INDEX

S. No.	Date	Name of the Experiment	Page No.	Marks Awarded	Remarks
1.	09-12-24 12-12-24	a. Add metadata to Homepage using HTML 5 b. Enhance Homepage with sectioning elements c. Add list items to About Us page d. Link pages and bookmark categories e. Add © symbol in footer f. Add global attributes to SignUp Page	1		
2.	16-12-24 20-12-24	a. Display inventory using HTML table b. Create Signup form using form elements c. Enhance Signup with input attributes d. Add media using audio, video, iframe	33		
3.	23-12-24 27-12-24	a. Calculate area using var, let, const b. Display movie details using template literals c. Ticket price calculation with discount d. Booking validation using conditions e. Booking using loops and conditions	54		
4.	30-12-24 03-01-25 20-01-25	a. Use nested and arrow functions b. Create Employee class extending Person c. Use event handlers for booking logic d. DOM manipulation on user click	85		
5.	24-01-25 27-01-25	a. Render movie array using JavaScript b. Simulate stock price with async function c. Validate login using modules	111		
6.	31-01-25 03-02-25	a. Run functions in Node.js b. Create web server using Node.js c. Use modularization in Node app d. Restart Node application e. File operation in Node.js	149		

INDEX

S. No.	Date	Name of the Experiment	Page No.	Marks Awarded	Remarks
7.	07-02-25 17-02-25	a. Implement routing using Express.js b. Use middleware for error, log, POST c. Connect to MongoDB with Mongoose d. Wrap schema in model	180		
8.	21-02-25 24-02-25	a. Perform CRUD with Mongoose b. Develop APIs for note application c. Session management using cookies d. Session management using sessions e. Add Helmet security in Express app	198		
9.	28-02-25 03-03-25	a. Display price using string values b. Filter products using arrow function c. Function with parameter and return type d. Filter manufacturers by price e. Optional parameters in function	230		
10.	07-03-25 10-03-25	a. Add products to cart using rest parameter b. Declare Product interface c. Use duck typing with product interface d. Access function type interface	241		
11.	17-03-25 21-03-25	a. Extend interfaces and use combined one b. Use Product class in array c. Class with constructor and method d. Access modifiers in class	258		
12.	24-03-25 28-03-25	a. Use getter and setter methods b. Use namespace and import class c. Module function for total price d. Create and sort generic array	270		

EXPERIMENT-1

COURSE NAME :HTML 5 – THE LANGUAGE

1a) INTRODUCTION TO HTML-1:

WHAT IS HTML?

Hyper Text Markup Language (HTML) is a standard markup language to create the structure of a web page. It annotates the content on a web page using HTML elements.

In a web page, all instructions to the browser are given in the form of HTML tags, also known as HTML elements.

The basic structure of html is:

```
<!DOCTYPE html>
<html>
  <head>
    <title> Homepage </title>
  </head>
  <body>
      //paragraph
</body>

</html>
```

Case-insensitivity :

HTML elements are case-insensitive. The browser understands the HTML tags irrespective of their cases.

Syntax:

```
<!DOCTYPE html>
<html>
 <head>
```

```
  <title>Homepage </title>
</head>
 <body>
   Hello World!
 </body>
</html>
```

OUTPUT: Hello World!

PLATFORM-INDEPENDENCY:

HTML Language is platform-independent. That means the same HTML code can run on different operating systems as shown below.

Syntax:

```
<!DOCTYPE html>
<html>
<head>
   <title>sample page</title>
</head>
<body>
  <p>Hello World!</p>
</body>
</html>
```

Output:

Machintosh

Windows

Cross-platform support

Doctype Declaration :

There are many versions of HTML out there such as - HTML 2.0, HTML 3.0, HTML 3.2, HTML4.0, HTML 4.01 and latest is HTML5.0. In each version, some elements and attributes are either added or depreciated. The appearance of your .html page depends on how the browser renders HTML elements. And how the browser renders HTML elements depends on how the browser understands them.

This is done by using <!DOCTYPE> declaration which stands for Document Type. It tells the browser what version of HTML it should follow for rendering the web page.

```
Syntax: <!DOCTYPE html>
```

Types of Elements:

HTML elements can be further categorized into two as below:

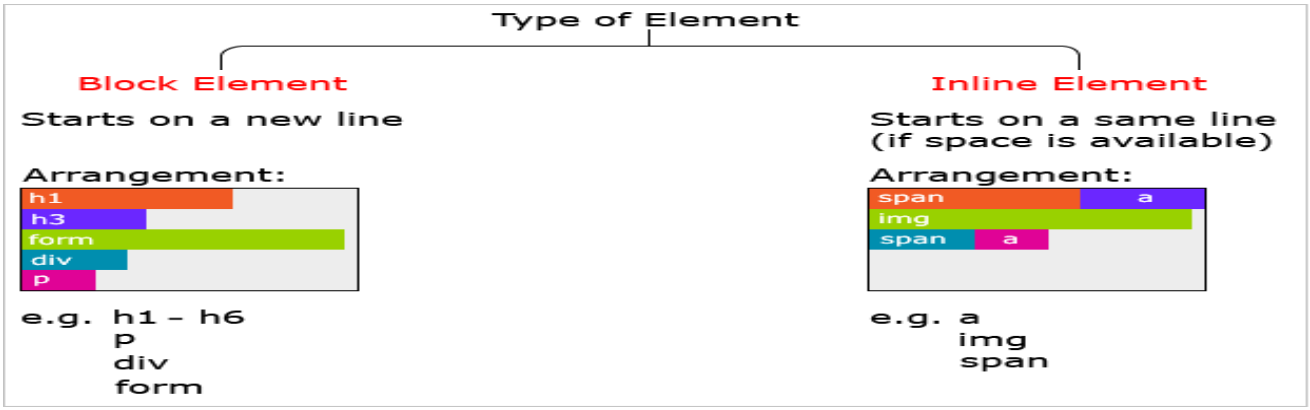
Block Element:

A block element begins on a new line occupying the entire width of the parent tag.

Inline Element:

An inline element occupies the necessary space to accommodate the content in the element. Inline elements can be nested within other inline elements, whereas, block elements cannot be nested within inline elements.

Some of the examples are illustrated below:



Html Elements-Attributes, metadata elements:

HTML elements can contain attributes that can be considered as an additional feature to set various properties and they are optional.

Some of the attributes can be used with any of the HTML elements and there can be referred to as 'global attributes'. Also, some attributes can be used only with particular elements. Following are some features of attributes:

- All the attributes can contain properties like name and value which can be used by a developer to assign respective details for that HTML elements.
- Attributes are to be set only in the start tag of a container HTML element.
- Attributes are case-insensitive, but it is recommended to use lowercase as a best practice.
- The best practice is always to quote attribute value even though we will not get any execution errors if they are not provided in quotes.

The lang attribute specifies the language of the content of the HTML page.

```
Syntax: <html lang="en-US">
```

Specifies that the content of .html page is written in U.S. version of English language

Metadata:

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

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

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

Metadata is used by browsers (how to display content or reload page), search engines (keywords), and other web services.

Syntax:

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

EXAMPLE:

Include the Metadata element in Homepage.html for providing description as "IEKart's is an online shopping website that sells goods in retail. This company deals with various categories like Electronics, Clothing, Accessories etc.

Source code:

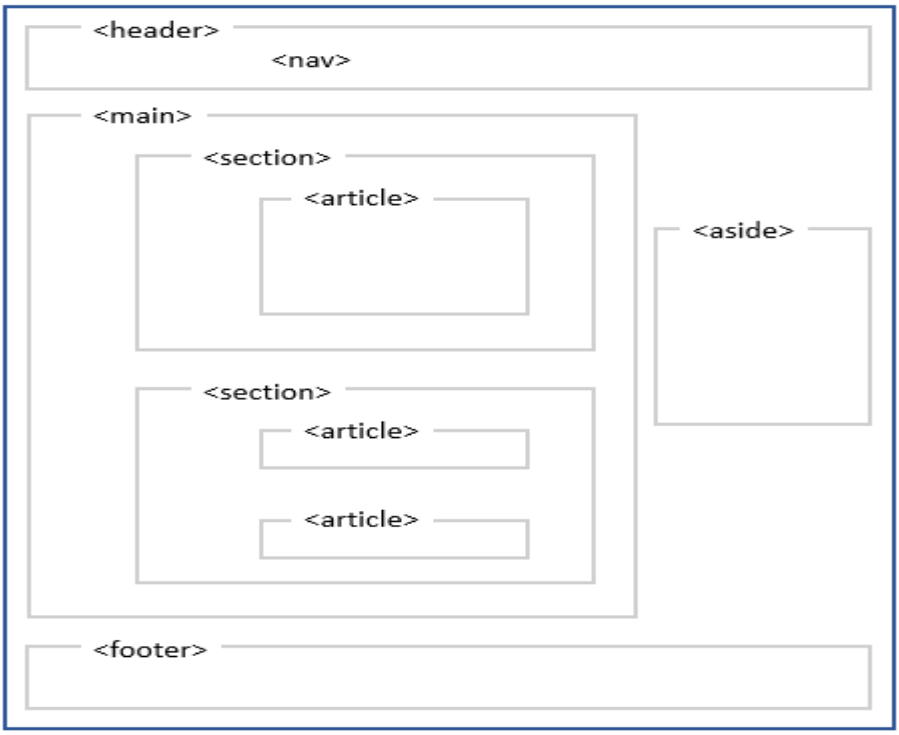
```
   <!DOCTYPE html>

<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="description" content="IEKart's is an online shopping website that sells goods in retail. This company deals with various categories like Electronics, Clothing, Accessories etc.">
    <title>IEKart online shopping</title>
</head>
<body>

</body>
</html>
```

1b) Sectioning Elements:

Web crawlers like Google, Bing, etc. are widely used for searching websites. They lookup each web page code and render the web page as per the HTML tags used and the styling associated.

Any regular user who is accessing any website will notice the below observations in most of the web pages of the website:

1. Right at the beginning of a web page, a header containing the website name is clearly displayed in the form of a logo or text. This helps the user to know which website they are currently referring to.
2. The links to navigate to other web pages of the website are displayed in the header. This makes a website user to figure out easily how to access other web pages of that website.
3. Details like copyright, about us, etc. are usually displayed at the bottom end of the screen, as part of the footer, as these details hold lesser importance, as compared to the actual data that they intend to read in the page.



<header>

The <header> element is used to include header content like web page logo, login link, website settings link, etc. Ideally, every web page has one header. However, multiple headers may also be included as per need.

Syntax:

```
<header>
   <h3>About Us</h3>
</header>
```

<footer>

The <footer> element is used to include footer content like copyright, about us, terms and conditions link, etc. One footer is included per page.

Syntax:

```
<footer>

Copyright @ WayFar, 2020

<a href="./AboutUs.html">About Us</a>
</footer>
```

<main>

The <main> element is used for demarking the main content of the web page. Only one main tag per web page is allowed.

Syntax:

```
<main>
<section>
   ..
</section>
<section>
   <article>
      ..
```

```
        </article>
        <article>
            ..
        </article>
    </section>
</main>
```

<nav>

The <nav> element is used for navigational content like navigation menu for the website. There is no limit to the number of times <nav> tag can be used on a web page. As long as there are navigation links, links can be wrapped inside <nav>.

Syntax:

```
        <nav>
    <a href="Home.html">Home</a>
    <a href="Login.html">Login</a>
</nav>
```

<section>

The <section> element is used to organize the web page into different sections.

Syntax:

```
    <main>
    <section>
        <p>Section 1</p>
    </section>
    <section>
        <p>Section2</p>
    </section>
</main>
```

<article>:

The <article> element is used to include self-contained composition on a web page.

Syntax:

```
<article>
   <h1>MEAN stack</h1>
   <p>MEAN stack training includes discussion on MongoDB, Node, Express and Angular with the corresponding certifications</p>
</article>
```

EXAMPLE:

Enhance the Homepage.html of IEKart's Shopping Application by adding appropriate sectioning elements.

Source Code:

```
<!DOCTYPE html>
<html>

    <head>
        <title>Hussian's Shopping</title>
        <meta name="description" content="Online, Shopping" />
    </head>

    <body>
        <section>
            <article>
              <h3>Electronics</h3>
              <p>Choose from a wide range of mobiles, washing machine, camera, laptop & many more</p>
            </article>
            <article>
              <h3>Clothing</h3>
              <p>Looking for Online Shopping Site for Fashion Clothing. We bring you the finest Collection of Women, Men and Children Wear.</p>
            </article>
            <article>
              <h3>Accessories</h3>
```

```
            <p>To your luck, we have recently added a large variety of gift
accessories in our online shop to get you the right kind of gift for any
festive season.</p>
          </article>
        </section>
    </body>

</html>
```

Output:

Electronics

Choose from a wide range of mobiles, washing machine, camera, laptop & many more

Clothing

Looking for Online Shopping Site for Fashion Clothing. We bring you the finest Collection of Women, Men and Children Wear.

Accessories

To your luck, we have recently added a large variety of gift accessories in our online shop to get you the right kind of gift for any festive season.

1c) Paragraph Element:

The paragraph element is generally used for denoting a paragraph. Any textual content can be mentioned inside this element.

It is defined using … tag.

Syntax:

```
<!DOCTYPE html>
<html>
   <body>
     <p>This is a Paragraph</p>
   </body>
</html>
```

Division Elements:

The division element is used to group various other HTML tags. This element helps us in organizing the web page into different sections.

If any common rule or style needs to be added to a particular section, the same can be applied to the corresponding division. The rule or style gets applied to all the contents of the division thereby.

It is defined using … tag.

Syntax:

```
<!DOCTYPE html>
<html>
   <body>
      <div>
         <p> This is first paragraph </p>
         <p> This is second paragraph </p>
         <p> This is third paragraph </p>
      </div>
   </body>
</html>
```

Span Element:

Similar to the division element, the span element is also used to group various other HTML tags to apply some common styles.

It is defined by using ... tag.

The span element is by default inline in nature, and hence no new line is added after the span ends. This tag is preferred only when we cannot use any other semantic tags.

Syntax:

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

```
        <div>
            <span>first section of paragraph</span>
            <span>second section of paragraph</span>
        </div>
    </body>
</html>
```

List Element:

HTML Lists are used to specify lists of information. All lists may contain one or more list elements. There are three different types of HTML lists:

1. Ordered List or Numbered List (ol)
2. Unordered List or Bulleted List (ul)
3. Description List or Definition List (dl)

Ordered List or Numbered list :

In the ordered HTML lists, all the list items are marked with numbers by default. It is known as numbered list also. The ordered list starts with <ol> tag and the list items start with <li> tag.

Syntax:

```
<h1>Courses offered:</h1>
    <ol>
        <li>HTML5</li>
        <li>CSS</li>
        <li>JS</li>
        <li>Bootstrap</li>
    </ol>
```

Unordered list or bulleted list:

In HTML Unordered list, all the list items are marked with bullets. It is also known as bulleted list also. The Unordered list starts with <ul> tag and list items start with the <li> tag.

Syntax:

```
<h1>Courses offered:</h1>
   <ul style="list-style-type: square;">
      <li>HTML5</li>
      <li>CSS</li>
      <li>JS</li>
      <li>Bootstrap</li>
   </ul>
```

Description list or Definition list:

HTML Description list is also a list style which is supported by HTML and XHTML. It is also known as definition list where entries are listed like a dictionary or encyclopedia.

The HTML definition list contains following three tags:

1. **<dl> tag** defines the start of the list.
2. **<dt> tag** defines a term.
3. **<dd> tag** defines the term definition (description).

Syntax:

```
<dl>
<dt> Description Term 1 </dt>
<dd> Description Definition 1 </dd>
<dt> Description Term 2 </dt>
<dd> Description Definition 2 </dd>
</dl>
```

EXAMPLE:

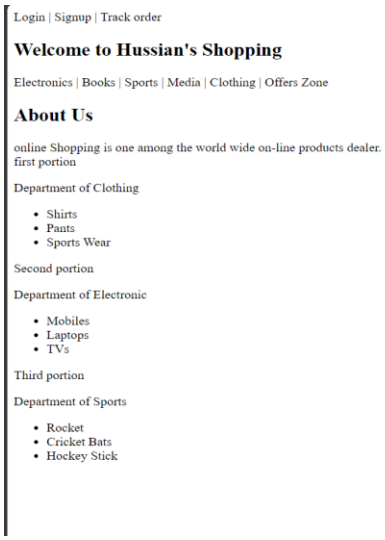
Make use of appropriate grouping elements such as list items to "About Us" page of IEKart's Shopping Application

Source code:

```
<!DOCTYPE html>
```

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="description" content="IEKart's is an online shopping website
that sells goods in retail. This company deals with various categories like
Electronics, Clothing, Accessories etc.">
    <title>IEKart online shopping</title>
</head>
<body>
        <header>
            <nav> Login | Signup | Track order </nav>
            <h2> Welcome to Hussian's Shopping </h2>
            <nav> Electronics | Books | Sports | Media | Clothing | Offers
Zone </nav>
            </header>
            <article>
            <h2>About Us</h2> online Shopping is one among the world wide on-
line products dealer.
        <div> <! -- Groups all 3 paragraphs -->
            <span>first portion </span>
            <p>Department of Clothing</p>
            <ul>
                <li>Shirts </li>
                <li>Pants</li>
                <li>Sports Wear</li>
            </ul>
            <span>Second portion </span>
            <p>Department of Electronic</p>
            <ul>
                <li>Mobiles</li>
                <li>Laptops</li>
                <li>TVs</li>
            </ul>
            <span>Third portion</span>
            <p>Department of Sports</p>
            <ul>
                <li>Rocket</li>
                <li>Cricket Bats</li>
                <li>Hockey Stick</li>
            </ul>

        </div>


</body>
</html>
```

Output:



1d) Link Element:

A website necessarily is a collection of related web pages, where each web page is typically created for a particular purpose.

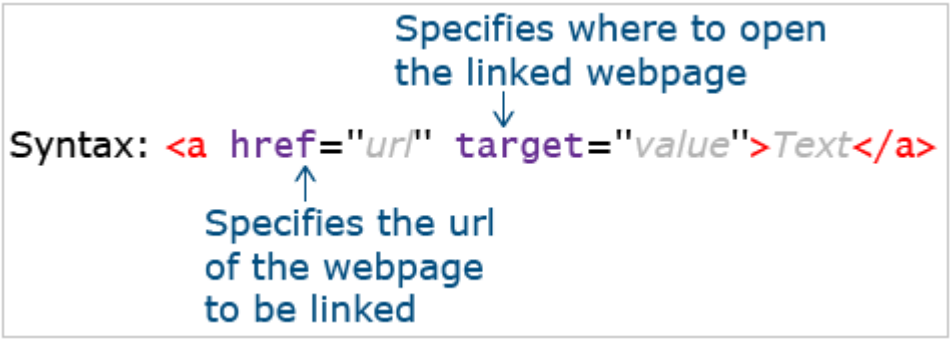
When developing any website:

- Each web page is necessarily coded in individual HTML files.
- To see a particular web page, the respective HTML file has to be opened up in a browser.

Below are the advantages if hyperlinks are used:

- We can create connections or links between HTML documents/web pages and users can navigate from one web page to another by clicking on "hyperlinks".
- We would now feel that we have a website which is a collection of interconnected web pages.

  Link elements are defined using <a> .. tag as below:



A hyperlink is a prime way in which users can navigate from one web page to another. A hyperlink can point to another web page, or website, or files, or even specific locations on the same web page.

Hyperlinks can be of any of the below types:

Text hyperlink:

- A clickable text is used to take the user to another web page. Largely, we use text-based hyperlinks.
- This text usually appears with an underline and in a different color.
- This colour mapping is automatically done by the browser for all text hyperlinks.

Syntax:

```
<a href="Enquire.html"> Click here to connect to us </a><br/>
<a href="http://www.google.com"> Click here to go to Google website </a>
```

Image hyperlink:

- A clickable image is used to take the user to another web page.

Syntax:

```
<a href="http://www.google.com">
<img src="google.jpg" />
</a>
```

Bookmark hyperlink:

When a web page is lengthy, we commonly come across icons or links that say "Go to Top" or "Go to Bottom". Click on these links does take the user to the top of the page or bottom, as applicable. Sometimes we also observe, on click of a text in the menu bar, the page auto scrolls to that particular section on that page. This is achieved by using the Bookmarking concept and the same is implemented by using hyperlinks.

Syntax:

```
<h2 id="top">Topic</h2>
<p>Detail……</p>
<p>Detail……</p>
<p>Detail……</p>
<a href="#top">Go to Top</a>
```

Email hyperlink:

- It allows users to send an email by clicking on that link.

Syntax:

```
<a  href="mailto:someone@xyz.com?Subject=Hello%20again">Send Mail</a>
```

Contact number hyperlink:

- It allows the user to call a number by clicking on that link.

Let us discuss various hyperlinks that can be created in an HTML page.

Syntax:

```
<a href="tel:+9999">Call Us</a>
```

EXAMPLE:

Source code:

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <a href="https://owasp.org/" target="_blank">
    Opens OWASP website in new tab
    </a>
    <br/>
     <a href="https://owasp.org/" target="_top">
        Opens OWASP website in same tab
    </a>
</body>
</html>
```

Output:

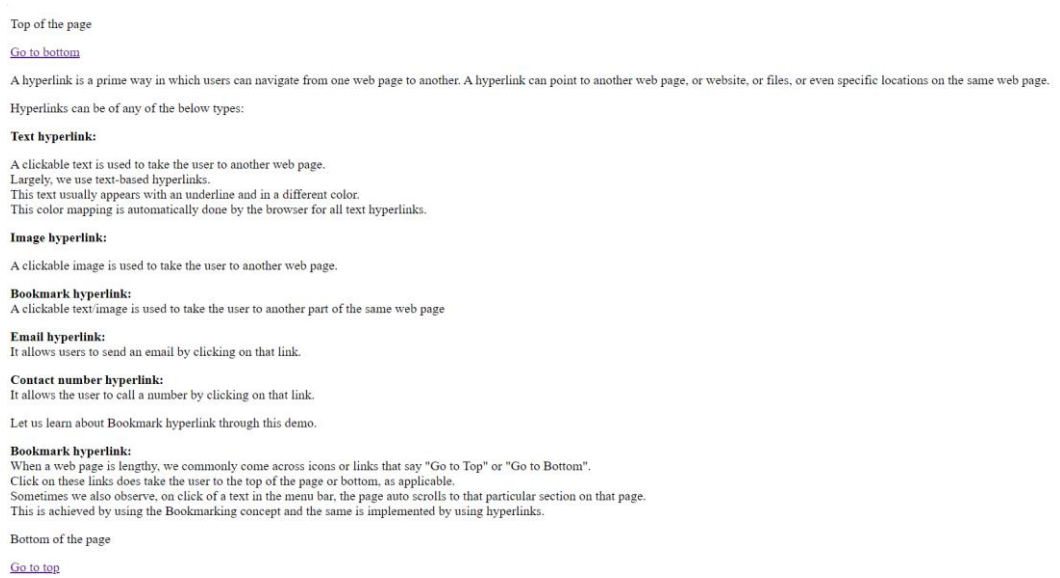
Source code(Bookmark):

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
        <p id="top">
            Top of the page
        </p>
        <a href="#bottom">
         Go to bottom
        </a>
        <p>
          A hyperlink is a prime way in which users can navigate from one web page to another. A hyperlink can point to another web page, or website, or files, or even specific locations on the same web page.</br>
```

```
        </p>
        <p>
        Hyperlinks can be of any of the below types:
        </p>
        <p>
        <p>
          <b> Text hyperlink:</b>
        </p>
        <p>
          A clickable text is used to take the user to another web page.</br>
          Largely, we use text-based hyperlinks.</br>
          This text usually appears with an underline and in a different
color.</br>
          This color mapping is automatically done by the browser for all text
hyperlinks.</br>
        </p>
        <p>
          <b> Image hyperlink:</b></br>
        </p>
        <p>
          A clickable image is used to take the user to another web page.</br>
        </p>
        <p>
          <b> Bookmark hyperlink:</b></br>
         A clickable text/image is used to take the user to another part of
the same web page</br>
        </p>
        <p>
          <b> Email hyperlink:</b></br>
          It allows users to send an email by clicking on that link.</br>
        </p>
        <p>
          <b> Contact number hyperlink:</b></br>
          It allows the user to call a number by clicking on that link.</br>
        </p>
        <p> Let us learn about Bookmark hyperlink through this demo. </p>
        <p>
        <b>Bookmark hyperlink:</b></br>
         When a web page is lengthy, we commonly come across icons or links
that say "Go to Top" or "Go to Bottom". </br>
         Click on these links does take the user to the top of the page or
bottom, as applicable.</br>
         Sometimes we also observe, on click of a text in the menu bar, the
page auto scrolls to that particular section on that page. </br>
         This is achieved by using the Bookmarking concept and the same is
implemented by using hyperlinks.
        </p>
        <p id="bottom">
```

```
            Bottom of the page
        </p>
        <a href="#top">
            Go to top
        </a>
</body>
</html>
```

Output:



Top of the page

Go to bottom

A hyperlink is a prime way in which users can navigate from one web page to another. A hyperlink can point to another web page, or website, or files, or even specific locations on the same web page.

Hyperlinks can be of any of the below types:

Text hyperlink:

A clickable text is used to take the user to another web page.
Largely, we use text-based hyperlinks.
This text usually appears with an underline and in a different color.
This color mapping is automatically done by the browser for all text hyperlinks.

Image hyperlink:

A clickable image is used to take the user to another web page.

Bookmark hyperlink:
A clickable text/image is used to take the user to another part of the same web page

Email hyperlink:
It allows users to send an email by clicking on that link.

Contact number hyperlink:
It allows the user to call a number by clicking on that link.

Let us learn about Bookmark hyperlink through this demo.

Bookmark hyperlink:
When a web page is lengthy, we commonly come across icons or links that say "Go to Top" or "Go to Bottom".
Click on these links does take the user to the top of the page or bottom, as applicable.
Sometimes we also observe, on click of a text in the menu bar, the page auto scrolls to that particular section on that page.
This is achieved by using the Bookmarking concept and the same is implemented by using hyperlinks.

Bottom of the page

Go to top

Source code(email and tel):

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
        Any suggestions or feedbacks regarding our services?
        <a href="mailto:abcd@xyz.com?Subject=Hello%20again">Send Mail</a> <br>
        <a href="tel:+916302900386">Call Us</a> <br>
</body>
</html>
```

Output:

Any suggestions or feedbacks regarding our services? Send Mail
Call Us

1e) Character Entities:

Some characters are reserved in HTML.

For example: If you use the less than (<) or greater than (>) sign in your content, the browser may mix them with HTML tags.

Also, some characters are unavailable on the keyboard.

For example: ©

Character entities are used to include such character content on a web page.

```
Syntax: &entity_name;
                OR
        &#entity_number;
```

The table below lists widely used character entities supported in HTML5.

CHARACTER	DESCRIPTION	ENTITY NAME	ENTITY NUMBER
	Non-breaking space		
<	Less than	&lt;	&#60;

>	Greater than	>	&#62
&	Ampersand	&amp	&
©	Copyright	©	©
€	Euro	&euro	€
£	Pound	£	£
®	Registered trademar	®	®

Source code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <p>
        &lt;body&gt; is start tag of body element.
    </p>
    <p>
        &lt;/body&gt; is end tag of body element.
    </p>
</body>
</html>
```

Output:

<body> is start tag of body element.

</body> is end tag of body element.

1f) HTML5 Global Attributes:

HTML global attributes are those attributes which are common for all HTML elements. The global attributes are supported by both standard and non-standard element.

The global attributes can be used with all elements, although it may not have any effect on some elements.

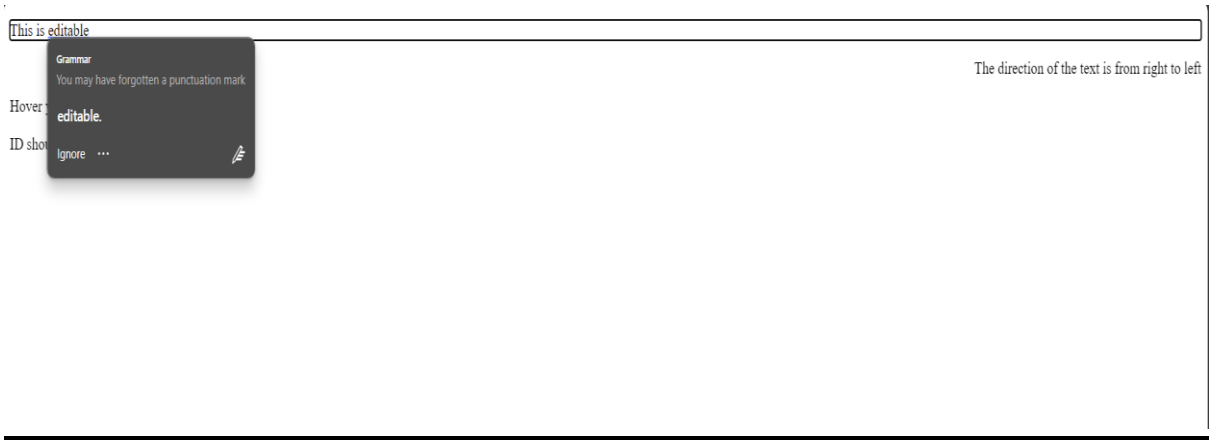
Attribute	Description
contenteditable	Allows the user to edit content. Possible values are true/false.
dir	Specifies text direction. Possible values are Itr/ rtl.
title	Displays the string message as a tooltip.
spellcheck	Specifies whether the spelling of an element's value should be checked or not. Possible values are true/false.
id	Gives a unique id to an element.

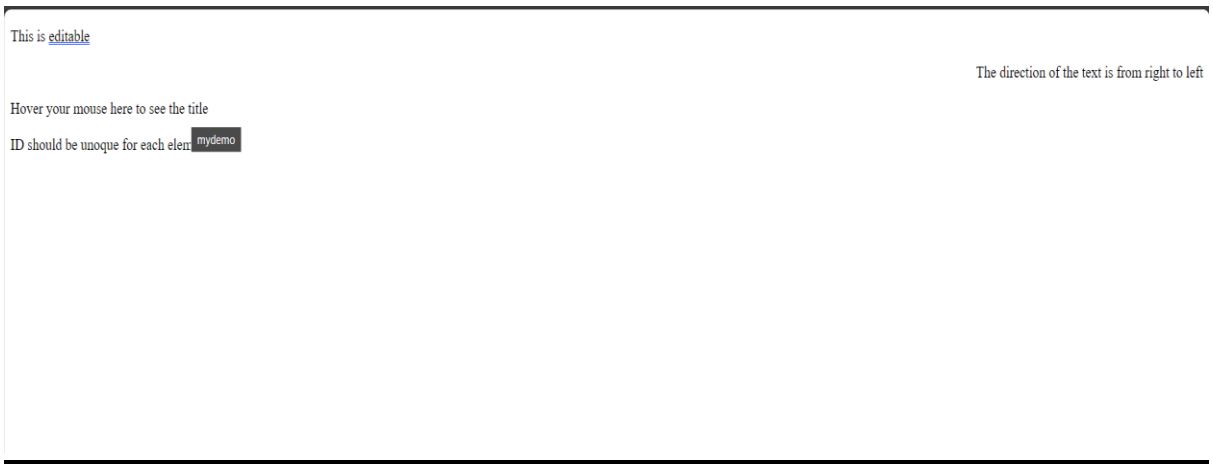
EXAMPLE:

Source code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <div>
        <p contenteditable="true">This is editable</p>
        <p dir="rtl">The direction of the text is from right to left</p>
        <p title="mydemo">Hover your mouse here to see the title</p>
        <p id="id1">ID should be unoque for each element</p>
    </div>
</body>
</html>
```

Output:





FINAL CODE FOR THE SIGNUP PAGE:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <style>
body{
    font-family: 'Franklin Gothic Medium', 'Arial Narrow', Arial, sans-serif;
    margin: auto;
  width: 80%;
  border: 4px solid green;
```

```
   border-bottom:100px ;
   margin-top: 100px;
   margin-left: 10%;
   margin-right: 5%;
}*{box-sizing: border-box;}
input[type=text] ,input[type=password]{
    width: 100%;
  padding: 17px;
  margin: 10px 10px 10px 10px;
  display: inline-block;
  border: none;
  background: #f1f1f1;

}
input[type=text]:focus,input[type=password]:focus{
    background-color:#ddd;
    outline: none;
}
hr{
    border: 2px solid #f1f1f1;
    margin-bottom: 25px;
}
button{
    background-color: aliceblue;
    padding: 10px 20px;
    margin: 9px 0px;
    opacity: 0.9;
    cursor: pointer;
```

```
    width:100% ;


}
button:hover{
    opacity: 1.5;
}
.cancelb{
    background-color: red;
   padding: 20px 20px ;
   float: left;
   width: 50%;
  font-size: 20px;
}
.signupb{
    background-color: green;
   padding: 20px 20px ;
   float: left;
   width: 50%;
   font-size: 20px;
}
<!-- clear fix the over flwo the content>
.b::after{
    content: "";
    clear: both;
    display: table;


}
.a{
```

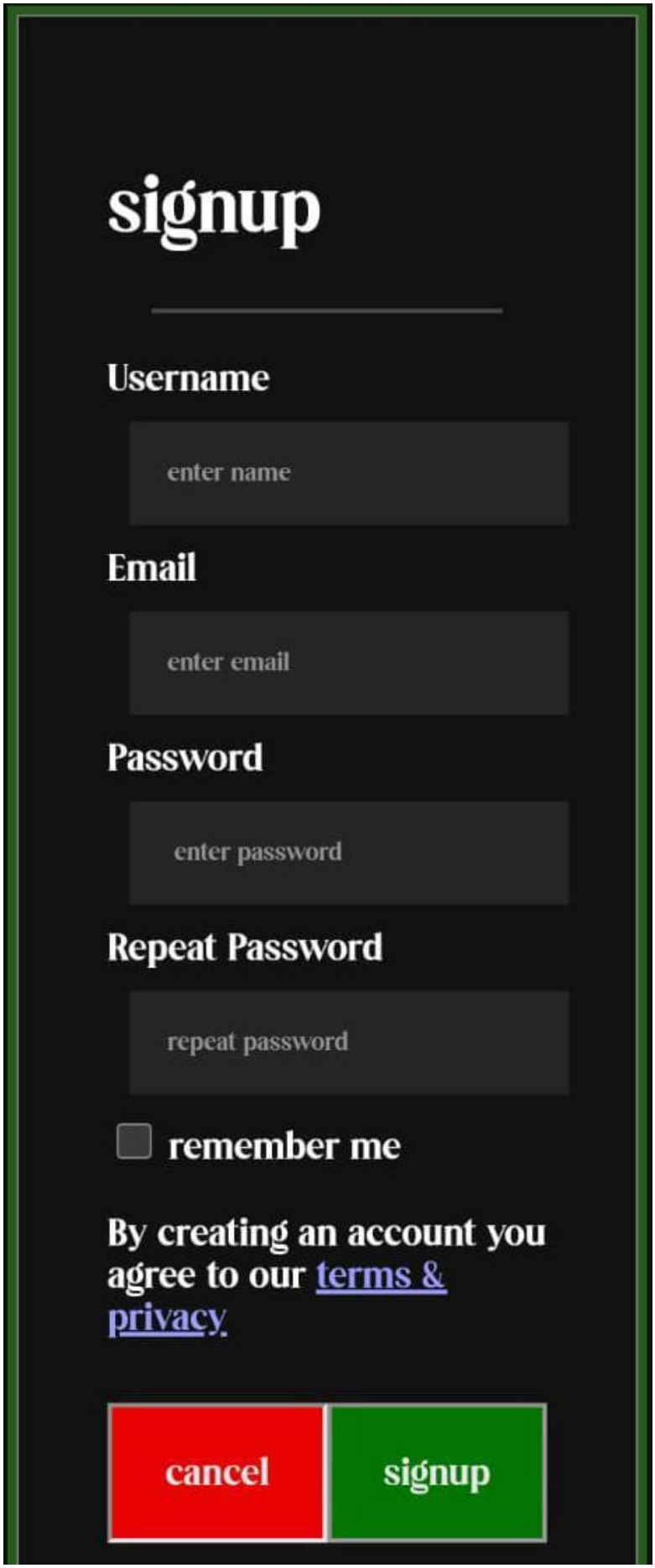
```
    padding: 40px;
}


@media screen and (max-width: 300px) {
  .cancelb,.signupb {
     width: 100%;}
     }
     .error {
        color: red;
 }




 #form{
    padding-bottom: 100px ;
    font-size: 20px;


 }
 hr{
    border:1px solid #f1f1f1 ;
    margin: 20px;
}


</style>
</head>
<body>
    <form  action="chandu.html" style="border: 1px solid #ccc ; padding-bottom: 12%;"
id="form">
```

```
        <div class="a">

            <h1 style="font-size: 40px;">signup</h1>

            <hr>

    <label for="username" id="name"><b>Username</b></label>

    <input type="text" placeholder="enter name" name="username" id="username"
spellcheck="true" contenteditable="true">

    <label for="email"><b>Email</b></label>

    <input type="text" placeholder="enter email" name="email"  required>

    <label for="password"><b>Password</b></label>

    <input type="password" placeholder=" enter password" name="password"
pattern="^(?=.[a-z])(?=.[A-Z])(?=.\d)(?=.[@$!%?&])[A-Za-z\d@$!%?&]{8,}$" id="password"
required>

    <label for="rpassword"><b>Repeat Password</b></label>

    <input type="password" placeholder="repeat password"
name="rpassword"  pattern="^(?=.[a-z])(?=.[A-Z])(?=.\d)(?=.[@$!%?&])[A-Za-
z\d@$!%?&]{8,}$" id="rpassword" required>

    <span id="passworderror" class="error"> </span>

    <label><input type="checkbox" name="" id="checkbox"> remember me</label>

    <p>By creating an account you agree to our <a href="./terms.html">terms &
privacy</a></p>

    <div class="b"><button type="button" class="cancelb">cancel</button>

    <button type="submit" class="signupb" id="signupform">signup</button></div>

    <span id="genaralerror" class="error">

    </span>

        </div>

       <br> <p style="padding-bottom: 5%; font-size: 25px; padding-left: 50px;">copy
right &copy</p>

       <br><p style="font-size: 25px;" dir="ltr"> left to right</p><br>

       <p  style="font-size: 25px"dir="rtl">right to left</p>



    </form>
```

```
    <p></p>
    <script>
        const password = document.getElementById('password');
        const rpassword = document.getElementById('rpassword');
        const signupButton = document.getElementById('signupform');
        const passwordError = document.getElementById('passworderror');
        const emailInput = document.getElementById('email');
        const emailError = document.getElementById('emailerror');
        const generalError = document.getElementById('generalerror');

        signupButton.disabled = true;

        rpassword.addEventListener('input', () => {
            if (password.value !== rpassword.value) {
                passwordError.textContent = "Passwords do not match.";
                signupButton.disabled = true;
            } else {
                passwordError.textContent = "";
                signupButton.disabled = false;
            }
        });

        signupButton.addEventListener('click', (event) => {
            let isValid = true;
            generalError.textContent = "";

            if (!emailInput.value.endsWith('@gmail.com')) {
                isValid = false;
```

```
                emailError.textContent = "Email must end with @gmail.com";

            } else {

                emailError.textContent = "";

            }



            const passwordRegex = /^(?=.\d)(?=.[a-z])(?=.[A-Z])(?=.[@#$%^&!])[0-9a-zA-
Z@#$%^&!]{8,}$/;

            if (!passwordRegex.test(password.value)) {

                isValid = false;

                generalError.textContent = "Password must contain at least 8 characters,
1 lowercase letter, 1 uppercase letter, and 1 special character";

            }

            if (!isValid) {

                event.preventDefault();

            }

        });

    </script>
```

OUTPUT :


signup
Username
enter name
Email
enter email
Password
enter password
Repeat Password
repeat password
remember me
By creating an account you agree to our terms & privacy
cancel
signup

HTML5 - The Language

Aim: Creating Table Elements, Table Elements : Colspan/Rowspan Attributes, border, cellspacing, cellpadding attributes.

Introduction:

HTML (HyperText Markup Language) is used to create and structure sections, paragraphs, and links on web pages. One of its key features is the ability to create tables for organizing data. This report explores the creation of table elements in HTML and explains how to use various attributes such as colspan, rowspan, border, cellspacing, and cellpadding.

Creating Table Elements:

HTML tables allow web developers to arrange data into rows and columns. Tables in HTML are created using the <table> element, which serves as the container for rows (<tr>), headers (<th>), and data cells (<td>). Each of these elements plays a specific role:

1.<table>: Defines the table structure.

Syntax:

```
<table>table name</table>
```

2.<tr>: Represents a row in the table.

Syntax:

```
<tr>
    <td>table data</td>
</tr>
```

3.<th>: Defines header cells, which are typically bold and centered.

Syntax:

```
<tr>
    <th>header name></th>
<tr>
    <td>table data</tr>
```

4.<td>: Represents standard data cells.

Syntax:

```
<td>table data</td>
```

Table Elements:

1. Colspan:

The colspan attribute allows a cell to span across multiple columns. This is useful for combining cells horizontally.

Example:

<th colspan="2">Header 1</th>

this merges two columns under "Header 1."

2. Rowspan:

The rowspan attribute allows a cell to span across multiple rows. This is useful for combining cells vertically.

Example:

<td rowspan="2">Merged Cell</td>

This merges two rows into a single cell.

3. Border:

The border attribute specifies the width of the table border. Although it is better to use CSS for styling, border can quickly add a simple border to the table.

Example:

<table border="1">

This creates a border around the table and its cells.

HTML tables can have borders of different styles and shapes.

1.Collapsed table borders:

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

This will make the borders collapse into a single border:

Example:

```
table, th, td {
        border: 1px solid black;
        border-collapse: collapse;
}
```

2.Styled table border:

If you set a background color of each cell, and give the border a white color (the same as the document background), you get the impression of an invisible border:

Example:

```
table, th, td {
        border: 1px solid white;
        border-collapse: collapse;
}
th, td {
        background-color: #96D4D4;
}
```

4. Cellspacing:

The cellspacing attribute defines the space between table cells. It specifies the width of the gap between the borders of adjacent cells.

Example:

```
<table cellspacing="5">
```

This creates a 5-pixel space between cells.

5. Cellpadding:

The cellpadding attribute defines the space between the cell content and the cell border, effectively padding the content inside the cell.

Example:

```
<table cellpadding="10">
```

This adds 10 pixels of padding inside each cell.

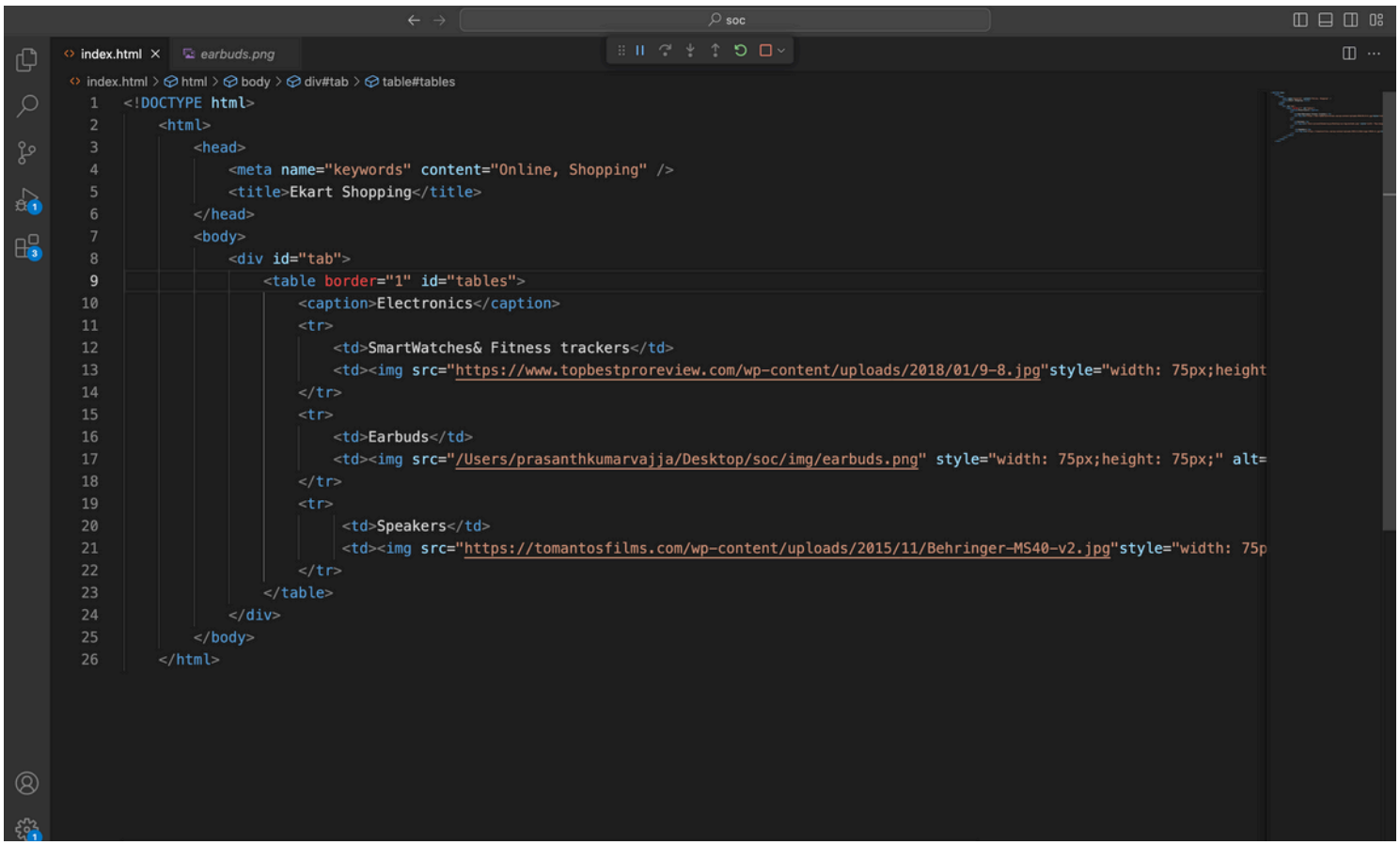
Conclusion:

Creating tables in HTML involves understanding the structure and attributes that define their appearance and behavior. By using colspan, rowspan, border, cellspacing, and cellpadding, developers can effectively manage table layouts and customize their appearance. Employing CSS alongside these attributes can further enhance the flexibility and styling of tables.

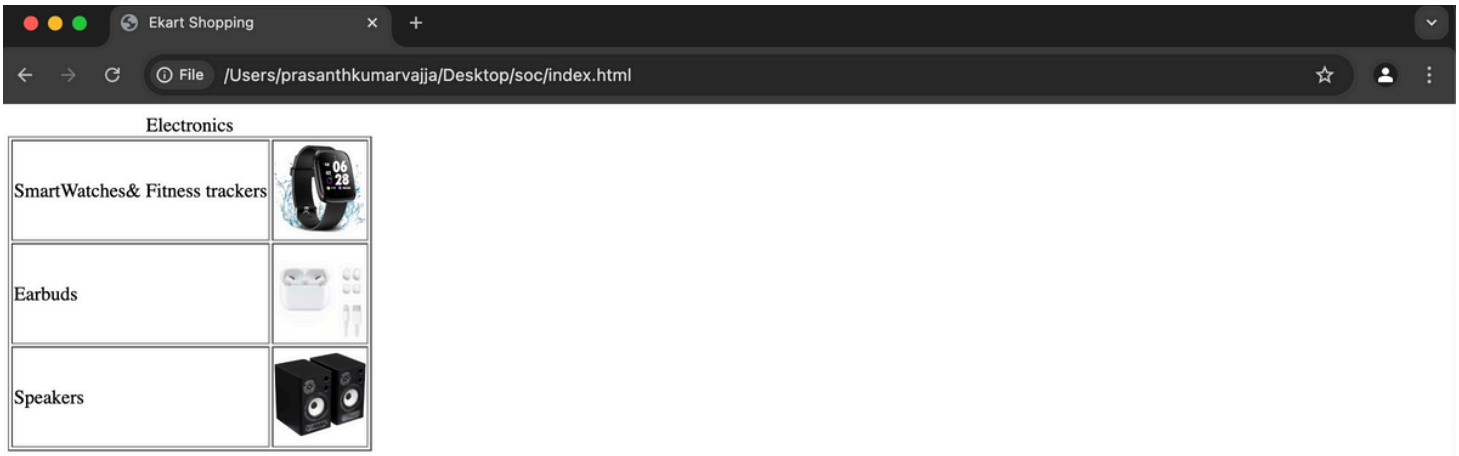
Source code:

```
<!DOCTYPE html>
 <html>
 <head>
 <meta name="keywords" content="Online, Shopping" />
 <title>Ekart Shopping</title>
 </head>
 <body>
 <div id="tab">
 <table border="1" id="tables">
 <caption>Electronics</caption>
 <tr>
 <td>SmartWatches& Fitness trackers</td>
                    <td><img                    src="https://www.topbestproreview.com/wp-content/uploads/2018/01/9-8.jpg"style="width:  75px;height:  75px;"  alt="watch Pic"></td>
 </tr>
 <tr>
 <td>Earbuds</td>
       <td><img       src="/Users/prasanthkumarvajja/Desktop/soc/img/earbuds.png" style="width: 75px;height: 75px;" alt="budsPic"></td>
 </tr>
 <tr>
 <td>Speakers</td>
                        <td><img                        src="https://tomantosfilms.com/wp-content/uploads/2015/11/Behringer-MS40-v2.jpg"style="width:  75px;height:  75px;" alt="Speakers Pic"></td>
 </tr>
 </table>
 </div>
 </body>
 </html>
```

screenshots:



output:



Aim: Using the form elements create Signup page for IEKart's Shopping applications.

Description: The <form> tag is used to create an HTML form for user input.

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

- <input>
- <textarea>
- <button>
- <select>
- <option>
- <optgroup>
- <fieldset>
- <label>
- <output>

Syntax:

```
<form action="/actionpage.php" method="get">
        <label for="f name">First name:</label>
        <input type="text" id="f name" name="f name"><br>
        <label for="l name">Last name:</label>
       <input type="text" id="l name" name="l name"><br>
       <input type="submit" value="Submit">
</form>
```

Radio button:

Radio buttons are normally presented in radio group. Only one radio button in a group can be selected at the same time.

Note:

- The radio group must share the same name to be treated as a group. Once the radio group is created, selecting any radio button in that group automatically deselects any other selected radio button in the same group. You can have as many radio groups on a page as you want, as long as each group has its own name.
- The value attribute defines the unique value associated with each radio button. The value is not shown to the user, but is the value that is sent to the server on "submit" to identify which radio button that was selected.

Select tag:

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

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

The name attribute is needed to reference the form data after the form is submitted .

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

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

Syntax:

```
<select name="cars" id="cars">
        <option value="volvo">Volvo</option>
        <option value="saab">Saab</option>
```

```
            <option value="mercedes">Mercedes</option>
            <option value="audi">Audi</option>
        </select>
```

Nav tag:

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

Notice that NOT all links of a document should be inside a <nav> element. The <nav> element is intended only for major blocks of navigation links.

Browsers, such as screen readers for disabled users, can use this element to determine whether to omit the initial rendering of this content.

Syntax:

```
<nav>
  <a href="/html/">HTML</a>
  <a href="/css/">CSS</a>
  <a href="/js/">JavaScript</a>
  <a href="/python/">Python</a>
</nav>
```

Source Code:

```
<!DOCTYPE html>

<html lang="en">

<head>

  <meta charset="UTF-8">

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

<title>create signup page</title>
```

```
  <h2>Sign up!</h2>
</head>
<form action="/action.php">
   <label for="username">UserName:</label>
   <input type="text" id="UserName" placeholder="Enter your Username"><br>
   <label for="email"> Email:</label>
   <input type="text" id="Email" placeholder="Enter your Email"><br>
   <label for="Password">Password:</label>
   <input type="text" id="Password" placeholder="Enter your Password" pattern="+"><br>
   <label for="gender">Gender:</label>
   <label for="male">Male<input type="radio" id="male" name="gender" >
   <label for="female">Female<input type="radio" id="female"name="gender">
   </form><br>
   <label for="dob">DOB:</label>
   <input type="date" id="dob" required />
   <br>
   <label for="phone_no">Phone number:</label>
   <input type="text" id="phone_no" placeholder="Enter your contact number" pattern="+"><br>
```

```
<label for="country">Country:</label>
<select name="country" id="country">
<option value="india">India</option>
<option value="usa">USA</option>
<option value="uk">UK</option>
</select><br>
<label for="language known">Language Known:</label>
<label for="English">English<input type="radio" id="English" name="language known" value="E">
<label for="Telugu">Telugu<input type="radio" id="Telugu" name="language known" value="T">
<label for="Hindi">Hindi<input type="radio" id="Hindi" name="language known" value="H"><br>
<button text="register">Register</button>
<button text="reset">Reset</button><br>
<nav> About Us | Privacy Policy | Contact Us | FAQ | Terms & Conditions </nav> Copyright &copy;
</footer>
<aside> Like and Connect with us FB Twitter  </aside>
</body>
</html>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>create signup page</title>
    <h2>Sign up!</h2>
</head>
<form action="/action.php">
    <label for="username">UserName:</label>
    <input type="text" id="UserName" placeholder="Enter your Username"><br>
    <label for="email"> Email:</label>
    <input type="text" id="Email" placeholder="Enter your Email"><br>
    <label for="Password">Password:</label>
    <input type="text" id="Password" placeholder="Enter your Password" pattern="+"><br>
    <label for="gender">Gender:</label>
    <label for="male">Male<input type="radio" id="male" name="gender" >
    <label for="female">Female<input type="radio" id="female"name="gender">
    </form><br>
    <label for="dob">DOB:</label>
    <input type="date" id="dob" required />
    <br>
    <label for="phone_no">Phone number:</label>
    <input type="text" id="phone_no" placeholder="Enter your contact number" pattern="+"><br>
    <label for="country">Country:</label>
    <select name="country" id="country">
        <option value="india">India</option>
        <option value="usa">USA</option>
        <option value="uk">UK</option>
    </select><br>
    <label for="language known">Language Known:</label>
    <label for="English">English<input type="radio" id="English" name="language known" value="E">
    <label for="Telugu">Telugu<input type="radio" id="Telugu" name="language known" value="T">
    <label for="Hindi">Hindi<input type="radio" id="Hindi" name="language known" value="H"><br>
    <button text="register">Register</button>
    <button text="reset">Reset</button><br>
    <nav> About Us | Privacy Policy | Contact Us | FAQ | Terms & Conditions </nav> Copyright &copy;
    </footer>
    <aside> Like and Connect with us FB Twitter  </aside>
    </body>
    </html>
```

Output:

Sign up!

UserName: Enter your Username

Email: Enter your Email

Password: Enter your Password

Gender: Male ○ Female ○

DOB: dd-mm-yyyy

Phone number: Enter your contact number

Country: India

Language Known: English ○ Telugu ○ Hindi ○

Register Reset

About Us | Privacy Policy | Contact Us | FAQ | Terms & Conditions

Copyright ©

Like and Connect with us FB Twitter

Enhancing Signup Page Functionality of IEKart's Shopping Application

Course Name: HTML5- The Language

Module Name: Input Elements- Attributes

Objective

To improve the user experience and validation mechanisms of IEKart's signup page by adding relevant HTML5 attributes to input elements. These enhancements aim to reduce errors, provide better guidance, and streamline the registration process.

Enhancements Made

1. Adding Placeholder Text
    - **Description:** Placeholder text provides users with a hint or example of the data expected in the input field.
    - **Implementation:**
      <input type="text" name="username" placeholder="Enter your username">
2. Using the Required Attribute
    - **Description:** The required attribute ensures that essential fields cannot be left blank.
    - **Implementation:**
      <input type="email" name="email" required placeholder="Enter your email">
3. Setting Input Validation
    - **Description:** Attributes like pattern, maxlength, and minlength enforce specific formats and limits.
    - **Implementation:**
      <input type="password" name="password" minlength="8" maxlength="20" required placeholder="Create a strong password">
4. Enhancing Numeric Inputs
    - **Description:** The type="number" and attributes like min and max provide control over numeric inputs.
    - **Implementation:**
      <input type="number" name="age" min="18" max="100" required placeholder="Enter your age">

5. Improving Mobile-Friendly Forms
    - **Description:** Attributes like type="tel" and inputmode ensure compatibility with touch keyboards on mobile devices.
    - **Implementation:**

      <input type="tel" name="phone" pattern="[0-9]{10}" required placeholder="Enter your 10-digit phone number">
6. Autofocus and Autocomplete
    - **Description:**
        - autofocus highlights an input field when the page loads.
        - autocomplete stores previously entered data for faster filling.
    - **Implementation:**

      <input type="text" name="firstname" autofocus autocomplete="given-name" placeholder="Enter your first name">
7. Date Input
    - **Description:** Simplify date entries using type="date".
    - **Implementation:**

      <input type="date" name="dob" required>

Benefits of Enhancements

1. Enhanced User Experience
    - Simplifies the registration process with clear guidelines and user-friendly features.
2. Improved Data Validation
    - Minimizes errors during data submission with built-in validation mechanisms.
3. Increased Accessibility
    - Ensures compatibility across devices and platforms, including mobile users.
4. Time Efficiency
    - Saves users time by reducing the need for repetitive data entry and errors.

Conclusion

By incorporating HTML5 input attributes, the signup page of IEKart's Shopping application has been made more interactive, intuitive, and reliable. These enhancements not only improve user satisfaction but also ensure robust data handling and validation, paving the way for a seamless registration experience.

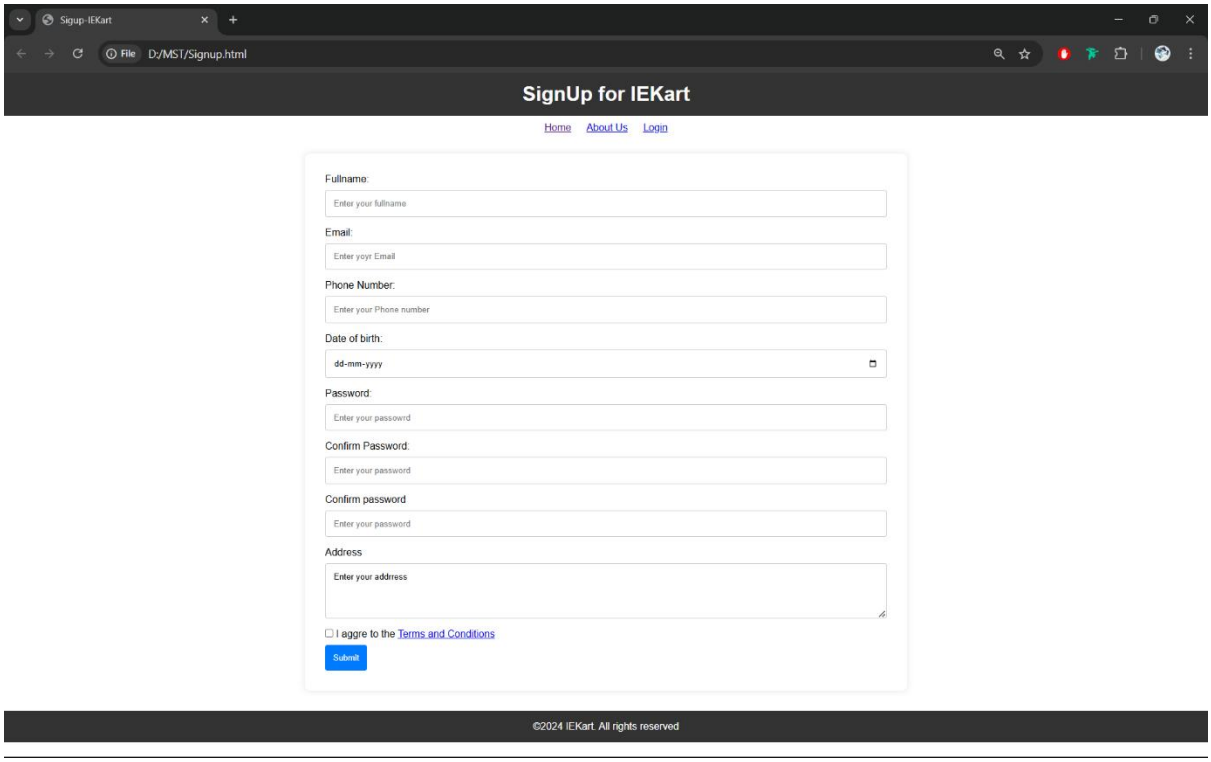
Source code

```
<!DOCTYPE html>
<html lang="en">

<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <link rel="stylesheet" href="style.css">
   <title>Sigup-IEKart</title>
</head>

<body>
   <header>
      <h1>SignUp for IEKart</h1>
   </header>
   <section id="navigation">
      <nav>
         <ul>
            <li><a href="homepage.html">Home</a></li>
            <li><a href="about.html">About Us</a></li>
            <li><a href="Login.html">Login</a></li>
         </ul>
      </nav>
   </section>
   <main>
      <section id="signupform">
         <form action="">
            <label for="text">Fullname:</label>
            <input type="text" id="fullname" name="fullname" placeholder="Enter your fullname"
maxlength="50"
```

```
        autofocus autocomplete="name">
    <label for="email">Email:</label>
    <input type="email" id="emial" name="emial" required placeholder="Enter yoyr Email"
        autocomplete="email">
    <label for="phone">Phone Number:</label>
    <input type="tel" id="phone" name="phone" required placeholder="Enter your Phone
number"
        pattern="[0-9]{10}" title="Enter a vaild 10-digit phone number" autocomplete="tel">
    <label for="dob">Date of birth:</label>
    <input type="date" id="dob" name="dob" required autocomplete="bday-day">
    <label for="password">Password:</label>
    <input type="password" id="password" name="passowrd" placeholder="Enter your
passowrd" minlength="8"
        maxlength="20">
    <label for="passowrd">Confirm Password:</label>
    <input type="password" id="password" name="passowrd" required placeholder="Enter
your password"
        minlength="8" maxlength="20">
    <label for="password">Confirm password</label>
    <input type="password" id="new-password" name="new-password" required
placeholder="Enter your password">
    <label for="address">Address</label>
    <textarea name="address" id="address" maxlength="200" rows="4">Enter your
addrress</textarea>
    <label for="terms">
        <input type="checkbox" id="terms" name="terms" required> I aggre to the <a
href="#">Terms and
            Conditions</a>
    </label>
    <div id="submit">
        <button>Submit</button>
    </div>
</form>
```

```
        </section>
    </main>
    <footer>
        <p>&copy;2024 IEKart. All rights reserved</p>
    </footer>
</body>


</html>
```

Output



Aim: Adding media content in a frame using audio, video, iframe elements to the Home page of IEKart's shopping application

Description:

Why do we need media?

- Grab attention
- Showcase products
- Create a Brand experience
- Boost sales

Audio:

The <audio> element is used to insert audio into the page. The following are some of the attributes of this element:

- src: The path to the music file
- autoplay: Starts playing automatically
- loop: Keeps the music going smoothly in the background
- controls: Allows users to pause, adjust volume, etc.

Video:

The <video> element is used to insert video files in the web page. The following are some of the attributes of this element:

- src: The path to the video file
- autoplay: Starts playing automatically (muted initially)
- loop: Optionally loops the video for continuous play
- controls: Provides play/pause, volume, and full-screen options

Iframe:

The <iframe> element can be used to insert any external content such as music, video, or even text. Any external link can also be accessed using the iframe. The following are some of the attributes for the iframe:

- src: The URL of external content (or any URL)
- width and height: Set the size of the iframe to fit
- frameborder: Define size of the border aroung the iframe
- allow: Specifies permissions for the iframe (e.g., autoplay, fullscreen,etc.).

Source code:

```
<!DOCTYPE html>
<html>
<head>
<meta name="keywords" content="Online, Shopping" />
<title>Ekart Shopping</title>
</head>
<body>
<header>
<nav>
<a href="Login.html"> Login </a> |
<a href="Signup.html">SignUp</a> |
<a href="TrackOrder.html"> Track order </a>
</nav>
```

<h1>Welcome to Ekart's Shopping</h1>

<nav>

 Clothing |

 Media

</nav>

</header>

<article>

Ekart& Co very own one-stop solution for all your shopping needs !

Dont believe us? Click on the offers and check it out for yourself !!

<h2 id="Clothing"> Clothing </h2>

<h2 id="Media"> Media </h2>

Listen to some music:

<audio src="music.mp3" controls ="controls"></audio>

View Video:

<video src="video.mp4" controls="controls" width="350" height="250"></video>

<iframe width="560" height="315" src="video.mp4" title="Video player" frameborder="0" allow="accelerometer; autoplay; clipboard-write; encrypted-media; gyroscope; picture-in-picture; web-share" allowfullscreen></iframe>

</article>

<footer>

<nav>

About Us | Privacy Policy | Contact Us | FAQ | Terms & Conditions

</nav>

Copyright © 2018 | Giri

</footer>

<aside>

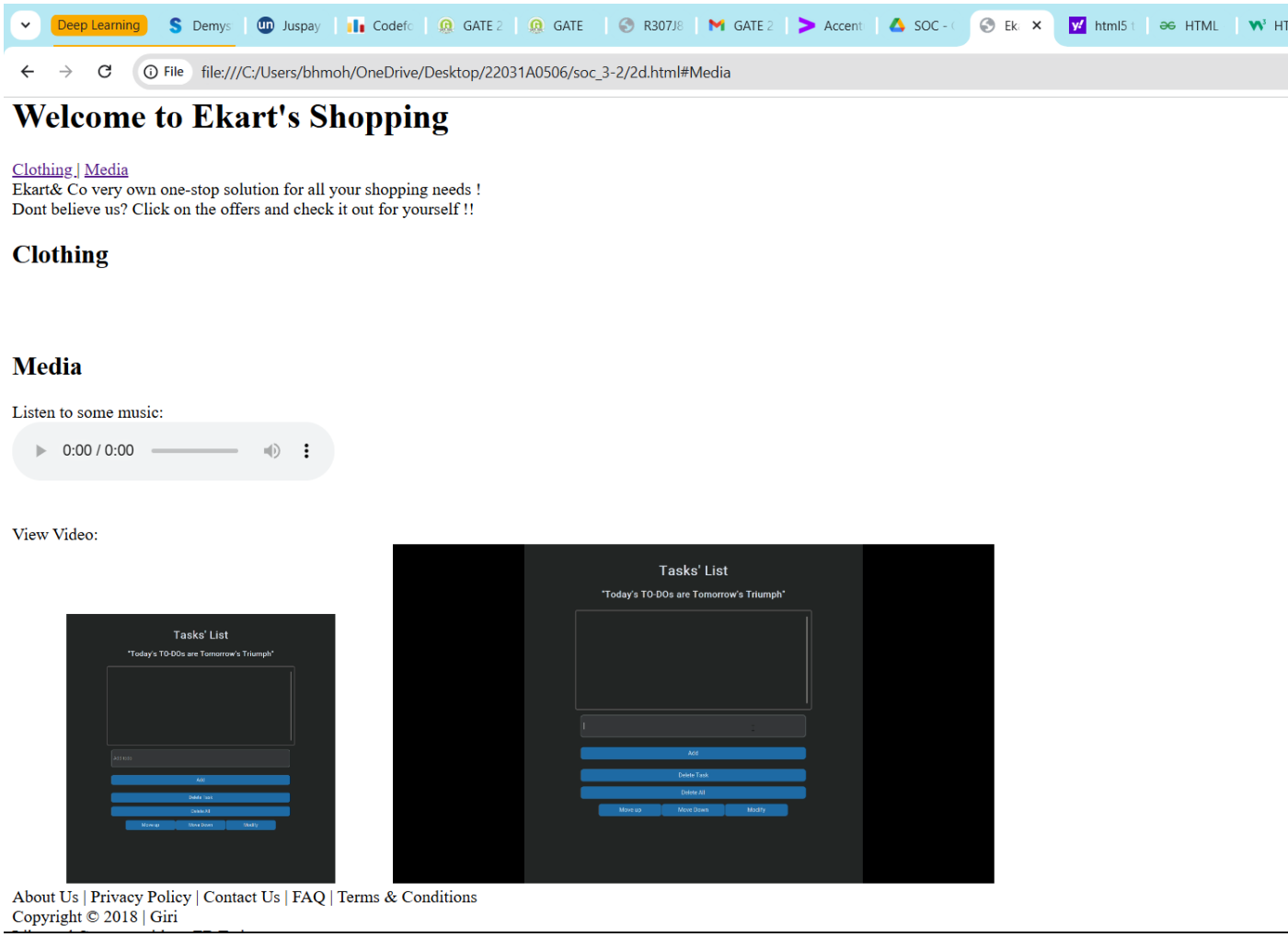
Like and Connect with us FB Twitter g+

</aside>

</body>

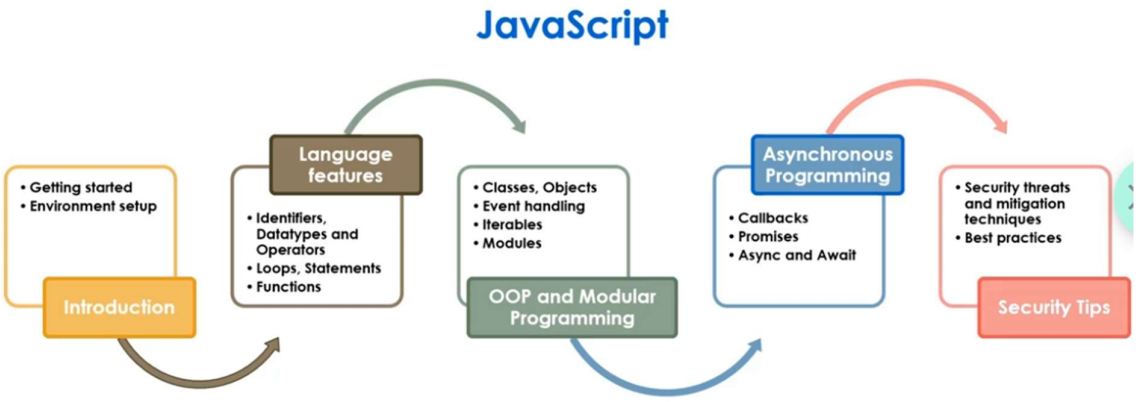
</html>

Output:



Exercise 3 : JavaScript

Js Learning Path

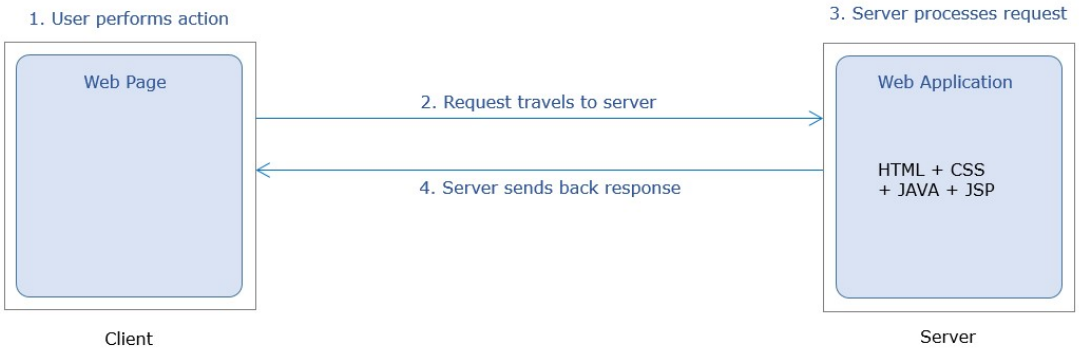


Why JavaScript?

JavaScript was introduced as a full-fledged client-side language used for developing web applications in 1995. JavaScript is easy to learn, debug, and test. It is an event-based, platform-independent, and an interpreted language with all the procedural programming capabilities.

JavaScript got introduced as a Client-Side programming language with the capability of executing user requests on the Client-Side. This could help in reducing the number of request-response cycles between client and server and decrease the network bandwidth thus reducing the overall response time.

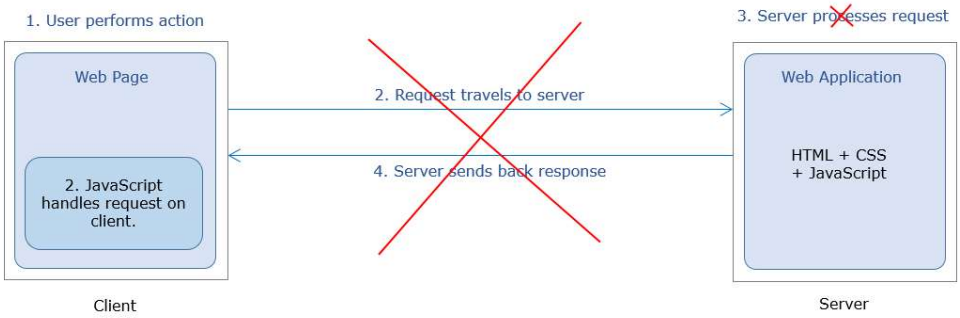
To implement the requirement of handling user action like a click of a button or link and to respond to these requests by displaying the expected output, server-side languages like Java/JSP can be used as shown in the below diagram.



But server-side languages have certain limitations such as :-

- Multiple request-response cycles to handle multiple user requests
- More network bandwidth consumption
- Increased response time

If client-side scripting language JavaScript is used then, this can be done without consulting the server as can be seen in the below diagram.



Ex 3a : JavaScript Identifiers

To model the real-world entities, they must be named to use it in the JavaScript program.

Identifiers are those names that help in naming the elements in JavaScript.

Example:

1. firstName;
2. placeOfVisit;

Identifiers should follow below rules:

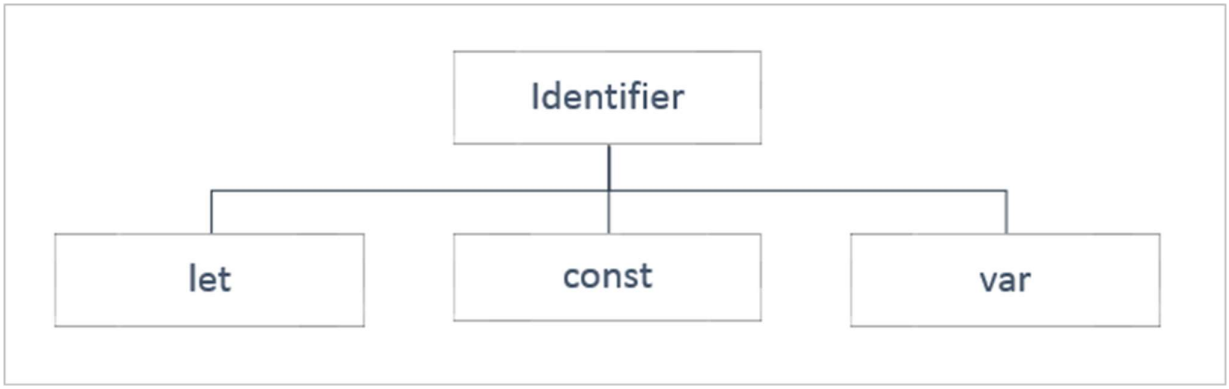
- The first character of an identifier should be letters of the alphabet or an underscore (_) or dollar sign ($).
- Subsequent characters can be letters of alphabets or digits or underscores (_) or a dollar sign ($).
- Identifiers are case-sensitive. Hence, firstName and FirstName are not the same.

Reserved keywords are part of programming language syntax and cannot be used as identifiers.

Types of Identifies

The identifiers in JavaScript can be categorized into three as shown below. They can be declared into specific type based on:

- The data which an identifier will hold and
- The scope of the identifier



let:

An identifier declared using 'let' keyword has a block scope i.e., it is available only within the block in which it is defined.

The value assigned to the identifier can be done either at the time of declaration or later in the code and can also be altered further.

All the identifiers known so far vary in their scope and with respect to the data it holds.

Example:

```
let name="William";
console.log("Welcome to JS course, Mr."+name);

let name = "Goth"; /* This will throw an error because the identifier  'name' has been already declared and we are redeclaring the variable, which is not allowed using the 'let' keyword. */
console.log("Welcome to JS course, Mr."+name);
```

Note: As a best practice, use the **let** keyword for identifier declarations that will change their value over time or when the variable need not be accessed outside the code block. For example, in loops, looping variables can be declared using let as they are never used outside the block.

const

The identifier to hold data that does not vary is called 'Constant' and to declare a constant, 'const' keyword is used, followed by an identifier. The value is initialized during the declaration itself and cannot be altered later.

The identifiers declared using 'const' keyword have block scope i.e., they exist only in the block of code within which they are defined.

Example:

```
const pi = 3.14;
console.log("The value of Pi is: "+pi);

pi = 3.141592; /* This will throw an error because the assignment to a const needs to be done at the time of declaration and it cannot be re-initialized. */
```

```
console.log("The value of Pi is: "+pi);
```

Note: As a best practice, the const declaration can be used for string type identifiers or simple number, functions or classes which does not need to be changed or value

var:

The identifiers declared to hold data that vary are called 'Variables' and to declare a variable, the 'var' keyword is optionally used.

The value for the same can be initialized optionally. Once the value is initialized, it can be modified any number of times later in the program.

Talking about the scope of the identifier declared using 'var' keyword, it takes the Function scope i.e., it is globally available to the Function within which it has been declared and it is possible to declare the identifier name a second time in the same function.

Example:

```
var name = "William";
console.log("Welcome to JS course, Mr." + name);
var name = "Goth"; /* Here, even though we have redeclared the same
identifier, it will not throw any error.*/
console.log("Welcome to JS course, Mr." + name);
```

Note: As a best practice, use the 'var' keyword for variable declarations for function scope or global scope in the program.

Below table shows the difference between let, const and var.

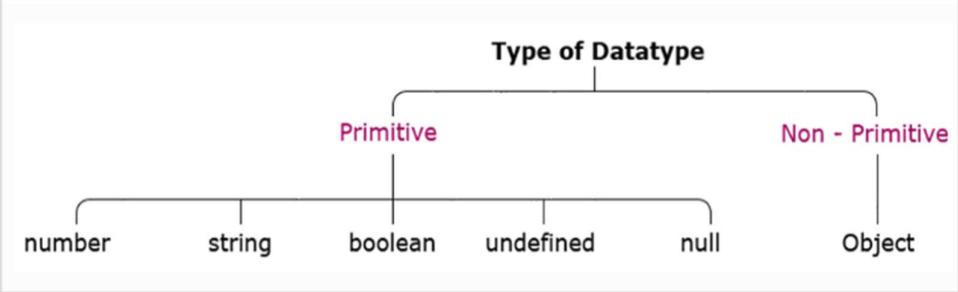
Keyword	Scope	Declaration	Assignment
let	Block	Redeclaration not allowed	Re-assigning allowed
const	Block	Redeclaration not allowed	Re-assigning not allowed
var	Function	Redeclaration allowed	Re-assigning allowed

Problem Solution :

```
1.
let radius=7;
const pi = 3.14;
//let radius=8; //redeclare the radius (o/p): SyntaxError: Identifier 'radius'
has already been declared
radius =8;//re-assign gives the area of new radius i.e 8
const area=pi*radius*radius;
console.log(area);
```

```
2.
var radius=7;
const pi = 3.14;
//var radius=8; //redeclare the radius (o/p): SyntaxError: Identifier 'radius'
has already been declared
radius =8; //re-assign gives the area of new radius i.e 8
const area=pi*radius*radius;
console.log(area);

3.
var radius=7;
const pi = 3.14;
//var radius=8; //redeclare the radius (o/p):SyntaxError: Identifier 'radius'
has already been declared
radius =8;//re-assign gives the area of new radius i.e 8
pi = 3.1452; //TypeError: Assignment to constant variable.

const area=pi*radius*radius;
console.log(area);
```

Ex 3b Datatypes: Primitive and Non primitive



Primitive Datatypes

The data is said to be primitive if it contains an individual value.

Let us explore each of the primitive data types individually.

Number

To store a variable that holds a numeric value, the primitive data type number is used. In almost all the programming languages a number data type gets classified as shown below:

But in JavaScript. the data type number is assigned to the values of type integer, long, float, and double. For example, the variable with number data type can hold values such as 300, 20.50, 10001, and 13456.89.

Constant of type number can be declared like this:

Example:

In JavaScript, any other value that does not belong to the above-mentioned types is not considered as a legal number. Such values are represented as NaN (Not-a-Number).

Example:

String

When a variable is used to store textual value, a primitive data type string is used. Thus, the string represents textual values. String values are written in quotes, either single or double.

Example:

1. let personName= “Rexha”; *//OR*
2. let personName = ‘Rexha’; *// both will have its value as Rexha*

You can use quotes inside a string but that shouldn't match the quotes surrounding the string. Strings containing single quotes must be enclosed within double quotes and vice versa.

Example:

1. let ownership= "Rexha's"; *//OR*
2. let ownership = 'Rexha"s';

This will be interpreted as Rexha's and Rexha"s respectively. Thus, use opposite quotes inside and outside of JavaScript single and double quotes.

But if you use the same quotes inside a string and to enclose the string:

Example:

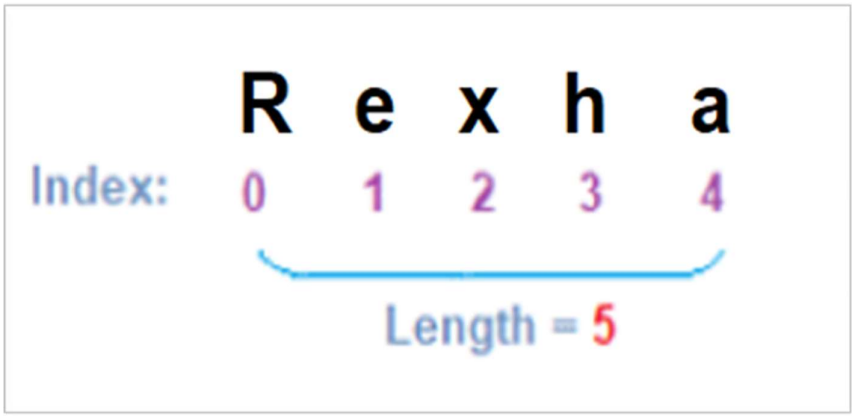
1. let ownership= "Rexha"s"; //OR
2. let ownership = 'Rexha's';

It is a syntax error.

Thus, remember, strings containing single quotes must be enlosed within double quotes and strings containing double quotes must be enclosed within single quotes.

To access any character within the string, it is important to be aware of its position in the string.

The first character exists at index 0, next at index 1, and so on.



Literals

Literals can span multiple lines and interpolate expressions to include their results.

Example:

```
let firstName="Kevin";
let lastName="Patrick";
console.log("Name: "+firstName+" "+lastName+"\n
Email:"+firstName+"_"+lastName+"@abc.com");
/*
OUTPUT:
Name: Kevin Patrick
Email:Kevin_Patrick@abc.com
*/
```

Here, '+' is used for concatenation of identifiers and static content, and '\n' for a new line.

To get the same output, literals can be used as shown below:

```
let firstName="Kevin";
let lastName="Patrick";
console.log(`Name:${firstName} ${lastName}
Email: ${firstName}_${lastName}@abc.com`);
/*
OUTPUT:
Name: Kevin Patrick
Email:Kevin_Patrick@abc.com
*/
```

Boolean

When a variable is used to store a logical value that can always be true or false then, primitive data type Boolean is used. Thus, Boolean is a data type which represents only two values: true and false.

Values such as 100, -5, “Cat”, 10<20, 1, 10*20+30, etc. evaluates to true whereas 0, “”, NaN, undefined, null, etc. evaluates to false.

```
// Booleans
let x = true;
let y = false;
```

Undefined

When the variable is used to store "no value", primitive data type undefined is used.

Any variable that has not been assigned a value has the value undefined and such variable is of type undefined. The undefined value represents "no value".

```
let custName; //here value and the data type are undefined
```

The JavaScript variable can be made empty by assigning the value undefined.

```
let custName = "John"; //here value is John and the data type is String
custName = undefined; //here value and the data type are undefined
```

null

The null value represents "no object".

Null data type is required as JavaScript variable intended to be assigned with the object at a later point in the program can be assigned null during the declaration.

Example 1:

```
let item = null;
// variable item is intended to be assigned with object later. Hence null is
assigned during variable declaration.
```

If required, the JavaScript variable can also be checked if it is pointing to a valid object or null.

```
document.write(item==null);
```

BigInt

It is a special numeric type that provides support for integers of random length.

A BigInt is generated by appending n to the end of an integer literal or by calling the function BigInt that generates BigInt from strings, numbers, etc.

```
const bigintvar = 67423478234689887894747472389477823647n;
OR
const bigintvar = BigInt("67423478234689887894747472389477823647");
const bigintFromNumber = BigInt(10); // same as 10n
```

Non-Primitive Datatypes

The data type is said to be non-primitive if it is a collection of multiple values.

The variables in JavaScript may not always hold only individual values which are with one of the primitive data types.

There are times a group of values are stored inside a variable.

JavaScript gives non-primitive data types named Object and Array, to implement this.

Objects

Objects in JavaScript are a collection of properties and are represented in the form of [key-value pairs].

The 'key' of a property is a string or a symbol and should be a legal identifier.

The 'value' of a property can be any JavaScript value like Number, String, Boolean, or another object.

JavaScript provides the number of built-in objects as a part of the language and user-defined JavaScript objects can be created using object literals.

Syntax:

```
{
    key1 : value1,
    key2 : value2,
    key3 : value3
  };
```

Example

```
let mySmartPhone = {
        name: "iPhone",
        brand: "Apple",
        platform: "iOS",
        price: 50000
    };
```

Array

The Array is a special data structure that is used to store an ordered collection, which cannot be achieved using the objects.

There are two ways of creating an array:

```
let dummyArr = new Array();
//OR
let dummyArr = [];
```

Either array can be declared as empty and can be assigned with value later, or can have the value assigned during the declaration.

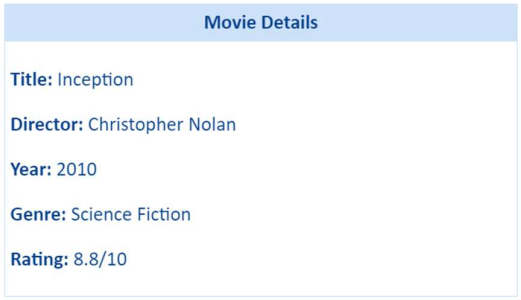
```
digits =[1,2,3,"four"];
```

Problem Statement:

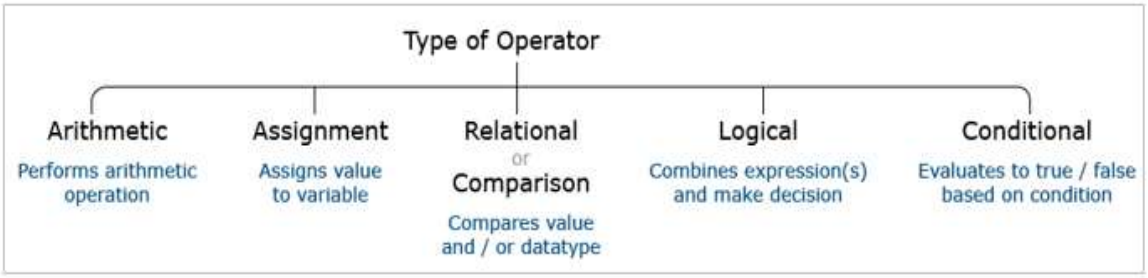
Write JavaScript code to display the movie details such as movie name, language, and ratings. Initialize the variables with values of appropriate types. Use template literals wherever necessary.

```
<script>
// Movie details
const movie = {
title: "Inception",
director: "Christopher Nolan",
year: 2010,
genre: "Science Fiction",
rating: "8.8/10"
};

// Display movie details in the maincontent div
const contentDiv = document.getElementById("maincontent");
```

```
contentDiv.innerHTML = `
<p><strong>Title:</strong> ${movie.title}</p>
<p><strong>Director:</strong> ${movie.director}</p>
<p><strong>Year:</strong> ${movie.year}</p>
<p><strong>Genre:</strong> ${movie.genre}</p>
<p><strong>Rating:</strong> ${movie.rating}</p>
`;
</script>
```

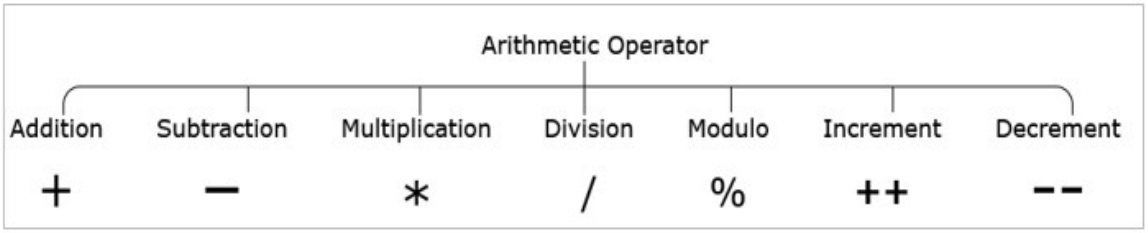


Operators and Types of Operators



Arithmetic

Arithmetic operators are used for performing arithmetic operations




```
let sum = 5 + 3; // sum=8
let difference = 5 - 3; // difference=2
let product = 5 * 3; // product=15
let division = 5/3; // division=1
let mod = 5%3; // mod=2
let value = 5;
value++; // increment by 1, value=6
let value = 10;
value--; // decrement by 1, value=9
```

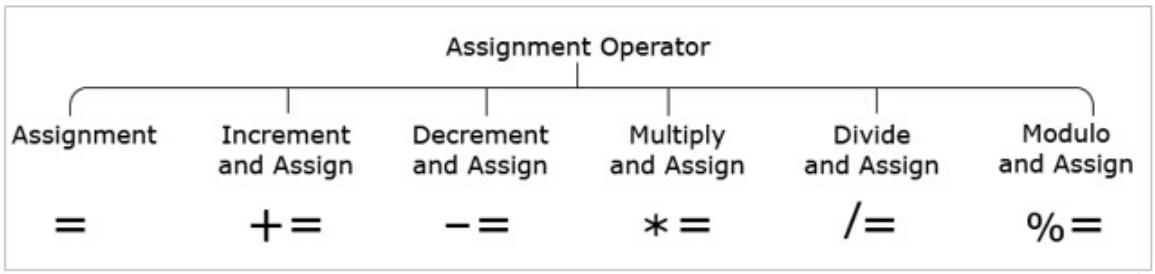
Arithmetic operator '+' when used with string type results in the concatenation.

```
let firstName = "James";
let lastName = "Roche";
let name = firstName + " " + lastName;  // name = James Roche
```

Arithmetic operator '+' when used with a string value and a numeric value, it results in a new string value.

```
let strValue="James";
let numValue=10;
let newStrValue= strValue + " " + numValue;  // newStrValue= James 10
```

Assignment Operators

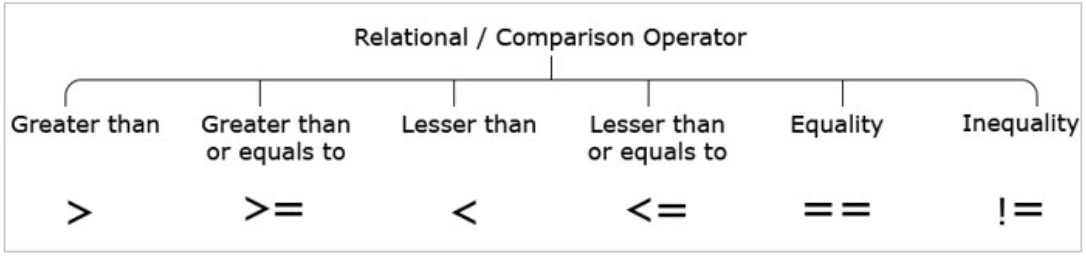



```
let num = 30; // num=30
let num += 10; // num=num+10 => num=40
let num -= 10; // num=num-10 => num=20
let num *= 30; // num=num*30 => num=900
let num /= 10; // num=num/10 => num=3
let num %= 10; // num=num%10 => num=0
```

Relational operators

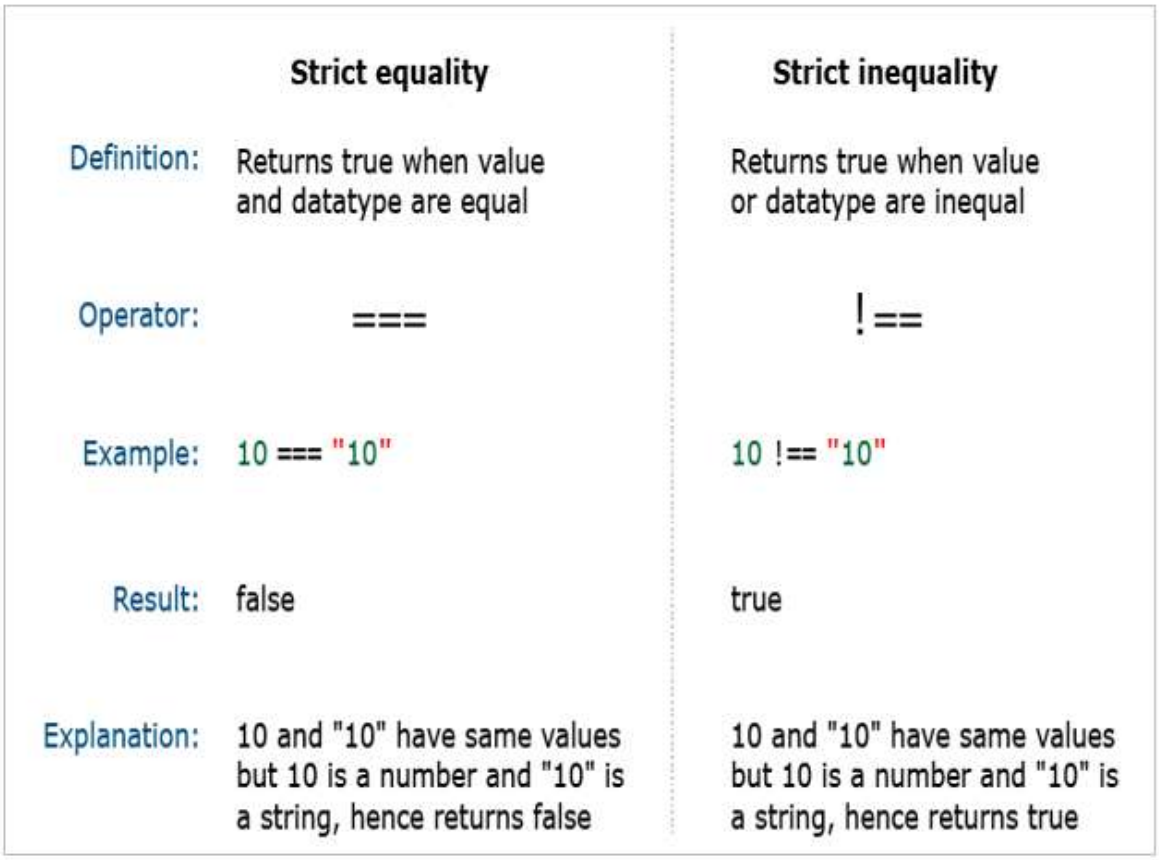
Relational operators are used for comparing values and the result of comparison is always either true or false.

Relational operators shown below do implicit data type conversion of one of the operands before comparison.




```
10 > 10; //false
10 >= 10; //true
10 < 10; //false
10 <= 10; //true
10 == 10; //true
10 != 10; //false
```

Strict Equality



Relational operators shown below compares both the values and the value types without any implicit type conversion.

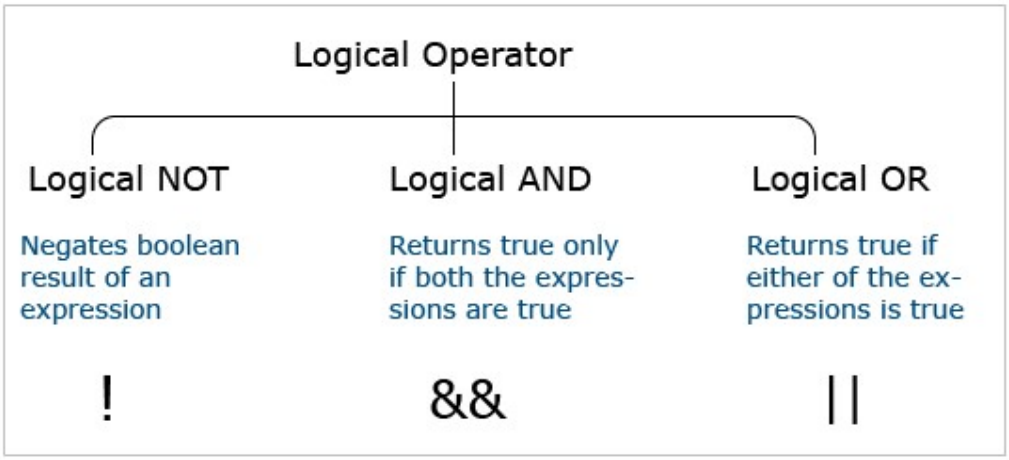
Strict equality (===) and strict inequality (!==) operators consider only values of the same type to be equal.

Hence, strict equality and strict inequality operators are highly recommended to determine whether two given values are equal or not.

Note: As a best practice, you should use === comparison operator when you want to compare value and type, and the rest of the places for value comparison == operator can be used.

Logical Operators

Logical operators allow a program to make a decision based on multiple conditions. Each operand is considered a condition that can be evaluated to true or false.




```
!(10 > 20); //true
(10 > 5) && (20 > 20); //false
(10 > 5) || (20 > 20); //true
```

typeof Operator

"typeof" is an operator in JavaScript.

JavaScript is a loosely typed language i.e., the type of variable is decided at runtime based on the data assigned to it. This is also called dynamic data binding.

As programmers, if required, the typeof operator can be used to find the data type of a JavaScript variable.

The following are the ways in which it can be used and the corresponding results that it returns.

```
typeof "JavaScript World" //string
typeof 10.5 // number
typeof 10 > 20 //boolean
typeof undefined //undefined
typeof null //Object
typeof {itemPrice : 500} //Object
```

Problem Statement

Write JavaScript code to do online booking of theatre tickets and calculate the total price, considering the price per ticket as $9. Also, apply a festive season discount of 10% and calculate the discounted amount.

```
<div id="heading">
<b>Theatre Drama</b>
</div>
<div id="maincontent">
<h3>Your Ticket Details :</h3>
<p id="seats"></p>
<p id="totalPrice"></p>
<p id="discount"></p>
<p id="discountAmount"></p>
<p id="finalPrice"></p>
</div>
<script>
    // Define the number of seats booked and the price per ticket
    let seats = 3; // Number of tickets booked
    let pricePerTicket = 9; // Price per ticket in dollars

    // Calculate the total price without discount
    let totalPrice = seats * pricePerTicket;

    // Define the festive discount percentage
    let discountPercentage = 10; // Discount in percentage

    // Calculate the discount amount
    let discountAmount = (totalPrice * discountPercentage) / 100;

    // Calculate the final price after applying the discount
    let discountedPrice = totalPrice - discountAmount;

    // Display the calculated information
    document.getElementById("seats").innerHTML = "The number of seats booked:
" + seats;
    document.getElementById("totalPrice").innerHTML = "Total cost of tickets
before discount: $" + totalPrice;
    document.getElementById("discount").innerHTML = "Festive season discount:
" + discountPercentage + "%";
    // document.getElementById("discountAmount").innerHTML = "Discount amount:
$" + discountAmount.toFixed(2);
    document.getElementById("finalPrice").innerHTML = "Total cost after
discount: $" + discountedPrice.toFixed(2);

</script>
```

Theatre Drama

Your Ticket Details :

The number of seats booked: 3

Total cost of tickets before discount: $27

Festive season discount: 10%

Total cost after discount: $24.30

Ex 3d JavaScript statements

JavaScript if, else, and else if

- Conditional statements are used to perform different actions based on different conditions.

Conditional Statements

Very often when you write code, you want to perform different actions for different decisions.

You can use conditional statements in your code to do this.

In JavaScript we have the following conditional statements:

- Use if to specify a block of code to be executed, if a specified condition is true
- Use else to specify a block of code to be executed, if the same condition is false
- Use else if to specify a new condition to test, if the first condition is false
- Use switch to specify many alternative blocks of code to be executed

The if Statement

Use the `if` statement to specify a block of JavaScript code to be executed if a condition is true.

Syntax

```
if (condition) {
  //  block of code to be executed if the condition is true
}
```

The else Statement

Use the `else` statement to specify a block of code to be executed if the condition is false.

```
if (condition) {
  //  block of code to be executed if the condition is true
} else {
  //  block of code to be executed if the condition is false
}
```

Example

If the hour is less than 18, create a "Good day" greeting, otherwise "Good evening":

```
if (hour < 18) {
  greeting = "Good day";
} else {
  greeting = "Good evening";
}
```

The result of greeting will be:

```
Good day
```

The else if Statement

Use the `else if` statement to specify a new condition if the first condition is false.

Syntax

```
if (condition1) {
  //  block of code to be executed if condition1 is true
} else if (condition2) {
  //  block of code to be executed if the condition1 is false and
condition2 is true
} else {
  //  block of code to be executed if the condition1 is false and
condition2 is false
}
```

Example

If time is less than 10:00, create a "Good morning" greeting, if not, but time is less than 20:00, create a "Good day" greeting, otherwise a "Good evening":

```
if (time < 10) {
  greeting = "Good morning";
} else if (time < 20) {
  greeting = "Good day";
} else {
  greeting = "Good evening";
}
```

The result of greeting will be: Good Morning

JavaScript Switch Statement

The `switch` statement is used to perform different actions based on different conditions.

The JavaScript Switch Statement

Use the `switch` statement to select one of many code blocks to be executed.

Syntax

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

This is how it works:

- The switch expression is evaluated once.
- The value of the expression is compared with the values of each case.
- If there is a match, the associated block of code is executed.
- If there is no match, the default code block is executed.

Example

The `getDay()` method returns the weekday as a number between 0 and 6.

(Sunday=0, Monday=1, Tuesday=2 ..)

This example uses the weekday number to calculate the weekday name:

```
switch (new Date().getDay()) {
  case 0:
    day = "Sunday";
    break;
  case 1:
    day = "Monday";
    break;
```

```
  case 2:
     day = "Tuesday";
    break;
  case 3:
    day = "Wednesday";
    break;
  case 4:
    day = "Thursday";
    break;
  case 5:
    day = "Friday";
    break;
  case 6:
    day = "Saturday";
}
```

The result of day will be:

```
Friday
```

The break Keyword

When JavaScript reaches a `break` keyword, it breaks out of the switch block.

This will stop the execution inside the switch block.

It is not necessary to break the last case in a switch block. The block breaks (ends) there anyway.

Note: If you omit the break statement, the next case will be executed even if the evaluation does not match the case.

The default Keyword

The `default` keyword specifies the code to run if there is no case match:

Example

The `getDay()` method returns the weekday as a number between 0 and 6.

If today is neither Saturday (6) nor Sunday (0), write a default message:

```
switch (new Date().getDay()) {
  case 6:
    text = "Today is Saturday";
    break;
  case 0:
    text = "Today is Sunday";
    break;
  default:
    text = "Looking forward to the Weekend";
}
```

The result of text will be:

```
Looking forward to the Weekend
```

The `default` case does not have to be the last case in a switch block:

Example

```
switch (new Date().getDay()) {
  default:
    text = "Looking forward to the Weekend";
    break;
  case 6:
    text = "Today is Saturday";
    break;
  case 0:
    text = "Today is Sunday";
}
```

If `default` is not the last case in the switch block, remember to end the default case with a break.

Common Code Blocks

Sometimes you will want different switch cases to use the same code.

In this example case 4 and 5 share the same code block, and 0 and 6 share another code block:

Example

```
switch (new Date().getDay()) {
  case 4:
  case 5:
    text = "Soon it is Weekend";
    break;
```

```
  case 0:
  case 6:
    text = "It is Weekend";
    break;
  default:
    text = "Looking forward to the Weekend";
}
```

Switching Details

If multiple cases matches a case value, the **first** case is selected.

If no matching cases are found, the program continues to the **default** label.

If no default label is found, the program continues to the statement(s) **after the switch**.

Strict Comparison

Switch cases use **strict** comparison (===).

The values must be of the same type to match.

A strict comparison can only be true if the operands are of the same type.

In this example there will be no match for x:

Example

```
let x = "0";
switch (x) {
  case 0:
    text = "Off";
    break;
  case 1:
    text = "On";
    break;
  default:
    text = "No value found";
}
```

The `break` statement can also be used to jump out of a loop:

Example

```
for (let i = 0; i < 10; i++) {
  if (i === 3) { break; }
  text += "The number is " + i + "<br>";
}
```

In the example above, the `break` statement ends the loop ("breaks" the loop) when the loop counter (i) is 3.

Our Code :Problem Statement :

Write a JavaScript code to book movie tickets online and calculate the total price based on the 3 conditions:

(a) If seats to be booked are not more than 2, the cost per ticket remains $ 8.

(b) If seats are 6 or more, booking is not allowed

(c) If seats to be booked are more than 2 but less than 5, based on the number of seats booked, do the following:

- Calculate total cost by applying discounts of 5, 7, 9, 11 percent, and so on for customer 1,2,3 and 4.
- Try the code with different values for the number of seats.

Our Code :

```
<!DOCTYPE html>
<html>
<head>
    <title>Booking Details</title>
    <style>
        div#maincontent {
            height: 200px;
            width: 600px;
            border: 1px solid #CEE2FA;
            text-align: left;
            color: #08438E;
            font-family: calibri;
            font-size: 20;
            padding: 5px;
        }
        div#heading {
            text-decoration: bold;
```

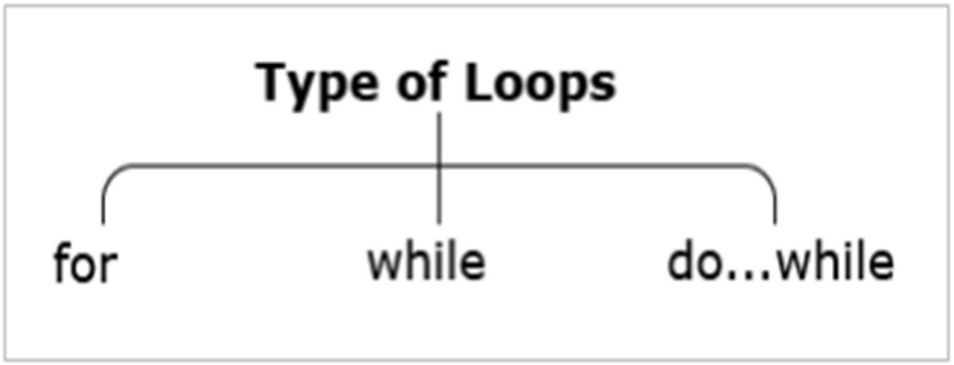
```
            text-align: center;
            margin-top: 80px;
            width: 600px;
            border: 1px solid #CEE2FA;
            text-align: center;
            color: #08438E;
            background-color: #CEE2FA;
            font-family: calibri;
            font-size: 20;
            padding: 5px;
        }
        h4 {
            padding: 0;
            margin: 0;
        }
    </style>
</head>
<body>
    <center>
        <div id="heading">
            <b>Theatre Drama</b>
        </div>
        <div id="maincontent">
            <h4>Your Ticket Details:</h4>
            <br>
            <script>

                function countedDiscount(seat){
                let cost = 8
                if( seat>0 && seat<=2){
                let totalCost = seat*cost;
                document.write("Ticket for Customer 1 get 0% discount,Cost is:
"+totalCost/2+"<br>");
                document.write("Ticket for Customer 2 get 0% discount,Cost is:
"+totalCost/2+"<br>");
                document.write("Total cost after discount is: "+totalCost);
                }
                else if(seat>=6){
                document.write("Booking is not allowed!");
                }
                else if(seat>2 && seat<5) {
                const discount = [];
                let dis = 5;
                for (let i = 0; i < seat; i++) {
                const discounts = {

                customer: dis
                };
```

```
                discount.push(discounts);
                dis +=2;
                }
                const keys = Object.keys(discount);
                let finalCost = 0;

                for (let i = 0; i < discount.length; i++){
                const currentDiscount = discount[i].customer;
                let disCost = 9-(currentDiscount/100)*9;
                finalCost += disCost ;

                document.write("Ticket for Customer "+(i+1)+ " gets "+
currentDiscount +" % discount!,Cost is: $"+disCost+"<br>");
                }
                document.write("For "+seat+" tickets,you need to pay:
$"+finalCost+" instead of $",cost*seat);
                }
                }
                countedDiscount(4);
                </script>
        </div>
    </center>
</body>
</html>
```

OUTPUT

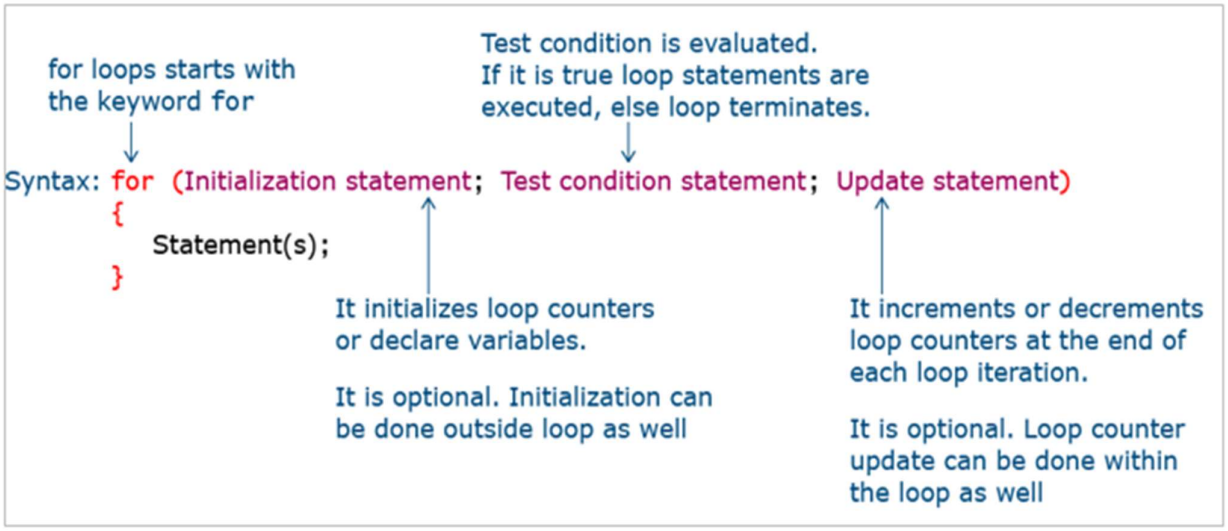
Ex 3e Loops

JavaScript supports popular looping statements as shown below:



FOR LOOP

'for' loop is used when the block of code is expected to execute for a specific number of times. To implement it, use the following syntax.



Example: Below example shows incrementing variable counter five times using 'for' loop:

Also, shown below is output for every iteration of the loop.

```
let counter = 0;

for (let loopVar = 0; loopVar < 5; loopVar++) {

counter = counter + 1;

console.log(counter);
```

}

Here, in the above loop

let loopVar=0; // Initialization

loopVar < 5; // Condition

loopVar++; // Update

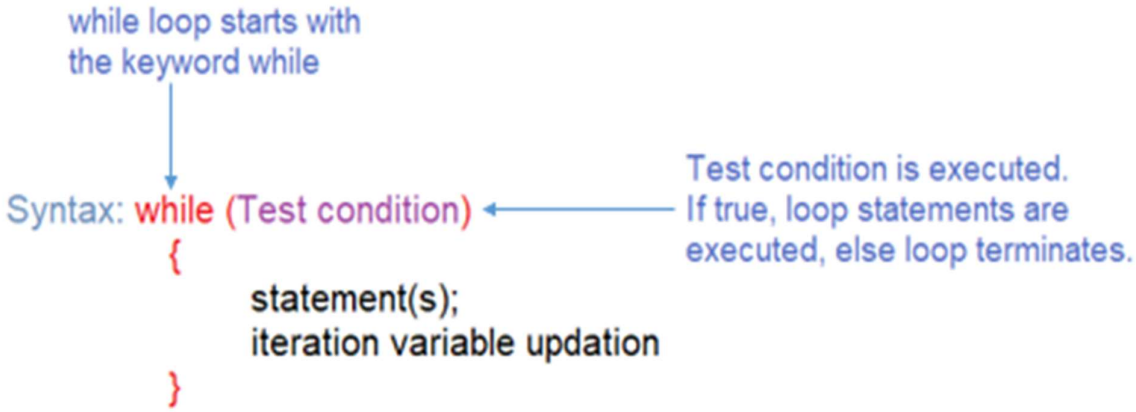
counter = counter + 1; // Action

To understand loops better refer the below table:

loopVar	**counter**
0	1
1	2
2	3
3	4
4	5

WHILE LOOP

while' loop is used when the block of code is to be executed as long as the specified condition is true. To implement the same, the following syntax is used:



The value for the variable used in the test condition should be updated inside the loop only.

Example: The below example shows an incrementing variable counter five times using a 'while' loop.

Also, shown below is the output for every iteration of the loop.

```
let counter = 0;
let loopVar = 0;
while (loopVar < 5) {
console.log(loopVar);
counter++;
loopVar++;
console.log(counter);
}
```

Here, in the above loop

```
let counter=0; // Initialization
    let loopVar=0; // Initialization
    loopVar < 5; // Condition
    loopVar++; // Update
    counter++; // Action
```

To understand loops better refer the below table:

loopVar	counter
0	1
1	2
2	3
3	4
4	5

DO-WHILE

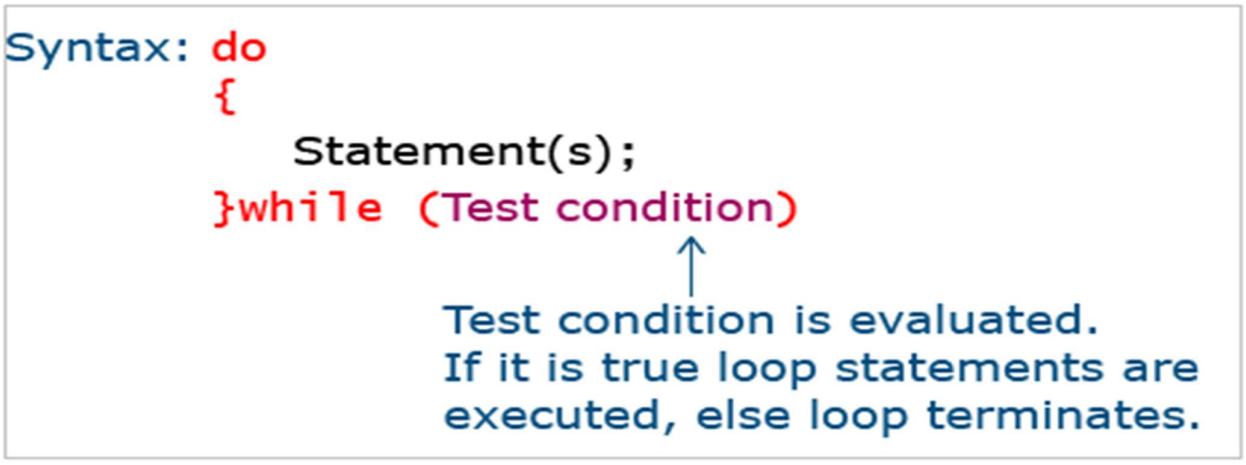
do-while' is a variant of 'while' loop.

This will execute a block of code once before checking any condition.

Then, after executing the block it will evaluate the condition given at the end of the block of code.

Now the statements inside the block of code will be repeated till condition evaluates to true.

To implement 'do-while' loop, use the following syntax:



The value for the variable used in the test condition should be updated inside the loop only.

Example: Below example shows incrementing variable counter five times using 'do-while' loop:

Also, shown below is output for every iteration of the loop.

```
let counter = 0;

let loopVar = 0;

do {

console.log(loopVar);

counter++;

loopVar++;

console.log(counter);

}

while (loopVar < 5);
```

Here, in the above loop

```
let counter=0; // Initialization

let loopVar=0; // Initialization

loopVar < 5; // Condition

loopVar++; // Update

counter++; // Action
```

To understand loops better refer the below table:

loopVar	counter
0	1
1	2
2	3
3	4
4	5

Objective:

To calculate ticket prices for varying seat numbers using a loop.

Code Example:

```
const pricePerTicket = 150;

for (let seats = 1; seats <= 7; seats++) {

if (seats <= 2) {

console.log(`Seats: ${seats}, Total Price: Rs. ${seats * pricePerTicket}`);

} else if (seats >= 6) {

console.log(`Seats: ${seats}, Booking not allowed.`);

} else {

console.log(`Seats: ${seats}, Total Price: Rs. ${seats * pricePerTicket}`);

}

}
```

Explanation:

- **For Loop: Iterates through seat numbers 1 to 7.**
- **Conditional Logic: Determines pricing or denies booking based on seat numbers.**

Input:

- **Price Per Ticket: Rs. 150**
- **Seats: 1 to 7**

Output:

```
Seats: 1, Total Price: Rs. 150
Seats: 2, Total Price: Rs. 300
Seats: 3, Total Price: Rs. 450
Seats: 4, Total Price: Rs. 600
Seats: 5, Total Price: Rs. 750
Seats: 6, Booking not allowed.
Seats: 7, Booking not allowed.
```

Exercise-4a

Description:

*A JavaScript function is a block of code designed to perform a particular task.

*A JavaScript function is executed when "something" invokes it (calls it).

*A JavaScript function is defined with the function keyword, followed by a **name**, followed by parentheses **()**.

*Function names can contain letters, digits, underscores, and dollar signs (same rules as variables).

*The parentheses may include parameter names separated by commas: (*parameter1, parameter2, ...*)

*The code to be executed, by the function, is placed inside curly brackets: {}

Syntax:

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

Functions are one of the integral components of JavaScript. A JavaScript function is a set of statements that performs a specific task. They become a reusable unit of code.

Types of Functions:

JavaScript has two types of functions.

1. User-defined functions

- JavaScript allows to write own functions called as user-defined functions. The user-defined functions can also be created using a much simpler syntax called arrow functions.

2. Built-in functions

- JavaScript provides several predefined functions that perform tasks such as displaying dialog boxes, parsing a string argument, timing-related operations, and so on.

Declaring and Invoking Function:

A function declaration also called a function definition, consists of the function keyword, followed by:

- Function name
- A list of parameters to the function separated by commas and enclosed in parentheses, if any.A set of JavaScript statements that define the function, also called a function body, enclosed in curly brackets {…}.

Syntax for Function Declaration:

```
function function_name(parameter 1, parameter 2 , …, parameter n) {
 }
```

Example:

```
function multiply(num1, num2) {
    return num1 * num2;
}
```

The code written inside the function body will be executed only when it is invoked or called.

Syntax for Function Invocation:

```
function_name(argument 1, argument 2, ..., argument n);
```

Example:

```
multiply (5,6);
```

Arrow Function:

In JavaScript, functions are first-class objects. This means, that you can assign a function as a value to a variable. For example,

```
let sayHello = function () {
  console.log("Welcome to JavaScript");
}
sayHello();
```

Here, a function without a name is called an anonymous function which is assigned to a variable sayHello.

JavaScript has introduced a new and concise way of writing functions using arrow notation. The arrow function is one of the easiest ways to declare an anonymous function.

There are two parts for the Arrow function syntax:

1. let sayHello = ()

- This declares a variable sayHello and assigns a function to it using () to just say that the variable is a function.

2. => { }

- This declares the body of the function with an arrow and the curly braces.

Function Parameters:

Function parameters are the variables that are defined in the function definition and the values passed to the function when it is invoked are called arguments.In JavaScript, function definition does not have any data type specified for the parameters, and type checking is not performed on the arguments passed to the function.

```
function multiply(num1, num2) {
    if (num2 == undefined) {
        num2 = 1;
    }
    return num1 * num2;
}
console.log(multiply(5, 6)); // 30
console.log(multiply(5)); // 5
```

Nested function:

In JavaScript, it is perfectly normal to have functions inside functions. The function within another function body is called a nested function.

The nested function is private to the container function and cannot be invoked from outside the container function.

```
function giveMessage(message) {
    let userMsg = message;
    function toUser(userName) {
        let name = userName;
        let greet = userMsg + " " + name;
        return greet;
    }
    userMsg = toUser("Bob");
    return userMsg;
}

console.log(giveMessage("The world says hello dear: "));
// The world says hello dear: Bob
```

Built-in functions:

JavaScript comes with certain built-in functions. To use them, they need to be invoked.

alert()	It throws an alert box and is often used when user interaction is required to decide whether execution should proceed or not.	alert("Let us proceed");
confirm()	It throws a confirm box where user can click "OK" or "Cancel". If "OK" is clicked, the function returns "true", else returns "false".	let decision = confirm("Shall we proceed?");
prompt()	It produces a box where user can enter an input. The user input may be used for some processing later. This function takes parameter of type string which represents the label of the box.	let userInput = prompt("Please enter your name:");

parseInt()	This function parses string and returns an integer number.
	
	It takes two parameters. The first parameter is the string to be parsed. The second parameter represents radix which is an integer between 2 and 36 that represents the numerical system to be used and is optional.
	
	The method stops parsing when it encounters a non-numerical character and returns the gathered number.
	
	It returns NaN when the first non-whitespace character cannot be converted to number.
		parseInt("10"); //10
		
		parseInt("10 20 30"); //10, only the integer part is returned
		
		parseInt("10 years"); //10
		
		parseInt("years 10"); //NaN, the first character stops the parsing

JavaScript provides two-timer built-in functions. Let us explore these timer functions.

Built-in functions	**Description**	**Description**

setTimeout()	It executes a given function after waiting for the specified number of milliseconds.
	
	It takes 2 parameters. First is the function to be executed and the second is the number of milliseconds after which the given function should be executed.
		function executeMe(){
		
		console.log("Function says hello!")
		
		}
		
		setTimeout(executeMe, 3000);
		
		//It executes executeMe() after 3 seconds.

clearTimeout()	It cancels a timeout previously established by calling setTimeout().
	
	It takes the parameter "timeoutID" which is the identifier of the timeout that can be used to cancel the execution of setTimeout(). The ID is returned by the setTimeout().
		function executeMe(){
		
		console.log("Function says hello!")
		
		}
		
		let timerId= setTimeout(executeMe, 3000);
		
		clearTimeout(timerId);

setInterval()	It executes the given function repetitively.
	
	It takes 2 parameters, first is the function to be executed and second is the number of milliseconds. The function executes continuously after every given number of milliseconds.
		function executeMe(){
		
		console.log("Function says hello!");
		
		}
		
		setInterval(executeMe,3000);
		
		//It executes executeMe() every 3 seconds

Variable Scope in functions:

Variable declaration in the JavaScript program can be done within the function or outside the function. But the accessibility of the variable to other parts of the same program is decided based on the place of its declaration. This accessibility of a variable is referred to as scope.

JavaScript scopes can be of three types:

- Global scope
- Local scope
- Block scope

Global scope:

Variables defined outside function have Global Scope and they are accessible anywhere in the program.

Local scope:

Variables declared inside the function would have local scope. These variables cannot be accessed outside the declared function block.If a local variable is declared without the use of keyword 'var', it takes a global scope.

Block Scope:

BLOCK SCOPE in JavaScript refers to the scope of variables and functions that are defined within a block of code, such as within a pair of curly braces {}.
Variables and functions declared with let and const keywords have block scope.
Variables declared inside a block are only accessible within that block and any nested blocks. They are not accessible outside of the block in which they are defined.
This means that variables declared within a block cannot be accessed before their declaration or outside of the block.

Aim:

Write a JavaScript code to book movie tickets online and calculate the total price based on the 3 conditions:

(a) If seats to be booked are not more than 2, the cost per ticket remains Rs. 150.

(b) If seats are 6 or more, booking is not allowed.

(c) If seats to be booked are more than 2 but less than 5, based on the number of seats booked, do the following:

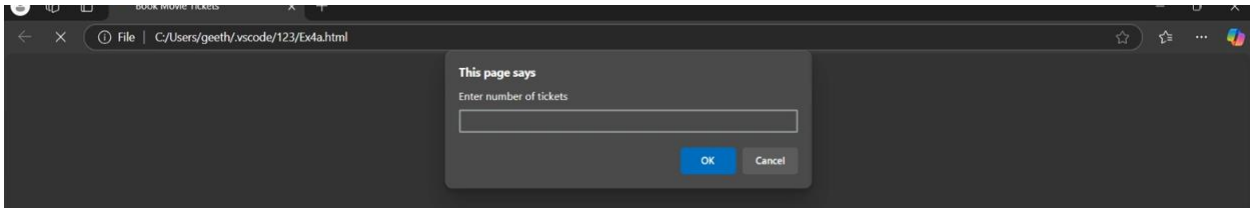
o Calculate total cost by applying discounts of 5, 7, 9, 11 percent, and so on for customer 1,2,3 and 4.

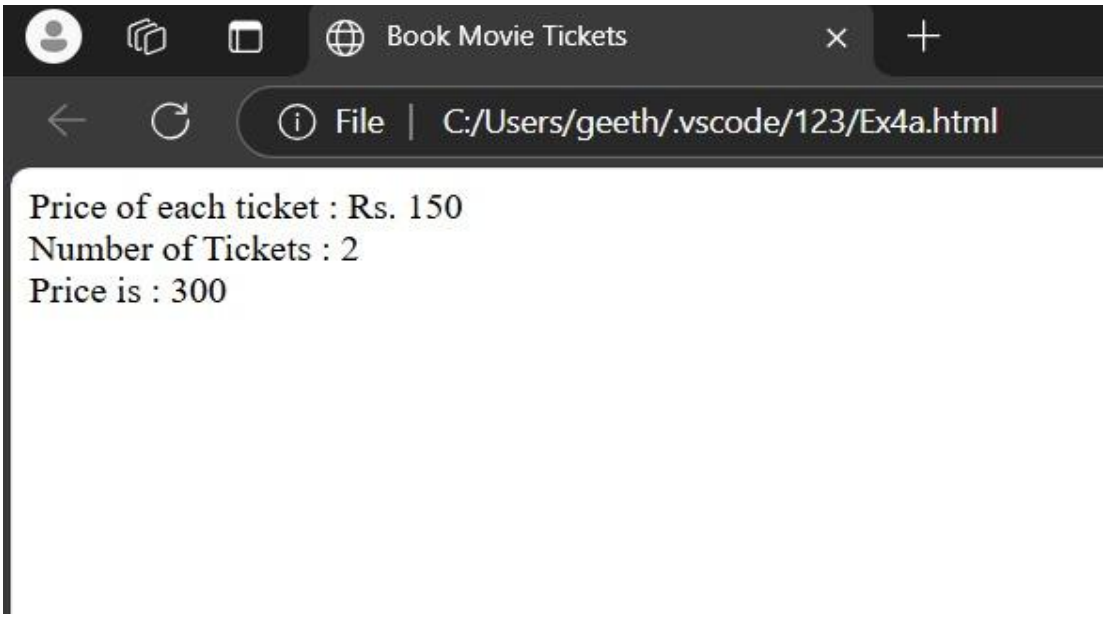
o Try the code with different values for the number of seats.

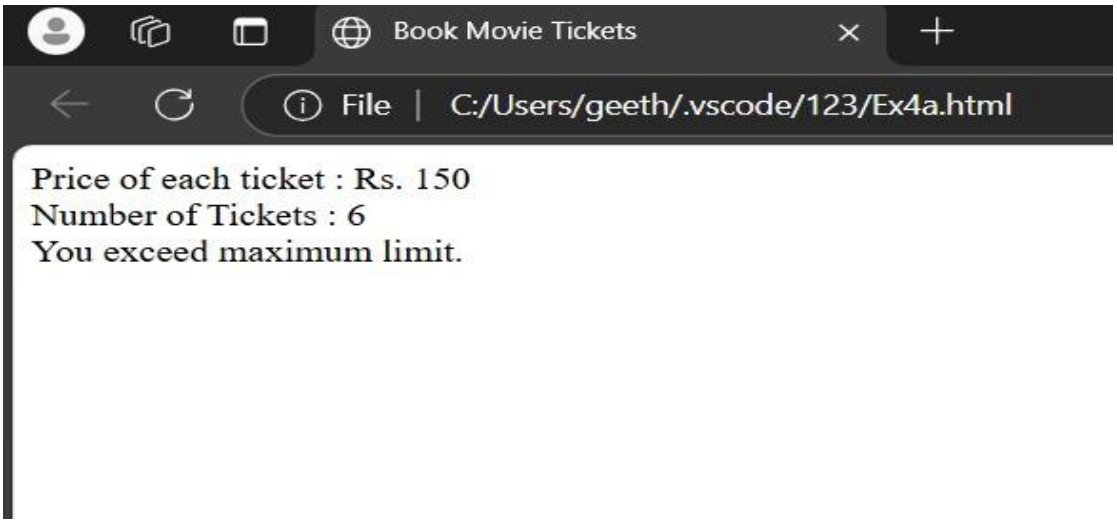
Source code:

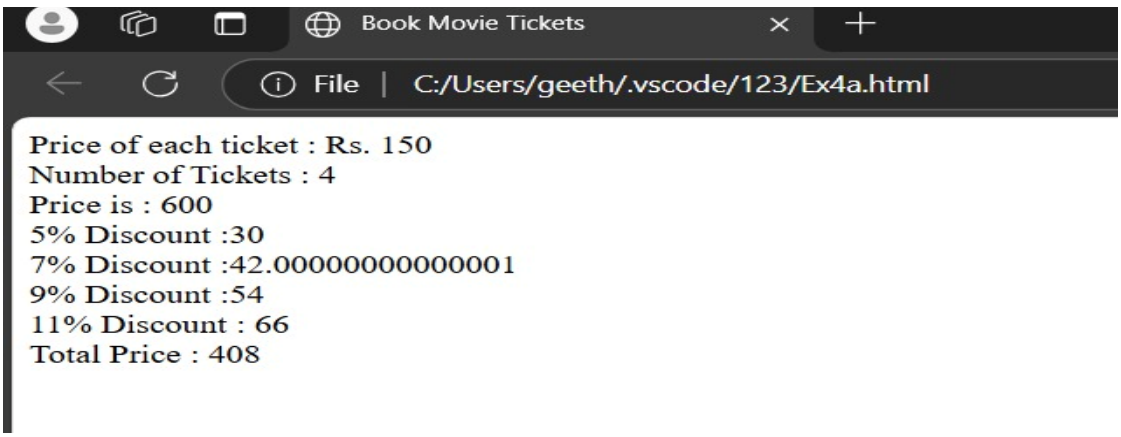
```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Book Movie Tickets</title>
        </head>
        <body>
            <script> var noOfTickets = parseInt(window.prompt("Enter number of tickets"))
            BookMovieTickets(noOfTickets)
            function BookMovieTickets(noOfTickets){
                document.write("Price of each ticket : Rs. 150")
                document.write("<br>Number of Tickets : "+noOfTickets)
                var price = noOfTickets*150
                if(noOfTickets<=2){
                    document.write("<br>Price is : "+price)
                }
                else if(noOfTickets>=6){
                    document.write("<br>You exceed maximum limit.")
                }
                else{
                    document.write("<br>Price is : "+price)
                    document.write("<br>5% Discount :"+price*0.05)
                    document.write("<br>7% Discount :"+price*0.07)
                    document.write("<br>9% Discount :"+price*0.09)
                    document.write("<br>11% Discount : "+price*0.11)
                    document.write("<br>Total Price : "+(price -(price*0.05+price*0.07+price*0.09+price*0.11)))
                    }
                }
            </script>
        </body>
</html>

```

Input & Output:




Book Movie Tickets
File | C:/Users/geeth/.vscode/123/Ex4a.html
Price of each ticket : Rs. 150
Number of Tickets : 2
Price is : 300


Book Movie Tickets
File | C:/Users/geeth/.vscode/123/Ex4a.html
Price of each ticket : Rs. 150
Number of Tickets : 6
You exceed maximum limit.


Book Movie Tickets
File | C:/Users/geeth/.vscode/123/Ex4a.html
Price of each ticket : Rs. 150
Number of Tickets : 4
Price is : 600
5% Discount :30
7% Discount :42.00000000000001
9% Discount :54
11% Discount : 66
Total Price : 408

Exercise-4b

Description:

In 2015, JavaScript introduced the concept of the Class.

- Classes and Objects in JavaScript coding can be created similar to any other Object-Oriented language.
- Classes can also have methods performing different logic using the class properties respectively.

Classes:

- In 2015, ECMAScript introduced the concept of classes to JavaScript
- The keyword class is used to create a class
- The constructor method is called each time the class object is created and initialized.

Example:

The below code demonstrates a calculator accepting two numbers to do addition and subtraction operations.

```
class Calculator {
    constructor(num1, num2){  // Constructor used for initializing the class instance

        /* Properties initialized in the constructor */
        this.num1 = num1;
        this.num2 = num2;
    }

    /* Methods of the class used for performing operations */
    add() {
        return this.num1 + this.num2;
    }

    subtract() {
        return this.num1 - this.num2;
    }
}

let calculator = new Calculator(300, 100); // Creating Calculator class object or instance
console.log("Add method returns" + calculator.add()); // Add method returns 400.
console.log("Subtract method returns" + calculator.subtract()); // Subtract method returns 200.
```

Inheritance:

*In JavaScript, one class can inherit another class using the extends keyword. The subclass inherits all the methods (both static and non-static) of the parent class.

*Inheritance enables the reusability and extensibility of a given class.

*JavaScript uses prototypal inheritance which is quite complex and unreadable. But, now you have '**extends**' keyword which makes it easy to inherit the existing classes.

*Keyword **super** can be used to refer to base class methods/constructors from a subclass

Example:

```
class Vehicle {
    constructor(make, model) {

        /* Base class Vehicle with constructor initializing two-member attributes */
        this.make = make;
        this.model = model;
    }
}

class Car extends Vehicle {
    constructor(make, model, regNo, fuelType) {
        super(make, model); // Sub class calling Base class Constructor
        this.regNo = regNo;
        this.fuelType = fuelType;
    }
    getDetails() {
        /* Template literals used for displaying details of Car. */
        console.log(`${this.make},${this.model},${this.regNo},${this.fuelType}`);
    }
}

let c = new Car("Hundai", "i10", "KA-016447", "Petrol"); // Creating a Car object
c.getDetails();
```

☑ **Best Practice:** Class methods should be either made reference using **this** keyword or it can be made into a static method.

Aim:

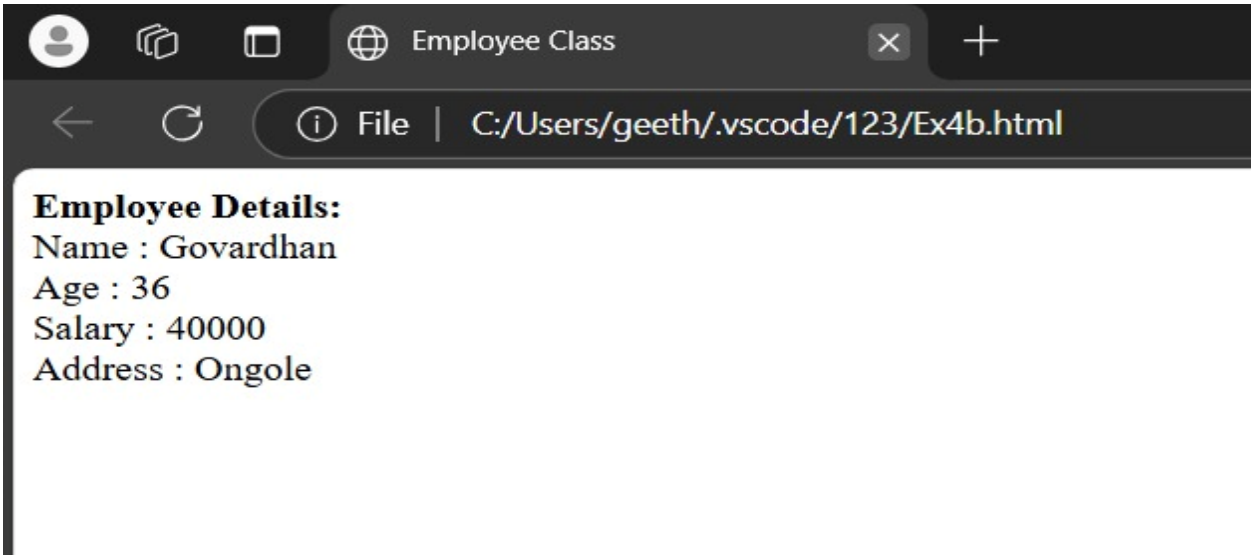
Create an Employee class extending from a base class Person. Hints:

- **Create a class Person with name and age as attributes.**
- **Add a constructor to initialize the values.**
- **Create a class Employee extending Person with additional attributes role**

Source code:

```html
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Employee Class</title>
        </head>
        <body>
            <script>
    class Person{
    constructor(name, age)
        {
        this.name = name
        this.age = age
        }

    }
        class Employee extends Person{
            constructor(name, age, salary, address)
            {
                super(name,age)
                this.salary = salary
                this.address = address
            }
                display(){
                    document.write("<b>Employee Details:</b><br>")
                    document.write("Name : "+this.name)
                    document.write("<br>Age : "+this.age)
                    document.write("<br>Salary : "+this.salary)
                    document.write("<br>Address : "+this.address)}
        }
                    var emp = new Employee('Govardhan',36,40000,'Ongole')
                    emp.display()

            </script>
        </body>
</html>
```

Output:

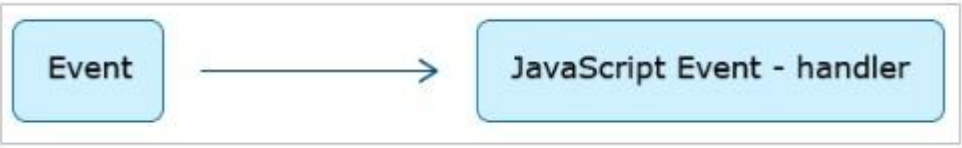


Exercise-4c

Description:

In-built Events and Handlers:

When the interaction happens, the event triggers. JavaScript event handlers enable the browser to handle them. JavaScript event handlers invoke the JavaScript code to be executed as a reaction to the event triggered.



When execution of JavaScript code is delayed or deferred till some event occurs, the execution is called deferred mode execution. This makes JavaScript an action-oriented language.

Let us understand how JavaScript executes as a reaction to these events.

Below are some of the built-in event handlers.

Event	Event-Handler	Description
click	onclick	When the user clicks on an element, the event handler onclick handles it.
keypress	onkeypress	When the user presses the keyboard's key, event handler onkeypress handles it.
keyup	onkeyup	When the user releases the keyboard's key, the event handler onkeyup handles it.
load	onload	When HTML document is loaded in the browser, event handler onload handles it
blur	onblur	When an element loses focus, the event handler onblur handles it.
change	onchange	When the selection of checked state change for input, select or text-area element changes, event handler onchange handles it.

Syntax:

<html-element eventHandler="JavaScript code">

Example:

1. Event Handler 'onclick' is associated with the HTML element 'p' to handle the 'click' on this element.

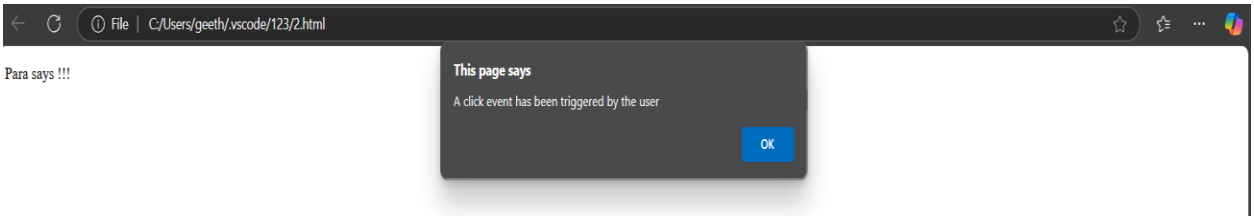
```
<!DOCTYPE html>
<html>
<head>
    </head>

    <script src="test.js"></script>


    </script>
        <body>
            <p onclick="executeMe();">Para says !!! </p>

        </body>
</html>
```

When the user clicks on element 'p', event handler 'onclick' listens to the event 'click' and executes the 'executeMe' code written in JavaScript file against the event handler.

2. When the user clicks on element 'p', event handler 'onclick' listens to the event 'click' and executes the code written against the event handler. The corresponding code is the function 'executeMe' written in the "test.js" file.

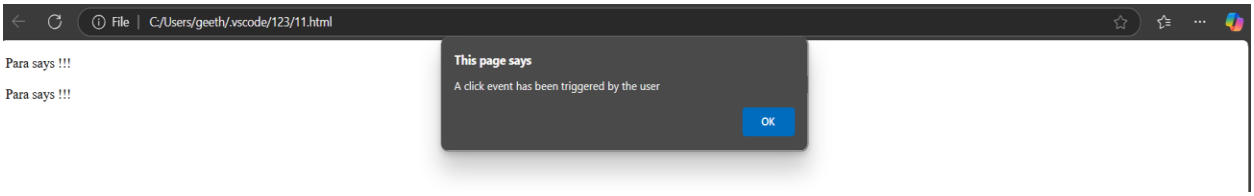
```
function executeMe() {
    alert('A click event has been triggered by the user');
}
```

3. The function 'executeMe' will now execute.



As seen in event handling code, event-handler is a piece of JavaScript code put inside the HTML Paragraph element .

```
<!DOCTYPE html>
<html>
<head>
    </head>

    <script></script>


    </script>
        <body>
            <p onclick="executeMe();">Para says !!! </p>
            <p onclick="alert('A click event has been triggered by the user');">Para says !!! </p>

        </body>
</html>
```



On the right-hand side of this code instead of invoking a function, lines of code can be directly written as shown below:

** Para says !!! **

** Para says !!! **

This is referred to as Inline Scripting where lines of JavaScript code is embedded inline to HTML elements.

However, due to tight coupling with the elements in which the code is written, this approach is not suggested. The alternate and much better approach is to use functions in JavaScript.

☑ **Best Practice:** Event listeners are the most preferred way to handle events in JavaScript. One of the major points to use event listeners is, it does allow us to add multiple event listeners on the same element when compared with the "on" properties like onmouseclick, onmouseover, etc..

Aim:

Write a JavaScript code to book movie tickets online and calculate the total price

based on the 3 conditions:

(a) If seats to be booked are not more than 2, the cost per

ticket remains Rs. 150.

(b) If seats are 6 or more, booking is not allowed.

(c) If seats to be booked are more than 2 but less than 5, based on the

number of seats booked, do the following:

o Calculate total cost by applying discounts of 5, 7, 9, 11 percent, and so on for customer 1,2,3 and 4.

o Try the code with different values for the number of seats.

Description:

Toggle() button:

A toggle button is a type of user interface (UI) element that allows the user to switch between two states, such as "on" and "off," "enabled" and "disabled," or "visible" and "hidden." It works by toggling between two opposite states with a single action, often with a click or tap.

document.getElementById:

The method document.getElementById() is used in JavaScript to select and manipulate HTML elements by their unique id attribute.

Syntax:

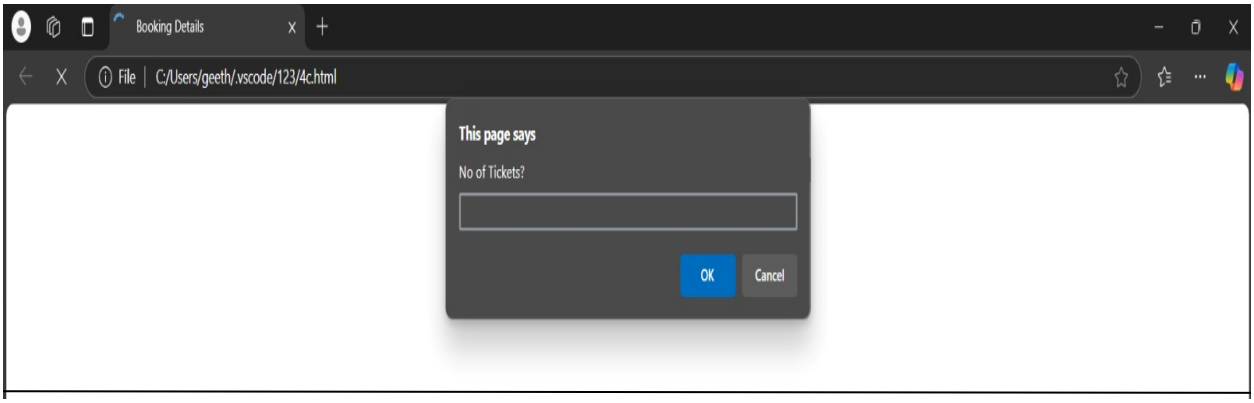
```
document.getElementById("id");
```

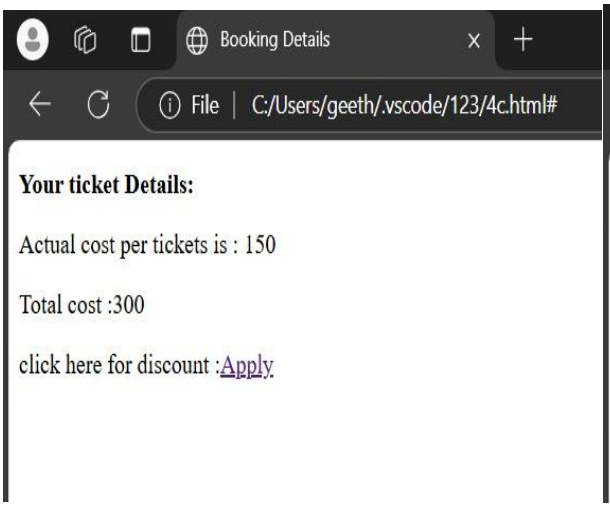
- **id: The id of the HTML element you want to select. The id must be unique within the document.**

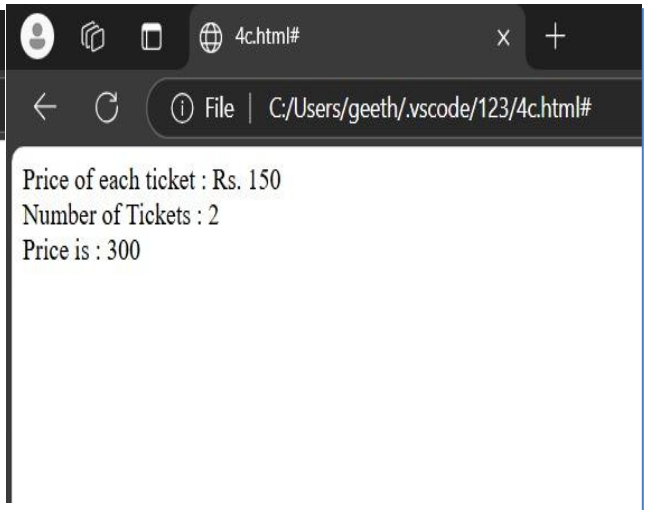
Source Code:

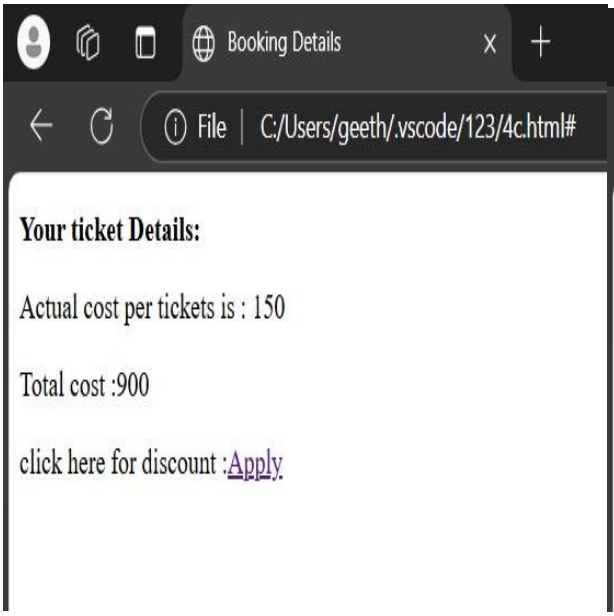
```
<!DOCTYPE html>
<html>
<head>
<title>Booking Details</title>
</head>
<body>
<p><b>Your ticket Details:</b></p>
<p>Actual cost per tickets is : 150 </p>
<p id="1"></p>
<div><p id="2">click here for discount :<a onclick="toggle()" href="#">Apply</a></p></div>
<script>
let x=prompt("No of Tickets?");
let total=0;
x=Number(x);
let total1=x*150;
document.getElementById("1").innerHTML="Total cost :"+total1;
function toggle(){
        document.write("Price of each ticket : Rs. 150")
        document.write("<br>Number of Tickets : "+x)
        var price = x*150
        if(x<=2){
            document.write("<br>Price is : "+price)
        }
        else if(x>=6){
            document.write("<br>You exceed maximum limit.")
        }
        else{
            document.write("<br>Price is : "+price)
            document.write("<br>5% Discount :"+price*0.05)
            document.write("<br>7% Discount :"+price*0.07)
            document.write("<br>9% Discount :"+price*0.09)
            document.write("<br>11% Discount :"+price*0.11)
            document.write("<br>Total Price : "+(price -(price*0.05+price*0.07+price*0.09+price*0.11)))}
        }
    </script>
</body>
</html>
```

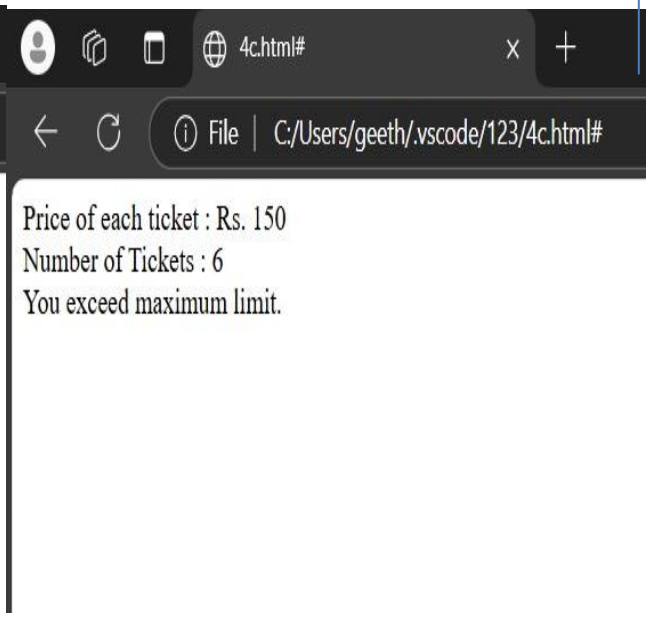
Input & Output:

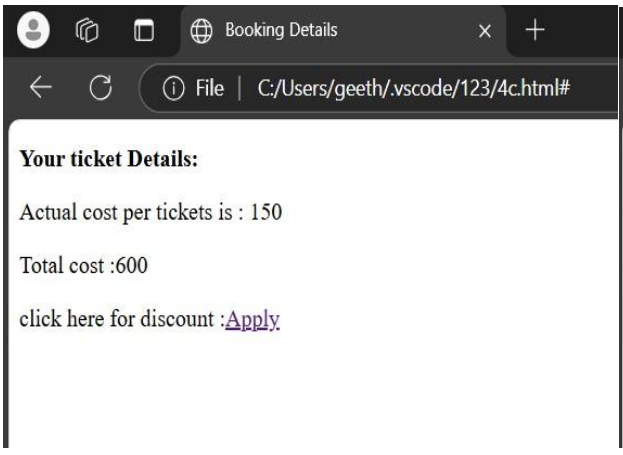


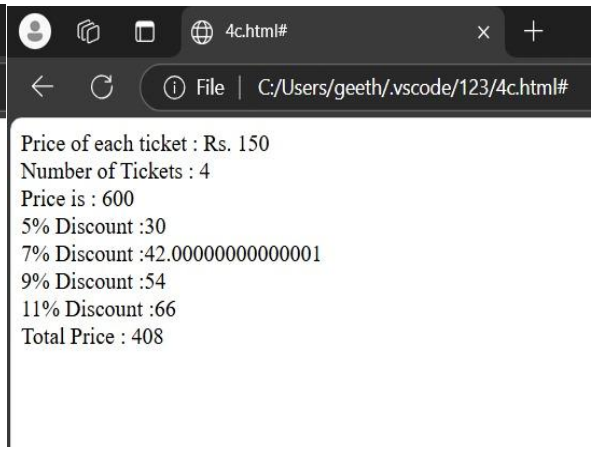

Booking Details
File | C:/Users/geeth/.vscode/123/4c.html#
Your ticket Details:
Actual cost per tickets is : 150
Total cost :300
click here for discount :Apply


4c.html#
File | C:/Users/geeth/.vscode/123/4c.html#
Price of each ticket : Rs. 150
Number of Tickets : 2
Price is : 300


Booking Details
File | C:/Users/geeth/.vscode/123/4c.html#
Your ticket Details:
Actual cost per tickets is : 150
Total cost :900
click here for discount :Apply


4c.html#
File | C:/Users/geeth/.vscode/123/4c.html#
Price of each ticket : Rs. 150
Number of Tickets : 6
You exceed maximum limit.


Booking Details
File | C:/Users/geeth/.vscode/123/4c.html#
Your ticket Details:
Actual cost per tickets is : 150
Total cost :600
click here for discount :Apply


4c.html#
File | C:/Users/geeth/.vscode/123/4c.html#
Price of each ticket : Rs. 150
Number of Tickets : 4
Price is : 600
5% Discount :30
7% Discount :42.00000000000001
9% Discount :54
11% Discount :66
Total Price : 408

Exercise-4d

Description:

Working with Objects:

An object consists of state and behavior. The State of an entity represents properties that can be modeled as key-value pairs. The Behavior of an entity represents the observable effect of an operation performed on it and is modeled using functions.

A Car is an object in the real world.

State of Car object:

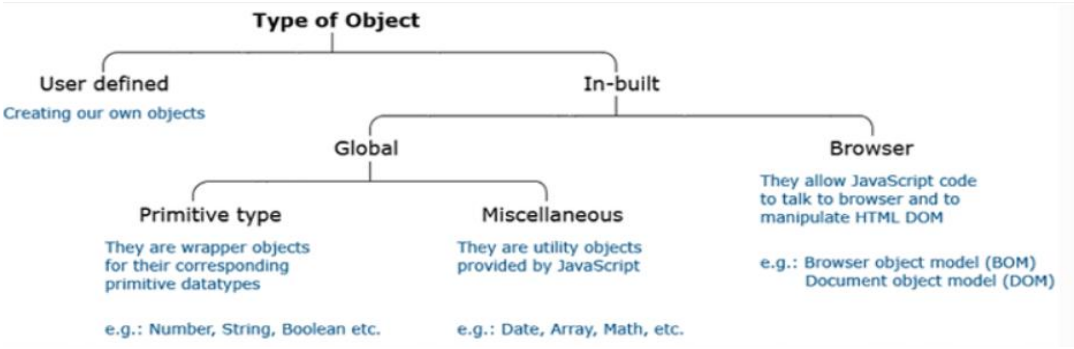
- Color=red
- Model = VXI
- Current gear = 3
- Current speed = 45 km / hr
- Number of doors = 4
- Seating Capacity = 5

The behavior of Car object:

- Accelerate
- Change gear
- Brake

Types of Objects:

JavaScript objects are categorized as follows:



In JavaScript objects, the state and behaviour is represented as a collection of properties.Each property is a [key-value] pair where the key is a string and the value can be any JavaScript primitive type value, an object, or even a function.

Ways of Creating Object:

- Object literals
- Constructor

Objects can be created using object literal notation. Object literal notation is a comma-separated list of name-value pairs wrapped inside curly braces.

```
objectName = {
    //-------------states of the object-----------
    key_1: value_1,
    key_2: value_2,
    ...
    key_n: value_n,
    //-------------behaviour of the object---------
    key_function_name_1: function (parameter) {
        //we can modify any of the property declared above
    },
    ...
    key_function_name_n: function(parameter) {
        //we can modify any of the property declared above
    }
}

```

Example:

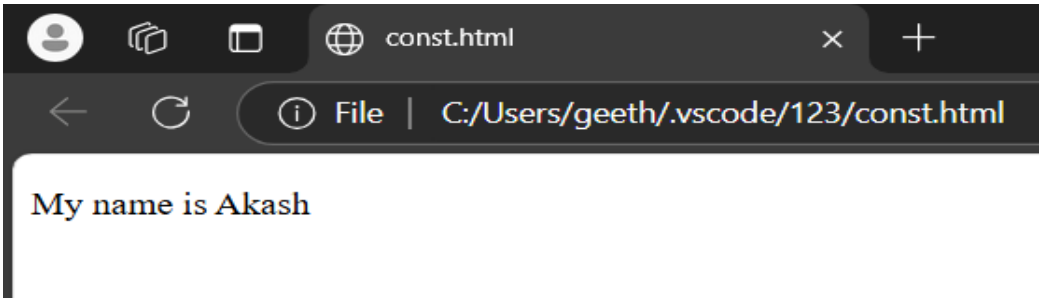
```
//-------------states of the object---------
let myCar = {
    name: "Fiat",
    model: "VXI",
    color: "red",
    numberOfGears: 5,
    currentGear: 3,
    currentSpeed: 45,
    //-------------Behaviour of the object---------
    accelerate: function (speedCounter) {
        this.currentSpeed = this.currentSpeed + speedCounter;
        return this.currentSpeed;
    },

    brake: function (speedCounter) {
        this.currentSpeed = this.currentSpeed - speedCounter;
        return this.currentSpeed;
    }
}


```

Constructor:

To construct multiple objects with the same set of properties and methods, function constructor can be used. Function constructor is like regular functions but it is invoked using a 'new' keyword.

Example:

```
<!DOCTYPE html>
<html>
<body>
    <p id="p" ></p>
    <script>
function Person(name, age)
{
this.name = name;
this.age = age;
this.sayHello = function(){
    |    return this.name + " "
    };
}
    const p1 = new Person("Akash", 30);
    const p2 = new Person("Anvesh", 25);
    document.getElementById("p").innerHTML =
    "My name is " +p1.sayHello();
    </script>
</body>
</html>
```



Combining and cloning Objects:

The spread operator is used to combine two or more objects. The newly created object will hold all the properties of the merged objects.

Syntax:

```
let object1Name = {
    //properties
};
let object2Name = {
    //properties
};
let combinedObjectName = {
    ...object1Name,
    ...object2Name
};
//the combined object will have all the properties of object1 and object2
```

It is possible to get a copy of an existing object with the help of the spread operator.

Syntax:

1. let copyToBeMade = { ...originalObject };

Example:

const sourceObject = { a: 1, b: 2, c: 3 };

let Clone_object = {};

Clone_object = {...sourceObject};

console.log(Clone_object);

Destructuring Objects:

Destructuring gives a syntax that makes it easy to create objects based on variables. It also helps to extract data from an object. Destructuring works even with the rest and spread operators.

In the below example an object is destructured into individual variables:

```
let myObject = { name: 'Arnold', age: 65, country: 'USA' };
let { name, age:currentAge } = myObject; //alias can be used with :
console.log(name);
console.log(currentAge);

//OUTPUT: Arnold 65
```

Browser Object Model:

As you know that, JavaScript is capable of dynamically manipulating the content and style of HTML elements of the web page currently rendered on the browser. The content given for para during HTML creation or the style given for heading during HTML creation can be changed even after the page has arrived on the browser.

For programming purposes, the BOM model virtually splits the browser into different parts and refers to each part as a different type of built-in object. BOM is a hierarchy of multiple objects. 'window' object is the root object and consists of other objects in a hierarchy, such as, 'history' object, 'navigator' object, 'location' object, and 'document' object.

Document Object Model:

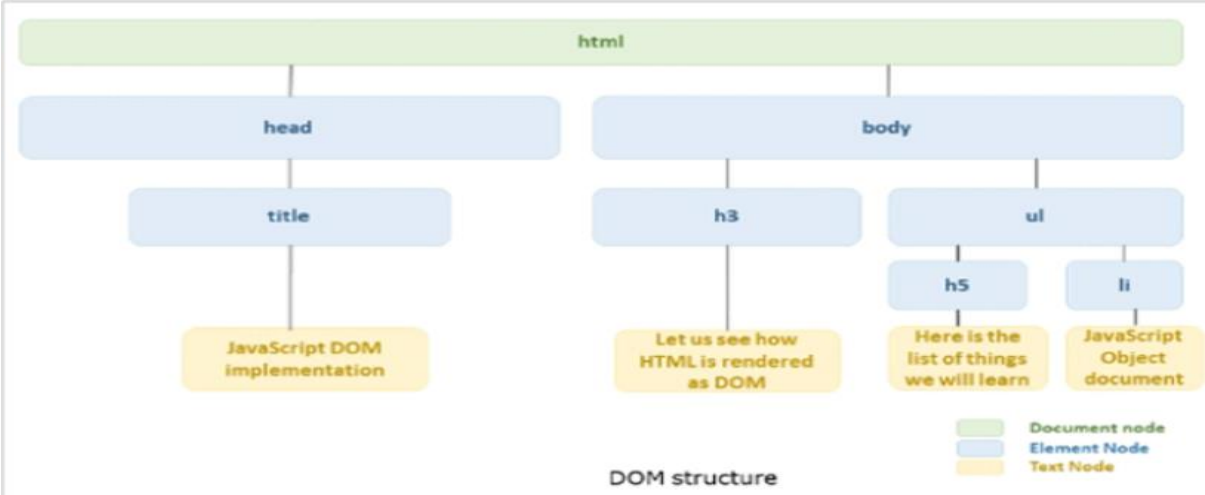
The HTML web page that gets loaded on the browser is represented using the 'document' object of the BOM model.

This object considers the web page as a tree which is referred to as Document Object Model(DOM). Each node of this tree represents HTML elements in the page as 'element' object and its attributes as properties of the 'element' object.

Sample HTML Code:

```
<html>
<head>
    <title>JavaScript DOM Implementation</title>
</head>
<body>
    <h3>Let us see how HTML is rendered as DOM</h3>
    <ul>
        <h5>Here is the list of things we will learn</h5>
        <li>JavaScript Object Document</li>
    </ul>
</body>
</html>
```

DOM Structure:



getElementById(x)

Finds element with id 'x' and returns an object of type element

Example:

```
<p id="p1"> Paragraph 1</p>
<p> Paragraph 2</p>
<script>
    //Selects paragraph having id 'p1'
    document.getElementById('p1');
</script>
```

getElementsByTagName(x)

Find element(s) whose tag name is 'x' and return NodeList, which is a list of element objects.

Example:

```
<p id="p1">Paragraph 1</p>
<p>Paragraph 2</p>
<script>
    document.getElementsByTagName('p');
</script>
//OUTPUT:
//Paragraph 1
//Paragraph 2
```

Some of the other properties of the 'document' object to access the HTML element are:

- the **body** returns body element. **Usage**: document.body;
- the **forms** return all form elements. **Usage**: document.forms;
- the **head** returns the head element. **Usage**: document.head;
- the **images** return all image elements. **Usage**: document.images;

To manipulate the content of HTML page, the following properties of 'element' object given by DOM API can be used:

innerHTML

It gives access to the content within HTML elements like div, p, h1, etc. You can set/get a text.

Example:

```
<div id="div1">
    <h1 id="heading1">Welcome to JavaScript Tutorial</h1>
          <p id="para1" style="color: blue;">Let us learn DOM API</p>
</div>
<script>
    //retieves current content
    document.getElementById("heading1").innerHTML;
    //sets new content
    document.getElementById.innerHTML = "Heading generated dynamically"
</script>
```

attribute : It is used to set new values to given attributes.

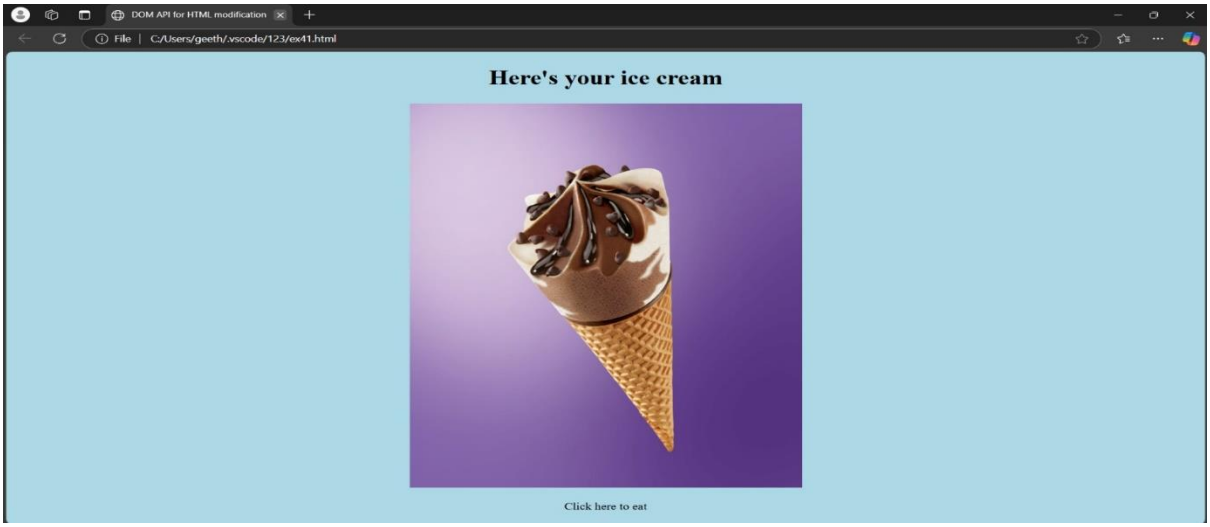
Aim:

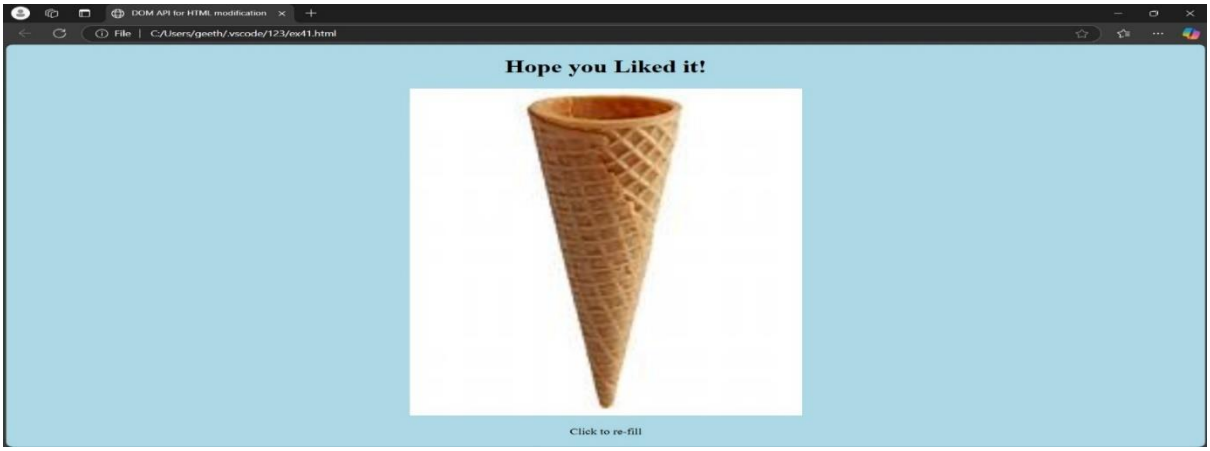
If a user clicks on the given link, they should see an empty cone, a different heading, and a different message and a different background color. If user clicks again, they should see a re-filled cone, a different heading, a different message, and a different background color.

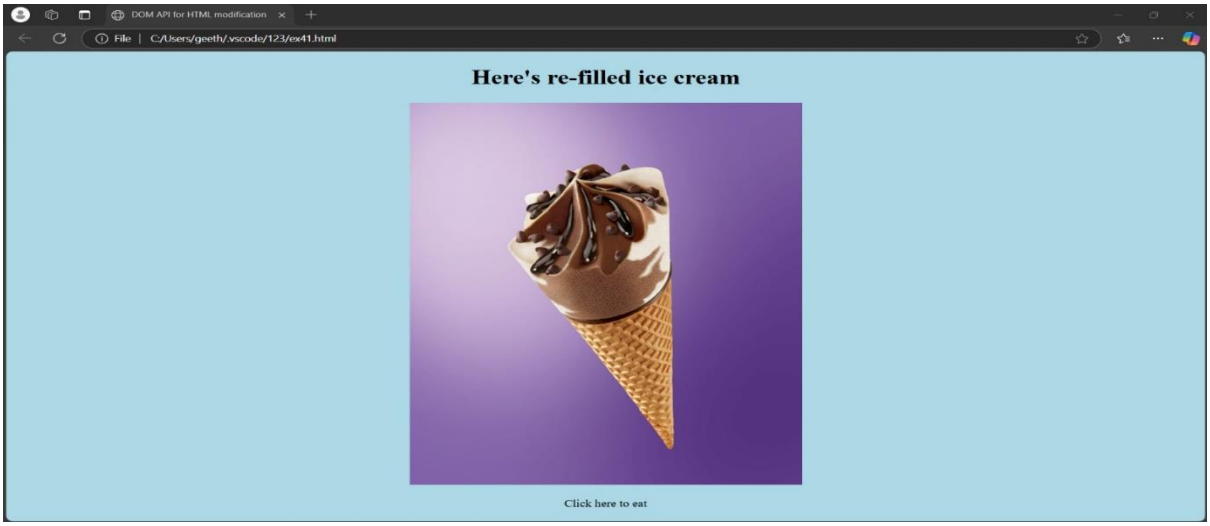
Source Code:

```
<!DOCTYPE html>
<html>
<head>
<title>DOM API for HTML modification</title>
</head>
<body style="background-color: lightblue;text-align:center">
<h1 id="hdr1">Here's your ice cream</h1>
<img id="img1" src="C:/Users/geeth/OneDrive/Desktop/ice1.jpg" alt="Ice cream is not ready" width="500" height="600">
<p id="p1" onclick="eat()">Click here to eat</p>
<script>
function eat() {
  const header = document.getElementById("hdr1");
  const img = document.getElementById("img1");
  const p = document.getElementById("p1");
  if (header.innerHTML == "Here's your ice cream") {
    header.innerHTML = "Hope you Liked it!";
    img.src = "C:/Users/geeth/OneDrive/Desktop/ice2.jpg";
    img.width = 500;
    img.height = 600;
    p.innerHTML = "Click to re-fill";
  } else if (header.innerHTML == "Hope you Liked it!") {
    header.innerHTML = "Here's re-filled ice cream";
    img.src = "C:/Users/geeth/OneDrive/Desktop/ice1.jpg";
    img.width = 500;
    img.height = 600;
    p.innerHTML = "Click here to eat";
  } else {
    header.innerHTML = "Hope you Liked it!";
    img.src = "C:/Users/geeth/OneDrive/Desktop/ice2.jpg";
    img.width = 500;
    img.height = 600;
    p.innerHTML = "Click to re-fill";
  }
}
</script>
</body>
</html>
```

Output:


DOM API for HTML modification
File | C:/Users/geeth/.vscode/123/ex41.html
Here's your ice cream
Click here to eat


DOM API for HTML modification
File | C:/Users/geeth/.vscode/123/ex41.html
Hope you Liked it!
Click to re-fill


DOM API for HTML modification
File | C:/Users/geeth/.vscode/123/ex41.html
Here's re-filled ice cream
Click here to eat

Exercise 5: JavaScript

Arrays

An array in JavaScript is a data structure used to store multiple values in a single variable. It can hold various data types and allows for dynamic resizing. Elements are accessed by their index, starting from 0.

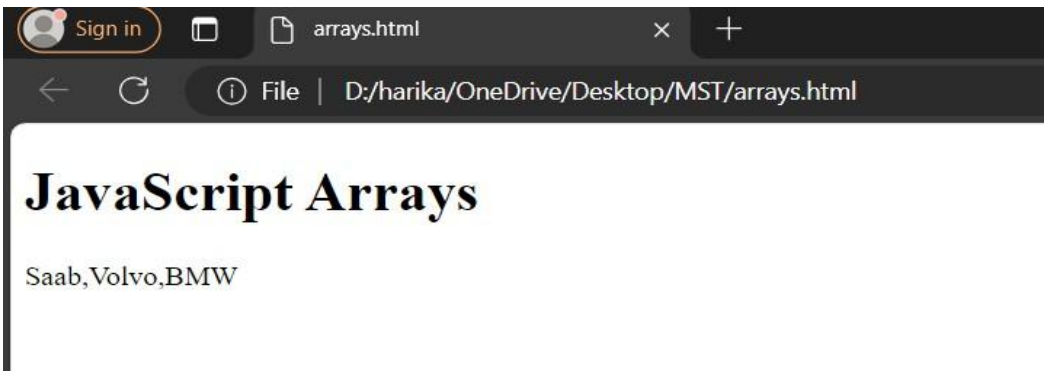
Example:

Welcome | arrays.html

arrays.html > html

```
<!DOCTYPE html>
<html>
<body>
<h1>JavaScript Arrays</h1>

<p id="demo"></p>

<script>
const cars = ["Saab", "Volvo", "BMW"];
document.getElementById("demo").innerHTML = cars;
</script>

</body>
</html>
```

Output:



Ex 5a: Creating Arrays, Destructuring Arrays, Accessing Arrays, Array Methods

Creating Arrays:

There are 3 ways to construct array in JavaScript

1. By array literal
2. By creating instance of Array directly (using new keyword)
3. By using an Array constructor (using new keyword)

JavaScript Array Literal:

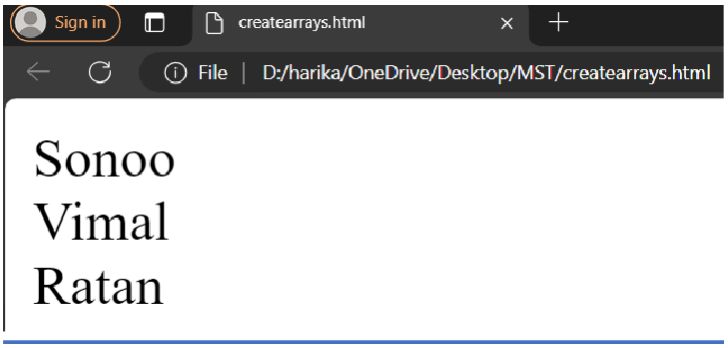
The syntax of creating array using array literal is given below:

var arrayname=[value1,value2. ... valueN];

As you can see, values are contained inside [ ] and separated by , (comma).

Example:

Output:



2) JavaScript Array directly (new keyword)

The syntax of creating array directly is given below:

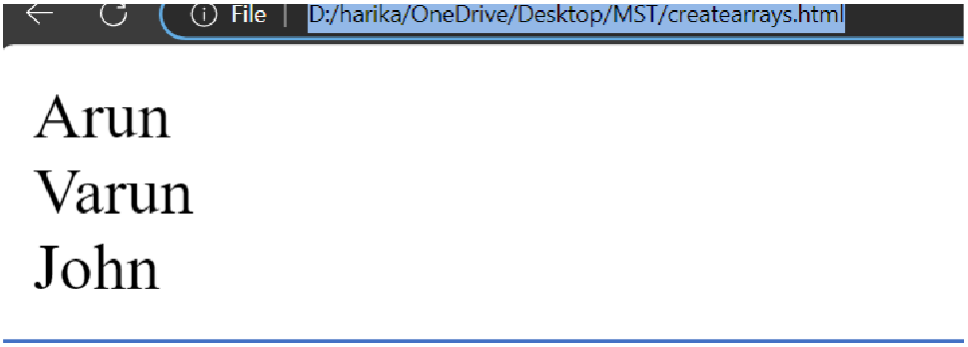
var arrayname=new Array();

Here, new keyword is used to create instance of array.

Example:

```
<script>
    var i;
    var emp = new Array();
    emp[0] = "Arun";
    emp[1] = "Varun";
    emp[2] = "John";

    for (i=0;i<emp.length;i++){
    document.write(emp[i] + "<br>");
    }
    </script>
```

Output:



3) JavaScript array constructor (new keyword)

Here, you need to create instance of array by passing arguments in constructor so that we don't have to provide value explicitly.

Example:

```
<script>
    var emp=new Array("Jai","Vijay","Smith");
    for (i=0;i<emp.length;i++){
    document.write(emp[i] + "<br>");
    }
    </script>
```

Output:

Jai
Vijay
Smith

Destructuring Arrays:

Destructuring Assignment is a JavaScript expression that allows to unpack of values from arrays, or properties from objects, into distinct variables data can be extracted from *arrays, objects, and nested objects,* and assigned to variables.

Array Destructuring

Array members can be unpacked into different variables. The following are different examples.

```
const a = [10, 20, 30, 40]
console.log("Example 1");
const [x, y, z, w] = a;
console.log(x);
console.log(y);
console.log(z);
console.log(w);

const [p, q, , r] = a;
console.log("Example 2");
console.log(p);
console.log(q);
console.log(r);

const [s, t] = a;
console.log("Example 3");
console.log(s);
console.log(t);
```

Output:

```
ka\OneDrive\Desktop\MST> node destructarrays.js
Example 1
10
20
30
40
Example 2
10
20
20
20
40
Example 3
40
Example 3
Example 3
10
20
PS D:\harika\OneDrive\Desktop\MST>
```

Rest Operator:

In order to assign some array elements to variable and rest of the array elements to only a single variable can be achieved by using rest operator (…) as in below implementation. But one limitation of rest operator is that it works correctly only with the last elements implying a subarray cannot be obtained leaving the last element in the array.

// Example 1

```
const numbers = [1, 2, 3, 4, 5];

const [first, second, ...rest] = numbers;

console.log("Example 1:")
```

console.log(first);

console.log(second);

console.log(rest);

// Example 2

let [fst, , ...last] = ["a", "b", "c", "d"];

console.log("Example 2:")

console.log(fst);

console.log(last);

```
// Example 1

const numbers = [1, 2, 3, 4, 5];
const [first, second, ...rest] = numbers;
console.log("Example 1:")
console.log(first);
console.log(second);
console.log(rest);

// Example 2

let [fst, , ...last] = ["a", "b", "c", "d"];
console.log("Example 2:")
console.log(fst);
console.log(last);
```

Output:

```
PS D:\harika\OneDrive\Desktop\MST> node destructarrays.js
Example 1:
1
2
[ 3, 4, 5 ]
Example 2:
a
[ 'c', 'd' ]
PS D:\harika\OneDrive\Desktop\MST>
```

Accessing Arrays:

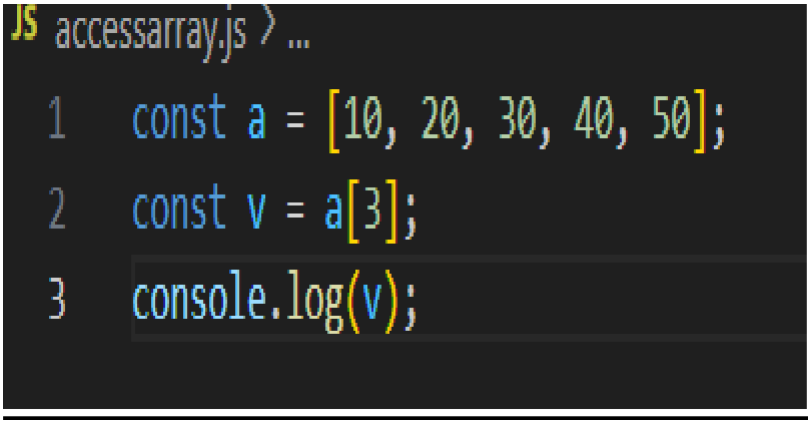
These are the following ways to Access Elements in an Array:

- **Using Square Bracket Notation**
- **Using forEach Loop**
- **Using map() Method**
- **Using find() Method**
- **Using Destructuring Assignment**
- **Using filter() Method**

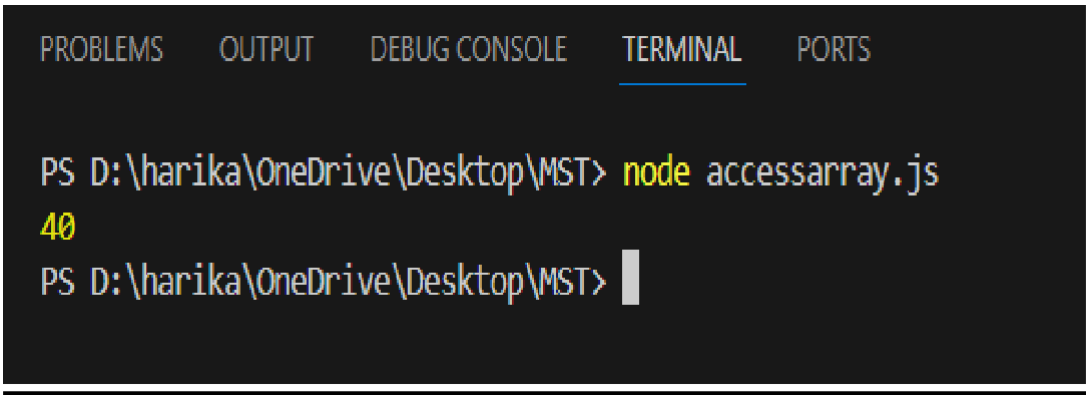
1.Using Square Bracket Notation:

We can access elements in an array by using their index, where the index starts from 0 for the first element. We can access using the bracket notation.

Code Snippet:



Output:



2. Using forEach Loop

In this approach, we will use a loop for accessing the element. We can use for, forEach, or for...of methods for looping. The forEach() method allows you to iterate over all elements in the array and perform an operation on each element.

Code Snippet:

```
5   const a = [100, 200, 300, 400, 500];
6   a.forEach((e, i) => {
7       console.log(e);
8   });
```

Output:

```
PS D:\harika\OneDrive\Desktop\MST> node accessarray.js
100
200
300
400
500
PS D:\harika\OneDrive\Desktop\MST>
```

4.Using map() Method

The Javascript map() method in JavaScript creates an array by calling a specific function on each element present in the parent array.

Code Snippet:

```
10    const a = [10, 20, 30, 40, 50];
11    const r = a.map((e, i) => {
12        console.log(e);
13    });
```

Output:

```
PS D:\harika\OneDrive\Desktop\MST> node accessarray.js
10
20
30
40
50
PS D:\harika\OneDrive\Desktop\MST>
```

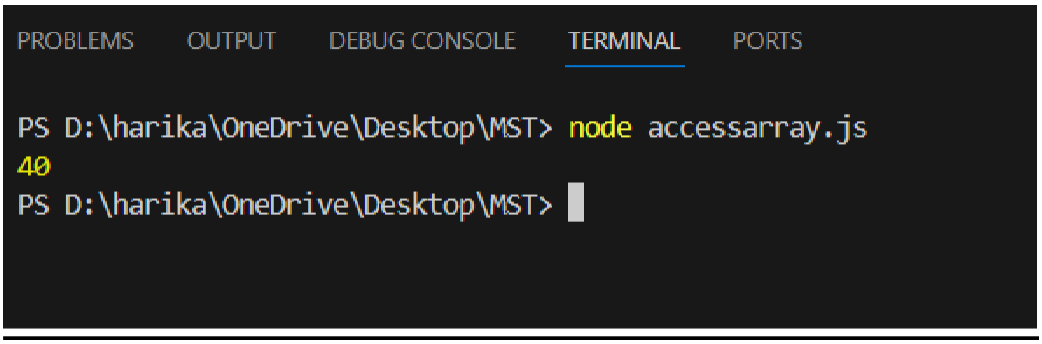
5.Using find() Method:

The find() method returns the first element in the array that satisfies a provided testing function.

Code Snippet:

```
const a = [10, 20, 30, 40, 50];
const r = a.find((e) => e > 30);
console.log(r);
```

Output:



5. Using filter() Method

The filter() method in JavaScript creates a new array containing elements that pass a specified condition. It iterates through each element of the array, executing the condition for each element and including elements that return true in the filtered array.

Code Snippet:

```
const a = [1, 2, 3, 4, 5];
const res = a.filter(e => e > 2);
console.log(res);
```

Output:

```
PS D:\harika\OneDrive\Desktop\MST> node accessarray.js
[ 3, 4, 5 ]
PS D:\harika\OneDrive\Desktop\MST>
```

Array Methods:

JavaScript array methods are built-in functions that allow efficient manipulation and traversal of arrays. They provide essential functionalities like adding, removing, and transforming elements, as well as searching, sorting, and iterating through array elements, enhancing code readability and productivity.

- **JavaScript Array length**
- **JavaScript Array toString() Method**
- **JavaScript Array join() Method**
- **JavaScript Array delete Operator**
- **JavaScript Array concat() Method**
- **JavaScript Array flat() Method**
- **Javascript Array.push() Method**
- **Javascript Array.unshift() Method**
- **JavaScript Array.pop() Method**
- **JavaScript Array.shift() Method**
- **JavaScript Array.splice() Method**
- **JavaScript Array.slice() Method**
- **JavaScript Array some() Method**
- **JavaScript Array reduce() Method**
- **JavaScript Array map() Method**

- **Javascript Array reverse() method**
- **Javascript Array values() method**

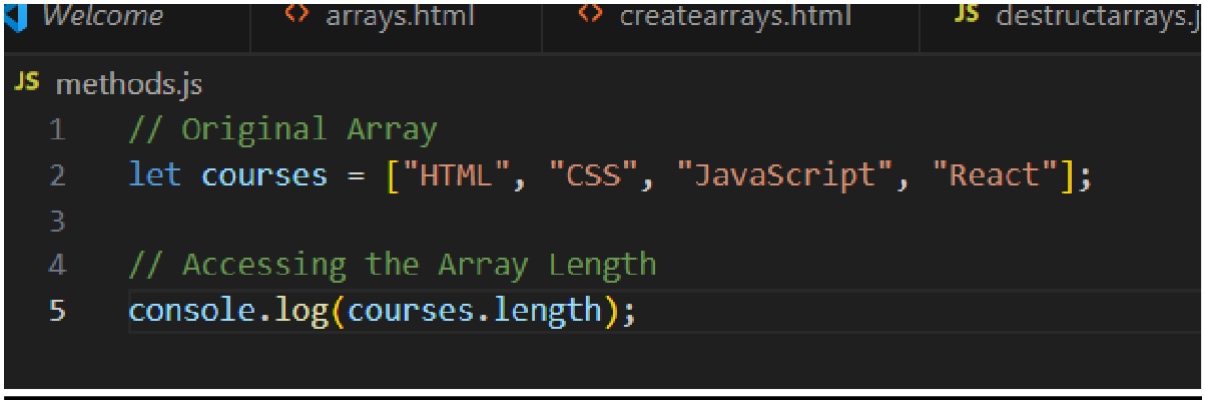
Javascript Array length:

The length property returns the length of the given array.

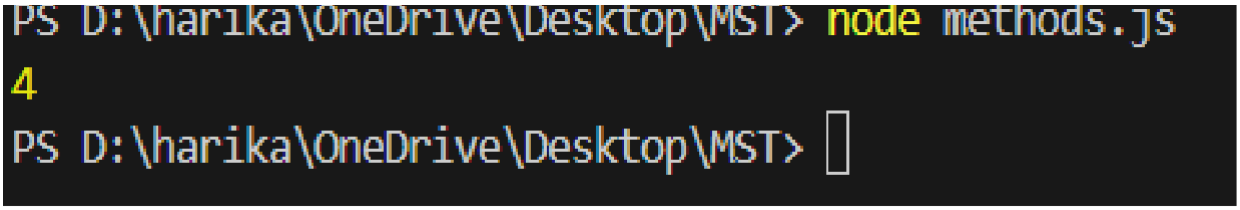
Syntax:

Array.length

Example:



Output:



JavaScript Array toString() Method

The toString() method converts the given value into the string.

Syntax:

arr.toString()

Code Snippet:

```
8   // Original Array
9   let courses = ["HTML", "CSS", "JavaScript", "React"];
10
11  // Converting array ot String
12  let str = courses.toString();
13
14  console.log(str);
```

Output:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS D:\harika\OneDrive\Desktop\MST> node methods.js
HTML,CSS,JavaScript,React
PS D:\harika\OneDrive\Desktop\MST>
```

JavaScript Array join() Method:

This join() method creates and returns a new string by concatenating all elements of an array. It uses a specified separator between each element in the resulting string.

Syntax

array.join(separator)

Code Snippet:

```
// Original Array
let courses = ["HTML", "CSS", "JavaScript", "React"];

// Joining the array elements
console.log(courses.join('|'));
```

Output:

```
PS D:\harika\OneDrive\Desktop\MST> node methods.js
HTML|CSS|JavaScript|React
PS D:\harika\OneDrive\Desktop\MST>
```

JavaScript Array delete Operator:

The delete operator is used to delete the given value which can be an object, array, or anything.

Syntax:

delete object
// or
delete object.property
// or
delete object['property']

Code Snippet:

```
let emp = {
    firstName: "Raj",
    lastName: "Kumar",
    salary: 40000
}

console.log(delete emp.salary);
console.log(emp);
```

Output:

```
PS D:\harika\OneDrive\Desktop\MST> node methods.js
true
{ firstName: 'Raj', lastName: 'Kumar' }
PS D:\harika\OneDrive\Desktop\MST>
```

JavaScript Array concat() Method:

The concat() method is used to concatenate two or more arrays and it gives the merged array.

Syntax:

let newArray = arr.concat() // or

let newArray = arr1.concat(arr2) // or

let newArray = arr1.concat(arr2, arr3, ...) // or

let newArray = arr1.concat(value0, value1)

Code Snippet:

```
32   // Declare three arrays
33   let arr1 = [11, 12, 13];
34   let arr2 = [14, 15, 16];
35   let arr3 = [17, 18, 19];
36
37   let newArr = arr1.concat(arr2, arr3);
38   console.log(newArr);
```

Output:

```
PS D:\harika\OneDrive\Desktop\MST>  node methods.js
[
  11, 12, 13, 14, 15,
  16, 17, 18, 19
]
PS D:\harika\OneDrive\Desktop\MST>
```

Javascript Array.push() Method:

The push() method is used to add an element at the end of an Array. As arrays in JavaScript are mutable objects, we can easily add or remove elements from the Array. And it dynamically changes as we modify the elements from the array.

Syntax:

Array.push(item1, item2 ...)

Code Snippet:

```
// Declaring and initializing arrays
let numArr = [10, 20, 30, 40, 50];

// Adding elements at the end of an array
numArr.push(60);
numArr.push(70, 80, 90);
console.log(numArr);


let strArr = ["piyush", "gourav", "smruti", "ritu"];
strArr.push("sumit", "amit");

console.log(strArr);
```

Output:

```
PS D:\harika\OneDrive\Desktop\MST> node methods.js
[
  10, 20, 30, 40, 50,
  60, 70, 80, 90
]
[ 'piyush', 'gourav', 'smruti', 'ritu', 'sumit', 'amit' ]
PS D:\harika\OneDrive\Desktop\MST>
```

JavaScript Array.pop() Method:

The pop() method is used to remove elements from the end of an array.

Syntax:

Array.pop()

Code Snippet:

```
88   // Declare and initialize array
89   let numArr = [20, 30, 40, 50];
90
91   // Removing elements from end
92   // of an array
93   numArr.pop();
94
95   console.log(numArr);
96
97
98   // Declare and initialize array
99   let strArr = ["amit", "sumit", "anil"];
100
101  // Removing elements from end
102  // of an array
103  strArr.pop();
104
105  console.log(strArr);
```

Output:

```
PS D:\harika\OneDrive\Desktop\MST> node methods.js
[ 20, 30, 40 ]
[ 'amit', 'sumit' ]
PS D:\harika\OneDrive\Desktop\MST>
```

JavaScript Array.slice() Method:

The slice() method returns a new array containing a portion of the original array, based on the start and end index provided as arguments

Syntax:

Array.slice (startIndex , endIndex);

Code Snippet:

```
const case1 = arr.slice(0, 3);

const case2 = arr.slice(-3);

const case3 = arr.slice(3, 7);

const case4 = arr.slice(5, 2);

const case5 = arr.slice(-4, 9);

const case6 = arr.slice(3, -2);

const case7 = arr.slice(5);

const case8 = arr.slice(15, 20);

const case9 = arr.slice(-15, -10);
```

```
// Original Array
const arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
// Case 1: Extract the first 3 elements of the array
const case1 = arr.slice(0, 3);
console.log("First 3 Array Elements: ", case1);
// Case 2: Extract the last 3 array elements
const case2 = arr.slice(-3);
console.log("Last 3 Array Elements: ", case2);
// Case 3: Extract elements from middle of array
const case3 = arr.slice(3, 7);
console.log("Case 3: Extract elements from middle: ", case3);
// Case 4: Start index is greater than end index
const case4 = arr.slice(5, 2);
console.log("Case 4: Start index is greater than end index: ", case4);
//Case 5: Negative start index
const case5 = arr.slice(-4, 9);
console.log("Case 5: Negative start index: ", case5);
// Case 6: Negative end index
const case6 = arr.slice(3, -2);
console.log("Case 6: Negative end index: ", case6);
// Case 7: Only start index is provided
const case7 = arr.slice(5);
console.log("Case 7: Only start index is provided: ", case7);
// Case 8: Start index and end index are out of range
const case8 = arr.slice(15, 20);
console.log("Case 8: Start and end index out of range: ", case8);
// Case 9: Start and end index are negative
// and out of range
const case9 = arr.slice(-15, -10);
console.log("Case 9: Start and end index are negative"
    + " and out of range: ", case9);
```

Output:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS D:\harika\OneDrive\Desktop\MST> node methods.js
First 3 Array Elements:  [ 1, 2, 3 ]
Last 3 Array Elements:  [ 8, 9, 10 ]
Case 3: Extract elements from middle:  [ 4, 5, 6, 7 ]
Case 4: Start index is greater than end index:  []
Case 5: Negative start index:  [ 7, 8, 9 ]
Case 6: Negative end index:  [ 4, 5, 6, 7, 8 ]
Case 7: Only start index is provided:  [ 6, 7, 8, 9, 10 ]
Case 8: Start and end index out of range:  []
Case 9: Start and end index are negative and out of range:  []
PS D:\harika\OneDrive\Desktop\MST>
```

Problem Statement:

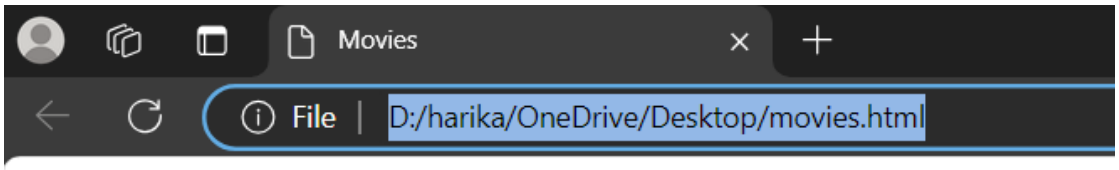
Create an array of objects having movie details. The object should include the movie name, starring, language, and ratings. Render the details of movies on the page using the array.

Code Snippet:


Welcome
arrays.html
createarrays.html
destructarrays.js
ligin.js
D: > harika > OneDrive > Desktop > movies.html > html > body > script > movies
<!DOCTYPE html>
<html>
<head>
<title>Movies</title>
</head>
<body>
<h1>Movies</h1>
<script>
const movies = [
{
id: 1,
name: "Devara",
starring: "JR NTR",
language: "TELUGU",
rating: 9.0
},
{
id: 2,
name: "Leo",
starring: "T VAJAY",
language: "TAMIL",
rating: 8.2
},
{
id: 3,
name: "PATAN",
starring: "S KHAN",
language: "HINDI",
rating: 7.6
},
{
id: 4,
name: "KGF",
starring: "YASH",
language: "KANNADA",
rating: 8.9
}

```
38            ];
39
40            for(let i = 0; i < movies.length; i++) {
41                document.write(`ID: ${movies[i].id}, Name: ${movies[i].name}, Starring: ${movies[i].starring}, Language: ${movies[i].language},
42                Rating: ${movies[i].rating}`);
43                document.write("<br>");
44            }
45        </script>
46    </body>
47 </html>
```

Output:



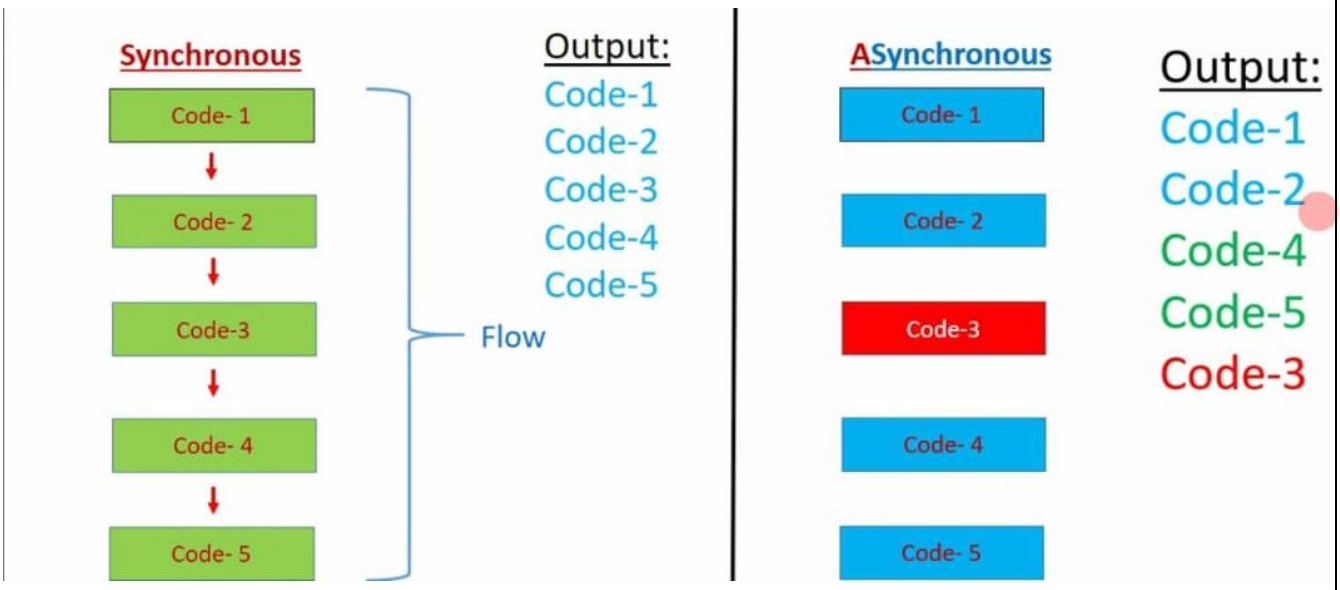
Movies

ID: 1, Name: Devara, Starring: JR NTR, Language: TELUGU, Rating: 9
ID: 2, Name: Leo, Starring: T VAJAY, Language: TAMIL, Rating: 8.2
ID: 3, Name: PATAN, Starring: S KHAN, Language: HINDI, Rating: 7.6
ID: 4, Name: KGF, Starring: YASH, Language: KANNADA, Rating: 8.9

5.b	**Course Name:** Javascript
	Module Name: Introduction to Asynchronous Programming, Callbacks, Promises, Async and Await, Executing Network Requests using Fetch API
	Simulate a periodic stock price change and display on the console. Hints: (i) Create a method which returns a random number - use Math.random, floor and other methods to return a rounded value. (ii) Invoke the method for every three seconds and stop when

Asynchronous programming:

Asynchronous programming in JavaScript allows the execution of tasks without blocking the main program flow, enabling efficient handling of operations like network requests, file I/O, and timers. This approach ensures that applications remain responsive, even when performing time- consuming tasks.



Callbacks: A callback is a function passed as an argument to another function, which is then invoked after the completion of a specific task. This pattern allows for the continuation of code execution once an asynchronous operation finishes.

Example:

JS 1.js > ...

```
function fetchData(callback) {
    setTimeout(() => {
      const data = { name: 'John', age: 30 };
      callback(data);
    }, 1000);
  }

  function handleData(data) {
    console.log(`Name: ${data.name}, Age: ${data.age}`);
  }

  fetchData(handleData);
```

Output:

```
[Running] node "c:\Users\Abhiram\OneDrive - Abhiram\Desktop\3-2\Mst lab\1.js"
Name: John, Age: 30

[Done] exited with code=0 in 1.129 seconds
```

Promises: A Promise represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

Example:

```
const fetchData = ()=>{
  return new Promise((resolve,reject)=>{
    setTimeout(()=>{
      const data =
      {message: "Data fetched success"};
      resolve(data);

      reject("Error fetching data")
    }, 2000);

  })
```

Async and Await: async and await provide a way to work with asynchronous code that looks synchronous, improving readability.

Example:

JS 5.js > main

```
// Define an asynchronous function
async function greet(name, delay) {
    console.log(`Starting to greet ${name}...`);
    await new Promise(resolve => setTimeout(resolve, delay)); // Simulates a delay
    console.log(`Hello, ${name}!`);
}

// Define the main function to run the tasks
async function main() {
    // Create multiple asynchronous tasks
    const task1 = greet("Alice", 2000); // Delay of 2 seconds
    const task2 = greet("Bob", 1000);   // Delay of 1 second
    const task3 = greet("Charlie", 3000); // Delay of 3 seconds

    // Wait for all tasks to complete
    await task1;
    await task2;
    await task3;
}

// Call the main function
main();
```

Output:

```
[Running] node "c:\Users\Abhiram\OneDrive - Abhiram\Desktop\3-2\Mst lab\5.js"
Starting to greet Alice...
Starting to greet Bob...
Starting to greet Charlie...
Hello, Bob!
Hello, Alice!
Hello, Charlie!

[Done] exited with code=0 in 3.116 seconds
```

Fetch API:

The Fetch API provides a modern way to make network requests. It returns Promises, making it easy to work with async and await.

Example:

```
JS 4.js > ...
async function fetchUser() {
    try {
      const response = await fetch('https://jsonplaceholder.typicode.com/users/1');
      if (!response.ok) {
        throw new Error('Network response was not ok');
      }
      const user = await response.json();
      console.log(`Name: ${user.name}, Email: ${user.email}`);
    } catch (error) {
      console.error('Fetch error:', error);
    }
  }

  fetchUser();
```

Output:

```
[Running] node "c:\Users\Abhiram\OneDrive - Abhiram\Desktop\3-2\Mst lab\4.js"
Name: Leanne Graham, Email: Sincere@april.biz

[Done] exited with code=0 in 0.667 seconds
```

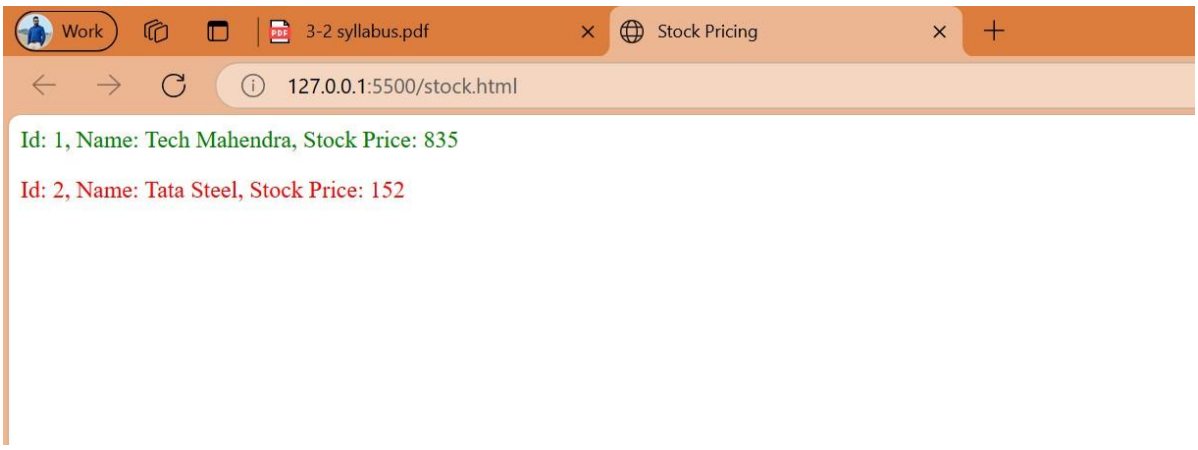
Problem Statement :

Simulate a periodic stock price change and display on the console. Hints: (i) Create a method which returns a random number - use Math.random, floor and other methods to return a rounded value. (ii) Invoke the method for every three seconds and stop when

Problem Statement Code:

```
stock.html > html > body > script > stock
<!DOCTYPE html>
<html>
    <head>
        <title>Stock Pricing</title>
    </head>
    <body>
        <div id="stock-container"></div>

        <script>
            const stock = [
                {
                    Id: 1,
                    stock_name: "Tech Mahendra",
                    stock_price: null,
                    status: 'unknown',
                    previousPrice: null
                },
                {
                    Id: 2,
                    stock_name: "Tata Steel",
                    stock_price: null,
                    status: 'unknown',
                    previousPrice: null
                }
            ];

            const stockContainer = document.getElementById('stock-container');

            function updateStockPrices() {
                stock.forEach((item) => {
                    const newPrice = Math.floor(Math.random() * 1000);
                    item.previousPrice = item.stock_price;
                    item.stock_price = newPrice;
                    item.status = item.previousPrice === null ? 'unknown' : (newPrice > item.previousPrice ? 'up' : 'down');
                });
                renderStockPrices();
            }

            function renderStockPrices() {
                let html = '';
                stock.forEach((item) => {
                    const color = item.status === 'up' ? 'green' : (item.status === 'down' ? 'red' : 'black');
                    html += `<p style="color: ${color}">Id: ${item.Id}, Name: ${item.stock_name}, Stock Price: ${item.stock_price}</p>`;
                });
                stockContainer.innerHTML = html;
            }

            setInterval(updateStockPrices, 3000);

            updateStockPrices();
        </script>
    </body>
</html>
```

Output:



Ex 5c: Creating Modules, Consuming Modules:

JavaScript modules help

- **Organize code**
- **Reuse functionality**
- **Avoid naming conflicts**

Understanding Modules in JavaScript

- **Modules:** Files containing reusable code.
- **Export:** Makes functions, objects, or variables available to other files.
- **Import:** Accesses exported code from other files.

Module Types:

- **Named Export/Import:** Export multiple items from a file.
- **Default Export/Import:** Export one main item per file.

Creating a JavaScript Module

Step 1: Create a Module File
Create a separate `.js` file for your module (math.js)

```
//Named Exports
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;

// Default Export
export default function multiply(a, b) {
 return a * b;
```

```
}
```

Importing a Module

Step 2: Import into Another File

Import Named Exports: When importing specific functions or variables

App.js

```
import { add, subtract } from './math.js';
console.log(add(5, 3)); // Output: 8
console.log(subtract(5, 3)); // Output: 2
```

Import Default Export: When importing the default export

App.js

```
 import multiply from './math.js';
console.log(multiply(5, 3)); // Output: 15
```

Using Modules in HTML:

Make sure to set the script type to `module` in your HTML file.

Index.html

```
<!DOCTYPE html>
 <html lang="en">
 <head>
 <meta charset="UTF-8">
 <title>JavaScript Modules</title>
 </head>
 <body>
```

```
<script type="module" src="app.js"></script>
</body>
</html>
```

Best Practices for JavaScript Module

- **One Default Export per File:** Use default export for the primary functionality.
- **Use Named Exports for Utilities:** Export multiple utilities from a single module.
- **Keep Modules Small and Focused:** Each module should handle a specific functionality.
- **Use Clear and Descriptive Names:** Ensure exported functions/variables have meaningful names.

Consuming Modules:

In JavaScript, consuming modules means importing and using functionalities (like functions, variables, or classes) that were exported from another file. This process ensures code modularity, reusability, and maintainability**.**

Key Points About Importing Modules

- Use **import** to bring exported functionality into your code.
- Modules must be explicitly referenced using their file path (`./module.js`) or package name (`lodash`).
- In HTML , the <script> tag must have type="module"` to support ES modules.

Alias Named Imports

You can rename imported variables using **as** .

```
import { add as sum } from './math.js';
console.log(sum(5, 3)); // Output: 8
```

Default Imports

Each module can have **one default export**, usually representing the main functionality.

Math.js

```
// Default export
export default function multiply(a, b) {
 return a * b;
 }
```

App.js

```
// Importing default export
import multiply from './math.js';
console.log(multiply(5, 3)); // Output: 15
```

Dynamic Imports (Lazy Loading)

JavaScript allows you to **dynamically import** modules using **import().**

```
 async function loadModule() {
const math = await import('./math.js');
 console.log(math.add(5, 3)); // Output: 8 }
```

Importing External Modules (Node.js / NPM Packages)

You can also consume external modules installed via **npm .**

Installation Example:**bash npm install lodash**

```
import _ from 'lodash';
console.log(_.capitalize('hello')); // Output: Hello
```

Best Practices for Consuming Modules

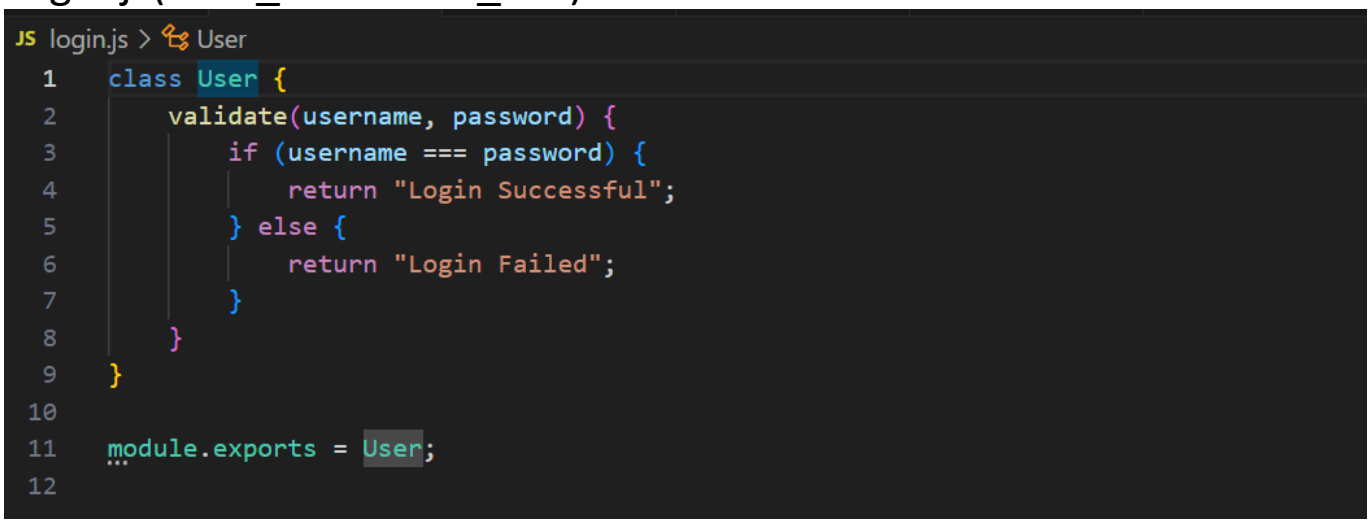
- **Keep Imports Organized:** Group similar imports together.
- **Use Named Imports for Specific Functions:** Avoid importing the entire module unnecessarily.
- **Prefer Default Exports for Single-Purpose Modules:** Clear and clean imports.
- **Lazy Load Modules When Possible:** Optimize performance.
- **Document Imported Functions:** Clarify module purposes.

Problem Statement:

Validate the user by creating a login module. Hints: (i) Create a file login.js with a User class. (ii) Create a validate method with username and password as arguments. (iii) If the username and password are equal it will return "Login Successful" else w

Code:

Login.js(User_Validation_File):



App.js(Main_File):

```
JS App.js > ...
1   //Import the User class from the login.js file
2   const User = require('./login.js');
3
4   // Create an instance of the User class
5   const user = new User();
6
7   // Test the validate method
8   const username = "admin";
9   const password = "admin";
10  console.log(user.validate(username, password));
11
```

Output:

```
[Running] node "c:\Users\Abhiram\OneDrive - Abhiram\Desktop\3-2\Mst lab\App.js"
Login Successful

[Done] exited with code=0 in 0.157 seconds
```

Experiment 6

Aim: Verify how to execute different functions successfully in the Node.js platform.

Introduction to Node.js :

Node.js is a powerful, open-source, server-side JavaScript runtime environment built on Chrome's V8 JavaScript engine. It allows developers to use JavaScript for server-side scripting, enabling the creation of dynamic, scalable, and high-performance web applications.

Features :

- **Asynchronous & Event-Driven:** Handles multiple requests without blocking.
- **Single-Threaded:** Uses an event loop for concurrency.
- **NPM:** Built-in package manager with thousands of libraries.
- **High Performance:** Converts JavaScript to machine code for speed.
- **Cross-Platform:** Runs on Windows, macOS, and Linux.

Installation of Node.js :

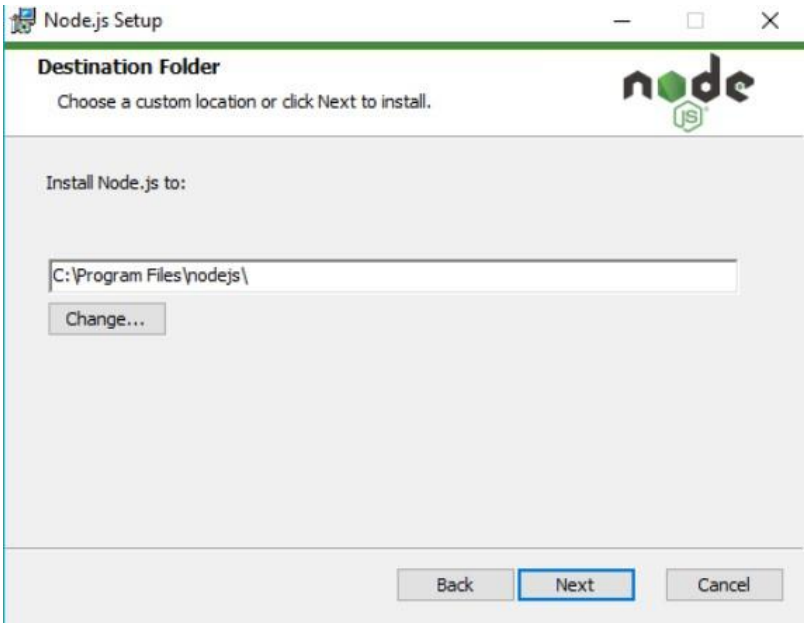
Step 1 : Download Node.js

- Go to the official Node.js website.
- Select the **LTS (Recommended)** version
- Download the .msi installer for Windows.

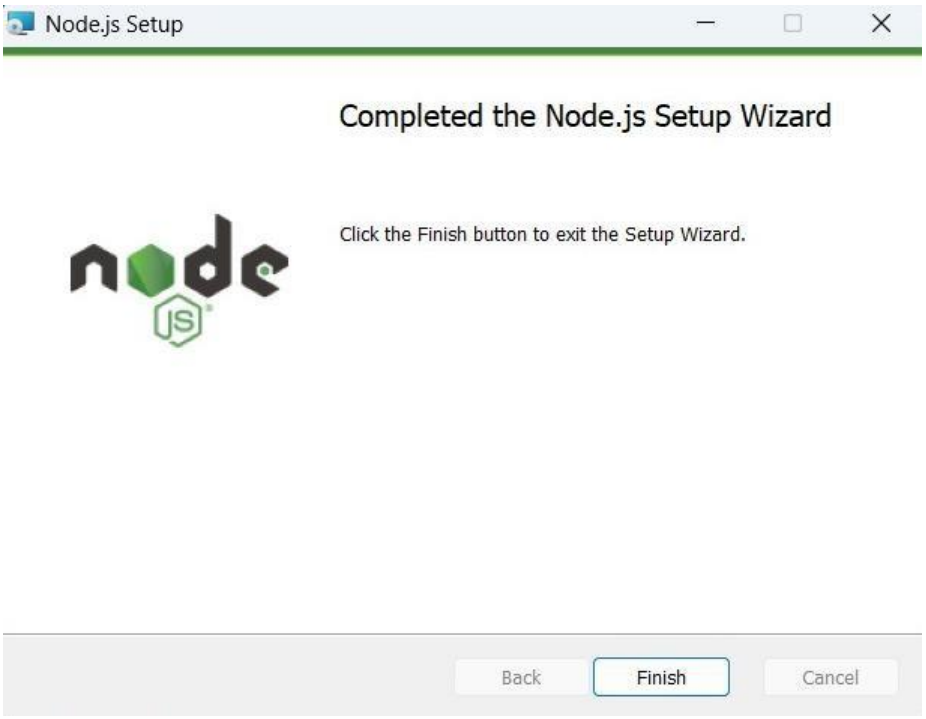
Step 2 : Install Node.js

- Run the downloaded .msi file.

- Click **Next** to proceed.
- Accept the license agreement.
- Choose the installation folder or keep the default location.



- Ensure the "Add to PATH" option is checked.
- Click **Install** to complete the process.



Step 3 : Verify Installation

- Open the **Command Prompt (cmd)** or **PowerShell**.
- Check the installed versions by running:

  node -v

  npm -v

Types of Functions in Node.js :

1. Synchronous Functions :

- These functions execute sequentially, blocking the execution of the program until they complete.
- They are straightforward but can lead to performance issues in I/O-heavy applications.

```
function functionName(parameters) {
  // Function body
  return result;
}
```

2. Asynchronous Functions :

- These functions allow the program to continue executing while waiting for an operation to complete.
- They are essential for non-blocking I/O operations and can be implemented using callbacks, promises, or async/await syntax.

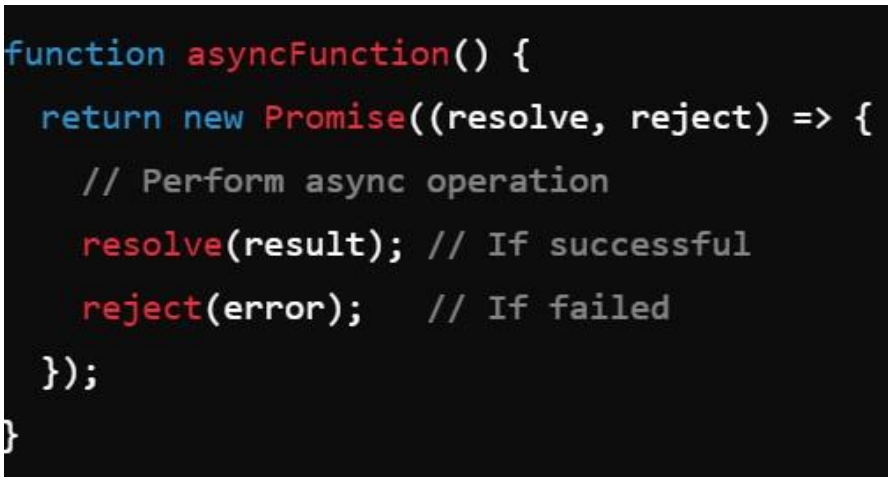
3. Callback Functions :

- These are functions passed as arguments to other functions and are executed after the completion of an asynchronous operation.
- They are a fundamental part of Node.js's asynchronous programming model.

```
function asyncFunction(callback) {
  // Perform asynchronous operation
  callback(err, result);
}
```

4. Promise-based Functions :

- Promises provide a cleaner way to handle asynchronous operations by allowing you to chain operations and handle errors more gracefully.
- They help avoid callback hell and make the code more readable.



5. Async/Await Functions :

- **async/await** syntax allows you to write asynchronous code that looks synchronous.
- It simplifies the handling of promises and makes the code easier to understand and maintain.

6. Higher-Order Functions :

- These functions take other functions as arguments or return them as results.
- They are commonly used in functional programming and can help create more modular and reusable code.

```
function higherOrderFunction(fn) {
  return function(param) {
    return fn(param);
  };
}
```

7. Arrow Functions :

- **arrow** functions provide a concise syntax for writing functions.
- They are especially useful for writing short, anonymous functions and have a lexical this binding.

```
const functionName = (parameters) => {
  return result;
};
```

Execution :

1. **Create a JavaScript File**
   Write your code in a .js file. For example, create a file named app.js.

2. **Define the Function and call the Function**
   Inside the file, define the function you want to execute and invoke the function.

3. **Run the File Using Node.js**
   Open a terminal in the directory where your file is saved and execute the file using the node command:

```
node app.js
```

Built-in Modules :

- Node.js comes with a rich set of **built-in modules** that provide essential functionalities for building efficient and scalable applications.
- These modules eliminate the need for external libraries for common tasks like file handling, networking, or cryptography.

How to Use Built-in Modules :

1. **Import the Module** : Use the require() function to include a module.

```
const moduleName = require('module_name');
```

2. **Call Methods** : Use the module's methods and properties to perform tasks.

Example :

The **fs** module enables you to interact with the file system to read,

```
const fs = require('fs');

// Reading a file
fs.readFile('example.txt', 'utf8', (err, data) => {
    if (err) throw err;
    console.log(data);
});

// Writing to a file
fs.writeFile('output.txt', 'Hello, Node.js!', (err) => {
    if (err) throw err;
    console.log('File written successfully!');
});
```

write, update, and delete files.

Module Name	Description
fs	Handles file system operations like reading, writing, and deleting files.
path	Provides utilities for working with file and directory paths.
http	Enables the creation of HTTP servers and clients.
https	Provides functionality for secure HTTPS communication.
url	Helps in parsing, constructing, and manipulating URLs.
querystring	Parses and formats URL query strings.
os	Provides information about the operating system, like memory and CPU details.
events	Implements an event-driven programming model.
crypto	Offers cryptographic functionalities like hashing and encryption.
stream	Facilitates working with data streams (e.g., readable and writable streams).

zlib	Enables compression and decompression of data using Gzip and other formats.
buffer	Works with binary data buffers in Node.js.
child_process	Allows the creation and control of subprocesses.
dns	Provides DNS resolution and domain name lookups.
timers	Handles scheduling tasks (e.g., **setTimeout**, **setInterval**).
assert	Provides a set of assertion testing utilities.
readline	Interfaces for reading input from a readable stream (e.g., console).
v8	Gives insights into the V8 JavaScript engine internals.
worker_threads	Enables multithreading for parallel processing.

Advantages :

1. Performance
2. Reliability
3. Integration

External Modules :

- Node.js allows developers to use external modules created by the community or third parties.
- These modules can be downloaded from the **Node Package Manager (npm)** and integrated into your project.

Step 1 : Install the Module

- Use the npm command to install the module in your project:

```
npm install <module-name>
```

- Import the installed module in your script using require
- Once imported, you can use the module's functions and features in your code.

Example: moment Module

1. Install the moment module

```
npm install moment
```

2. Now import the module and use it.

```
const moment = require('moment');

// Display current date and time
console.log(moment().format('YYYY-MM-DD HH:mm:ss'));

// Add days to the current date
console.log(moment().add(7, 'days').format('YYYY-MM-DD'));

// Check if a date is valid
console.log(moment('2025-01-01', 'YYYY-MM-DD').isValid());
```

Advantages :

1. Code Reusability
2. Community Support
3. Efficiency
4. Scalability

Module Name	Description
express	Web framework for building APIs and web applications.
mongoose	MongoDB object modeling for Node.js.
axios	Promise-based HTTP client for making API requests.
dotenv	Load environment variables from a .env file.
chalk	Add colorful output to the console.
cors	Handle Cross-Origin Resource Sharing in APIs.
bcrypt	Password hashing for authentication.
jsonwebtoken	JSON Web Token for authentication and authorization.
nodemailer	Send emails from your Node.js application.

Uninstalling an External Module : To remove the module we use the following command.

```
npm uninstall <module-name>
```

Debugging Tools :

- Debugging is an essential part of software development.
- Node.js provides a variety of tools for debugging your applications effectively.
- Below are some of the most commonly used debugging tools in Node.js:

1. console.log()

- The simplest form of debugging. console.log() outputs values to the console, helping you track the flow of your program and inspect variables.

2. node-inspect

```
node inspect app.js
```

- node-inspect is a command-line debugger for Node.js.

3.debug Module

The debug module is a popular external package used for advanced logging and debugging. It allows conditional logging based on namespaces, providing a more flexible way to debug applications.

4.Visual Studio Code (VSCode) Debugger

VSCode is one of the most popular IDEs for JavaScript and Node.js development, and it provides excellent debugging capabilities for Node.js.

B) Module Name: Create a web server in Node.js

Aim: Write a program to show the workflow of JavaScript code executable by creating web server in Node.js.

Introduction

Node.js provides an efficient way to create a web server using its built-in `http` module.

What is a Web Server?

A web server is a software application that listens for incoming HTTP requests from clients, such as web browsers, and responds with the requested resources, such as HTML pages, images, and data.

Node.js Web Server Architecture

The Node.js web server architecture consists of the following components:

1. *HTTP Server*: The HTTP server is the core component of the web server, responsible for listening for incoming HTTP requests and responding with the requested resources.

2. *Request Handler*: The request handler is a function that handles incoming HTTP requests and sends responses back to the client.

3. *Request Object*: The request object represents the incoming HTTP request and contains properties such as the request method, URL, headers, and body.

4. *Response Object*: The response object represents the outgoing HTTP response and contains properties such as the response status code, headers, and body.

Creating a Web Server in Node.js

Steps to create a web server in Node.js:

1. Import the ‘http’ module.

2. Create an HTTP server instance using the ‘http.createServer()’ method.

3. Define a request handler function that will handle incoming HTTP requests.

4. Start the HTTP server using the ‘server.listen()’ method.

Here is a step-by-step explanation of the program:

Step 1: Import the http module

const http = require(‘http’);

- This line imports the ‘http’ module, which is a built-in module in Node.js.

- The ‘http’ module provides functionality for creating an HTTP server.

- The ‘require’ function is used to import modules in Node.js.

Step 2: Create an HTTP server instance

```
const server =
 http.createServer((req, res) => {
…
});
```

- This line creates a new instance of an HTTP server using the ‘http.createServer()’ method.

- The ‘createServer()’ method takes a callback function as an argument, which is called whenever the server receives a new request.

- The callback function takes two arguments: ‘req’ (the request object) and ‘res’ (the response object).

Step 3: Handle incoming HTTP requests

```
res.writeHead(200, {‘Content-Type’: ‘text/plain’});
res.end(‘Hello World! I have created my first server!\n’);
```

- This code handles incoming HTTP requests by sending a response back to the client.

- The ‘writeHead()’ method is used to set the HTTP status code and headers of the response.

- In this case, we're setting the status code to 200 (OK) and the ‘Content-Type’ header to ‘text/plain’.

- The ‘end()’ method is used to send the response body back to the client.

- In this case, we're sending the string “Hello World!\n” as the response body.

Step 4: Start the HTTP server

```
server.listen(3000, () => {
  console.log(‘Server running on http://localhost:3000’);
});
```

- This line starts the HTTP server listening on port 3000 using the ‘listen()’ method.

- The ‘listen()’ method takes two arguments: the port number to listen on, and a callback function to call when the server is listening.

- In this case, we're logging a message to the console when the server is listening, indicating that the server is running on ‘http://localhost:3000’.

Running the Program

To run the program, follow these steps:

1. Save the program in a file named ‘WebServer.js’.

2. Open a terminal or command prompt and navigate to the directory where you saved the file.

3. Type ‘node WebServer.js’ to run the program.

4. Press Enter to execute the command.

Output:

When you run the program, you should see the following output:

```

Server running on http://localhost:3000

```

This indicates that the server is running and listening on port 3000.

Testing the Server

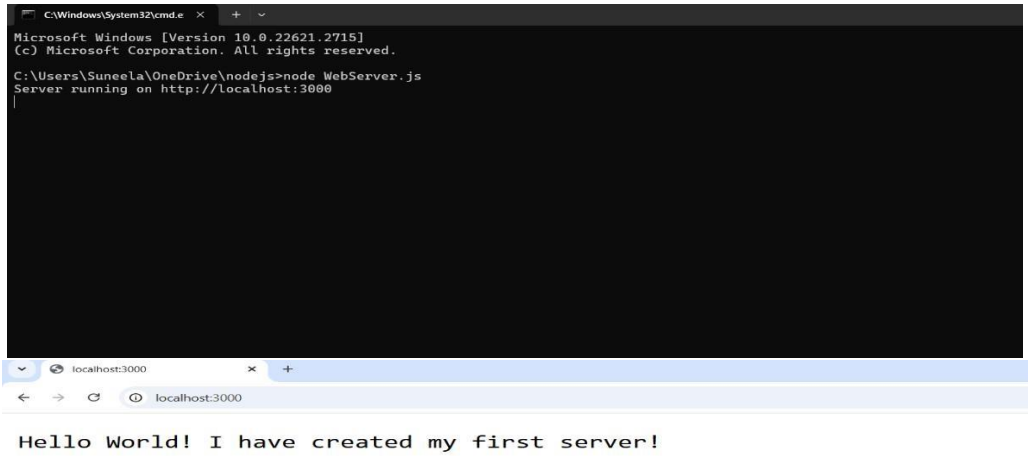
To test the server, follow these steps:

1. Open a web browser and navigate to ‘http://localhost:3000’.

2. You should see the following output:

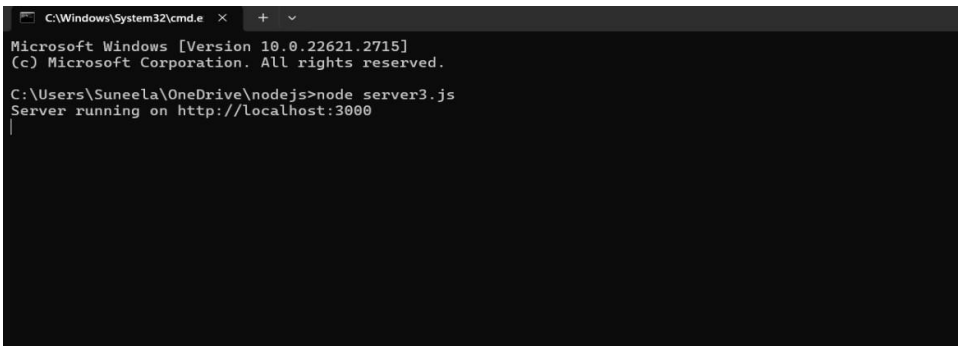
```

Hello World! I have created my first server!

```

This indicates that the server is working correctly and sending the expected response.



Advanced Web Server

```
const http = require('http');
const server = http.createServer((req, res) => {
if (req.method === 'GET' && req.url === '/') {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World!\n');
    } else if (req.method === 'GET' && req.url === '/about') {
      res.writeHead(200, {'Content-Type': 'text/plain'});
      res.end('This is the about page!\n');
    } else {
      res.writeHead(404, {'Content-Type': 'text/plain'});
      res.end('Not Found\n');
    }
});
server.listen(3000, () => {
  console.log('Server running on http://localhost:3000');
});
```



Uses:

1. **Handle multiple routes**: This server handles multiple routes,including the root URL("/") and the "/about" URL.

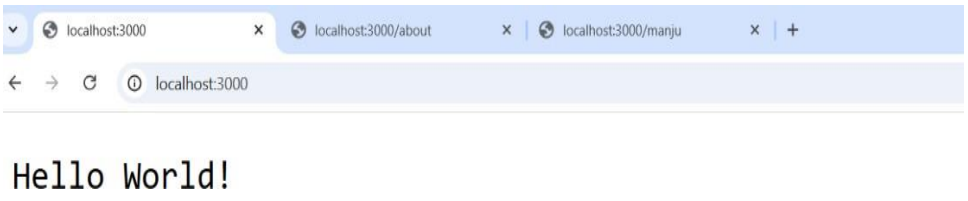
2. **Returns different responses**: Returns different responses based on requested URL.

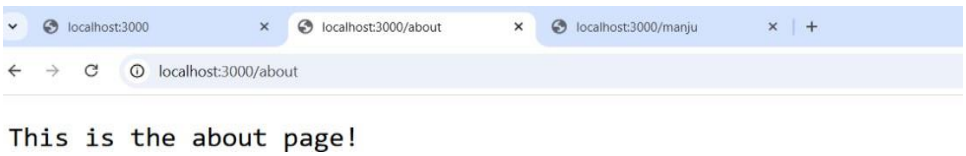
Here are the URLs that can be used to test the advanced web server:

1. **Home Page**: http://localhost:3000/

2. **About Page**: http://localhost:3000/about

3. **Unknown (Not Found):** http://localhost:3000/unknown





Exercise-6c

Aim: Write a Node.js module to show the workflow of Modularization of Node application.

Description:

In 2009, modules were introduced in node.js.

- The initial module system adopted by Node.js was the CommonJS module system, which uses require() to import modules.
- We use module.exports to export them.
- This system became the standard for Node.js modules and is still widely used today.
- Modules are a fundamental part of Node.js, enabling developers to build scalable and maintainable applications.

What are modules:

- In Node.js, a module is a reusable piece of code that encapsulates related functionality.
- Modules help in organizing code into smaller, manageable parts, promoting reusability and separation of concerns.
- Functions defined in a module are not accessible outside of it unless explicitly exported.

Types:

There are some main types of modules in Node.js:

Built-in Modules:

- These are modules provided by Node.js itself, such as http, fs, path, and many others.
- They are part of the Node.js runtime and can be used without any additional installation.

Some important modules:

- http: For creating HTTP servers and clients.
- fs: For interacting with the file system.
- path: For working with file and directory paths.
- url: For URL resolution and parsing etc…

User-defined Modules:

- These are custom modules created by developers to encapsulate specific functionality.
- They can be organized into separate files and directories within a project.

Third Party Modules:

- Third-party modules in Node.js are packages created by the community and shared via package managers like npm (Node Package Manager).
- These modules extend the functionality of Node.js applications and cover a wide range of use cases.

Some important examples:

- **Express:** For building web applications.
- **Lodash:** Utility functions.
- **Mongoose:** MongoDB object modelling.
- **Axios:** A promise-based HTTP client for making HTTP requests from Node.js

Built in Module Wrapping:

- Every module in Node.js is wrapped inside a function when loaded to create its own scope. This prevents variables in one module from interfering with others.

Es Modules (ECMAScript Modules):

- Introduced in Node.js 12+, ES modules use the import and export syntax and provide a modern way to handle modules.

Example for built-in modules:

```
// Importing the built-in 'fs' module
const fs = require('fs');

// Reading the contents of a file
fs.readFile('example.txt', 'utf8', (err, data) => {
    if (err) {
        console.error('Error reading file:', err);
        return;
    }
    console.log('File contents:', data);
});
```

Example for User defined modules:

```
// math.js
module.exports.add = (a, b) => a + b;
module.exports.subtract = (a, b) => a - b;

// app.js
const math = require('./math');

console.log('Addition:', math.add(2, 3));
console.log('Subtraction:', math.subtract(5, 2));
```

Example for third party modules:

Express: A fast, unopinionated, and minimalist web framework for Node.js. It simplifies the process of building web applications and APIs.

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
    res.send('Hello, world!');
});

app.listen(3000, () => {
    console.log('Server is running on port 3000');
});
```

Example for Built-In Module Wrapping:

Wrapper Function:

```
(function (exports, require, module, __filename, __dirname) {
    // Module code here
});
```

Example for Es Modules: math.js

```
export const add = (a, b) => a + b;
```

Usage:

```
import { add } from './math.js';
console.log(add(5, 3));
```

Why do we use modules:

Code Organization: Modules help us in organizing code into smaller, manageable pieces.

Reusability: We can reuse the code at different parts of our code in different projects.

Separation of Concerns: Modules allow us to separate different functionalities into distinct units.

Namespace Management: Modules help us in avoiding naming conflicts by providing their own scope.

Dependency Management: With the help of package managers like npm, modules make it easy to manage dependencies.

Community and Ecosystem: The Node.js ecosystem has a vast number of modules available through npm.

Source Code:

DBModule.js

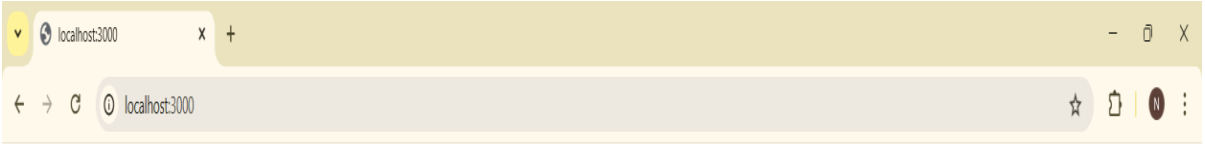
```
exports.authenticateUser=(username,password)=>{

    if(username==="admin"&&password==="admin"){ return "Valid User";
    }
    else return"InvalidUser";

    };
```

app.js

```
const http=require("http");
var dbmodule= require("./DBModule");
var server = http.createServer((request, response) => {
result = dbmodule.authenticateUser("admin","admin");
response.writeHead(200,{"Content-Type":"text/html"});
response.end("<html><body><h1>"+result +"</h1></body></html>");
console.log("Request received");
});
server.listen(3000);
console.log("Serverisrunningatport3000");
```

```
PS C:\Users\MANAS\OneDrive\Desktop\node.js> node app.js
Serverisrunningatport3000
Request received
Request received
```

Output:



Valid User

Write a program to show the workflow of restarting a Node application?

A) At-First we have to know about the restart of a node application and about how to use the workflow for node application also.

In the above task you have already can understand about these let me give small definition of restart and workflow of node application.

Restarting a node application definition:

Restarting a Node.js application typically means stopping the currently running application process and starting it again.

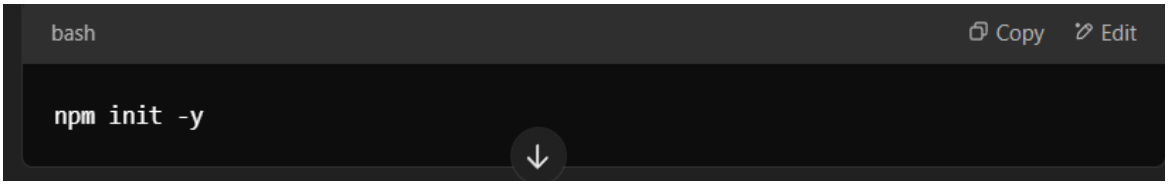
Workflow definition:

A workflow refers to the sequence of processes and tasks involved in building, running, and managing a Node.js application. It includes the steps from writing the code to deploying it and handling requests. Node.js workflows are often **event-driven** and **non-blocking**, leveraging the asynchronous nature of JavaScript.

General Workflow in Node.js

1. Setup and Initialization

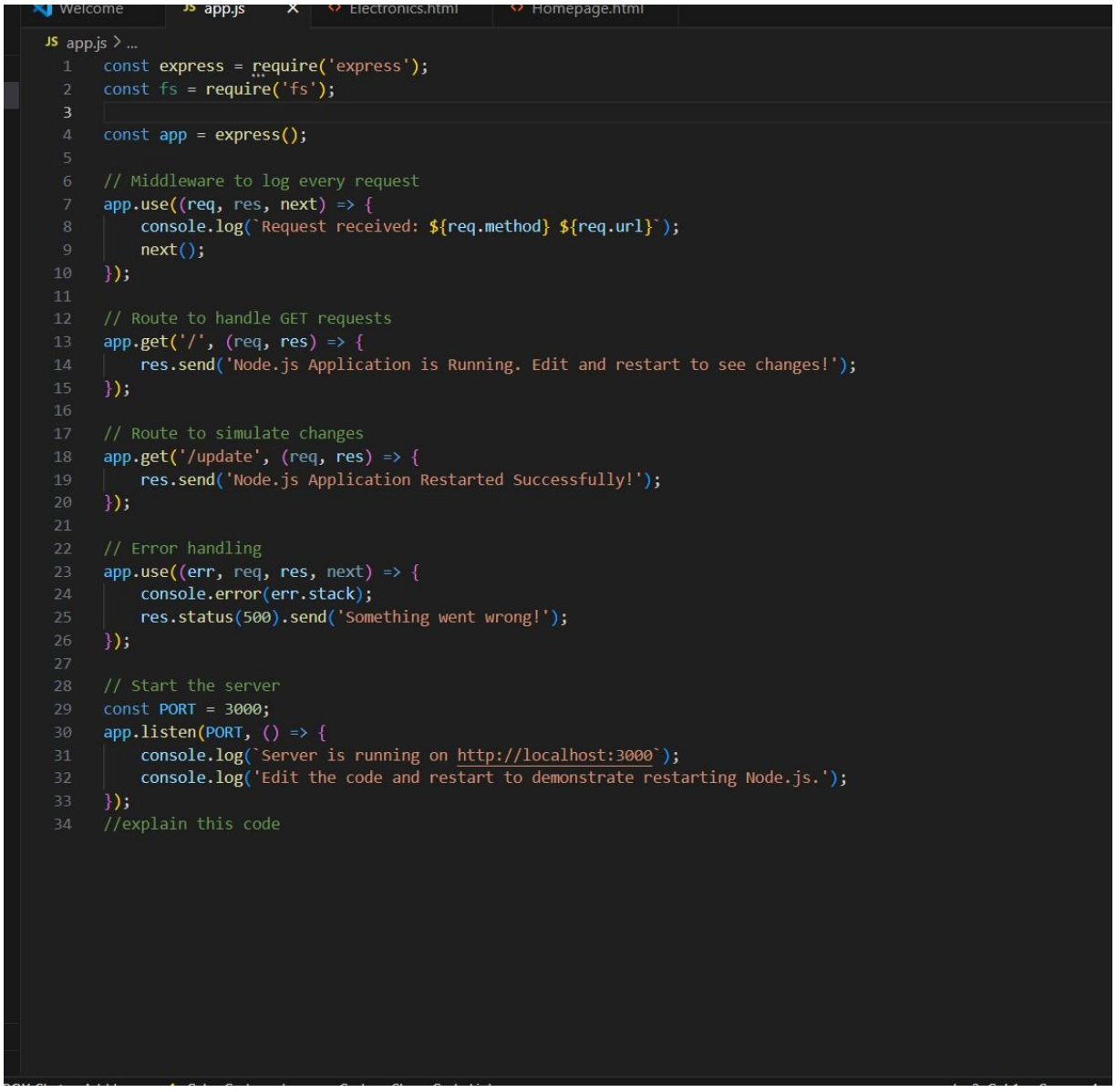
- Install **Node.js** on your system.
- Create a new project directory.



Install dependencies using npm:

Workflow to restarting a Node Application:

Source code:



This code sets up a Node.js application using the Express framework. It demonstrates a simple workflow for running and restarting a Node.js application while also showing how requests are handled and logged. Let’s break it down step by step:

1. **Importing Required Modules:**

```javascript
const express = require('express');
const fs = require('fs');
```

- **express:** A popular web framework for Node.js that simplifies handling HTTP requests and routing.
- **fs:** Node.js' built-in module for interacting with the file system. Though it's imported, it's not actively used in this code. It could be used later for file operations like logging or reading data.

2. **Creating the Express Application:**

```javascript
const app = express();
```

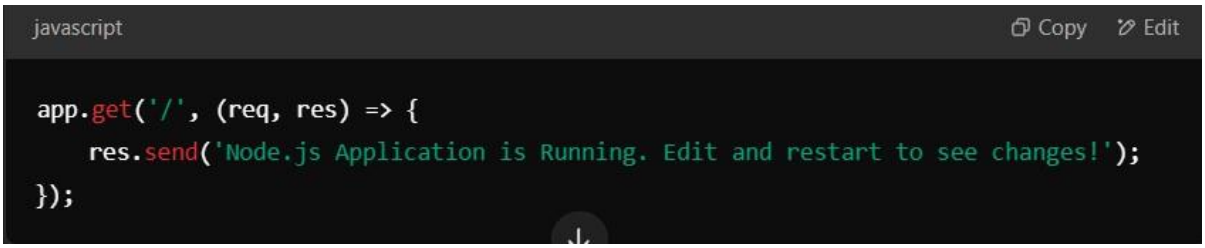
- **app**: Represents the Express application instance. This is used to define routes, middleware, and server configurations.

3. **Middleware to Log Requests:**

```javascript
app.use((req, res, next) => {
    console.log(`Request received: ${req.method} ${req.url}`);
    next();
});
```

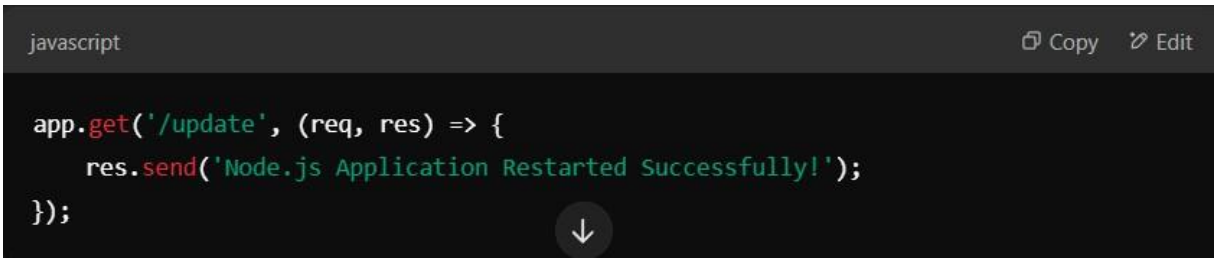
- Middleware is a function that executes during the lifecycle of an HTTP request.
- This middleware logs the HTTP method (e.g., GET, POST) and the URL of each incoming request to the console.
- The next () function is called to pass control to the next middleware or route handler.

4. **Defining Routes:**
   a. **Root Route:**



- This handles GET requests to the root URL (/).
- The res.send() method sends a plain text response to the client, confirming that the application is running.

b. Update Route:

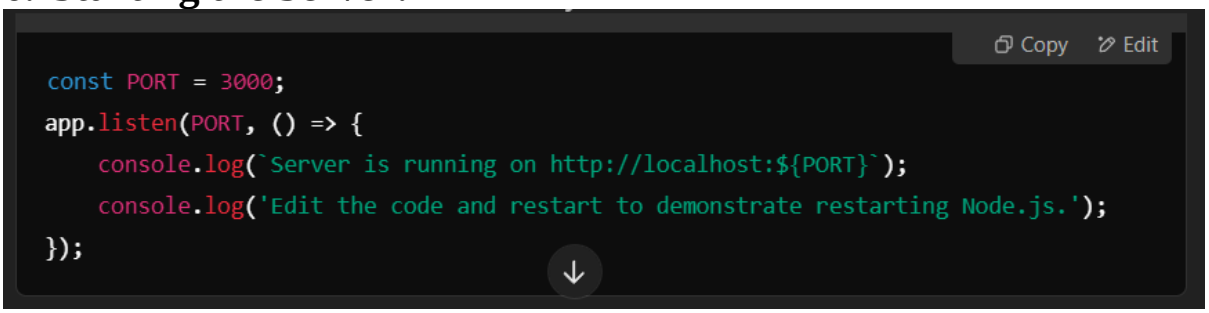


- This handles **GET requests** to /update.
- The res.send() method sends a message to simulate that the application has restarted (though no actual restart logic is included here).

5. Error Handling Middleware:

- This middleware catches errors occurring in any part of the application.
- err.stack logs the full error stack trace to the console.
- Sends a 500 (Internal Server Error) response with a message, "Something went wrong!".
- The next parameter is included but not used here since this is the last middleware in the stack.

6. Starting the Server:



- The **app.listen(PORT)** method starts the Express server on port 3000.
- The callback function is executed once the server starts successfully, logging the URL to access the application.

Purpose of This Application

1. **Basic Request Handling**:
   - The app responds to two routes: / (default route) and /update.
   - Both routes demonstrate how to send responses to clients.
2. **Middleware Demonstration**:
   - A custom middleware logs every incoming request to the console.
3. **Error Handling**:
   - Ensures proper handling of unexpected errors with middleware.
4. **Demonstrating Restart**:
   - The **restart simulation** is done using the /update route. To truly restart the app, you would need to manually stop and restart the Node.js process or use a tool like **Nodemon** or **PM2**.

How to Demonstrate Restarting

Start the application:

```bash
node app.js
```

Access the root route in your browser or via a tool like Postman:

```arduino
http://localhost:3000/
```

- Response: *"Node.js Application is Running. Edit and restart to see changes!"*
- Modify the app.js file (e.g., change the response text of any route).
- Restart the app manually by stopping the current process (CTRL+C) and running node app.js again.

Enhancements for Real Restarts:

To automate restarts when changes are made to the code, you can use Nodemon:

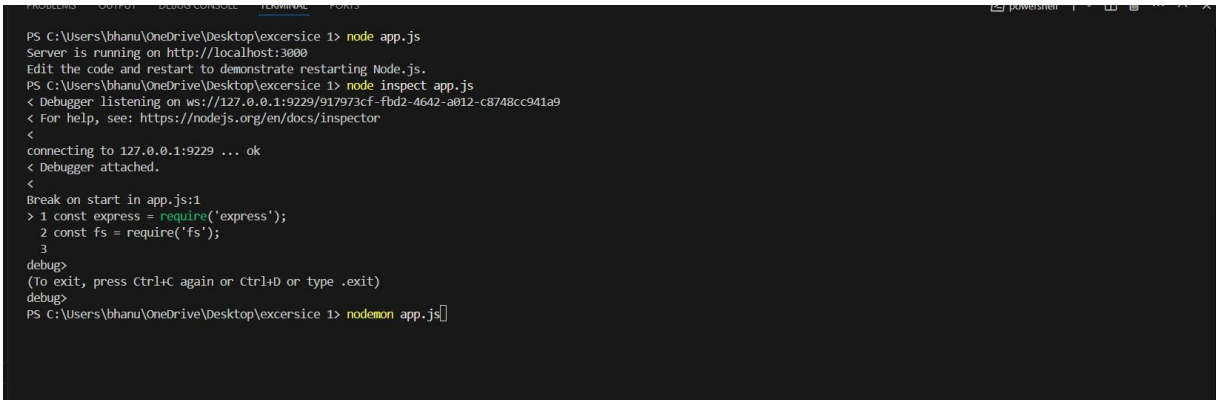
1. **Install Nodemon:**

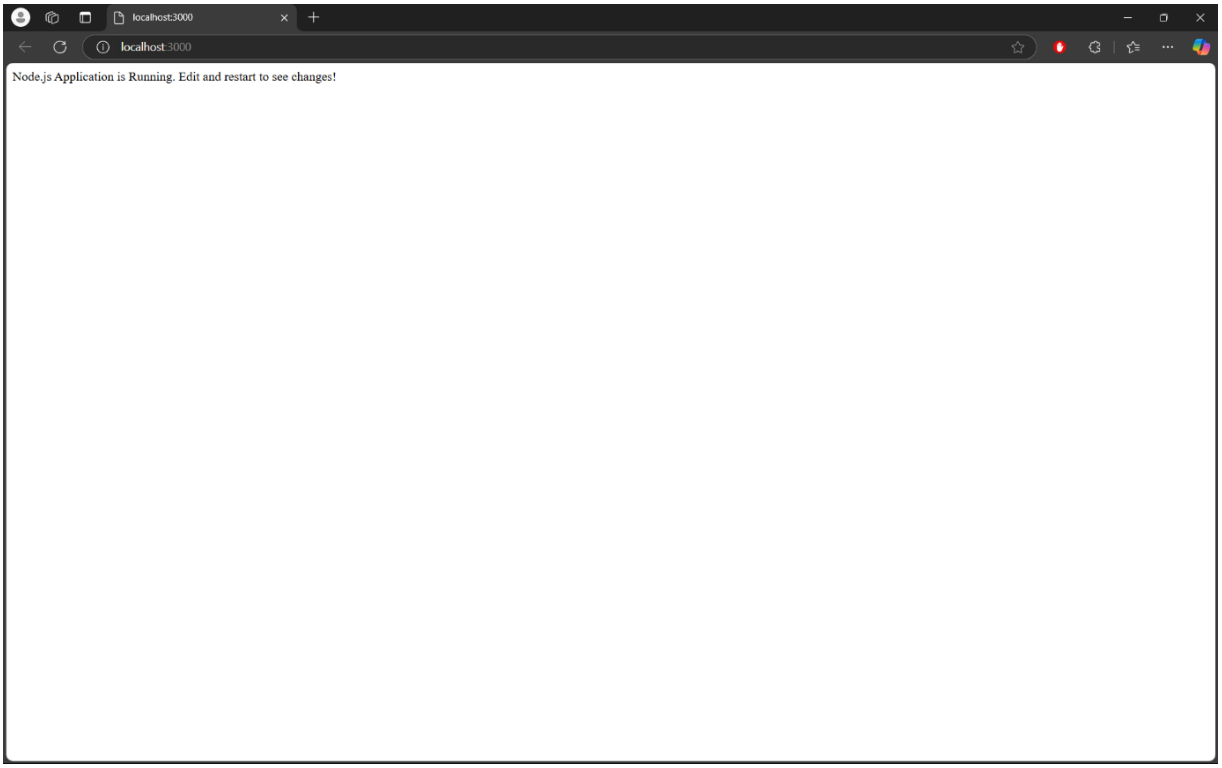
```bash
npm install -g nodemon
```

2. **Start the app with Nodemon:**

```bash
nodemon app.js
```

Now, the app will restart automatically whenever you make changes to the code.





This application demonstrates the following:

1. Setting up an Express-based Node.js application.
2. Handling HTTP requests and middleware.
3. Logging and error handling.
4. Simulating a workflow where the app is restarted to reflect changes.

File Operations in node.js

The **`fs`** module in Node.js provides functions to interact with the server's file system. It supports **synchronous** (blocking) and **asynchronous** (non-blocking) methods for reading and writing files.

Importing the `fs` Module

```
const fs = require('fs');
```

---

1. Writing a File

- **Synchronous Method**
  Blocks the execution until the write operation is complete.

```
fs.writeFileSync('input.txt', 'Hello, this is synchronous write!');
console.log('File written synchronously.');
```

- **Asynchronous Method**
  Non-blocking, uses a callback for error handling.

```
fs.writeFile('input.txt', 'Hello, this is asynchronous write!', (err)
=> {
    if (err) console.error(err);
    else console.log('File written asynchronously.');
});
```

---

2. Reading a File

- **Synchronous Method**
  Reads the file and returns its content as a string.

```
const data = fs.readFileSync('input.txt', 'utf8');
console.log('Synchronous read:', data);
```

- **Asynchronous Method**
  Uses a callback to handle the file content or errors.

```
fs.readFile('input.txt', 'utf8', (err, data) => {
    if (err) console.error(err);
    else console.log('Asynchronous read:', data);
});
```

Key Points

- **Synchronous Methods** block the program until the operation completes.
- **Asynchronous Methods** use callbacks and are non-blocking.
- Use **file descriptors** (`fd`) for advanced operations with `fs.open()`, `fs.write()`, and `fs.read()`.

Course name: Expess.js

Group no-7

Introduction to Express: Express is a layer built on the top of the Node.js that helps manage server and routes.

-Node.js does not support request handling, HTTP methods so, this is where Express.js comes into picture.

-There are 34,372 companies using Express.js.

-It is a back-end framework.

-It also provides a robust set of features to develop applications (web and mobile).

-It is easy to learn, fast, free, flexible and unopinionated (there isn't a best way to do something, what you do is the best way)

-It is all JavaScript.

-It goes well with node.js.

-Express links quickly with databases like MySQL, MongoDB etc.

-Express makes debugging easier as it identifies the exact part where bugs are.

Advantages of ExpressJS:

1. Express is very easy we can customize and use it as per our needs.
2. A single language JavaScript is used for both front-end and back-end development.
3. It is very fast to link with databases like MySOL, MongoDB.

Dis-Advantages of ExpressJS:

1. No structural way to organizing things.
2. The error messages are not very helpful.

Installing Express (In windows):

Step 1- Download Node.js

Step 2- Now, open your visual studio code, open a new terminal and install npm (node package manager)

Step 3- After installing npm, you must check the version of npm(Command: npm -v)

Step 4- Create folder(that stores your express file) and open it in visual stdio

Step 5- Now, inside that folder create a file app.js

Step 6- write command: npm init (in terminal).

Press enter

Enter required details...

Package name: ExpressJS group7

Version: 1.0.0

Description: SOC lab

Entry point: app.js

Click enter..

Step 7- Now you will finally install express js for that write command: npm install express. With this Express js will have gotten installed on your system.

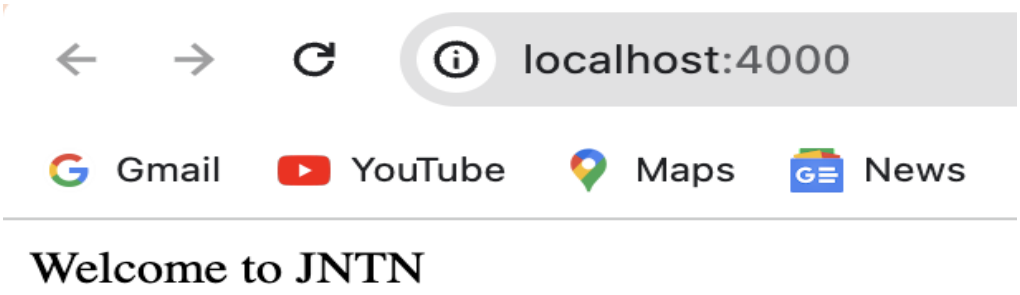
Step 8- Now write your first basic program (in app.js).

```
//First import express
const express= require('express')
//Initialize app using express. .
//with which we can perform tasks(app)
const app=express();
//Creating a route for the method get in the path "/"
app.get('/',(req,res)=>{
res.send("Welcome to JNTN");
})
//The response to our req is shown in the port:4000
app.listen(4000,()=>{
console.log("listening to port 4000");
})
```

Step 9- Open the terminal, go to the main folder write ‘cd <<folder name>>’, press enter, and then write the command to run the code: node app.js.

Step 10- Now open chrome and go to local host: 4000 to see the result.

Route:



Each route corresponds to an HTTP method (e.g., GET, POST, PUT, DELETE) and a specific URL pattern/path.

Example: We define routes using methods like **"app.get()"** and **"app.post()".**

Routers():

Defining routes like above is very tedious to maintain. To separate the routes from our main index.js file, we will use Express.Route

Route methods:(Examples)

// **GET method** route

```
app.get('/', (req, res) => {
  res.send('GET request to the homepage')
})
```

// **POST method** route

```
app.post('/', (req, res) => {
  res.send('POST request to the homepage')
})
```

Route paths:(Examples)

```
//Route path is "/random.text"
        app.get('/random.text', (req, res) => {
          res.send('random.text')
        })
//Route path is "/about"
```

```
app.get('/about', (req, res) => {
  res.send('about')
})
```

Route parameters:

Route parameters are named URL segments that are used to capture the values specified at their position in the URL.

EXAMPLE:

Request URL: http://localhost:3000/user/42

req.params: {"userId": "42"}

1. Implementing routing for Adventure Trails application by embedding necessary code in the routes/ route.js file.

An ***adventure trails application*** is a web application, built with Express.js could be a platform for users to discover, explore, and share information about outdoor trails for activities such as hiking, biking, or running.

Index.js file:

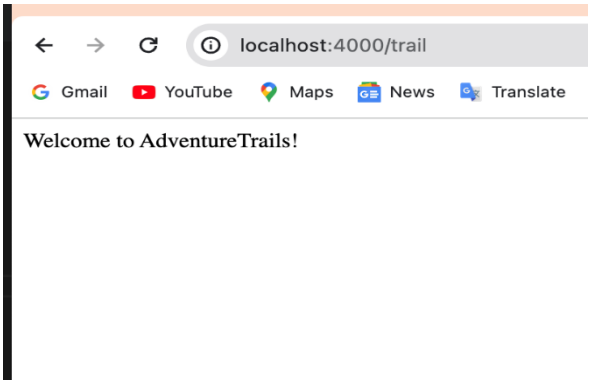
```
const express = require('express');
const app = express();
  //naming the path as route1
const route1 = require('./routes/route');
// opening file route for path /trail
app.use('/trail', route1);
// Start the server
app.listen(4000,()=>{
console.log("listening to port 4000")
})
```

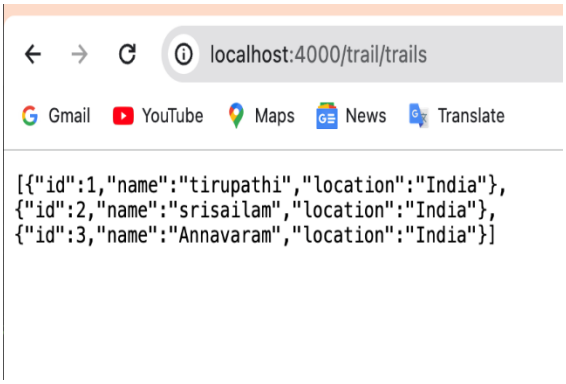
routes/route.js file:

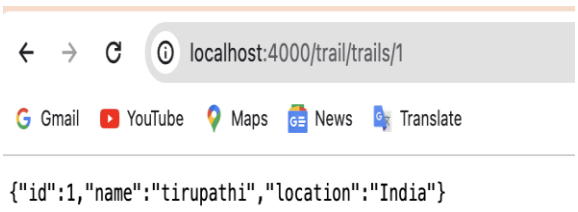
```
const express = require('express');
const router = express.Router();
// Sample trails data (replace this with your actual data source)
const trails = [
```

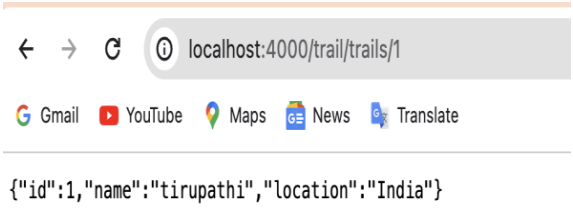
```
{ id: 1, name: 'tirupathi', location: 'India' },
{ id: 2, name: 'srisailam', location: 'India' },
{ id: 3, name: 'Annavaram', location: 'India' }
];
// Route for homepage or list of trails
router.get('/', (req, res) => {
res.send('Welcome to AdventureTrails!');
// Example: res.render('index');
});
// Route for getting all trails
router.get('/trails', (req, res) => {
res.send(trails);
});
// Route for getting a specific trail by ID
router.get('/trails/:id', (req, res) => {
const trailId = parseInt(req.params.id);
const trail = trails.find(trail => trail.id === trailId);
if (trail) {
res.json(trail);
}
});
// Other routes for creating, updating, and deleting trails can be added here
module.exports = router;
```

Output:









(7b) Middleware

What's middleware?

They are functions. The middleware functions have access to response obj (res), request object(req) and next in the application req-res cycle.

It is used for--

1.Security check of user's access/authorization.

2.Body parsing of the request (whether it has correct requested data).

3.What region data is from.

4.What was the data that was requested.

Order of writing these middleware functions is important

Code example for Middleware:

router. use (‘/', function (req, res, next) {

console.log(" req call for middle ware ");

next();

//next method is used to chain the middleware in application.

});

Types of Middleware --

1.Application level middleware –The function is executed every time the app receives a request.For any route this middleware function works.

```
const express = require('express');
const app = express();

app.use((req, res, next)=> {
 console.log('Time:', Date.now());
  next();});
//next method is used to chain the middleware in application.
```

2. **Router level middleware** – when a route is activated then only these middlewares will work. Here we can use multiple middlewares:

```
const express = require('express');
const app = express();
const router = express.Router();
router.use((req, res, next)  =>{
  if (!req.headers['x-auth']) return next('router');
  next();
});

router.get('/user/:id', (req, res)=> {
  res.send('hello, user!');
});
app.use('/admin', router, (req, res) => {
  res.sendStatus(401);
});
```

3. **Builtin middleware** -

There will be some builtin middlewares.

Ex: express.json(), express.urlencoded(), express.static().

4. **Third party middleware** – Which adds functionality to Express JS as it cannot parse data, for which we use a third-party middleware i.e, BodyParser.

```
const express = require('express')
const app = express()
const cookieParser = require('cookie-parser')
app.use(cookieParser())
```

5. **Error handling middleware** - To handle errors in requests

```
app.use((err, req, res, next) => {
  console.error(err.stack)
  res.status(500).send('Something broke!')})
```

2.(i) we want to handle POST submissions.

The POST request is handled in Express using the Post express method. This is used when there is a requirement of fetching data from the client such as form data. The data sent through the POST request is stored in the request object

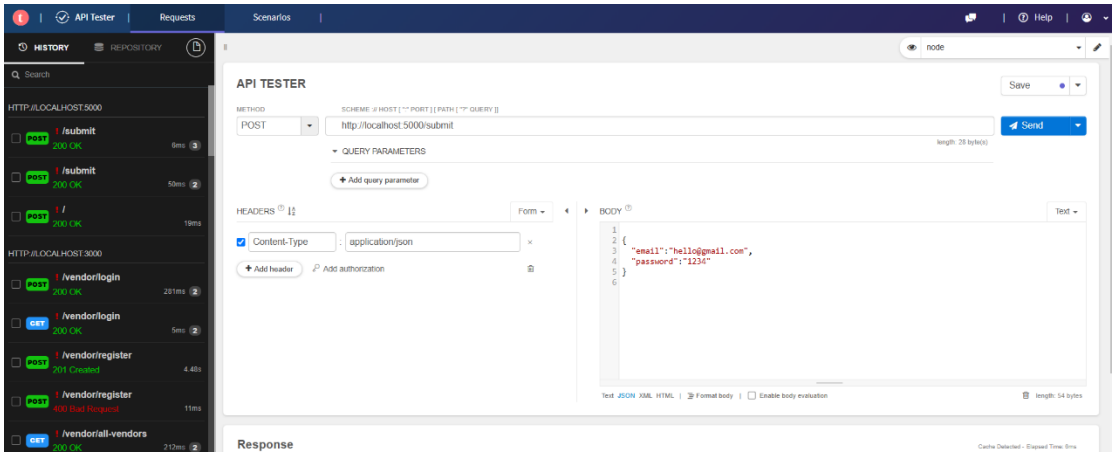
Express requires an additional middleware module to extract incoming data of a POST request. This middleware is called the **"body-parser".** We need to install it and configure it with Express instance.

You can install **body-parser** using the following command: **npm install body-parser**

You need to import this package into your project and tell Express to use this as middleware.

Source code:

```
import express from 'express';
const app=express();
import bodyParser from 'body-parser';
const PORT = 5000
app.use(bodyParser.json());

app.post('/submit',(req,res)=>{
    try {
        const data = req.body;
        console.log("data recieved")
        res.json({message:"data recieved successfully",recivedData: data})
    } catch (error) {
        console.log(error);
        res.status(500).json({message:"internal server errror"})
    }
})
app.listen(PORT,()=>{
    console.log(`server running successfully ${PORT}`);
})
app.use('/',(req,res)=>{
    res.send("hello world");
})
```



Choose the HTTP method as "POST" from the dropdown menu. Enter the URL of your Express.js server. For example, if your server is running locally on port 5000, the URL would be http://localhost:5000/submit.

Optionally, you can add headers, query parameters, or authentication if your server requires them **Add request body:** Switch to the "Body" tab in Api Tester.

- Select the "raw" option.
- Choose JSON format from the dropdown menu.
- Enter some sample data in JSON format. For example:

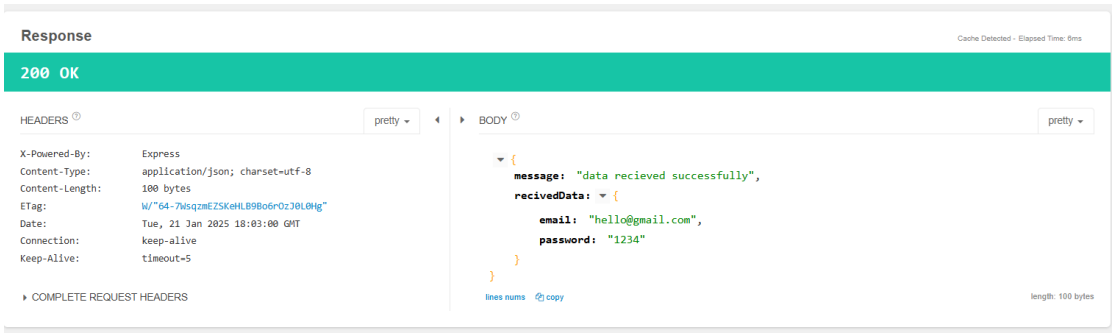
{

"email": “hello@gmail.com”

“passowrd”:”1234”

}

Click on the "Send" button to send the POST request to your Express.js server. Once the request is sent, you should see the response from your server (localhost: 5000).



Errors in express.js: In Express.js, errors can occur for various reasons during the execution of your application. some common types of errors you might encounter are: "**Syntax Errors, Runtime Errors, Database Errors, Custom Errors.**"

(II). Custom Errors: Custom errors are errors that you define yourself to handle specific scenarios in your application.

1. Create Custom Error Handler middleware:

// ErrorHandler.js in middlewares folder.

```
const errorhandles=(err, req, res, next) => {
  console.error(err.message);
  res.status(500).json({
    success: false,
    message: err.message || 'Something went wrong!',
  });
};
export default errorhandles
```

2. Attach Custom Error Handler as The Last Middleware to Use

import ErrorHandler from "./middlewares/ErrorHandler.js";

// ERROR HANDLER MIDDLEWARE (Last middleware to use)

```
app.use(errorhandler);
```

3. How to Call the Error-handler

to call the Error-handler, use the next() in Express.
The next function is a function in the Express router which, when invoked, executes the middleware succeeding the current middleware.

Example:

```
app.post('/submit',(req,res,next)=>{
  try {
      const data = req.body;
      console.log("data recieved")
      res.json({message:"data recieved successfully",recivedData: data})
```

```
    } catch (error) {
        next(errorhandler)
    }
})
```

Error Response(output):


```
{

 "success": false,

    "status": 500,
    "message": "Something went wrong",
    "stack": "Error: Custom error message\n    at app.get (/path/to/your/app.js:10:15)\n    at Layer.handle [as handle_request] (/path/to/node_modules/express/lib/router/layer.js:95:5)\n    at next (/path/to/node_modules/express/lib/router/route.js:137:13)\n    at Route.dispatch (/path/to/node_modules/express/lib/router/route.js:112:3)\n    at Layer.handle [as handle_request] (/path/to/node_modules/express/lib/router/layer.js:95:5)\n    at /path/to/node_modules/express/lib/router/index.js:281:22\n    at Function.process_params (/path/to/node_modules/express/lib/router/index.js:335:12)\n    at next (/path/to/node_modules/express/lib/router/index.js:275:10)\n    at Function.handle (/path/to/node_modules/express/lib/router/index.js:174:3)\n    at router (/path/to/node_modules/express/lib/router/index.js:47:12)"
}
```


(III). Logging in express.js:

Logging is the process of recording events, actions, or information from a software application to a log file or another output destination. In the context of web development and server-side programming, logging involves capturing and storing various types of information related to the operation of the application.

Logging is an essential part of software development and maintenance for several reasons: “Debugging and Troubleshooting, Monitoring and Performance Analysis, Security Analysis”.

There are various logging libraries available for Node.js and Express.js, such as “morgan, winston, and bunyan”.

Winston: It is a flexible and versatile logging library for **Express.js** that allows developers to log messages at various levels and send those logs to multiple destinations (called "transports"), such as the console, files, or external services.

1. Install winston:

If you haven't already installed winston, you can do so using npm: npm install Winston

2. Use winston :

```
import winston from 'winston';

const logger = winston.createLogger({
  level: 'info',
  format: winston.format.combine(
    winston.format.timestamp(),
    winston.format.printf(({ timestamp, level, message }) => {
      return `${timestamp} [${level.toUpperCase()}]: ${message}`;
    })
  ),
  transports: [
    new winston.transports.Console(),
    new winston.transports.File({ filename: 'logs/app.log' })
  ]
});

export default logger;
```

If you use the provided logging setup in your Express.js application, the output will be displayed in the console where your application is running. Here's what the output might look like:

Server is running on localhost: 5000

2025-01-22T12:00:00.000Z [INFO]: HTTP GET / 2025-01-22T12:00:05.000Z [INFO]: Handled GET request for /

2025-01-22T17:19:52.531Z [INFO]: Server is running on localhost:5000

2025-01-22T17:20:29.932Z [INFO]: Server is running on localhost:5000

2025-01-22T17:20:54.953Z [INFO]: Server is running on localhost:5000

2025-01-22T17:21:26.663Z [INFO]: Server is running on localhost:5000

Custom Logging:

If you need more customized logging or want to log additional information, you can create custom middleware to handle logging.

Code:

```
app.use((req, res, next) => {

  console.log(`${new Date().toISOString()} - ${req.method} ${req.url}`);
  next();

});
```

Output for custom logging is:

2025-01-22T12:30:45.678Z - GET /

2025-01-22T12:30:48.123Z - GET /about

(7c) What is MongoDB?

MongoDB is a popular NoSQL database known for its flexibility, scalability, and high performance. It stores data in a document-oriented format (JSON-like objects called BSON).

Key Features:

- Schema-less, allowing dynamic fields in documents.
- Stores data as collections (analogous to tables) and documents (analogous to rows).
- Suitable for modern web applications, real-time analytics, and big data systems.

What is Mongoose?

Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js. It provides a schema-based solution to model application data, enabling the use of MongoDB with robust validation, type casting, and other features.

Key Features:

- Schema definition for collections.
- Built-in validation.
- Middleware for pre/post-hooks.
- Query helpers and virtual properties.

Steps to Connect MongoDB Using Mongoose

1. Install MongoDB and Mongoose

- Install MongoDB on your machine or use a hosted service like MongoDB Atlas.
- Install Mongoose using npm:

npm install mongoose

2. Set Up Your Node.js Project

- Create a new Node.js project and initialize it.

3. Import Mongoose

- In your project, create an index.js file and import Mongoose:

```
const mongoose = require('mongoose');
```

4. Connect to MongoDB

- Use the mongoose.connect function to establish a connection.
- Replace yourDatabaseName with the name of your database.

Example:

```
const mongoose = require('mongoose'); // Connect to MongoDB
mongoose.connect('mongodb://localhost:27017/yourDatabaseName', {
  useNewUrlParser: true,
  useUnifiedTopology: true
}).then(() => console.log('Connected to MongoDB'))
 .catch(err => console.error('Error connecting to MongoDB:', err));
```

5. Define a Schema

Schema: A blueprint that defines the structure of documents in a MongoDB collection.

Example:

```
const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true },
  age: { type: Number, min: 18 }
}); // Create a model based on the schema
const User = mongoose.model('User', userSchema);
```

6. Perform Database Operations

Once connected and schemas are defined, perform CRUD (Create, Read, Update, Delete) operations.

Example: Creating a New User

```
const createUser = async () => {
  const user = new User({
    name: "Alice",
    email: "alice@example.com",
    age: 30
  });
  try {
    const result = await user.save();
    console.log("User Created:", result);
  } catch (err) {
    console.error("Error creating user:", err.message);
  }
};
createUser();
```

Summary of Steps:

1. Install MongoDB and Mongoose.
2. Import Mongoose in your project.
3. Connect to the MongoDB server using mongoose.connect.
4. Define a schema using mongoose.Schema.
5. Perform database operations using Mongoose methods (save, find, update, delete, etc.).

```
const PORT=process.env.PORT || 7000;
const MONGOURL=process.env.MONGO_URL;
mongoose.connect(MONGOURL).then(()=>{
    console.log("connected to database");
    app.listen(PORT,()=>{
        console.log(`Server is running on port ${PORT}`);
    });
}).catch((error)=>console.log(error));

const userSchema = new mongoose.Schema({
    name: { type: String, required: true },
    email: { type: String, required: true },
    age: { type: Number, min: 18 }
  });

  const UserModel= mongoose.model("users",userSchema);
  app.get("/getUsers",async(req,res)=>{
    const userData=await UserModel.find();
    res.json(userData);
  });
```

(7d) What is Schema?

SCHEMA: A schema in Mongoose is a blueprint that defines the structure of documents within a MongoDB collection. It represents the shape of the data, including the fields and their types, along with any validation rules, default values, and other options.

MODEL OBJECT:

In the context of Mongoose and MongoDB, a model object represents a collection in your MongoDB database. It provides an interface for querying and manipulating documents within that collection.

When you define a Mongoose schema and then compile it into a Model using 'mongoose.model()', you create an instance of the Model object. This Model object allows you to perform CRUD (Create, Read, Update, Delete) operations on documents that adhere to the schema.

Here is a breakdown of what a model object does:

1. Abstraction of Collection: The Model object abstracts away the underlying MongoDB collection. You interact with the Model object to perform database operations instead of directly accessing the MongoDB collection.

2. Validation and Type Casting: When you create documents using the Model object's methods, Mongoose automatically validates the data against the schema's defined structure and types. It also performs type casting based on the schema's definitions.

3. Querying: The Model object provides methods for querying the database to retrieve documents. You can perform find, findOne, or aggregate queries to retrieve data based on specific criteria.

4. Manipulating Documents: You can create, update, and delete documents using the Model object's methods. These methods ensure that the operations are performed according to the schema's rules and validation constraints.

5. Middleware and Hooks: Mongoose allows you to define middleware and hooks at the Model level, which enables you to execute custom logic before or after certain operations, such as saving or removing documents.

In summary, a Model object in Mongoose encapsulates the behavior and structure of a MongoDB collection, providing a high-level interface for interacting with the database while enforcing schema validation and data consistency.

//Program

1.Open vs code create a file app.js

2.Open terminal give the command npm init –y

3.This will create a package.json file

4.Give the command npm install mongodb mongoose code in app.js

File: APP.JS

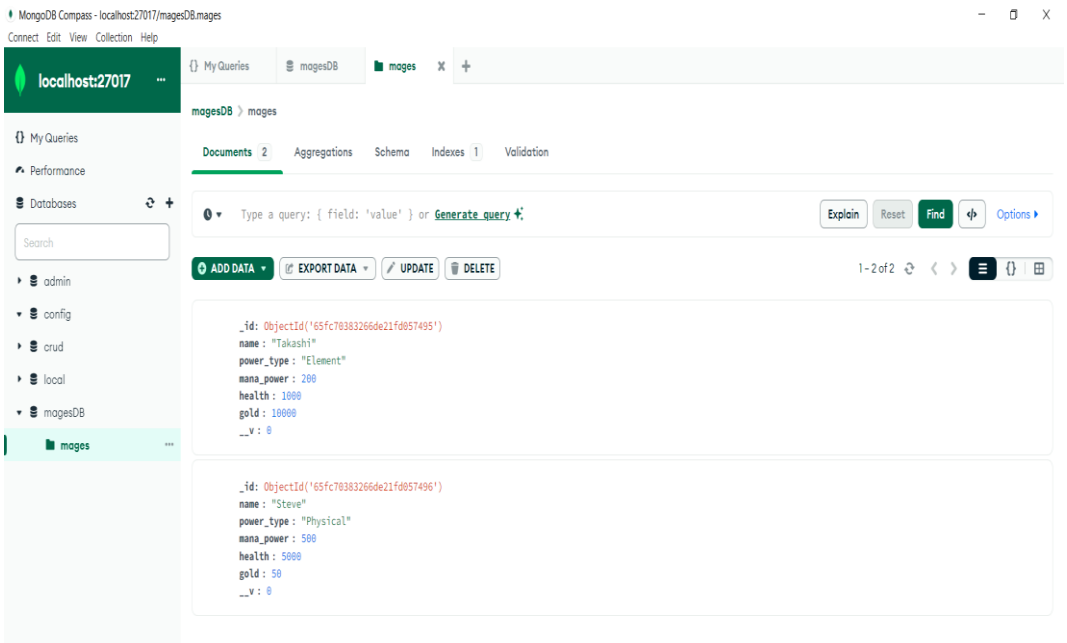
```
const mongoose = require('mongoose');
mongoose.connect("mongodb://localhost:27017/magesDB");
const mageSchema = new mongoose.Schema({
        name: {
                type: String,
                require: true
        },
        power_type: {
                type: String,
                require: true
        },
        mana_power: Number,
        health: Number,
        gold: Number
})
const Mage = new mongoose.model("Mage", mageSchema)
const mage_1 = new Mage({
        name: "Takashi",
        power_type: 'Element',
        mana_power: 200,
```

```
        health: 1000,
        gold: 10000
});
mage_1.save();
const mage_2 = new Mage({
        name: "Steve",
        power_type: 'Physical',
        mana_power: "500",
        health: "5000",
        gold: "50"
})
mage_2.save();
```

Output:

Here the model will be saved in the given hierarchy that is mentioned in the code(magesDB/images)

>node app.js



Exercise 8

8 a. Aim: Write a program to perform various CRUD (Create-Read-Update-Delete) operations using Mongoose library function.

Description:

CRUD stands for **Create**, **Read**, **Update**, and **Delete**, which are the basic operations to interact with a database. When combined with **Mongoose** (a MongoDB object modeling library) and **Express.js** (a web framework for Node.js), it allows you to build APIs for efficient database management.

CRUD Operations

Create: Add new data.

Read: Fetch existing data.

Update: Modify existing data.

Delete: Remove data.

Each operation corresponds to an HTTP method:

POST for Create.

GETfor Read.

PUT (or PATCH) for Update.

DELETEfor Delete.

Note:

Only **GET** operations can be directly performed in a browser, as they are used to retrieve and display data. For other operations like **POST**, **PUT**, and **DELETE**,

you will need tools like **Postman**, **curl**, or similar API testing tools to send appropriate requests with the required data.

Source Code:

1. Import Required Modules

```
constexpress=require("express");
constmongoose=require("mongoose");
constapp=express();
```

express: A web framework for building APIs and handling HTTP requests/responses.

mongoose: A library to interact with MongoDB using a object-oriented approach.

app: An instance of the Express application to define routes and middleware.

2. Middleware

```
// Middleware
app.use(express.json());
```

This middleware parses incoming JSON request bodies and makes them accessible via req.body.

3. Connect to MongoDB

```
// Connect to MongoDB
mongoose.connect("mongodb://127.0.0.1:27017/notesDB", {
  useNewUrlParser:true,
  useUnifiedTopology:true,
});
```

Connects to a local MongoDB instance on the notesDB database.

Options:

- useNewUrlParser: Uses the new MongoDB connection string parser.
- useUnifiedTopology: Enables new server discovery and monitoring engine.

4. Define a Schema and Model

```
// Schema and Model
constnoteSchema=newmongoose.Schema({
  title:String,
  content:String,
});
constNote=mongoose.model("Note", noteSchema);
```

Schema: Defines the structure of a document in the "Notes" collection. Here, each note has a title (String) and content (String).

Model: Represents the MongoDB collection and provides methods to interact with the database.

5. Create Sample Notes

```
// Create sample instances (if not already created)
asyncfunctioncreateSampleNotes() {
  constexistingNotes=awaitNote.find();
  if (existingNotes.length===0) {
    awaitNote.create({ title:"First Note", content:"This is the first note." });
    awaitNote.create({ title:"Second Note", content:"This is the second note." });
    console.log("Sample notes created.");
  } else {
    console.log("Sample notes already exist.");
  }
}
createSampleNotes();
```

Purpose: Checks if the collection already contains notes. If not, it creates two sample notes.

Logic:

- Note.find(): Fetches all existing notes.
- If no notes exist (length === 0), it adds two sample notes using Note.create().

6. Define CRUD Routes

a. Create a Note

```
// Create a new note
app.post("/notes", async (req, res) => {
  try {
    constnote=newNote(req.body);
    awaitnote.save();
    res.status(201).send(note);
  } catch (error) {
    res.status(400).send({ error:"Failed to create note", details:error.message });
  }
});
```

Purpose: Adds a new note.

Steps:

1. Creates a new Note instance using req.body data.
2. Saves it to the database using .save().
3. Responds with the created note or an error if saving fails.

b. Get All Notes

```
// Get all notes
app.get("/notes", async (req, res) => {
  try {
    constnotes=awaitNote.find();
    res.send(notes);
  } catch (error) {
    res.status(500).send({ error:"Failed to fetch notes", details:error.message });
  }
});
```

Purpose: Fetches all notes from the database.

Steps:

1. Uses Note.find() to retrieve all documents.
2. Sends the array of notes in the response.

c. Get a Note by ID

```
// Get a specific note by ID
app.get("/notes/:id", async (req, res) => {
  try {
    constnote=awaitNote.findById(req.params.id);
    if (note) {
      res.send(note);
    } else {
      res.status(404).send({ error:"Note not found" });
    }
  } catch (error) {
    res.status(500).send({ error:"Invalid ID or server error", details:error.message
});
  }
});
```

Purpose: Fetches a specific note using its ID.

Steps:

1. Extracts the id from req.params.
2. Uses Note.findById(id) to find the document.
3. If found, returns the note; otherwise, sends a 404 error.

d. Update a Note by ID

```
// Update a specific note by ID
app.put("/notes/:id", async (req, res) => {
  try {
    constnote=awaitNote.findByIdAndUpdate(req.params.id, req.body, { new:true });
    if (note) {
      res.send(note);
    } else {
      res.status(404).send({ error:"Note not found" });
    }
  } catch (error) {
    res.status(500).send({ error:"Failed to update note", details:error.message });
  }
});
```

Purpose: Updates a specific note by ID.

Steps:

1. Extracts the id from req.params and updated data from req.body.
2. Uses Note.findByIdAndUpdate(id, data, { new: true }) to update the note.
3. If successful, returns the updated note; otherwise, sends an error response.

e. Delete a Note by ID

```
// Delete a specific note by ID
app.delete("/notes/:id", async (req, res) => {
  try {
    constnote=awaitNote.findByIdAndDelete(req.params.id);
    if (note) {
      res.send({ message:"Note deleted successfully" });
    } else {
      res.status(404).send({ error:"Note not found" });
    }
  } catch (error) {
    res.status(500).send({ error:"Failed to delete note", details:error.message });
  }
});
```

Purpose: Deletes a specific note by ID.

Steps:

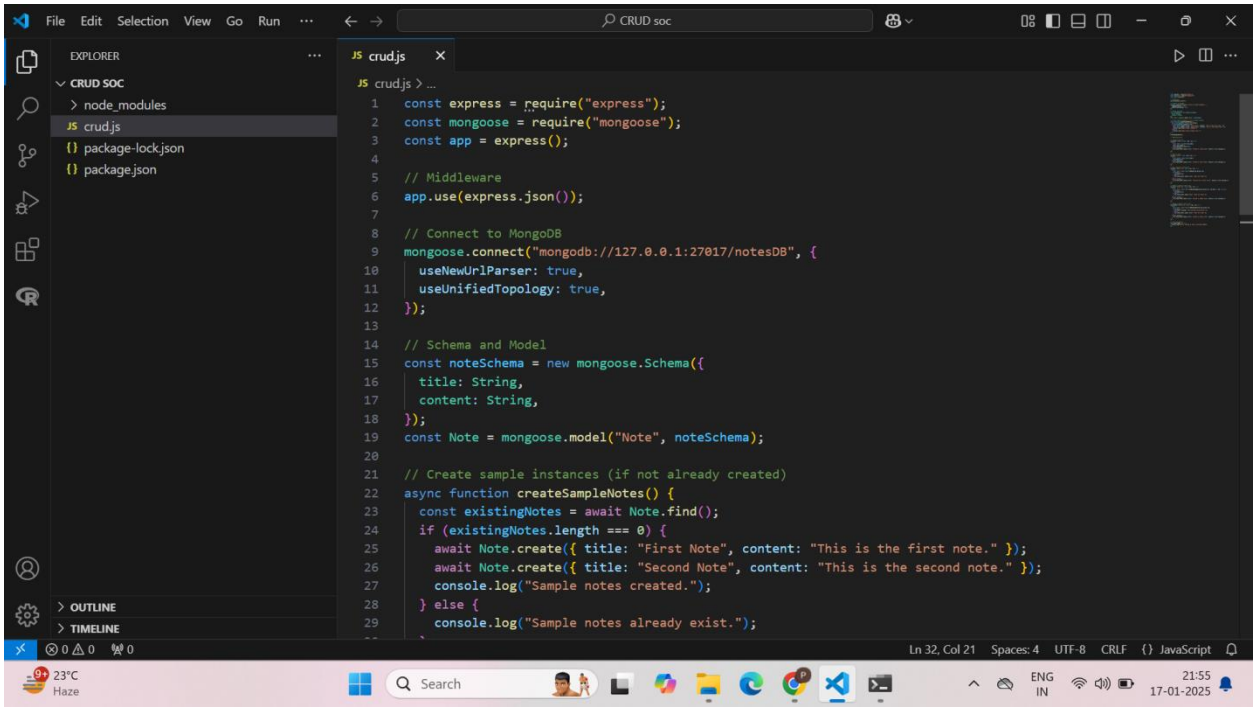
1. Extracts the id from req.params.
2. Uses Note.findByIdAndDelete(id) to remove the document.
3. Sends a success message or a 404 error if the note is not found.

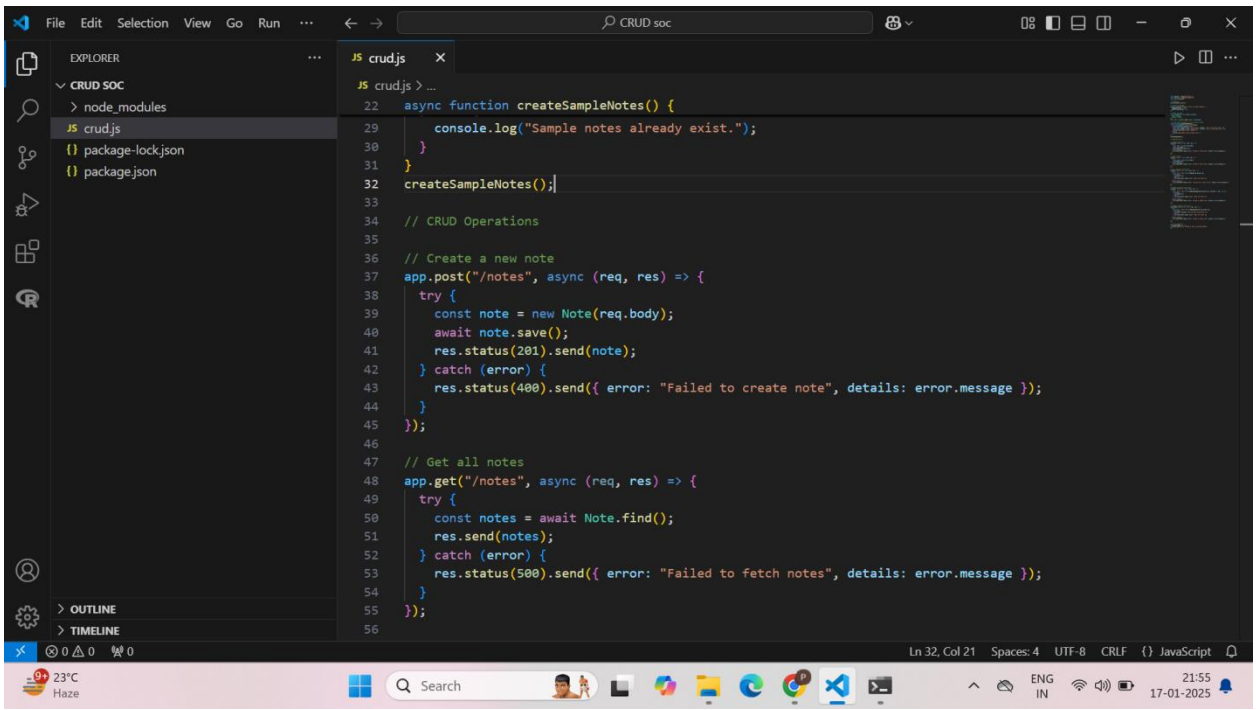
7. Start the Server

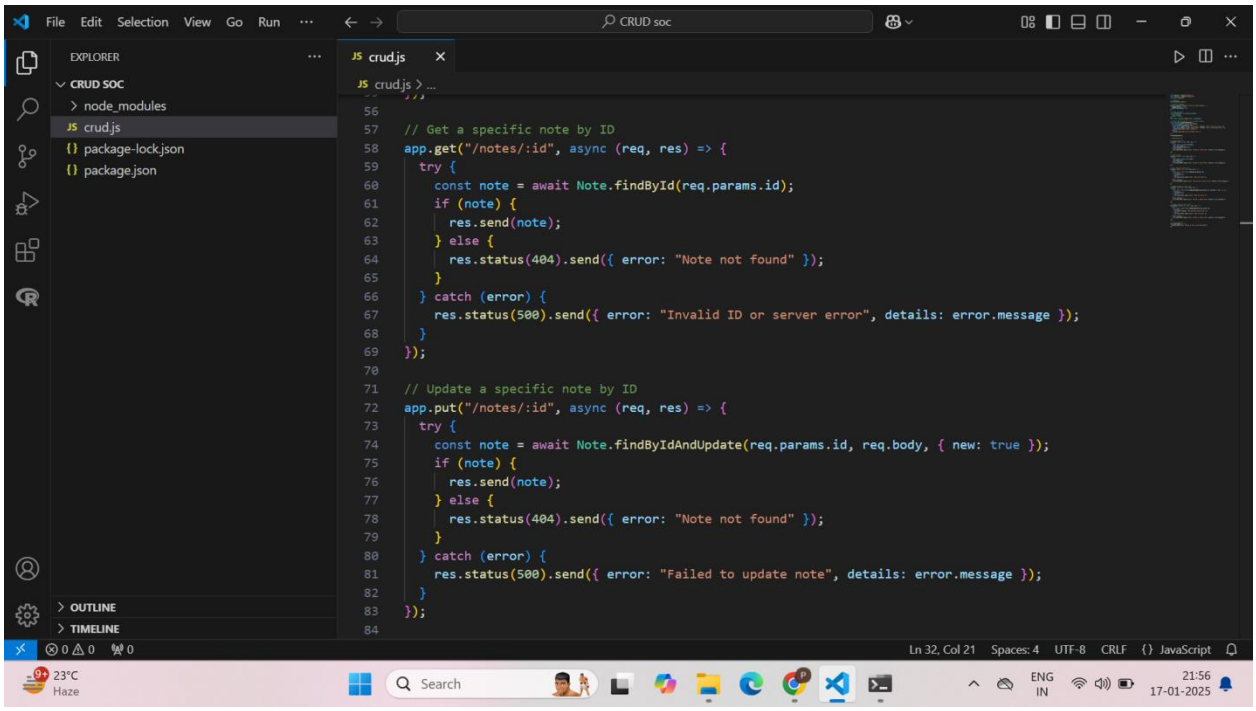
```
// Start the Server
app.listen(3000, () => {
  console.log("Server running on http://localhost:3000");
});
```

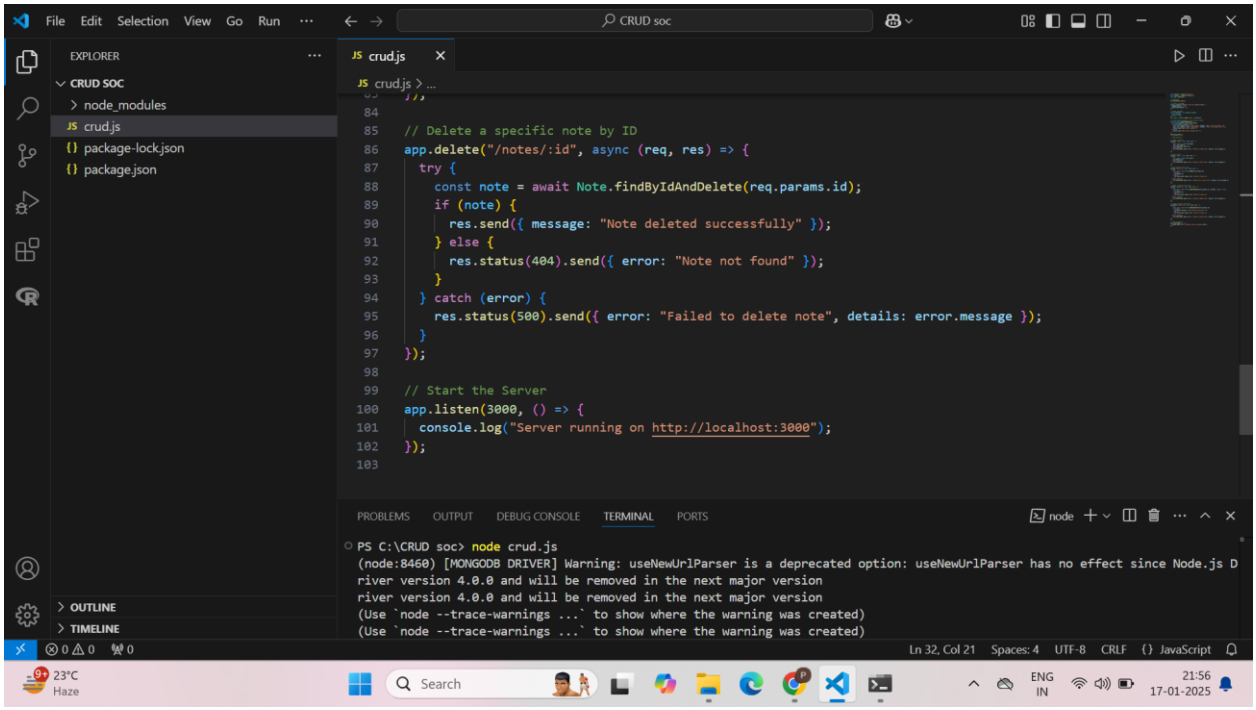
Starts the Express server on port 3000.

Code:

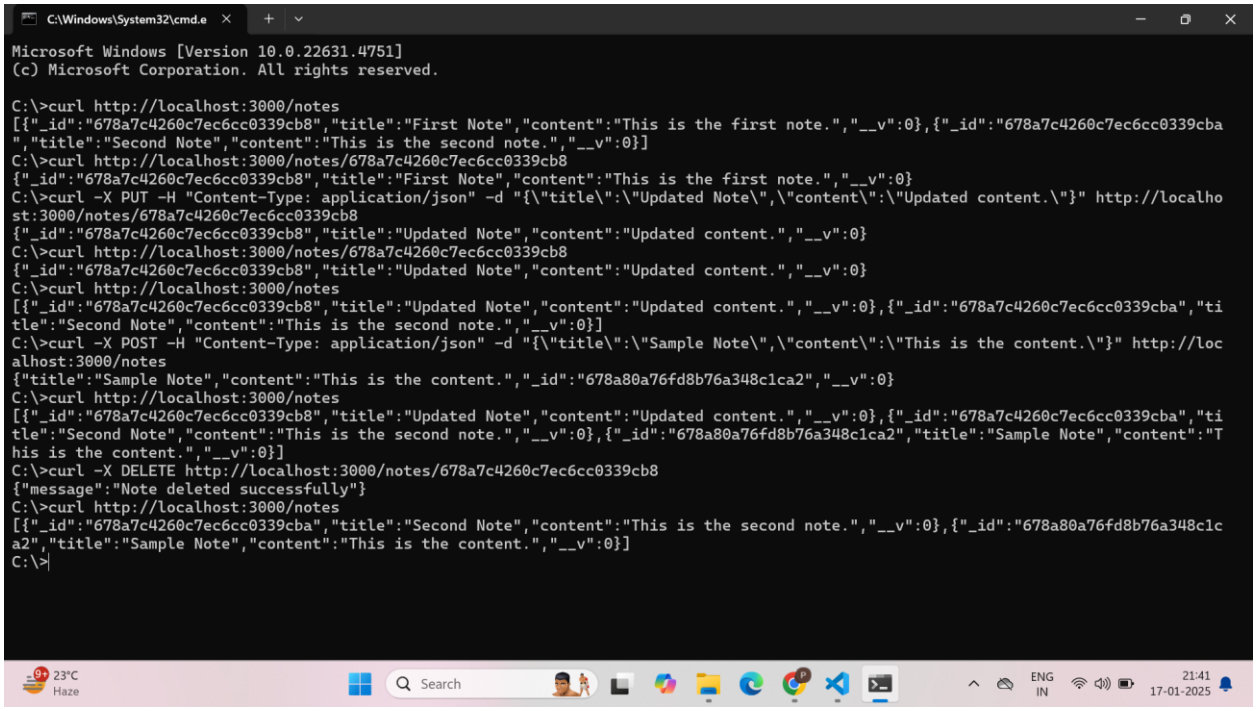

File Edit Selection View Go Run
CRUD soc
EXPLORER
CRUD SOC
node_modules
crud.js
package-lock.json
package.json
OUTLINE
TIMELINE
crud.js
crud.js > ...
1 const express = require("express");
2 const mongoose = require("mongoose");
3 const app = express();
4
5 // Middleware
6 app.use(express.json());
7
8 // Connect to MongoDB
9 mongoose.connect("mongodb://127.0.0.1:27017/notesDB", {
10 useNewUrlParser: true,
11 useUnifiedTopology: true,
12 });
13
14 // Schema and Model
15 const noteSchema = new mongoose.Schema({
16 title: String,
17 content: String,
18 });
19 const Note = mongoose.model("Note", noteSchema);
20
21 // Create sample instances (if not already created)
22 async function createSampleNotes() {
23 const existingNotes = await Note.find();
24 if (existingNotes.length === 0) {
25 await Note.create({ title: "First Note", content: "This is the first note." });
26 await Note.create({ title: "Second Note", content: "This is the second note." });
27 console.log("Sample notes created.");
28 } else {
29 console.log("Sample notes already exist.");
Ln 32, Col 21 Spaces: 4 UTF-8 CRLF {} JavaScript
23°C Haze
Search
ENG IN
21:55 17-01-2025


File Edit Selection View Go Run
CRUD soc
EXPLORER
CRUD SOC
node_modules
crud.js
package-lock.json
package.json
OUTLINE
TIMELINE
crud.js
crud.js > ...
22 async function createSampleNotes() {
29 console.log("Sample notes already exist.");
30 }
31 }
32 createSampleNotes();
33
34 // CRUD Operations
35
36 // Create a new note
37 app.post("/notes", async (req, res) => {
38 try {
39 const note = new Note(req.body);
40 await note.save();
41 res.status(201).send(note);
42 } catch (error) {
43 res.status(400).send({ error: "Failed to create note", details: error.message });
44 }
45 });
46
47 // Get all notes
48 app.get("/notes", async (req, res) => {
49 try {
50 const notes = await Note.find();
51 res.send(notes);
52 } catch (error) {
53 res.status(500).send({ error: "Failed to fetch notes", details: error.message });
54 }
55 });
56
Ln 32, Col 21 Spaces: 4 UTF-8 CRLF {} JavaScript
23°C Haze
Search
ENG IN
21:55 17-01-2025


File Edit Selection View Go Run
CRUD soc
EXPLORER
CRUD SOC
node_modules
crud.js
package-lock.json
package.json
crud.js
crud.js > ...
// Get a specific note by ID
app.get("/notes/:id", async (req, res) => {
try {
const note = await Note.findById(req.params.id);
if (note) {
res.send(note);
} else {
res.status(404).send({ error: "Note not found" });
}
} catch (error) {
res.status(500).send({ error: "Invalid ID or server error", details: error.message });
}
});
// Update a specific note by ID
app.put("/notes/:id", async (req, res) => {
try {
const note = await Note.findByIdAndUpdate(req.params.id, req.body, { new: true });
if (note) {
res.send(note);
} else {
res.status(404).send({ error: "Note not found" });
}
} catch (error) {
res.status(500).send({ error: "Failed to update note", details: error.message });
}
});
OUTLINE
TIMELINE
Ln 32, Col 21 Spaces: 4 UTF-8 CRLF {} JavaScript
23°C
Haze
Search
ENG
IN
21:56
17-01-2025


File Edit Selection View Go Run
CRUD soc
EXPLORER
CRUD SOC
node_modules
crud.js
package-lock.json
package.json
crud.js
crud.js > ...
// Delete a specific note by ID
app.delete("/notes/:id", async (req, res) => {
try {
const note = await Note.findByIdAndDelete(req.params.id);
if (note) {
res.send({ message: "Note deleted successfully" });
} else {
res.status(404).send({ error: "Note not found" });
}
} catch (error) {
res.status(500).send({ error: "Failed to delete note", details: error.message });
}
});
// Start the Server
app.listen(3000, () => {
console.log("Server running on http://localhost:3000");
});
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
node
PS C:\CRUD soc> node crud.js
(node:8460) [MONGODB DRIVER] Warning: useNewUrlParser is a deprecated option: useNewUrlParser has no effect since Node.js D
river version 4.0.0 and will be removed in the next major version
river version 4.0.0 and will be removed in the next major version
(Use `node --trace-warnings ...` to show where the warning was created)
(Use `node --trace-warnings ...` to show where the warning was created)
OUTLINE
TIMELINE
Ln 32, Col 21 Spaces: 4 UTF-8 CRLF {} JavaScript
23°C
Haze
Search
ENG
IN
21:56
17-01-2025

Output:


C:\Windows\System32\cmd.e
Microsoft Windows [Version 10.0.22631.4751]
(c) Microsoft Corporation. All rights reserved.
C:\>curl http://localhost:3000/notes
[{"_id":"678a7c4260c7ec6cc0339cb8","title":"First Note","content":"This is the first note.","__v":0},{"_id":"678a7c4260c7ec6cc0339cba","title":"Second Note","content":"This is the second note.","__v":0}]
C:\>curl http://localhost:3000/notes/678a7c4260c7ec6cc0339cb8
{"_id":"678a7c4260c7ec6cc0339cb8","title":"First Note","content":"This is the first note.","__v":0}
C:\>curl -X PUT -H "Content-Type: application/json" -d "{\"title\":\"Updated Note\",\"content\":\"Updated content.\"}" http://localhost:3000/notes/678a7c4260c7ec6cc0339cb8
{"_id":"678a7c4260c7ec6cc0339cb8","title":"Updated Note","content":"Updated content.","__v":0}
C:\>curl http://localhost:3000/notes/678a7c4260c7ec6cc0339cb8
{"_id":"678a7c4260c7ec6cc0339cb8","title":"Updated Note","content":"Updated content.","__v":0}
C:\>curl http://localhost:3000/notes
[{"_id":"678a7c4260c7ec6cc0339cb8","title":"Updated Note","content":"Updated content.","__v":0},{"_id":"678a7c4260c7ec6cc0339cba","title":"Second Note","content":"This is the second note.","__v":0}]
C:\>curl -X POST -H "Content-Type: application/json" -d "{\"title\":\"Sample Note\",\"content\":\"This is the content.\"}" http://localhost:3000/notes
{"title":"Sample Note","content":"This is the content.","_id":"678a80a76fd8b76a348c1ca2","__v":0}
C:\>curl http://localhost:3000/notes
[{"_id":"678a7c4260c7ec6cc0339cb8","title":"Updated Note","content":"Updated content.","__v":0},{"_id":"678a7c4260c7ec6cc0339cba","title":"Second Note","content":"This is the second note.","__v":0},{"_id":"678a80a76fd8b76a348c1ca2","title":"Sample Note","content":"This is the content.","__v":0}]
C:\>curl -X DELETE http://localhost:3000/notes/678a7c4260c7ec6cc0339cb8
{"message":"Note deleted successfully"}
C:\>curl http://localhost:3000/notes
[{"_id":"678a7c4260c7ec6cc0339cba","title":"Second Note","content":"This is the second note.","__v":0},{"_id":"678a80a76fd8b76a348c1ca2","title":"Sample Note","content":"This is the content.","__v":0}]
C:\>
23°C
Haze
Search
ENG
IN
21:41
17-01-2025

8 b. Aim: In the myNotes application, include APIs based on the requirements provided. i) API should fetch the details of the notes based on a notesID which is provided in the URL. Test URL- http://localhost:3000/notes/7555. ii)API should update the details based on name which is provided in the URL and the data in the request body. iii)API should delete the details based on the name which is provided in the URL.

Description:

API stands for **Application Programming Interface**. It is a set of rules and protocols that allow different software applications to communicate with each other. APIs are commonly used to expose functionality or data to other applications or developers in a structured and secure way.

For example:

- In the context of a web application, APIs enable clients (like a web browser or mobile app) to request data or perform actions on the server.
- The **HTTP methods** (GET, POST, PUT, DELETE, etc.) are often used to interact with APIs.

Source Code:

1. Import Required Modules

```
2.  constexpress=require('express');
3.  constmongoose=require('mongoose');
4.  constbodyParser=require('body-parser');
5.
```

body-parser: Middleware to parse incoming JSON request bodies into JavaScript objects.

2. Set up Express Application

```
constapp=express();
```

3.Middleware Setup

```
app.use(bodyParser.json());
```

UsebodyParser.json() to automatically parse the JSON data from incoming requests.

4.Connect to MongoDB

```
// Connect to MongoDB
mongoose.connect('mongodb://localhost:27017/myNotes', {
    useNewUrlParser:true,
    useUnifiedTopology:true,
}).then(() =>console.log('Connected to MongoDB'))
    .catch(err=>console.error('MongoDB connection failed:', err));
```

- Establish a connection with MongoDB using mongoose.connect().
- Options (useNewUrlParser, useUnifiedTopology, etc.) are used to avoid deprecation warnings.

5.Define the Schema

```
// Define a Mongoose schema and model
constnoteSchema=newmongoose.Schema({
    _id:String, // Use a string ID for notesID
    name:String,
    content:String,
    date:String,
});
```

- A noteSchema is created to define the structure of the documents in the notes collection.
- Each document will have _id, name, content, and date.

6.Create a Model

```
constNote=mongoose.model('Note', noteSchema);
```

The Note model is created using the noteSchema. It provides methods to interact with notes collection.

7.Add Sample data

```
// Add sample data
constaddSampleData=async () => {
    try {
        constexistingNotes=awaitNote.countDocuments();
        if (existingNotes===0) {
            awaitNote.insertMany([
                {
                    _id:"7555",
                    name:"Mathan",
                    content:"This is Mathan's sample note.",
                    date:"2025-01-17",
                },

            ]);
            console.log('Sample data added to the database');
        } else {
            console.log('Sample data already exists');
        }
    } catch (error) {
        console.error('Error adding sample data:', error.message);
    }
};

addSampleData();
```

8.Fetch a Note by notesID

```
// (i) Fetch details of a note by notesID
app.get('/notes/:notesID', async (req, res) => {
    try {
        constnote=awaitNote.findById(req.params.notesID);
        if (!note) returnres.status(404).send('Note not found');
        res.send(note);
    } catch (error) {
        res.status(500).send('Error fetching note: '+error.message);
    }
});
```

- GET /notes/:notesID: Fetches a note by its unique notesID.
- req.params.notesID extracts the ID from the URL.
- If a matching note is found, it is returned as a JSON response. Otherwise, an error message is returned.

9.Update a Note by name

```
// (ii) Update details of a note by name
app.put('/notes/:name', async (req, res) => {
    try {
        constupdatedNote=awaitNote.findOneAndUpdate(
            { name:req.params.name },
            req.body, // Updates from the request body
            { new:true } // Return the updated document
        );
        if (!updatedNote) returnres.status(404).send('Note not found');
        res.send(updatedNote);
    } catch (error) {
        res.status(500).send('Error updating note: '+error.message);
    }
});
```

- PUT /notes/:name: Updates a note by its name.
- req.params.name extracts the name from the URL.
- req.body contains the updated data.
- If a matching note is found, it is updated and returned in the response. Otherwise, an error message is returned.

10.Delete a Note by name

```
// (iii) Delete a note by name
app.delete('/notes/:name', async (req, res) => {
    try {
        constdeletedNote=awaitNote.findOneAndDelete({ name:req.params.name });
        if (!deletedNote) returnres.status(404).send('Note not found');
        res.send('Note deleted successfully');
    } catch (error) {
        res.status(500).send('Error deleting note: '+error.message);
    }
});
```

- DELETE /notes/:name: Deletes a note by its name.

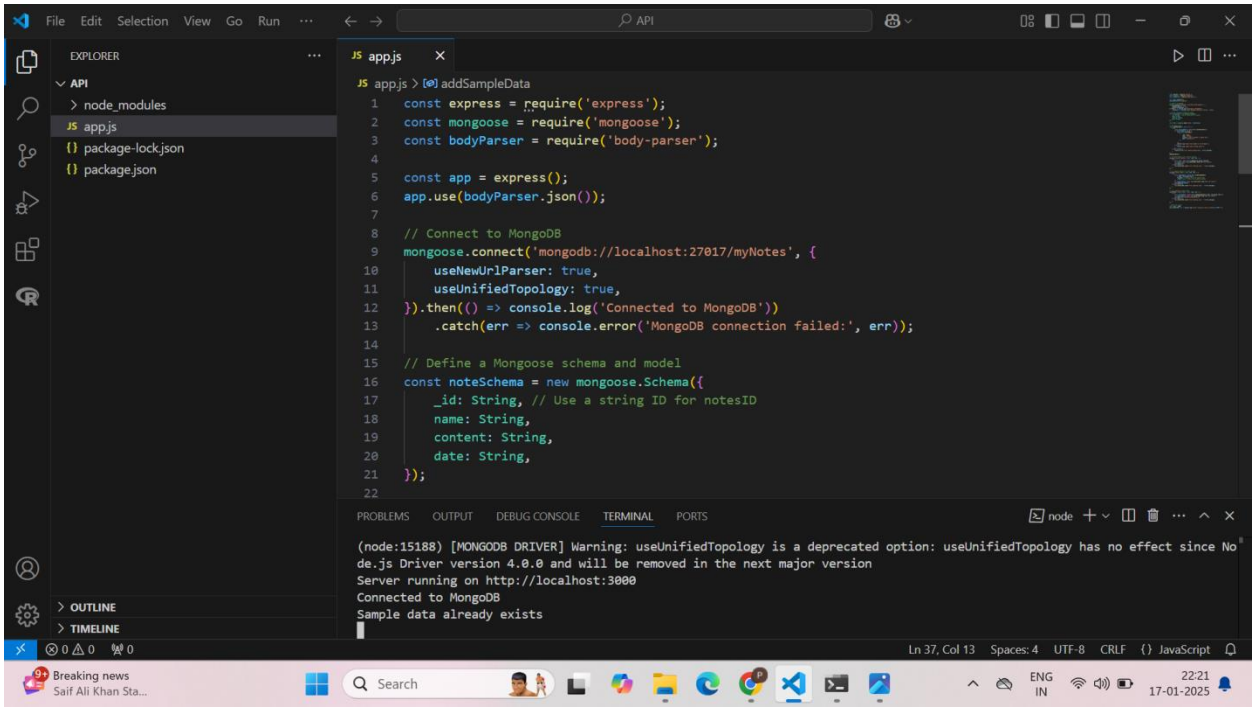
- req.params.name extracts the name from the URL.
- If a matching note is found, it is deleted and a success message is returned. Otherwise, an error message is returned.

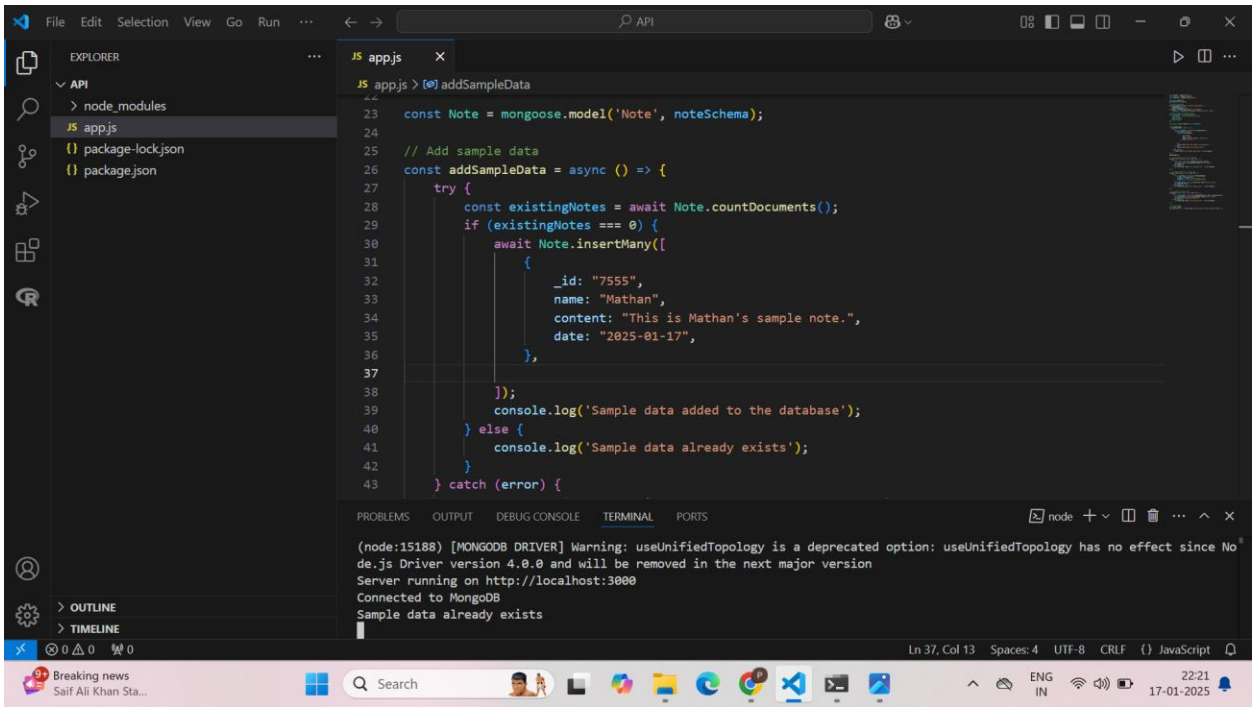
11. Start the server

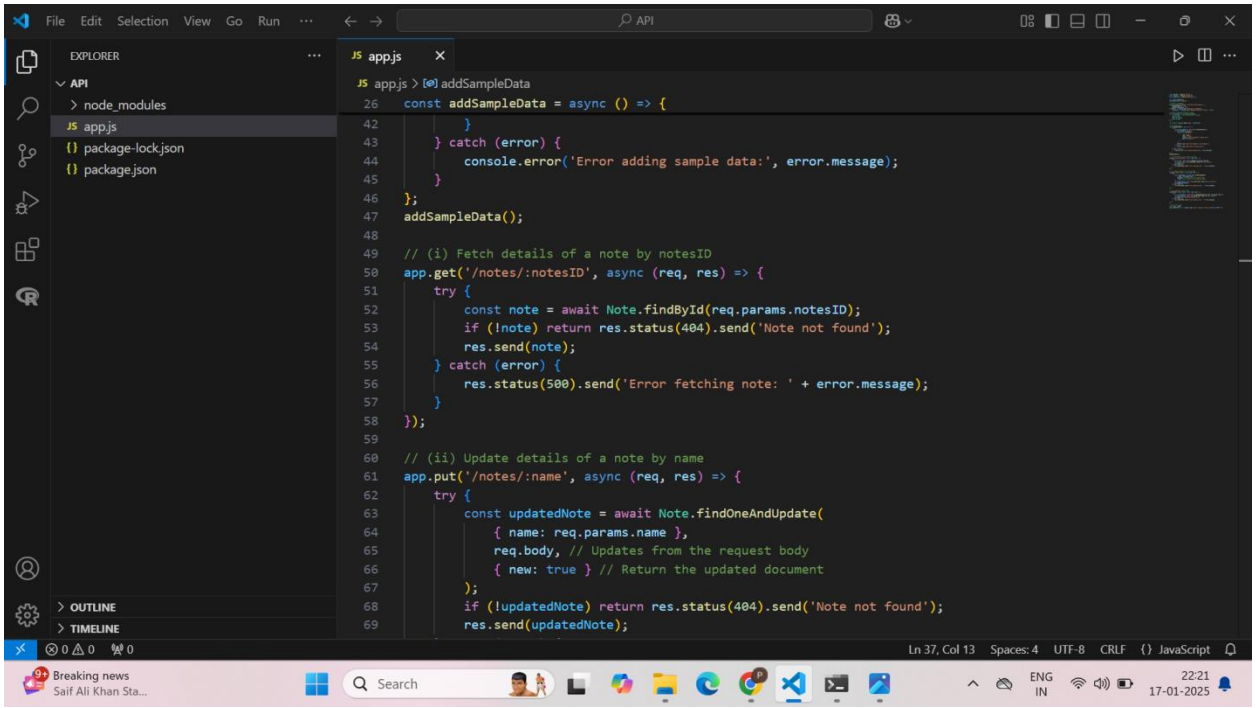
```
// Start the server
constPORT=3000;
app.listen(PORT, () =>console.log(`Server running on http://localhost:${PORT}`));
```

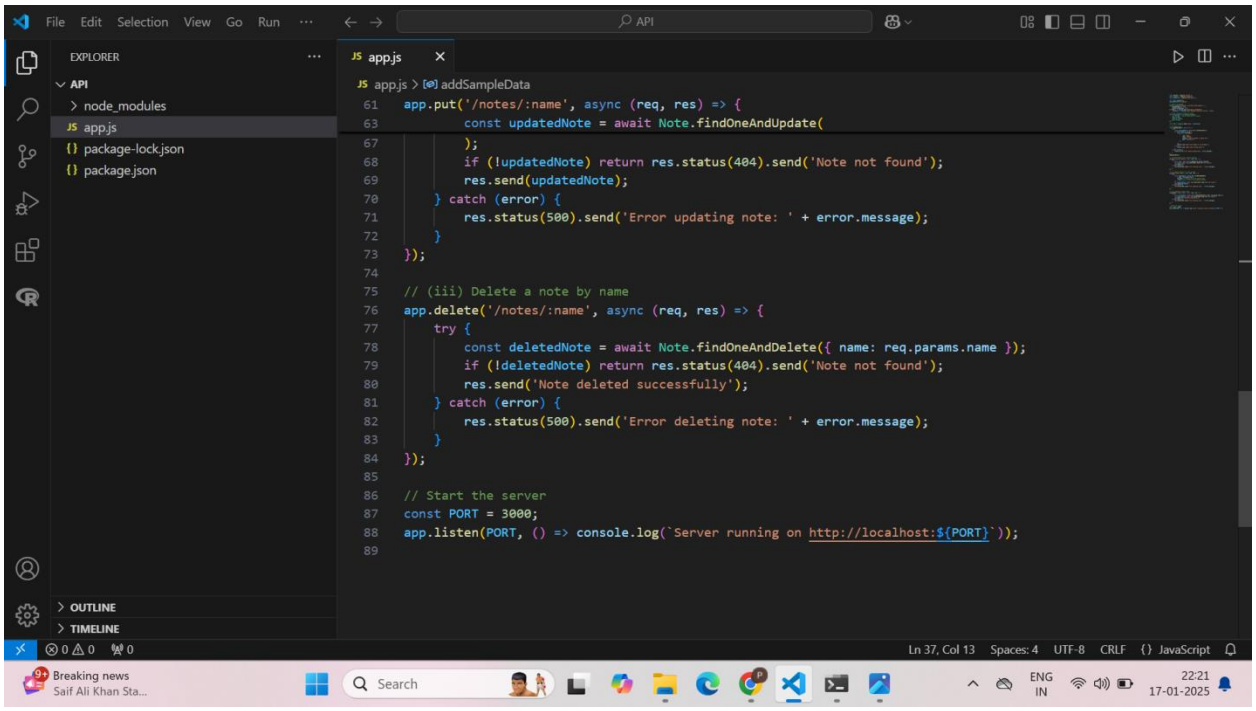
The server starts listening on the specified port (3000) and prints a message to the console.

Code:

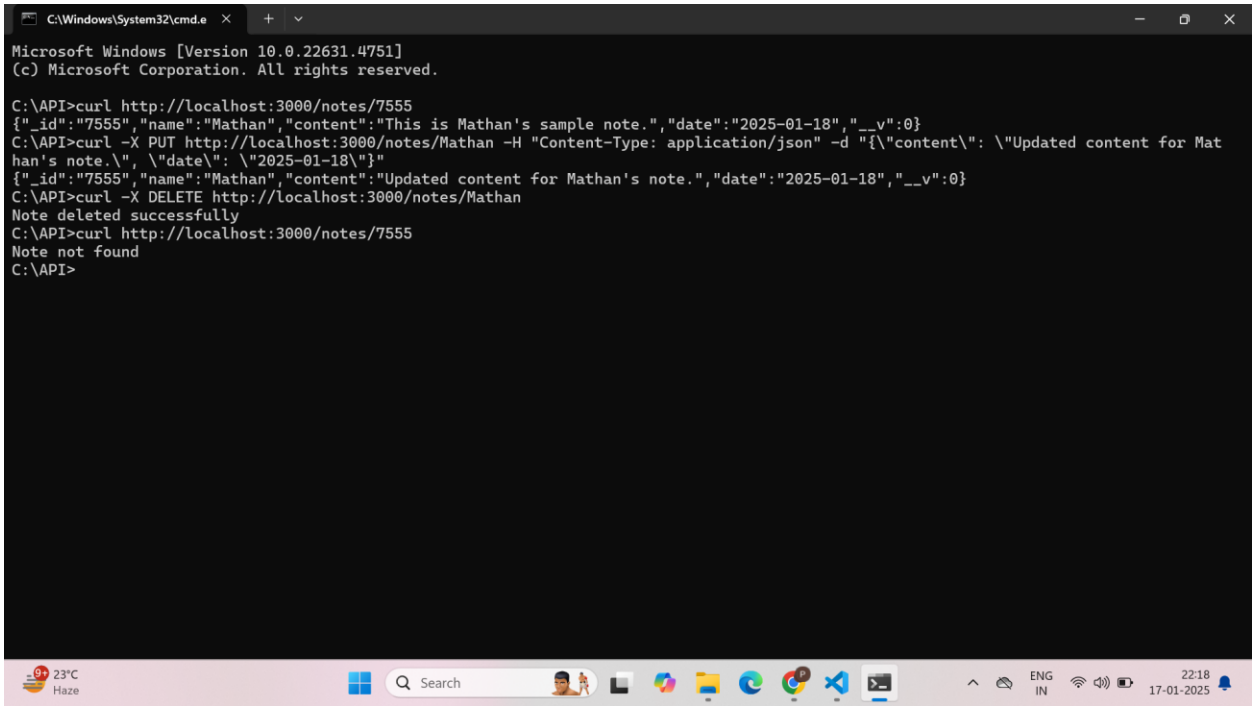



File Edit Selection View Go Run
API
EXPLORER
API
node_modules
app.js
package-lock.json
package.json
app.js
app.js > addSampleData
const Note = mongoose.model('Note', noteSchema);
// Add sample data
const addSampleData = async () => {
try {
const existingNotes = await Note.countDocuments();
if (existingNotes === 0) {
await Note.insertMany([
{
_id: "7555",
name: "Mathan",
content: "This is Mathan's sample note.",
date: "2025-01-17",
},
]);
console.log('Sample data added to the database');
} else {
console.log('Sample data already exists');
}
} catch (error) {
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
node
(node:15188) [MONGODB DRIVER] Warning: useUnifiedTopology is a deprecated option: useUnifiedTopology has no effect since Node.js Driver version 4.0.0 and will be removed in the next major version
Server running on http://localhost:3000
Connected to MongoDB
Sample data already exists
OUTLINE
TIMELINE
Ln 37, Col 13 Spaces: 4 UTF-8 CRLF {} JavaScript
Breaking news
Saif Ali Khan Sta...
Search
ENG
IN
22:21
17-01-2025


File Edit Selection View Go Run
API
EXPLORER
API
node_modules
app.js
package-lock.json
package.json
app.js
app.js > addSampleData
const addSampleData = async () => {
}
} catch (error) {
console.error('Error adding sample data:', error.message);
}
};
addSampleData();
// (i) Fetch details of a note by notesID
app.get('/notes/:notesID', async (req, res) => {
try {
const note = await Note.findById(req.params.notesID);
if (!note) return res.status(404).send('Note not found');
res.send(note);
} catch (error) {
res.status(500).send('Error fetching note: ' + error.message);
}
});
// (ii) Update details of a note by name
app.put('/notes/:name', async (req, res) => {
try {
const updatedNote = await Note.findOneAndUpdate(
{ name: req.params.name },
req.body, // Updates from the request body
{ new: true } // Return the updated document
);
if (!updatedNote) return res.status(404).send('Note not found');
res.send(updatedNote);
OUTLINE
TIMELINE
Ln 37, Col 13 Spaces: 4 UTF-8 CRLF {} JavaScript
Breaking news
Saif Ali Khan Sta...
Search
ENG
IN
22:21
17-01-2025


File Edit Selection View Go Run
API
EXPLORER
app.js
API
node_modules
app.js
package-lock.json
package.json
app.js > addSampleData
app.put('/notes/:name', async (req, res) => {
const updatedNote = await Note.findOneAndUpdate(
);
if (!updatedNote) return res.status(404).send('Note not found');
res.send(updatedNote);
} catch (error) {
res.status(500).send('Error updating note: ' + error.message);
}
});
// (iii) Delete a note by name
app.delete('/notes/:name', async (req, res) => {
try {
const deletedNote = await Note.findOneAndDelete({ name: req.params.name });
if (!deletedNote) return res.status(404).send('Note not found');
res.send('Note deleted successfully');
} catch (error) {
res.status(500).send('Error deleting note: ' + error.message);
}
});
// Start the server
const PORT = 3000;
app.listen(PORT, () => console.log(`Server running on http://localhost:${PORT}`));
OUTLINE
TIMELINE
Ln 37, Col 13 Spaces: 4 UTF-8 CRLF {} JavaScript
Breaking news
Saif Ali Khan Sta...
Search
ENG
IN
22:21
17-01-2025

Output:


C:\Windows\System32\cmd.e
Microsoft Windows [Version 10.0.22631.4751]
(c) Microsoft Corporation. All rights reserved.
C:\API>curl http://localhost:3000/notes/7555
{"_id":"7555","name":"Mathan","content":"This is Mathan's sample note.","date":"2025-01-18","__v":0}
C:\API>curl -X PUT http://localhost:3000/notes/Mathan -H "Content-Type: application/json" -d "{\"content\": \"Updated content for Mat
han's note.\", \"date\": \"2025-01-18\"}"
{"_id":"7555","name":"Mathan","content":"Updated content for Mathan's note.","date":"2025-01-18","__v":0}
C:\API>curl -X DELETE http://localhost:3000/notes/Mathan
Note deleted successfully
C:\API>curl http://localhost:3000/notes/7555
Note not found
C:\API>
23°C
Haze
Search
ENG
IN
22:18
17-01-2025

8 c. Aim: Write a program to explain session management using cookies

Description:

What are Cookies?

Cookies are small pieces of data (in the form of text files) that are stored on a user's device by a web browser when they visit a website. They help websites remember information about the user, such as their preferences, login status, or browsing behavior.

Session Management Using Cookies

Session management using cookies is a way to maintain a user's interaction with a website by storing a small piece of data (a cookie) in their browser. This cookie typically contains a **Session ID** that uniquely identifies the user's session on the server.

Prerequisite:

npm install express cookie-parser

Source Code for Session Management Using Cookies

1. **Importing Required Libraries**

```
constexpress=require('express');
constcookieParser=require('cookie-parser');
```

- **cookie-parser:** This is a middleware used to parse cookies sent by the client (browser) to the server. It helps access cookies from the req.cookies object.

2. Setting Up the Express Application

```
constapp=express();
constPORT=3000;
```

3. Middleware Configuration

```
// Middleware to parse cookies
app.use(cookieParser());

// Middleware to parse incoming request body
app.use(express.urlencoded({ extended:true }));
```

- **cookieParser()**: This middleware is used to parse the cookies in the incoming HTTP requests, making it easy to access cookies using req.cookies.
- **express.urlencoded()**: This middleware is used to parse incoming requests with URL-encoded data (like form submissions). The option { extended: true } allows for complex objects and arrays to be encoded in the URL-encoded format.

4. Login Route (/login)

```
// Route to set a session cookie
app.get('/login', (req, res) => {
    constusername=req.query.username||"Guest";
    res.cookie('username', username, { maxAge:600000, httpOnly:true }); // Cookie valid for 10 minutes
    res.send(`Welcome, ${username}! Your session has started.`);
});
```

/login Route: This route is triggered when a user accesses the /login URL.

- **req.query.username || "Guest"**: The username is retrieved from the query string (e.g., ?username=JohnDoe). If no username is provided, it defaults to "Guest".
- **res.cookie('username', username, { maxAge: 600000, httpOnly: true })**:
    - This sets a cookie named username with the value of the username variable.

- **maxAge: 600000**: This makes the cookie expire after 10 minutes (600,000 milliseconds).
- **httpOnly: true**: This ensures that the cookie is not accessible via JavaScript in the browser (for security reasons, to prevent XSS attacks).

- **res.send(...)**: Sends a response to the client saying, "Welcome" and confirming the session has started.

5. Dashboard Route (/dashboard)

```
// Route to access session cookie
app.get('/dashboard', (req, res) => {
    constusername=req.cookies.username;
    if (username) {
        res.send(`Hello, ${username}! Welcome back to your dashboard.`);
    } else {
        res.send('No active session. Please log in first.');
    }
});
```

/dashboard Route: This route is triggered when the user accesses the /dashboard URL.

- **req.cookies.username**: The server checks for the username cookie sent with the incoming request. If the cookie is found, the session is considered active.
- **If the username cookie exists**:
  - It sends a personalized response welcoming the user back to their dashboard.
- **If the username cookie does not exist**:
  - It sends a message telling the user that there is no active session and suggests logging in first.

6. Logout Route (/logout)

```
// Route to clear session cookie
app.get('/logout', (req, res) => {
    res.clearCookie('username');
    res.send('You have been logged out successfully.');
});
```

/logout Route: This route is triggered when the user accesses the /logout URL.

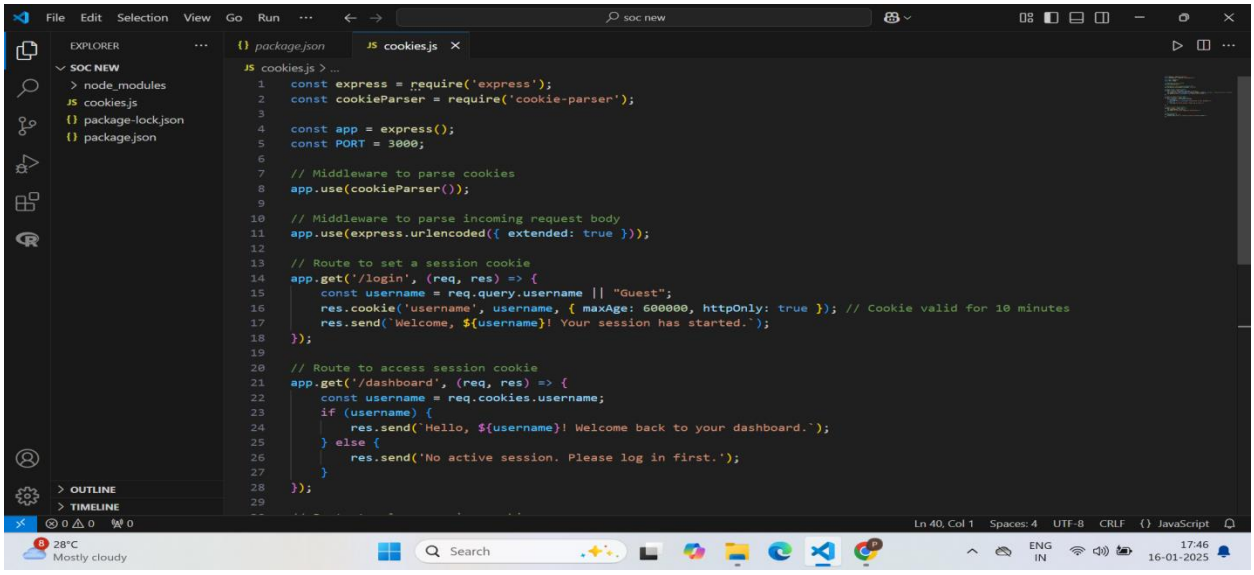
- **res.clearCookie('username')**: This clears the username cookie, effectively logging the user out and terminating the session.
- **res.send('You have been logged out successfully.')**: Sends a message confirming the user has been logged out.

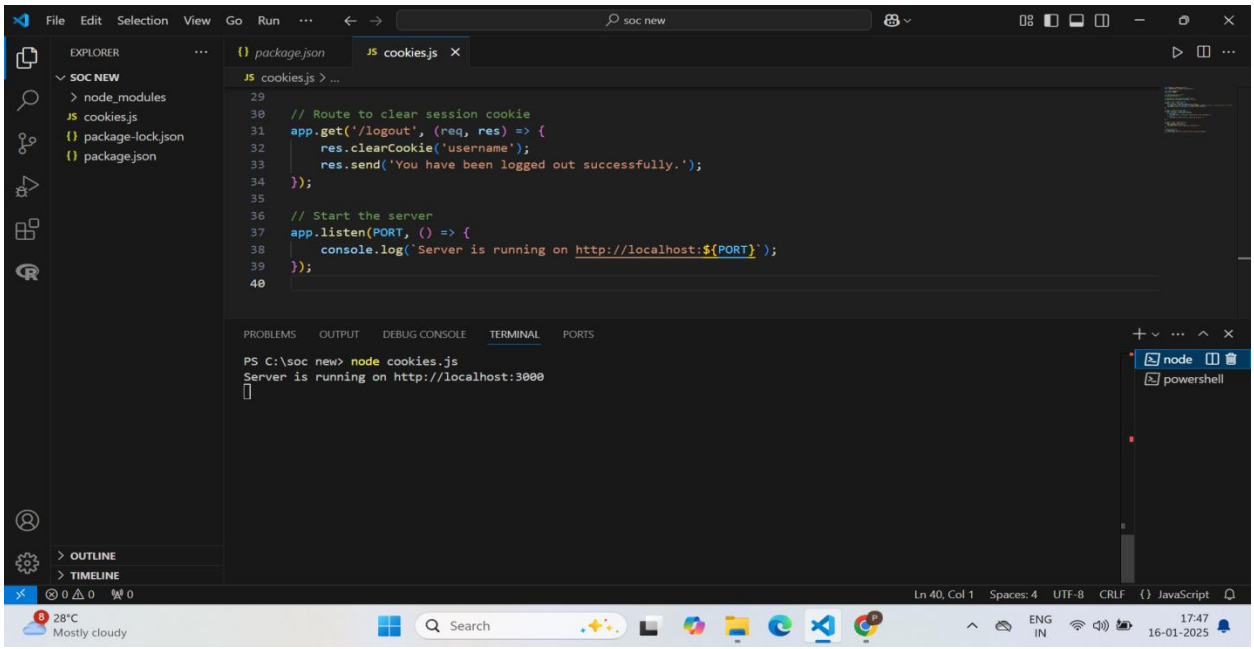
7. Starting the Server

```
// Start the server
app.listen(PORT, () => {
    console.log(`Server is running on http://localhost:${PORT}`);
});
```

app.listen(PORT, callback): This starts the Express server on the specified port (in this case, port 3000).

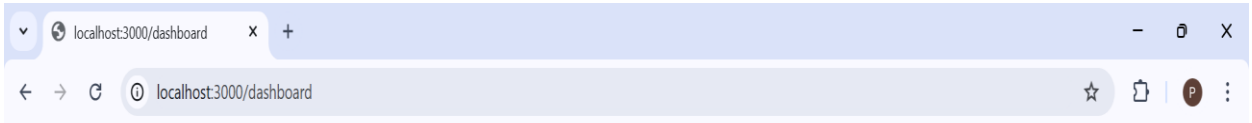
Source Code:



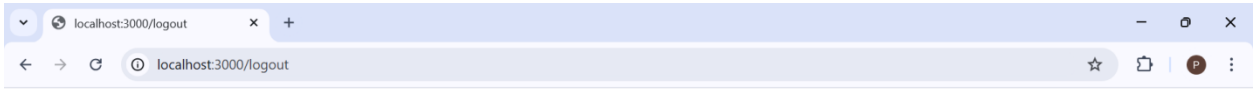


Output:

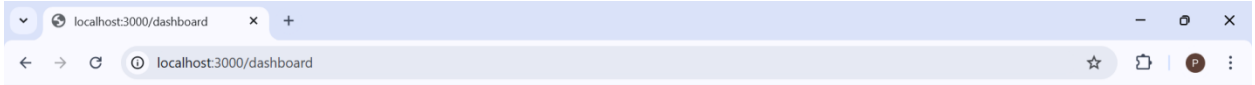
Welcome, rupa! Your session has started.



Hello, rupa! Welcome back to your dashboard.



You have been logged out successfully.



No active session. Please log in first.

8 d. Aim: Write a program to explain session management using sessions.

Description:

What are Sessions?

Sessions are server-side mechanisms used to store user-specific data for the duration of a user's interaction with the website (or until the session expires). Unlike cookies, sessions store the data on the server and only use a small identifier (usually stored in a cookie) to track the user.

Session Management Using Sessions

Session management using sessions involves storing user-specific data on the server, with the browser maintaining a reference (Session ID) to that data. This is commonly used for secure and efficient tracking of user interactions on a website.

Prerequisite:

npm install express express-session

Source Code for Session Management Using Sessions

1. Importing Required Libraries

```
constexpress=require('express');
constsession=require('express-session');
```

express-session: Middleware for managing user sessions.

2. Setting Up the Express Application

```
constapp=express();
constPORT=3000;
```

app: This initializes an instance of the Express application.

PORT: The port on which the server will listen for incoming requests. In this case, it's set to 3000.

3. Add Middleware for Parsing Data

```
// Middleware for parsing request bodies
app.use(express.urlencoded({ extended:true }));
```

This allows the app to parse data sent in requests (e.g., form submissions or query strings).

4. Configure the Session Middleware

```
// Configure session middleware
app.use(
  session({
    secret:'your-secret-key',
    resave:false,
    saveUninitialized:false,
    cookie: { maxAge:60000 },
  })
);
```

- **secret**: A unique string used to secure the session (replace 'your-secret-key' with a strong value).
- **resave: false**: Prevents saving unchanged sessions.
- **saveUninitialized: false**: Only creates a session if data is added to it.
- **cookie.maxAge: 60000**: Sets the session expiration time to 1 minute (60,000 milliseconds).

5. Home Route

```
// Home route
app.get('/', (req, res) => {
  if (req.session.username) {
    res.send(`Welcome back, ${req.session.username}! Type "/logout" to log out.`);
  } else {
    res.send('Welcome to the app! Type "/login?username=YourName" to log in.');
  }
});
```

- If the user **has logged in** (i.e., req.session.username exists), it greets them:

  Example: Welcome back, John! Type "/logout" to log out.
- If the user **has not logged in**, it suggests logging in:

  Example: Welcome! Type "/login?username=YourName" to log in.

6. Login Route

```
// Login route
app.get('/login', (req, res) => {
  constusername=req.query.username;
  if (username) {
    req.session.username=username;
    res.send(`Logged in as ${username}. Go to "/" to see your session.`);
  } else {
    res.send('Please provide a username using "/login?username=YourName".');
  }
});
```

Check for username in the query string:

- For /login?username=John, the username is "John".

If a username is provided:

- Save it in the session (req.session.username = username).
- Respond with: Logged in as John. Go to "/" to see your session.

If no username is provided:

- Respond with: Please provide a username using "/login?username=YourName".

7. Logout Route

```
// Logout route
app.get('/logout', (req, res) => {
  req.session.destroy((err) => {
    if (err) {
      returnres.status(500).send('Error logging out.');
    }
    res.send('Logged out successfully. Go to "/" to log in again.');
  });
});
```

- **Destroy the session** using req.session.destroy():

  Deletes the session data (e.g., username).

- If there's an error**:**

  Send: Error logging out.

- If successful:

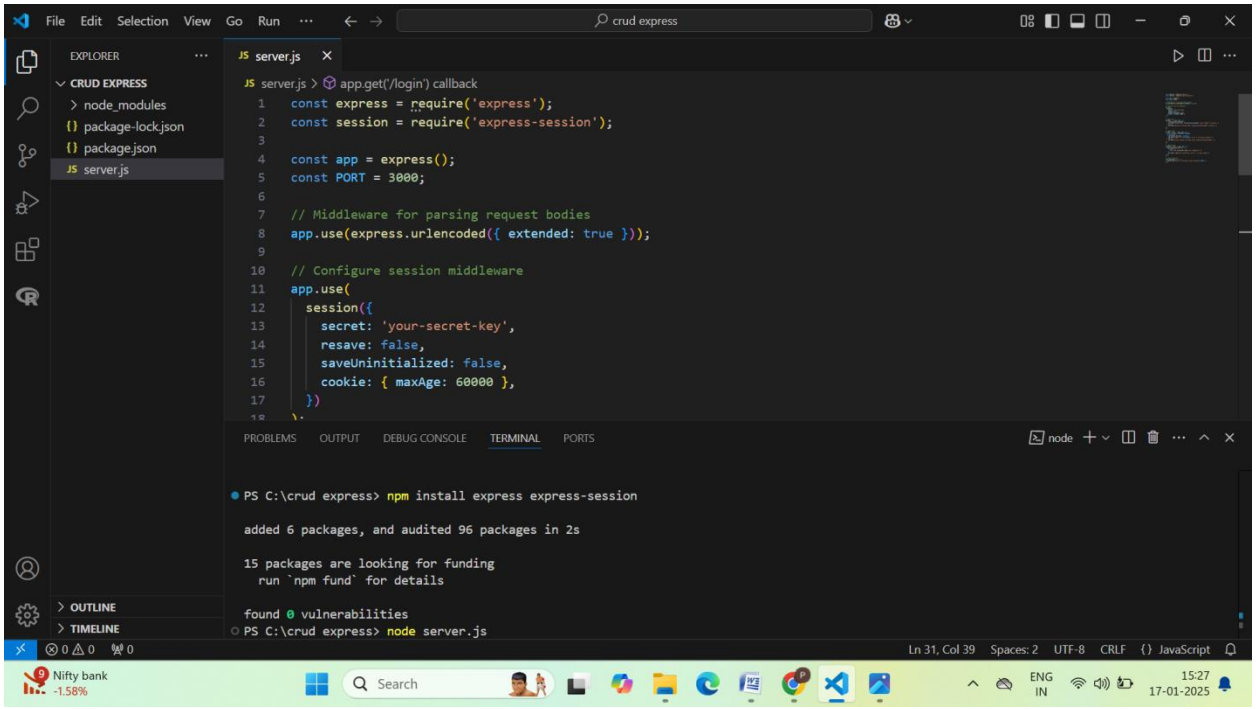
  Send: Logged out successfully. Go to "/" to log in again.

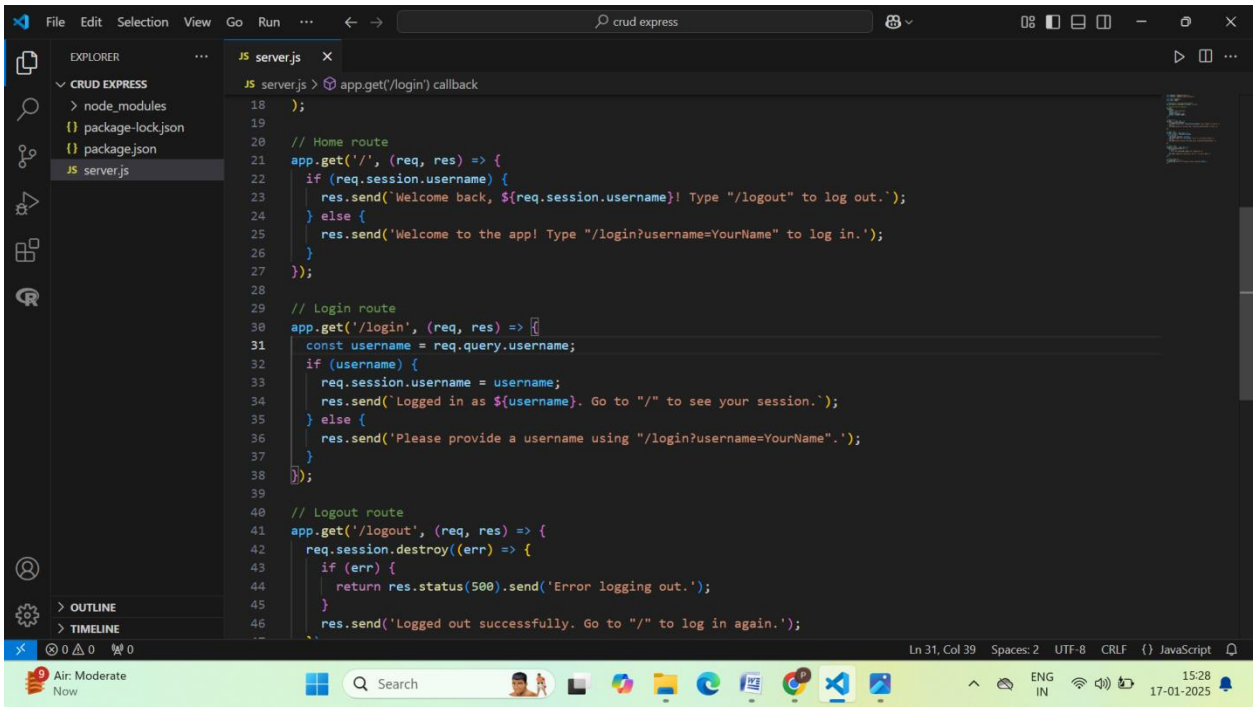
8. Starting the Server

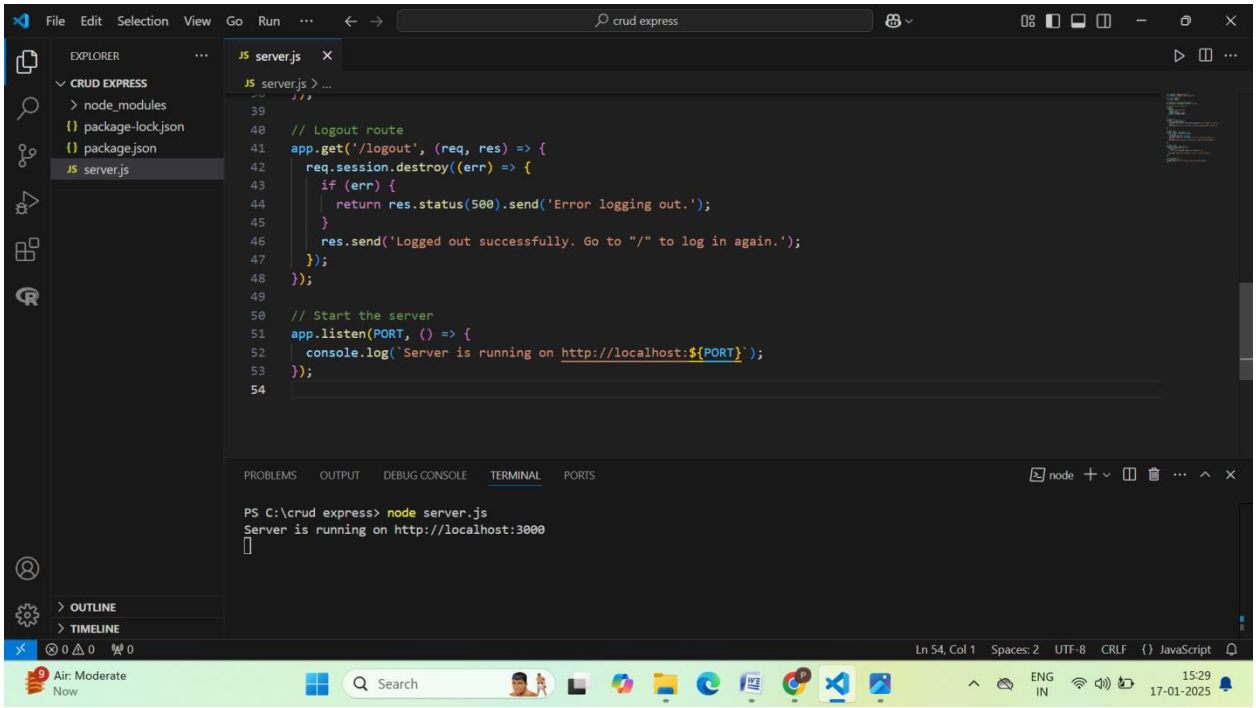
```
// Start the server
app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```

app.listen(PORT, callback): This starts the Express server on the specified port (in this case, port 3000).

Source Code:




File Edit Selection View Go Run
crud express
EXPLORER
CRUD EXPRESS
node_modules
package-lock.json
package.json
server.js
server.js
server.js > app.get('/login') callback
18);
19
20 // Home route
21 app.get('/', (req, res) => {
22 if (req.session.username) {
23 res.send(`Welcome back, ${req.session.username}! Type "/logout" to log out.`);
24 } else {
25 res.send('Welcome to the app! Type "/login?username=YourName" to log in.');
26 }
27 });
28
29 // Login route
30 app.get('/login', (req, res) => {
31 const username = req.query.username;
32 if (username) {
33 req.session.username = username;
34 res.send(`Logged in as ${username}. Go to "/" to see your session.`);
35 } else {
36 res.send('Please provide a username using "/login?username=YourName".');
37 }
38 });
39
40 // Logout route
41 app.get('/logout', (req, res) => {
42 req.session.destroy((err) => {
43 if (err) {
44 return res.status(500).send('Error logging out.');
45 }
46 res.send('Logged out successfully. Go to "/" to log in again.');
OUTLINE
TIMELINE
Ln 31, Col 39 Spaces: 2 UTF-8 CRLF JavaScript
Air: Moderate
Now
Search
ENG
IN
15:28
17-01-2025


File Edit Selection View Go Run
crud express
EXPLORER
CRUD EXPRESS
node_modules
package-lock.json
package.json
server.js
server.js > ...
40 // Logout route
41 app.get('/logout', (req, res) => {
42 req.session.destroy((err) => {
43 if (err) {
44 return res.status(500).send('Error logging out.');
45 }
46 res.send('Logged out successfully. Go to "/" to log in again.');
47 });
48 });
49
50 // Start the server
51 app.listen(PORT, () => {
52 console.log(`Server is running on http://localhost:${PORT}`);
53 });
54
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
node
PS C:\crud express> node server.js
Server is running on http://localhost:3000
OUTLINE
TIMELINE
Ln 54, Col 1 Spaces: 2 UTF-8 CRLF JavaScript
Air: Moderate
Now
Search
ENG
IN
15:29
17-01-2025

Output:

Welcome to the app! Type "/login?username=YourName" to log in.

Logged in as harshi. Go to "/" to see your session.

Welcome back, harshi! Type "/logout" to log out.

Logged out successfully. Go to "/" to log in again.

8 e. Aim: Implement security features in myNotes application.

Description:

What is the importance of security?

- **Protecting user data:**User's notes likely contain sensitive information. Security measures safeguard this data from unauthorized access, modification, or deletion.

- **Preventing Attacks:**Web applications are vulnerable to various attacks like cross-site scripting (XSS), clickjacking, and cross-site request forgery (CSRF). Security features help mitigate these risks.

- **Maintaining trust:**A secure application builds user trust by ensuring their data is protected and transactions are legitimate.

The aim of this module is to understand the importance of security in web application and learn how to implement security features using the Helmet middleware in an express.js application. Specifically, we will implement security features in the "myNotes" application to protect it from common web vulnerabilities.

Express.js is a popular web application framework for Node.js that provides a minimal and flexible foundation for building robust web applications. Helmet is a middleware package for Express.js that helps secure express applications by setting various HTTP headers related to security.

What is Middleware?

Middleware is software or code that acts as a bridge or intermediary between different parts of an application, such as the client and server, or between different layers in a system. It is commonly used in frameworks like Express.js (Node.js) or Django (Python) to handle tasks before reaching the main application logic

Types of Middleware :

1. **Application level middleware**–Applicationlevel middleware is designed to enhance and support the functionality of software applications. It sits between the application and the operating system, providing various services that improve performance, security, and communication.

2. **Router level middleware –**Routerlevel middleware functions at the routing layer of your application, and it's responsible for handling and processing requests before they reach the specific route handlers. This type of middleware is often used to perform tasks like validation, authentication, logging, and more specifically at the route level.

3. **Built-in middleware –**Built-in middleware refers to the middleware that comes bundled with a framework or platform, providing out-of-the-box functionality for common tasks. In the context of Express.js, there are several built-in middleware functions that you can use to handle various aspects of request and response processing.

4. **Third party middleware –**Thirdparty middleware refers to middleware components that are not built into the core framework but are instead provided by external libraries or modules. These middleware components

offer additional functionality that can be easily integrated into your application.

5. **Error handling middleware** –Errorhandling middleware in express.jsis crucial for managing errors that occur during the request-response cycle. It allows you to catch and handle errors gracefully, providing meaningful error messages to the client and logging errors for debugging purposes.

Helmet Middleware :

Helmet is a popular and powerful node.js middleware for express.js applications that simplifies adding essential HTTP headers to enhance security. By incorporating Helmet, you configure several security-related headers automatically, reducing the need for manual configuration.

Key security features with Helmet:

- **Content Security Policy (CSP):**It restricts resources (scripts, styles, images) that can be loaded from external sources, preventing XSS and other attacks.
- **X-XSS-Protection:** It mitigates XSS vulnerabilities by enabling browser-side defenses.
- **X-Frame-Options:** It prevents your application from being rendered within a frame or iframe, reducing clickjacking risks.
- **Strict-Transport-Security (HSTS):** It enforces HTTPS communication, protecting against insecure connections.
- **X-Permitted-Cross-Domain-Policies:** It disables certain cross-origin requests if needed.

- **Referrer-Policy:** It controls how much referrer information is included in outgoing requests.
- **Public-Key-Pins (HPKP):** (Optional) It pins public keys to prevent certificate authority (CA) substitution attacks (advanced).

Implementing Helmet in myNotes:

Installation:

```
npm install helmet
```

Require Helmet in your express app:

```
constexpress=require('express');
consthelmet=require('helmet');

const app = express();

app.use(helmet());//Apply Helmet middleware
```

Source Code:

```
constexpress =require('express');
consthelmet =require('helmet');
constapp = express();

//Apply Helmet middleware to secure the application
app.use(helmet());
//…Rest of the application code
//Start the server
app.listen(3000,() =>{
console.log('Server started on port 3000');
});
```

Output:

By applying the Helmet middleware to the ‘myNotes’ application, several security features will be automatically enabled. These features include setting appropriate HTTP headers to prevent various attacks, such as XSS, MIME sniffing, and clickjacking.

The output of running the application will be running server listening on port 3000. However, the output related to security will be observed in the network traffic and interactions with the application.

With the Helmet middleware enabled, the application will have improved security by default, reducing the risk of common web vulnerabilities.

It’s important to note that security is an ongoing process, and while implementing the Helmet middleware provides a good starting point, it’s crucial to stay updated with the latest security practices and regularly review and enhance the security measures in your application.

The provided code is a basic example to illustrate the implementation of Helmet middleware for security purposes. In a real-world scenario, additional security practices and considerations, such as input validation, authentication, and authorization, should be implemented to ensure a robust and secure application.

Typescript

9a:

Course Name: Typescript

Module Name: Basics of TypeScript On the page, display the price of the mobile-based in three different colors. Instead of using the number in our code, represent them by string values like GoldPlatinum, PinkGold, SilverTitanium.

Installation of Node: (The commands for WINDOWS):

Download and install fnm:

winget install Schniz.fnm

Download and install Node.js:

fnm install 22

Verify the Node.js version:

node -v # Should print "v22.13.1".

Verify npm version:

npm -v # Should print "10.9.2".

Installation of **typescript**:

Command to install:

npm install -g typescript

Verify the installation:

tsv -v

< image of verified in cmd>

The <u>Basics of Typescript:</u>

What is Typescript:

TypeScript is a strongly typed, object-oriented, compiled language developed and maintained by Microsoft. It is a superset of JavaScript, meaning that any valid JavaScript code is also valid TypeScript code.

How to configure source and destination files:

Create a tsconfig file by this command:

tsc -init

This command will create a tsconfig file in your root directory.

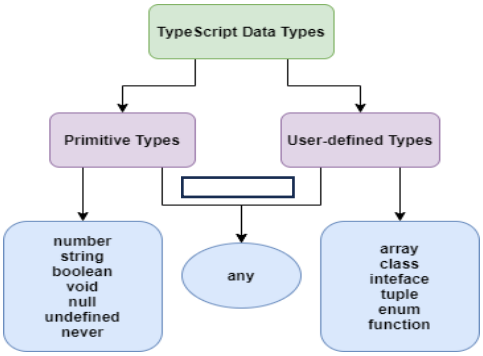
In the tsconfig file:

- Make changes in module section, in module section change: “rootDir” for replacing your **typescript** source file.
- And in “Emit” section change “outDir” for replacing your generated **javascript** file.

```jsonc
tsconfig.json > {} compilerOptions > outDir
  "compilerOptions": {
    // "jsxFactory": "",                        /* Specify the JSX factory function used when targeting React JSX emit, e.g. 'React.createElement' or 'h
    // "jsxFragmentFactory": "",                /* Specify the JSX Fragment reference used for fragments when targeting React JSX emit e.g. 'React.Fragm
    // "jsxImportSource": "",                   /* Specify module specifier used to import the JSX factory functions when using 'jsx: react-jsx*'. */
    // "reactNamespace": "",                    /* Specify the object invoked for 'createElement'. This only applies when targeting 'react' JSX emit. */
    // "noLib": true,                           /* Disable including any library files, including the default lib.d.ts. */
    // "useDefineForClassFields": true,         /* Emit ECMAScript-standard-compliant class fields. */
    // "moduleDetection": "auto",               /* Control what method is used to detect module-format JS files. */

    /* Modules */
    "module": "commonjs",                       /* Specify what module code is generated. */
    "rootDir": "./src",                         /* Specify the root folder within your source files. */
    // "moduleResolution": "node10",            /* Specify how TypeScript looks up a file from a given module specifier. */
    // "baseUrl": "./",                         /* Specify the base directory to resolve non-relative module names. */
    // "paths": {},                             /* Specify a set of entries that re-map imports to additional lookup locations. */
    // "rootDirs": [],                          /* Allow multiple folders to be treated as one when resolving modules. */
```

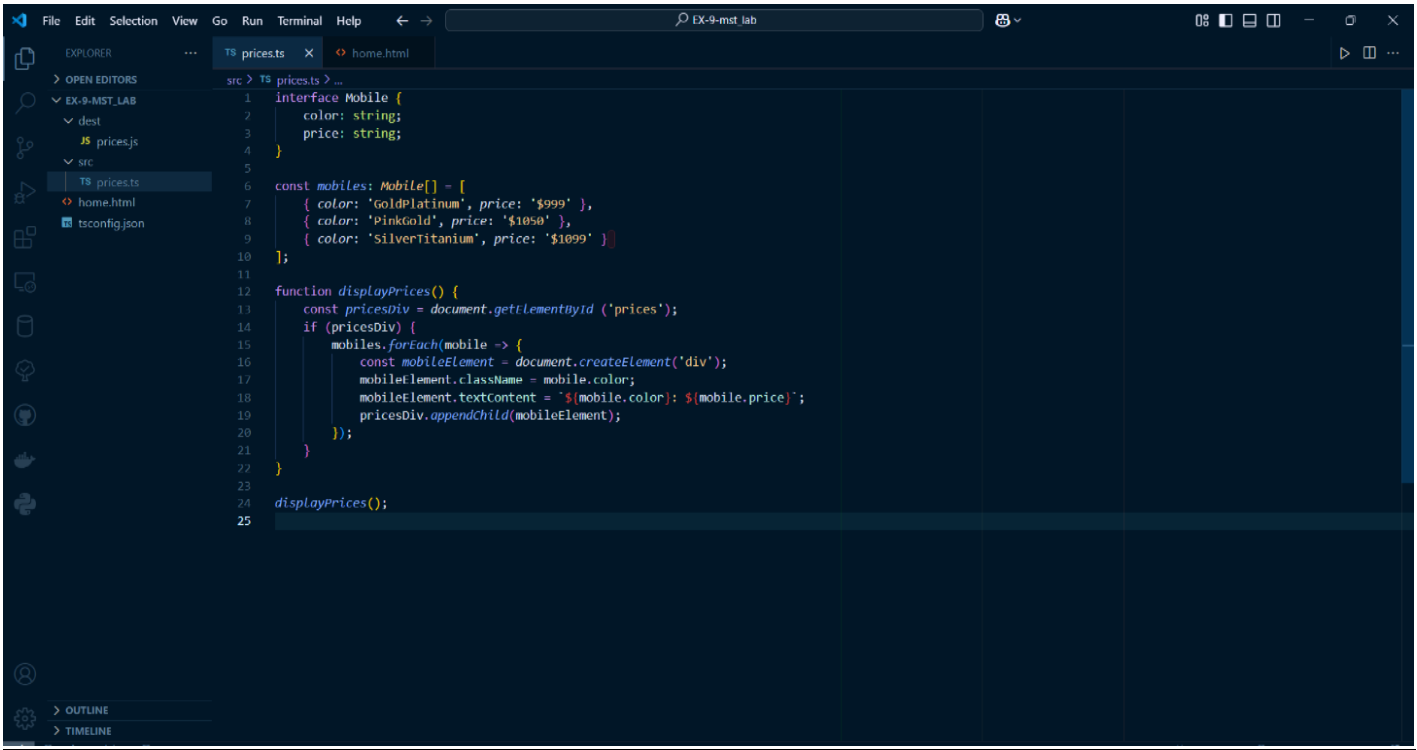
```jsonc
    /* Emit */
    // "declaration": true,                     /* Generate .d.ts files from TypeScript and JavaScript files in your project. */
    // "declarationMap": true,                  /* Create sourcemaps for d.ts files. */
    // "emitDeclarationOnly": true,             /* Only output d.ts files and not JavaScript files. */
    // "sourceMap": true,                       /* Create source map files for emitted JavaScript files. */
    // "inlineSourceMap": true,                 /* Include sourcemap files inside the emitted JavaScript. */
    // "noEmit": true,                          /* Disable emitting files from a compilation. */
    // "outFile": "./",                         /* Specify a file that bundles all outputs into one JavaScript file. If 'declaration' is true, also desi
    "outDir": "./dest",                         /* Specify an output folder for all emitted files. */
    // "removeComments": true,                  /* Disable emitting comments. */
    // "importHelpers": true,                   /* Allow importing helper functions from tslib once per project, instead of including them per-file. */
    // "downlevelIteration": true,              /* Emit more compliant, but verbose and less performant JavaScript for iteration. */
```

Data types in Typescript:



Display the price of the mobile-based in three different colors. Instead of using the number in our code, represent them by string values like GoldPlatinum, PinkGold, SilverTitanium.

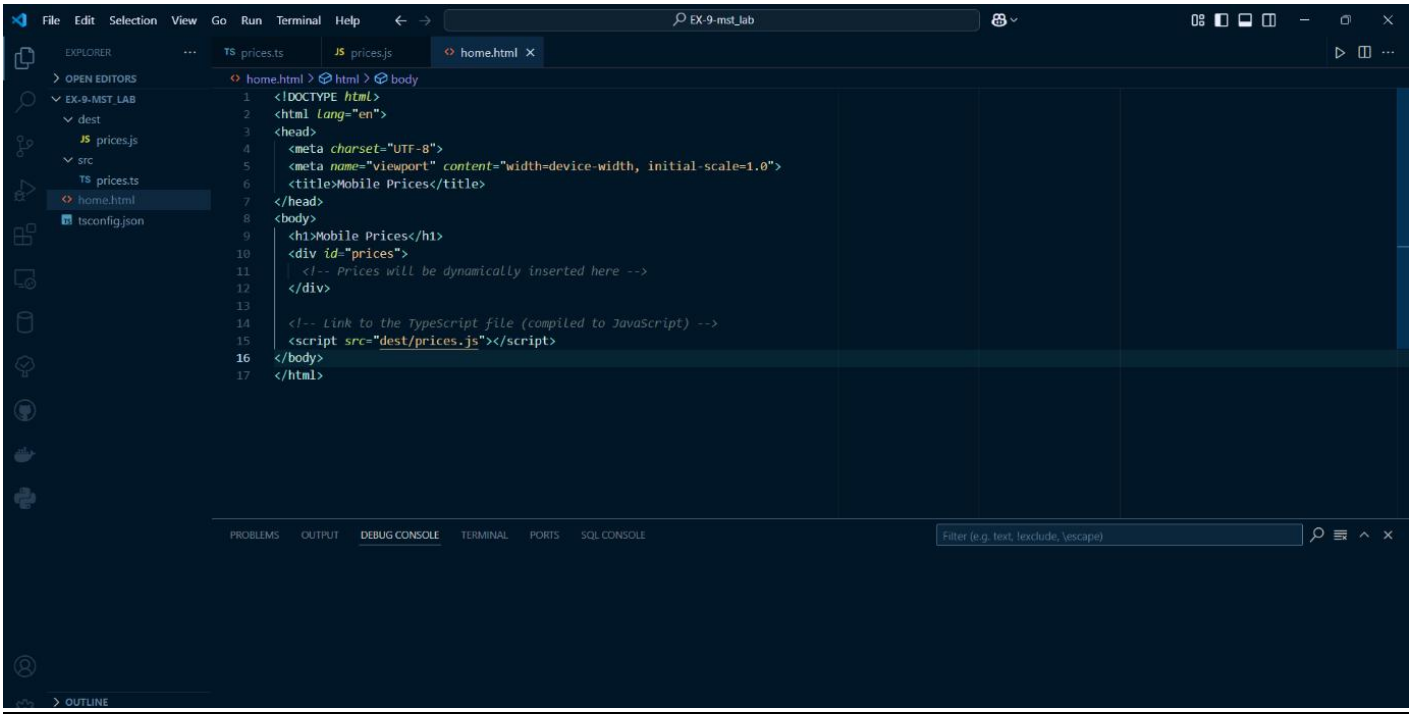
The typescript code:



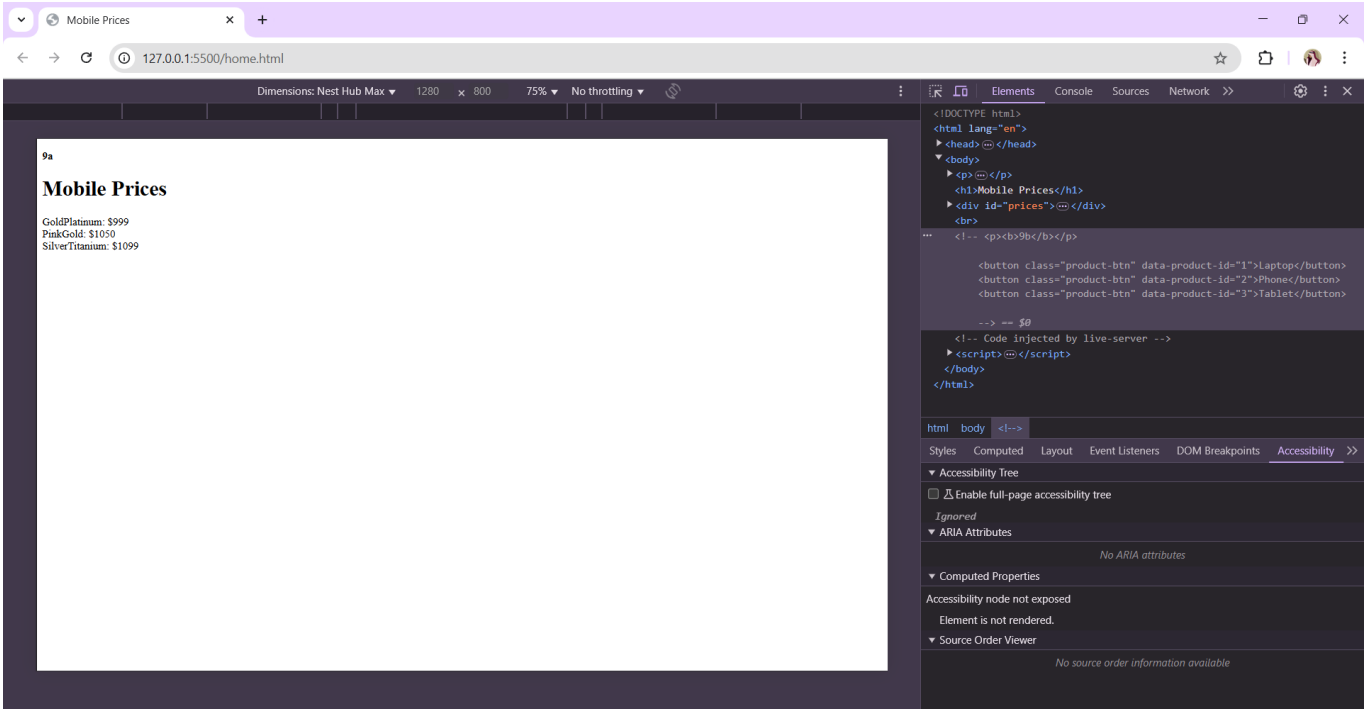
To run this code:

Go to your terminal and type: **tsc <name of typescript file.ts>**
The code is compiled and javascript code file is generated in your given destination path.

The HTML part:



The output:



9b:

Course Name: Typescript

Module Name: Function Define an arrow function inside the event handler to filter the product array with the selected product object using the product Id received by the function. Pass the selected product object to the next screen.

Functions in typescript:

Functions are one of the core building blocks of TypeScript. They allow you to encapsulate reusable logic and can have explicit type definitions for parameters and return values.

Function Syntax in TypeScript

A basic function in TypeScript follows this syntax:

```
function functionName(param1: type, param2: type): returnType {
    // function body
    return value;
}
```

- Compare the above typescript function using javascript function.

Types of Functions in TypeScript:

a) Named Functions:

Named functions have an explicit name and can be called using that name.

```
function add(a: number, b: number): number {
    return a + b;
}

console.log(add(5, 10)); // Output: 15
```

b) Anonymous Functions (Function Expressions):

Functions can be assigned to variables without naming them.

```
const multiply = function (x: number, y: number): number {
    return x * y;
};

console.log(multiply(3, 4)); // Output: 12
```

c) Arrow Functions (Lambda Functions):

Arrow functions provide a concise way to write functions.

```ts
const divide = (x: number, y: number): number => x / y;

console.log(divide(10, 2)); // Output: 5
```

d) Optional Parameters

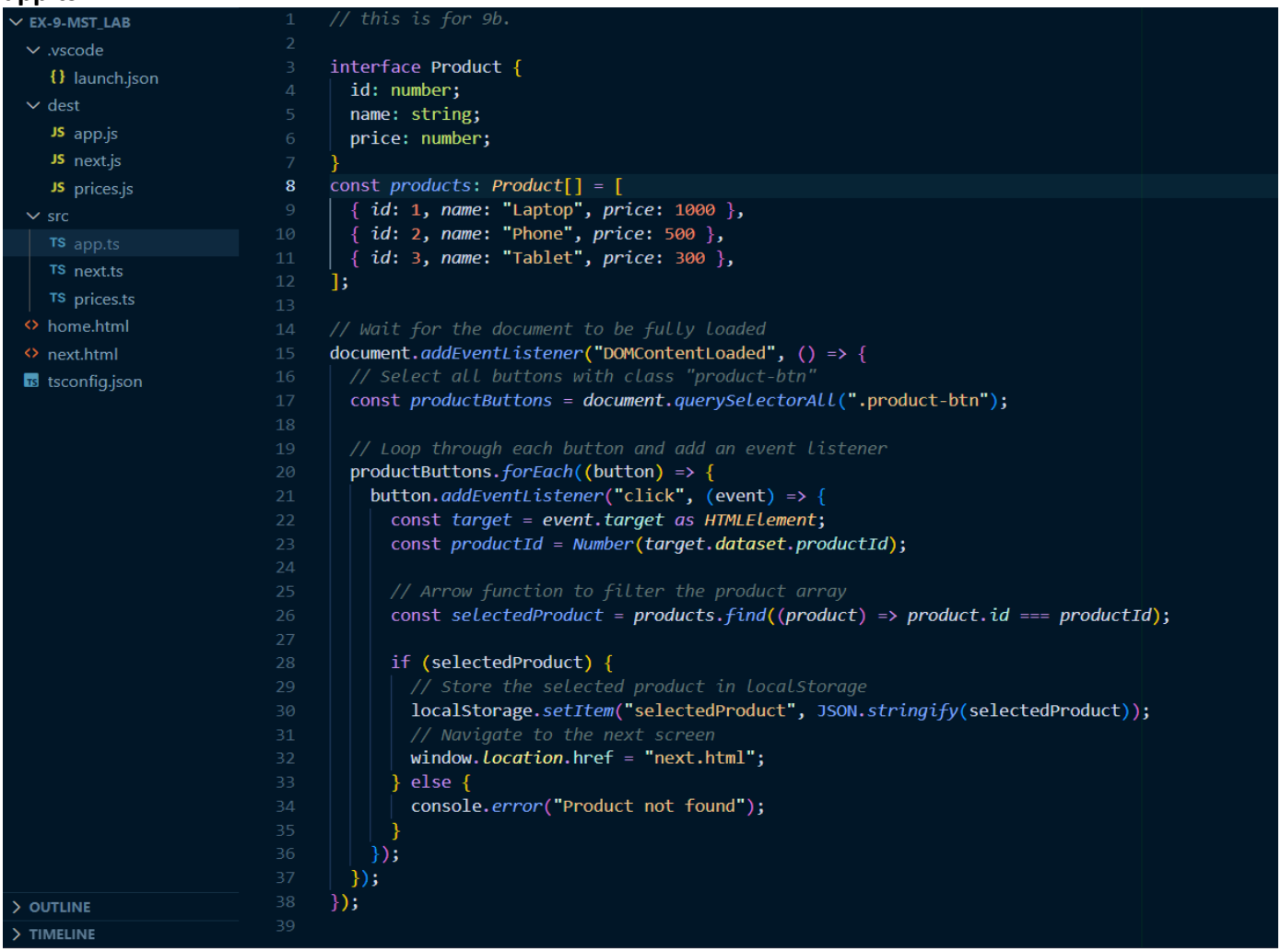
Use **?** to make parameters optional.

```ts
function greet(name: string, age?: number): string {
    return age ? `Hello, ${name}, you are ${age} years old.` : `Hello, ${name}`;
}

console.log(greet("John"));       // Output: Hello, John
console.log(greet("John", 25));   // Output: Hello, John, you are 25 years old.
```

Function Define an arrow function inside the event handler to filter the product array with the selected product object using the product Id received by the function. Pass the selected product object to the next screen.

To achieve the above, we created two .ts files they are: **app.ts** and **next.ts**

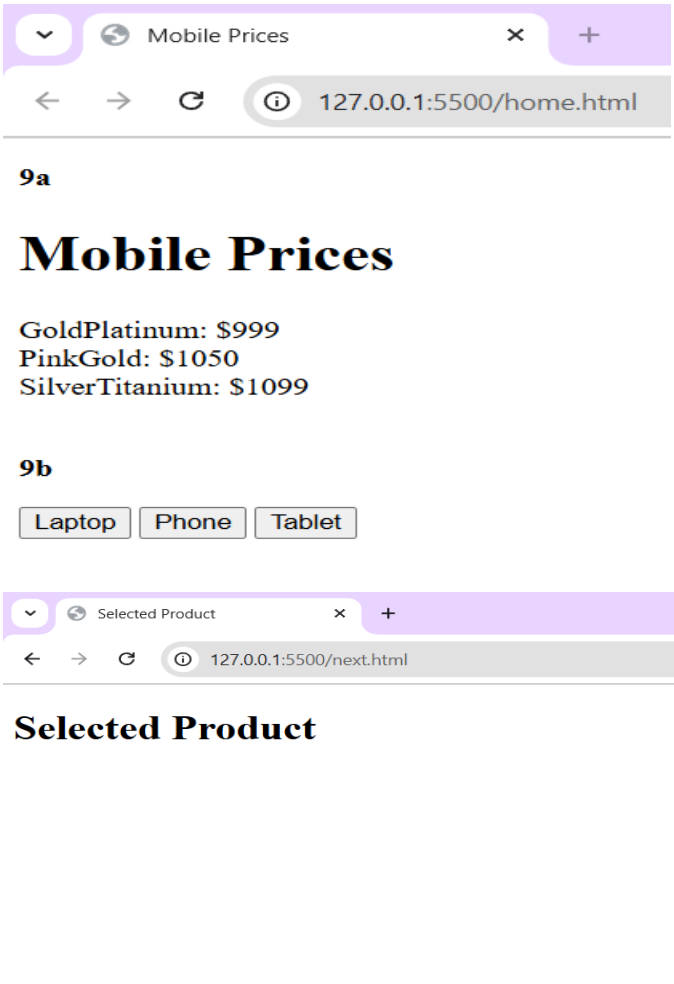
app.ts:



To pass the selected product object to the next screen, we use the concept of **localStorage** as we can observe in the above image.

next.ts:

The output:



9c:

Course Name: Typescript

Module Name: Parameter Types and Return Types Consider that developer needs to declare a function - getMobileByVendor which accepts string as input parameter and returns the list of mobiles.

Parameter Types and Return Types:

Parameter Type – Specifies the type of input the function accepts.

Return Type – Specifies the type of output the function returns.

Consider that developer needs to declare a function - getMobileByVendor which accepts string as input parameter and returns the list of mobiles.

The program:

The output:

9a

Mobile Prices

GoldPlatinum: $999
PinkGold: $1050
SilverTitanium: $1099

9b

Laptop | Phone | Tablet

9c

Mobiles from vivo: vivo t1 44w, vivo T3x 5g

9d:

Course Name: Typescript

Module Name: Arrow Function Consider that developer needs to declare a manufacturer's array holding 4 objects with id and price as a parameter and needs to implement an arrow function - myfunction to populate the id parameter of manufacturers array whose price is greater than or equal.

Arrow function in Typescript:

Arrow functions provide a concise way to write functions in TypeScript. Here's the general syntax:

Basic Syntax:

```
const functionName = (param1: Type, param2: Type): ReturnType => {
    // Function body
    return value;
};
```

Examples of Arrow Functions in TypeScript:

i. Arrow Function Without Parameters:

```
const greet = (): void => console.log("Hello, TypeScript!");
greet(); // Output: Hello, TypeScript!
```

ii. Arrow Function with Parameters:

```
const add = (a: number, b: number): number => a + b;
console.log(add(5, 3)); // Output: 8
```

iii. Arrow Function Returning an Object:

```
const getUser = (id: number) => ({ id: id, name: "John Doe" });
console.log(getUser(1)); // Output: { id: 1, name: 'John Doe' }
```

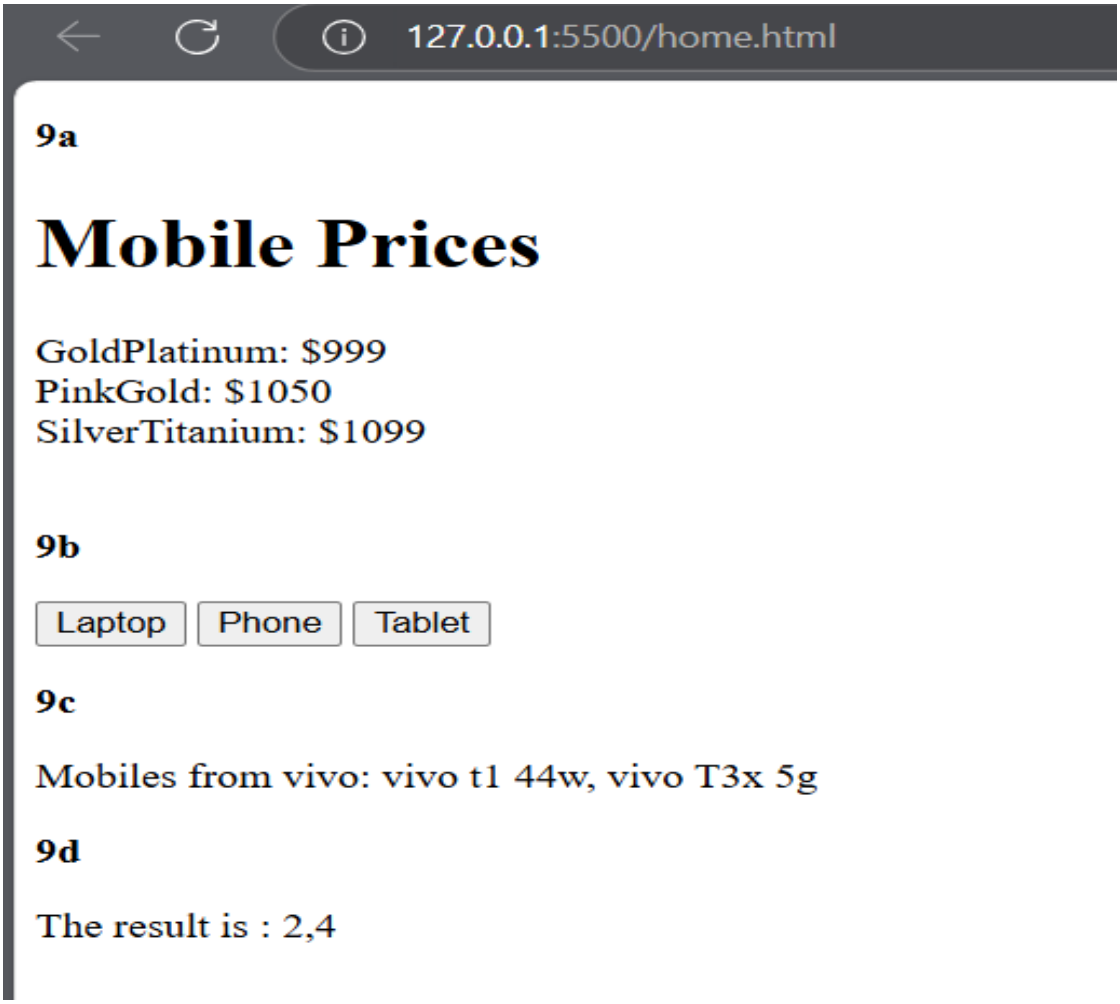
iv. Arrow Function with Array Methods:

```
const numbers = [1, 2, 3, 4, 5];
const squares = numbers.map(num => num * num);
console.log(squares); // Output: [1, 4, 9, 16, 25]
```

Consider that developer needs to declare a manufacturer's array holding 4 objects with id and price as a parameter and needs to implement an arrow function - myfunction to populate the id parameter of manufacturers array whose price is greater than or equal.

The program

The output



9e:

Course Name: Typescript

Module Name: Optional and Default Parameters Declare a function - getMobileByManufacturer with two parameters namely manufacturer and id, where manufacturer value should passed as Samsung and id parameter should be optional while invoking the function, if id is passed as 101 then this function should print Mobile with ID 101 found!

Optional and Default Parameters in Typescript:

Optional Parameter (?)

An optional parameter may or may not be provided when calling the function.

The example of optional parameter in typescript:

```
const greet = (name?: string): string => {
    return name ? `Hello, ${name}!` : "Hello, Guest!";
};

console.log(greet());             // Output: Hello, Guest!
console.log(greet("Ucenjntuk")); // Output: Hello, Ucenjntuk!
```

Default Parameter (=)

A default parameter has a **predefined value** if not passed.

The example of default parameter in typescript:

```
const greet = (name: string = "Guest"): string => {
    return `Hello, ${name}!`;
};

console.log(greet());              // Output: Hello, Guest!
console.log(greet("Ucenjntuk"));   // Output: Hello, Ucenjntuk!
```

Combined Example: Optional + Default Parameters:

```
const greet = (name: string = "Guest", age?: number): string => {
    return age ? `Hello, ${name}! You are ${age} years old.` : `Hello, ${name}!`;
};

console.log(greet());             // Output: Hello, Guest!
console.log(greet("Alice"));      // Output: Hello, Alice!
console.log(greet("Bob", 30));    // Output: Hello, Bob! You are 30 years old.
```

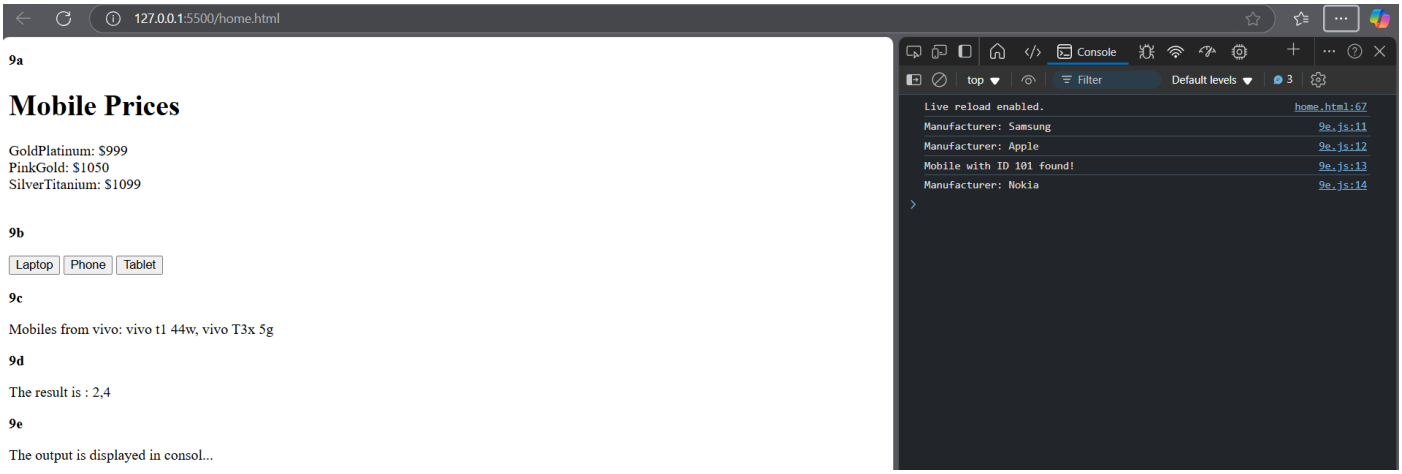
Parameters Declare a function - getMobileByManufacturer with two parameters namely manufacturer and id, where manufacturer value should passed as Samsung and id parameter should be optional while invoking the function, if id is passed as 101 then this function should print Mobile with ID 101 found!

The program

src > TS 9e.ts > ...

```ts
//9e
// Function with default and optional parameters
const getMobileByManufacturer = (manufacturer: string = "Samsung", id?: number): string => {
    if (id === 101) {
        return "Mobile with ID 101 found!";
    }
    return `Manufacturer: ${manufacturer}`;
};

// Example Usage
console.log(getMobileByManufacturer());          // Output: Manufacturer: Samsung
console.log(getMobileByManufacturer("Apple"));    // Output: Manufacturer: Apple
console.log(getMobileByManufacturer("OnePlus", 101)); // Output: Mobile with ID 101 found!
console.log(getMobileByManufacturer("Nokia", 200)); // Output: Manufacturer: Nokia
```

The output



Experiment-10

A) Module Name: Rest Parameter

Aim: Implement business logic for adding multiple Product values into a cart variable which is type of string array.

Description:

Rest Parameter : The rest parameter in TypeScript allows a function to accept an indefinite number of arguments as an array. It enables the developer to work with variable numbers of arguments without explicitly defining each one in the function signature.

Syntax : The rest parameter is represented using the ellipsis (**...**) followed by a variable name in the parameter list.

```
function functionName (...rest: type[]) {

// Function body

}
```

Parameters:

- **functionName:** The name of your function.
- **…rest:** The rest parameter that collects all additional arguments into an array.
- **type[]:** Specifies the type of elements in the rest array (e.g., number[], string[]).

Implementation of business logic for adding multiple Product values into a cart variable which is type of string array.

```typescript
//Ex-10.a Rest Parameter
let cart:String[] = []; //declaration of array to store strings

//Function definition to add items into cart using rest parameter
let addCartItems = (...products : String[]):void=>{
    console.log("Products parameter value : ",products)
    cart = [...cart,...products]
    console.log("After adding items to cart, the cart contains : ",cart)
}

addCartItems("Mobile","Laptop","Headphones")
addCartItems("smart watch","Air conditioner")
```

Output :

```
Products parameter value :  [ 'Mobile', 'Laptop', 'Headphones' ]
After adding items to cart, the cart contains :  [ 'Mobile', 'Laptop', 'Headphones' ]
Products parameter value :  [ 'smart watch', 'Air conditioner' ]
After adding items to cart, the cart contains :  [ 'Mobile', 'Laptop', 'Headphones', 'smart watch', 'Air conditioner' ]
```

B) Module Name: Creating an Interface

Aim: Declare an interface named - Product with two properties like productId and productName with a number and string datatype and need to implement logic to populate the Product details.

Description:

Interface :

In TypeScript, an interface is a powerful and flexible way to define the shape of objects, classes, and functions. It allows you to specify the structure of an object or a class without specifying its exact implementation, enabling TypeScript to enforce a contract for how objects and classes should be structured.

Syntax:

Interface interfaceName {

Property1 : property type,

Property2 : property type,

….}

Using interfaces with objects:

```
interface Person {
    name: string; age: number;
}
let personObj: Person = { name: "John", age: 30 };
```

Here, the Person interface defines an object with two properties: name (string) and age (number).

The variable personObj must match the shape defined by the Person interface. Any deviation (e.g., missing properties) will result in an error.

Properties of Interfaces:

1. Extending Properties:
Interfaces in TypeScript can extend other interfaces, allowing for property reuse and additional customization.

```
interface For_Array {
   var1: string;
}
interface For_List extends For_Array {
   var2: string;
}
```

- The For_List interface extends For_Array, inheriting the var1 property while introducing a new property var2.
- This allows a hierarchical design of interfaces, promoting modularity and reuse.

Example:

```
interface NumberArray {
    [index: number]: number;
}

let numbers: NumberArray = [1, 2, 3, 4];
```

For objects with dynamic keys, an index signature can be used:

```
interface Dictionary {
    [key: string]: string;
}

let dict: Dictionary = {
    first: "apple",
    second: "banana"
};
```

The above code will create an object with first and second as properties.

2. Read-Only Properties:
Properties marked as readonly cannot be modified after their initial assignment.
Example:

```
interface For_class {
   readonly name: string;
   id: number;
}
```

- The name property in the For_class interface is immutable, ensuring its value remains constant.
- This is ideal for scenarios where certain object attributes must remain unchanged, such as IDs or constants.

Example:

```
interface Point {
    readonly x: number;
    readonly y: number;
}

let point: Point = { x: 10, y: 20 };
point.x = 30;
```

```
[ERROR] 16:01:18 x Unable to compile TypeScript:
src/index.ts(20,7): error TS2540: Cannot assign to 'x' because it is a read-only property.
```

readonly is used to declare properties of objects or elements of arrays that

cannot be modified after they are assigned. It can be applied to properties of an interface, class, or array elements to prevent changes to them, but the reference to the object or array itself can still be modified.

3. Optional Properties

The ? symbol makes properties optional, adding flexibility to object definitions.

```
interface For_function {
   (key: string, value?: string): void;
}
```

- The value parameter is optional, meaning functions can be defined with or without it.
- This reduces constraints on function parameters, accommodating varying use cases while maintaining type safety.

Example:

```
interface Person {
    name: string;
    age?: number;  // Optional property
}

let person1: Person = { name: "Alice" };
let person2: Person = { name: "Bob", age: 25 };
console.log("person 1: ",person1)
console.log("person 2:",person2)
```

Output:

```
person 1:  { name: 'Alice' }
person 2: { name: 'Bob', age: 25 }
```

Interfaces for functions:

Interfaces can be used to define function signatures, including the types of parameters and the return type.

```
interface Greeter {
    (name: string): string;
}

let greet: Greeter = function (name: string) {
    return `Hello, ${name}!`;
};

console.log(greet("Alice"));
```

Output :

Hello, Alice!

Extending Interfaces:

You can extend an interface, creating a new interface that inherits from one or more existing interfaces.

Example:

```
interface Animal {
    name: string;
    sound(): void;
}

interface Dog extends Animal {
    breed: string;
}

const dog: Dog = {
    name: "Max",
    breed: "Golden Retriever",
    sound() {
        console.log("Woof!");
    }
};
```

Interfaces with classes:

Interfaces can be used to enforce the structure of a class. When a class implements an interface, it must adhere to the structure defined by that interface.

```
interface Animal {
    name: string;
    speak(): void;
}

class Dog implements Animal {
    name: string;

    constructor(name: string) {
        this.name = name;
    }

    speak() {
        console.log("Woof!");
    }
}

const dog = new Dog("Max");
dog.speak(); // Output: Woof!
```

Declare an interface named - Product with two properties like productId and productName with a number and string datatype and need to implement logic to populate the Product details.

Code:

```
//Ex-10.b Creating an interface
interface Product {
    productId : number,
    productName : String
}

//Basic object declaration
const product1:Product = {productId: 1,productName:"Smart Phone"}
const product2:Product = {productId: 2,productName:"Smart Watch"}
console.log("Product-1 Details : ")
console.log(`Product ID: ${product1.productId}, Product Name: ${product1.productName}`);
console.log("Product-2 Details : ")
console.log(`Product ID: ${product2.productId}, Product Name: ${product2.productName}`);

//Dynamic creation using function
const populateProductInterface = (productId : number,productName : String):Product=>{
    return {productId,productName};
}
const myProd1 : Product = populateProductInterface(3,"Tablet")
const myProd2 : Product = populateProductInterface(4,"Laptop")
console.log("Product-3 Details : ")
console.log(`Product ID: ${myProd1.productId}, Product Name: ${myProd1.productName}`);
console.log("Product-4 Details : ")
console.log(`Product ID: ${myProd2.productId}, Product Name: ${myProd2.productName}`);

//Class implementation
class ProductClass implements Product {
    productId: number;
    productName: String;
    constructor( productId : number, productName : string) {
        this.productId = productId,
        this.productName = productName
    }
}
const classProd1 = new ProductClass(5,"Television")
const classProd2 = new ProductClass(6,"Drone")
console.log("Product-5 Details : ")
console.log(`Product ID: ${classProd1.productId}, Product Name: ${classProd1.productName}`);
console.log("Product-6 Details : ")
console.log(`Product ID: ${classProd2.productId}, Product Name: ${classProd2.productName}`);
```

Output:

```
Product-1 Details :
Product ID: 1, Product Name: Smart Phone
Product-2 Details :
Product ID: 2, Product Name: Smart Watch
Product-3 Details :
Product ID: 3, Product Name: Tablet
Product-4 Details :
Product ID: 4, Product Name: Laptop
Product-5 Details :
Product ID: 5, Product Name: Television
Product-6 Details :
Product ID: 6, Product Name: Drone
```

C) **Module Name:** Duck Typing

Aim: Declare an interface named - Product with two properties like productId and productName with the number and string datatype and need to implement logic to populate the Product details.

Description:

Duck typing:

TypeScript is a structural typing language, which means it uses the shape or structure of an object to determine whether it fits a particular type. In other words, as long as an object has the necessary properties or methods, TypeScript doesn't care about the actual type or class from which the object originates.

How Duck Typing Works in TypeScript

- You don’t need to explicitly declare a class or type to implement an interface.
- If an object or variable contains the expected structure, TypeScript will treat it as compatible with the interface.

Example:

```
interface Animal {
    name: string;
    sound(): void;
}
class Dog {
    name: string;
    constructor(name: string) {
        this.name = name;
    }
    sound() {
        console.log("Woof!");
    }
}
class Car {
    name: string;
    constructor(name: string) {
        this.name = name;
    }
    sound() {
        console.log("Vroom!");
    }
}

// Both Dog and Car have the same structure
function makeSound(animal: Animal) {
    animal.sound();
}
const dog = new Dog("Max");
const car = new Car("Tesla");
makeSound(dog); // Woof!
makeSound(car); // Vroom!
```

Declare an interface named - Product with two properties like productId and productName with the number and string datatype and need to implement logic to populate the Product details.

Duck typing with interface:

```
const product1 = { productId: 1, productName: "Laptop" }; // Duck typing
const product2 = { productId: 2, productName: "Smartphone" }; // Duck typing
const product3 = { productId: 3, productName: "Laptop",price : 10000 };//Duck typing

interface Product {
    productId : number,
    productName : String
}

// Function to display product details
function displayProductDetails(product: Product): void {
  console.log(`Product ID: ${product.productId}, Product Name: ${product.productName}`);
}

// Populate and display
displayProductDetails(product1);
displayProductDetails(product2);
displayProductDetails(product3)
```

In this approach, you create plain objects that conform to the Product interface structure. When it comes to product3, there is an extra property price, but the typescript accepts it. Because it resembles the properties of interface Product extending with price property. This works only when object is created explicitly.

Output:

```
Product ID: 1, Product Name: Laptop
Product ID: 2, Product Name: Smartphone
Product ID: 3, Product Name: Laptop
```

Duck typing without interface:

```
const product1 = { productId: 1, productName: "Laptop" }; // Duck typing
const product2 = { productId: 2, productName: "Smartphone" }; // Duck typing
const product3 = { productId: 3, productName: "Laptop",price : 10000 };//Duck typing

function displayProductDetails(product: { productId: number; productName: string }): void { //duck typing
    console.log(`Product ID: ${product.productId}, Product Name: ${product.productName}`);
  }

// Populate and display
displayProductDetails(product1);
displayProductDetails(product2);
displayProductDetails(product3)
```

The above code checks the structure of product parameter with the object structure mentioned in the parameters of function. The object which matches the structure of object mentioned in the parameter, the function accepts it. Otherwise it will throw an error.

Output:

```
Product ID: 1, Product Name: Laptop
Product ID: 2, Product Name: Smartphone
Product ID: 3, Product Name: Laptop
```

Duck typing with inline objects:

```
const product3 = { productId: 3, productName: "Laptop",price : 10000 };//Duck typing

function displayProductDetails(product: { productId: number; productName: string }): void { //duck typing
    console.log(`Product ID: ${product.productId}, Product Name: ${product.productName}`);
}
displayProductDetails(product3)// Product ID: 3, Product Name: Laptop
displayProductDetails({ productId: 4, productName: "Headset",price:500 }); // Inline object with Duck Typing
```

Output:

```
[ERROR] 16:57:42 x Unable to compile TypeScript:
src/index.ts(145,64): error TS2353: Object literal may only specify known properties, and 'price' does not exist in type '{ productId: number; productName: string; }'.
```

When you pass an inline object directly into a function, TypeScript performs strict checks to ensure that:

- The object has only the exact properties defined in the expected type.
- Extra properties are not allowed because TypeScript assumes the inline object is only being used for that specific function call.

When you assign the object to a variable first, TypeScript assumes the object might be used elsewhere in your code, even in contexts where the extra properties could be useful. So, it relaxes the excess property check**.**

Here, TypeScript focuses only on whether the required properties (productId and productName) exist in the product variable. The extra price property is ignored because the variable might have broader usage.

Difference between interface and duck typing:

Aspect	Creating an Interface	Duck Typing
Explicit Declaration	Requires explicit definition using the `interface` keyword.	No explicit interface is defined; the object shape is inferred.
Reusability	Interfaces are reusable in multiple parts of the code.	Reuse is limited; requires repeated inline structure definitions.
Compile-Time Safety	Provides strong compile-time type-checking and better clarity in large codebases.	Also provides type safety but lacks explicit documentation or clarity.
Documentation	Self-documenting due to explicitly named interfaces.	Less clear; developers must infer the structure from usage.
Flexibility	Slightly less flexible; objects must match the interface exactly.	More flexible; extra properties in objects are ignored.
Scalability	Suitable for large projects with multiple developers and shared interfaces.	More suited for small projects or one-off use cases.

D)Module Name: Function Types.

Aim: Declare an interface with function type and access its value.

Description:

Function: A function is a block of code that performs a specific task, and it can be called or invoked to execute that task. Functions in TypeScript are similar to functions in JavaScript but with added type safety features. This allows you to define the types of inputs (parameters) and the type of output (return value), making your code more predictable and less error-prone.

Syntax:

Function functionName(arg1,arg2,….argn){

//block of code

}

There are two types of functions in typescript:

1.Named functions

2.Anonymous functions

Named functions:

A named function is defined as one that is declared and called by its given name. They may include parameters and have return types.

Syntax:

Function functionName(arg1:type,arg2:type…){

//block of code

}

```
function add(a: number, b: number): number {
  return a + b;
}
```

Anonymous functions:

An anonymous function is a function without a name. At runtime, these kinds of functions are dynamically defined as an expression. We may save it in a variable and eliminate the requirement for function names. They accept inputs and return outputs in the same way as normal functions do. We may use the variable name to call it when we need it. The functions themselves are contained inside the variable.

```
const greet = function (name: string): string {
    return `Hello, ${name}!`;
};

console.log(greet("Alice")); // Output: Hello, Alice!
```

3. Arrow Functions

A concise way to write functions using the => syntax:

```
const divide = (x: number, y: number): number => x / y;
```

4. Function Expressions

Functions defined as expressions and stored in variables:

```
const subtract: (x: number, y: number) => number = function (x, y)
  return x - y;
};
```

5. Function Types

Explicitly defining the type signature for a function:

```
type Operation = (x: number, y: number) => number;

const add: Operation = (x, y) => x + y;
```

6. Optional Parameters

Parameters that are not mandatory to pass:

```
function greet(name: string, age?: number): string {
  return age ? `Hello ${name}, age ${age}` : `Hello ${name}`;
}
```

7. Default Parameters

Parameters with default values:

```
function greet(name: string, age: number = 30): string {
  return `Hello ${name}, age ${age}`;
}
```

8. Overloaded Functions

Functions with multiple type signatures:

```
function combine(x: number, y: number): number;
function combine(x: string, y: string): string;
function combine(x: any, y: any): any {
  return x + y;
}
```

9. Callbacks

Functions passed as arguments to other functions:

```
function operate(x: number, y: number, callback: (a: number, b: number) => number): number {
  return callback(x, y);
}

operate(5, 10, (a, b) => a + b); // 15
```

10. Void Functions

Functions that do not return a value:

```
function logMessage(message: string): void {
  console.log(message);
}
```

11. Asynchronous Functions

Functions that return a Promise:

```
async function fetchData(url: string): Promise<string> {
  const response = await fetch(url);
  return response.text();
}
```

Declare an interface with function type and access its value.

```
//Ex-10.d
// Interface with a function type
interface ProductHandler {
    (productId: number, productName: string): string; // Function type
  }

  // Function implementation
  const addProduct: ProductHandler = (productId, productName) => {
    return `Product Added: ID = ${productId}, Name = ${productName}`;
  };

  // Accessing the function
  console.log(addProduct(101, "Laptop")); // Output: Product Added: ID = 101, Name = Laptop
```

Output:

```
Product Added: ID = 101, Name = Laptop
```

EXERCISE 11

In TypeScript, you can create an interface that extends other interfaces by using the **extends** keyword. Below is an example of how to declare a **ProductList** interface that extends properties from two other interfaces, **Category** and **Product**. Additionally, I'll show you how to create a variable of this interface type.

Step 1: Declare the Interfaces

First, we will declare the **Category** and **Product** interfaces.

Code:

```
interface Category {
    id: number;
    name: string;
}
interface Product {
    id: number;
    name: string;
    price: number;
}
```

Step 2: Declare the ProductList Interface

Next, we will declare the **ProductList** interface that extends both **Category** and **Product**.

Typescript.

Code:

```
interface ProductList extends Category, Product {
    stock: number;
```

Step 3: Create a Variable of ProductList Type

Now, we can create a variable of type **ProductList** and initialize it with appropriate values.

typescript

const productItem: ProductList = {

id: 1,

name: "Laptop",

price: 999.99,

stock: 50,

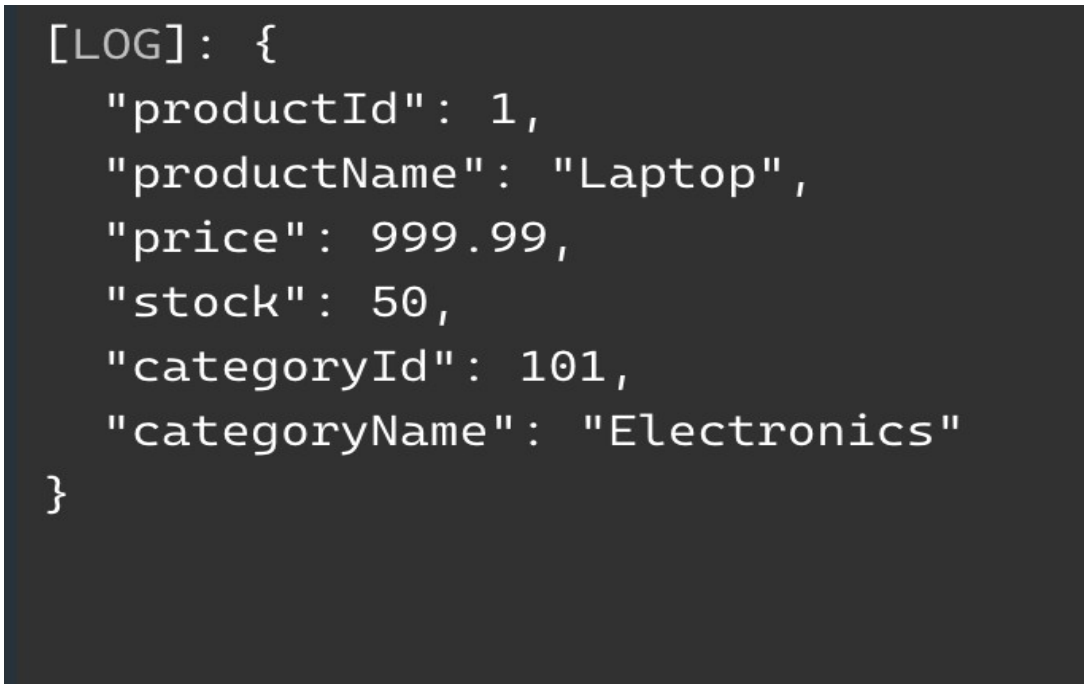
categoryId: 101,

categoryName: "Electronics"

};

```typescript
// Step 1: Declare the Category interface
interface Category {
    categoryId: number;      // Unique identifier for the category
    categoryName: string;    // Name of the category
}

// Step 2: Declare the Product interface
interface Product {
    productId: number;       // Unique identifier for the product
    productName: string;     // Name of the product
    price: number;           // Price of the product
}

// Step 3: Declare the ProductList interface that extends Category and Product
interface ProductList extends Category, Product {
    stock: number;           // Additional property for stock quantity
}

// Step 4: Create a variable of ProductList type
const productItem: ProductList = {
    productId: 1,                // Product ID
    productName: "Laptop",       // Product Name
    price: 999.99,               // Product Price
    stock: 50,                   // Stock quantity
    categoryId: 101,             // Category ID
    categoryName: "Electronics"  // Category Name
};

// Step 5: Log the productItem to the console
console.log(productItem);
```

Output:



1. **Interfaces Definition**:
    - **Category Interface**:
        - Represents a product category with two properties:
            - **categoryId**: A unique identifier for the category (type: **number**).
            - **categoryName**: The name of the category (type: **string**).
    - **Product Interface**:
        - Represents a product with three properties:
            - **productId**: A unique identifier for the product (type: **number**).
            - **productName**: The name of the product (type: **string**).
            - **price**: The price of the product (type: **number**).
    - **ProductList Interface**:
        - Extends both **Category** and **Product** interfaces, combining their properties and adding one additional property:
            - **stock**: The quantity of the product available in stock (type: **number**).

2. **Variable Creation**:
    - A variable named **productItem** is created, which is of type **ProductList**. This variable is initialized with specific values for all properties defined in the **ProductList** interface, including both product and category details.
3. **Output**:
    - The **console.log(productItem);** statement outputs the **productItem** object to the console, displaying its structure and values. The output includes all properties, demonstrating how the interfaces work together to create a comprehensive data model.

Key Features:

- **Type Safety**: The use of TypeScript interfaces ensures that the properties of the objects adhere to defined types, reducing runtime errors and improving code reliability.
- **Extensibility**: The structure allows for easy addition of new properties or methods in the future, making it adaptable to changing requirements.
- **Clarity**: By renaming properties in the **Category** interface, the code avoids potential naming conflicts, enhancing readability and maintainability.

11b. Consider the Mobile Cart application, Create objects of the Product class and place them into the productlist array.

The focus is on creating a **Product** class, instantiating objects of this class, and storing them in an array called **productList**. This approach allows for efficient management of product data within the application.

Code Structure

1. **Product Class Definition**: The **Product** class encapsulates the properties and methods related to a product.
2. **Product List Array**: An array named **productList** is created to hold instances of the **Product** class.
3. **Object Creation**: Multiple product objects are instantiated and added to the **productList** array.

Implementation

Below is the complete TypeScript code for the Mobile Cart application:

Code:

```
class Product {
        productId: number
        productName: string;
        price: number;
        stock: number;
    constructor(productId: number, productName: string, price: number, stock: number) {
        this.productId = productId;
        this.productName = productName;
        this.price = price;
        this.stock = stock;
     }
    displayProductInfo(): string {
        return `Product ID: ${this.productId}, Name: ${this.productName}, Price: $${this.price.toFixed(2)}, Stock: ${this.stock}`;
      }
    }
```

```
const productList: Product[] = [];

const product1 = new Product(1, "Smartphone", 699.99, 30);
const product2 = new Product(2, "Tablet", 399.99, 20);
const product3 = new Product(3, "Laptop", 999.99, 15);

productList.push(product1);
productList.push(product2);
productList.push(product3);

productList.forEach(product => {
  console.log(product.displayProductInfo());
});
```

Detailed Explanation

1. **Product Class**:
   - **Properties**:
     - **productId**: A numeric identifier for the product.
     - **productName**: A string representing the name of the product.
     - **price**: A numeric value representing the price of the product.
     - **stock**: A numeric value indicating the available quantity of the product.
   - **Constructor**: Initializes the properties of the **Product** class when a new object is created.
   - **Method**: **displayProductInfo()** returns a formatted string containing the product's details.
2. **Product List Array**:
   - An array named **productList** is declared to hold instances of the **Product** class.
3. **Object Creation**:
   - Three instances of the **Product** class (**product1**, **product2**, and **product3**) are created with specific values for their properties.

- These instances are added to the **productList** array using the **push()** method.

4. **Displaying Product Information**:
    - The **forEach** method is used to iterate over the **productList** array, calling the **displayProductInfo()** method for each product to log its details to the console.

Expected Output

When the above TypeScript code is executed, the output in the console will be:

plaintext

1Product ID: 1, Name: Smartphone, Price: $699.99, Stock: 30

2Product ID: 2, Name: Tablet, Price: $399.99, Stock: 20

3Product ID: 3, Name: Laptop, Price: $999.99, Stock: 15

11c. Declare a class named - Product with the below-mentioned declarations: (i) productid as number property (ii) Constructor to initialize this value (iii) getProductId method to return the message "Product id is <<id value>>".

This report details the implementation of a TypeScript class named **Product**. The class is designed to encapsulate the properties and behaviors associated with a product, specifically focusing on the **productId** property. The implementation includes a constructor for initializing the **productId** and a method to retrieve the product ID in a formatted message.

Code Structure

1. **Class Declaration**: The **Product** class is declared with a property for the product ID.
2. **Constructor**: A constructor is defined to initialize the **productId**.
3. **Method**: A method named **getProductId** is implemented to return a formatted message containing the product ID.

Implementation

Below is the complete TypeScript code for the **Product** class:

```
class Product {
    private productId: number; // Private property to store the product ID
   constructor(productId: number) {
   this.productId = productId; // Initialize the productId property
  }
   getProductId(): string {
            return `Product id is ${this.productId}`; // Return formatted message
   }
}


const product = new Product(101);


console.log(product.getProductId());
```

Detailed Explanation

1. **Class Declaration**:
    - The **Product** class is defined using the **class** keyword.
2. **Property Declaration**:
    - **productId**: A private property of type **number** that stores the unique identifier for the product. The **private** access modifier ensures that this property cannot be accessed directly from outside the class.
3. **Constructor**:
    - The constructor takes a parameter **productId** of type **number** and initializes the class property with this value. This allows for the creation of **Product** objects with specific IDs.
4. **Method**:
    - **getProductId()**: This method returns a string message that includes the product ID. The message is formatted as **"Product id is <<id value>>"**.
5. **Object Creation**:
    - An instance of the **Product** class is created with a specific product ID (e.g., **101**).
6. **Output**:
    - The **getProductId()** method is called on the **product** instance, and the result is logged to the console.

Expected Output

When the above TypeScript code is executed, the output in the console will be:

plaintext

Product id is 101

11d. Create a Product class with 4 properties namely productId, productName, productPrice, productCategory with private, public, static, and protected access modifiers and accessing them through Gadget class and its methods.

Objective

To create a **Product** class with four properties: **productId**, **productName**, **productPrice**, and **productCategory**, utilizing various access modifiers (private, public, static, and protected). Additionally, a **Gadget** class will be implemented to access and manipulate these properties.

Class Design

1. **Product Class**
    - **Properties**:
        - **productId**: A unique identifier for the product (private).
        - **productName**: The name of the product (public).
        - **productPrice**: The price of the product (protected).
        - **productCategory**: The category of the product (static).
    - **Access Modifiers**:
        - **private**: Only accessible within the class itself.
        - **public**: Accessible from anywhere.
        - **protected**: Accessible within the class and by derived classes.
        - **static**: Belongs to the class itself rather than instances of the class.
2. **Gadget Class**
    - This class will create instances of the **Product** class and provide methods to access and manipulate the properties of the **Product** class.

Program:

```
class Product {
    private productId: number;
    public productName: string;
    protected productPrice: number;
    static productCategory: string = "Electronics"; // Static property

    constructor(productId: number, productName: string, productPrice: number) {
        this.productId = productId;
        this.productName = productName;
        this.productPrice = productPrice;
    }

    // Public method to get productId
    public getProductId(): number {
        return this.productId;
    }

    // Public method to get productPrice
    public getProductPrice(): number {
        return this.productPrice;
    }

    // Static method to get productCategory
    public static getProductCategory(): string {
        return Product.productCategory;
    }
}


class Gadget {
    private product: Product;

    constructor(productId: number, productName: string, productPrice: number) {
        this.product = new Product(productId, productName, productPrice);
    }

    public displayProductDetails(): void {
        console.log(`Product ID: ${this.product.getProductId()}`);
        console.log(`Product Name: ${this.product.productName}`);
        console.log(`Product Price: ${this.product.getProductPrice()}`);
        console.log(`Product Category: ${Product.getProductCategory()}`);
    }
}

// Example usage
const gadget = new Gadget(1, "Smartphone", 699);
gadget.displayProductDetails();
```

Output

```
Product ID: 1
Product Name: Smartphone
Product Price: 699
Product Category: Electronics

[Execution complete with exit code 0]
```

lana

Explanation of the Code:

1. **Product Class**:
   - The **Product** class has a constructor that initializes the properties.
   - The **productId** is private, meaning it cannot be accessed outside the **Product** class.
   - The **productName** is public, allowing it to be accessed from anywhere.
   - The **productPrice** is protected, which means it can be accessed in subclasses.
   - The **productCategory** is static, meaning it is shared across all instances of the **Product** class.
2. **Gadget Class**:
   - The **Gadget** class creates an instance of the **Product** class.
   - It has a method **displayProductDetails** that calls the public method **getProductDetails** of the **Product** class to display product information.
   - The **getPrice** method demonstrates how to access a protected method from the **Product** class using bracket notation.

Experiment 12

a.)

Class Inheritance in TypeScript

Objective:

Demonstrate object-oriented programming through class creation and inheritance.

Detailed Description:

A base class Product is created with a protected property productId. The getProduct() method displays the product ID. A subclass Gadget extends Product and adds a new property productName. It overrides the getProduct() method to also print the product name. This shows how inheritance allows reuse and extension of code in TypeScript. The use of super() ensures proper constructor chaining from parent to child.

Use Case:

Useful in real-world applications like e-commerce where a general Product class can be extended into specific types such as Electronics, Books, etc.

```
// Product.ts
class Product {
  protected productId: number;

  constructor(productId: number) {
    this.productId = productId;
  }

  getProduct(): void {
    console.log('Product id is: ' +
this.productId);
  }
}

class Gadget extends Product {
  constructor(public productName:
string, productId: number) {
    super(productId);
  }

  getProduct(): void {
    super.getProduct();
    console.log(`Product id is: $
{this.productId}, Product name is: $
{this.productName}`);
  }
}

// Usage
const g = new Gadget('Tablet',
1234);
g.getProduct();
```

Output:

```
Product id is: 1234
Product id is: 1234, Product name is:
Tablet
```

b.)

Working with Namespaces

Objective:

Use TypeScript namespaces to modularize code logically.

Detailed Description:

A namespace Utility is created to group related functions. It contains a sub-namespace Payment for financial calculations like CalculateAmount(). MaxDiscountAllowed() function is also included in the main namespace. A separate file imports and uses these namespaces via triple-slash directives and aliasing. This demonstrates encapsulation and prevents global scope pollution.

Use Case:

Helpful in large projects where code organization and scope management are crucial without relying on external module loaders.

File: namespace_demo.ts

Ts

```ts
namespace Utility {
  export namespace Payment {
    export function
CalculateAmount(price: number, quantity:
number): number {
      return price * quantity;
    }
  }

  export function
MaxDiscountAllowed(noOfProduct: number):
number {
    return noOfProduct > 5 ? 40 : 10;
  }

  function privateFunc(): void {
    console.log('This is private...');
  }
}
```

File: Namespace_import.ts

Ts

```ts
/// <reference path="./
namespace_demo.ts" />
import util = Utility.Payment;

let paymentAmount =
util.CalculateAmount(1255, 6);
console.log(`Amount to be paid: $
{paymentAmount}`);

let discount =
Utility.MaxDiscountAllowed(6);
console.log(`Maximum discount allowed
is: ${discount}`);
```

c.)

ES Modules for Code Organization

Objective:

Show how to create and import modules in TypeScript.

Detailed Description:

The module_demo.ts file exports multiple entities: a class (Utility), an interface (Category), constants, and functions. These are imported into 2.module_import.ts using ES6 import statements. A new instance of the utility class is used to calculate amount and discount. This promotes modularity and code reusability, key in modern JavaScript/TypeScript apps.

Use Case:

This approach is used in enterprise-level web applications like shopping carts, content management systems, etc., to keep code organized and maintainable.

File: module_demo.ts

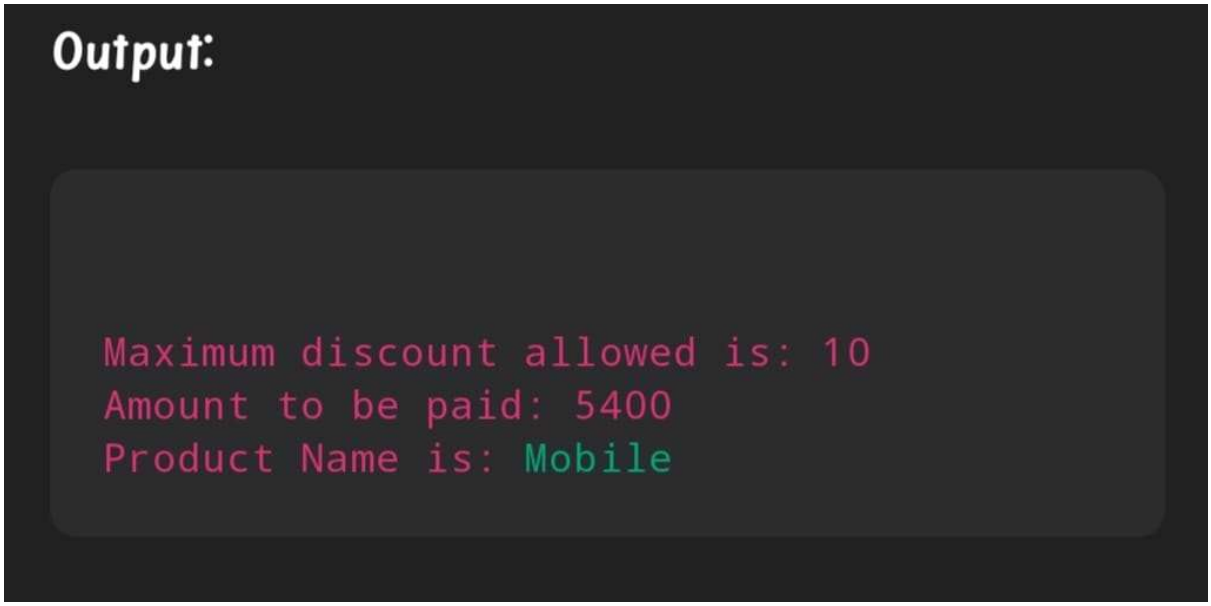
Ts

```ts
export function
MaxDiscountAllowed(noOfProduct: number):
number {
  return noOfProduct > 5 ? 30 : 10;
}

export class Utility {
  CalculateAmount(price: number,
quantity: number): number {
    return price * quantity;
  }
}

export interface Category {
  getCategory(productId: number):
string;
}

export const productName = 'Mobile';
```

File: 2.module_import.ts

Ts

```ts
import { Utility as mainUtility,
Category, productName,
MaxDiscountAllowed } from "./
module_demo";

const util = new mainUtility();
const price = util.CalculateAmount(1350,
4);
const discount = MaxDiscountAllowed(2);

console.log(`Maximum discount allowed
is: ${discount}`);
console.log(`Amount to be paid: ${price}
`);
console.log(`Product Name is: $
{productName}`);
```



d.)

Generics in TypeScript Functions

Objective:

Implement a generic function that can sort arrays of different data types.

Detailed Description:

A generic function sortedArr() takes an array of any type (number, string, etc.) and returns a sorted array. It uses a comparator inside the sort() method to handle sorting logic. The function is demonstrated on arrays of numbers and strings. This shows how generics provide flexibility while maintaining type safety.

Use Case:

Ideal when building utility libraries where the same logic (like sorting) can be applied to different data types, avoiding duplication of code.

Ts

```ts
function sortedArr<T>(arg: Array<T>):
Array<T> {
  return arg.sort((n1, n2) => {
    if (n1 > n2) return 1;
    if (n1 < n2) return -1;
    return 0;
  });
}

const num: Array<number> = [5, 12, 23,
42, 34];
const str: Array<string> = ['C', 'F',
'V', 'A'];

console.log(sortedArr(num));
console.log(sortedArr(str));
```

Output:

```
[5, 12, 23, 34, 42]
['A', 'C', 'F', 'V']
```