Jafar Beldar

Weekly Assignment - 01

1. What is a closure in JavaScript? Provide an example.

A closure in JavaScript occurs when an inner function has access to variables from an outer function that has already returned. This allows the function to maintain and access its lexical scope, even after the outer function has finished execution. Closures are useful for data encapsulation, creating private variables, and function factories. They are made possible due to JavaScript's lexical scoping rules. Example:

```
35 function outer() {
36     let count = 0;
37     return function inner() {
38       count++;
39       console.log(count);
40     }
41   }
```

2. Explain the concept of lexical scope.

Lexical scope means that variable accessibility is determined by the physical structure of the code. Functions are executed using the scope that was defined at the time of function declaration, not invocation. This leads to predictable behavior where inner functions can access variables of their outer functions. Lexical scope enables closures, which are powerful in asynchronous programming and data encapsulation. It helps in understanding how JavaScript resolves variable references.

3. What is the difference between == and ===?

The `==` operator compares two values after performing type coercion, meaning it converts both operands to a common type before comparison. On the other hand, `===` compares both the value and the type, so it does not perform any conversion. `===` is also known as the strict equality operator. For example, `5 == '5'` is true, but `5 === '5'` is false. Using `===` is generally safer to avoid unexpected bugs from implicit type conversion.

4. How does this behave inside arrow functions vs regular functions?

In regular functions, the value of `this` depends on how the function is called, which can lead to unexpected behavior. Arrow functions do not have their own `this` binding; instead, they inherit `this` from the surrounding lexical context. This makes arrow functions useful in callbacks or when using `this` from the parent scope. In contrast, regular functions are more flexible when you want to dynamically assign `this`. Understanding the difference is crucial in object-oriented JavaScript.

5. What is the event loop, and how does it work?

The event loop is a mechanism in JavaScript that handles asynchronous operations in a non-blocking way. It allows JavaScript to perform non-blocking I/O by offloading operations and

processing them in the background. The loop continuously checks the call stack and task queue. If the call stack is empty, it pushes the next task from the queue to the stack. This helps in managing timers, promises, and user interactions efficiently.

6. What is a Promise in JavaScript? How is it different from a callback?

A Promise represents a future value that will eventually be resolved or rejected. It provides cleaner syntax and better error handling than traditional callbacks. Callbacks can lead to nested and hard-to-read code (callback hell), while Promises allow chaining using `.then()` and `.catch()`. Promises improve readability and structure of asynchronous code. They can be combined with `async/await` for even more clarity.

7. What are async/await in JavaScript? Provide an example.

`async` and `await` are modern JavaScript features used to handle asynchronous operations in a cleaner way. An `async` function always returns a Promise, and `await` pauses the execution until the Promise resolves. This makes asynchronous code look synchronous, improving readability and reducing nested structures. You can catch errors using try-catch blocks. Example:

```
async function fetchData() {
  const res = await fetch('url');
  const data = await res.json();
  console.log(data);
}
```

8. What are JavaScript modules and how do you use export and import?

JavaScript modules allow you to break your code into reusable files. You can export functions, variables, or classes from a module using `export` and include them in other files using `import`. This promotes modularity and helps manage large codebases. Modules run in strict mode by default. Example:

```
// utils.js
export function add(a, b) { return a + b; }
// main.js
import { add } from './utils.js';
```

9. What is object destructuring and how is it useful?

Object destructuring is a syntax that allows you to extract properties from objects and assign them to variables. It helps write cleaner and more readable code, especially when dealing with objects with many properties. It also supports default values and nested destructuring. It is commonly used in function parameters. Example:

```
const {name, age} = person;
```

10. Explain the methods map(), filter(), and reduce() with examples.

`map()` creates a new array by applying a function to each element of an array. `filter()` returns a new array with elements that satisfy a condition. `reduce()` applies a function to accumulate values into a single result. These are functional programming methods that simplify array operations. Examples:

```
[1,2,3].map(x => x*2);
[1,2,3].filter(x => x > 1);
[1,2,3].reduce((a,b) => a+b);
```

11. What are template literals? How are they different from regular strings?

Template literals are enclosed in backticks (`) and allow embedded expressions using ${}. They support multi-line strings and expression interpolation. This makes them more flexible than regular strings enclosed in single or double quotes. They improve readability, especially when dealing with dynamic values. Example: `Hello, ${name}!`

12. How is prototypal inheritance implemented in JavaScript?

Prototypal inheritance allows objects to inherit properties from other objects. Every JavaScript object has a prototype property that points to another object. You can use `Object.create()` to create an object with a specific prototype. Inheritance happens via the prototype chain. It is more flexible compared to classical inheritance.

13. What are the default parameters in functions?

Default parameters allow you to set default values for function parameters. If a value is not provided or is undefined during function call, the default will be used. This simplifies code by avoiding the need for manual checks. It improves function robustness and clarity. Example: function greet(name = 'Guest') { console.log('Hello ' + name); }

14. What is the difference between null and undefined?

`null` is an assignment value that represents no value or empty value. `undefined` means a variable has been declared but not assigned any value. Both represent absence of value, but `null` is intentional, while `undefined` is default. Typeof null returns 'object', and typeof undefined returns 'undefined'. They are not the same and should not be used interchangeably.

15. What is hoisting in JavaScript?

Hoisting is JavaScript's default behavior of moving declarations to the top of the current scope. Only declarations are hoisted, not initializations. This applies to variables (with var) and function declarations. Let and const are also hoisted but stay in the Temporal Dead Zone (TDZ). Understanding hoisting helps avoid runtime errors and bugs.

16. What is Node.js and what makes it different from browser JavaScript?

Node.js is a runtime environment that allows JavaScript to run outside the browser. It is built on the V8 engine and is used for building server-side applications. Unlike browser JavaScript, Node.js has access to file systems, networking, and operating system APIs. It is non-blocking and event-driven, which makes it suitable for scalable applications. Browser JavaScript is limited to the client-side environment.

17. What is the use of require() in Node.js?

`require()` is used to import modules in Node.js. It allows code to be split into reusable files. Built-in, third-party, and custom modules can be loaded using require. This promotes modular coding and separation of concerns. Example: const fs = require('fs');

18. How do you read a file using Node.js?

You can read a file in Node.js using the `fs` module. Both synchronous and asynchronous methods are available. The asynchronous version prevents blocking of the event loop. Example: fs.readFile('file.txt', 'utf8', (err, data) => { console.log(data); }); Always handle errors to avoid application crashes.

19. What is the role of the package.json file?

`package.json` holds metadata about a Node.js project. It defines dependencies, scripts, version, author, and more. NPM uses this file to install packages and manage project configurations. It ensures consistent builds across environments. It is essential for project setup and maintenance.

20. What are global objects in Node.js?

Global objects in Node.js are available in all modules without importing them. Examples include `__dirname`, `__filename`, `setTimeout`, and `global`. They help with system-level tasks and utility operations. They differ from browser globals like `window`. Using them responsibly is important for clean code.

21. Explain the process object in Node.js.

The `process` object provides information and control over the current Node.js process. It can be used to access environment variables, read command-line arguments, and manage process events. Methods like `process.exit()` and events like `uncaughtException` are commonly used. It is part of Node's global scope and does not require a `require` call. It is essential for managing runtime behavior.

22. How do you handle exceptions in Node.js applications?

Exceptions in Node.js can be handled using try-catch blocks and event listeners. For asynchronous operations, callbacks should include error parameters. Using `.catch()` with Promises and `async/await` with try-catch is also common. You can listen to `process.on('uncaughtException')` to handle unhandled exceptions. Proper error handling ensures stability and prevents crashes.

23. How do you create a basic HTTP server using Node.js?

You can create an HTTP server using Node.js by requiring the `http` module. Use `http.createServer()` to define request and response logic. Call `.listen()` to assign a port for the server to listen on. This sets up a basic web server that handles incoming requests.
Example:
const http = require('http');
http.createServer((req, res) => { res.end('Hello'); }).listen(3000);

24. What are the different HTTP methods supported by Node.js HTTP module?

Node.js HTTP module supports standard HTTP methods like GET, POST, PUT, DELETE, PATCH, and OPTIONS. Each method serves a specific purpose in RESTful APIs. we can identify the method using `req.method` in the server handler. Handling them properly ensures API functionality and correctness. These methods follow the HTTP protocol standards.

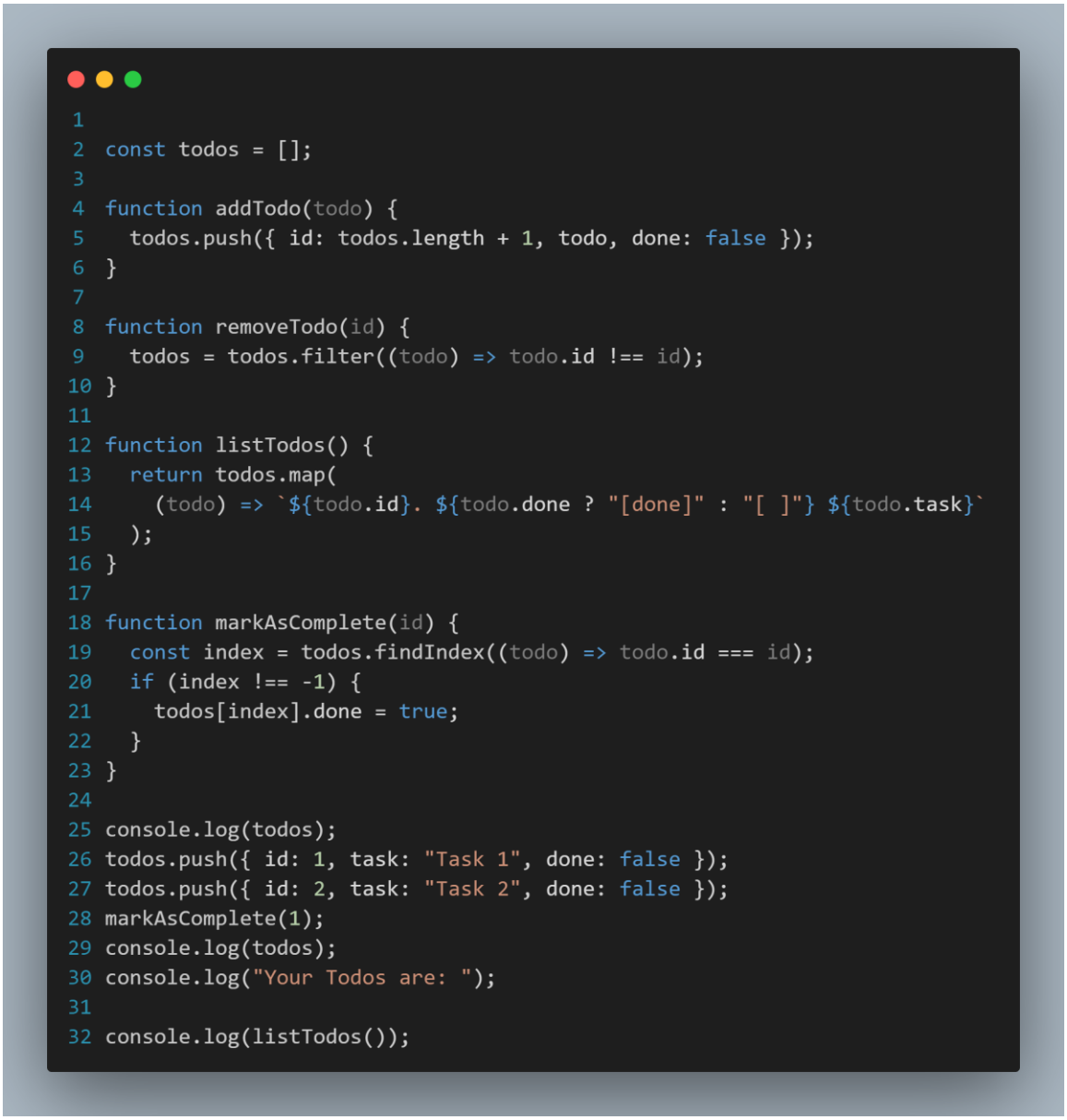
25. How can you parse query parameters in a URL using Node.js?

You can parse query parameters using the `url` and `querystring` modules. The `url.parse()` function breaks the URL into components. Then `querystring.parse()` can convert the query string into an object. Alternatively, `URLSearchParams` can be used in modern Node versions. Example: const params = new URL(req.url, 'http://localhost').searchParams;

Practical Tasks

Github Link

Todos Application:



```
const todos = [];

function addTodo(todo) {
  todos.push({ id: todos.length + 1, todo, done: false });
}

function removeTodo(id) {
  todos = todos.filter((todo) => todo.id !== id);
}

function listTodos() {
  return todos.map(
    (todo) => `${todo.id}. ${todo.done ? "[done]" : "[ ]"} ${todo.task}`
  );
}

function markAsComplete(id) {
  const index = todos.findIndex((todo) => todo.id === id);
  if (index !== -1) {
    todos[index].done = true;
  }
}

console.log(todos);
todos.push({ id: 1, task: "Task 1", done: false });
todos.push({ id: 2, task: "Task 2", done: false });
markAsComplete(1);
console.log(todos);
console.log("Your Todos are: ");

console.log(listTodos());
```

```
30  console.log(todos);
31  todos.push({ id: 1, task: 'Task 1', done: false });
32  todos.push({ id: 2, task: 'Task 2', done: false });
33  markAsComplete(1);
34  console.log(todos);
35  console.log("Your Todos are: ")
36
37  console.log(listTodos());
38
```

PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

```
[nodemon] clean exit - waiting for changes before restart
[nodemon] restarting due to changes...
[nodemon] starting `node .\weekly_Assignment-1.js`
[]
[
  { id: 1, task: 'Task 1', done: true },
  { id: 2, task: 'Task 2', done: false }
]
Your Todos are:
[ '1. [done] Task 1', '2. [ ] Task 2' ]
[nodemon] clean exit - waiting for changes before restart
```

Jafar Beldar

Form Validator:

```
1
2  const data = {
3      name: "John  Doe",
4      email: "abcd@@gmail.com",
5      phone: "1234567890",
6      age: 51,
7      password: "password123",
8  };
9
10 export function validateForm(formData) {
11     const regularExpression=/^[^\s@]+@[^\s@]+\.[^\s@]+$/;  //for email validation
12
13     switch (true) {
14         case formData === undefined || formData === null:
15             return "Form data is required";
16
17         case !formData.name || formData.name === "":
18             return "Name is required";
19
20         case formData.name.length < 3:
21             return "Name must be at least 3 characters long";
22
23         case !formData.email || formData.email === "":
24             return "Email is required";
25
26         case !regularExpression.test(formData.email):
27             return "Email is invalid";
28
29         case !formData.phone || formData.phone === "":
30             return "Phone number is required";
31
32         case formData.phone.length !== 10:
33             return "Phone number must be exactly 10 digits long";
34
35         case isNaN(formData.phone):
36             return "Phone number must be a number";
37
38         case !formData.age || formData.age === "":
39             return "Age is required";
40
41         case isNaN(formData.age):
42             return "Age must be a number";
43
44         case formData.age < 18:
45             return "Age must be at least 18 years old";
46
47         case !formData.password || formData.password === "":
48             return "Password is required";
49
50         case formData.password.length < 6:
51             return "Password should be more than 6 characters";
52
53         default:
54             return "Form is valid";
55     }
56 }
57
58 console.log(validateForm(data));
59
60
```

```
20  const data = {
21      name: "John  Doe",
22      email: "abcd@@gmail.com",
23      phone: "1234567890",
24      age: 51,
25      password: "password123",
26  };
27
28  export function validateForm(formData) {
```

PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

```
[nodemon] restarting due to changes...
[nodemon] starting `node .\FormValidator.js`
Email is invalid
[nodemon] clean exit - waiting for changes before restart
```

Shopping Cart:

```
1
2  const cart = [
3    { id: 1, name: "Product 1", price: 100, quantity: 2 }, //200
4    { id: 2, name: "Product 2", price: 100, quantity: 3 }, //300
5    { id: 3, name: "Product 3", price: 100, quantity: 5 }, //500  total==1000
6  ];
7
8  function calculateTotal(cart) {
9    let total = 0;
10   cart.forEach((product) => {
11     total += product.price * product.quantity;
12   });
13   console.log(`Your Total ${total} with discount of 10% is: `);
14
15   if (total > 100) {
16     return (total *= 0.9);
17   }
18
19   console.log("Your Total is: ");
20   return total;
21 }
22
23 console.log(calculateTotal(cart));
24
```

```
const cart = [
  { id: 1, name: "Product 1", price: 100, quantity: 2 }, //200
  { id: 2, name: "Product 2", price: 100, quantity: 3 }, //300
  { id: 3, name: "Product 3", price: 100, quantity: 5 }, //500  total==1000
];

function calculateTotal(cart) {
  let total = 0;
  cart.forEach((product) => {
    total += product.price * product.quantity;
  });
  console.log(`Your Total ${total} with discount of 10% is: `);

  if (total > 100) {
```

(property) price: number

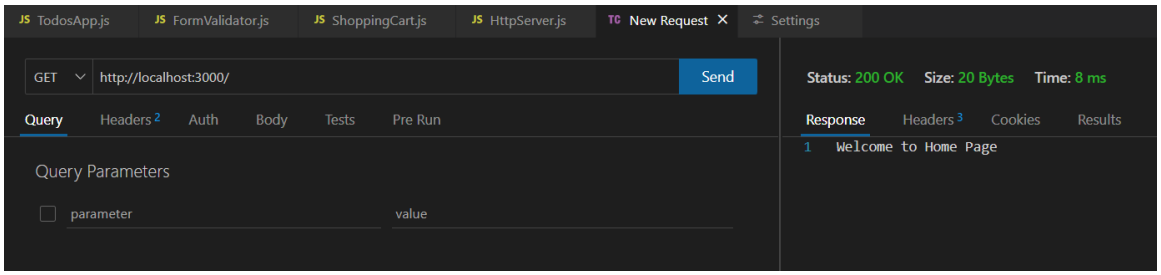
```
PROBLEMS  OUTPUT  TERMINAL  PORTS  DEBUG CONSOLE        node - Weekly_

[nodemon] starting `node .\ShoppingCart.js`
Your Total 1000 with discount of 10% is:
900
[nodemon] clean exit - waiting for changes before restart
```

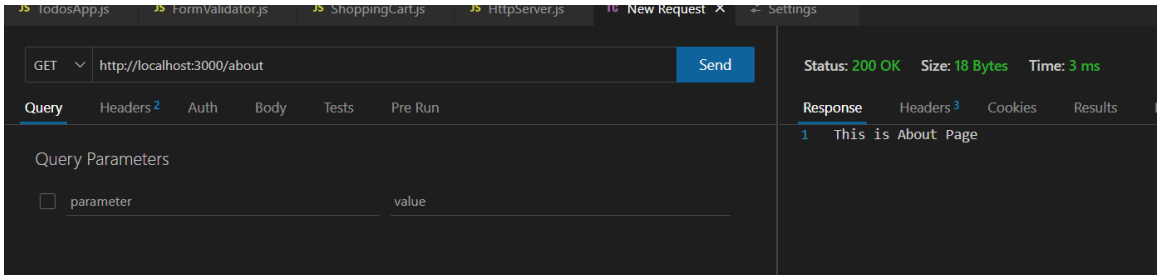
HTTP-Server:

```
1
2
3  import http from 'http';
4
5  const server = http.createServer((req, res) => {
6    if (req.url === '/') {
7      res.end('Welcome to Home Page');
8    } else if (req.url === '/about') {
9      res.end('This is About Page');
10   } else {
11     res.end('404 Not Found');
12   }
13 });
14
15 server.listen(3000, () => {
16   console.log('Server running on http://localhost:3000');
17 });
18
```

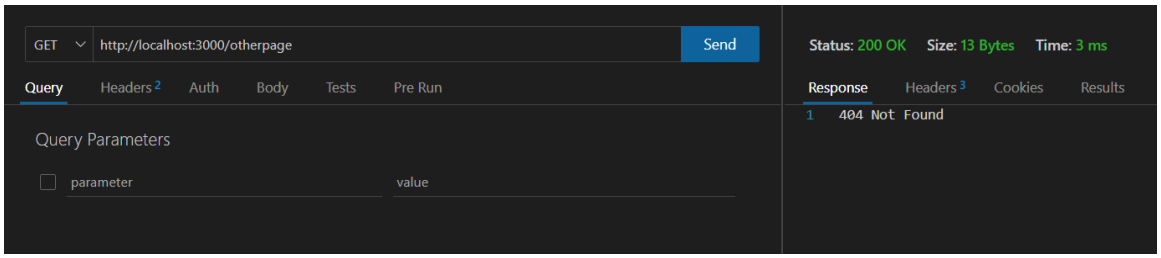
“/” page:



“/about” page:



“404 not found”:



Promise:

```
1
2  function fetchData() {
3    const rejectrequest = Math.random() < 0.3;
4
5    return new Promise((resolve, reject) => {
6  setTimeout(()=>{
7      if (rejectrequest) {
8          reject("Api call rejected!");
9      } else {
10         resolve({
11             name: "Jafar",
12             email: "beldarjafar@gmail.com",
13             password: "root123jafar",
14         });
15     }
16 },2000)
17 });
18 }
19
20 async function fetchUser() {
21
22     try{
23         const response = await fetchData();
24         console.log(response);
25     }catch(err){console.log(err);
26     }
27 }
28 fetchUser();
```

```
19  async function fetchUser() {
20
21      try{
22          const response = await fetchData();
23          console.log(response);
24      }catch(err){console.log(err);
25      }
26  }
27  fetchUser();
28
```

PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

```
}
[nodemon] clean exit - waiting for changes before restart
[nodemon] restarting due to changes...
[nodemon] starting `node .\Promise.js`
{
  name: 'Jafar',
  email: 'beldarjafar@gmail.com',
  password: 'root123jafar'
}
[nodemon] clean exit - waiting for changes before restart
[nodemon] restarting due to changes...
[nodemon] starting `node .\Promise.js`
Api call rejected!
[nodemon] clean exit - waiting for changes before restart
```

Jafar Beldar

Social media simulation:

```
class User {
    constructor(id, username) {
      this.id = id;
      this.username = username;
    }
  }

  class Comment {
    constructor(id, commenter, text) {
      this.id = id;
      this.commenter = commenter;
      this.text = text;
    }
  }

  class Post {
    constructor(id, content, author) {
      this.id = id;
      this.content = content;
      this.author = author;
      this.likes = [];
      this.comments = [];
      this.createdAt = new Date();
    }

    addLike(user) {
      if (user.id !== this.author.id && !this.likes.includes(user)) {
        this.likes.push(user);
      }
    }

    addComment(comment) {
      if (comment.commenter.id !== this.author.id) {
        this.comments.push(comment);
      }
    }
  }

  // Sample data
  const usernames = ['raj', 'simran', 'arjun', 'neha', 'ravi', 'priya', 'aman', 'anita', 'vijay', 'kiran'];
  const postsText = ['Hello world!', 'Great day!', 'Love coding.', 'JS is fun!', 'Just chilling.', 'Weekend vibes.', 'College life rocks!', 'Feeling motivated.', 'New post!', 'Keep learning.'];
  const commentsText = ['Nice!', 'Awesome.', 'Cool post.', 'Agreed!', 'Well said.', 'True that!', 'Haha ', 'Interesting.', 'Good one!', '👌'];

  let users = [];
  let posts = [];
  let commentId = 1;

  // creating users
  usernames.forEach((name, i) => {
    users.push(new User(i + 1, name));
  });

  // creating posts
  users.forEach((user, i) => {
    const post = new Post(i + 1, postsText[i], user);
    posts.push(post);
  });

 // Adding likes and comments
posts.forEach(post => {
    // Add 3 likes from random users
    for (let i = 0; i < 3; i++) {
      const randomUser = users[Math.floor(Math.random() * users.length)];
      post.addLike(randomUser);
    }

    // Add 2 comments from random users
    for (let i = 0; i < 2; i++) {
      const randomUser = users[Math.floor(Math.random() * users.length)];
      const randomComment = commentsText[Math.floor(Math.random() * commentsText.length)];
      const comment = new Comment(commentId++, randomUser, randomComment);
      post.addComment(comment);
    }
  });

  // Display output
  users.forEach(user => {
    console.log(`\nUser: ${user.username}`);
    posts
      .filter(p => p.author.id === user.id)
      .forEach(post => {
        console.log(`  Post: "${post.content}"`);
        console.log(`  Created At: ${post.createdAt}`);
        console.log(`  Likes: ${post.likes.map(u => u.username).join(', ')}`);
        console.log(`  Comments:`);
        post.comments.forEach(c => {
          console.log(`    - ${c.commenter.username}: ${c.text}`);
        });
      });
  });
```

```
[nodemon] starting `node .\SocialNetworkSimulation.js`

User: raj
  Post: "Hello world!"
  Created At: Sun Apr 06 2025 15:37:24 GMT+0530 (India Standard Time)
  Likes: simran, neha, vijay
  Comments:
    - vijay: True that!
    - kiran: 👌

User: simran
  Post: "Great day!"
  Created At: Sun Apr 06 2025 15:37:24 GMT+0530 (India Standard Time)
  Likes: raj, aman, arjun
  Comments:
    - priya: Awesome.
    - arjun: Awesome.

User: arjun
  Post: "Love coding."
  Created At: Sun Apr 06 2025 15:37:24 GMT+0530 (India Standard Time)
  Likes: neha, simran, aman
  Comments:
    - ravi: Agreed!
    - raj: Cool post.

User: neha
  Post: "JS is fun!"
  Created At: Sun Apr 06 2025 15:37:24 GMT+0530 (India Standard Time)
  Likes: raj
  Comments:
    - priya: Haha
    - arjun: Well said.
```

Jafar Beldar

School Management simulation:

```
1
2  class Student {
3      constructor(id, name, age, classId) {
4        this.id = id;
5        this.name = name;
6        this.age = age;
7        this.classId = classId;
8        this.grades = []; // stores { subject, score }
9      }
10
11     addGrade(subject, score) {
12       this.grades.push({ subject, score });
13     }
14
15     getAverageGrade() {
16       const total = this.grades.reduce((sum, g) => sum + g.score, 0); // total of all subject scores
17       return this.grades.length ? total / this.grades.length :0     // average = total / number of grades
18     }
19   }
20
21   class Teacher {
22     constructor(id, name, subject) {
23       this.id = id;
24       this.name = name;
25       this.subject = subject;
26       this.classIds = []; // classes assigned to teacher
27     }
28
29     // Assign a class to this teacher
30     assignClassToTeacher(classId) {
31       this.classIds.push(classId);
32     }
33   }
34
35   class Classroom {
36     constructor(id, name) {
37       this.id = id;
38       this.name = name;
39       this.students = [];
40     }
41
42     // Add a student to this class
43     addStudent(student) {
44       this.students.push(student);
45     }
46
47     // Calculate average grade of all students in this class
48     getAverageClassGrade() {
49       const total = this.students.reduce((sum, s) => sum + s.getAverageGrade(), 0); // toal of all students average grades
50       return this.students.length ? total / this.students.length : 0;              // class average = total / student count
51     }
52   }
```

```
//sample teacher records
  const teachers = [
    new Teacher(1, "deepak iyer", "math"),
    new Teacher(2, "anita khanna", "science"),
    new Teacher(3, "rohit chavan", "english"),
    new Teacher(4, "sneha rao", "social studies"),
    new Teacher(5, "vikas sharma", "computer science"),
  ];

  //  10 classes
  const classes = [];
  for (let i = 0; i < 10; i++) {
    classes.push(new Classroom(i + 1, `class-${i + 1}`)); // class id = i+1, name = class-1 to class-10
  }

  // Creating 30 students
  const students = [];
  let id = 1;

  for (let i = 0; i < 30; i++) {
    const classId = (i % 10) + 1; // Distributed students into 10 classes (1 to 10)
    const s = new Student(id, `student-${id}`, 13 + (i % 6), classId); // Age is random between 13 to 18

    // Add grades (random between 60 to 100)
    s.addGrade("math", Math.floor(Math.random() * 41) + 60);    // random between 60-100
    s.addGrade("science", Math.floor(Math.random() * 41) + 60);
    s.addGrade("english", Math.floor(Math.random() * 41) + 60);
    students.push(s);
    classes[classId - 1].addStudent(s); // Add student to the  classroom
    id++;
  }

  // Assign each teacher to a class
  teachers.forEach((teacher, index) => {
    const classId = index + 1; // teacher 0 → class 1, teacher 1 → class 2....
    teacher.assignClassToTeacher(classId);
  });


  // Top 5 students by average grade
  const top5 = [...students]
    .sort((a, b) => b.getAverageGrade() - a.getAverageGrade()) // Sorting descending by average grade
    .slice(0, 5); // Getting first 5 students

  console.log(" Top 5 Students:");
  top5.forEach(s => {
    console.log(`${s.name} - Avg Grade: ${s.getAverageGrade().toFixed(2)}`);
  });

  // Find the teacher with the highest student average
  let bestTeacher = null;
  let highestAvg = 0;

  teachers.forEach(t => {
    // finding students assigned to the classes this teacher handles
    const relevantStudents = students.filter(s => t.classIds.includes(s.classId));

    // calculating average of these students
    const avg = relevantStudents.reduce((sum, s) => sum + s.getAverageGrade(), 0) /
                (relevantStudents.length || 1); // avoid division by 0

    // checking if this average is the highest so far
    if (avg > highestAvg) {
      highestAvg = avg;
      bestTeacher = t;
    }
  });

  console.log(`\n Teacher with Highest Student Avg: ${bestTeacher.name} (${highestAvg.toFixed(2)})`);

  // Print students per class with their performance
  console.log("\n Class-wise Student Performance:");
  classes.forEach(c => {
    console.log(`\n${c.name}`);
    c.students.forEach(s => {
      console.log(`- ${s.name} | Avg: ${s.getAverageGrade().toFixed(2)}`);
    });
  });
```

```
[nodemon] restarting due to changes...
[nodemon] starting `node .\SchoolMgtSimulation.js`
 Top 5 Students:
student-26 - Avg Grade: 98.00
student-2 - Avg Grade: 92.33
student-14 - Avg Grade: 88.00
student-30 - Avg Grade: 86.67
student-25 - Avg Grade: 86.33

 Teacher with Highest Student Avg: vikas sharma (81.78)

 Class-wise Student Performance:

class-1
- student-1 | Avg: 74.33
- student-11 | Avg: 75.33
- student-21 | Avg: 82.33

class-2
- student-2 | Avg: 92.33
- student-12 | Avg: 75.00
- student-22 | Avg: 73.33

class-3
- student-3 | Avg: 75.00
- student-13 | Avg: 82.00
- student-23 | Avg: 75.00

class-4
- student-4 | Avg: 79.00
- student-14 | Avg: 88.00
- student-24 | Avg: 65.33

class-5
- student-5 | Avg: 81.00
- student-15 | Avg: 78.00
- student-25 | Avg: 86.33

class-6
- student-6 | Avg: 79.67
- student-16 | Avg: 76.00
- student-26 | Avg: 98.00
```