**Tasks:**

**1.  JS-MODULE-T-1 (1 point)**

**a. What are modules in JavaScript?**

In JavaScript, modules are a way to structure code by encapsulating related functionality into separate files or modules. Each module can have its own variables, functions, and classes, making it a self-contained unit of code. Modules help in organizing code, making it more maintainable and scalable.

In the past, JavaScript did not have built-in support for modules, but with the introduction of ECMAScript 6 (ES6), also known as ECMAScript 2015, JavaScript gained native support for modules.

**b. What kinds of benefits can modules provide?**

***Benefits of Modules in JavaScript:***

**Encapsulation:**

Modules allow you to encapsulate related functionality, keeping variables and functions local to the module. This helps avoid naming conflicts and unintended interference with other parts of the code.

**Organization and Maintainability:**

Modules provide a clean and organized way to structure code. Developers can easily locate and manage different parts of the application, making it more maintainable, especially in larger projects.

**Reuse and Composition:**

Modules promote code reuse. You can export specific functionality from a module and import it into another module or file. This promotes a modular and compositional approach to building applications.

**Dependency Management:**

Modules can declare dependencies on other modules, and modern module systems provide mechanisms for loading dependencies in a controlled manner. This helps manage the dependencies between different parts of the code.

**Improved Namespacing:**

Modules provide a way to create a private scope for variables and functions. This helps avoid polluting the global namespace, reducing the chances of naming conflicts.

**Asynchronous Module Loading:**

Some module systems, like the CommonJS and AMD (Asynchronous Module Definition) formats, support Asynchronous loading of modules. This is beneficial for optimizing performance by loading modules only when they are needed.

**Tooling Support:**

Modern development tools and bundlers, such as Webpack, Rollup, and Parcel, support modules and provide features like code splitting, dead code elimination, and tree shaking, which help optimize and bundle the code for production.

In summary, modules in JavaScript offer a more organized, maintainable, and scalable approach to structuring code, promoting code reuse and encapsulation while enhancing dependency management.

**2. JS-MODULE-T-2 (1 point)**

Explain the code lines from two different modules.

const {days, months} = require("date-names");

import {days, months} from "date-names";

The code lines demonstrate the use of modules in JavaScript, but they are written in different module systems: CommonJS and ES6 Modules.

**CommonJS Syntax:**

const {days, months} = require("date-names");

In this line, the require function is used to import the "date-names" module using the CommonJS module system.

The module exports an object with properties days and months, and the const {days, months} syntax is used to destructure these properties from the imported object.

After this line, the variables days and months will contain the values exported from the "date-names" module.  
  
**ES6 Module Syntax:**

import {days, months} from "date-names";

In this line, the import statement is used to import specific named exports (days and months) from the "date-names" module using the ES6 Modules syntax.

This line is equivalent to the CommonJS example but uses the newer ECMAScript 6 module system.

After this line, the variables days and months will contain the values exported from the "date-names" module.

It's important to note that the choice between CommonJS and ES6 Modules depends on the context and the environment in which the code is running. CommonJS is commonly used in Node.js environments, while ES6 Modules are the standard for modern web browsers and many front-end development tools. Mixing these module systems in the same codebase may require additional configuration and consideration of compatibility.

**3. JS-MODULE-P-ES6 (2 points)**

Create an ES6 module PriceList. The PriceList module publishes two operations: setPriceList and getPrice. The setPriceList operation accepts one argument: thePriceList. You can see the form of the argument in the example below.

[

  {

    itemid: 1,

    itemname: ‘bottle’,

    prices: [{ qty: 1, price: 1.25},{ qty: 100, price: 1.10},{ qty: 1000, price: 1.00}]

  },

  {

    itemid: 2,

    itemname: ‘glass’,

    prices: [{ qty: 1, price: 2.50},{ qty: 50, price: 2.00},{ qty: 200, price: 1.60}]

  }

]

The setPriceList operation assigns the given pricelist array into the module’s private variable priceList. Please note that the qty in the pricelist is the minimum quantity that must be bought to get the item with a related unit price.

You can use the pricelist shown above, but please certain to add at least on more item and its prices to it.

The getPrice operation returns the total price of the quantity of the item. It accepts two arguments: itemId and quantity. For example, by using the pricelist shown above the function call getPrice(2, 150) returns 300.00.

Import the ES6 module to another JavaScript file, set the pricelist, and call the getPrice operation several times with different products and quantities.

// PriceList.js

let priceList = [];

const setPriceList = (newPriceList) => {

  priceList = newPriceList;

};

const getPrice = (itemId, quantity) => {

  const item = priceList.find((item) => item.itemid === itemId);

  if (item) {

    const matchingPrice = item.prices.find((price) => quantity >= price.qty);

    if (matchingPrice) {

      return quantity \* matchingPrice.price;

    }

  }

  return 0; // Return 0 if the item or matching price is not found

};

export { setPriceList, getPrice };

// main.js

import { setPriceList, getPrice } from './PriceList';

// Example PriceList

const examplePriceList = [

  {

    itemid: 1,

    itemname: 'bottle',

    prices: [{ qty: 1, price: 1.25 }, { qty: 100, price: 1.10 }, { qty: 1000, price: 1.00 }]

  },

  {

    itemid: 2,

    itemname: 'glass',

    prices: [{ qty: 1, price: 2.50 }, { qty: 50, price: 2.00 }, { qty: 200, price: 1.60 }]

  },

  // Add at least one more item and its prices

  {

    itemid: 3,

    itemname: 'plate',

    prices: [{ qty: 1, price: 3.00 }, { qty: 75, price: 2.50 }, { qty: 150, price: 2.00 }]

  }

];

// Set the PriceList

setPriceList(examplePriceList);

// Call the getPrice operation with different products and quantities

console.log(getPrice(2, 150)); // Output: 300.00

console.log(getPrice(1, 50));  // Output: 55.00

console.log(getPrice(3, 100)); // Output: 200.00

**4. JS-MODULE-P-COMMONJS (2 points)**

// PriceList.js

let priceList = [];

const setPriceList = (newPriceList) => {

  priceList = newPriceList;

};

const getPrice = (itemId, quantity) => {

  const item = priceList.find((item) => item.itemid === itemId);

  if (item) {

    const matchingPrice = item.prices.find((price) => quantity >= price.qty);

    if (matchingPrice) {

      return quantity \* matchingPrice.price;

    }

  }

  return 0; // Return 0 if the item or matching price is not found

};

module.exports = { setPriceList, getPrice };

// main.js

const { setPriceList, getPrice } = require('./PriceList');

// Example PriceList

const examplePriceList = [

  {

    itemid: 1,

    itemname: 'bottle',

    prices: [{ qty: 1, price: 1.25 }, { qty: 100, price: 1.10 }, { qty: 1000, price: 1.00 }]

  },

  {

    itemid: 2,

    itemname: 'glass',

    prices: [{ qty: 1, price: 2.50 }, { qty: 50, price: 2.00 }, { qty: 200, price: 1.60 }]

  },

  // Add at least one more item and its prices

  {

    itemid: 3,

    itemname: 'plate',

    prices: [{ qty: 1, price: 3.00 }, { qty: 75, price: 2.50 }, { qty: 150, price: 2.00 }]

  }

];

// Set the PriceList

setPriceList(examplePriceList);

// Call the getPrice operation with different products and quantities

console.log(getPrice(2, 150)); // Output: 300.00

console.log(getPrice(1, 50));  // Output: 55.00

console.log(getPrice(3, 100)); // Output: 200.00

**5. JS-PROMISE-T-1 (2 points)**

Answer shortly for the following questions:

1. **What is the idea of using promise objects in JavaScript?**

**Idea of Using Promise Objects in JavaScript:**

The idea behind using promise objects in JavaScript is to handle Asynchronous operations more effectively and avoid callback hell (nested callbacks). Promises to provide a Cleaner and more readable way to deal with Asynchronous code.

A promise represents the eventual completion or failure of an Asynchronous operation and allows you to attach callbacks (handlers) that will be executed when the operation is completed (fulfilled) or Encounters an error (rejected).

1. **How does the promise object move to state fulfilled?**

**How Promise Object Moves to State Fulfilled:**

The promise object transitions to the fulfilled state when the Asynchronous operation it represents is successfully completed.

This happens when the function passed to the resolve parameter inside the promise constructor is called. For example:

const myPromise = new Promise((resolve, reject) => {

    // Asynchronous operation

    // When successfully completed:

    resolve(result);

  });

1. **How do you chain handler functions to a promise object so that they are run one after another?**

**Chaining Handler Functions to a Promise Object:**

You can chain handler functions to a promise using the then method. Each then call Returns a new promise, allowing you to chain multiple Asynchronous operations in a sequential manner.

Example:

myPromise

  .then((result) => {

    // First handler function

    return modifiedResult;

  })

  .then((modifiedResult) => {

    // Second handler function

    return finalResult;

  })

  .then((finalResult) => {

    // Third handler function

    console.log(finalResult);

  })

  .catch((error) => {

    // Handle errors in any of the above steps

    console.error(error);

  });

**d.      How do you attach a handler function that is performed in a case of a rejected promise?**  
Attaching a Handler Function for a Rejected Promise:

You can attach a handler function for a rejected promise using the catch method. This method is called if any of the preceding handlers encounter an error or if the promise is explicitly rejected.

Example:

 myPromise

  .then((result) => {

    // Handler for fulfilled state

    return modifiedResult;

  })

  .catch((error) => {

    // Handler for rejected state

    console.error(error);

  });

**6. JS-PROMISE-P-1 (2 points)**

This task will be done together with the instructor during one of the contact lessons.

The idea is to get some insight to the basics of JavaScript promises and AJAX by programming together the example below.

const getMembers = count => //XXXXXXXXXXXX

    const apiUrl = `<https://api.randomuser.me/?nat=FI&results=$>{count}`;

    const request = new XMLHttpRequest();

    request.open('GET', apiUrl);

    request.onload = () => (request.status === 200) ?

//XXXXXXX

    request.onerror = //XXXXXXXXXXXXXXXXX

    request.send();

})

var displayFiveFinnishFakeMembers = () => {

  //We’ll implement this!

}

function initiateApplication() {

    var requireBtn = document.querySelector('#b1');

    requireBtn.addEventListener('click', displayFiveFinnishFakeMembers);

}

window.addEventListener('load', () => {

    initiateApplication();

});

const getMembers = count => {

    return new Promise((resolve, reject) => {

      const apiUrl = `https://api.randomuser.me/?nat=FI&results=${count}`;

      const request = new XMLHttpRequest();

      request.open('GET', apiUrl);

      request.onload = () => {

        if (request.status === 200) {

          const response = JSON.parse(request.responseText);

          resolve(response.results);

        } else {

          reject(`Error: ${request.status}`);

        }

      };

      request.onerror = () => {

        reject('Network error');

      };

      request.send();

    });

  };

  const displayFiveFinnishFakeMembers = () => {

    const count = 5;

    // Call the getMembers function to fetch Finnish fake members

    getMembers(count)

      .then(members => {

        // Process and display the members

        const memberList = document.getElementById('member-list');

        memberList.innerHTML = ''; // Clear previous results

        members.forEach(member => {

          const listItem = document.createElement('li');

          listItem.textContent = `${member.name.first} ${member.name.last}`;

          memberList.appendChild(listItem);

        });

      })

      .catch(error => {

        // Handle errors

        console.error(error);

      });

  };

  function initiateApplication() {

    var requireBtn = document.querySelector('#b1');

    requireBtn.addEventListener('click', displayFiveFinnishFakeMembers);

  }

  window.addEventListener('load', () => {

    initiateApplication();

  });