EXPERIMENT-7

7 a) Course Name: Express.js
Module Name: Defining a route, Handling Routes, Route Parameters, Query Parameters Implement routing for the AdventureTrails application by embedding the necessary code in the routes/route.js file.

Aim: Defining a route, Handling Routes, Route Parameters, Query Parameters Implement routing for the AdventureTrails application by embedding the necessary code in the routes/route.js file.

Description:

Routing: The application object has different methods corresponding to each of the HTTP verbs (GET, POST, PUT, DELETE). These methods are used to receive HTTP requests.

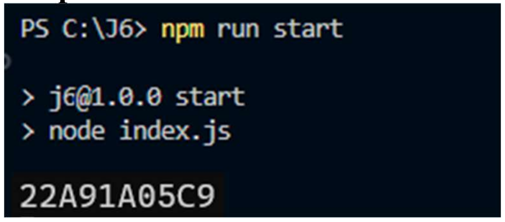
Syntax:router.method(path,handler) router: express instance or router instance **method:** one of the HTTP verbs**path:** is the route where request runs
handler: is the callback function that gets triggered whenever a request comes to a particular path for a matching request type

Program:

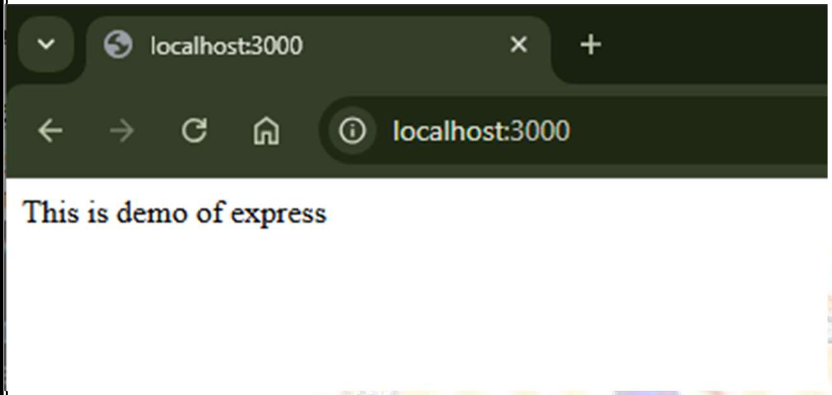
Index.js:

```
const express = require('express');
const { process_params } = require('express/lib/router');
const app = express()
app.get('/', (req, res) => {
   res.send("This is demo of express");
});
const products = [
   {id:1, name:"iphone"},
   {id:2, name:"samsung"},
   {id:3, name:"oppo"},
]
app.get('/prod',(req,res) => {
   res.json(products);
});
app.get('/product/:id',(req,res) => {
   const newData = parseInt(req.params.id);
   const item = products.find(i => i.id === newData);
   res.json(item);
   //res.json(products);
});

app.get('/productss',(req,res) => {
   const newData = parseInt(req.query.id);
   const item = products.find(i => i.id === newData);
   console.log(item);
   if(item)
   res.json(item);
   else
   res.status(404).send("Item not found");
});
console.log("22A91A05C9");
app.listen(3000);
```

Output:

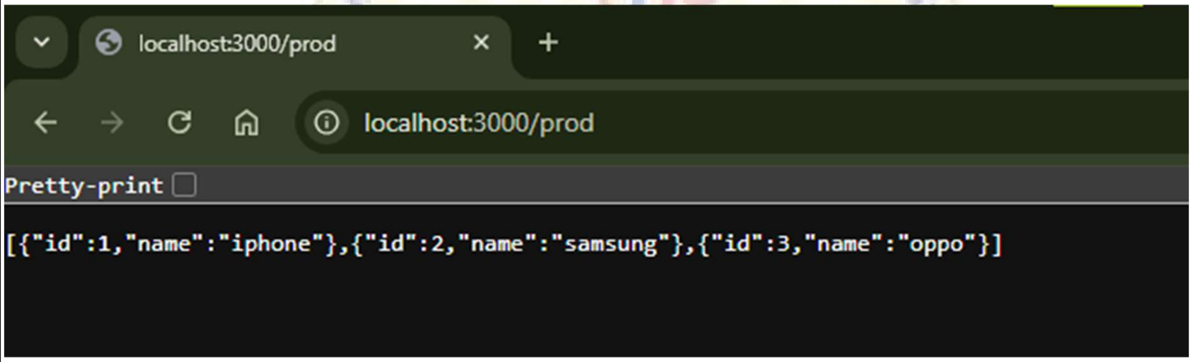


```
PS C:\J6> npm run start

> j6@1.0.0 start
> node index.js

22A91A05C9
```



localhost:3000

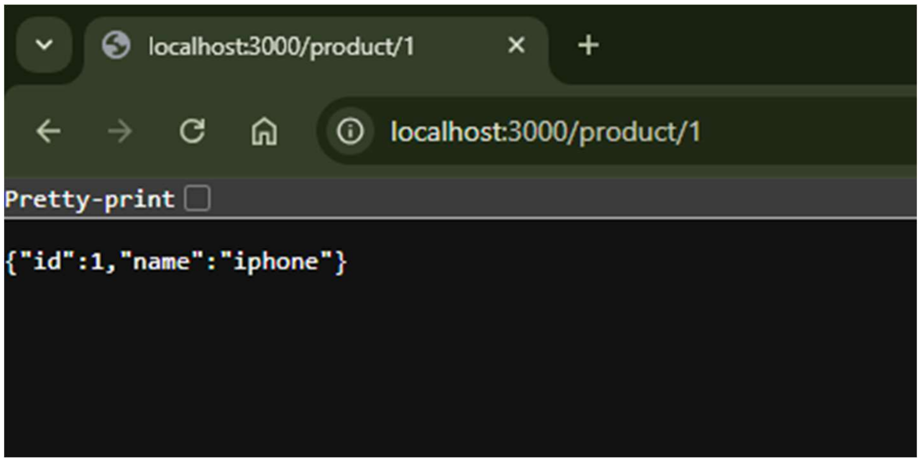
This is demo of express



localhost:3000/prod

Pretty-print

```
[{"id":1,"name":"iphone"},{"id":2,"name":"samsung"},{"id":3,"name":"oppo"}]
```



localhost:3000/product/1

Pretty-print

```
{"id":1,"name":"iphone"}
```

7 b) Course Name: Express.js
Module Name: How Middleware works, Chaining of Middlewares, Types of Middlewares In myNotes application: (i) we want to handle POST submissions. (ii) display customized error messages. (iii) perform logging.

Aim: How Middleware works, Chaining of Middlewares, Types of Middlewares In myNotes application: (i) we want to handle POST submissions. (ii) display customized error messages. (iii) perform logging.

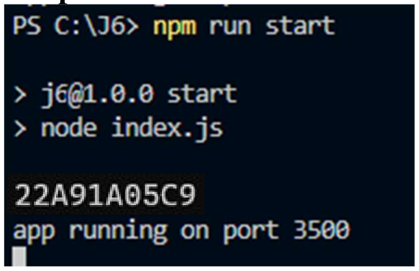
Description: A middleware can be defined as a function for implementing different cros-cutting concerns such as authentication, logging, etc.
The main arguments of a middleware function are the **request** object, **response** object, and also the **next** middleware function defined in the application.
A function defined as a middleware can execute any task mentioned below:

- Any code execution.
- Modification of objects - request and response.
- Call the next middleware function.
- End the cycle of request and response.

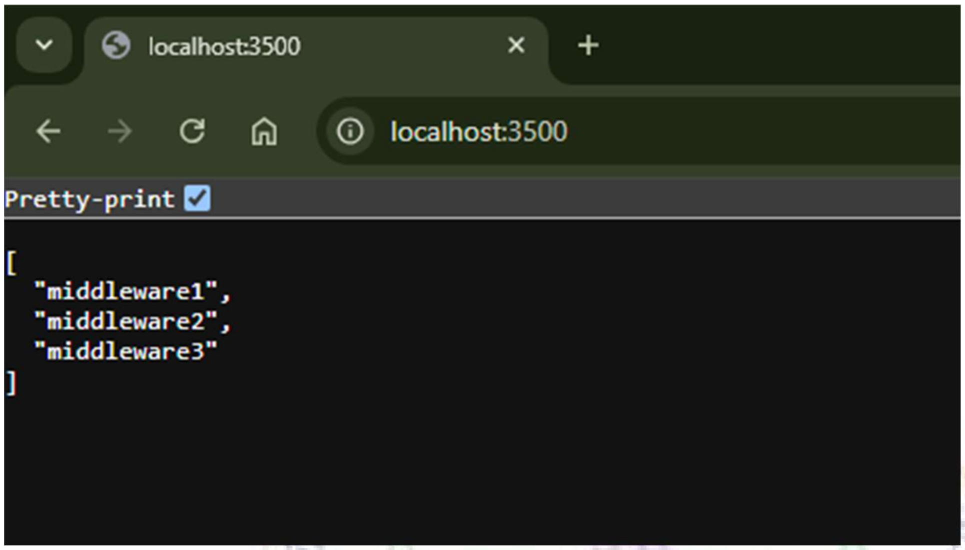
Example to Define a Middle Ware:
```
const mylogger = async (req, res, next) => {
        console.log(new Date(), req.method, req.url); next();
};
```

Program:
```
var express = require('express');
var app = express();
const PORT = 3500;

app.get("/", function(req, res, next){
  //middleware 1
  req.middlewares = ["middleware1"];
  next()
},
function(req, res, next){
  //middleware 2
  req.middlewares.push("middleware2")
  next()
},
function(req, res, next){
  //middleware 3
  req.middlewares.push("middleware3")
  res.json(req.middlewares);
})

app.listen(PORT, ()=>{
console.log('22A91A05C9');
console.log("app running on port "+PORT)
})
```

Output:



```
PS C:\J6> npm run start

> j6@1.0.0 start
> node index.js

22A91A05C9
app running on port 3500
```



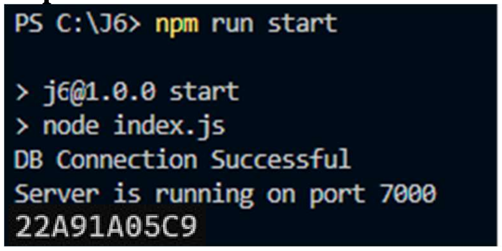
```
[
  "middleware1",
  "middleware2",
  "middleware3"
]
```

7 c) Course Name: Express.js
Module Name: Connecting to MongoDB with Mongoose, Validation Types and Defaults
Write a Mongoose schema to connect with MongoDB.
Aim: Write a Mongoose schema to connect with MongoDB.
Description:
Before we get into the specifics of validation syntax, please keep the following rules in mind:
•Validation is defined in the SchemaType
•Validation is middleware. Mongoose registers validation as a pre('save') hook on every schema by default.
•You can disable automatic validation before save by setting the validateBeforeSave option
•You can manually run validation using doc.validate(callback) or doc.validateSync()
•You can manually mark a field as invalid (causing validation to fail) by using doc.invalidate(...)
•Validators are not run on undefined values. The only exception is the required validator.
Program:
Index.js:

```
import express from "express";
import mongoose from "mongoose";
import dotenv from "dotenv";
const app = express();
dotenv.config();
const PORT = process.env.PORT || 7000;
const MONGOURL = process.env.MONGO_URL;
mongoose.connect(MONGOURL).then(()=>{
    console.log("DB Connection Successful");
    app.listen(PORT, ()=>{
        console.log("Server is running on port",PORT);
        console.log("22A91A05C9");
    });
}).catch((error)=>console.log(error));
```

.env file:

```
PORT = 8000
MONGO_URL = "mongodb://localhost:27017/MST"
```

Output:


```
PS C:\J6> npm run start

> j6@1.0.0 start
> node index.js
DB Connection Successful
Server is running on port 7000
22A91A05C9
```

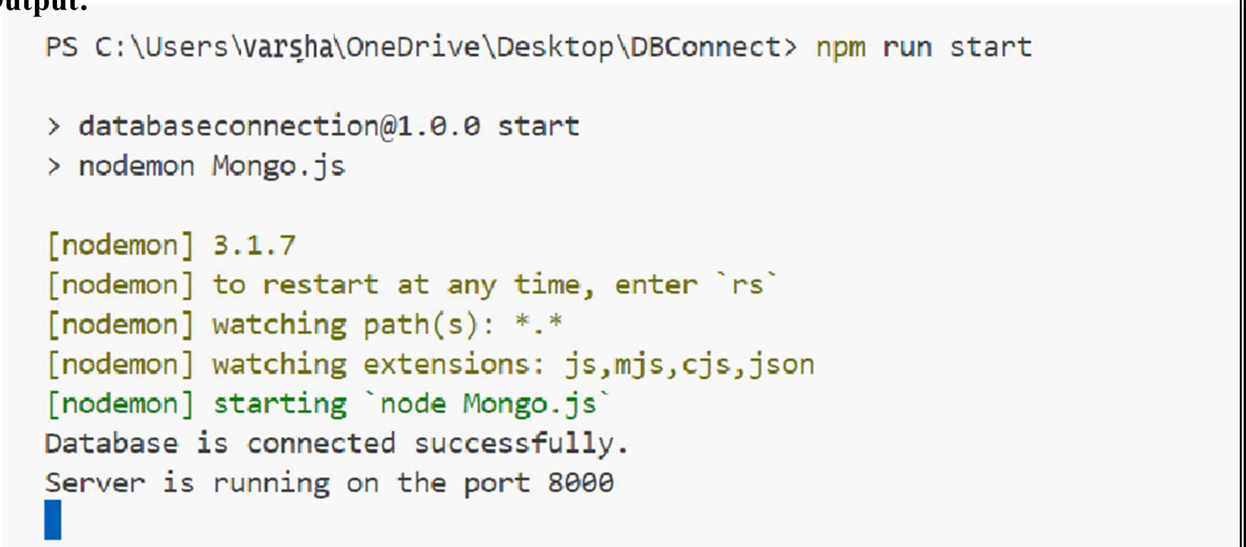
7 d) Course Name: Express.js
Module Name: Models
Write a program to wrap the Schema into a Model object.
Aim: Write a program to wrap the Schema into a Model object.
Description: Schema wrapping (@graphql-tools/wrap) creates a modified version of a schema that proxies, or "wraps", the original unmodified schema. This technique is particularly useful when the original schema cannot be changed, such as with remote schemas.
Schema wrapping works by creating a new "gateway" schema that simply delegates all operations to the original subschema. A series of transforms are applied that may modify the shape of the gateway schema and all proxied operations; these operational transforms may modify an operation prior to delegation, or modify the subschema result prior to its return.
Program:
Exp7d.js:

```
import express from "express"
import mongoose from "mongoose"
import dotenv from "dotenv"

const app = express();
dotenv.config();

const PORT = process.env.PORT || 7000;
const MONGOURL = process.env.MONGO_URL;

mongoose.connect(MONGOURL).then(()=>{
   console.log("Database is connected successfully.");
   app.listen(PORT, ()=> {
      console.log(`Server is running on the port ${PORT}`);
   });
})
.catch((error) => console.log(error));

const userSchema = new mongoose.Schema({
   name: String,
   age: Number
});
const userModel = mongoose.model("names",userSchema);

app.get("/getUsers", async(req,res) => {
   const userData = await userModel.find();
   res.json(userData);
})
```

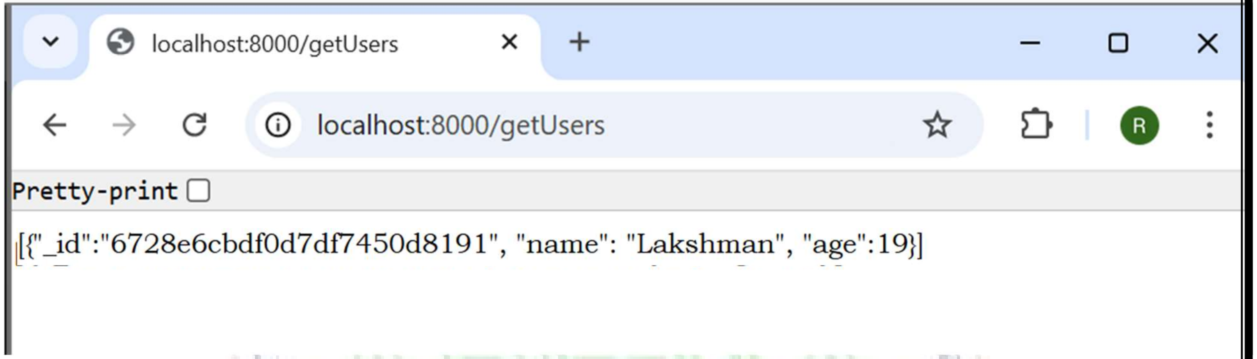
.env:

```
PORT=8000
MONGO_URL="mongodb://localhost:27017/DBConnect"
```

Output:



```
PS C:\Users\varsha\OneDrive\Desktop\DBConnect> npm run start

> databaseconnection@1.0.0 start
> nodemon Mongo.js

[nodemon] 3.1.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node Mongo.js`
Database is connected successfully.
Server is running on the port 8000
```



localhost:8000/getUsers

Pretty-print

```
[{"_id":"6728e6cbdf0d7df7450d8191", "name": "Lakshman", "age":19}]
```

EXPERIMENT-8

8 a) Course Name: Express.js
Module Name: CRUD Operations
Write a program to perform various CRUD (Create-Read-Update-Delete) operations using Mongoose library functions.

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

Description:

CRUD OPERATIONS

Create: We'll be setting up a post request to '/save' and we'll create a new student object with our model and pass with it the request data from Postman.Once this is done, we will use .save() to save it to the database.

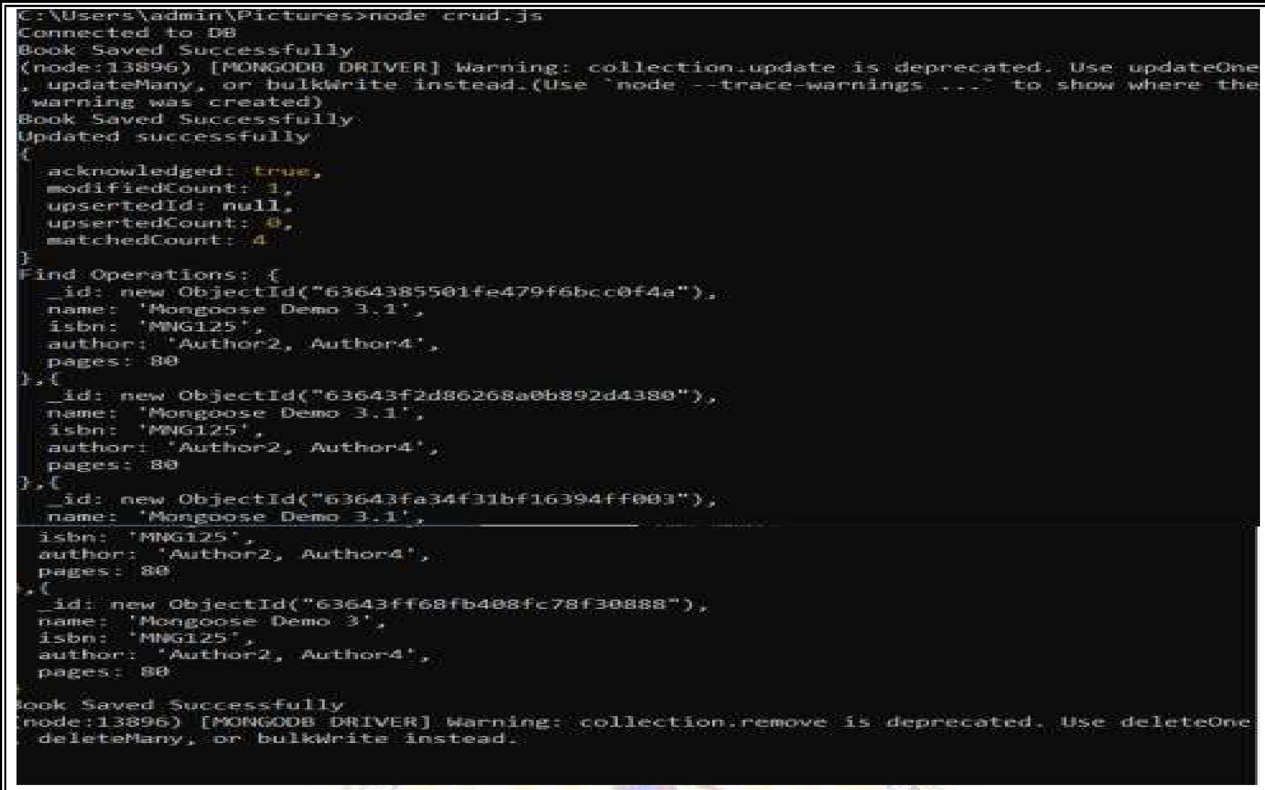
Retrieve: To retrieve records from a database collection we make use of the .find() function.

Update: Just like with the delete request, we'll be using the _id to target the correct item. .findByIdAndUpdate() takes the target's id, and the request data you want to replace it with

Program:

```
var mongoose = require("mongoose");
var dbHost = "mongodb://localhost:27017/test";
mongoose.connect(dbHost);
var bookSchema = mongoose.Schema({
name: String,
//Also creating index on field isbn
isbn: {type: String, index: true},
author: String,
pages: Number
});
var Book = mongoose.model("Book", bookSchema, "mongoose_demo");
var db = mongoose.connection;
db.on("error", console.error.bind(console, "connection error:"));
db.once("open", function(){
console.log("Connected to DB");
var book1 = new Book({
   name:"Mongoose Demo 1",
   isbn: "MNG123",
   author: "Author1, Author2",
   pages: 123
   });
   book1.save(function(err){
   if ( err )
   throw err;
   console.log("Book Saved Successfully");
   });
var book2 = new Book({
      name:"Mongoose Demo 2",
      isbn: "MNG124",
      author: "Author2, Author3",
        pages: 90
});
var book3 = new Book({
    name:"Mongoose Demo 3",
    isbn: "MNG125",
    author: "Author2, Author4",
    pages: 80
   });
   book3.save(function(err){
   if ( err )
   throw err;
   console.log("Book Saved Successfully");
   queryBooks();
   updateBook();
   });
});
var queryBooks = function(){
    Book.find({pages : {$lt:100}}, "name isbn author pages",
    function(err, result){
    if ( err )
    throw err;
    console.log("Find Operations: " + result);
    }); }
    var updateBook = function(){
    Book.update({isbn : {$eq: "MNG125"}}, {$set: {name: "Mongoose Demo 3.1"}},
function(err, result){
    console.log("Updated successfully");
    console.log(result);
    });}
    var deleteBook = function(){
    Book.remove({isbn:{$eq: "MNG124"}}).exec();
}
```

Output:



```
C:\Users\admin\Pictures>node crud.js
Connected to DB
Book Saved Successfully
(node:13896) [MONGODB DRIVER] Warning: collection.update is deprecated. Use updateOne
, updateMany, or bulkWrite instead.(Use `node --trace-warnings ...` to show where the
 warning was created)
Book Saved Successfully
Updated successfully
{
  acknowledged: true,
  modifiedCount: 1,
  upsertedId: null,
  upsertedCount: 0,
  matchedCount: 4
}
Find Operations: {
  _id: new ObjectId("6364385501fe479f6bcc0f4a"),
  name: 'Mongoose Demo 3.1',
  isbn: 'MNG125',
  author: 'Author2, Author4',
  pages: 80
},{
  _id: new ObjectId("63643f2d86268a0b892d4380"),
  name: 'Mongoose Demo 3.1',
  isbn: 'MNG125',
  author: 'Author2, Author4',
  pages: 80
},{
  _id: new ObjectId("63643fa34f31bf16394ff003"),
  name: 'Mongoose Demo 3.1',
  isbn: 'MNG125',
  author: 'Author2, Author4',
  pages: 80
},{
  _id: new ObjectId("63643ff68fb408fc78f30888"),
  name: 'Mongoose Demo 3',
  isbn: 'MNG125',
  author: 'Author2, Author4',
  pages: 80
}
Book Saved Successfully
(node:13896) [MONGODB DRIVER] Warning: collection.remove is deprecated. Use deleteOne
, deleteMany, or bulkWrite instead.
```

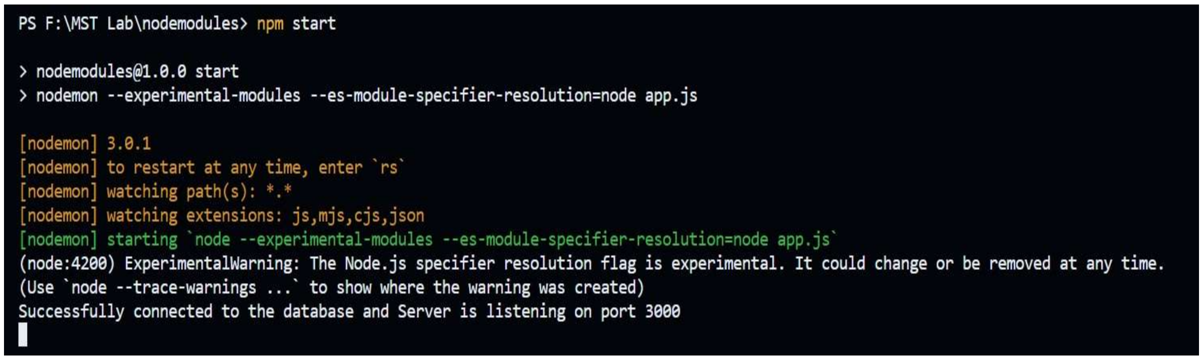
8 b) Course Name: Express.js
Module Name: API Development
In the myNotes application, include APIs based on the requirements provided. (i) API should fetch the details of the notes based on a notesID which is provided in the URL. Test URL - http://localhost:3000/notes/7555 (ii) API should update the details based on the name which is provided in the URL and the data in the request body. Test URL - http://localhost:3000/notes/Mathan Note: Only one document in the collection needs to be updated. (iii) API should delete the details based on the name which is provided in the URL. Test URL - http://localhost:3000/notes/Mathan Note: Only one document in the collection needs to be deleted.
Aim: API Development. In the myNotes application, include APIs based on the requirements provided.
Program:

```
import express from "express";
import mongoose from "mongoose";
const app = express();
app.use(express.json());
// Creation of Notes Schema
const Schema=mongoose.Schema;
const NoteSchema = new Schema({
   title:{
      type: String
   },
   content:{
      type:String
   }
});
// Creating Note model
const Note=mongoose.model("Note",NoteSchema);
// Creation of a new note using app.post
app.post("/notes/create",(req, res) => {
   if(!req.body.content) {
      return res.status(400).send({
         message: "Note content can not be empty"
      });
   }
   const note = new Note({
      title: req.body.title || "Untitled Note",
      content: req.body.content
   });
   note.save()
   .then(data => {
      res.send(data);
   }).catch(err => {
      res.status(500).send({
         message: err.message || "Some error occurred while creating the Note."
```

```
        });
    });
});
// Finding an existing note by it's ID using app.post
app.post("/notes/:noteId",(req, res) => {
    Note.findById(req.params.noteId)
    .then(note => {
        if(!note) {
            return res.status(404).send({
                message: "Note not found with id " + req.params.noteId
            });
        }
        return res.send(note);
    }).catch(err => {
        return res.status(500).send({
            message: "Error retrieving note with id " + req.params.noteId
        });
    });
});
// Updating an existing note details using it's title with app.put
app.put("/notes/:title",async(req, res) => {
    const title=req.params.title;
    let exists;
    try{
        exists=await Note.findOne({"title":title});
    }
    catch(err){
        return console.log(err);
    }
    if(!exists){
        return res.status(500).json({message:"Notes not found"});
    }
    const noteId=exists._id;
    Note.findByIdAndUpdate(noteId, {
        title: req.body.title || "Untitled Note",
        content: req.body.content
    }, {new: true})
    .then(note => {
        if(!note) {
            return res.status(404).send({
                message: "Note not found with id " + req.params.noteId
            });
        }
        res.send(note);
    }).catch(err => {
        return res.status(500).send({
         message: "Error updating note with id " + req.params.noteId
      });
   });
});

// Deleting an existing note details using it's title with app.delete
app.delete("/notes/:title",async(req, res) => {
   const title=req.params.title;
   let exists;
   try{
      exists=await Note.findOne({"title":title});
   }
   catch(err){
      return console.log(err);
   }
   if(!exists){
      return res.status(500).json({message:"Notes not found"});
   }
   const noteId=exists._id;
   Note.findByIdAndDelete(noteId)
   .then(note => {
      if(!note) {
         return res.status(404).send({
            message: "Note not found with id " + req.params.noteId
         });
      }
      res.send({message: "Note deleted successfully!"});
   }).catch(err => {
      if(err.kind === 'ObjectId' || err.name === 'NotFound') {
         return res.status(404).send({
            message: "Note not found with id " + req.params.noteId
         });
      }
      return res.status(500).send({
         message: "Could not delete note with id " + req.params.noteId
      });
   });
});
// MongoDB Atlas connection
const
url="mongodb+srv://admin:admin@cluster0.zsygsqy.mongodb.net/?retryWrites=true&w=majori
ty";
mongoose.connect(url)
.then(() => app.listen(3000))
.then(() => {
   console.log("Successfully connected to the database and Server is listening on port 3000");
```

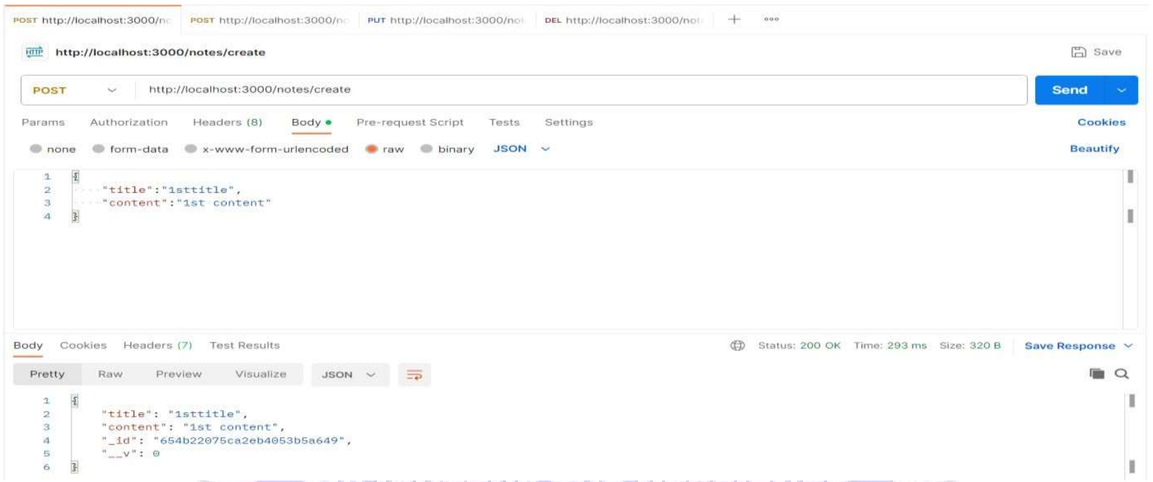
```
}).catch(err => {
    console.log('Could not connect to the database. Exiting now...', err);
});
```

Output:



```
PS F:\MST Lab\nodemodules> npm start

> nodemodules@1.0.0 start
> nodemon --experimental-modules --es-module-specifier-resolution=node app.js

[nodemon] 3.0.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node --experimental-modules --es-module-specifier-resolution=node app.js`
(node:4200) ExperimentalWarning: The Node.js specifier resolution flag is experimental. It could change or be removed at any time.
(Use `node --trace-warnings ...` to show where the warning was created)
Successfully connected to the database and Server is listening on port 3000
```

CREATION OF A NOTE:



POST http://localhost:3000/notes/create

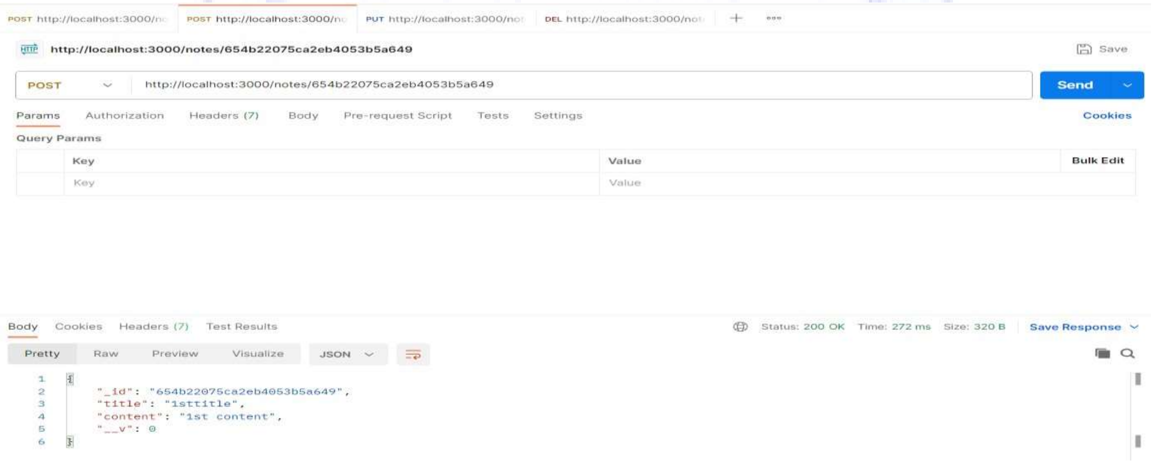
Body (raw, JSON):

```
{
    "title":"1sttitle",
    "content":"1st content"
}
```

Status: 200 OK Time: 293 ms Size: 320 B

```
{
    "title": "1sttitle",
    "content": "1st content",
    "_id": "654b22075ca2eb4053b5a649",
    "__v": 0
}
```

FINDING A NOTE BY IT'S ID:

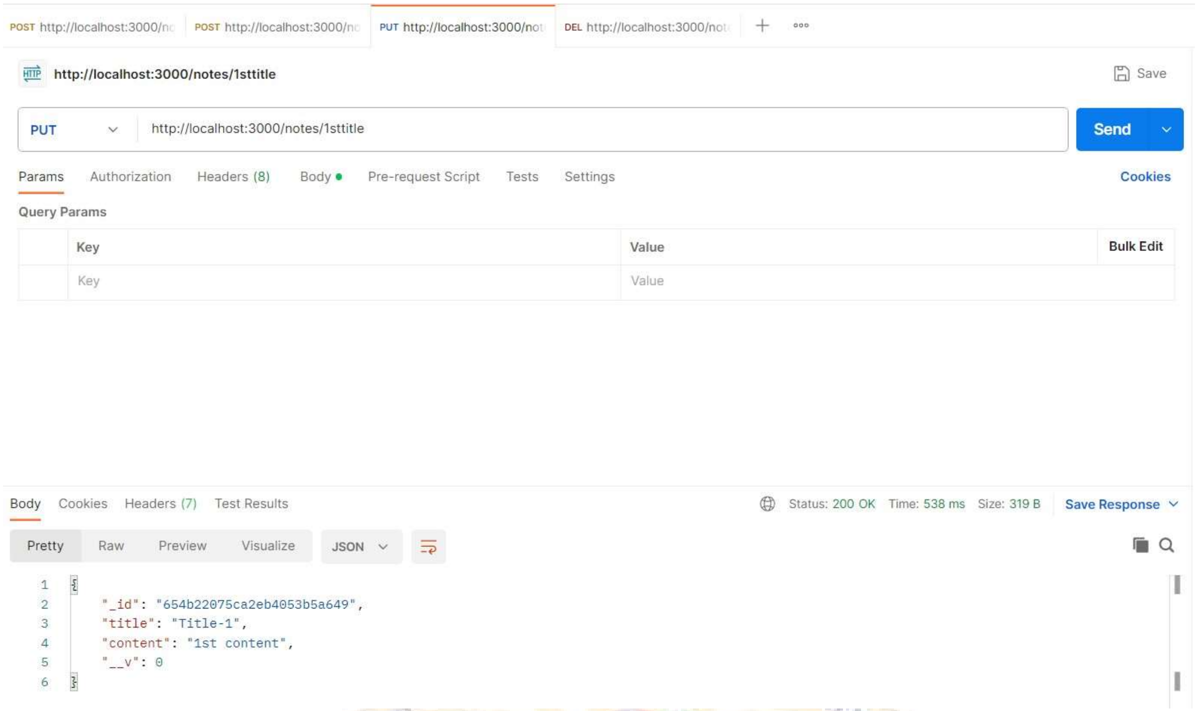


POST http://localhost:3000/notes/654b22075ca2eb4053b5a649

Status: 200 OK Time: 272 ms Size: 320 B

```
{
    "_id": "654b22075ca2eb4053b5a649",
    "title": "1sttitle",
    "content": "1st content",
    "__v": 0
}
```

UPDATION OF A NOTES BY IT'S TITLE:



http://localhost:3000/notes/1sttitle

PUT http://localhost:3000/notes/1sttitle

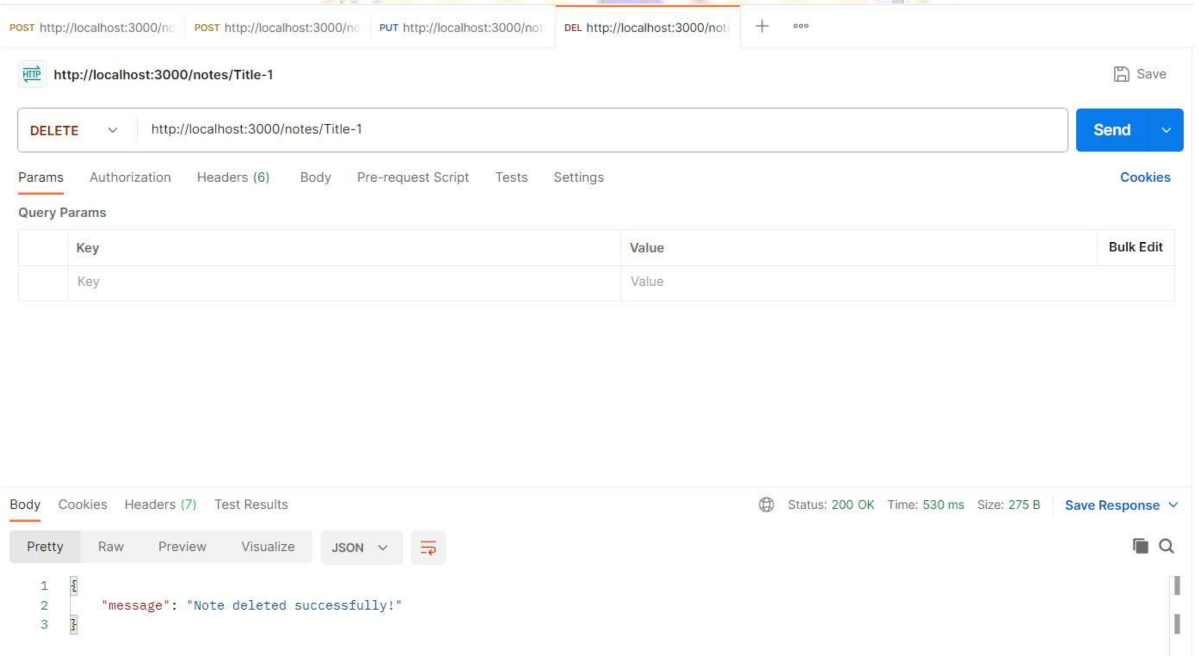
Query Params

Key	Value
Key	Value

Status: 200 OK Time: 538 ms Size: 319 B

```
{
    "_id": "654b22075ca2eb4053b5a649",
    "title": "Title-1",
    "content": "1st content",
    "__v": 0
}
```

DELETION OF A NOTES BY IT'S TITLE:



http://localhost:3000/notes/Title-1

DELETE http://localhost:3000/notes/Title-1

Query Params

Key	Value
Key	Value

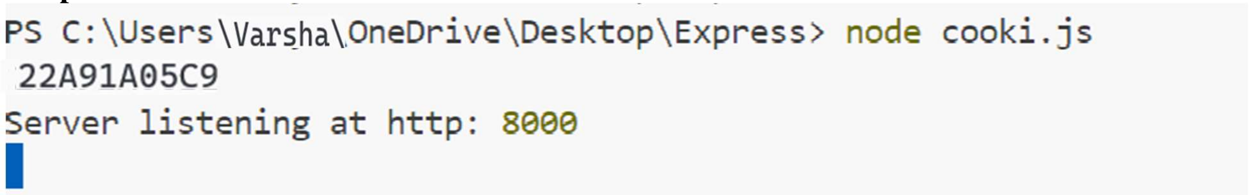
Status: 200 OK Time: 530 ms Size: 275 B

```
{
    "message": "Note deleted successfully!"
}
```

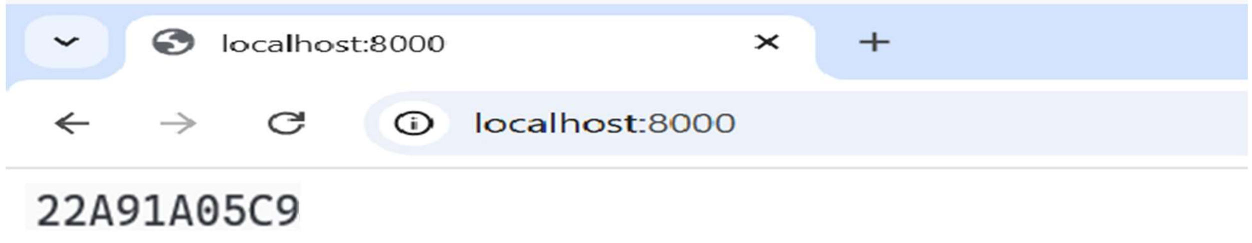
8 c) Course Name: Express.js
Module Name: Why Session management, Cookies
Write a program to explain session management using cookies.
Aim: Write a program to explain session management using cookies
Program:

```
var express = require('express');
var cookieParser = require('cookie-parser');
var app = express();
app.use(cookieParser());
app.get('/cookieset',function(req, res){
   res.cookie('cookie_name', 'cookie_value');
   res.cookie('College', 'Aditya');
   res.cookie('Branch', 'Cse');
   res.status(200).send('Cookie is set');
});
app.get('/cookieget', function(req, res) {
res.status(200).send(req.cookies);
});
app.get('/', function (req, res)
{
res.status(200).send('22A91A05C9');
});
var server = app.listen(8000, function ()
 {
 var host = server.address().address;
 var port = server.address().port;
 console.log("22A91A05C9")
console.log('Server listening at http:', port);
});
```

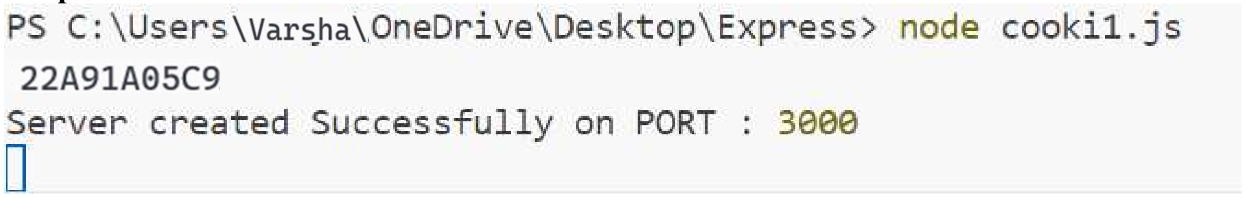
Output:

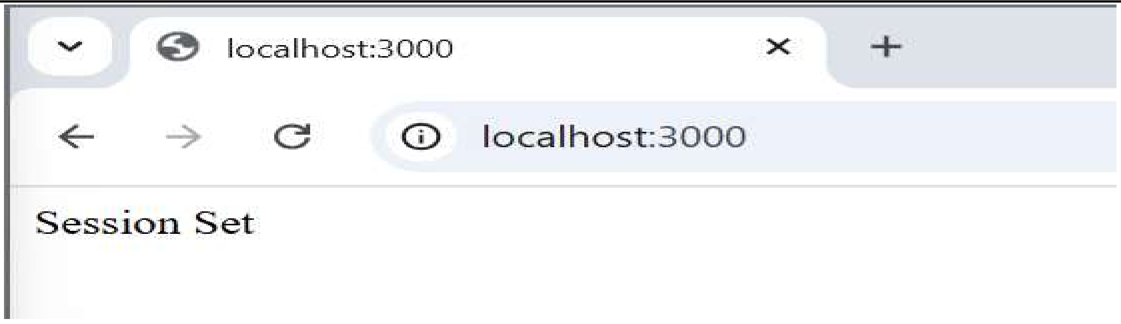


```
PS C:\Users\Varsha\OneDrive\Desktop\Express> node cooki.js
 22A91A05C9
Server listening at http: 8000
```

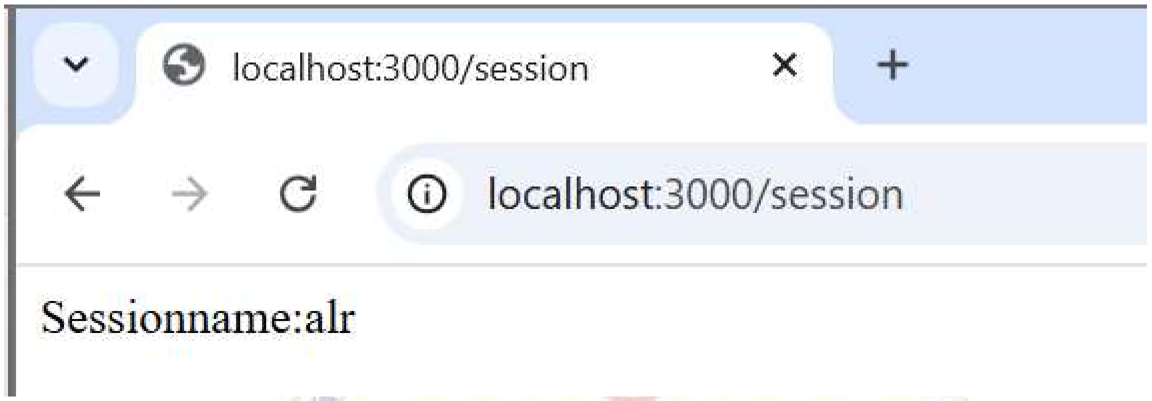


```
localhost:8000

22A91A05C9
```

8 d) Course Name: Express.js
Module Name: Sessions
Write a program to explain session management using sessions
Aim: Write a program to explain session management using sessions.
Description:
A website is based on the HTTP protocol. HTTP is a stateless protocol which means at the end of every request and response cycle, the client and the server forget about each other. This is where the session comes in. A session will contain some unique data about that client to allow the server to keep track of the user's state. In session-based authentication, the user's state is stored in the server's memory or a database.
The following libraries will help us setup a Node.js session. Express - a web framework for Node.js used to create HTTP web servers. Express provides an easy-to-use API to interact with the webserver. Express-session - an HTTP server-side framework used to create and manage a session middleware. This tutorial is all about sessions. Thus Express-session library will be the main focus. Cookie-parser - used to parse cookie header to store data on the browser whenever a session is established on the server-side
Program:
```
const express = require("express")
const session = require('express-session')
const app = express()
var PORT = process.env.port || 3000
app.use(session({
   secret: 'Your_Secret_Key',
   resave: true,
   saveUninitialized: true
}))
app.get("/", function (req, res) {
   req.session.name = 'Sessionname:alr'
   res.send("Session Set")
})
app.get("/session", function (req, res) {
   var name = req.session.name
   res.send(name)
})
app.listen(PORT, function (error) {
   if (error) throw error
   console.log("22A91A05C9");
   console.log("Server created Successfully on PORT :", PORT)
})
```
Output:

```
PS C:\Users\Varsha\OneDrive\Desktop\Express> node cooki1.js
 22A91A05C9
Server created Successfully on PORT : 3000
```



```
Session Set
```



```
Sessionname:alr
```

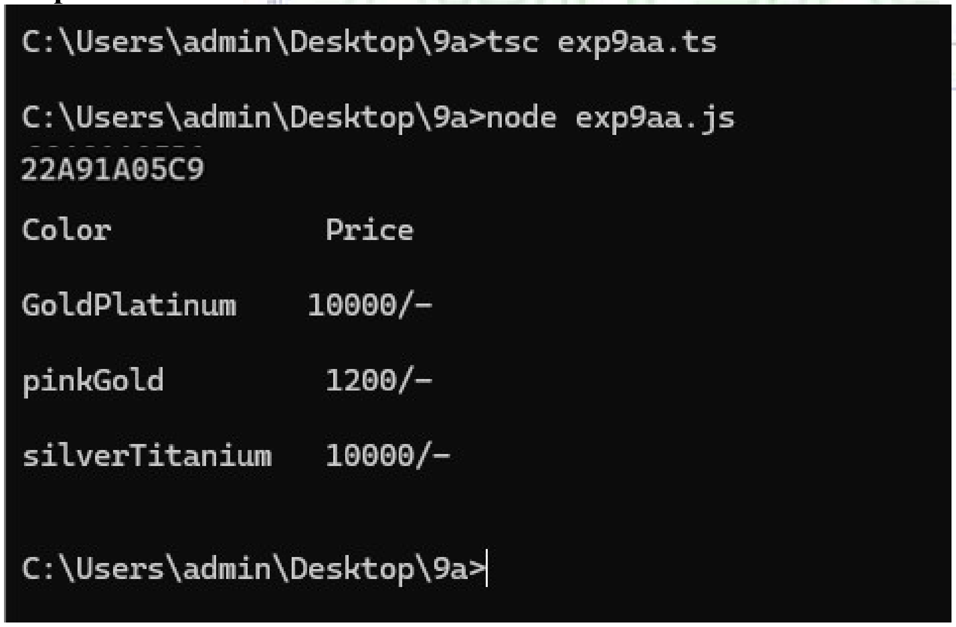
EXPERIMENT-9

9 a) Course Name: Typescript
Module Name: Basics of TypeScript
On the page, display the price of the mobile-based in three different colors. Instead of using the number in our code, represent them by string values like GoldPlatinum, PinkGold, SilverTitanium.

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

Description: TypeScript is a syntactic superset of JavaScript which adds static typing.
This basically means that TypeScript adds syntax on top of JavaScript, allowing developers to add types. TypeScript lets you write JavaScript the way you really want to. TypeScript is a typed superset of JavaScript that compiles to plain JavaScript. TypeScript is pure object oriented with classes, interfaces and statically typed like C# or Java. The popular JavaScript framework Angular 2.0 is written in TypeScript. Mastering TypeScript can help programmers to write object-oriented programs and have them compiled to JavaScript, both on server side and client side

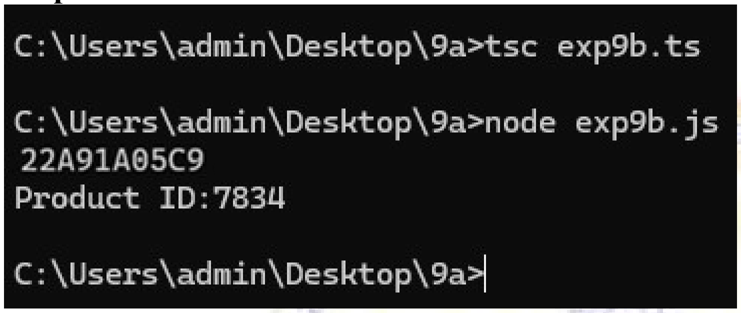
Program:
```
const obj:{GoldPlatinum:String}={GoldPlatinum:"10000/-"}
const obj1:{pinkGold:String}={pinkGold:"1200/-"}
const obj2:{SilverTitanium:String}={SilverTitanium:"10000/-"}
console.log("22A91A05C9\n");
console.log("Color\t\t  Price\n");
console.log("GoldPlatinum \t"+ obj.GoldPlatinum+"\n");
console.log("pinkGold\t "+obj1.pinkGold+"\n");
console.log("silverTitanium\t "+obj2.SilverTitanium+"\n");
```
Output:


```
C:\Users\admin\Desktop\9a>tsc exp9aa.ts

C:\Users\admin\Desktop\9a>node exp9aa.js
22A91A05C9

Color            Price

GoldPlatinum    10000/-

pinkGold         1200/-

silverTitanium   10000/-


C:\Users\admin\Desktop\9a>
```

9 b) Course Name: Typescript
Module Name: Function
Define an arrow function inside the event handler to filter the product array with the selected product object using the productId received by the function. Pass the selected product object to the next screen.
Aim: Define an arrow function inside the event handler to filter the product array with the selected product object using the productId received by the function. Pass the selected product object to the next screen
Program:

```
var getProductDetails = (productId: number): string => {
    return "Product ID:"+productId
}
console.log("22A91A05C9");
console.log(getProductDetails(7834));
```

Output:



```
C:\Users\admin\Desktop\9a>tsc exp9b.ts

C:\Users\admin\Desktop\9a>node exp9b.js
22A91A05C9
Product ID:7834

C:\Users\admin\Desktop\9a>
```

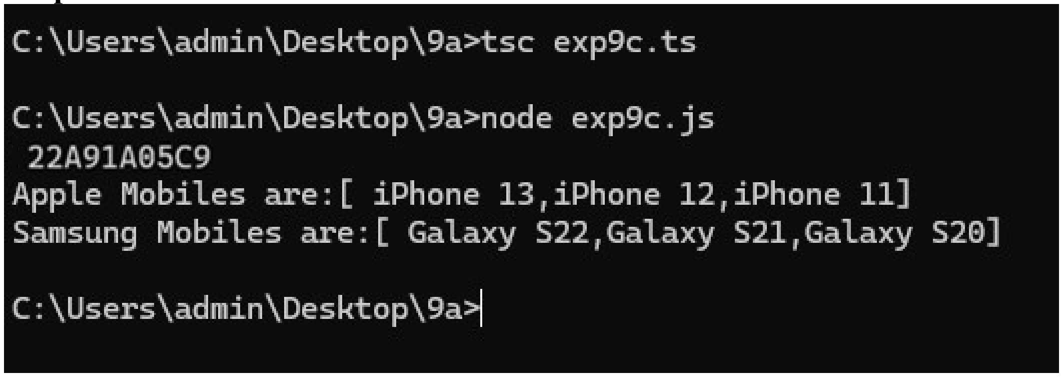
9 c) Course Name: Typescript
Module Name: Parameter Types and Return Types
Consider that developer needs to declare a function - getMobileByVendor which accepts string as input parameter and returns the list of mobiles.
Aim: Consider that developer needs to declare a function -getMobileByVendor which accepts string as input parameter and returns the list of mobiles.
Description: Functions are the basic building block of any application which holds some business logic. The process of creating a function in TypeScript is similar to the process in JavaScript. In functions, parameters are the values or arguments that passed to a function. The TypeScript, compiler accepts the same number and type of arguments as defined in the function signature. If the compiler does not match the same parameter as in the function signature, then it will give the compilation error.
We can categorize the function parameter into the three types:

- Optional Parameter
- Default Parameter
- Rest Parameter

Function return type in TypeScript is nothing but the value which we want to return from the function. Function return type used when we return value from the function. We can return any type of value from the function or nothing at all from the function in TypeScript. Some of the return types is a string, number, object or any, etc. If we do not return the expected value from the function, then we will have an error and exception.
Program:

```
function getMobileByVendor (manfact:string):string[]{
   let mobilesList:string[];
   if (manfact === "Samsung"){
      mobilesList = ["Galaxy S22", "Galaxy S21", "Galaxy S20"];
      return mobilesList;
   }
   else if (manfact === "Apple"){
      mobilesList = ["iPhone 13", "iPhone 12", "iPhone 11"];
      return mobilesList;
   }
   else{
      mobilesList = ["Vivo V25","Vivo T1 5g"];
      return mobilesList;
   }
}
console.log("22A91A05C9");
console.log("Apple Mobiles are:[ "+getMobileByVendor ('Apple')+']');
console.log("Samsung Mobiles are:[ "+getMobileByVendor ("Samsung")+']');
```

Output:

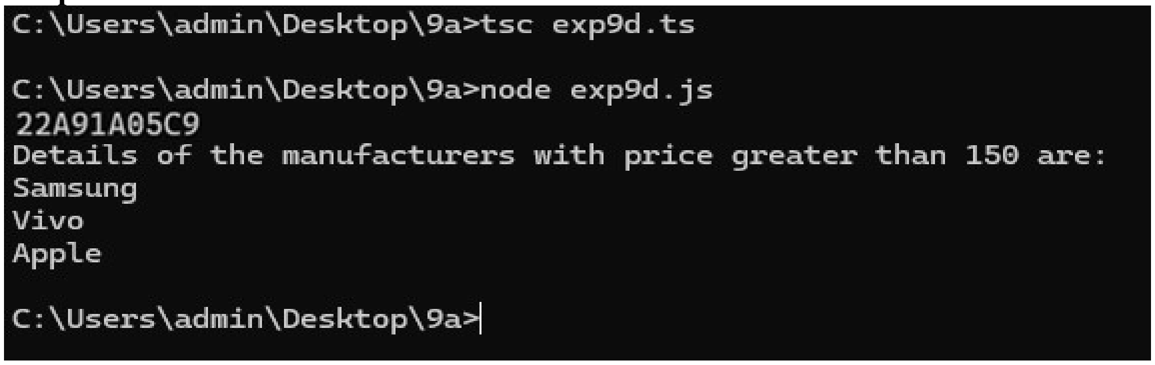


```
C:\Users\admin\Desktop\9a>tsc exp9c.ts

C:\Users\admin\Desktop\9a>node exp9c.js
 22A91A05C9
Apple Mobiles are:[ iPhone 13,iPhone 12,iPhone 11]
Samsung Mobiles are:[ Galaxy S22,Galaxy S21,Galaxy S20]

C:\Users\admin\Desktop\9a>
```

9 d) Course Name: Typescript
Module Name: Arrow Function
Consider that developer needs to declare a manufacturer's array holding 4 objects with id and price as a parameter and needs to implement an arrow function - myfunction to populate the id parameter of manufacturers array whose price is greater than or equal to 150 dollars then below mentioned code-snippet would fit into this requirement.
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 or equal to 150 dollars then below mentioned code snippet would fit into this requirement.
Description:
Arrow Function:
Fat arrow notations are used for anonymous functions i.e for function expressions. They are also called lambda functions in other languages.
Syntax:
(param1, param2, ..., paramN) => expression
Using fat arrow =>, we dropped the need to use the function keyword. Parameters are passed in the parenthesis (), and the function expression is enclosed within the curly brackets { }.
Program:

```
var manufacturers = [{ id: 'Samsung', price: 150 },
   { id: 'Vivo', price: 200 },
   { id: 'Apple', price: 500 },
   { id: 'Oppo', price: 100 }];
   var test;
   console.log('22A91A05C9');
   console.log("Details of the manufacturers with price greater than 150 are:");
   function myFunction() {
      test = manufacturers.filter((m) =>m.price >= 150);
      for (var item of test) {
         console.log(item.id);
      }
   }
myFunction();
```

Output:



```
C:\Users\admin\Desktop\9a>tsc exp9d.ts

C:\Users\admin\Desktop\9a>node exp9d.js
22A91A05C9
Details of the manufacturers with price greater than 150 are:
Samsung
Vivo
Apple

C:\Users\admin\Desktop\9a>
```

9 e) Course Name: Typescript
Module Name: Optional and Default Parameters
Declare a function - getMobileByManufacturer with two parameters namely manufacturer and id, where manufacturer value should passed as Samsung and id parameter should be optional while invoking the function, if id is passed as 101 then this function should return Moto mobile list and if manufacturer parameter is either Samsung/Apple then this function should return respective mobile list and similar to make Samsung as default Manufacturer. Below mentioned code-snippet would fit into this requirement.

Aim: Optional and Default Parameters Declare a function - getMobileByManufacturer with two parameters namely manufacturer and id, where manufacturer value should passed as Samsung and id parameter should be optional while invoking the function, if id is passed as 101 then this function should return Moto mobile list and if manufacturer parameter is either Samsung/Apple then this function should return respective mobile list and similar to make Samsung as default Manufacturer.

Description:

Optional Parameters

In TypeScript, every parameter is assumed to be required by the function. You can add a ? at the end of a parameter name to set it as optional.

For example, the lastName parameter of this function is optional:

```
function buildName(firstName: string, lastName?: string) {
  // ...
}
```

Optional parameters must come after all non-optional parameters:

```
function buildName(firstName?: string, lastName: string) // Invalid
```

Default Parameters

If the user passes undefined or doesn't specify an argument, the default value will be assigned. These are called default-initialized parameters.

For example, "Smith" is the default value for the lastName parameter.

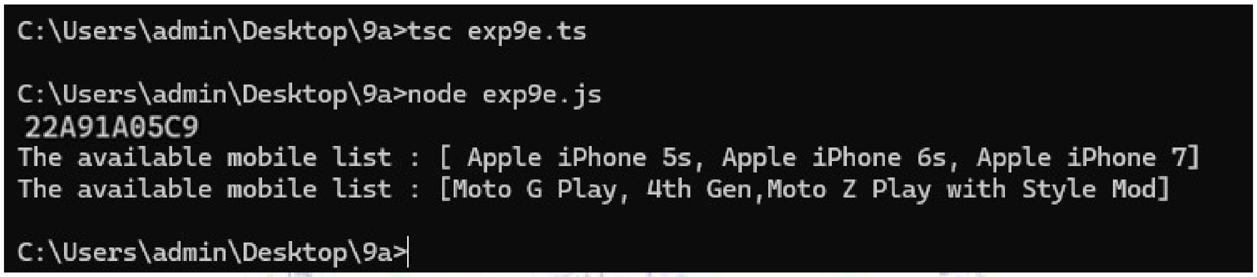
```
function buildName(firstName: string, lastName = "Smith") {
  // ...
}
buildName('foo', 'bar');      // firstName == 'foo', lastName == 'bar'
buildName('foo');             // firstName == 'foo', lastName == 'Smith'
buildName('foo', undefined);  // firstName == 'foo', lastName == 'Smith'
```

Program:

```
function getMobileByManufacturer(manufacturer: string = 'Samsung', id?: number): string[] {
  let mobileList: string[];
  if (id) {
    if (id === 101) {
      mobileList = ['Moto G Play, 4th Gen', 'Moto Z Play with Style Mod'];
      return mobileList;
    }
  }
  if (manufacturer === 'Samsung') {
    mobileList = [' Samsung Galaxy S6 Edge', ' Samsung Galaxy Note 7',
      ' Samsung Galaxy J7 SM-J700F'];
```

```
        return mobileList;
    }
    else if (manufacturer === 'Apple') {
        mobileList = [' Apple iPhone 5s', ' Apple iPhone 6s', ' Apple iPhone 7'];
        return mobileList;
    }
    else {
        mobileList = [' Nokia 105', ' Nokia 230 Dual Sim'];
        return mobileList;
    }
}
console.log("22A91A05C9");
console.log('The available mobile list : [' + getMobileByManufacturer('Apple')+']');
console.log('The available mobile list : [' + getMobileByManufacturer(undefined, 101)+']')
```

Output:



```
C:\Users\admin\Desktop\9a>tsc exp9e.ts

C:\Users\admin\Desktop\9a>node exp9e.js
22A91A05C9
The available mobile list : [ Apple iPhone 5s, Apple iPhone 6s, Apple iPhone 7]
The available mobile list : [Moto G Play, 4th Gen,Moto Z Play with Style Mod]

C:\Users\admin\Desktop\9a>
```

EXPERIMENT-10

10 a) Course Name: Typescript
Module Name: Rest Parameter
Implement business logic for adding multiple Product values into a cart variable which is type of string array.

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

Description:
TypeScript - Rest Parameters

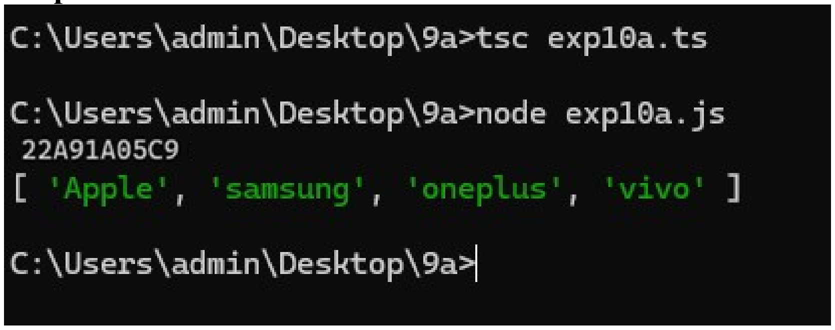
In the function chapter, you learned about functions and their parameters. TypeScript introduced rest parameters to accommodate n number of parameters easily. When the number of parameters that a function will receive is not known or can vary, we can use rest parameters. In JavaScript, this is achieved with the "arguments" variable. However, with TypeScript, we can use the rest parameter denoted by ellipsis ....

We can pass zero or more arguments to the rest parameter. The compiler will create an array of arguments with the rest parameter name provided by us

Program:

```
const cart:string[] = []
const pushToCart = (item:string) => {
   cart.push(item);
};
function addToCart(ProductName:string[]):string[] {
        for(const item of ProductName){
           pushToCart(item)
        }
        return cart;
}
console.log("22A91A05C9");
console.log(addToCart(["Apple","samsung","oneplus","vivo"]));
```

Output:



```
C:\Users\admin\Desktop\9a>tsc exp10a.ts

C:\Users\admin\Desktop\9a>node exp10a.js
 22A91A05C9
[ 'Apple', 'samsung', 'oneplus', 'vivo' ]

C:\Users\admin\Desktop\9a>
```

10 b) Course Name: Typescript
Module Name: Creating an Interface
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.

Aim: Declare an interface named - Product with two properties like productId and productName with a number and string data type and need to implement logic to populate the Product details.

Description: An interface is a syntactical contract that an entity should conform to. In other words, an interface defines the syntax that any entity must adhere to. Interfaces define properties, methods, and events, which are the members of the interface. Interfaces contain only the declaration of the members. It is the responsibility of the deriving class to define the members. It often helps in providing a standard structure that the deriving classes would follow. Let's consider an object −

```
var person = {
  FirstName:"Tom",
  LastName:"Hanks",
  sayHi: ()=>{ return "Hi"}
};
```

If we consider the signature of the object, it could be −

```
{
  FirstName:string,
  LastName:string,
  sayHi()=>string
}
```

To reuse the signature across objects we can define it as an interface.

Declaring Interfaces

The interface keyword is used to declare an interface. Here is the syntax to declare an interface −

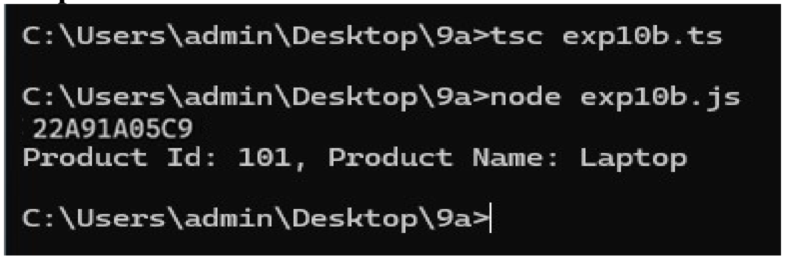
Syntax

```
interface interface_name {
}
```

Program:

```
interface Product{productId:number,productName:string;}
function getProductDetails(productobj:Product):string{
   return `Product Id: ${productobj.productId}, Product Name: ${productobj.productName}`;
}
const prodObject = {productId:101,productName:"Laptop"};
const productDetails:string = getProductDetails(prodObject);
console.log("22A91A05C9");
console.log(productDetails);
```

Output:



```
C:\Users\admin\Desktop\9a>tsc exp10b.ts

C:\Users\admin\Desktop\9a>node exp10b.js
 22A91A05C9
Product Id: 101, Product Name: Laptop

C:\Users\admin\Desktop\9a>
```

10 c) Course Name: Typescript
Module Name: Duck Typing
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.

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:

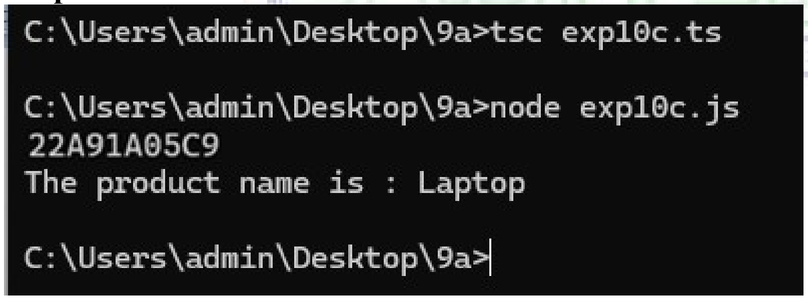
Duck-Typing

Duck-Typing is a method/rule used by TypeScript to check type compatibility for more complex variable types. This method is used to compare two objects by determining whether they have the same type of matching names or not. It means we can't change a variable's signature.
The duck-typing technique in TypeScript is used to compare two objects by determining if they have the same type matching properties and objects members or not. For example, if we assign an object with two properties and a method and the second object is only assigned with two properties. The typescript compiler raises a compile-time error in such situations when we create a variable of object1 and assign it a variable of the second object type.

Program:

```
interface Product { productId: number; productName: string; }
function getProductDetails(productobj: Product): string {
   return 'The product name is : ' + productobj.productName;
}
const prodObject = { productId: 1001, productName: 'Laptop', productCategory: 'Electronics' };
const productDetails: string = getProductDetails(prodObject);
console.log("22A91A05C9");
console.log(productDetails);
```

Output:

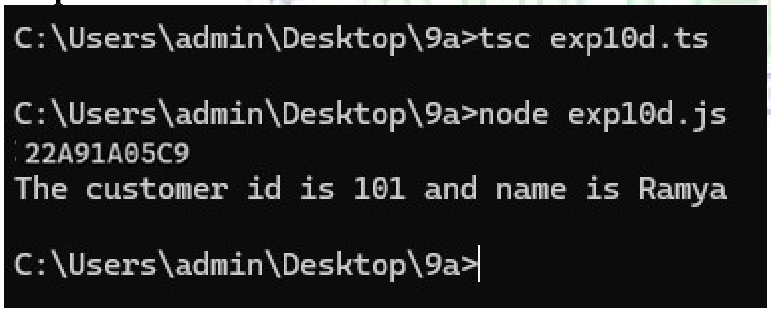


```
C:\Users\admin\Desktop\9a>tsc exp10c.ts

C:\Users\admin\Desktop\9a>node exp10c.js
22A91A05C9
The product name is : Laptop

C:\Users\admin\Desktop\9a>
```

10 d) Course Name: Typescript
Module Name: Function Types
Declare an interface with function type and access its value.
Aim: Declare an interface with function type and access its value.
Description: A function type has two parts: parameters and return type. When declaring a function type, you need to specify both parts with the following syntax:
(parameter: type, parameter:type,...) => type
Code language: PHP (php)
The following example shows how to declare a variable which has a function type that accepts two numbers and returns a number:
let add: (x: number, y: number) => number;
Code language: JavaScript (javascript)
In this example:
The function type accepts two arguments: x and y with the type number.
The type of the return value is number that follows the fat arrow (=>) appeared between parameters and return type.
Program:

```
function CreateCustomerID(name: string, id: number): string {
   return 'The customer id is ' + id + ' and name is ' + name;
}
interface StringGenerator { (chars: string, nums: number): string; }
let idGenerator: StringGenerator; idGenerator = CreateCustomerID;
const customerId: string = idGenerator('Ramya', 101);
console.log("22A91A05C9");
console.log(customerId);
```

Output:



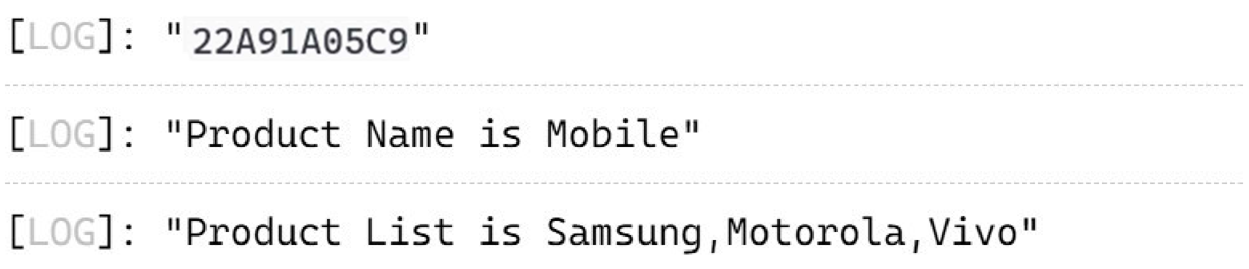
```
C:\Users\admin\Desktop\9a>tsc exp10d.ts

C:\Users\admin\Desktop\9a>node exp10d.js
22A91A05C9
The customer id is 101 and name is Ramya

C:\Users\admin\Desktop\9a>
```

Experiment-11

11 a) Course Name: Typescript
Module Name: Extending Interfaces
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.
Aim: To declare a productList interface which extends properties from two other declared interfaces like Category.
Description: An interface can be extended from an already existing one using the extends keyword. In the code below, extend the productList interface from both the Category interface and Product interface.
Example:
interface Category{ categoryName:string; }
interface Product{ productName:string; productid:number; }
interface productList extends Category,Product{ list:['Samsung','Motorola','LG'] }
Program:

```
interface category{
   categoryName:string;
}
interface Product{
   productName:string;
   productId:number;
}
interface ProductList extends Category,Product{
   list:Array;
}
const productDetails: ProductList={
   categoryNamae:'Gadget',productName: 'Mobile',
   productId: 1234, list: ['Samsung', 'Motorola', 'Vivo']
};
const listProduct = productDetails.list;
const pname: string = productDetails.productName;
console.log("22A91A05C9");
console.log('Product Name is ' + pname);
console.log('Product List is ' + listProduct);
}
```

Output:



```
[LOG]: "22A91A05C9"

[LOG]: "Product Name is Mobile"

[LOG]: "Product List is Samsung,Motorola,Vivo"
```

11 b) Course Name: Typescript
Module Name: Classes
Consider the Mobile Cart application, Create objects of the Product class and place them into the productlist array.

Aim: To consider the Mobile Cart application, Create objects of the Product class and place them into the productlist array.

Description: TypeScript is object oriented JavaScript. TypeScript supports object-oriented programming features like classes, interfaces, etc. A class in terms of OOP is a blueprint for creating objects. A class encapsulates data for the object. Typescript gives built in support for this concept called class. JavaScript ES5 or earlier didn't support classes. Typescript gets this feature from ES6. Use the class keyword to declare a class in TypeScript.

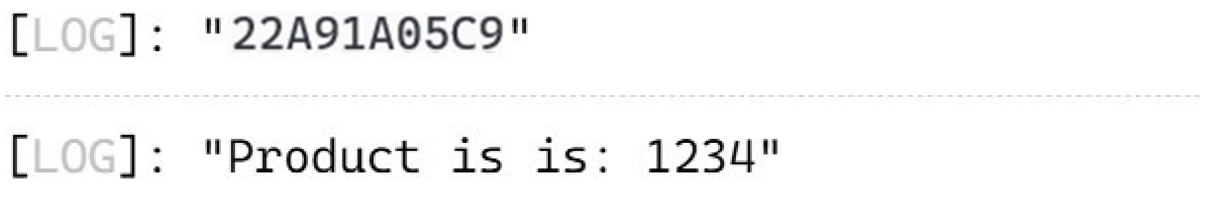
The syntax for the same is given below –

```
class class_name
{
 //class scope
}
```

Program:

```
class Product{
   static productPrice:string;
   productId:number;
   constructor(){
      this.productId = 1234;
   }
   getProductId():string{
      return 'Product is is: '+ this.productId;
   }

const product:Product=new Product();
const p={
   producti:product.getProductId(),
};
console.log("22A91A05C9");
console.log(p.producti);
```

Output:



```
[LOG]: "22A91A05C9"
[LOG]: "Product is is: 1234"
```

11 c) Course Name: Typescript
Module Name: Constructor
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>>".

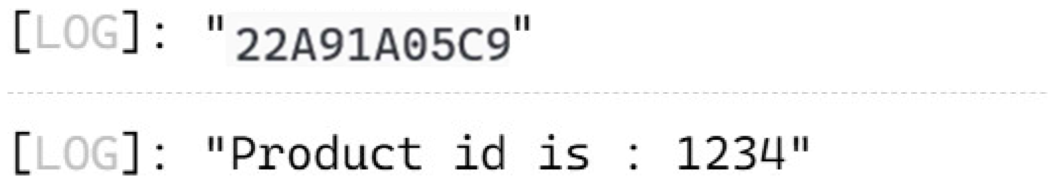
Aim: To declare a class named - Product with the below-mentioned declarations:
(i)productId as number property (ii) Constructor to initialize this value (iii) getProductId method to return the message "Product id is <>" .

Description: A constructor is a special function of the class that is automatically invoked when we create an instance of the class in Typescript. We use it to initialize the properties of the current instance of the class. Using the constructor parameter properties or Parameter shorthand syntax, we can add new properties to the class. We can also create multiple constructors using the technique of constructor method overload. The constructor method in a class must have the name constructor. A class can have only one implementation of the constructor method. The constructor method is invoked every time we create an instance from the class using the new operator. It always returns the newly created object.

Program:

```
class Product {
   static productPrice: string;
   productId: number;
   constructor(productId: number) {
      this.productId = productId;
   }
   getProductId(): string {
      return 'Product id is : ' + this.productId;
   }
}
const product: Product = new Product(1234);
console.log("22A91A05C9");
console.log(product.getProductId());
```

Output:



```
[LOG]: "22A91A05C9"

[LOG]: "Product id is : 1234"
```

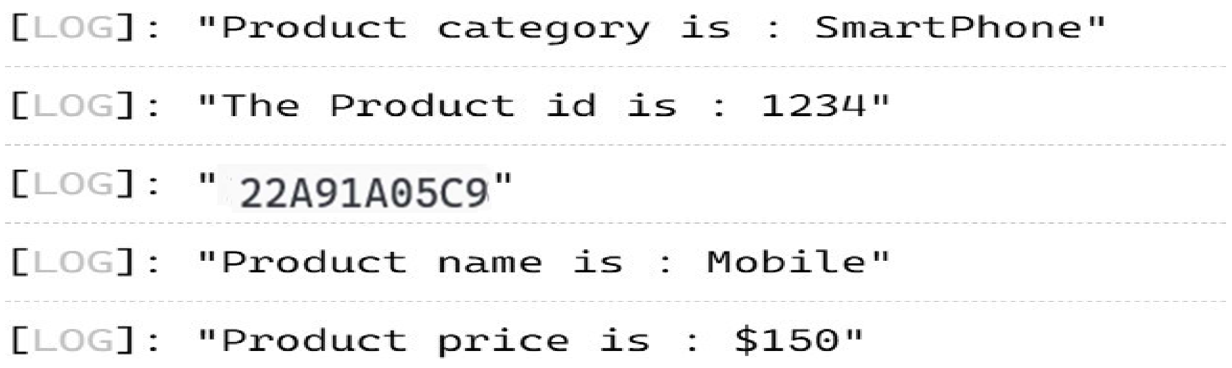
11 d) Course Name: Typescript
Module Name: Access Modifiers
Create a Product class with 4 properties namely productId, productName, productPrice, productCategory with private, public, static, and protected access modifiers and accessing them through Gadget class and its methods.

Aim: To 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: Like other programming languages, Typescript allows us to use access modifiers at the class level. It gives direct access control to the class member. These class members are functions and properties. We can use class members inside its own class, anywhere outside the class, or within its child or derived class.The access modifier increases the security of the class members and prevents them from invalid use. We can also use it to control the visibility of data members of a class. If the class does not have to be set any access modifier, TypeScript automatically sets public access modifier to all class members.

Program:
```
class Product {
   static productPrice = 150;
   private productId: number;
   public productName: string;
   protected productCategory: string;
   constructor(productId: number, productName: string, productCategory: string) {
      this.productId = productId;
      this.productName = productName;
      this.productCategory = productCategory;
   }
   getProductId() {
      console.log('The Product id is : ' + this.productId);
   }
}
class Gadget extends Product {
   getProduct(): void {
      console.log('Product category is : ' + this.productCategory);
   }
}
const g: Gadget = new Gadget(1234, 'Mobile', 'SmartPhone');
g.getProduct();
g.getProductId();
console.log("22A91A05C9");
console.log('Product name is : ' + g.productName);
console.log('Product price is : $' + Product.productPrice);
```

Output:



```
[LOG]: "Product category is : SmartPhone"
[LOG]: "The Product id is : 1234"
[LOG]: " 22A91A05C9"
[LOG]: "Product name is : Mobile"
[LOG]: "Product price is : $150"
```

Experiment-12

12 a) Course Name: Typescript
Module Name: Properties and Methods
Create a Product class with 4 properties namely productId and methods to setProductId() and getProductId().

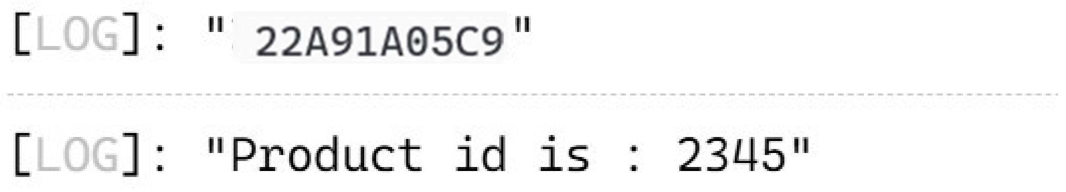
Aim: To create a Product class with 4 properties namely productId and methods to setProductId() and getProductId().

Description: In typescript, the method is a piece of code that has been declared within the class and it can be carried out when it is called. Method property in it can split a huge task into little sections and then execute the particular operation of that program so that code can be reusable which can improve the module from the program

Program:

```
class Product {
   static productPrice: string;
   productId: number;
   constructor(productId: number) {
      this.productId = productId;
   }
   getProductId(): string {
      return 'Product id is : ' + this.productId;
   }
}
const product: Product = new Product(2345);
console.log("22A91A05C9");
console.log(product.getProductId());
```

Output:



```
[LOG]: " 22A91A05C9"
[LOG]: "Product id is : 2345"
```

12 b) Course Name: Typescript
Module Name: Creating and using Namespaces
Create a namespace called ProductUtility and place the Product class definition in it. Import the Product class inside productlist file and use it.

Aim: To 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, the method is a piece of code that has been declared within the class and it can be carried out when it is called. Method property in it can split a huge task into little sections and then execute the particular operation of that program so that code can be reusable which can improve the module from the program. The classes or interfaces which should be accessed outside the namespace should be marked with keyword export. To access the class or interface in another namespace, the syntax will be

namespaceName.className

Program:

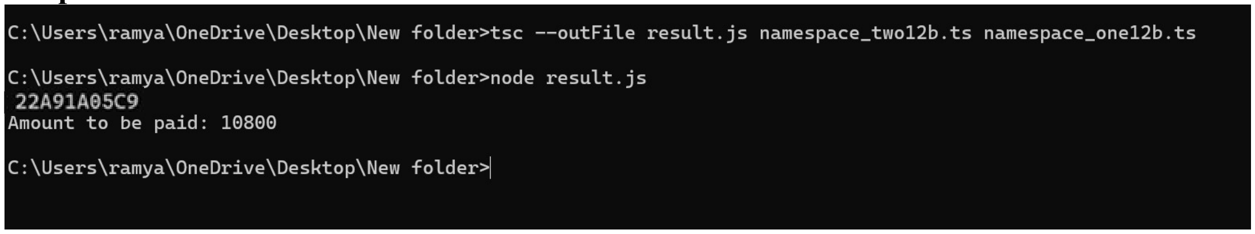
namespace_one12b.ts:

```
import util = Utility.Payment;
let paymentAmount = util.CalculateAmount(1800, 6);
console.log("22A91A05C9");
console.log(`Amount to be paid: ${paymentAmount}`);
```

namespace_two12b.ts:

```
namespace Utility {
    export namespace Payment {
        export function CalculateAmount(price: number, quantity: number): number {
            return price * quantity;
        }
    }
}
```

Output:



```
C:\Users\ramya\OneDrive\Desktop\New folder>tsc --outFile result.js namespace_two12b.ts namespace_one12b.ts

C:\Users\ramya\OneDrive\Desktop\New folder>node result.js
 22A91A05C9
Amount to be paid: 10800

C:\Users\ramya\OneDrive\Desktop\New folder>
```

12 c) Course Name: Typescript
Module Name: Creating and using Modules
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.
Aim: To creating and using Modules Consider the Mobile Cart application.
Description: A module refers to a set of standardized parts or independent units that can be used to construct a more complex structure. TypeScript modules provides a way to organize the code for better reuse.

```
export interface InterfaceName {
        //Block of statements
}
```

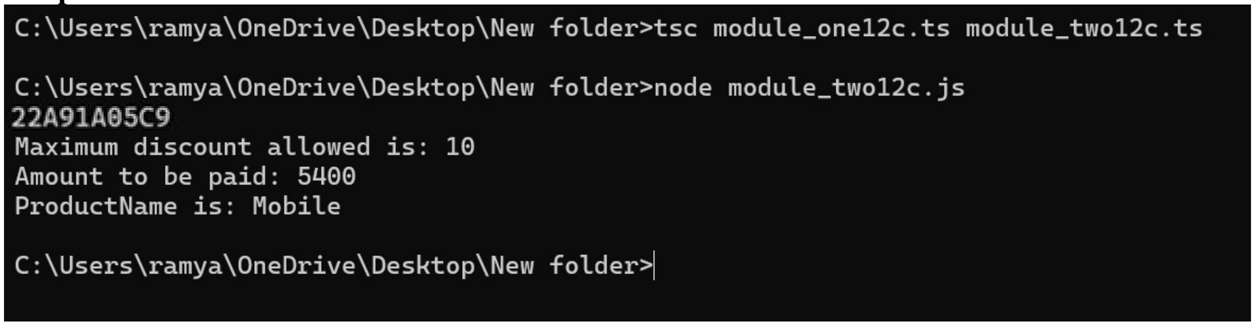
Program:
Module_one12c.ts:

```
export function MaxDiscountAllowed(noOfProduct: number): number {
   if (noOfProduct > 5) { return 30; }
   else { return 10; }
}
class Utility {
   CalculateAmount(price: number, quantity: number): number {
      return price * quantity;
   }
}
interface Category { getCategory(productId: number): string; }
export const productName = 'Mobile';
export { Utility, Category };
```

Module_two12c.ts:

```
import {Utility as mainUtility, Category, productName, MaxDiscountAllowed }
from "./module_one12c";
const util = new mainUtility();
const price = util.CalculateAmount(1350, 4);
const discount = MaxDiscountAllowed(2);
console.log("22A91A05C9");
console.log(`Maximum discount allowed is: ${discount}`);
console.log(`Amount to be paid: ${price}`);
console.log(`ProductName is:  ${productName}`);
```

Output:

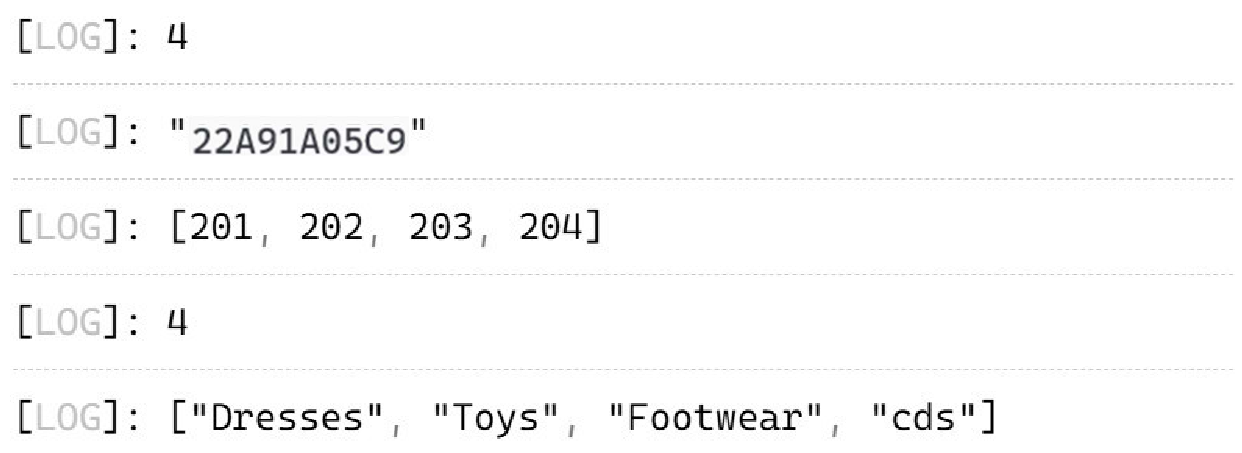


```
C:\Users\ramya\OneDrive\Desktop\New folder>tsc module_one12c.ts module_two12c.ts

C:\Users\ramya\OneDrive\Desktop\New folder>node module_two12c.js
22A91A05C9
Maximum discount allowed is: 10
Amount to be paid: 5400
ProductName is: Mobile

C:\Users\ramya\OneDrive\Desktop\New folder>
```

12 d) Course Name: Typescript
Module Name: What is Generics, What are Type Parameters, Generic Functions, Generic Constraints
Create a generic array and function to sort numbers as well as string values.
Aim: To create a generic array and function to sort numbers as well as string values.
Description: Whenever any program or code is written or executed, one major thing one always takes care of which is nothing but making reusable components which further ensures the scalability and flexibility of the program or the code for a long time. Generics, thus here comes into the picture as it provides a user to flexibly write the code of any particular data type (or return type) and that the time of calling that user could pass on the data type or the return type specifically. Generics provides a way to make the components work with any of the data types (or return types) at the time of calling it for a certain number of parameters (or arguments). In generics, we pass a parameter called type parameter which is put in between the lesser sign (<),and the greater sign (>) for example, it should be like <type_parameter_name>.
Program:

```
function orderDetails<T>(arg: Array<T>): Array<T> {
   console.log(arg.length);
   return arg;
}
const orderid: Array = [201, 202, 203, 204];
const ordername: Array = ['Dresses', 'Toys', 'Footwear', 'cds'];
const idList = orderDetails(orderid);
console.log("22A91A05C9");
console.log(idList);
const nameList = orderDetails(ordername);
console.log(nameList);
```

Output:



```
[LOG]: 4
[LOG]: "22A91A05C9"
[LOG]: [201, 202, 203, 204]
[LOG]: 4
[LOG]: ["Dresses", "Toys", "Footwear", "cds"]
```