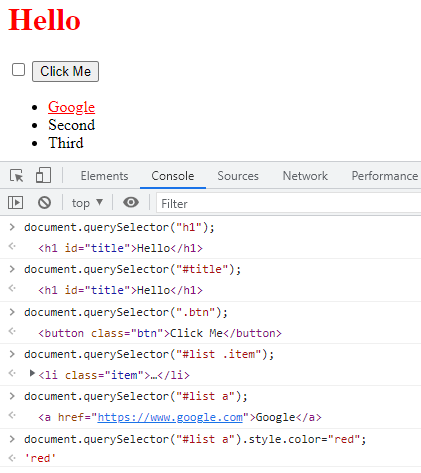
**getElementsByTagName(“tag”);**

****

**getElementsByClassName(“class”);**



**getElementById(“id”);**



**querySelector(“tag/class/id/combined selectors”);**



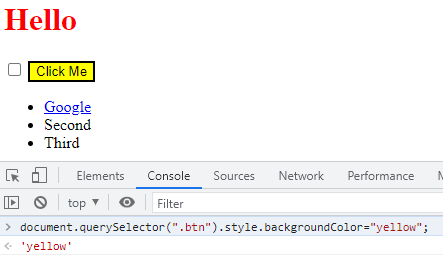
**querySelectorAll(“tag/class/id/combined selectors”);**

**MANIPULATING AND CHANGING STYLES OF HTML ELEMENTS USING JAVASCRIPT**

Unlike css, we have different usage for property like

**CSS JAVASCRIPT**

1. **font-size: 10rem; fontSize = ”10rem”;**
2. **background-color: red; backgroundColor = ”red”;**



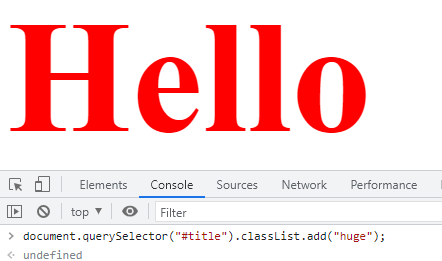
**SEPERATION OF CONCERNS**

**HTML –** only for content.

**CSS –** only for styles.

**JAVASCRIPT –** only for behaviour.

At the same time we can change the style of HTML element using javascript methods like

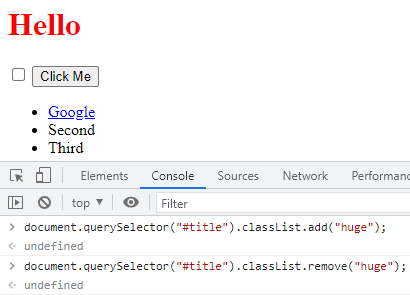


**css/styles.css**

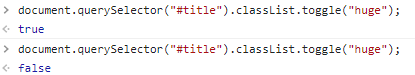
.huge {

Font-size:10 rem;

}



**.toggle(“class name”); -** this **method add the class if not added and remove the class if not removed**



**MANIPULATION OF TEXT AND ITS PROPERTY**

**<h1 id = "title"><strong>Hello</strong></h1>**

****

Here **.innerHTML** gives the entire

Text(**<strong>Hello</strong>)** b between the **h1** tags whereas

**.textcontent** gives only the text(**hello)**

****

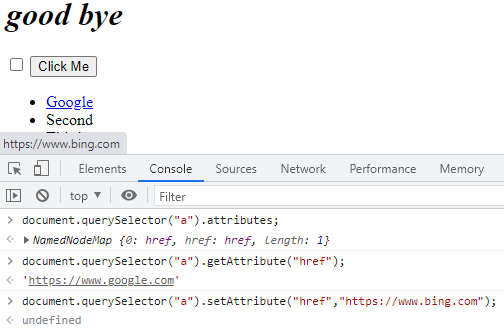
Here we can change **inner text only** using **.textcontent** whereas using **.innerHTML** we can change the **text property** also.

**MANIPULATION OF HTML ELEMENT ATTRIBUTES**

**.attributes –** gives the list of attributes which is associated with the HTML element.

**.getAttribute(“name of the attribute”); –** get the value of that specified attribute.

**.setAttribute(“attribute name”, “value”); –** set the value of the specified attribute.

****

**ADDING EVENT LISTENER TO THE BUTTON**

**EventTarget.addEventListener()**

method of the EventTarget interface sets up a function that will be called whenever the specified event is delivered to the target.

**.addEventListener(type, listener)**

**type -** A case-sensitive string representing the [event type](https://developer.mozilla.org/en-US/docs/Web/Events) to listen for.

**Listener -** The object that receives a notification (an object that implements the [Event](https://developer.mozilla.org/en-US/docs/Web/API/Event) interface) when an event of the specified type occurs. ( or simply a **javascript function**)

**document.querySelector("button").addEventListener("click",handleClick); -** here when the button got clicked then EventListener will call the function handleClick whereas the alert message shown.

**function handleClick(){**

**alert("i got clicked");**

**}**

EventTarget - **document.querySelector("button") – button.**

Event type – **click –** whenever the button clicked.

Listener – **handleClick** – javascript function – here the function call didn’t include parenthesis because **handleClick()** will call the function as soon as line gets loaded without waiting for the **click** event type.

**document.querySelector("button").addEventListener("click",function (){**

**alert("i got clicked");**

**});**

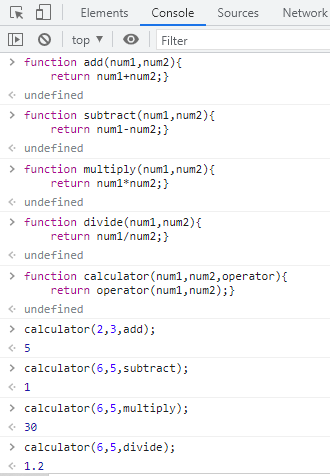
here the **listener – anonymous function.**

**HIGHER OREDER FUNCTIONS**

Functions which can take other functions as inputs are called higher order functions.

For debugging:

1. type - **debugger;** - then **hold shift + enter.**
2. **Code for debugging. –** press **enter.**



**PLAYING SOUNDS ON WEBSITES**

**HTML AUDIO ELEMENT**

var audio = **new Audio**("sounds/tom-1.mp3");

audio**.play();**

here **new Audio** method used to identify and load the sound files

**play()** method used to play the sound.

**ADDING IMAGE TO THE BUTTON**

<button class="w drum">w</button> **css/styles.css**

.w {

 **background-image: url(“images/tom1.png”);**

}

**JAVASCRIPT OBJECT WITH PROPERTIES**

Similar to **structure** in C, here we have **javascript objects – storing different datatype in single variable.**

var houseKeeper1 = {

name:"angela",

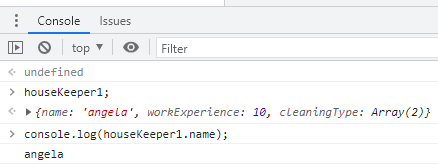
workExperience: 10,

cleaningType: ["bathroom","kitchen"]

}

Here **houseKeeper1** is javascript object whereas **name, workExperience, cleaningType** are its properties.

Alert(“hello, my name is “+”**houseKeeper1.name”**);



here the problem is, if we want to create another housekeeper object means we want to write whole bunch of properties (name, workExperience, cleaningType) again. So this can be overcome by constructor function.

**CONSTRUCTOR FUNCTION**

**function HouseKeeper** (name, experience, cleaningType) {

**this.name**=name;

**this.workExperience** = experience;

**this.cleaningType** = cleaningType

}

Here the **function name (HouseKeeper)** should be **capitalized not camel casing.**

**var houseKeeper1 = new HouseKeeper("angela",10, ["bathroom","kitchen"]);** - creating objects with constructor function.

**var houseKeeper2 = new HouseKeeper("vennila",12,["kitchen","bedroom"]);**

**SWITCH STATEMENT IN JAVASCRIPT**

Switch (condition) {

Case (expression):

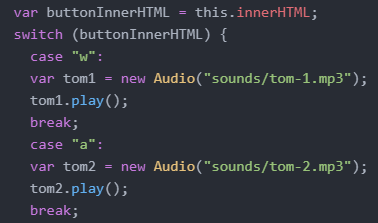
Break;

Case (expression):

Break;

………

Default:





**JAVASCRIPT OBJECTS WITH PROPERTIES AND METHODS**

function HouseKeeper (name, experience, cleaningType) {

this.name=name;

this.workExperience = experience;

this.cleaningType = cleaningType;

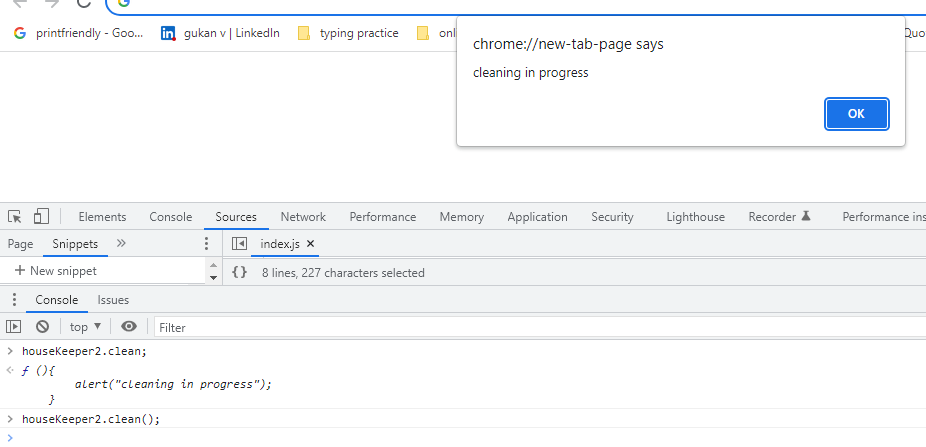
**this.clean = function(){**

**alert("cleaning in progress");**

**}**

}

Here the constructor function with both properties(name, workExperience and cleaningType) and methods(**clean**)



**KEYBOARD EVENT LISTENER**

**document.addEventListener("keydown",function(event){**

**drumSound(event.key);**

**});**

Here the entire webpage look for the event (**key pressed)** to be occurred. At the time when event occurs, addEventListener will execute the anonymous function.

Here **EventTarget** – document – entire web page.

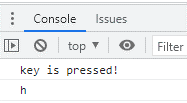
**EventType** - **Element:” keydown” event -**

The keydown event is fired when a key is pressed.

Unlike the deprecated [keypress](https://developer.mozilla.org/en-US/docs/Web/API/Element/keypress_event) event, the keydown event is fired for all keys, regardless of whether they produce a character value.

**LISTENER –** anonymous function executes whenever the **event(key press in keyboard)** occurs.

**drumSound(event.key); -** custom function created in atom editor. Where **event.key** gives the key which was pressed.



document.addEventListener("**keydown**",function(event){

console.log("key is pressed!");

console.log(**event.key**);

});  
here .addEventListener – **higher order function**  and function(event) – **callback function.**

**EXPLANATION**

function **anotherAddEventListener**( **typeOfEvent, callback** ){

//event occured

var **eventThatHappened** = {

eventType : "keydown",

key : "p",

duration : 2

}

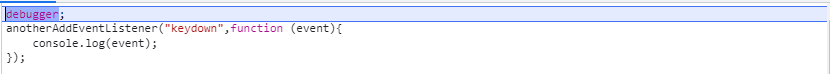
if (**eventThatHappened.eventType === typeOfEvent**){

**callback**(eventThatHappened);

}

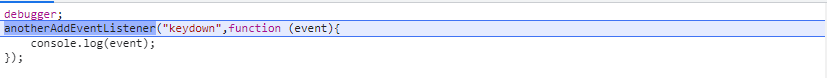
}

**EXECUTION SEQUENCE**

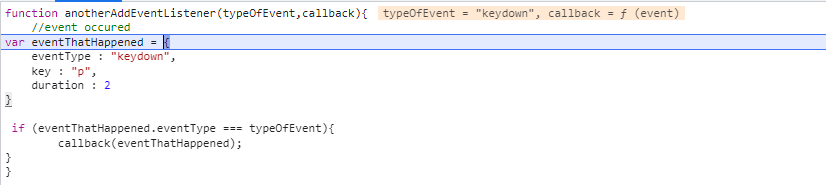


1. we begin by calling **anotherAddEventListener**, passing in these two inputs.

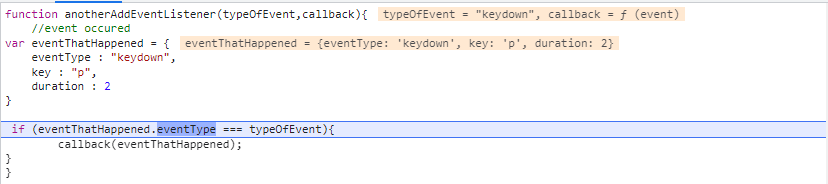
So that takes us to where that function was declared.



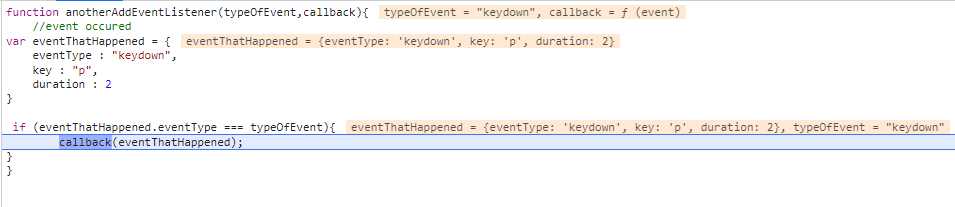
1. The first thing that happens is we're **waiting to detect an event**. And this part of the code waits to detect an event. Let's say that **I press the P key on the keyboard.** Now at that point then this **eventThatHappened** **object gets created** and it captures a lot of information about my key press.



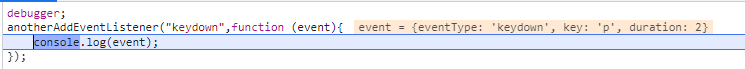
1. this event object created after I triggered the event then our code will **check if the type of the event that happened is equal to the type** that the program was looking for then it will go ahead and trigger the callback function.



1. then the **callback function** (the function that we used as an input) is **triggered**.



1. it goes ahead and tries **to log the event object.**



**SET TIMEOUT FUNCTION**

The setTimeout() method calls a function after a number of milliseconds.

1 second = 1000 milliseconds.

**setTimeout(myGreeting, 5000); -** myGreeting is function, 5000- waiting time(milliseconds).

function buttonAnimation(currentKey){

var btnPressed = document.querySelector("." + currentKey);

btnPressed.classList.add("pressed");

**setTimeout(function(){**

**btnPressed.classList.remove("pressed");**

**},100); - (after 0.1 second the function will be executed.)**

}

**JQUERY**

Javascript library used to minimize our code such as

**Document.querySelector(“h1”);** can be replaced by **jQuery(“h1”); OR $(“h1”);**

**ADDING JQUERY TO WEBSITES**

1. Simply add the script(CDN) before the end of the body element and external javascript also.

**<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.6.3/jquery.min.js"></script>**

<script src="index.js" charset="utf-8"></script>

</body>

**Index.js**

$(“h1”).css(“color”,”red”);

1. Use **ready function** to ensure that jquery script fully loaded before it was used in the website.

**Index.js**

$(document).**ready**(function(){

$("h1").css("color","red");

})

**OR**

$(function(){

$("h1").css("color","red");

})

**SELECTING ELEMENTS WITH JQUERY**

Document.querySelector(“h1”); replaced by **$(“h1”);**

Document.querySelectorAll(“button”); replaced by **$(“button”);**

**MANIPULATING STYLES WITH JQUERY**

Get property -



Set property -

By keeping in mind the **separation of concerns,** the styling is only done by CSS.

**Styles.css**

.big-title{

font-size:5rem;

font-family:cursive;

color:yellow;

}

.margin-50{

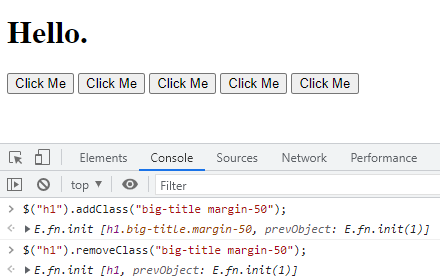
margin:50px;

}

1. **addClass –** used to add classes using jquery



1. **removeClass –** used to remove classes using jquery.

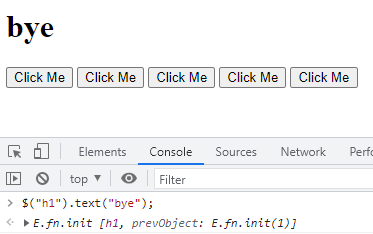


1. **hasClass –** gives true/false whether the specified class is applied or not.



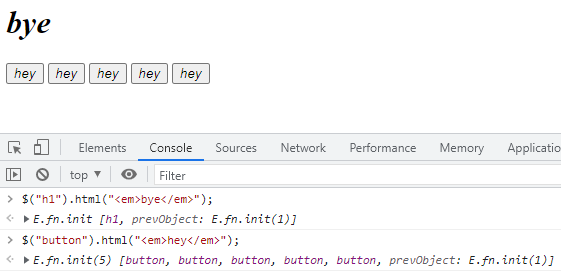
**MANIPULATING TEXT WITH JQUERY**

**.text(“content”); -** method to manipulate the text only using jquery.

****

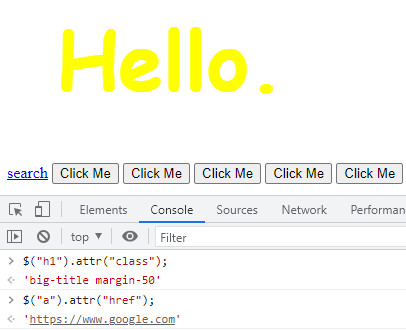
**MANIPULATING TEXT PROPERTY**

**.html(“content”); -** method to manipulate the text and its property also using jquery.

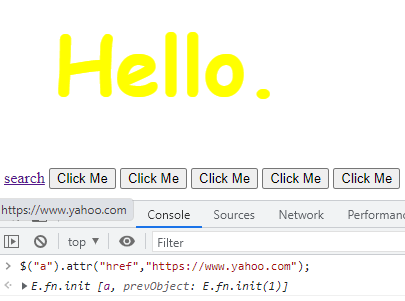
****

**MANIPULATING ATTRIBUTES WITH JQUERY**

**.attr(“attribute name”); -** to get the attribute value.



**.attr(“attribute name”, “value to be changed for that attribute”); -** to set attribute value.



**ADDING EVENT LISTENERS WITH JQUERY**

1. $(document).**keydown**(function(){

$("h1").text(event.key);

});

$("button").**click**(function(){

$("h1").css("color","purple");

});

1. $("button").**on**("**click**",function(){

$("h1").css("color","purple");

});

$(document).**on**("**keydown**",function(){

$("h1").text(event.key);

});

**ADDING ELEMENTS WITH JQUERY**

**Before method –**

$("h1").**before**("<button>new</button>"); 🡪

**<button>new</button>**<h1>hello.</h1>

****

**After method –**

$("h1").**after**("<button>new</button>"); 🡪

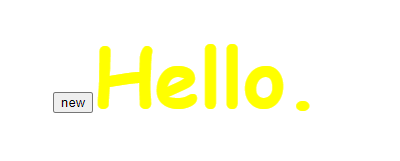
<h1>hello.</h1**><button>new</button>**

****

**Prepend method –**

$("h1").**prepend**("<button>new</button>"); 🡪

<h1>**<button>new</button>**hello.</h1>

****

**Append method –**

$("h1").**append**("<button>new</button>"); 🡪

<h1> hello**.<button>new</button>** </h1>

****

**REMOVING ELEMENTS WITH JQUERY**

**$("button").remove(); -** removes all the specified elements(button).

**WEBSITE ANIMATIONS USING JQUERY**

$("button").on(“click”, function(){

$("h1").[**hide, show, toggle, slideUp, slideDown, slideToggle, fadeIn, fadeOut, fadeToggle]**();

});

$("button").on(“click”, function(){

$("h1").**animate({opacity:0.5, margin:20}); -** here the **animate** method only acceptproperty with **numeric value.**

});

**CHAINING THE ANIMATIONS**

$("button").on(“click”, function(){

$("h1").**slideUp().slideDown().animate({opacity:0.5, margin:20});**

});

**BACKEND – NODEJS**

A backend framework used to execute the javaScript codes directly by the computer itself with the help of Hyper terminal and also helps to access the local files of the computer.

**Some of the hyper terminal commands**

1. **pwd –** print working directory (used to show the current location).
2. **cd –** change directory (used to change the path).
3. **mkdir –** make directory (create a folder).
4. **touch –** create a file.
5. **start –** open a file with desired application.
6. **Node filename –** execute the javascript file.

**NODE REPL (**Read Evaluation Print Loops)

Allows us to execute code in bite sized like Chrome developer Console.

1. Type **node** and click **enter.**
2. Write the code to be executed and **enter.**
3. in order to exit the current process type **.exit** or press **ctrl + C.**

**NATIVE NODE MODULES**

Libraries of code that are written by Node team to do useful work using Node. These are come with Node at the time of installation itself.

Example – File System

**const fs = require("fs");**

here the (“require”) keyword is used to acquire the file system (“fs”)modules.

**fs.copyFileSync("file1.txt","file2.txt");**

**copyFileSync –** keyword for copying from source(file.txt) to destination(file2.txt).

here if destination file doesn’t exist then it will create the file.

For External Node Modules we need to install the packages separately using **npm install** commands via terminal.

**EXPRESS**

Back-end framework for Nodejs. It makes back-end work more easy and avoid code repetition.

Steps to incorporating Express into project

1. install it using **npm install express** command in hyper terminal.
2. Head over to **server.js** file and require it by **const express = require("express");**
3. Create a new constant called app, and this is simply a function that represents the Express module, and we bind that to the word **app**. **const app = express();**

**CREATING THE FIRST SERVER**

const express = require("express");

const app = express();

app.get("/",function(request,response){

response.send("<h1>hello, world!<h1>");

});

app.listen(3000,function(){

console.log("server started listening to 3000");

});

**app.listen** tells app(express functionality) to listen on a specific port for any HTTP requests that get sent to our server. Here we choose port 3000 to listen for HTTP request.

**app.get** is a method that is provided by Express that allows us to specify what should happen when a browser gets in touch with our server and makes a get request.

The **first parameter** is the **location** of the get request which is represented by a forward slash**(“/”)**. So this is basically our **home page or home route**(localhost:3000/).

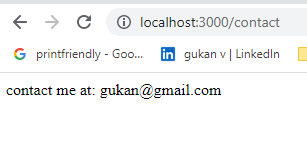
The **second parameter** is the **callback function** (can have two parameters: **request and response)** that tells the server what to do when that request happens.

**EXECUTION SEQUENCE**

1. when we typed in **localhost:3000**, we specified a **location of a server**. So when we hit enter,
2. the browser will go to that location and **make a request** to get some data back.
3. And when that request gets made at that home location, then **this callback function in the get method gets triggered**, and **we send the browser a response**, which is just the plain text of ‘Hello’.
4. Now that gets sent back to our browser and it renders it on screen.

**WORKING WITH ROUTES**

We can add multiple pages to our website using **app.get** with various routes.



app.get("**/contact**",function(req,res){

res.send("contact me at: gukan@gmail.com");

});

app.get("**/about**",function(req,res){

res.send("owner:gukan");

});

### **res.send([body])**

Sends the HTTP response. The**body** parameter can be a Buffer object, a String, an object, Boolean, or an Array.

### **res.sendFile(path [, options] [, fn])**

Transfers the file at the given path. **Path** must be an **absolute** path to the file.

For the absolute path, we use **\_\_dirname** – gives the path up to that folder where it is present(like cloud server or other computer).

res.sendFile(\_\_dirname + "/index.html");

console.log(**\_\_dirname**) **- F:\udemy\web development\Calculator**

**GETTING INPUTS FROM FORM**

<h1>calculator</h1>

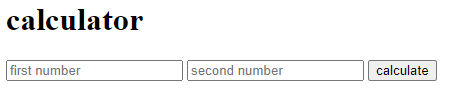
<form **action="/" method="post"**>

<input type="text" name="**num1**" placeholder="first number">

<input type="text" name="**num2**" placeholder="second number">

<button type="submit" name="submit">calculate</button>

</form>



The **method** is the **post** method, so we are sending this **form data(**here it is input values **num1 and num2)** somewhere, and that somewhere is what is defined by the **action attribute** here it is home route “**/**”(by default if it doesn't exist, then the form will simply send the data to the current page where it's on).

**app.post** method is to handle any post requests that come to our home route and sends information about that post request which includes **form data**.

**app.post**("/",function(req,res){

res.send(“thanks for posting”); });

**BODY PARSER**

Allow us to parse the information that we get from the post request. We can tap into that information and use it for calculations using **body-parser package.**

1. **npm install body-parser** from the hyper terminal.
2. Require it using **const bodyParser = require("body-parser");**
3. in order to use it **app.use(bodyParser.urlencoded({extended: true}));**

**app.use** method to use body parser which is works with Express.

Body parser has few modes

**bodyParser.text** – parse all the request into text.

**bodyParser.json** – parse all the request into json format.

**bodyParser.urlencoded** - parse data that comes from an HTML form.

**extended: true –** allows us to post nested objects.

1. Tap the data from post request using **req.body** and use it for calculation.

var num1 = **Number**(**req.body.num1**); //*Number – converts string into number*.

var num2 = Number(**req.body.num2); *//parseFloat -*** *The parseFloat() function is used****to accept a string and convert it into a floating-point number****. If the input string does not contain a numeral value or If the first character of the string is not a number then it returns NaN i.e, not a number.*

var result = num1 + num2;

**console.log(req.body); - **

**API – APPLICATION PROGRAMMING INTERFACE**

API is a set of commands, functions, protocols and objects that programmers can use to create software or interact with an external system(someone’s server).

API allows us to use someone’s data into our websites.

**FOUR IMPORTANT THINGS FOR API**

**Endpoints, Paths, Parameters, Authentication.**

**Endpoints -** Every API that interacts with a external system, like a server, will have an endpoint. An endpoint is an access point to send the HTTP requests to and get your response.

**Eg -** [**https://sv443.net/jokeapi/v2/**](https://sv443.net/jokeapi/v2/)

**Paths -** API paths and parameters that you can use in order to narrow down on a specific piece of data you want from an external server.

**Eg -**https://v2.jokeapi.dev/joke/**Programming** – Programming Path

[https://v2.jokeapi.dev/joke/**Pun**](https://v2.jokeapi.dev/joke/Pun) **-** Pun Path.

**Parameters -** parameters are like the settings or the configuration of an HTTP request.  
The parameters need to be prefixed by a single question mark (?) and separate key/value pairs need to be delimited from another by an ampersand (&). Keys are separated from values with an equals sign (=).



**Authentication -** Every time you make a request through the API, they must be able to identify you as the developer, and they have to keep track of how often you're using their server to get data, and then charge you, or limit you, accordingly.

**JSON -**  **Javascript Object Notation.**

**XML – Extensible Markup Language.**

Both JSON and XML can be used to receive data from a web server.

**Similarities**

* Both JSON and XML are "self-describing" (human readable)
* Both JSON and XML are hierarchical (values within values)
* Both JSON and XML can be parsed and used by lots of programming languages
* Both JSON and XML can be fetched with an XMLHttpRequest.

**Differences**

* JSON does not use end tag
* JSON is shorter
* JSON is quicker to read and write
* JSON can use arrays
* JSON can be parsed by a standard JavaScript function.
* XML has to be parsed with an XML parser.

**GET REQUEST WITH HTTPS MODULES**

**https.get** is native module which is used to fetch some data from external server to use in our application.

const express = require("express");

**const https = require("https");**

const app = express();

app.get("/",function(req,res){

**const url = "https://api.openweathermap.org/data/2.5/weather?q=cuddalore&appid=260d4d1c843ce91c505578ae957421e8&units=metric";**

**https.get(url,function(response){**

**response.on("data",function(data){**

**const weatherData = JSON.parse(data);**

**const temp = weatherData.main.temp;**

**const description = weatherData.weather(0).description;**

**const icon = weatherData.weather[0].icon;**

**const imgUrl = "https://openweathermap.org/img/wn/" + icon + "@2x.png";**

**res.write("<p>The weather is currently "+ description + "</p>");**

**res.write("<h1>THe temperature in cuddalore is "+ temp +"degrees Celcius.</h1>");**

**res.write("<img src="+imgUrl+">");**

**res.send()**

**});**

**});**

});

**Response.statusCode**

**200 –** request successful and resources transmitted into message body.

**401 –** unauthorized request. (because API request must have authorized API ID )

**404 –** resources not found. (error in API url).

 response.on('data',callback(){}) listens for the data event which is fired when we get data. Data which tapped from **response.on** is in hexadecimal

string. We can convert this string into text (javaScript object) using **JSON.parse(data).**

We can convert object into string using **JSON.stringify(data).**

we can also tap into the response that we get back, and call a method called **on**, and search through it for some data that will correspond to the actual message body that we got back, that OpenWeatherMap has sent us.

For one **app.get** method there should be only one **res.send()** but can have many **res.write()** for many multiple lines in website.

**app.use(express.static(“public”));**

Express only serve-up the file(.js) along with some folders like **views(ejs),** etc. For our server to serve up static files such as CSS(stylesheet), pure javaScript file and images, then we need to use a special function of Express called **static**. So we need to say **app.use(express.static(“public”)).** and inside the parentheses we are going to add the name of a folder that we are going to keep as our static folder.

app.post("/", function(req,res){

const firstName = req.body.fname;

const lastName = req.body.lname;

const email = req.body.email;

**const data = {** data that needs to be send to mailchimp server which

**members: [** should be in JSON format, so we first write the data

**{** in JAVASCRIPT object and then converted into JSON

**email\_address: email,** using **json.stringfy();**

**status: "subscribed",**

**merge\_fields:{**

**FNAME: firstName,**

**LNAME: lastName**

**}**

**}**

**]**

**};**

**const jsonData = JSON.stringify(data);**

**const url = "https://us12.api.mailchimp.com/3.0/lists/b2ac4856ce";**

**const options = {**

**method:"POST",**

**auth:"gukan1:bc1cd3529bd2efe310452d135bbf68e9-us12"**

**}**

**const request = https.request(url, options, function(response){**

**if (response.statusCode ===200){**

**res.sendFile(\_\_dirname + "/success.html")**

**} else {**

**res.sendFile(\_\_dirname + "/failure.html")**

**}**

**response.on("data",function(data){**

**console.log(JSON.parse(data));**

**})**

**})**

**request.write(jsonData);**

**request.end();**

**});**

**https.request(url, options, function(response){});**

**https.request** used to post the data into mailchimp server. In addition to **url**, we can add **options**(in that we can add **method** which allows us to specify the type of request made and **auth: “<any string>:API key”** used for basic authentication).

**Request.write(jsonData);** used to send the data into mailchimp server and **request.end();** is to indicate the end of that request.

**App.js failure.html**

app.post("/**failure**",function(req,res){ <form action="**/failure**" method="**post**">

**res.redirect("/");**  <**button** class = "btn btn-lg btn-warning" }); type="submit" name="button">Try

again</button>

</form>

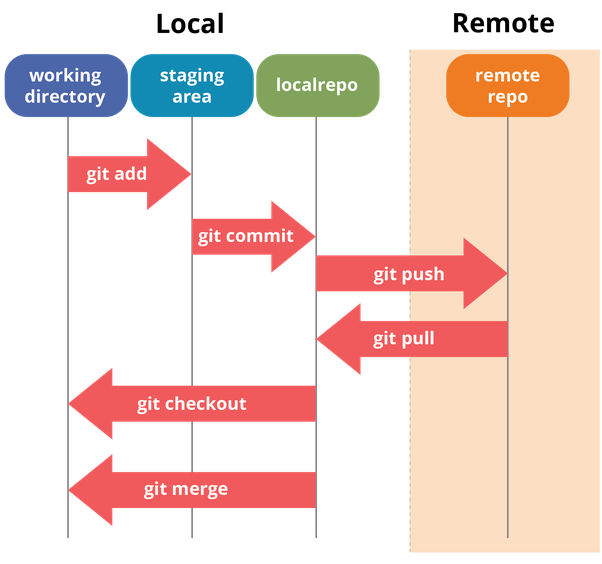
When I click this **button** that says try again, we're going to trigger a **POST** request to

the failure route(**/failure**) and that is going to be caught by our server over here

and it's going to **redirect to the home route(res.redirect("/");)**, which triggers this app.get and it sends the signup page as the file to be rendered on screen.

**GIT – Global Information Tracker**

Version control used to keep track of changes made at different period of time.

****

The **working directory** is the folder or the directory where you initialize your Git repository.

Git is going to track the changes that it sees between the working directory

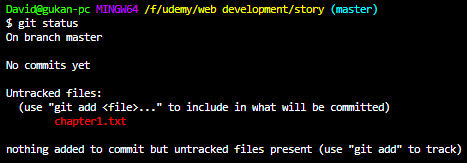
and the **local** **(git)repository**. The **staging area** is a good place to figure out what are the things that you want Git to ignore and what are the things that you want to be tracked. So once we've used **git add**(**command**) we've put all file into the staging area and when we're assure with the changes that we are going to commit, then go ahead and commit it using the **git commit command** inside our local(git) repository. The **remote repository** is the repository inside the **GitHub.**

**Command line version control**

**git init –** to initialize the empty git repository.

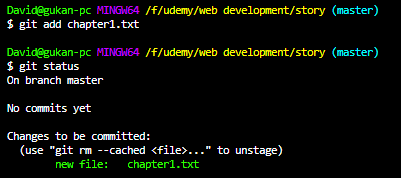


**git status –** to see whether the files are included in the staging area or not (for keep track of that file).



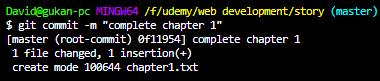
Here the chapter1.txt(red colour) indicates the untracked files.

**Git add <file name> OR git add .(**for all files in the repository**) –** to add the files to staging area where it is tracked.

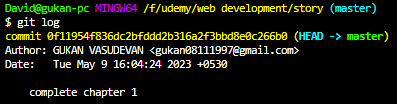


Here the chapter1.txt(green colour) indicates the tracked files.

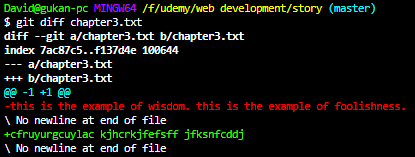
**Git commit -m “message” –** to commit the changes made in the file(present in staging area) to git repository.

****

**git log –** to see what are the files and when the files are committed.



**git diff <file name> -**  use to check out one of the differences(in changes made in file) between the current version of chapter 3 and the last save point in our Git repository.



Statement(red colour) changed to statement(green colour).

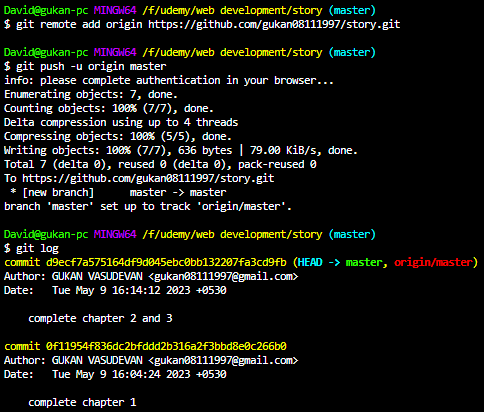
**Git checkout <file name> -** use to rollback from current version of the file to the previous versions.



**COMMIT THE FILES TO REMOTE REPOSITORY**

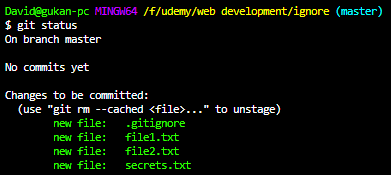
**git remote add origin** [**https://github.com/gukan08111997/story.git**](https://github.com/gukan08111997/story.git) **-**used to create remote repository with name **origin** and the URL of our remote repository on GitHub.

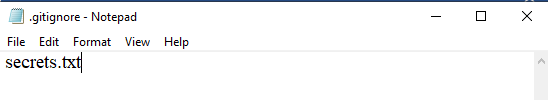
**Git push -u origin master –**  it **pushes** your local repository to the remote repository (called **origin/master branch** is the main branch of all of your commits.) using the **u** flag which basically links up your remote and local repositories.

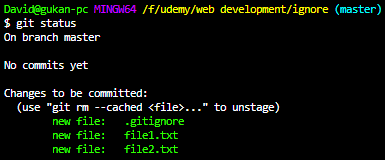


**GIT IGNORE** used for safe-guarding the secret information from others. used to indicate that what are the files that git wants to ignore.

1. Create a file called **.gitignore**  in working directory
2. In that write the **file names (**that should be ignored) and save it. Here we use (**#)** for comments and (\*.txt) for all files with that extension.

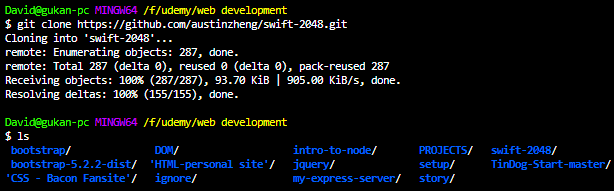






**GIT CLONE**

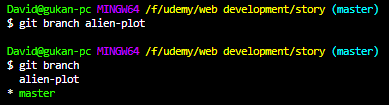
Used to pull down all the commits and all the versions of a particular remote repository and to store the files inside your working directory.



**BRANCHING AND MERGING**

**git branch <name of the branch> -** used to create branches.

**git branch –** to see the name of the branches and also see which branch we are currently on.

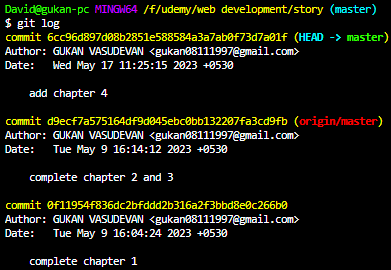
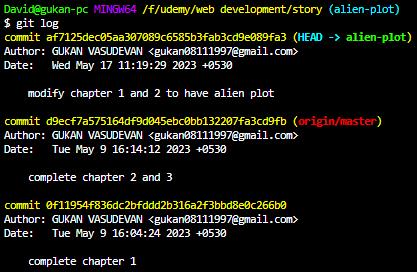


Here branch (green in colour with asterisk) indicates the branch we are currently on.

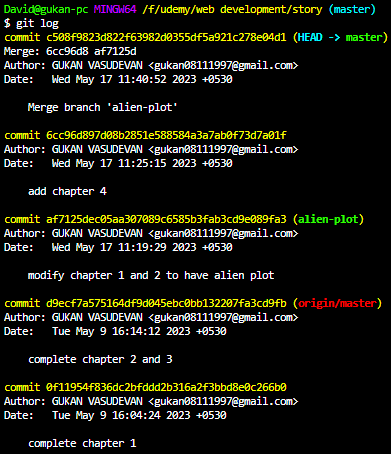
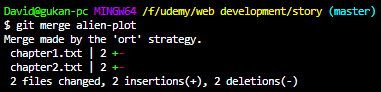
**git checkout <name of the branch> -** to switchover the branches.



For the particular branch **git log** shows only the commits made within that branch.



**git merge <name of the branch> -** used to merge one branch to another branch.



**git log** after merging with master branch 🡪

here we can push this commits to remote using **git push origin master -u.**

**GIT FORKING**

we are basically just copying a repository that is hosted on GitHub and we are keeping the copy under our own GitHub account where we can make changes to it.

So, this is called forking. When you fork a remote repository, you have full permissions to do whatever you want. Then you contribute to the repository(owner) by making a **pull request**.

**EJS -** is a simple templating language that lets you generate HTML markup with plain JavaScript.

var today = **new Date(); -** this method gives the current date.

var currentDay = today.**getDay(); -**  this method gives day of the week in **number(0-6),** 0- Sunday, 1-Monday, 2-Tuesday, . …………, 6-Saturday.

1. Install EJS module with command in hyper terminal – **npm i ejs.**
2. then tell our **app** to use ejs using **app.set('view engine', 'ejs');**
3. **res.render("list",{ kindOfDay : day}); -** this method res.**render** uses **view engine** to render a particular page(**list.ejs**). here we are rendering a page called list.ejs which is inside the folder called **views** because view engine by default look for folder **views.**
4. Inside list.ejs we want specify the **EJS** **marker 🡪 <%= kindOfDay %> marker** tells the file to replace whatever is inside here with the value of the variable.

**Running code inside EJS template**

**<%** if (kindOfDay === "Saturday" ||kindOfDay === "Sunday"){ **%>**

<h1 style="color: blue"><%= kindOfDay %> list</h1>

**<%** } else { **%>**

<h1 style="color: purple"><%= kindOfDay %> list</h1>

**<%** } **%>**

**<% 🡪** scriptlet tag used for control flow statements. Need to specify non html code using this scriptlet tags line by line.

**Passing data from our webpage to our server**

var **options**={

weekday:"long",

day:"numeric",

month:"long"

};

var day = today.**toLocaleDateString("en-US",options); 🡪** converts a date to a string, returning the date (using operating system’s locale’s conventions) portion with specified format mentioned in **options.**

**Concept of scope in context of javascript**

We can create variable using three keywords namely – **var, let, const.**

**Local variable –** variables declared within the function/control statements/simply within curly braces can be accessed only within that function/block of code.

**Global variable –** variables declared outside the function/control statements/globally can be accessed anywhere else inside the code file.

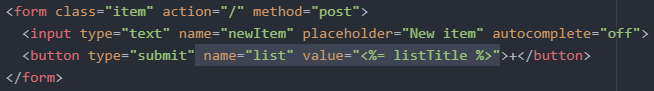
**Var, let –** value declared once can be changed in future.

**Const –** value once declared can’t be changed in future.

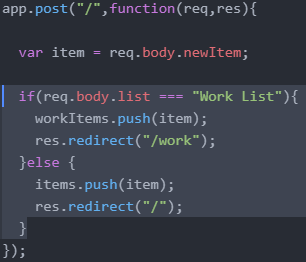
|  |  |  |  |
| --- | --- | --- | --- |
|  | **declared inside the function** | **Declared inside the control statements** | **Declared outside curly braces** |
| Var | Access local only | Access **global** | Access global |
| Let | Access local only | Access local only | Access global |
| Const | Access local only | Access local only | Access global |

**TEMPLATE VS LAYOUT**

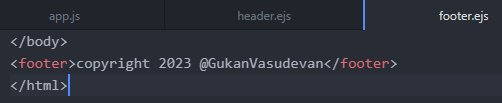
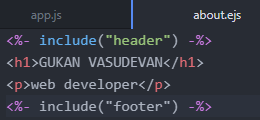
We can use same **list.ejs** for rendering multiple pages by making simple logic as follows.

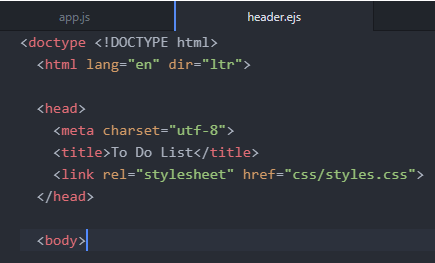
In list.ejs file, make **name(**any name**)** and **value(ejs marker )** of button (because bodyParser catches those information) 

In app.js file, make some logic in app.post() so that posting different data to different route with same list.ejs file.



**LAYOUTS**

In many situations we use the same layout for multiple pages in our website. In order to reduce the code(responsible for layout design namely **header and footer**) repetition, we can save the code(which is repeated) in separate files(**header.ejs and footer.ejs**) in **views** folder and include it in different pages as follows.

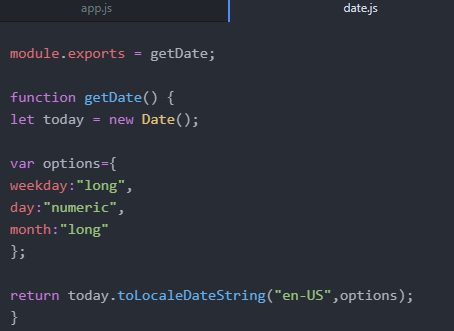


**NODE MODULES EXPORTS**

Code that is not strictly related to the route should not be in cluttering the app.js. we can solve this with node module exports.

**Steps to incorporate modules**

1. Create a file with **.js** extension(date.js) and inside it makes **function** that returns desired result from this module.
2. We can exports this functionality by binding this function to the **module.exports** in the **date.js** and **require** it in **app.js (**since this module is not from npm but is in our local directory we want to use \_\_dirname) using **const date = require(\_\_dirname + "/date.js"); 🡪** here we bind this date.js functionality to **date** word.

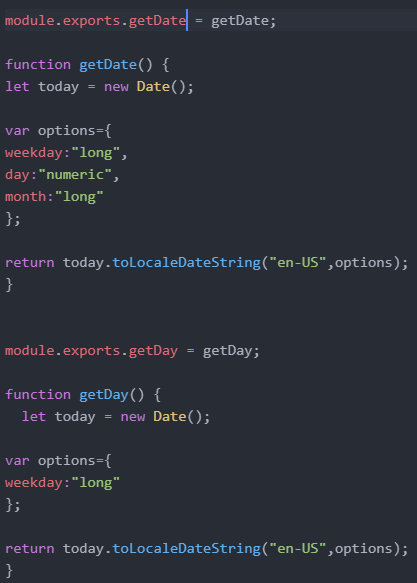
****we call a function with getDate(); but here we call a function **getDate;** because it executes the function straightaway if we include those **parenthesis()** which does not allow app.js when to execute the function.

1. In app.js we simply use this module by calling its name. it will show only the function corresponds to that module but when we include **()**  then it will execute the function and gives the result.

**console.log(date); **

**console.log(date());**

if the module has more than one function then

we want to separately export it and we can tap into that specific function which is in the module by

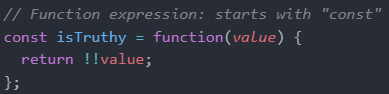
**console.log(date.getDate());**



**console.log(date.getDay());**

****

**Code refactoring**

****Here we can omit **module** since **module.exports = exports.**

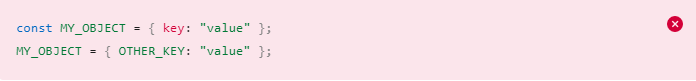
We can declare the function using

**function expression**

****

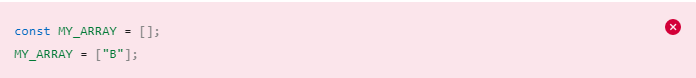
**Const in objects and arrays**

const also works on objects and arrays. **Attempting to overwrite the object throws an error "Assignment to constant variable".**

****

However, **object keys are not protected**, so the following statement is executed without problem.

****

**Assigning a new array to the variable throws an error** "Assignment to constant variable".

Still, it is **possible to push items into the array** and thus mutate it.

****

**ALTERNATE FORLOOP – forEach()**

The forEach() method calls a function for each element in an array. It is not executed for empty elements.

Here forEach() method will execute the function three(no. of items in fruits array) times until all the elements in the array gets logged in the console.

Const fruits = [“apple”,”mango”,”lemon”]; **output**

fruits.**forEach**(function(fruit){ apple

console.log(fruit); mango

}); lemon

**ROUTE PARAMETERS**

Route parameters are named URL segments that are used to capture the values specified at their position in the URL. The captured values are populated in the **req.params** object, with the name of the route parameter specified in the path as their respective **keys.**

Route path: /users/**:userId**/books/**:bookId**

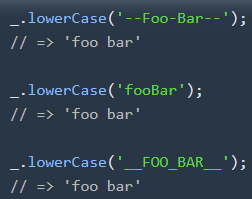
Request URL: http://localhost:3000/users/**34**/books/**8989**

req.params: { "userId": "34", "bookId": "8989" }

here the **req.params.userId** will give the value captured in theURL.

**LODASH MODULE**

1. Install it with hyper terminal using command – **npm install lodash.**
2. Require it in app.js using - **const \_ = require("lodash");**
3. Use it’s functionality to get desired result.



**Substring()**

The substring() method extracts characters, between two indices (positions), from a string, and returns the substring.

let text = "Hello world!"; **output**  
let result = text.substring(1, 4); ell

**DATABASES**

|  |  |
| --- | --- |
| **SQL – Structure Query Language** | **NOSQL – Not only SQL** |
| More mature – old technology | Shiny and new – new technology |
| table structure – like excel sheets | Document structure – JSON objects |
| Require a schema (predetermined table structure). Inflexible to changes in schema | More flexible to changes in structure since it JSON objects. |
| Great with relationships (between parent and child tables) | Not great with complex relationship |
| Scale vertically – can only add data in excel sheets which gets slowdown after certain no. of rows. | Scale horizontally – distributed storage |

**SQL –** refer w3schools website

**MongoDB**

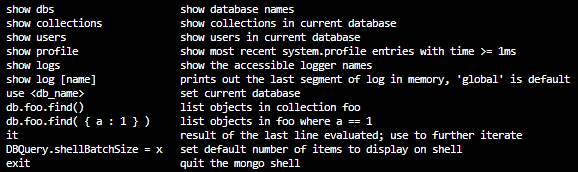
Popular NoSQL database system

**Steps to incorporate MongoDB in hyper terminal**

1. Download and install MongoDB server. In order to launch using hyper terminal by following the simple steps:
   1. Create a file called **.bash\_profile** and open the file with **vim** editor.
   2. Copy the app path and paste it inside the .bash\_profile, then save and quit it.
   3. Reopen the hyper terminal you will see the MongoDB version using command **mongo –version.**
2. In order to spin up the database – open hyper and type **mongod**.
3. Open another tab in hyper terminal and type **mongo –** in order to open the mongo shell.

**Some of the useful commands**

**help –** shows what are possible commands we can use.





* + Show the currently available database names.
  + use to create database or switchover to database.



* + show the current database.



* + To create collections in db



🡪shows the collections in the current database.

**CRUD OPERATIONS**

Create or insert operations add new **documents(**like single row **record** in SQL **)** to a **collection(**like **tables** in SQL**)**. If the collection does not currently exist, insert operations will create the collection.

To insert - db.collection.insertOne() 🡪 one document insert



db.collection.insertMany()🡪 many document insert.

**Read operation -** Read operations retrieve documents from a collection; i.e. query a collection for documents. MongoDB provides the following methods to read documents from a collection:

**db.collection.find()**



Here query criteria(optional) is condition(similar to where condition in SQL) to find the documents and projections(optional) is to indicate which fields(similar to columns in SQL) we want.

**Read without query and projections**

****

**Read with query criteria only**

select \* from products where name = “pencil”

select \* from products where price > 1.

**Read with query and projections**

select name from products where price>1

**by default id will show in all find(), if we specify \_id:0 then it won’t show up(0 – false; 1 - true)**



select name from products where price>1

here (\_id:0), so id didn’t show up.

According to the documentation, first argument in find is filter and second is projection. projection allows you to specify fields to return.\_id is the only field which you need to explicitly exclude in the projection. For all other fields you just need to state the inclusion. You will have to follow the below format.

db.posts.find({}, {title: 1, body:1, category:1, likes:1, tags:1})

**Update operation**

Update operations modify existing documents in a collection. MongoDB provides the following methods to update documents of a collection:

db.collection.updateOne()

db.collection.updateMany()

db.collection.replaceOne()





Like 🡪**update products set stock = 32 where id =1;**

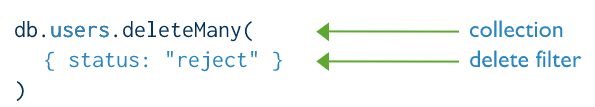
Here the first field inside the parentheses is **filter fields**(which document to be updated) and second field is **value(**what to be updated**).**

**Delete operation**

Delete operations remove documents from a collection. MongoDB provides the following methods to delete documents of a collection:

db.collection.deleteOne()

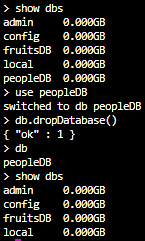
db.collection.deleteMany()





**Drop database operation**

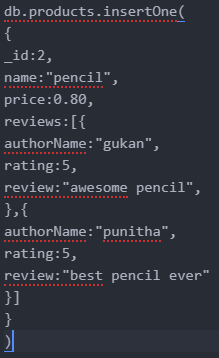
In hyper terminal switch to database using **use <database>** and type **db.dropDatabase()**



**DROP A COLLECTION**

Be sure that you are inside the database where the collection is present and then execute the following command **db.collectionName.drop().**

**Relationships**

here **one to many** relationship

one product will have multiple reviews.

We can make node application which is work along with mongo DB using two ways

* Native Node driver – it is tedious because it includes lot of validation codes.
* Mongoose – simplified usage.

**MONGOOSE**

Called as ODM (Object Document Mapper) will allow your Node.js app which speaks the language of Javascript objects to be able to talk to your MongoDB database which speaks in the language of documents and collections and databases.

The main objective of this framework is to simplify the writing of validation code, the writing of your business logic boilerplate and just to make the code just shorter and easier to work with MongoDB.

**Steps to incorporate mongoose using node.js**

1. Install mongoose using hyper terminal – **npm install mongoose.**
2. Require it in app.js using - **const mongoose = require("mongoose");**
3. Connect to MongoDB database using - **mongoose.connect("mongodb://127.0.0.1:27017/fruitsDB"); -** here 127.0.0.1 🡪 localhost address; 27017 🡪 port in which MongoDB server listening; fruitsDB 🡪name of database to create or to connect.
4. In order to insert some data into database

* Create a Schema which is the structure of our data that we are going store inside database –

**const fruitSchema = new mongoose.Schema({**

**name:String,**

**rating:Number,**

**review:String**

**});** here this schema tells how we want data in a particular collection to be structured.

* Create a mongoose model which tells the **fruits** collection to use the **fruitSchema** – **const Fruit = mongoose.model("Fruit",fruitSchema);** here “Fruit” is name of collection in singular form because mongoose convert it plural form using lodash.
* Create a data(document) to be inserted -

**const fruit = new Fruit({**

**name:"Banana",**

**rating:8,**

**review:"healthy Banana!"**

**});**

And save it using **fruit.save(); 🡪** this calls the save method in Mongoose to save this fruit document into a fruit collection inside our fruitsDB.

* For inserting multiple document

**const banana = new Fruit({**

**name:"Banana",**

**rating:8,**

**review:"healthy Banana!"**

**});**

**const orange = new Fruit({**

**name:"Orange",**

**rating:8,**

**review:"healthy Orange!"**

**});**

**const mango = new Fruit({**

**name:"Mango",**

**rating:9,**

**review:"tasty mango!"**

**});**

**Fruit.insertMany([banana,orange,mango]);**

**READING FROM DATABASE WITH MONGOOSE IN NODEJS**

We can read from the database using two methods

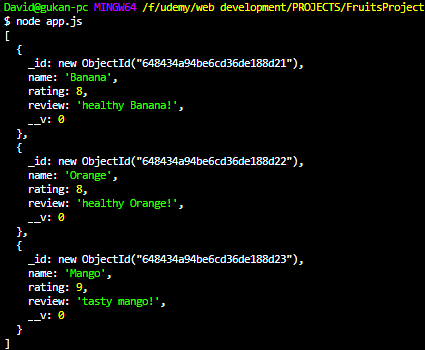
1. **Asyn function**

const mongoose = require("mongoose");

main().catch(err=>console.log(err));

**async function main(){**

mongoose.connect("mongodb://127.0.0.1:27017/fruitsDB");

****

const fruitSchema = new mongoose.Schema({

name:String,

rating:Number,

review:String

});

const Fruit = mongoose.model("Fruit",fruitSchema);

**const fruits = await Fruit.find();**

**console.log(fruits);**

**}**

1. **Using Promises(then)**

const mongoose = require("mongoose");

mongoose.connect("mongodb://127.0.0.1:27017/fruitsDB");

const fruitSchema = new mongoose.Schema({

name:String,

rating:Number,

review:String

});

const Fruit = mongoose.model("Fruit",fruitSchema);

**Fruit.find().then((fruits)=>{**

**console.log(fruits);**

**});**

Here we can filter the **find()** using two ways

1. Fruit.find(**{name:"Banana",rating:{$gt:6}}**).then((fruits)=>{

console.log(fruits);

});

1. Fruit.find().then((fruits)=>{

console.log(**fruits[0]**);

});



CLOSING THE CONNECTION USING – **mongoose.connection.close();**

**DATA VALIDATION WITH MONGOOSE**

Used to check the correctness of data entered and it is defined in schema.

const fruitSchema = new mongoose.Schema({

name:{

type:String,

**required:true 🡪this prevents from saving data without name field.**

},

rating:{

type:Number,

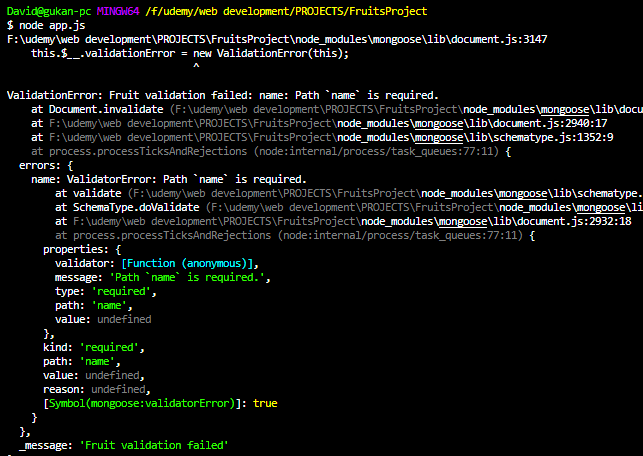
**min:1, 🡪 this prevents from saving data with rating beyond the min and max values.**

**max:10**

},

review:String

});



**UPDATING DOCUMENT IN MONGOOSE**

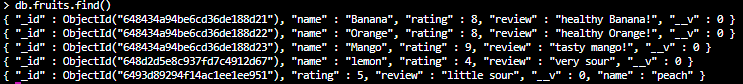
**Fruit.updateOne({\_id:"6493d89294f14ac1ee1ee951"},{name:"peach"}).then((fruits) =>{ console.log(fruits);**

**});**

**[OR] using async function**

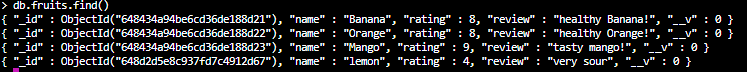
**await Fruit.updateOne({\_id:"6493d89294f14ac1ee1ee951"},{name:"peach"})**

Here we updating **name field** in a document with **\_id** (which document we want to update)



**DELETING DOCUMENT**

**Fruit.deleteOne({name:"peach"}).then();**

****

**Fruit.deleteMany({name:"peach"}).then(); -** delete all the documents with the name – peach.

**ESTABLISHING RELATIONSHIP AND EMBEDDING DOCUMENTS**

const mongoose = require("mongoose");

mongoose.connect("mongodb://127.0.0.1:27017/fruitsDB");

const fruitSchema = new mongoose.Schema({

name:{

type:String,

},

rating:{

type:Number,

min:1,

max:10

},

review:String

});

const Fruit = mongoose.model("Fruit",fruitSchema);

const pineapple = new Fruit({

name:"Pineapple",

rating:9,

review:"Great fruit"

});

pineapple.save();

const personSchema = new mongoose.Schema({

name: String,

age: Number,

**favouriteFruit: fruitSchema**

});

const Person = mongoose.model("Person",personSchema);

const person = new Person({

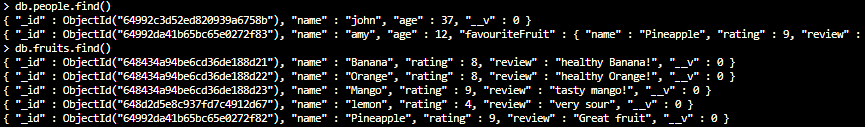
name:"amy",

age:12,

**favouriteFruit:pineapple**

});

person.save();



Here we establish relationship between people collection and fruits collection.

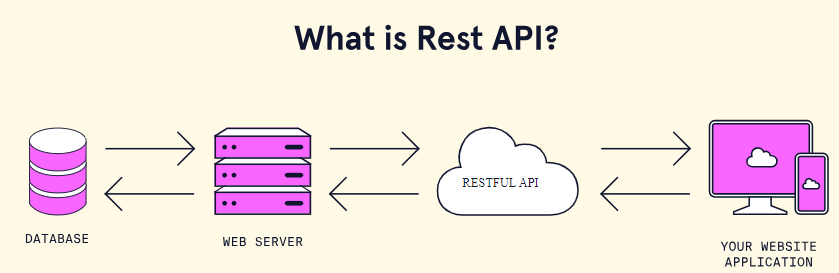
**John in people collection have favouriteFruit field which have fruitSchema.**

Person.updateOne({name:"john"},{favouriteFruit:pineapple}).then((people)=>{console.log(people);});

Here we update john in people collection to have favouriteFruit.

**REST API**

**RE**presentational **S**tate **T**ransfer



API is a set of defined rules that enable different applications to communicate with each other. It also indicates what are the services(resources) that server can give to clients similar to **menu card** in restaurant.

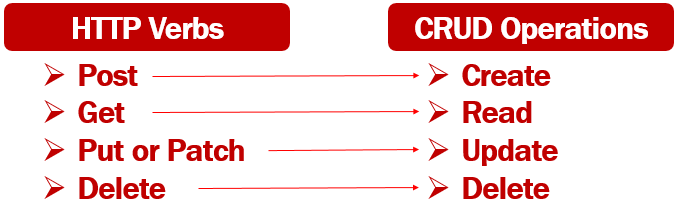
REST is simply an architectural style for designing an API.

Two important rules for making an API RESTful

* Using HTTP Request verbs
* Using a specific pattern of Routes/Endpoint URLs.

Using HTTP Request verbs

GET, POST, PUT, PATCH and DELETE



For updating a one thing, PUT – Update the whole thing but PATCH – update only a thing which needs to be updated.

Using HTTP Request Route

|  |  |  |
| --- | --- | --- |
| HTTP verbs | /articles | /articles/new |
| GET | Fetches **all** the articles | Fetches **the** article on new |
| POST | Create **one** new article | - |
| PUT/PATCH | - | Updates **the** article on new |
| DELETE | Deletes **all** the articles | Deletes **the** article on new |

GET - FETCH ALL THE ARTICLES

**app.get**("/articles",function (req,res){

Article.**find().**then((foundArticles)=>{

res.send(foundArticles);

});

});

POST A NEW ARTICLE

**app.post**("/articles", function(req,res){

const newArticle = new Article({

title:req.body.title,

content:req.body.content

});

newArticle.**save().**then(()=>{

res.send("successfully saved to database");

});

});

DELETE ALL DOCUMENTS

**app.delete**("/articles",function(req,res){

Article.**deleteMany().**then(()=>{

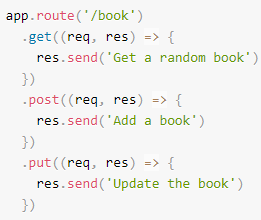
res.send("successfully deleted all the documents");

});

});

**app.route()**

You can create **chainable route handlers** for a route path by using **app.route().** Because the path is specified at a single location



Here get(), post(), put() methods have same route path **/book.**

**CONNECTING AND ACCESSING TO PRE-EXISTING MONGODB DATABASE**

const {MongoClient} = require("mongodb");

const url = "mongodb://127.0.0.1:27017/task";

const client = new MongoClient(url);

const database = client.db("task");

const bandwidths = database.collection("bandwidth");

app.get("/",async function(req,res){

const bandwidth = await bandwidths.findOne({bandwidth:394});

console.log(bandwidth);

});

GET A SPECIFIED ARTICLE

**app.route("/articles/:articleTitle")**

.**get**(function(req,res){

const articleTitle = req.params.articleTitle;

**Article.findOne({title:articleTitle}).then((article)=>{**

if (article){

res.send(article);

} else {

res.send("No articles found with the searched title");

}

});

});

PUT A SPECIFIED ARTICLE update an entire document

.**put**(function(req,res){

**Article.findOne({**title:req.params.articleTitle})**.replaceOne**({title:req.body.title,content:req.body.content}).then(()=>{

res.send("successfully updated");

})

});

PATCH A SPECIFIED ARTICLE update a selected field value in specified document

**.patch**(function(req,res){

**Article.updateOne**({title:req.params.articleTitle},**{$set:req.body}).**then(()=>{

res.send("successfully updated");

})

})

DELETE A SPECIFIED ARTICLE

**.delete**((req,res)=>{

**Article.deleteOne**({title:req.params.articleTitle}).then(()=>{

res.send("successfully deleted");

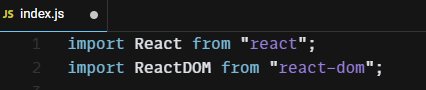
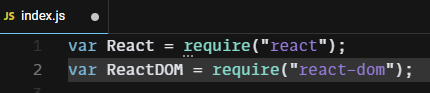
})

});

**REACT JS**

Front end framework/library, Project creation–**npx create-react-app <folder-name>**

React lets you build user interfaces out of individual pieces called components. Create your own React components like Thumbnail, LikeButton, and Video. Then combine them into entire screens, pages, and apps.

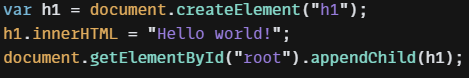
 **ES6 Common JS**

**React** module use **babel (**Babel is a JavaScript compiler that converts modern JavaScript code into a version compatible with all browsers. Babel enables React developers to use the latest JavaScript syntax in their components. Babel transpiles modern JavaScript for use in React components and all browsers.)

Here the below two codes perform the same function but **ReactDOM** use modern javascript format called **JSX(**acquired from react module**)** which enables to embed HTML element inside javascript.

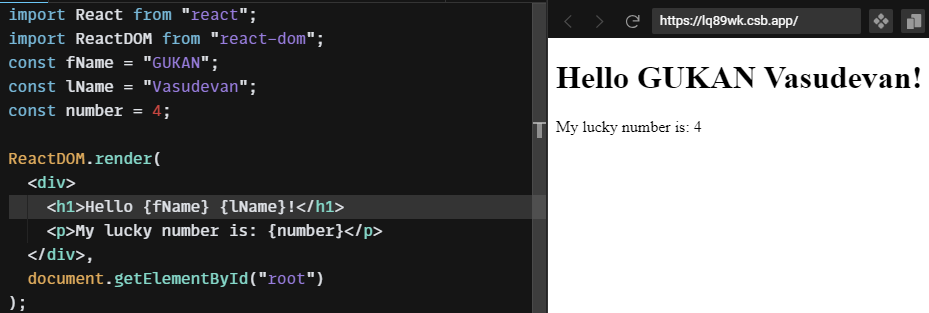
This **render** function uses three inputs (WHAT TO SHOW, WHERE TO SHOW IT, CALLBACK FUNCTION)





**Javascript expressions with JSX**

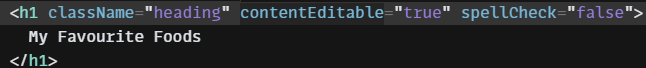
using JSX we can embed HTML inside javascript code and embed javascript (only expressions (piece of code that resolves to a value) but not statements (if-else, while loop, etc)) inside HTML.





**JSX attributes and styling React elements**

In **normal HTML elements** the attributes should be **all lowercase** but in **JSX format** attributes must be javascript method in **camel casing** as follows

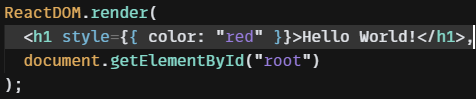


**In JSX** format syntax should be strict for example (**<img /> OR <img></img>)** should include **alt attribute** to avoid warnings.

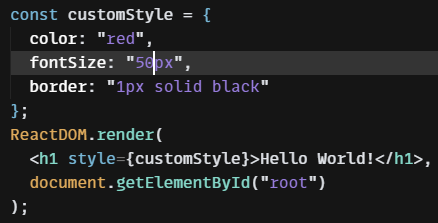


<img className="image" alt="random" src="https://picsum.photos/200" />

**INLINE STYLING IN JSX**

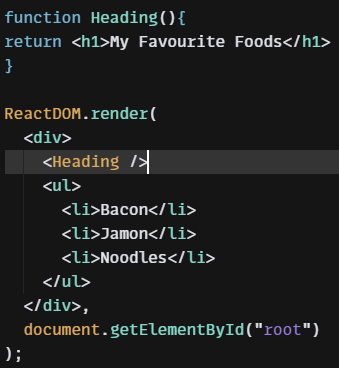


in JSX the inline styles require CSS property to be in **javascript object**. Here the two curly braces (one for javascript objects and other for placing javascript code inside HTML element)

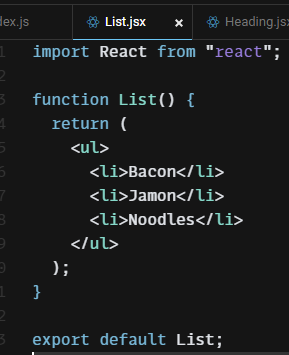
in CSS the properties should be **kebab case (font-size)** but in Javascript object the properties should be **camel casing.**

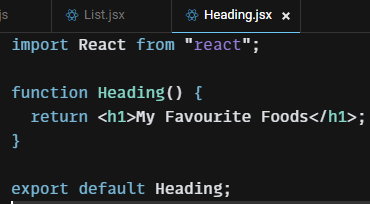
**REACT COMPONENTS**

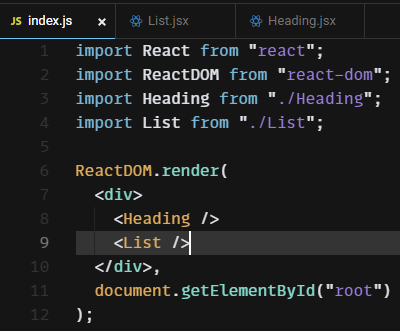
Components allow us to split up a large file or a complex web structure into smaller components and allow us reusing each of these components when we need the same functionality.



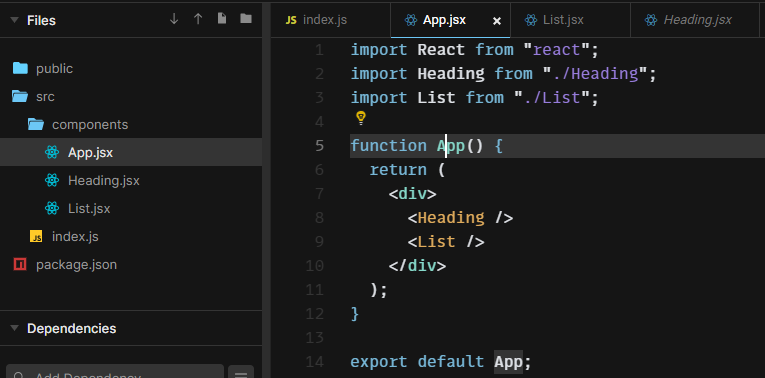
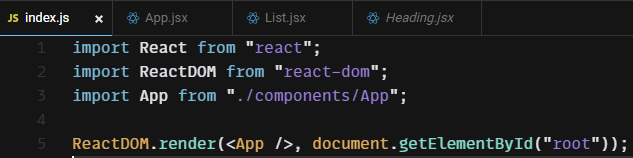
We can reuse same components using **import and export**





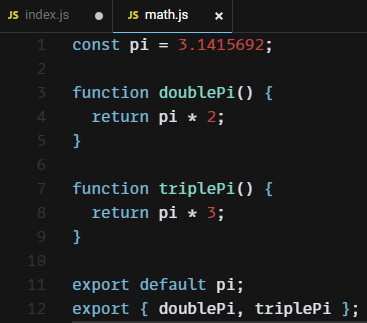


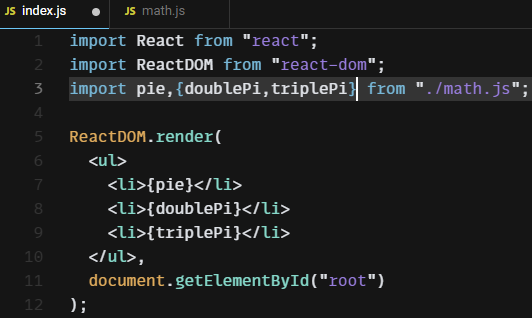
Grouping all components to components folder

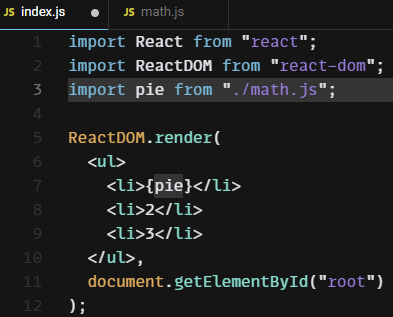


**IMPORT EXPORT MODULES**

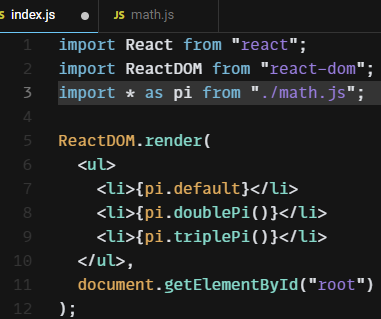
Help to modularize large javascript file into small modules/components and import it in other file.



Import the **default** **pi** with any name Import other module with **same {pi ⬄pie} name {doublePi, triplePi}**

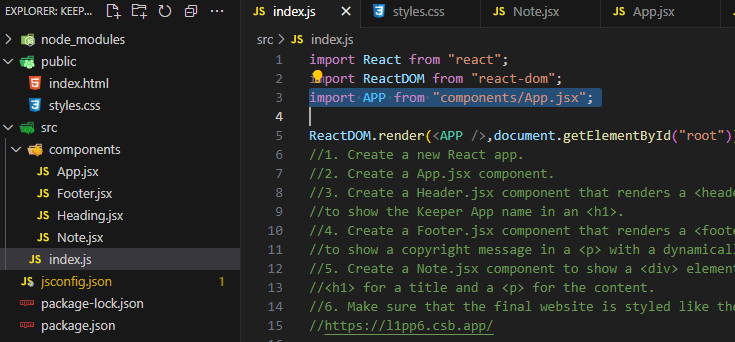


Import all modules with **wildcards**

****

**Import components** from the absolute path

If you're using Create React App, you can set up the absolute imports path in a **jsconfig.json** (need to create in a fresh JavaScript template) or **tsconfig.json** (already created in the TypeScript template) at the root of your project to serve your usage.

Example:

{

"compilerOptions": {

"baseUrl": "src"

},

"include": ["src"]

}

**React PROPS**

function Card(props) {

console.log(props);}

****

ReactDOM.render(<div>

<Card name=”Gukan” />

</div>,

document.getElementById("root")

);

import React from "react";

import ReactDOM from "react-dom";

function Card(props) {

console.log(props);

return (

<div>

<h2>{props.name}</h2>

<img src={props.img} alt="avatar\_img" />

<p>{props.tel}</p>

<p>{props.email}</p>

</div>

);

}

ReactDOM.render(

<div>

<h1>My Contacts</h1>

<Card

name="Beyonce"

img="https://blackhistorywall.files.wordpress.com/2010/02/picture-device-independent-bitmap-119.jpg"

tel="+123 456 789"

email="b@beyonce.com"

/>

<Card

name="Jack Bauer"

img="https://pbs.twimg.com/profile\_images/625247595825246208/X3XLea04\_400x400.jpg"

tel="+987 654 321"

email="jack@nowhere.com"

/>

<Card

name="Chuck Norris"

img="https://i.pinimg.com/originals/e3/94/47/e39447de921955826b1e498ccf9a39af.png"

tel="+918 372 574"

email="gmail@chucknorris.com"

/>

</div>,

document.getElementById("root")

);

**Props 🡪** custom attributes (name,img,tel,email) given to our custom React components(Card) are called properties/**props.**

**MAPPING DATA TO COMPONENTS**

The map function loops through this array of **contacts** and for every single item that exists in the array, it calls the **createCard** function and it passes over each of the objects inside the array to this function as parameter.

{contacts.map(createCard)}

function createCard(contact) {

return (

<Card

Key={contact.id}

id={contact.id}

name={contact.name}

img={contact.imgURL}

tel={contact.phone}

email={contact.email}

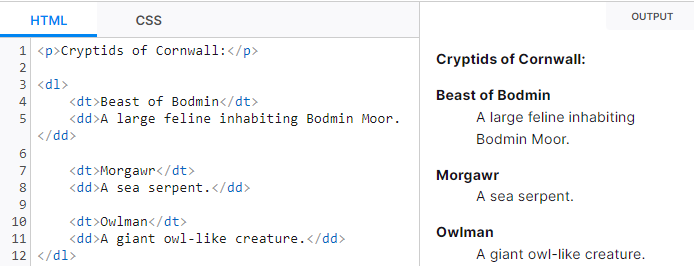
/> );

}

Each component in a list should have unique ID**(key)**, if React to be able to efficiently render components for every single component which is being rendered using loop such as **map function.** This **key** won’t show in **props** list and can’t access with props.

# <dl>: The Description List element

The **<dl>** [HTML](https://developer.mozilla.org/en-US/docs/Web/HTML) element represents a description list. The element encloses a list of groups of terms (specified using the [<dt>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/dt) element) and descriptions (provided by [<dd>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/dd) elements).



**MAP/FILTER/REDUCE**

Same with **map** and **forEach.**

var numbers = [3, 56, 2, 48, 5];

const newNumbers = numbers.map(function (x) {

return x \* 2;

});

////////////////////////forEach//////////////////////////////////

let newNumbers1 = [];

numbers.forEach(function (x) {

newNumbers.push(x \* 2);

});

console.log(newNumbers);

**CONSOLE **

**FILTER -** filter function is going to look through each of the numbers inside this numbers array and for each of these numbers, it's going to return only the ones that meet a particular criterion.

var numbers = [3, 56, 2, 48, 5];

const newNumbers = numbers.filter(function(num){

return num >10

});

////////////////////////forEach//////////////////////////////////

const newNumbers = [];

numbers.forEach(function (num) {

if (num > 10) {

newNumbers.push(num);

}

});

console.log(newNumbers); 

**REDUCE** - Accumulate a value by doing something to each item in an array.

var numbers = [3, 56, 2, 48, 5];

const number = numbers.reduce(function (accumulator, currentvalue) {

return accumulator + currentvalue;

});

console.log(number);

//////////////////////forEach///////////////////////////////////

let newNumber = 0;

numbers.forEach(function(currentNumber){

newNumber += currentNumber;

});

console.log(newNumber); 

**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](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined) is returned.

var numbers = [3, 56, 2, 48, 5];

const number = numbers.find(function(num){

return num > 10

});

console.log(number); 

**FIND INDEX -**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.

var numbers = [3, 56, 2, 48, 5];

const number = numbers.findIndex(function(num){

return num > 10

});

console.log(number); 

**ARROW FUNCTION IN ES6**

Simplifying the code for writing a function.

var numbers = [3, 56, 2, 48, 5];

const newNumbers = numbers.map(function square(x){

return x \* x;

});

var numbers = [3, 56, 2, 48, 5];

const newNumbers = numbers.map(function (x){

return x \* x;

});

Here we simplify by omit the function name 🡪anonymous function

var numbers = [3, 56, 2, 48, 5];

const newNumbers = numbers.map((x)=>{

return x \* x;

});

Here we replace the **function** keyword with **=>**(arrow function)

var numbers = [3, 56, 2, 48, 5];

const newNumbers = numbers.map( x => x \* x);

here we omit the **parenthesis, curly braces, return** keyword.

If function with two parameters, then parenthesis must be included

const newNumbers = numbers.map( (x, y) => x \* y);

if function with two parameters and multi-line expressions then parenthesis, curly braces, return keyword must be included

const newNumbers = numbers.map( (x, y) => { let z = 0;

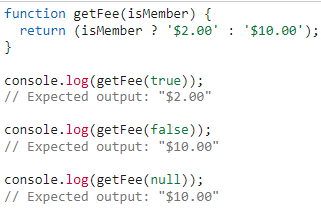
z = x \* y;

return(z\*z);});

# Conditional (ternary) operator

The **conditional (ternary) operator** is the only JavaScript operator that takes three operands: a **condition** followed by a question mark **(?)**, then an **expression** to execute if the condition is **true** followed by a colon (:), and finally the **expression** to execute if the condition is **false** This operator is frequently used as an alternative to an **if…else** statement.

**condition ? exprIfTrue : exprIfFalse**



if we want expression to be executed only the condition is true then

**condition ? exprIfTrue : null**

else use conditional AND operator as below.

**Conditional AND(&&) operator**

It works because in JavaScript, **true && expression** always evaluates to expression, and **false && expression** always evaluates to false.

Therefore, if the condition is true, the element right after && will appear in the output. If it is false, React will ignore and skip it.

{time >= 12 ? <h1>Why are you still working</h1> : null}

Both these lines do same functionality

{time >= 12 && <h1>Why are you still working</h1> }



**REACT STATE**

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**.

React components has a **built-in state object**. The state object is where **you store property values** that belong to the component. When **the state object changes**, **the component re-renders.**

**DECLARATIVE PROGRAMMING**

We are declaring when we are writing our code how our user interface should look under different conditions dependent upon the state.

**IMPERATIVE PROGRAMMING**

is a software development paradigm where functions are implicitly coded in every step required to solve a problem. In imperative programming, every operation is coded and the code itself specifies how the problem is to be solved.

**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 allow us to "hook" into React features such as state and lifecycle methods.

You must import Hooks from react.

import React, { useState } 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.

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

Hooks will not work in React class components.

**useState HOOKS**

The React useState Hook allows us to track state in a function component.

State generally refers to data or properties that need to be tracking in an application.

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.

Initialize state at the top of the function component.

import { useState } from "react";

function FavoriteColor() {

const [color, setColor] = useState("");

}

Notice that again, we are destructuring the returned values from useState.

The first value, color, is our current state.

The second value, setColor, is the function that is used to update our state.

Lastly, we set the initial state to an empty string: useState("")

## **Read State**

We can now include our state anywhere in our component.

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>

}

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<FavoriteColor />);

## **Update State**

To update our state, we use our state updater function.

We should never directly update state. Ex: color = "red" is not allowed. Use a button to update 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>

<button

type="button"

onClick={() => setColor("blue")}

>Blue</button>

</>

)

}

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<FavoriteColor />);

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.

Create multiple state Hooks:

import { useState } from "react";

import ReactDOM from "react-dom/client";

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>

</>

)

}

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Car />);

Or, we can just use one state and include an object instead!

Create a single Hook that holds an object:

import { useState } from "react";

import ReactDOM from "react-dom/client";

function Car() {

const [car, setCar] = useState({

brand: "Ford",

model: "Mustang",

year: "1964",

color: "red"

});

return (

<>

<h1>My {car.brand}</h1>

<p>

It is a {car.color} {car.model} from {car.year}.

</p>

</>

)

}

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Car />);

Since we are now tracking a single object, we need to reference that object and then the property of that object when rendering the component. (Ex: car.brand)

## **Updating Objects and Arrays in State**

When state is updated, the entire state gets overwritten.

What if we only want to update the color of our car?

If we only called setCar({color: "blue"}), this would remove the brand, model, and year from our state.

We can use the JavaScript spread operator to help us.

Use the JavaScript spread operator to update only the color of the car:

import { useState } from "react";

import ReactDOM from "react-dom/client";

function Car() {

const [car, setCar] = useState({

brand: "Ford",

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>

</>

)

}

const root = ReactDOM.createRoot(document.getElementById('root'));

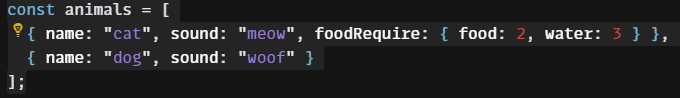
root.render(<Car />);

Because we need the current value of state, we pass a function into our setCar function. This function receives the previous value.

We then return an object, spreading the previousState and overwriting only the color.

**DESTRUCTURING**

Destructuring an array or object is to extract only what is needed and helps in reducing the complex structure.



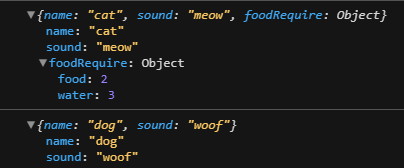
**Destructuring an array**

When destructuring arrays, the variables declared should be unique and the order that variables are declared is important.

const [cat,dog] = animals;

console.log(cat); 🡸🡺 console.log(animals[0]);

console.log(dog);



**Destructuring an object**

When destructuring objects, the object properties do not have to be declared in a specific order. The variables declared should match with the object properties.

const [cat,dog] = animals;

const {name,sound} = cat;

console.log(name); 🡸🡺 console.log(animals[0].name);

console.log(sound);

**Destructuring an object with custom names**

const [cat,dog] = animals;

const {name : animalName,sound : animalSound} = cat;

console.log(animalName);

console.log(animalSound);

**Destructuring an object with custom values**

This can’t overwrite the property values but if the property is not there, then it will insert that values.

const [cat,dog] = animals;

const {name ="cow",sound ="maa"} = cat;

console.log(name);

console.log(sound);

**Destructuring an object with nested object**

****const [cat,dog] = animals;

const {name,sound,foodRequire:{food,water}} = cat;

console.log(name);

console.log(sound);

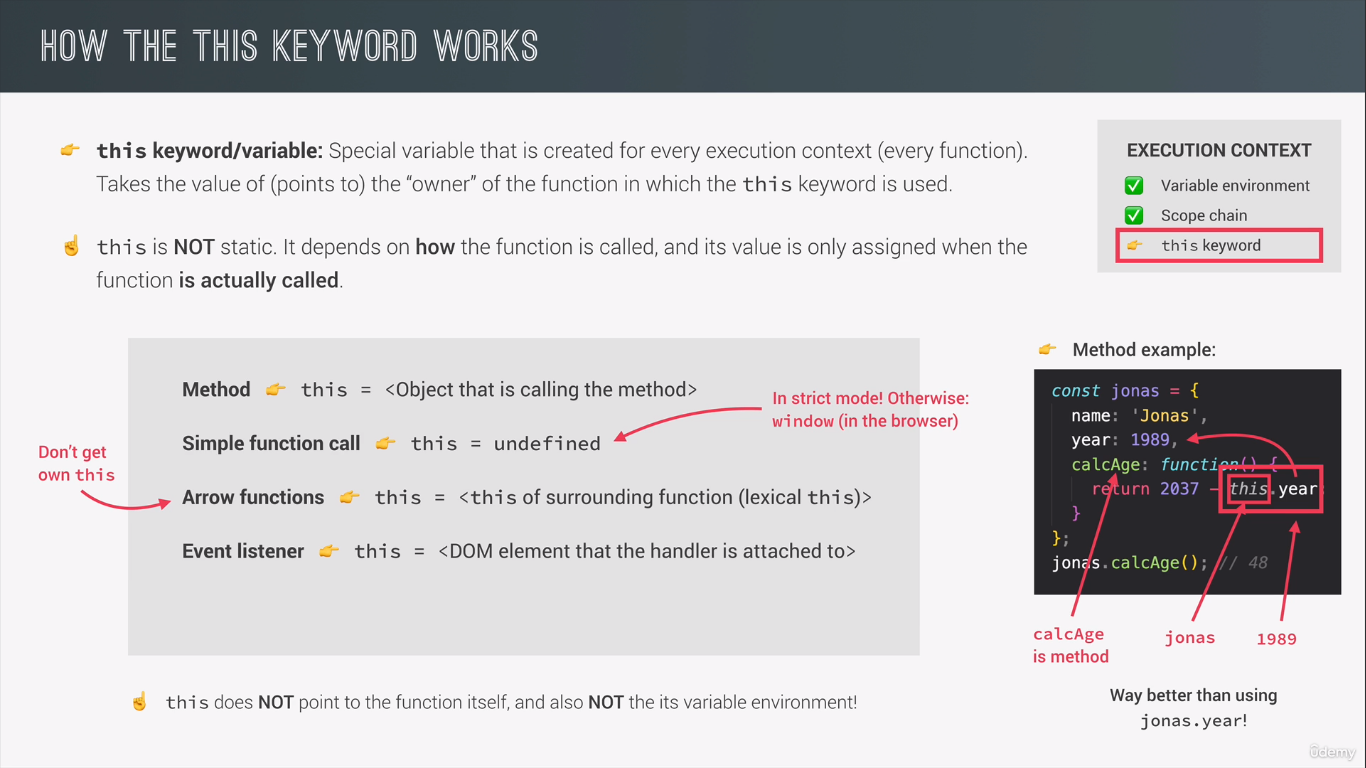
console.log(food);

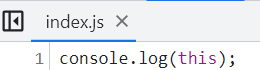
console.log(water);

we can only destructure what we require and leaving out the rest.

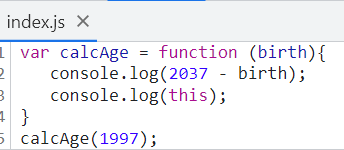
const {foodRequire:{food,water}} = cat;

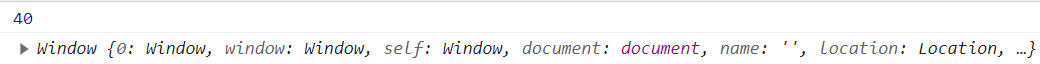
**THIS KEYWORD**

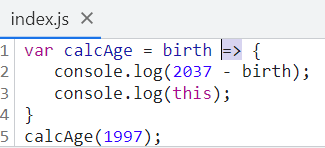


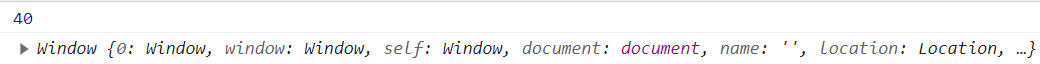
****

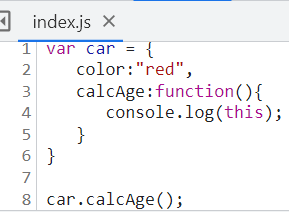
****



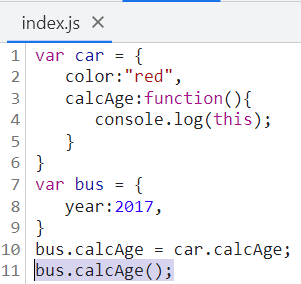




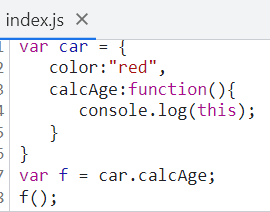














**NODEJS WITH MYSQL**

Install mysql package using **npm install mysql.**

const mysql = require("mysql");

const db = mysql.createConnection({

  host: "localhost",

  user: "root",

  password: "Gk[]@",

  database: "cruddatabase",

});

db.connect((err) => {

    if (err) throw err;

    const sqlInsert =

      "INSERT INTO movie\_review (movieName,movieReview) values ('baasha2','good movie');";

    db.query(sqlInsert, (err, result) => {

      if (err) throw err;

      console.log("1 row inserted");

      res.send("hello world");

    });

  });

**For inserting multiple records**

var mysql = require('mysql');  
  
var db = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword",  
  database: "mydb"  
});  
  
db.connect(function(err) {  
  if (err) throw err;  
  console.log("Connected!");  
  var sql = "INSERT INTO customers (name, address) VALUES ?";  
  var values = [  
    ['John', 'Highway 71'],  
    ['Peter', 'Lowstreet 4'],  
    ['Amy', 'Apple st 652'],  
    ['Hannah', 'Mountain 21'],  
    ['Michael', 'Valley 345'],  
    ['Sandy', 'Ocean blvd 2'],  
    ['Betty', 'Green Grass 1'],  
    ['Richard', 'Sky st 331'],  
    ['Susan', 'One way 98'],  
    ['Vicky', 'Yellow Garden 2'],  
    ['Ben', 'Park Lane 38'],  
    ['William', 'Central st 954'],  
    ['Chuck', 'Main Road 989'],  
    ['Viola', 'Sideway 1633']  
  ];  
  db.query(sql,**[values]**, function (err, result) {  
    if (err) throw err;  
    console.log("Number of records inserted: " + result.affectedRows);  
  });  
});

**JavaScript functions**

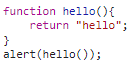
Function declaration

1. **regular function**

function without arguments and without return values



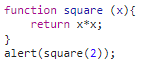
function without arguments and with return values



Function with argument and without return values

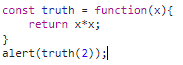


Function with arguments and return value

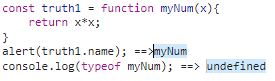


1. function expression with anonymous

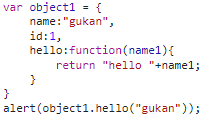
here the right side anonymous function should be expression(return a value)



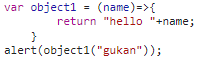
Function expression with named function



1. method declaration

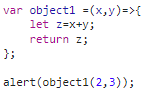


1. arrow functions





Arrow functions with two arguments and two statements



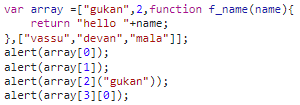
**ARRAYS**

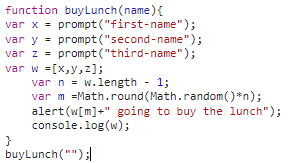
## Array Elements Can Be Objects

JavaScript variables can be objects. Arrays are special kinds of objects.

Because of this, you can have variables of different types in the same Array.

You can have objects in an Array. You can have functions in an Array. You can have arrays in an Array:



****