| **Experiment No. – 3** | | | | |
| --- | --- | --- | --- | --- |
| **Date of Performance:** | **22/1/25** | | | |
| **Date of Submission:** | **29/1/25** | | | |
| Program Execution/  formation/  correction/  ethical practices  (06) | Timely  Submission  (01) | Viva  (03) | Experiment  Total (10) | Sign with Date |
|  |  |  |  |  |

**Experiment No. 3**

**3.1 Aim:** Implement Inheritance using TypeScript and HTML

**3.2 Course Outcome:** CO2: Understand how TypeScript and AngularJS framework can build dynamic, responsive single-page web applications

**3.3 Learning Objectives:** To understand how inheritance works in TypeScript by creating base and derived classes, compiling and running the code in the browser, and displaying the output using HTML.

**3.4 Requirement:**

* Visual Studio Code / Any IDE
* TypeScript Compiler (tsc)
* Node.js (for compilation)
* Browser (Chrome/Edge/Firefox)

3.5 Related Theory:

Inheritance is a fundamental concept in Object-Oriented Programming (OOP) that allows one class (called the *child* or *derived* class) to inherit the properties and behaviors (methods) of another class (called the *parent* or *base* class). This helps in code reusability, readability, and maintaining a cleaner and modular codebase.

Key Concepts in TypeScript Inheritance:

1. Class Declaration: A class in TypeScript is declared using the class keyword. It can contain properties (variables) and methods (functions).
2. Base (Parent) Class: The base class defines common features that can be inherited by derived classes.
3. Derived (Child) Class: The derived class uses the extends keyword to inherit the parent class.
4. Constructor and super(): If a derived class has a constructor, it must call the parent class constructor using the super() method before using this.
5. Method Overriding: A derived class can override methods of the base class to provide its own implementation.
6. Access Modifiers: TypeScript uses access modifiers like:
   * public: Accessible anywhere.
   * private: Accessible only within the class.
   * protected: Accessible within the class and its subclasses.

3.6 Procedure:

Step 1: Create a TypeScript file (inheritance.ts) containing base and derived classes.

Step 2: Use the tsc inheritance.ts command to compile it into JavaScript.

Step 3: Link the compiled JS file in an HTML file.

Step 4: Use innerHTML to show the output.

Step 5: Run the HTML file in the browser.

**3.7 Program and Output:**

index.html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>Library Management System</title>

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

</head>

<body>

<h1>Library Management System</h1>

<div class="container">

<h2>Add a Book</h2>

<input type="text" id="bookTitle" placeholder="Enter Book Title">

<input type="text" id="authorName" placeholder="Enter Author Name">

<button id="addBookBtn">Add Book</button>

</div>

<div class="container">

<h2>Library Books</h2>

<ul id="bookList"></ul>

</div>

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

</body>

</html>

style.css

body {

font-family: Arial, sans-serif;

text-align: center;

background-color: #f4f4f4;

}

.container {

background: white;

padding: 20px;

width: 50%;

margin: 20px auto;

border-radius: 8px;

box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);

}

input {

width: 80%;

padding: 8px;

margin: 10px 0;

border: 1px solid #ccc;

border-radius: 5px;

}

button {

padding: 10px 15px;

background-color: #28a745;

color: white;

border: none;

border-radius: 5px;

cursor: pointer;

}

button:hover {

background-color: #218838;

}

ul {

list-style-type: none;

padding: 0;

}

li {

background: #fff;

padding: 10px;

margin: 5px;

border-radius: 5px;

box-shadow: 0 0 5px rgba(0, 0, 0, 0.2);

}

app.ts

// Base Class: Person

class Person {

name: string;

constructor(name: string) {

this.name = name;

}

getName(): string {

return this.name;

}

}

// Intermediate Class: User (inherits from Person)

class User extends Person {

userId: number;

constructor(name: string, userId: number) {

super(name); // Call the constructor of Person

this.userId = userId;

}

getUserId(): number {

return this.userId;

}

}

// Final Class: Librarian (inherits from User)

class Librarian extends User {

books: string[] = [];

constructor(name: string, userId: number) {

super(name, userId); // Call the constructor of User

}

addBook(bookTitle: string, author: string) {

this.books.push(`${bookTitle} by ${author}`);

console.log(`Book added: ${bookTitle} by ${author}`);

}

getBooks(): string[] {

return this.books;

}

}

// UI Interaction

const librarian = new Librarian("Alice", 101);

document.getElementById("addBookBtn")?.addEventListener("click", () => {

const bookTitle = (document.getElementById("bookTitle") as HTMLInputElement).value;

const authorName = (document.getElementById("authorName") as HTMLInputElement).value;

if (bookTitle && authorName) {

librarian.addBook(bookTitle, authorName);

displayBooks();

(document.getElementById("bookTitle") as HTMLInputElement).value = "";

(document.getElementById("authorName") as HTMLInputElement).value = "";

}

});

// Function to display books

function displayBooks() {

const bookList = document.getElementById("bookList") as HTMLUListElement;

bookList.innerHTML = "";

librarian.getBooks().forEach(book => {

const li = document.createElement("li");

li.textContent = book;

bookList.appendChild(li);

});

}

app.js

var \_\_extends = (this && this.\_\_extends) || (function () {

var extendStatics = function (d, b) {

extendStatics = Object.setPrototypeOf ||

({ \_\_proto\_\_: [] } instanceof Array && function (d, b) { d.\_\_proto\_\_ = b; }) ||

function (d, b) { for (var p in b) if (Object.prototype.hasOwnProperty.call(b, p)) d[p] = b[p]; };

return extendStatics(d, b);

};

return function (d, b) {

if (typeof b !== "function" && b !== null)

throw new TypeError("Class extends value " + String(b) + " is not a constructor or null");

extendStatics(d, b);

function \_\_() { this.constructor = d; }

d.prototype = b === null ? Object.create(b) : (\_\_.prototype = b.prototype, new \_\_());

};

})();

var \_a;

// Base Class: Person

var Person = /\*\* @class \*/ (function () {

function Person(name) {

this.name = name;

}

Person.prototype.getName = function () {

return this.name;

};

return Person;

}());

// Intermediate Class: User (inherits from Person)

var User = /\*\* @class \*/ (function (\_super) {

\_\_extends(User, \_super);

function User(name, userId) {

var \_this = \_super.call(this, name) || this; // Call the constructor of Person

\_this.userId = userId;

return \_this;

}

User.prototype.getUserId = function () {

return this.userId;

};

return User;

}(Person));

// Final Class: Librarian (inherits from User)

var Librarian = /\*\* @class \*/ (function (\_super) {

\_\_extends(Librarian, \_super);

function Librarian(name, userId) {

var \_this = \_super.call(this, name, userId) || this; // Call the constructor of User

\_this.books = [];

return \_this;

}

Librarian.prototype.addBook = function (bookTitle, author) {

this.books.push("".concat(bookTitle, " by ").concat(author));

console.log("Book added: ".concat(bookTitle, " by ").concat(author));

};

Librarian.prototype.getBooks = function () {

return this.books;

};

return Librarian;

}(User));

// UI Interaction

var librarian = new Librarian("Alice", 101);

(\_a = document.getElementById("addBookBtn")) === null || \_a === void 0 ? void 0 : \_a.addEventListener("click", function () {

var bookTitle = document.getElementById("bookTitle").value;

var authorName = document.getElementById("authorName").value;

if (bookTitle && authorName) {

librarian.addBook(bookTitle, authorName);

displayBooks();

document.getElementById("bookTitle").value = "";

document.getElementById("authorName").value = "";

}

});

// Function to display books

function displayBooks() {

var bookList = document.getElementById("bookList");

bookList.innerHTML = "";

librarian.getBooks().forEach(function (book) {

var li = document.createElement("li");

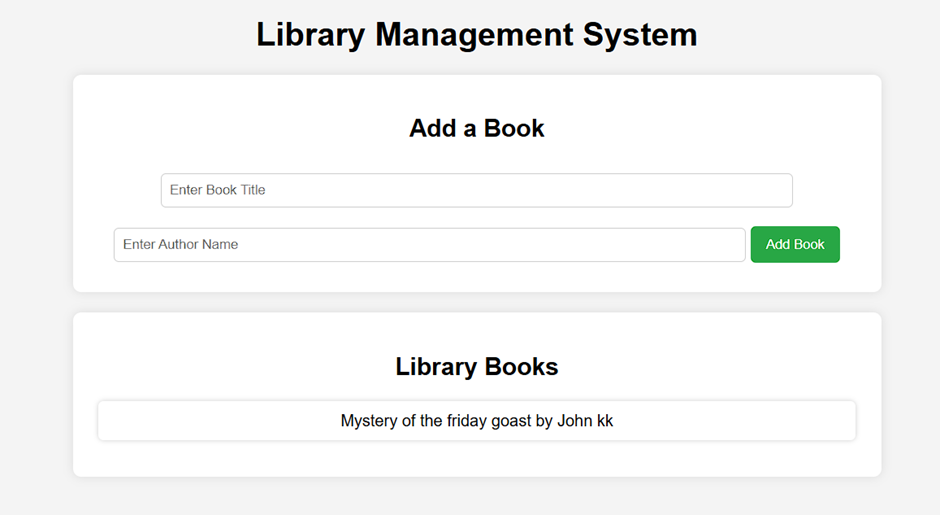
li.textContent = book;

bookList.appendChild(li);

});

}

Output:



**3.8 Conclusion:** The experiment successfully demonstrated how inheritance works in TypeScript using classes and how the output can be shown using an HTML page. It reinforced key OOP concepts in a web development context.

**3.9 Questions:**

1. What is the role of the super() keyword in TypeScript?

The super() keyword in TypeScript is used within a derived class to call the constructor of its parent class. It ensures that the base class is properly initialized before the derived class starts using the inherited properties or methods. Additionally, super can also be used to call parent class methods from the child class when method overriding occurs. This helps maintain and extend the behavior of the base class in a clean and structured way.

1. How is inheritance different in Typescript compared to JavaScript?

The super() keyword in TypeScript is used within a derived class to call the constructor of its parent class. It ensures that the base class is properly initialized before the derived class starts using the inherited properties or methods. Additionally, super can also be used to call parent class methods from the child class when method overriding occurs. This helps maintain and extend the behavior of the base class in a clean and structured way.

1. Can TypeScript support multiple inheritance? If not, how is it achieved?

TypeScript does not support multiple inheritance directly, but it can be achieved using interfaces and mixins.