EXPERIMENT 3

AIM: Demonstrate basic calculator & function overload-ing using TypeScript.

THEORY:

- What is TypeScript?

TypeScript is a subset of JavaScript that adds static typing, interfaces & Modern ECMAScript features to improve code maintainability & scalability. It compiles down to JavaScript, making it usable in web development, Node.js applications & other environments where JavaScript runs.

Key features:

i) Static Typing:
- It helps catch errors at compile time.

ii) Interfaces & Generics:
- Enables better object structure definitions & reusable func-tions

iii) Object-Oriented Programming:
- Supports classes, inheritances & access modifiers like public, private & protected.

- What is Function Overloading?

Function Overloading in TypeScript allows multiple func-tion signatures for a single function, enabling different input parameters & return types. The actual function implementation determines which signature to execute based on the passed arguements.

Syntax :

```
function add(a: number, b: number): number;  ... (1)
function add(a: string, b: string): string;  ... (2)
function add(a: any, b: any): any {          ... (3)
    return a+b;
}

console.log(add(5,10));
console.log(add("Hello ", "World!"));
```

- First Two lines [1,2] : Define two first / function signatures.
- Third Line [3] : Uses any type to handle different data types.
- Execution : TypeScript selects the appropriate overload at compile time.

CONCLUSION : Hence, we have successfully implemented basic calculator & function overloading using TypeScript.

EXPERIMENT 4

AIM: Write a program to demonstrate Single Inheritance & Multiple Inheritance using TypeScript.

THEORY:

Inheritance in typescript:

Inheritance is a key feature of Object-Oriented Programming (OOP) that allows a class (child) to inherit properties & methods from another class (parent). This promote code reusability, modularity & maintainability.

In TypeScript, inheritance is achieved using the 'extends' keyword for single inheritance & the 'implements' keyword for multiple inheritance. TypeScript doesn't support direct multiple class inheritance but can be achieved through multiple interface implementation.

① Single Inheritance in TypeScript

- Single inheritance occurs when a child class (sub-class) derives from a single parent class (superclass).

Syntax:

```
class Parent {
    parentProperty: string = "I am the parent";
    showParentProperty(): void {
        console.log(this.parentProperty);
    }
}
class Child extends Parent {
    childProperty: string = "I am the child";
    showChildProperty(): void {
        console.log(this.childProperty);
```

```
    }
}
let obj = new Child();
obj.showParentProperty();
obj.showChildProperty();
```

② Multiple Inheritances using Interfaces in TypeScript

- TypeScript does not allow/support multiple class inheritance, but it allows a class to implement multiple interfaces using 'implement' keyword.

Syntax:

```
interface A {
    methodA(): void;
}
interface B {
    methodB(): void;
}
class C implements A, B {
    methodA(): void {
        console.log("A from interface A");
    }
    methodB(): void {
        console.log("B from interface B");
    }
}
let obj = new C();
obj.methodA();
obj.methodB();
```

CONCLUSION: Hence, we have successfully demonstrated Single Inheritance & Multiple Inheritance in TypeScript.

EXPERIMENT 5

AIM: Demonstrate Access Modifiers example in TypeScrip

THEORY:

- Access Modifiers

Access Modifiers in TypeScript define the visibility & accessibility of class members (properties & methods). They control how the data is accessed within & outside the class, ensuring data encapsulation & security.

- Types of Access Modifiers:

① public:
- Accessible from anywhere.
- It can be accessed anywhere inside or outside the class.

② private:
- Accessible only within the same class.
- It cannot be accessed in derived classes or outside a class.

③ protected:
- Accessible within the same class & derived (child) classes.
- It cannot be accessed outside the class.

Syntax:

```
class Example {
    public publicProperty : string = "I am public";
    private privateProperty : string = "I am private";
    protected protectedProperty : string = "I am protected";
```

```
    public showProperties(): void {
        console.log(this.publicProperty);
        console.log(this.privateProperty);
        console.log(this.protectedProperty);
    }
}
class Derived extends Example {
    showProtected(): void {
        console.log(this.publicProperty); ✓
        console.log(this.privateProperty); ✗
        console.log(this.protectedProperty); ✓
    }
}

let obj = new Example();
console.log(obj.publicProperty); ✓
console.log(obj.privateProperty); ✗
console.log(obj.protectedProperty); ✗
```

CONCLUSION: Hence, we have successfully demonstrated s Access modifiers example in TypeScript.

EXPERIMENT 6

AIM: Write a JavaScript program for AJAX & to use AJAX for user validation & to show the result on the same page below the submit button.

THEORY:

- AJAX

→ - AJAX (Asynchronous JavaScript & XML) is a technique that allows web pages to send & recieve data from a server without reloading the page. It enhances user experience by enabling real-time updates, such as form validation, chat applications & live search.

- Role of AJAX

- Asynchronous Communication:
AJAX allows JavaScript to fetch data from a server in the background, without interrupting the user's experience.

→ - Faster Page Loads:
Instead of reloading an entire page / web page, only parts of the page update dynamically.

- Improved user experience:
Forms, search results, & notifications appear instantly without reloading the page.

- Server Interaction:
It allows sending requests to the server via 'GET' or 'POST' & processing responses dynamically.

- Syntax

AJAX is implemented using the 'XMLHttpRequest' object or 'Fetch API'.

```
let xhr = new XMLHttpRequest();
xhr.open("GET", "server-url", true);
xhr.onreadystatechange = function () {
    if (xhr.readyState === 4 && xhr.status === 200) {
        console.log(xhr.responseText);
    }
};
xhr.send();
```

- open (method, url, async): Defines the request type 'GET', 'POST' & whether its asynchronous.
- onreadystatechange: Triggers every time the request state changes.
- xhr.readyState: Holds request states.
- xhr.status: Checks if the request was successful.
- send(): Sends request to the server.

CONCLUSION: Hence, we have successfully implemented AJAX for user validation & to display result on the same page below submit button.

EXPERIMENT 7

AIM: Create a simple HTML "Hello World" project using AngularJS Framework & apply ng-controller, ng-model & expressions.

THEORY:

- **AngularJS Framework**
- AngularJS is a JavaScript-based front-end framework developed by Google for building dynamic web applications. It extends HTML with additional directives & makes it easy to bind data to the UI using two-way data binding.
- **MVC (Model - View - Controller):**
- It consists of MVC architecture for structured development.
- **Two way data binding:**
- It keeps the UI and model data in sync.
- **Directives:**
- Directives like ng-model, ng-controller, ng-bind, etc extend HTML functionality.
- **Dependency:**
- Dependency injection makes the code modular & reusable.
- **Single Page Application (SPA):**
- SPA (Single Page Application) support for building faster web applications.

- **Ng-Controller:**
It is an angularjs directive used to define a controller for a specific section of an HTML Page. It manages app data & logic inside JS function. This controller connects View (HTML) with Model (Data).

Syntax:
'HTML'
```
<div ng-controller = "MyController">
    <p> {{ message }} </p>
</div>
```
'JavaScript'
```
app.controller ("MyController", function ($scope) {
    $scope.message = "Hello from Controller";
});
```

- **Ng-Model:**
It is an angularJS directive that binds input fields input fields to the model data. It enables two-way binding, meaning that when the user updates the input field, the model is updated automatically.
Syntax:
'HTML'
```
<input type = "text" ng-model = "name">
<p> Hello, {{name}} </p>
```

- **Expressions:**
They are used to evaluate & display values inside HTML. They work like JS expressions but are written inside '{{..}}'.
SYNTAX:
'HTML'
```
<p>{{ name }} </p>
```

CONCLUSION: Hence, we have successfully implemented "Hello World" program using AngularJS & applied ng-controller, model & expressions.

EXPERIMENT 8

AIM: Demonstrate Events & Validations in AngularJS (Create functions & add events, adding HTML validators using $valid property of Angular, etc).

THEORY:

- Events in AngularJS

Events in AngularJS are interactions (like clicks, keypresses, mouse movements) that trigger functions in the controller. AngularJS provides directives to handle these events efficiently. Common AngularJS Event directives are 'ng-click', 'ng-change', 'ng-keyup', 'ng-mouseover'.

Syntax: 'HTML':

```
<button ng-click="sayHello()"> Click Me </button>
<p> {{message}} </p>
```

'Javascript':

```
$scope.sayHello = function() {
    $scope.message = "Hello, AngularJS!";
};
```

- Validations in AngularJS

AngularJS provides built-in form validations using 'ng-model' 'ng-form', & built-in validators. It ensures user enter valid data before submitting a form.

- Client side validation: It is done before sending data to the server.
- Error handling: It can display validation messages when the input is correct.

- Pre-defined validators : AngularJS provides built-in validation for common scenarios like required fields, emails, numbers, & minimum/maximum lengths.

- HTML Validators in AngularJS
AngularJS extends standard HTML5 validation attributes. Some are:
- required = Ensures the input field is not empty.
- ng-minlength = "n" = Sets a minimum number of characters.
- ng-maxlength = "n" = Set a maximum number of characters.
- ng-pattern = "/regex/" = Ensures the input matches a pattern.
- type = "email" = Validates email format.
- type = "number" = Ensures numeric input

Syntax:

```
<form name = "myform">
   <input type = "text" name = user ng-model = "user.name"
                                                required>
   <span ng-show = "myform.user.$error.required"> Name is
                                                required.</span>
</form>
```

- $valid property in AngularJS
- The '$valid' property is a part of AngularJS form validation. It checks if all form fields meet their validation requirements. If the form is fully valid, '$valid' returns true, otherwise it returns false.

Syntax:

```
<form name = "myform">
  <input type = "text" name = "email" ng-model = "user.email"
                                type = "email" required >
  <p ng-show = "myform.email.$valid"> Valid Email ! </p>
</form>
```

CONCLUSION: Hence, we have successfully demonstrated Events & Validations in AngularJS.

EXPERIMENT 9

AIM: Write a program to create a database & collection in MongoDB & write a program to demonstrate find, insert, update & delete ops in MongoDB using NodeJS.

THEORY:

- MongoDB

MongoDB is a NoSQL database that stores data in flexible, JSON-like format called BSON. Unlike relational databases (SQL), it does not use tables & rows. Insted, it uses collections & documents, making it ideal for handling large-scale, dynamic data.

- Database & Collection in MongoDB

- Database: A container for collections (like a database in SQL).
- Collection: A group of MongoDB documents (like a table in SQL).
- Document: A JSON-like structure that stores actual data (like a row in SQL).

Syntax to create database & collection using Node

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

async function main() {
  const client = new MongoClient("mongodb://localhost:27017");
  await client.connect();
  console.log("Connected");

  const db = client.db("myDatabase");
  const collection = db.collection("users");
```

```
    console.log("Database & Collection created");

    await client.close();
}
main();
```

- Find, Insert, Update & Delete using Node.js

① Insert:
- 'insertOne()' = Inserts a single document.
- 'insertMany()' = Inserts multiple documents.

② Find:
- 'findOne()' = Retrieves a single document.
- 'find()' = Retrieves multiple documents.

③ Update:
- 'updateOne()' = Updates a single document.
- 'updateMany()' = Updates multiple documents.

④ Delete:
- 'deleteOne()' = Deletes a single document:
- 'deleteMany()' = Deletes multiple documents.

CONCLUSION: Hence, we have successfully implemented a program to create database & collection in MongoDB using Node.js.

EXPERIMENT 10

AIM: Design Weather App using Flask.

THEORY:

- Flask

Flask is a light weight & powerful Python web frame-work used to build web applications quickly. It follows the WSGI (Web Server Gateway Interface) standard & does not require specific tools or libraries to get started.

- Lightweight & Minimalistic:

Flask has simple core, making it flexible & easy to extend.

- Built-in Dev Server:

Provides a debugger & automatic reloading during development.

- Jinja2 Templating Engine:

Allows embedding Python code inside HTML.

- URL Routing:

- It maps URLs to specific functions for handling requests.

- RESTful Support:

Flask is well-suited for building APIs & handling JSON data.

- Database Support:

It can integrate with database like SQLite, MySQL & MongoDB.

SYNTAX:

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def home():
    return "Hello, Flask!"

if __name__ == "__main__":
    app.run(debug=True)
```

CONCLUSION: Hence, we have successfully designed a Weather App using Flask.