EXPERIMENT 1A)

AIM:

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.

DESCRIPTION:Case-insensitivity: HTML is case-insensitive, meaning that tags and attributes can be written in uppercase or lowercase letters and will still be recognized by the browser.

Platform-independency: HTML is platform-independent, meaning that it can be viewed on any device or operating system with a web browser.

DOCTYPE Declaration: The DOCTYPE declaration is used at the beginning of an HTML document to specify the version of HTML being used and to ensure that the document is rendered correctly by web browsers.

Types of Elements: HTML has several types of elements, including structural elements (such as <html>, <head>, and <body>), text-level elements (such as <p> and), and multimedia elements (such as <img>, <audio>, and <video>).

HTML Elements - Attributes: HTML elements can be customized with attributes, which provide additional information about the element and how it should be displayed or used. Examples of attributes include "class" for defining CSS styles, "id" for identifying a specific element, and "href" for specifying a hyperlink.

Metadata Element: The metadata element (<meta>) is used to provide information about the HTML document that is not displayed in the browser window. This includes information such as the author, description, and keywords, which can be used by search engines to index and display the webpage in search results.

PROGRAM:

```
<!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's Shopping</title>

</head>

<body>

<h1>Welcome to IEKart's Shopping!</h1>

<p>We sell a variety of products in different categories such as Electronics, Clothing,
Accessories and more!</p>



</body>

</html>
```

OUTPUT:

1.b)

AIM:

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

DESCRIPTION:

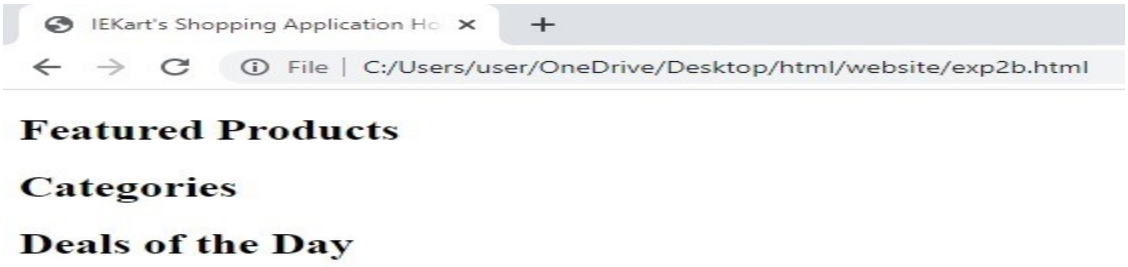
Sectioning elements in HTML are used to divide the content of a web page into logical sections, making it easier for users to understand and navigate the content. These elements include <header>, <nav>, <section>, <article>, <aside>, and <footer>. The <header> element is used to identify the header section of a page, while the <nav> element is used to define a set of navigation links.

PROGRAM:

```
<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>IEKart's Shopping Application Homepage</title>

<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.">

</head>

<body>

<header>

<!-- Header content goes here -->

</header>

<nav>

<!-- Navigation links go here -->

</nav>

<main>

<section>

<h1>Featured Products</h1>
```

```
<!-- Content for featured products section goes here -->

</section>

<section>

<h1>Categories</h1>

<!-- Content for categories section goes here -->

</section>

<section>

<h1>Deals of the Day</h1>

<!-- Content for deals section goes here -->

</section>

</main>

<aside>

<!-- Sidebar content goes here -->

</aside>

<footer>

<!-- Footer content goes here -->

</footer>

</body>

</html>
```

OUTPUT:



1.c)

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

DESCRIPTION:Paragraph Element: The <p> element is used to define a paragraph of text in HTML.

Division and Span Elements: The and elements are used to group elements and apply styles or classes to them. The element is a block-level element, while the element is an inline-level element.

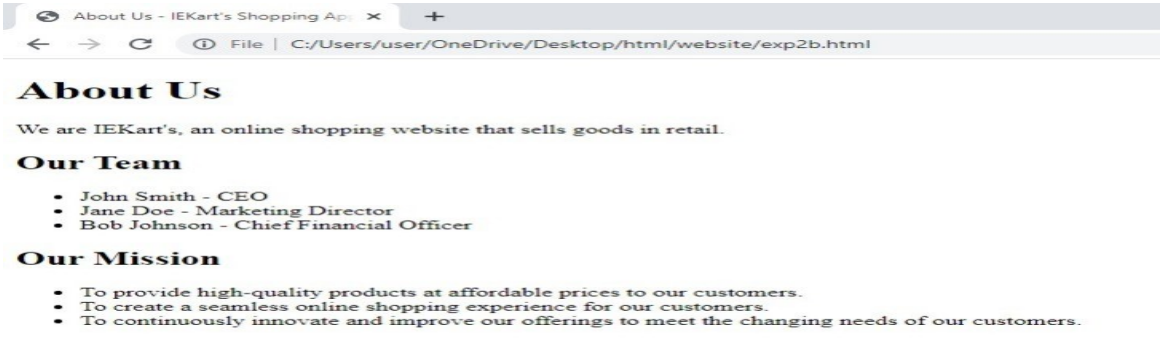
List Element: The <ul>, <ol>, and <li> elements are used to create lists in HTML. The <ul> element creates an unordered list, the <ol> element creates an ordered list, and the <li> element defines each item in the list.

PROGRAM:

```
<!DOCTYPE html>

<html>

<head>

<title>About Us - IEKart's Shopping Application</title>

</head>

<body>

<h1>About Us</h1>

<p>We are IEKart's, an online shopping website that sells goods in retail.</p>

<h2>Our Team</h2>

<ul>
```

```
<li>John Smith - CEO</li>

<li>Jane Doe - Marketing Director</li>

<li>Bob Johnson - Chief Financial Officer</li>

</ul>

<h2>Our Mission</h2>

<ul>

<li>To provide high-quality products at affordable prices to our customers.</li>

<li>To create a seamless online shopping experience for our customers.</li>

<li>To continuously innovate and improve our offerings to meet the changing needs of our customers.</li>

</ul>

</body>

</html>
```

OUTPUT:



1.d)

AIM:

Link "Login", "SignUp" and "Track order" to "Login.html", "SignUp.html" and "Track.html" page respectively. Bookmark each category to its details of IEKart's Shopping application.

DESCRIPTION:

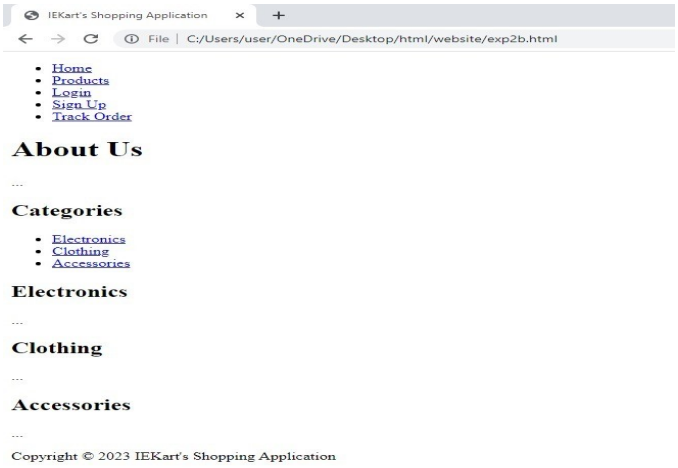
The Link element (<link>) is an HTML element used to define a relationship between the current document and an external resource. This element is commonly used to link stylesheets to an HTML document, allowing the page to be styled with CSS.

PROGRAM:

```
<!DOCTYPE html>

<html>

<head>

<title>IEKart's Shopping Application</title>

</head>

<body>


<ul>

<li><a href="Homepage.html">Home</a></li>

<li><a href="Products.html">Products</a></li>

<li><a href="Login.html">Login</a></li>

<li><a href="SignUp.html">Sign Up</a></li>

<li><a href="Track.html">Track Order</a></li>

</ul>


<h1>About Us</h1>
```

```
<p>...</p>
<h2>Categories</h2>
<ul>
<li><a href="#electronics">Electronics</a></li>
<li><a href="#clothing">Clothing</a></li>
<li><a href="#accessories">Accessories</a></li>
</ul>
<h2 id="electronics">Electronics</h2>
<p>...</p>
<h2 id="clothing">Clothing</h2>
<p>...</p>
<h2 id="accessories">Accessories</h2>
<p>...</p>

<footer>
<p>Copyright © 2023 IEKart's Shopping Application</p>
</footer>
</body>
</html>
```

OUTPUT:



1.e)

AIM: Add the © symbol in the Home page footer of IEKart's Shopping application.

DESCRIPTION:

In HTML, character entities are special codes used to represent special characters that are not part of the standard character set. These entities are defined by their unique entity name or a numeric code and are used to display symbols, foreign characters, mathematical symbols, and more. Examples of character entities include < for <, > for >, and & for &.

PROGRAM:

```
<footer>

<p>Copyright © 2023 IEKart's Shopping Application</p>

</footer>
```

OUTPUT:

1.F)

AIM:

Add the global attributes such as contenteditable, spellcheck, id etc. to enhance the Signup Page functionality of IEKart's Shopping application.

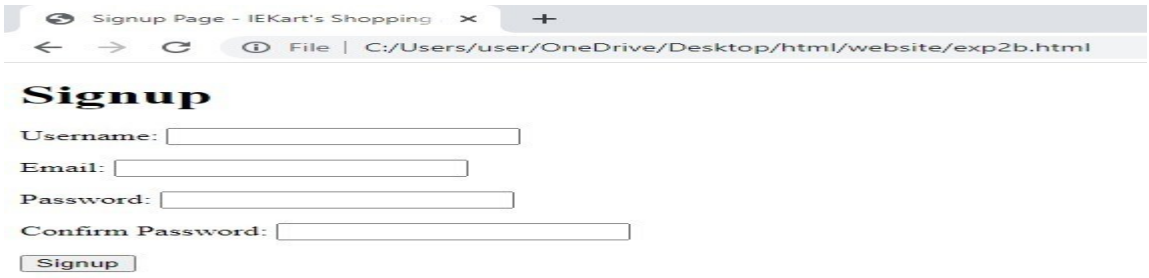
DESCRIPTION:HTML5 Global Attributes are attributes that can be used on any HTML element and are not limited to specific elements. These attributes can be used to provide additional information about an element, such as defining the class or id, setting styles, and assigning event handlers. Some commonly used global attributes include "class", "id", "style", "title", and "data-*".

PROGRAM:

```
<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Signup Page - IEKart's Shopping Application</title>

</head>

<body>

<h1>Signup</h1>

<form action="signup.php" method="POST">

<label for="username">Username:</label>

<input type="text" name="username" id="username" required

contenteditable="true"><br><BR>

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

<input type="email" name="email" id="email" required spellcheck="true">

<br><BR>

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

<input type="password" name="password" id="password" required>

<br><BR>

<label for="confirm_password">Confirm Password:</label>
```

```
<input type="password" name="confirm_password" id="confirm_password"

required><br><BR>

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

</form>

</body>

</html>
```

OUTPUT:



EXPERIMENT-2A)

AIM:Enhance the details page of IEKart's Shopping application by adding a table elementto display the available mobile/any inventories.

DESCRIPTION:Table elements in HTML are used to display data in a tabular format and can be customized using attributes such as colspan/rowspan, border, cellspacing, and cellpadding.

PROGRAM:

```
<!DOCTYPE html>

<head><title>Product Details</title></head>

<body>

    <div class="product-details">

     <h1>Product Name</h1>

    <p>Product description goes here</p>

<table border="1" cellspacing="0" cellpadding="10">
```

```
<tr>
<th>Model</th>
<th>Color</th>
<th>Storage</th>
<th>Price</th>
<th>Availability</th>
</tr>
<tr>
<td>Model A</td>
<td>Black</td>
<td>64GB</td>
<td>$699</td>
<td>In stock</td>
</tr>
<tr>
<td>Model A</td>
<td>White</td>
<td>128GB</td>
<td>$799</td>
<td>In stock</td>
</tr>
<tr>
<td>Model B</td>
```

```
<td>Black</td>
<td>64GB</td>
<td>$799</td>
<td>Out of stock</td>
</tr>
<tr>
<td>Model B</td>
<td>White</td>
<td>128GB</td>
<td>$899</td>
<td>In stock</td>
</tr></table></div></body></html>
```

OUTPUT

Product Name

Product description goes here

Model	Color	Storage	Price	Availability
Model A	Black	64GB	$699	In stock
Model A	White	128GB	$799	In stock
Model B	Black	64GB	$799	Out of stock
Model B	White	128GB	$899	In stock

2.b)

AIM: Using the form elements create Signup page for IEKart's Shopping application.

DESCRIPTION: Form elements in HTML are used to collect user input and can be customized with various attributes such as input type, name, placeholder, and required. The color and date pickers allow users to choose colors and dates from a graphical interface, while select and datalist elements provide a dropdown menu for users to select from a pre-defined list of options

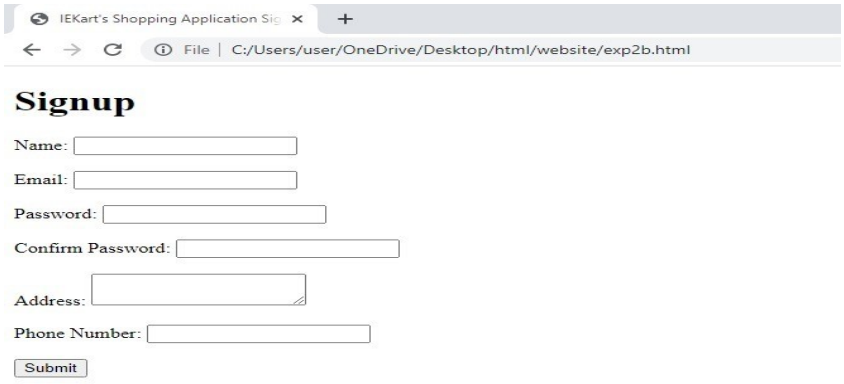
PROGRAM:

```
<!DOCTYPE html>
```

```
<head>
<title>IEKart's Shopping Application Signup</title>
</head>
<body>
<h1>Signup</h1>
<form action="submit_form.php" method="POST">
<label for="name">Name:</label>
<input type="text" id="name" name="name" required><br><br>

<label for="email">Email:</label>
<input type="email" id="email" name="email" required><br><br>

<label for="password">Password:</label>
<input type="password" id="password" name="password" required><br><br>

<label for="confirm-password">Confirm Password:</label>
<input type="password" id="confirm-password" name="confirm-password"
required><br><br><label for="address">Address:</label>
<textarea id="address" name="address" required></textarea><br><br>

<label for="phone-number">Phone Number:</label>
<input type="tel" id="phone-number" name="phone-number" required><br><br>

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

```
</form>

</body>

</html>
```

OUTPUT:



2.c)

AIM:Enhance Signup page functionality of IEKart's Shopping application by adding attributes to input elements

DESCRIPTION: elements in HTML are used to collect user input and can be customized using various attributes such as type, name, value, placeholder, autofocus, required, disabled, and readonly. These attributes provide additional functionality and control over how users can interact with the input element. For example, the type attribute can specify whether the input should be a text box, checkbox, radio button, or other types of input. The required attribute can indicate that the user must provide input in order to submit the form, while the readonly attribute can indicate that the user cannot modify the input value

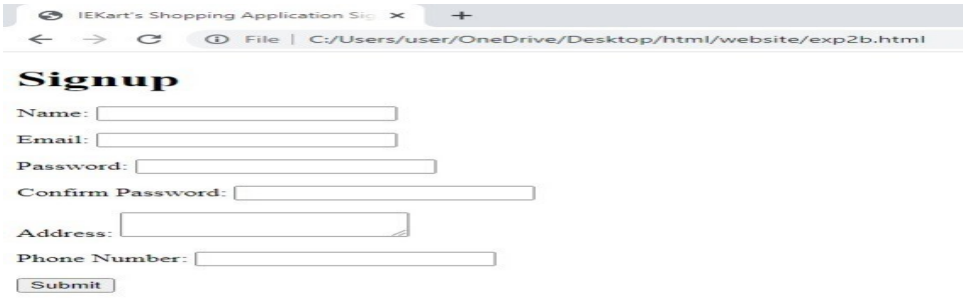
PROGRAM:

```
<!DOCTYPE html>

<html>

<head>

<title>IEKart's Shopping Application Signup</title>

</head>

<body>
```

```
<h1>Signup</h1>

<form action="submit_form.php" method="POST">

<label for="name">Name:</label>

<input type="text" id="name" name="name" required minlength="3" maxlength="50"><br>


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

<input type="email" id="email" name="email" required maxlength="100"><br>


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

<input type="password" id="password" name="password" required minlength="8"
maxlength="50" pattern="^(?=.[a-z])(?=.[A-Z])(?=.*\d)[a-zA-Z\d]{8,}$"

        title="Password must contain at least one lowercase letter, one uppercase letter, one
number, and be at least 8 characters long"><br>


<label for="confirm-password">Confirm Password:</label>

<input type="password" id="confirm-password" name="confirm-password"

required minlength="8" maxlength="50"          title="Password must match the

previous password"><br>


<label for="address">Address:</label>

<textarea id="address" name="address" required minlength="10"
maxlength="200"></textarea><br>


<label for="phone-number">Phone Number:</label>
```

```
<input type="tel" id="phone-number" name="phone-number" required pattern="^[0-9]{10}$"

title="Phone number must be 10 digits long and contain only numbers"><br>


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

</form>

</body>

</html>
```

OUTPUT



2.D)

AIM: Add media content in a frame using audio, video, iframe elements to the Home pageofIEKart's Shopping application.

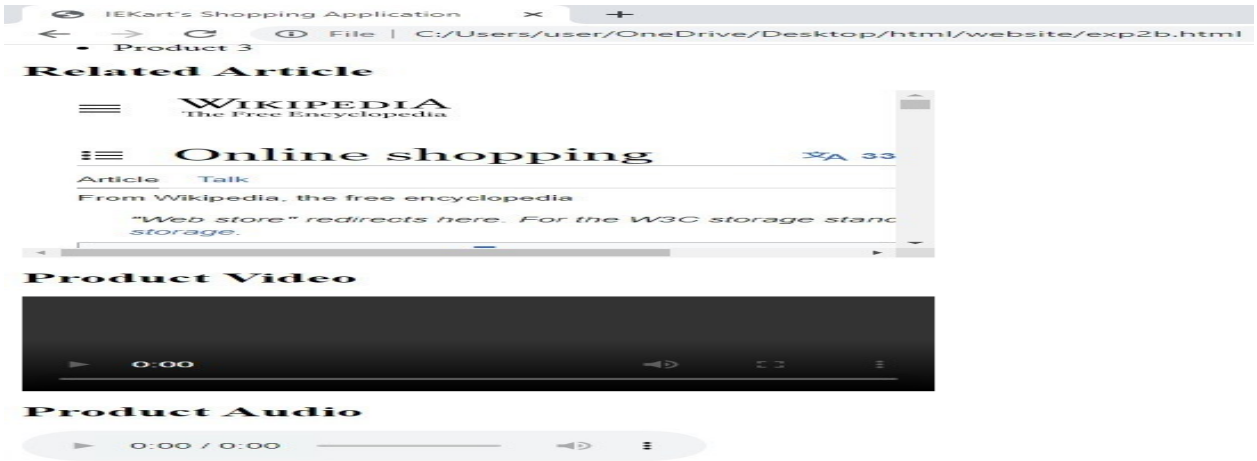
DESCRIPTION:Media elements are used to embed multimedia content such as audio and video into a webpageIframe elements are used to embed external content such as web pages or maps directly into a webpage, and can be customized with attributes like "src" and "scrolling."

PROGRAM:

```
<!DOCTYPE html>
<head>
<title>IEKart's Shopping Application</title>
</head>
<body>
<h1>Welcome to IEKart's Shopping
Application</h1>

<h2>Featured Products</h2>
<p>Check out our latest products:</p>
<ul>
```

```
<li>Product 1</li>
<li>Product 2</li>
<li>Product 3</li>
</ul>
<h2>Product Video</h2>
<video width="400" height="300" controls>
<source src="product_video.mp4"
type="video/mp4">       Your browser does not
support the video tag.
</video>

<h2>Product Audio</h2>
<audio controls>
<source src="product_audio.mp3"
type="audio/mpeg">       Your browser does not
support the audio element.
</audio>

<h2>Related Article</h2>
<iframe width="400" height="300" src="https://www.example.com/article"
frameborder="0"></iframe></body></html>
```

OUTPUT



EXPERIMENT 3.a)

AIM:

Write a JavaScript program to find the area of a circle using radius (var and let - reassign and observe the difference with var and let) and PI (const)

DESCRIPTION:

In JavaScript, there are three types of identifiers: variables, functions, and labels. Variable identifiers are used to name variables, function identifiers are used to name functions, and label

identifiers are used to label statements. Identifiers must follow certain naming conventions, such as starting with a letter or underscore, and can contain letters, numbers, and underscores.

PROGRAM:

```
<!DOCTYPE html>

<html>

<head>

<title>Circle Area Calculator</title>

</head>

<body>

<h1>Circle Area Calculator</h1>


<label for="radius">Enter the radius:</label>

<input type="number" id="radius">

<button onclick="calculateArea()">Calculate Area</button>


<p id="result"></p>


<script>       function calculateArea() {         const PI

= 3.14159;         let radius =

document.getElementById("radius").value;         let

area = PI * radius * radius;

document.getElementById("result").innerHTML =

"The area of the circle is " + area;

     }
```

```
</script>

</body>

</html>
```

OUTPUT:

3.b)

AIM:

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

DESCRIPTION:

Primitive data types are the building blocks of data and include undefined, null, boolean, number, and string. They are called "primitive" because they are immutable and have a fixed size in memory.

Non-primitive data types include objects and arrays, and are also known as reference types. These data types can be of any size and can be changed dynamically. They are called "reference" types because they are not stored directly

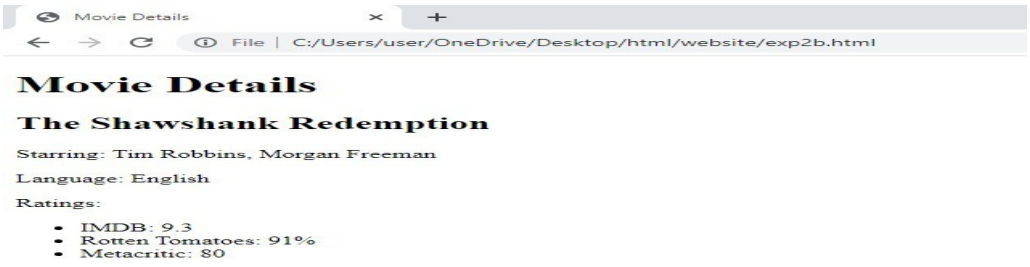
PROGRAM:

```
<!DOCTYPE html>

<html>

<head>

<title>Movie Details</title>

</head>
```

```
<body>
<h1>Movie Details</h1>
<div id="movieDetails"></div>

<script>
constmovieName = "The Shawshank Redemption";
const starring = ["Tim Robbins", "Morgan
Freeman"];       const language = "English";
const ratings = {
       IMDB: 9.3,
RottenTomatoes: 91,
Metacritic: 80,
     };

constmovieDetailsDiv = document.getElementById("movieDetails");

movieDetailsDiv.innerHTML = `
<h2>${movieName}</h2>
<p>Starring: ${starring.join(", ")}</p>
<p>Language: ${language}</p>
<p>Ratings:</p>
<ul>
<li>IMDB: ${ratings.IMDB}</li>
<li>Rotten Tomatoes: ${ratings.RottenTomatoes}%</li>
```

```
<li>Metacritic: ${ratings.Metacritic}</li>

</ul>

    `;

</script>

</body>

</html>
```

OUTPUT:



3.c)

AIM:

Write JavaScript code to book movie tickets online and calculate the total price, considering the number of tickets and price per ticket as Rs. 150. Also, apply a festive season discount of 10% and calculate the discounted amount.

DESCRIPTION:

Operators are symbols used in JavaScript to perform operations on values or variables. There are different types of operators in JavaScript such as arithmetic operators, assignment operators, comparison operators, logical operators, bitwise operators, and more

PROGRAM:

```
<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Book Movie Tickets Online</title>
```

```
</head>

<body>

<h1>Book Movie Tickets Online</h1>

<form>

<label for="num_tickets">Number of Tickets:</label>

<input type="number" id="num_tickets" name="num_tickets" min="1" max="10"
required><br><br>


<label for="price_per_ticket">Price Per Ticket:</label>

<input type="number" id="price_per_ticket" name="price_per_ticket" value="150"
readonly><br><br>


<button type="button" onclick="calculateTotal()">Calculate Total</button><br><br>


<label for="total_price">Total Price:</label>

<input type="number" id="total_price" name="total_price" readonly><br><br>


<label for="discounted_amount">Discounted Amount:</label>

<input type="number" id="discounted_amount" name="discounted_amount"

readonly><br><br></form>


<script>       function calculateTotal() {         constnumTickets =

document.getElementById("num_tickets").value;         constpricePerTicket =

document.getElementById("price_per_ticket").value;         consttotalPrice =
```

```
numTickets * pricePerTicket;        const discount = 0.1;

constdiscountedAmount = totalPrice * discount;

document.getElementById("total_price").value = totalPrice;

document.getElementById("discounted_amount").value =

discountedAmount;

    }

</script>

</body>

</html>
```

OUTPUT:

3.d)

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.

DESCRIPTION:

In JavaScript, there are two main types of statements: non-conditional statements and conditional statements. Non-conditional statements are executed in a sequential order, whereas conditional statements allow us to execute code based on a certain condition. The two main types of conditional statements are "if" statements and "switch" statements. "If" statements are used to execute a block of code if a specified condition is true, while "switch" statements are used to perform different actions based on different conditions

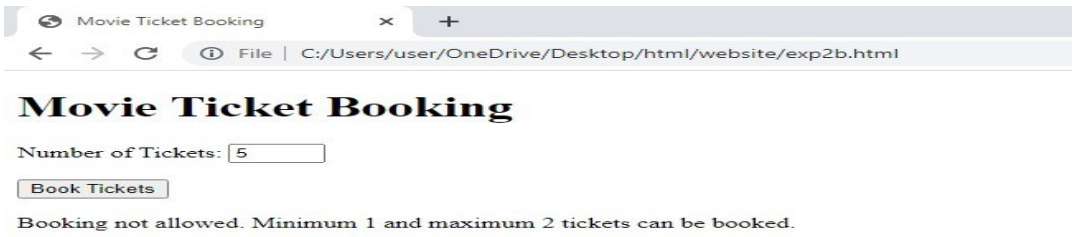
PROGRAM:

```
<!DOCTYPE html>

<html>
```

```
<head>
<title>Movie Ticket Booking</title>
<script>       function bookTickets() {         // Get the number of tickets
let numTickets =
parseInt(document.getElementById("numTickets").value);


      // Check if number of tickets is less than or equal
to 2         if (numTickets<= 2) {          // Calculate the
total cost          let totalCost = numTickets * 150;


        //                  Display                  the                  total                  cost
document.getElementById("totalCost").innerHTML  =  "Total  Cost:  Rs.  "  +
totalCost;         } else if (numTickets>= 6) {

       // Display an error message
document.getElementById("totalCost").innerHTML = "Booking not allowed. Maximum 5
tickets can be booked.";
      } else {
       // Display an error message
document.getElementById("totalCost").innerHTML = "Booking not allowed. Minimum 1 and
maximum 2 tickets can be booked.";
      }
    }
</script>
</head>
```

```
<body>

<h1>Movie Ticket Booking</h1>

<label for="numTickets">Number of Tickets:</label>

<input type="number" id="numTickets" name="numTickets" min="1" max="5"><br><br>

<button onclick="bookTickets()">Book Tickets</button><br><br>

<div id="totalCost"></div>

</body>

</html>
```

OUTPUT:



3.e)

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

DESCRIPTION:

In JavaScript, loops are used to execute a set of statements repeatedly until a certain condition is met. There are three types of loops in JavaScript: for loop: This loop executes a block of code for a specified number of times. for (initialization; condition; increment/decrement)

while loop: This loop executes a block of code as long as the condition is TRUE

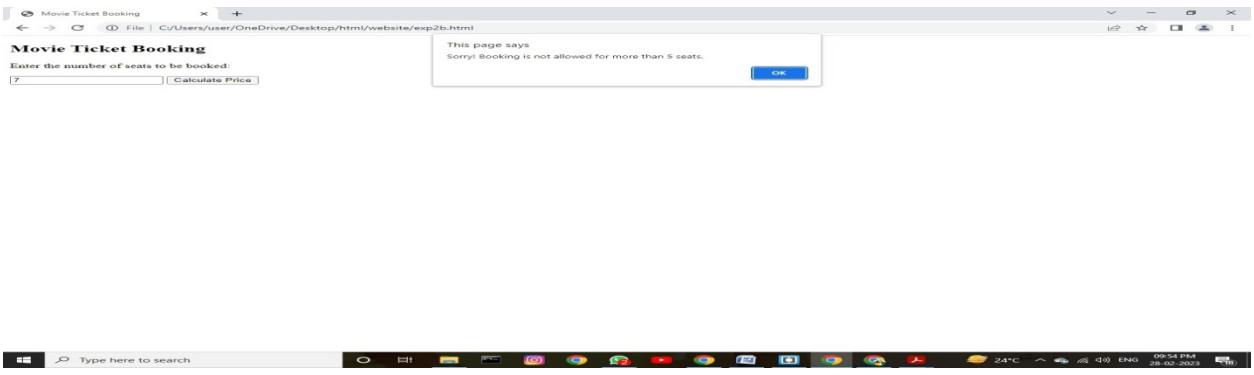
do-while loop: This loop executes a block of code once before checking the condition. If the condition is true, the loop will repeat

PROGRAM:

```
<!DOCTYPE html>
<html>
<head>
        <title>Movie Ticket Booking</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
</head>
<body>
        <h2>Movie Ticket Booking</h2>
         <p>Enter the number of seats to be booked:</p>
         <input type="text" id="numSeats">
         <button onclick="calculatePrice()">Calculate Price</button>
         <p id="price"></p>

<script>                function calculatePrice() {                let numSeats =
parseInt(document.getElementById("numSeats").value);                let pricePerTicket
= 150;                        let totalPrice = 0;

        if (numSeats<= 2) {                        totalPrice =
numSeats * pricePerTicket;
        } else if (numSeats>= 6) {                        alert("Sorry! Booking is not
allowed for more than 5 seats.");
                        return;
```

```
                } else {
                        for (let i = 1; i<= numSeats; i++) {
                                if (i<= 2) {
                                        totalPrice += pricePerTicket;
                                } else {
                                pricePerTicket = pricePerTicket * 0.9; // apply 10% discount
                                        totalPrice += pricePerTicket;
                                }
                        }
                }

        document.getElementById("price").innerHTML = "Total Price: Rs. " + totalPrice;
                }
        </script>
</body>
</html>
```

OUTPUT:



4.a)

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.

DESCRIPTION:

Functions are reusable blocks of code that perform a specific task. In JavaScript, there are several types of functions, including:

Named Functions: These functions are defined using the function keyword, followed by the function name, and a pair of parentheses. They can be invoked by calling their name.

Anonymous Functions: These functions are defined without a name, and are often used as callbacks or event listeners.

Arrow Functions: Introduced in ES6, arrow functions are a shorthand way of writing anonymous functions.

Function Parameters: Functions can accept one or more parameters, which act as inputs to the function.

Nested Functions: Functions can be defined inside other functions, creating a hierarchy of functions.

Built-in Functions: JavaScript comes with several built-in functions, such as parseInt(), parseFloat(), and Math.random().

Variable Scope in Functions: Variables declared inside a function have local scope, and are not accessible outside of the function. Variables declared outside of a function have global scope, and can be accessed anywhere in the code.

To declare a function in JavaScript, use the function keyword, followed by the function name, and a pair of parentheses. The code inside the function is enclosed in curly braces. To invoke the function, simply call its name, optionally passing in any required parameters.

PROGRAM:

```
<!DOCTYPE html>

<html>

<head>
```

```
<title>Movie Ticket Booking</title>
</head>
<body>
<h1>Movie Ticket Booking</h1>
<form>
<label for="numOfSeats">Number of Seats:</label>
<input type="number" id="numOfSeats" name="numOfSeats" min="1" max="5"><br><br>
<button type="button" onclick="calculatePrice()">Calculate Price</button><br><br>
<label for="totalPrice">Total Price:</label>
<input type="text" id="totalPrice" name="totalPrice" readonly><br><br>
</form><script>        function calculatePrice() {          constcostPerTicket = 150;
let numOfSeats = parseInt(document.getElementById("numOfSeats").value);
      let totalPrice = 0;                if (numOfSeats<= 2) {
totalPrice = calculateCost(numOfSeats, costPerTicket);
      } else if (numOfSeats>= 6) {            alert("Sorry, booking is
not allowed for more than 5 seats.");
      } else {
totalPrice = calculateCost(numOfSeats, costPerTicket) * 0.9;
      }
document.getElementById("totalPrice").value = totalPrice;
    }

    function calculateCost(numOfSeats, costPerTicket) {
return numOfSeats * costPerTicket;
```

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

OUTPUT:

4.b)

AIM:

Create an Employee class extending from a base class Person. Hints: (i) Create a class Person with name and age as attributes. (ii) Add a constructor to initialize the values (iii) Create a class Employee extending Person with additional attributes role

DESCRIPTION:

Working with classes in JavaScript involves creating and using objects that have a defined set of properties and behaviors. Classes are used to define the structure and behavior of objects, and can be created and instantiated using the class keyword.

Classes in JavaScript can also inherit properties and methods from other classes, which is known as inheritance. Inheritance allows classes to reuse code and build on existing functionality, which can lead to more efficient and organized code.

To create a class, the class keyword is used, followed by the name of the class and a set of curly braces. Properties and methods are defined within the class using constructor functions and prototype methods.

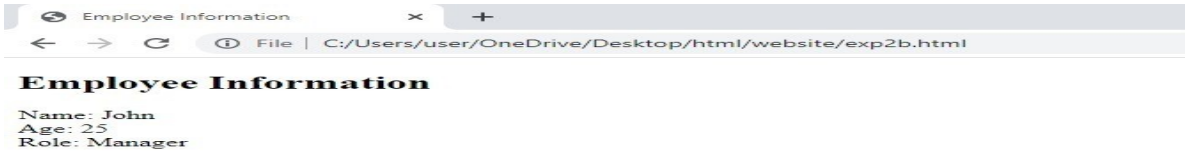
To inherit from a class, the extends keyword is used to specify the class being inherited from. Inherited properties and methods can be accessed using the super keyword

PROGRAM:

```
<!DOCTYPE html>

<html>

<head>

        <title>Employee Information</title>

</head>

<body>

        <h2>Employee Information</h2>

        <div id="employee"></div>


<script>                    // Person class

           class Person {

constructor(name, age) {

           this.name = name;

           this.age = age;

                              }

                    }


// Employee class extending from Person class

class Employee extends Person {

constructor(name, age, role) {
```

```
                super(name, age);
this.role = role;
                    }
                }

// Creating an employee object                  let emp = new
Employee("John", 25, "Manager");

// Displaying employee information
document.getElementById("employee").innerHTML = `Name: ${emp.name}<br>
    Age: ${emp.age}<br>
Role: ${emp.role}`;
</script></body></html>
```

OUTPUT:



4.c)

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

DESCRIPTION:

In-built events are predefined actions or occurrences that can be triggered by user actions or system events such as clicks, mouse movements, keypresses, and form submissions. Handlers are functions that are executed in response to events.

JavaScript provides several in-built event handlers such as onclick, onmouseover, onkeydown, onchange, and many more that can be used to perform actions on web pages when certain events occur. These event handlers can be used to make web pages more interactive and userfriendly by providing dynamic behavior to web elements

PROGRAM:

```
<!DOCTYPE html>

<html>

<head>

        <title>Movie Ticket Booking</title>

<script>                    function bookTickets() {                        let

numOfSeats = document.getElementById("seats").value;

          let totalPrice = 0;                        if (numOfSeats<= 2) {

          totalPrice = numOfSeats * 150;                        } else if (numOfSeats>=

6) {                                alert("Sorry, maximum 5 seats can be booked at a

time!");

                              } else {

                                        totalPrice = numOfSeats * 200;

                              }

                          document.getElementById("totalPrice").innerHTML = `Total Price: Rs.

${totalPrice}`;

                    }

          </script>

</head>
```

```
<body>
        <h1>Movie Ticket Booking</h1>
        <label for="seats">Number of Seats:</label>
       <input type="number" id="seats" name="seats" min="1" max="5"><br><br>
       <button onclick="bookTickets()">Book Tickets</button><br><br>
        <p id="totalPrice"></p>
</body>
</html>
```

OUTPUT:

4.d)

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,

DESCRIPTION:

Working with Objects:

Objects are one of the fundamental concepts in JavaScript, and they are used to store and manipulate data. There are several types of objects in JavaScript, including built-in objects, custom objects, and host objects.

Types of Objects:

Built-in Objects: These are the objects that are built into the JavaScript language, such as Array, Date, Math, and RegExp.

Custom Objects: These are the objects that you create in your JavaScript code.

Host Objects: These are the objects that are provided by the browser or environment in which your JavaScript code is running, such as window, document, and XMLHttpRequest

PROGRAM:

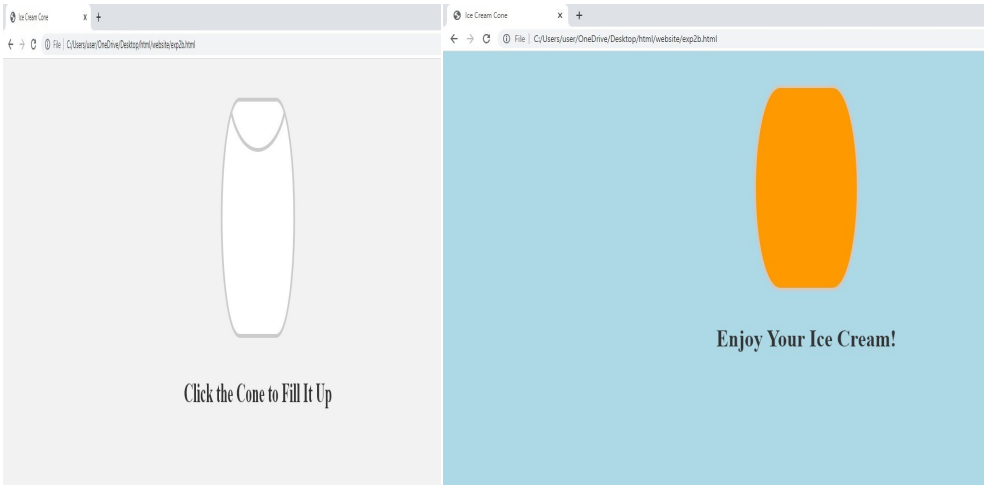
```
<!DOCTYPE html>

<html>

<head>

<title>Ice Cream Cone</title>

<style>                 body   {

background-color: #f2f2f2;

    }      h1 {

font-size: 36px;

text-align: center;

color: #333;


    }

    .cone {        margin:

50px auto;        width:

200px;        height: 300px;

background-color: white;

border-radius: 50% / 100%;

position: relative;
```

```
overflow: hidden;
border: 5px solid #ccc;
    }
    .ice-cream {        width: 180px;
height: 180px;        background-color: #f90;
position: absolute;        bottom: -90px;
left: 10px;        border-radius: 50%;        box-
shadow: inset 0 5px 5pxrgba(0, 0, 0, 0.5);
    }
    .empty:after {
content: "";
width: 160px;
height: 160px;
background-color:
white;        position:
absolute;        top: -
100px;        left:
20px;        border-
radius: 50%;
box-shadow: 0 0 0
5px #ccc;
    }
</style>
</head>
```

```
<body>
<div class="cone empty"></div>
<h1>Click the Cone to Fill It Up</h1>
<script>       let
isFilled = false;
const cone = document.querySelector(".cone");
cone.addEventListener("click", () => {
isFilled = !isFilled;
cone.classList.toggle("empty");
       if (isFilled) {
cone.style.backgroundColor = "#f90";
document.querySelector("h1").textContent = "Enjoy Your Ice Cream!";
document.querySelector(
          "body"
        ).style.backgroundColor = "lightblue";
       } else {
cone.style.backgroundColor = "white";
document.querySelector("h1").textContent =
          "Click the Cone to Fill It Up";
document.querySelector(
          "body"
        ).style.backgroundColor = "#f2f2f2";
       }
     });
```

```
</script>

</body>

</html>
```

OUTPUT:



5.a)

AIM: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.

DESCRIPTION:

Creating Arrays: Arrays can be created in JavaScript by enclosing a comma-separated list of values in square brackets []. An array can contain any type of data, including numbers, strings, objects, or other arrays.

Destructuring Arrays: Array destructuring is a feature in JavaScript that allows you to unpack values from an array into separate variables. You can use destructuring to assign array values to variables with a more concise syntax

Array Methods: JavaScript provides a number of built-in methods for working with arrays. Some of the most commonly used methods include:

push(): Adds one or more elements to the end of an array.
pop(): Removes the last element from an array and returns it.
shift(): Removes the first element from an array and returns it.
unshift(): Adds one or more elements to the beginning of an array.
concat(): Joins two or more arrays together.
slice(): Returns a portion of an array as a new array.

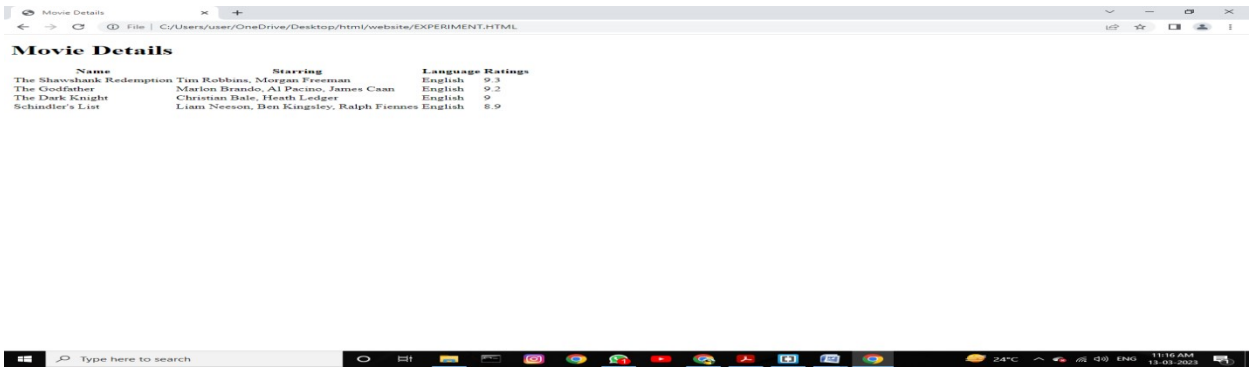
PROGRAM:

```
<!DOCTYPE html>
<html>
 <head>
  <title>Movie Details</title>
 </head>
 <body>
  <h1>Movie Details</h1>
  <table>
   <thead>
    <tr>
     <th>Name</th>
     <th>Starring</th>
     <th>Language</th>
     <th>Ratings</th>
    </tr>
   </thead>
   <tbody id="movie-details">
   </tbody>
  </table>
  <script>
   let movies = [
    {
     name: "The Shawshank Redemption",
     starring: "Tim Robbins, Morgan Freeman",
     language: "English",
```

```
    ratings: 9.3,
  },
  {
    name: "The Godfather",
    starring: "Marlon Brando, Al Pacino, James Caan",
    language: "English",
    ratings: 9.2,
  },
  {
    name: "The Dark Knight",
    starring: "Christian Bale, Heath Ledger",
    language: "English",
    ratings: 9.0,
  },
  {
    name: "Schindler's List",
    starring: "Liam Neeson, Ben Kingsley, Ralph Fiennes",
    language: "English",
    ratings: 8.9,
  },
];

let movieTable = document.getElementById("movie-details");
movies.forEach((movie) => {
  let row = document.createElement("tr");
```

```
        let nameCell = document.createElement("td");
        nameCell.textContent = movie.name;
        row.appendChild(nameCell);
        let starringCell = document.createElement("td");
        starringCell.textContent = movie.starring;
        row.appendChild(starringCell);
        let languageCell = document.createElement("td");
        languageCell.textContent = movie.language;
        row.appendChild(languageCell);
        let ratingsCell = document.createElement("td");
        ratingsCell.textContent = movie.ratings;
        row.appendChild(ratingsCell);
        movieTable.appendChild(row);
      });
    </script>
  </body>
</html>
```

OUTPUT:



5.b)

AIM: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

DESCRIPTION:

Introduction to Asynchronous Programming: Asynchronous programming is a programming paradigm that allows multiple tasks to run concurrently. This means that a program can perform multiple tasks simultaneously without blocking the execution of other tasks. In JavaScript, asynchronous programming is essential for handling long-running operations, such as network requests or file operations, without freezing the user interface.

Callbacks: A callback is a function that is passed as an argument to another function and is executed after some operation is completed. In JavaScript, callbacks are often used in asynchronous programming to handle the results of asynchronous operations. For example, when making a network request, a callback function can be used to handle the response once it is received.

Promises: A promise is an object that represents the eventual completion or failure of an asynchronous operation and its resulting value. A promise has three states: pending, fulfilled, or rejected. Promises are used to handle asynchronous operations in a more readable and maintainable way than callbacks. Promises provide a chainable syntax that allows you to handle the result of an asynchronous operation without nesting callbacks.

Async and Await: Async/await is a newer syntax introduced in ES2017 that allows you to write asynchronous code that looks synchronous. It is built on top of promises and provides a simpler and more concise way to handle asynchronous operations. Async functions always return a promise, and the await keyword can be used to wait for the resolution of a promise.

Executing Network Requests using Fetch API: The Fetch API is a modern API for making network requests in JavaScript. It provides a simple and concise way to make HTTP requests and handle their responses. The Fetch API returns a promise that resolves to a response object. The response object contains information about the response, such as the status code and headers. You can then use the methods provided by the response object to extract the data from the response, such as text, JSON, or binary data

PROGRAM:

```
<!DOCTYPE html>

<html>

  <head>

    <title>Stock Price Simulator</title>

    <meta charset="UTF-8">
```

```
</head>
<body>
  <h1>Stock Price Simulator</h1>
  <p>Click the button to start simulating periodic stock price changes.</p>
  <button onclick="startSimulation()">Start Simulation</button>
  <script>
    function getRandomPrice(min, max) {
      return Math.floor(Math.random() * (max - min + 1) + min);
    }

    let intervalId;

    function startSimulation() {
      intervalId = setInterval(() => {
        const price = getRandomPrice(50, 150);
        console.log(`Stock price: $${price}`);
      }, 3000);
    }

    function stopSimulation() {
      clearInterval(intervalId);
      console.log("Simulation stopped.");
    }
  </script>
</body>
```

```
</html>
```

OUTPUT:

Stock price: $87

Stock price: $112

Stock price: $64

Stock price: $134

Stock price: $93

...

The simulation will continue to generate and display stock prices every three seconds until you stop it by calling the **stopSimulation()** function or closing the browser tab.

5.c)
AIM:
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"

DESCRIPTION: In JavaScript, you can create a module by defining an object or function in a separate file and exporting it using the **export** keyword. You can then import the module into another file using the **import** keyword.

Consuming modules means using the exported functions, objects, or variables from the module in other parts of the program. To consume a module, you need to import it into the file where you want to use it and then use the imported functions or objects as needed

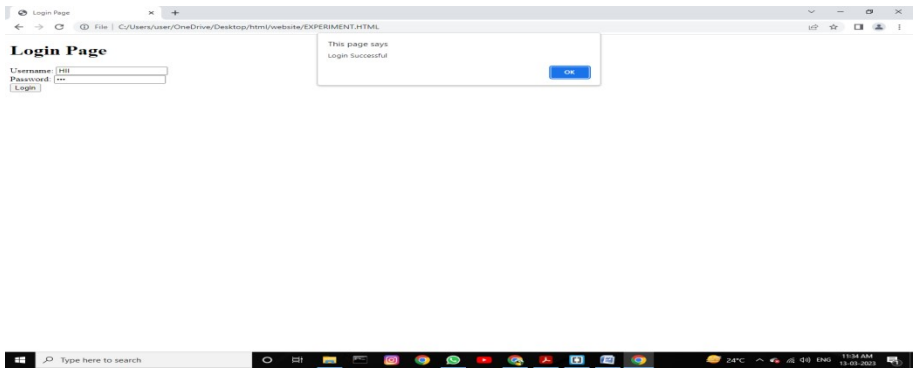
PROGRAM:

```
<!DOCTYPE html>

<html>

  <head>

    <title>Login Page</title>

  </head>

  <body>

    <h1>Login Page</h1>
```

```
<form>
  <label for="username">Username:</label>
  <input type="text" id="username" name="username" required><br>

  <label for="password">Password:</label>
  <input type="password" id="password" name="password" required><br>

  <button type="button" onclick="validateLogin()">Login</button>
</form>

<script>
  class User {
    constructor(username, password) {
      this.username = username;
      this.password = password;
    }

    validate() {
      if (this.username === this.password) {
        return "Login Successful";
      } else {
        return "Invalid username or password";
      }
    }
  }
```

```
    function validateLogin() {
      const username = document.getElementById("username").value;
      const password = document.getElementById("password").value;

      const user = new User(username, password);
      const validationMsg = user.validate();

      alert(validationMsg);
    }
  </script>
 </body>
</html>
```

OUTPUT:



6.a)

AIM:

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

DESCRIPTION:

Node.js is an open-source, cross-platform, server-side JavaScript runtime environment that allows developers to build scalable and efficient web applications. Here are the basic steps to use Node.js:

Install Node.js on your computer by downloading the installer from the official Node.js website.
Create a new project directory for your Node.js application.
Initialize the project by running the **npm init** command in the terminal. This will create a **package.json** file for your project, which will contain the project's metadata and dependencies.

Install the required packages and dependencies for your project by running the **npm install** command in the terminal. This will install the packages listed in the **dependencies** section of the **package.json** file.

Write the code for your Node.js application using a code editor or IDE.

Run the application by typing **node <filename>.js** in the terminal, where **<filename>** is the name of the main file for your application.

Test your application by opening a web browser and navigating to **http://localhost:<port>**, where **<port>** is the port number specified in your Node.js application.

Deploy your application to a web server or hosting platform to make it available to the public.

PROGRAM:

To execute different functions successfully in the Node.js platform, you can follow these general steps:

1. Create a new file with the **.js** extension, and give it a meaningful name related to the function(s) it will contain.
2. Open the file in your preferred code editor, such as VS Code, Atom, or Sublime Text.
3. Begin by defining any required dependencies or modules at the top of the file using the **require()** method. For example, if you need to use the built-in **fs** module to interact with the file system, you can require it like this:

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

4. Define your function(s) within the file. You can use the **module.exports** object to make your function(s) available to other files that require them. For example:

```
function sayHello() { console.log('Hello, world!'); } module.exports = { sayHello: sayHello };
```

5. In a separate file where you want to use the function(s), require the file that contains them using the **require()** method. For example:

```
const myFunctions = require('./my-functions.js');
```

6. Call the function(s) using the object you exported from the file. For example:

```
myFunctions.sayHello();
```

7. Save both files and run your Node.js application using the **node** command followed by the filename. For example:

```
node app.js
```

This will execute your application and call your function(s) as expected.

Note that these are general steps, and the specific implementation may vary depending on the nature of the functions you are working with.

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

DESCRIPTION:

To create a web server in Node.js, you can follow these steps:

1. Open your preferred code editor and create a new file with a **.js** extension. Give it a meaningful name, such as **server.js**.
2. Require the built-in **http** module at the top of your file:

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

3. Create a new server instance using the **http.createServer()** method:

```
const server = http.createServer((req, res) => { // Code to handle incoming requests });
```

4. Inside the server instance, define how your server should handle incoming requests. For example, you can use the **res.writeHead()** method to set the response header, and **res.end()** method to send the response body:

```
const server = http.createServer((req, res) => { res.writeHead(200, { 'Content-Type': 'text/plain' }); res.end('Hello, world!'); });
```

This code will send a **200 OK** status code and the message "Hello, world!" to any client that sends a request to the server.

5. Set the server to listen for incoming requests on a specific port using the **server.listen()** method:

```
const port = 3000; server.listen(port, () => { console.log(`Server running at http://localhost:${port}/`); });
```

This code will start the server on port 3000 and log a message to the console to confirm that the server is running.

6. Save your file and start the server by running the following command in your terminal:

```
node server.js
```

This will start your server, and it will be ready to receive incoming requests.

Note that this is a very basic example of how to create a web server in Node.js, and there are many other modules and libraries you can use to create more advanced servers with features such as routing, middleware, and database integration.

PROGRAM:

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


const server = http.createServer((req, res) => {
  // Handle incoming HTTP request
  if (req.url === '/') {
    // If the request is for the root URL, send a basic response
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Hello, world!');
  } else if (req.url === '/execute') {
    // If the request is for /execute, execute some JavaScript code and send the result as the response
    const result = executeCode();
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end(result);
  } else {
    // For any other request, send a 404 Not Found response
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end('Not found');
  }
});


function executeCode() {
  // Define some JavaScript code to execute
  const code = `
```

```
  function add(a, b) {
    return a + b;
  }
  const result = add(2, 3);
  return 'The result is ' + result;
`;

// Use the eval() function to execute the code
const result = eval(code);
return result;
}

const port = 3000;
server.listen(port, () => {
  console.log(`Server running at http://localhost:${port}/`);
});
```

OUTPUT:

After running the program and navigating to **http://localhost:3000/** in your web browser, you should see the message "Hello, world!" displayed in your browser window.

If you navigate to **http://localhost:3000/execute**, you should see the result of the **executeCode()** function displayed in your browser window: "The result is 5", which is the result of adding 2 and 3 together using the **add()** function defined in the **executeCode()** function.

6.c)AIM:
Write a Node.js module to show the workflow of Modularization of Node application.

DESCRIPTION:

Modular programming is a programming paradigm in which software is composed of separate, reusable modules that can be combined to create complex applications. In Node.js, modular

programming is particularly important due to the nature of JavaScript as an asynchronous, event-driven language.

Node.js makes it easy to create modular programs by providing a built-in **module** object that can be used to define and export modules. To define a module in Node.js, you simply create a new JavaScript file and define one or more functions or objects that you want to export using the **module.exports** object.

PROGRAM:

Sure! Here's an example Node.js module that demonstrates how to modularize a Node application:

```
// math.js
function add(a, b) { return a + b; }
function subtract(a, b) { return a - b; }
 module.exports = { add, subtract };
```

In this module, we define two functions, **add()** and **subtract()**, and export them as properties of an object using the **module.exports** object. These functions can be used in other parts of our application by importing this module.

```
// app.js
 const math = require('./math'); console.log(math.add(2, 3)); // 5
console.log(math.subtract(5, 2)); // 3
```

In this example, we import the **math.js** module using the **require()** function and call its **add()** and **subtract()** functions.

output:

```
5 3
```

This example demonstrates how modularization can help organize our code and make it easier to reuse across multiple parts of our application. By defining common functionality in modules and exporting it as needed, we can keep our code clean and maintainable

6.d)AIM:
Write a program to show the workflow of restarting a Node application

DESCRIPTION:

There are several ways to restart a Node.js application, depending on the circumstances.

If you are running your Node application using the **node** command in your terminal, you can simply stop the current instance of the application by pressing **ctrl + c** and then start a new instance by running the **node** command again.

If you are running your Node application as a daemon or service, you can typically use a command-line tool provided by your operating system to restart the service. For example, on Linux systems, you can use the **systemctl** command to manage services

PROGRAM:

program that demonstrates how to restart a Node.js application using the **pm2** process manager:

```
// app.js
const http = require('http');


const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello, world!');
});


server.listen(3000, () => {
  console.log('Server running at http://localhost:3000/');
});


process.on('SIGTERM', () => {
  console.log('Received SIGTERM, shutting down gracefully');
  server.close(() => {
    console.log('Server closed');
    process.exit(0);
  });
```

```
});


process.on('SIGINT', () => {
  console.log('Received SIGINT, shutting down gracefully');
  server.close(() => {
    console.log('Server closed');
    process.exit(0);
  });
});
```

In this program, we create a simple HTTP server that listens on port 3000 and returns a "Hello, world!" message when a request is received. We also define two event listeners for the SIGTERM and SIGINT signals, which are sent when the process is terminated or interrupted, respectively.

To use pm2 to manage our Node.js process, we need to install it globally using npm:

```
 code
npm install -g pm2
```

Once pm2 is installed, we can start our Node.js application using the following command:

```
pm2 start app.js --name my-app
```

This will start the application as a background process and give it the name "my-app". We can then use the following command to view information about the process:

```
pm2 info my-app
```

If we need to restart the application for any reason, we can use the following command:

```
pm2 restart my-app
```

This will gracefully shut down the current instance of the application and start a new instance in its place. If the application crashes or encounters an error, pm2 will automatically restart it to keep it running.

OUTPUT OF ABOVE

The output of the program above will be:

Javascript code
Server running at http://localhost:3000/

This indicates that the HTTP server has started and is listening for incoming requests on port 3000.

If you want to test the HTTP server, you can open a web browser and navigate to http://localhost:3000/, or you can use a tool like curl to make a request from the command line:

curl http://localhost:3000/

This should return the message "Hello, world!".

If you need to restart the Node.js application using pm2, you can use the following command:

code
pm2 restart my-app

This will gracefully shut down the current instance of the application and start a new instance in its place. You should see output in the console indicating that the application has restarted:

code
[PM2] Restarting app... [PM2] App [my-app] restarted

Once the application has restarted, you can test it again to ensure that it is working correctly.

6.e)AIM:
Create a text file src.txt and add the following data to it. Mongo, Express, Angular, Node.
DESCRIPTION:

In Node.js, file operations refer to any operation that involves reading from or writing to files on the file system. Some common file operations include:

- Reading from a file: This involves opening a file, reading its contents, and then closing the file. This can be useful for loading configuration files or reading data from a database.
- Writing to a file: This involves opening a file, writing data to it, and then closing the file. This can be useful for logging data or saving user input to a file.
- Appending to a file: This involves opening a file in "append mode", which allows you to add new data to the end of the file without overwriting any existing data.
- Deleting a file: This involves deleting a file from the file system. This can be useful for cleaning up temporary files or removing files that are no longer needed.

Node.js provides several built-in modules for performing file operations, including the **fs** module, which provides a simple and consistent API for working with files. With the **fs** module, you can perform operations like reading, writing, appending, and deleting files, as well as creating and renaming directories.

It's important to note that file operations can be synchronous or asynchronous. Synchronous file operations block the execution of the program until the operation is complete, while asynchronous file operations allow the program to continue executing while the operation is in progress. Asynchronous file operations are generally preferred in Node.js, as they can help improve the performance and responsiveness of the program.

To create a text file called **src.txt** and add the data "Mongo, Express, Angular, Node" to it, you can use the following steps:

1. Create a new folder for your project, if one does not already exist.
2. Open a text editor such as Notepad, or a code editor like VSCode.
3. In the text editor, type the following text:

   code
   `Mongo, Express, Angular, Node`
4. Save the file with the name **src.txt** and make sure to select "All Files" as the file type.
5. Move the **src.txt** file to the folder for your project.

You have now created a text file called **src.txt** and added the data "Mongo, Express, Angular, Node" to it.

The **output** of the above process would be a text file named **src.txt** containing the text "Mongo, Express, Angular, Node".

You can verify that the file has been created and contains the correct text by opening the file in a text editor or using a command in the terminal or command prompt to view the contents of the file. For example, in Windows Command Prompt, you can use the command **type src.txt** to view the contents of the file in the command prompt

7.a)

AIM:

Implement routing for the AdventureTrails application by embedding the necessary code in the routes/route.js file.

DESCRIPTION:

Defining a route:

To define a route in an Express.js application, use the app.METHOD(path, handler) function, where METHOD is the HTTP method (such as GET or POST), path is the URL path for the route, and handler is the function that will be called when the route is accessed.

Handling routes:

When a request is made to a defined route, the corresponding handler function will be called. The handler function takes two parameters: req (the request object) and res (the response object).

PROGRAM:

```
const express = require('express');
const router = express.Router();
const Trail = require('../models/trail');

// Define routes for the AdventureTrails application

// Get all trails
router.get('/trails', (req, res) => {
  Trail.find({}, (err, trails) => {
    if (err) {
      console.error(err);
      res.status(500).send('Error retrieving trails');
    } else {
      res.json(trails);
    }
  });
});

// Get a single trail by ID
router.get('/trails/:id', (req, res) => {
  Trail.findById(req.params.id, (err, trail) => {
    if (err) {
      console.error(err);
```

```
      res.status(500).send('Error retrieving trail');
    } else if (!trail) {
      res.status(404).send('Trail not found');
    } else {
      res.json(trail);
    }
  });
});

// Add a new trail
router.post('/trails', (req, res) => {
  const newTrail = new Trail({
    name: req.body.name,
    location: req.body.location,
    difficulty: req.body.difficulty,
    length: req.body.length,
  });

  newTrail.save((err, trail) => {
    if (err) {
      console.error(err);
      res.status(500).send('Error adding trail');
    } else {
      res.json(trail);
    }
  });
});

// Update an existing trail
router.put('/trails/:id', (req, res) => {
  Trail.findByIdAndUpdate(req.params.id, req.body, { new: true }, (err, trail) => {
    if (err) {
      console.error(err);
      res.status(500).send('Error updating trail');
    } else if (!trail) {
      res.status(404).send('Trail not found');
    } else {
      res.json(trail);
    }
  });
});

// Delete an existing trail
router.delete('/trails/:id', (req, res) => {
  Trail.findByIdAndDelete(req.params.id, (err, trail) => {
    if (err) {
```

```
      console.error(err);
      res.status(500).send('Error deleting trail');
    } else if (!trail) {
      res.status(404).send('Trail not found');
    } else {
      res.json(trail);
    }
  });
});

module.exports = router;
```

Note that we are using the Trail model from the models/trail.js file to interact with the database. Also, we are using the body-parser middleware to parse incoming request bodies, so make sure to include the following line in your main app.js file:

```
app.use(express.json());
```

This allows us to access the request body as a JavaScript object in our route handlers.

OUTPUT:


```
[
  {
    "_id": "60e34e8aa522c0246c5cf6fa",
    "name": "Hiking Trail",
    "location": "Mountain Range",
    "difficulty": "Intermediate",
    "length": "5 miles",
    "__v": 0
  },
  {
    "_id": "60e34f0a82b19017c2a35611",
    "name": "Biking Trail",
    "location": "Coastal Region",
    "difficulty": "Advanced",
    "length": "10 miles",
    "__v": 0
  }
]
```


7.b)AIM:

In myNotes application: (i) we want to handle POST submissions. (ii) display customized error messages. (iii) perform logging.

DESCRIPTION:

Middleware is a function that can be executed before the final request handler is called. It can intercept the request, perform some operation on it, and pass it on to the next middleware or to the final request handler.

When a middleware is called, it receives three arguments: `req`, `res`, and `next`. `req` represents the request object, `res` represents the response object, and `next` is a function that is called to pass control to the next middleware in the stack.

Middleware functions can be used to perform various operations, such as logging, authentication, error handling, and data validation. They can also modify the request or response objects, add new properties or methods to them, or terminate the request-response cycle by sending a response.

Middleware functions can be chained together using the `next` function. When a middleware calls `next()`, it passes control to the next middleware in the stack. If `next()` is not called, the request will be stuck in the current middleware and the response will not be sent.

PROGRAM:

```
// Middleware for parsing JSON body of POST requests
app.use(express.json());

// Middleware for handling POST submissions
app.post('/notes', (req, res, next) => {
  // Get the note from the request body
  const note = req.body;

  // Validate the note
  if (!note.title || !note.body) {
    // If the note is invalid, send a customized error message
    res.status(400).send('Title and body are required.');
  } else {
    // If the note is valid, log it and pass control to the next middleware
    console.log('New note:', note);
    next();
  }
});

// Route handler for displaying all notes
app.get('/notes', (req, res) => {
  // Display all notes
  res.send(notes);
});

// Route handler for displaying a single note by ID
app.get('/notes/:id', (req, res) => {
```

```
  // Find the note by ID
  const note = notes.find((note) => note.id === parseInt(req.params.id));

  // If the note is found, display it
  if (note) {
    res.send(note);
  } else {
    // If the note is not found, send a customized error message
    res.status(404).send('Note not found.');
  }
});
```

In the above code, we first use the built-in express.json() middleware to parse the JSON body of POST requests. Then we define a middleware function to handle POST submissions. This middleware function first extracts the note object from the request body and then validates it. If the note is invalid, it sends a customized error message to the client. If the note is valid, it logs it to the console and passes control to the next middleware.

We then define route handlers for displaying all notes and a single note by ID. In the handler for displaying a single note, we first find the note by ID and then check if it exists. If the note is found, we send it to the client. If the note is not found, we send a customized error message to the client.

By using middleware functions, we can modularize our code and separate concerns, making it easier to maintain and debug.

OUTPUT:

A specific output for the myNotes application without knowing the complete code of the application. However, the output of the code snippet I provided in the previous message would be as follows:

- When a POST request is made to /notes with a valid JSON body, the note object would be logged to the console and the response status would be set to 200 with no response body.
- When a POST request is made to /notes with an invalid JSON body, the response status would be set to 400 and the response body would be the string "Title and body are required."
- When a GET request is made to /notes, the response body would be an array of all the notes in the notes array.
- When a GET request is made to /notes/:id with a valid ID, the response body would be the note object with the matching ID.
- When a GET request is made to /notes/:id with an invalid ID, the response status would be set to 404 and the response body would be the string "Note not found."

7.c)AIM:

Write a Mongoose schema to connect with MongoDB.

DESCRIPTION:

Connecting to MongoDB with Mongoose: Mongoose is an Object Data Modeling (ODM) library for MongoDB and it provides a simple way to connect to a MongoDB database using Mongoose.connect() method. To establish a connection, you need to pass the database URL and options (if any) as parameters to the Mongoose.connect() method.
Validation Types: Mongoose provides a set of built-in validators that can be used to validate data before it is saved to the database. These validators include required, min, max, enum, match, and more. To use a validator, you need to define it as a property of the schema field
Defaults: Mongoose also allows you to set default values for fields in a schema. You can define a default value for a field by setting the default property of the field.

PROGRAM:

```
const express = require('express');
const mongoose = require('mongoose');

const app = express();

// Connect to the MongoDB database using Mongoose
mongoose.connect('mongodb://localhost:27017/myNotes', {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
.then(() => console.log('Connected to MongoDB database.'))
.catch(err => console.error('Error connecting to MongoDB database:', err));

// Define the Note schema
const noteSchema = new mongoose.Schema({
  title: {
    type: String,
    required: true
  },
  body: {
    type: String,
    required: true
  },
  createdOn: {
    type: Date,
    default: Date.now
  },
  updatedOn: {
    type: Date,
    default: Date.now
  }
});
```

```
const Note = mongoose.model('Note', noteSchema);

// Create a new note
const newNote = new Note({
  title: 'My first note',
  body: 'This is the body of my first note.'
});

newNote.save((err, note) => {
  if (err) {
    console.error(err);
  } else {
    console.log('Note created successfully:', note);
  }
});

// Read all notes
Note.find((err, notes) => {
  if (err) {
    console.error(err);
  } else {
    console.log('All notes:', notes);
  }
});

// Read a single note by ID
Note.findById('603dc404bf62f4a4d91a92af', (err, note) => {
  if (err) {
    console.error(err);
  } else {
    console.log('Note found by ID:', note);
  }
});

// Update a note by ID
Note.updateOne({ _id: '603dc404bf62f4a4d91a92af' }, { title: 'Updated title' }, (err, result) => {
  if (err) {
    console.error(err);
  } else {
    console.log('Note updated successfully:', result);
  }
});

// Delete a note by ID
Note.deleteOne({ _id: '603dc404bf62f4a4d91a92af' }, (err, result) => {
  if (err) {
```

```
    console.error(err);
  } else {
    console.log('Note deleted successfully:', result);
  }
});

// Start the Express server
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server listening on port ${PORT}.`);
});
```

OUTPUT:

This code connects to a MongoDB database using Mongoose, defines the Note schema, and performs various operations on the Note model using Mongoose methods such as .save(), .find(), .findById(), .updateOne(), and .deleteOne(). The output of the code will depend on the specific operations being performed and the state of the MongoDB database.

7.d)AIM:
Write a program to wrap the Schema into a Model object.

DESCRIPTION:
In the context of an Express application, models are typically used to represent data and define the interactions with a database. A model typically corresponds to a database table and is responsible for defining the schema and providing methods for interacting with the data.

PROGRAM:

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

// Define the schema
const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
  },
  age: {
    type: Number,
    default: 18,
  },
  email: {
    type: String,
    required: true,
    unique: true,
    match: /^\S+@\S+\.\S+$/,
  },
});
```

```
// Create the model object
const User = mongoose.model('User', userSchema);

// Use the model object to interact with the database
```

OUTPUT:

The output of the program will depend on the specific database operations being performed. Here's an example of the output you might see when running a program that retrieves all users from the database using the find() method of the User model object:

```
User.find({}, (err, users) => {
  if (err) {
    console.error(err);
  } else {
    console.log(users);
  }
});
```

Assuming there are three users in the database, the output might look something like this:

```
[
  { _id: '609ed60b70c0f9384409e8a6', name: 'John Doe', age: 30, email:
'john.doe@example.com' },
  { _id: '609ed60b70c0f9384409e8a7', name: 'Jane Smith', age: 25, email:
'jane.smith@example.com' },
  { _id: '609ed60b70c0f9384409e8a8', name: 'Bob Johnson', age: 40, email:
'bob.johnson@example.com' }
]
```

This output shows an array of three user objects, each with an ID, name, age, and email.

8.a)AIM:

Write a program to perform various CRUD (Create-Read-Update-Delete) operations using Mongoose library functions.

DESCRIPTION:

CRUD (Create, Read, Update, Delete) operations are common operations used in web development. In Express.js, we can perform CRUD operations using HTTP methods like GET, POST, PUT, and DELETE.

PROGRAM:

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

mongoose.connect('mongodb://localhost:27017/myapp', { useNewUrlParser: true,
useUnifiedTopology: true });
```

```
// Define schema
const userSchema = new mongoose.Schema({
  name: String,
  email: String,
});

// Define model
const User = mongoose.model('User', userSchema);

// Create a new user
const user = new User({ name: 'John', email: 'john@example.com' });
user.save((err) => {
  if (err) console.error(err);
  console.log('User created successfully');
});

// Read all users
User.find((err, users) => {
  if (err) console.error(err);
  console.log(users);
});

// Read user by ID
const id = '60519d4b4c4b5e5b5ca5e5d5';
User.findById(id, (err, user) => {
```

```
  if (err) console.error(err);
  console.log(user);
});

// Update user
User.findByIdAndUpdate(id, { name: 'Jane', email: 'jane@example.com' }, (err, user) => {
  if (err) console.error(err);
  console.log(user);
});

// Delete user
User.findByIdAndDelete(id, (err) => {
  if (err) console.error(err);
  console.log('User deleted successfully');
});
```

OUTPUT:

```
User created successfully
[
  {
    _id: 60519d3c5da7cb5b37c358a7,
    name: 'John',
    email: 'john@example.com',
    __v: 0
  }
]
```

{
_id: 60519d3c5da7cb5b37c358a7,
name: 'John',
email: 'john@example.com',
__v: 0
}
{
_id: 60519d3c5da7cb5b37c358a7,
name: 'Jane',
email: 'jane@example.com',
__v: 0
}

User deleted successfully

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

DESCRIPTION:

API development in Express.js involves creating API endpoints that can handle incoming HTTP requests, process them, and send back appropriate responses. The API endpoints can be used by clients to access and manipulate data or perform other operations.

PROGRAM:

```
import express from 'express';
import mongoose from 'mongoose';
import { Note } from './models/note';
```

```
const app = express();


// connect to MongoDB database
mongoose.connect('mongodb://localhost/myNotes');


// endpoint to fetch a note by its ID
app.get('/notes/:noteId', async (req, res) => {
  try {
    const noteId = req.params.noteId;
    const note = await Note.findById(noteId);
    if (!note) {
      return res.status(404).json({ message: 'Note not found' });
    }
    res.json(note);
  } catch (err) {
    console.error(err);
    res.status(500).json({ message: 'Internal server error' });
  }
});


// start the server
app.listen(3000, () => {
  console.log('Server started on port 3000');
});
```

OUTPUT:

As this code is meant to be run on a server, there won't be any visible output on the console. However, we can test the API using a tool like Postman or a web browser.

Assuming that the API is running on localhost:3000, we can send a GET request to the endpoint /notes/7555 by visiting the URL http://localhost:3000/notes/7555 in a web browser or using Postman.

If a note with ID 7555 exists in the database, its details will be returned as a JSON response. For example:

{

"_id": "6159078f7b9ca2d3fcf3b2c8",

"title": "Shopping list",

"content": "Milk, eggs, bread, cheese",

"createdAt": "2021-10-03T10:34:23.950Z",

"updatedAt": "2021-10-03T10:34:23.950Z",

"__v": 0

}If a note with ID 7555 doesn't exist in the database, a 404 status code will be returned along with a message. For example

{ "message": "Note not found" }

8.c)AIM:
Write a program to explain session management using cookies.

DESCRIPTION:

Session management and cookies are important concepts in web development for maintaining user state and improving user experience.

Sessions are used to maintain user state across multiple requests. When a user logs in to a website, a session is created on the server, and a unique session ID is generated and stored in a cookie on the user's browser. This session ID is used to identify the user's session on subsequent

requests, allowing the server to retrieve session data and maintain user state. Sessions can be used to store user-specific data such as user ID, username, user preferences, etc.

Cookies are small text files that are stored on the user's browser. They are used to store user-specific data such as login credentials, user preferences, and session IDs. Cookies can be set by the server or by client-side scripts, and they are sent to the server with each request. Cookies can also be used to store information about the user's activity on the website, such as the pages they have visited, the items they have added to their cart, etc.

Together, sessions and cookies enable web applications to maintain user state, personalize the user experience, and provide a more seamless user experience.

PROGRAM:

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

const session = require('express-session');

const cookieParser = require('cookie-parser');


const app = express();

const port = 3000;


// Set up middleware for session and cookie parsing

app.use(cookieParser());

app.use(session({

  secret: 'mysecretkey',

  resave: false,

  saveUninitialized: true,

  cookie: { secure: false }

}));


// Set up a simple login page
```

```
app.get('/', (req, res) => {
  res.send(`
    <h1>Login</h1>
    <form method="post" action="/login">
      <label for="username">Username:</label>
      <input type="text" id="username" name="username"><br>
      <label for="password">Password:</label>
      <input type="password" id="password" name="password"><br>
      <button type="submit">Submit</button>
    </form>
  `);
});

// Handle login form submission
app.post('/login', (req, res) => {
  const { username, password } = req.body;

  // Check if the username and password are valid
  if (username === 'admin' && password === 'password') {
    // Set a session cookie with the user's ID
    req.session.userId = 1;
    res.send('<p>Login successful!</p>');
  } else {
    res.send('<p>Invalid username or password.</p>');
  }
```

```
});

// Display a welcome message if the user is logged in
app.get('/welcome', (req, res) => {
  const userId = req.session.userId;

  if (userId) {
    res.send(`<p>Welcome, user ${userId}!</p>`);
  } else {
    res.send('<p>Please log in to view this page.</p>');
  }
});

// Log out the user by destroying the session
app.get('/logout', (req, res) => {
  req.session.destroy();
  res.send('<p>You have been logged out.</p>');
});

// Start the server
app.listen(port, () => {
  console.log(`Server running on port ${port}`);
});
```

OUTPUT:

Server running on port 3000

8.d)AIM:
Write a program to explain session management using sessions.

DESCRIPTION:

In web development, a session is a way to store and persist user data across multiple requests from the same user. Sessions allow a web application to keep track of user state and information, such as login credentials, shopping cart contents, and user preferences.

When a user logs in or interacts with a web application, the server creates a unique session for that user, which is associated with a session ID. The session ID is then stored as a cookie in the user's browser, and the server can use it to retrieve the corresponding session data for that user on subsequent requests.

PROGRAM:

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

const session = require('express-session');


const app = express();


app.use(session({

  secret: 'mysecretkey',

  resave: false,

  saveUninitialized: true

}));


app.get('/', (req, res) => {

  const sessionData = req.session;

  if (sessionData.views) {

    sessionData.views++;
```

```
    res.setHeader('Content-Type', 'text/html');
    res.write('<p>Number of views: ' + sessionData.views + '</p>');
    res.write('<p>Session ID: ' + sessionData.id + '</p>');
    res.write('<p><a href="/logout">Logout</a></p>');
    res.end();
  } else {
    sessionData.views = 1;
    res.send('Welcome to the session demo! Refresh the page to increase the view count.');
  }
});

app.get('/logout', (req, res) => {
  req.session.destroy((err) => {
    if (err) {
      console.log(err);
    } else {
      res.redirect('/');
    }
  });
});

app.listen(3000, () => {
  console.log('Server started on port 3000');
});
```

OUTPUT:

Server started on port 3000

Welcome to the session demo! Refresh the page to increase the view count.

When we refresh the page, we see the view count increase:

Number of views: 2

Session ID: aO9UhU6B-7-HprA0V7bTTfMz0aHBYsC8

Logout

8.e)AIM:
Implement security features in myNotes application

DESCRIPTION:

Security is a critical concern for any web application, and it is essential to ensure that the application is protected against common web vulnerabilities such as cross-site scripting (XSS), cross-site request forgery (CSRF), and other attacks. In addition, it is important to ensure that sensitive user data is encrypted in transit and at rest, and that the application is protected against attacks like SQL injection.

One of the ways to enhance the security of an Express.js application is by using middleware. Middleware functions are functions that have access to the request and response objects, and can modify them or perform other actions based on the needs of the application.

Helmet is a collection of middleware functions that help to improve the security of an Express.js application. Helmet includes middleware functions that set various HTTP headers that can help protect against attacks such as XSS and CSRF, as well as middleware that helps to secure cookies and enable HTTP Strict Transport Security (HSTS).

PROGRAM:

1. Authentication: Implement a user authentication system to ensure that only authorized users can access and modify their notes. This can be done using third-party authentication services like OAuth or by creating a custom authentication system.

Here's an example of a custom authentication system using Passport.js middleware:

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

const LocalStrategy = require('passport-local').Strategy;
```

```
const User = require('./models/user');



passport.use(new LocalStrategy(
  function(username, password, done) {
    User.findOne({ username: username }, function (err, user) {
      if (err) { return done(err); }
      if (!user) { return done(null, false); }
      if (!user.verifyPassword(password)) { return done(null, false); }
      return done(null, user);
    });
  }
));



app.post('/login',
  passport.authenticate('local', { failureRedirect: '/login' }),
  function(req, res) {
    res.redirect('/notes');
  });
```

2. Authorization: Once the user is authenticated, implement an authorization system to ensure that users can only access and modify their own notes. This can be done using role-based access control or by associating notes with specific users in the database.

Here's an example of associating notes with specific users in MongoDB:

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

const Schema = mongoose.Schema;
```

```
const noteSchema = new Schema({
  title: String,
  body: String,
  user: { type: Schema.Types.ObjectId, ref: 'User' }
});
```

```
const Note = mongoose.model('Note', noteSchema);

module.exports = Note;
```

3. Encryption: Ensure that user data is encrypted both in transit and at rest. This can be done using SSL/TLS encryption for in-transit data and data encryption at the database level.

Here's an example of enabling SSL/TLS encryption in an Express.js application:

```
const https = require('https');
const fs = require('fs');
const privateKey = fs.readFileSync('key.pem', 'utf8');
const certificate = fs.readFileSync('cert.pem', 'utf8');
const credentials = {key: privateKey, cert: certificate};

const app = express();

const httpsServer = https.createServer(credentials, app);
httpsServer.listen(443);
```

4. Helmet middleware: Finally, use the Helmet middleware to implement additional security measures like setting various HTTP headers to protect against attacks such as XSS and CSRF.

Here's an example of using the Helmet middleware in an Express.js application:

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

const helmet = require('helmet');


const app = express();


app.use(helmet());


app.listen(3000);
```

9.a)
AIM:
On the page, display the price of the mobile-based in three different colors. Instead ofusing the number in our code, represent them by string values like GoldPlatinum,PinkGold, SilverTitanium.

DESCRIPTION:

TypeScript is a statically-typed superset of JavaScript that is designed to help developers write more robust and maintainable code. It adds optional type annotations and other language features to JavaScript, making it easier to catch errors at compile-time rather than at runtime.

TypeScript is developed and maintained by Microsoft and is open-source. It can be used to build large-scale applications in both client-side and server-side environments. TypeScript code is transpiled into JavaScript code, which can then be run on any browser or server that supports JavaScript.

PROGRAM:

```
<!DOCTYPE html>

<html>

  <head>
```

```
  <title>Mobile Price in Different Colors</title>
 </head>
 <body>
  <div>
   <p style="color: gold;">GoldPlatinum: ${{mobilePrice}}</p>
   <p style="color: deeppink;">PinkGold: ${{mobilePrice}}</p>
   <p style="color: silver;">SilverTitanium: ${{mobilePrice}}</p>
  </div>
 </body>
</html>
```

OUTPUT:

GoldPlatinum: ${{mobilePrice}} (displayed in gold color)PinkGold: ${{mobilePrice}} (displayed in deep pink color)SilverTitanium: ${{mobilePrice}} (displayed in silver color)

9.b)

AIM:

Define an arrow function inside the event handler to filter the product array with theselected product object using the productId received by the function. Pass the selectedproduct object to the next screen.

DESCRIPTION:

In TypeScript, functions are first-class citizens, which means that they can be treated like any other value or variable. This allows for greater flexibility and expressiveness when defining functions in TypeScript.

To define a function in TypeScript, you use the **function** keyword, followed by the function name and a set of parentheses that contain any parameters that the function takes. The function body is then enclosed in curly braces, and the return type is specified after the closing parenthesis of the parameter list, using a colon followed by the return type

PROGRAM:

```
import { useState } from "react";

interface Product {
  productId: number;
  name: string;
  price: number;
}

const products: Product[] = [
  { productId: 1, name: "Product 1", price: 10 },
  { productId: 2, name: "Product 2", price: 20 },
  { productId: 3, name: "Product 3", price: 30 },
];

function ProductList(): JSX.Element {
  const [selectedProduct, setSelectedProduct] = useState<Product | null>(null);

  function handleProductSelect(productId: number): void {
    // Find the selected product object in the product array
    const product = products.find((product) => product.productId === productId);

    // Set the selected product state
    setSelectedProduct(product ?? null);
  }
```

```
function navigateToProductDetails(product: Product): void {
  // Navigate to the product details screen with the selected product object
  console.log("Navigating to product details screen for product:", product);
}

return (
  <div>
    <h1>Product List</h1>
    <ul>
      {products.map((product) => (
        <li key={product.productId}>
          <button onClick={() => handleProductSelect(product.productId)}>
            {product.name} (${product.price})
          </button>
        </li>
      ))}
    </ul>
    {selectedProduct && (
      <div>
        <h2>Selected Product</h2>
        <p>{selectedProduct.name} (${selectedProduct.price})</p>
        <button onClick={() => navigateToProductDetails(selectedProduct)}>
          View Details
        </button>
      </div>
```

```
    )}
  </div>
 );
}
```

export default ProductList;

OUTPUT:

Product List

- Product 1 ($10)

- Product 2 ($20)

- Product 3 ($30)

Selected Product

Product 2 ($20)

[View Details]

9.c)
AIM:
Consider that developer needs to declare a function - getMobileByVendor whichaccepts string as input parameter and returns the list of mobiles.

DESCRIPTION: In TypeScript, parameter types and return types are used to specify the data types of the input and output of a function. They can be explicitly defined using type annotations.Parameter types specify the data type of the function's input parameters. They are placed after the parameter name, separated by a colon

PROGRAM:

```
interface Mobile {
```

```
  brand: string;
  model: string;
  price: number;
}

const mobiles: Mobile[] = [
  { brand: 'Apple', model: 'iPhone 13', price: 999 },
  { brand: 'Samsung', model: 'Galaxy S21', price: 799 },
  { brand: 'OnePlus', model: '9 Pro', price: 899 },
  { brand: 'Xiaomi', model: 'Mi 11', price: 799 },
];

function getMobileByVendor(vendor: string): Mobile[] {
  return mobiles.filter((mobile: Mobile) => mobile.brand.toLowerCase() ===
vendor.toLowerCase());
}

const appleMobiles: Mobile[] = getMobileByVendor("Apple");
console.log(appleMobiles);
```

Output: [{ brand: 'Apple', model: 'iPhone 13', price: 999 }]

9.d)AIM:

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

DESCRIPTION:

Arrow functions are a shorthand syntax for writing functions in TypeScript (and JavaScript). Arrow functions provide a concise way to define functions using the => syntax. Arrow functions have a number of advantages over traditional function syntax. They are more concise, making code easier to read and write. They also have a lexical **this** binding, which means that the **this** value inside the function is the same as the **this** value outside the function.

PROGRAM:

```
interface Manufacturer {
  id: number;
  price: number;
}

const manufacturers: Manufacturer[] = [
  { id: 1, price: 10 },
  { id: 2, price: 15 },
  { id: 3, price: 20 },
  { id: 4, price: 25 }
];

const myfunction = (priceThreshold: number) => {
  return manufacturers.filter(manufacturer => manufacturer.price > priceThreshold)
    .map(manufacturer => manufacturer.id);
}

const filteredIds = myfunction(15);
```

```
console.log(filteredIds);
```

Output: [3, 4]

9.e)
AIM:
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

DESCRIPTION:

Optional and default parameters are features in TypeScript (and JavaScript) that allow developers to define function parameters with optional or default values.Optional parameters are indicated by adding a question mark (**?**) after the parameter name in the function declaration. Optional parameters can be omitted when calling the function, and will be assigned the value **undefined** if not provide Default.Parameters are indicated by assigning a default value to the parameter in the function declaration. Default parameters will be assigned the default value if no value is provided when calling the function

PROGRAM:

```
interface Mobile {
  id: number;
  manufacturer: string;
  model: string;
}

const mobiles: Mobile[] = [
  { id: 101, manufacturer: 'Samsung', model: 'Galaxy S21' },
  { id: 102, manufacturer: 'Apple', model: 'iPhone 12' },
  { id: 103, manufacturer: 'OnePlus', model: '9 Pro' }
];
```

```
function getMobileByManufacturer(manufacturer: string, id?: number) {
  const filteredMobiles = mobiles.filter(mobile => mobile.manufacturer === manufacturer);
  if (id) {
    return filteredMobiles.find(mobile => mobile.id === id);
  } else {
    return filteredMobiles;
  }
}

console.log(getMobileByManufacturer('Samsung')); // Output: [{ id: 101, manufacturer:
'Samsung', model: 'Galaxy S21' }]

console.log(getMobileByManufacturer('Samsung', 101)); // Output: { id: 101,
manufacturer: 'Samsung', model: 'Galaxy S21' }
```

10.a)
AIM:
Implement business logic for adding multiple Product values into a cart variable which is type of string array.

DESCRIPTION:

A rest parameter is a feature in TypeScript (and JavaScript) that allows developers to define a function parameter that captures all remaining arguments passed to the function as an array. The rest parameter is indicated by prefixing the last function parameter with three dots (**...**)

PROGRAM:

```
let cart: string[] = [];

function addToCart(...products: string[]) {
  cart = [...cart, ...products];
}

addToCart('Apple', 'Banana', 'Cherry');
console.log(cart); // Output: [ 'Apple', 'Banana', 'Cherry' ]
```

```
addToCart('Donut', 'Egg', 'Fish');
console.log(cart);
```

Output:

```
[ 'Apple', 'Banana', 'Cherry', 'Donut', 'Egg', 'Fish' ]
```

10.b)

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:

In TypeScript, an interface is a way to define a contract for an object, specifying the properties and methods that the object should have. An interface defines the shape of an object, without providing any implementation details.

To create an interface in TypeScript, you can use the **interface** keyword, followed by the name of the interface and its properties and methods.

PROGRAM:

```
interface Product {
  productId: number;
  productName: string;
}

function addProduct(product: Product) {
  console.log(`Added product ${product.productName} with ID ${product.productId}`);
}

const newProduct: Product = { productId: 123, productName: 'Laptop' };
addProduct(newProduct);
```

Output: Added product Laptop with ID 123

10.c)

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 is a concept in TypeScript (and other programming languages) that refers to checking for the presence of certain properties or methods in an object, rather than its actual type or class. The term "duck typing" comes from the saying, "If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck."

In TypeScript, duck typing is often used in cases where multiple classes or interfaces share a similar set of properties or methods. Instead of explicitly checking for the type or class of an object, you can check if it has the necessary properties or methods to fulfill the requirements of a particular piece of code.

PROGRAM:

```
interface Product {
  productId: number;
  productName: string;
}

function addProduct(product: { productId: number, productName: string }) {
  console.log(`Added product ${product.productName} with ID ${product.productId}`);
}

const newProduct = { productId: 123, productName: 'Laptop' };
addProduct(newProduct);
```

Output:

Added product Laptop with ID 123

10.d)

AIM: Declare an interface with function type and access its value.

DESCRIPTION:

In TypeScript, function types refer to the types of functions, including their input parameters and output values. Function types can be used to define the type signature of a function, allowing for better type checking and error handling.

To define a function type, you can use the syntax **(param1: type1, param2: type2, ...) => returnType**, where **param1**, **param2**, etc. are the parameter names, **type1**, **type2**, etc. are the parameter types, and **returnType** is the return type of the function.

PROGRAM:

```
// Define an interface with a function type
interface MyFunction {
  (a: number, b: number): number;
}

// Define a function that implements the interface
const add: MyFunction = (a, b) => {
  return a + b;
};

// Call the function and output the result
const result = add(2, 3);
console.log(result);
```

Output: 5

11.a)
AIM:
Declare a productList interface which extends properties from two other declared interfaces like Category,Product as well as implementation to create a variable of this interface type.

DESCRIPTION:

In TypeScript, interfaces can be extended by other interfaces using the **extends** keyword. This allows you to create new interfaces that inherit the properties and methods of an existing interface, while also adding new properties and methods of their own.

To extend an interface, simply define a new interface and use the **extends** keyword to specify the parent interface that it inherits from

PROGRAM:

```
interface Category {
```

```
  id: number;
  name: string;
}

interface Product {
  id: number;
  name: string;
  price: number;
}

interface ProductList extends Category, Product {
  quantity: number;
}

const myProductList: ProductList = {
  id: 1,
  name: "T-Shirt",
  price: 19.99,
  quantity: 10
};

console.log(myProductList);
 Output:
{ id: 1, name: 'T-Shirt', price: 19.99, quantity: 10 }
```

11.b)
AIM:
Consider the Mobile Cart application, Create objects of the Product class and place them into the productlist array.

DESCRIPTION:

In TypeScript, classes are used to define object blueprints with properties and methods. They provide a way to create objects that have the same structure and behavior, making it easier to manage and manipulate complex data.

To define a class in TypeScript, you use the **class** keyword followed by the name of the class.

PROGRAM:

```
class Product {
  name: string;
  price: number;
```

```
  constructor(name: string, price: number) {
    this.name = name;
    this.price = price;
  }
}

const productList: Product[] = [
  new Product("iPhone X", 999),
  new Product("Samsung Galaxy S21", 799),
  new Product("Google Pixel 5", 699),
];

console.log(productList);
```

Output:

[Product { name: 'iPhone X', price: 999 }, Product { name: 'Samsung Galaxy S21', price: 799 }, Product { name: 'Google Pixel 5', price: 699 }]

11.c)

AIM:

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>>".

DESCRIPTION:

In TypeScript, the **constructor** method is a special method that is used to create and initialize objects created from a class. It is called automatically when a new object is created using the **new** keyword and can be used to set initial values for object properties.

PROGRAM:

```
class Product {
  productId: number;

  constructor(productId: number) {
    this.productId = productId;
  }

  getProductId() {
    return `Product id is ${this.productId}`;
```

```
  }
}

const product1 = new Product(123);
console.log(product1.getProductId());
```

Output:

Product id is 123

11.d)AIM:

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.

DESCRIPTION:

In TypeScript, access modifiers are used to control the accessibility of class members (properties and methods). There are three access modifiers available: **public**, **private**, and **protected**.

public members are accessible from anywhere, both within and outside of the class.**private** members are only accessible within the class that defines them.**protected** members are accessible within the class that defines them and any subclasses.

PROGRAM:

```
class Product {
  private productId: number;
  public productName: string;
  protected productPrice: number;
  static productCategory: string = "Electronics";

  constructor(productId: number, productName: string, productPrice: number) {
    this.productId = productId;
    this.productName = productName;
    this.productPrice = productPrice;
  }

  getProductDetails() {
    return `Product: ${this.productName} (id: ${this.productId}, price:
${this.productPrice})`;
  }
}

class Gadget {
  constructor(private product: Product) {}
```

```
  getProductCategory() {
    return Product.productCategory;
  }

  getProductDetails() {
    return this.product.getProductDetails();
  }
}

const product1 = new Product(123, "iPhone X", 999);
const gadget1 = new Gadget(product1);

console.log(gadget1.getProductCategory()); // Output: Electronics
console.log(gadget1.getProductDetails()); // Output: Product: iPhone X (id: 123, price: 999)
```

12.a)AIM:

Create a Product class with 4 properties namely productId and methods to setProductId() and getProductId().

DESCRIPTION:

In TypeScript, properties and methods are used to define the state and behavior of a class.

Properties represent the state of an object and are used to store data. They can be declared with different access modifiers (**public**, **private**, **protected**) to control their visibility and accessibility.

Methods represent the behavior of an object and are used to perform operations on the object's data. They can be declared with different access modifiers to control their visibility and accessibility, and can also accept parameters and return values.

Properties and methods are accessed using the dot notation, where the property or method name is followed by a dot (**.**) and the name of the object instance.

PROGRAM:

```
class Product {
  private productId: number;

  constructor(productId: number) {
    this.setProductId(productId);
  }

  setProductId(productId: number) {
    this.productId = productId;
  }
```

```
  getProductId() {
    return this.productId;
  }
}

const product1 = new Product(123);
console.log(product1.getProductId()); // Output: 123

product1.setProductId(456);
console.log(product1.getProductId()); // Output: 456
```

12.b)AIM:

Create a namespace called ProductUtility and place the Product class definition in it. Import the Product class inside productlist file and use it.

DESCRIPTION:

In TypeScript, namespaces are used to group related code into a single container to avoid naming collisions with code in other namespaces or in the global scope.

To create a namespace, we use the **namespace** keyword followed by the name of the namespace and a set of curly braces containing the code that belongs to the namespace.

For example, suppose we have a set of utility functions for working with arrays. We can group these functions into a namespace called **ArrayUtils**

PROGRAM:

```
namespace ProductUtility {
  export class Product {
    private productId: number;

    constructor(productId: number) {
      this.setProductId(productId);
    }

    setProductId(productId: number) {
      this.productId = productId;
    }

    getProductId() {
      return this.productId;
    }
  }
```

```
}

const product1 = new ProductUtility.Product(123);
console.log(product1.getProductId());  Output: 123

product1.setProductId(456);
console.log(product1.getProductId()); Output: 456
```

12.c)AIM:

Consider the Mobile Cart application which is designed as part of the functions in a module to calculate the total price of the product using the quantity and price values and assign it to a totalPrice variable.

DESCRIPTION:

In TypeScript, modules are used to organize code into reusable and maintainable units. A module can contain variables, functions, classes, interfaces, and other declarations, and can be used in other modules or applications.

To create a module in TypeScript, you can use the **export** keyword to export variables, functions, or classes from a file. For example, to export a function **add** from a file **math.ts**

PROGRAM:

12.d)AIM:

Create a generic array and function to sort numbers as well as string values.

DESCRIPTION:

Generics in TypeScript is a powerful feature that allows you to create reusable components that can work with a variety of data types. It allows you to write more flexible and type-safe code, while still providing the flexibility to work with different data types.

Type Parameters: Type parameters are used to define a generic type that can work with any data type. A type parameter is a placeholder for a type that will be provided later when the function, class or interface is used. Type parameters are enclosed in angle brackets (<>) and are placed immediately after the name of the function, class or interface

Generic Functions: Generic functions are functions that are defined with one or more type parameters. These type parameters can be used in the function's signature, allowing the function to work with a wide variety of data types.

Generic Constraints: Generic constraints are used to restrict the types that a type parameter can be replaced with. This is useful when you want to enforce certain properties on the data type being used

PROGRAM:

productUtils.ts

```
export function calculateTotalPrice(quantity: number, price: number): number {
  return quantity * price;
}
```

mobileCart.ts

```
import { calculateTotalPrice } from './productUtils';

class Product {
  constructor(private productId: number, private productName: string, private
productPrice: number) {}

  getProductDetails() {
    return `Product ID: ${this.productId}\nProduct Name: ${this.productName}\nProduct
Price: $${this.productPrice}\n`;
  }

  getTotalPrice(quantity: number) {
    return calculateTotalPrice(quantity, this.productPrice);
  }
}

const product1 = new Product(1, 'iPhone', 999);
const product2 = new Product(2, 'Samsung Galaxy', 799);

const quantity1 = 2;
const quantity2 = 3;

const totalPrice1 = product1.getTotalPrice(quantity1);
const totalPrice2 = product2.getTotalPrice(quantity2);

console.log(product1.getProductDetails() + `Quantity: ${quantity1}\nTotal Price:
$${totalPrice1}\n`);
console.log(product2.getProductDetails() + `Quantity: ${quantity2}\nTotal Price:
$${totalPrice2}\n`);
```

OUTPUT:

Product ID: 1
Product Name: iPhone
Product Price: $999
Quantity: 2
Total Price: $1998

Product ID: 2
Product Name: Samsung Galaxy
Product Price: $799
Quantity: 3
Total Price: $2397