**Exercise 1: Counter with Closures**

Create a counter function that returns an object containing two methods: increment and getValue. The increment method should increase the value of a counter, and getValue should return the current value of the counter.

**Solution:**

function counter() {

let count = 0; // private variable

return {

increment: function() {

count++; // closure that accesses count

},

getValue: function() {

return count; // closure that returns count

}

};

}

const myCounter = counter();

myCounter.increment();

myCounter.increment();

console.log(myCounter.getValue()); // Output: 2

**Explanation:**

Here, count is a private variable, accessible only inside the increment and getValue methods due to closures. The inner functions form closures that "remember" the count variable even after the counter function has executed.

**Exercise 2: Customizable Greeting**

Write a function createGreeter that takes a greeting message (like "Hello") and returns a function that takes a name and prints a greeting using the message.

**Solution:**

function createGreeter(greeting) {

return function(name) {

console.log(`${greeting}, ${name}!`);

};

}

const greeter = createGreeter("Hello");

greeter("Alice"); // Output: Hello, Alice!

greeter("Bob"); // Output: Hello, Bob!

**Explanation:**

Here, createGreeter returns a function that uses the greeting variable, which is captured in the closure, allowing each created greeter to use its own greeting message.

**Exercise 3: Once Function**

Write a function once that takes another function as an argument and returns a new function that can only be called once. Subsequent calls should do nothing.

**Solution:**

function once(fn) {

let called = false;

return function(...args) {

if (!called) {

called = true;

return fn(...args); // call the function only once

}

};

}

const logOnce = once(console.log);

logOnce("Hello!"); // Output: Hello!

logOnce("World!"); // No output

**Explanation:**

The called variable is remembered by the closure, and once the inner function is executed, called is set to true, preventing further executions.

**Exercise 4: Multiply with Closures**

Create a function multiply that takes a number and returns a function that multiplies any given number by the first number.

**Solution:**

function multiply(x) {

return function(y) {

return x \* y; // closure that remembers x

};

}

const double = multiply(2);

console.log(double(5)); // Output: 10

const triple = multiply(3);

console.log(triple(4)); // Output: 12

**Explanation:**

The multiply function returns a function that captures the value of x in its closure, allowing you to create customized multiplier functions.

**Exercise 5: Private Variables with Closures**

Write a function bankAccount that returns an object with two methods: deposit and withdraw. Use closures to keep the balance private.

**Solution:**

function bankAccount(initialBalance) {

let balance = initialBalance; // private balance

return {

deposit: function(amount) {

balance += amount;

console.log(`Deposited: $${amount}. New balance: $${balance}`);

},

withdraw: function(amount) {

if (amount > balance) {

console.log('Insufficient funds!');

} else {

balance -= amount;

console.log(`Withdrew: $${amount}. New balance: $${balance}`);

}

},

getBalance: function() {

return balance;

}

};

}

const myAccount = bankAccount(100);

myAccount.deposit(50); // Output: Deposited: $50. New balance: $150

myAccount.withdraw(30); // Output: Withdrew: $30. New balance: $120

console.log(myAccount.getBalance()); // Output: 120

**Explanation:**

The balance variable is private and can only be accessed or modified through the methods defined in the returned object. These methods form closures around the balance variable, making it private to the outer bankAccount function.

**Exercise 6: Task Manager with Closures**

Create a taskManager function that allows you to manage tasks. It should return an object with methods to:

* addTask(task): Add a task.
* removeTask(task): Remove a task by its name.
* getAllTasks(): Return all the tasks.
* clearTasks(): Clear all tasks.

The tasks should be private and only accessible via the provided methods.

**Solution:**

function taskManager() {

let tasks = []; // private task list

return {

addTask: function(task) {

tasks.push(task);

console.log(`Task '${task}' added!`);

},

removeTask: function(task) {

const index = tasks.indexOf(task);

if (index > -1) {

tasks.splice(index, 1);

console.log(`Task '${task}' removed!`);

} else {

console.log(`Task '${task}' not found!`);

}

},

getAllTasks: function() {

return tasks.slice(); // return a copy of tasks

},

clearTasks: function() {

tasks = [];

console.log('All tasks cleared!');

}

};

}

const myTasks = taskManager();

myTasks.addTask('Complete JavaScript project');

myTasks.addTask('Go for a run');

console.log(myTasks.getAllTasks()); // Output: ['Complete JavaScript project', 'Go for a run']

myTasks.removeTask('Go for a run');

console.log(myTasks.getAllTasks()); // Output: ['Complete JavaScript project']

myTasks.clearTasks();

console.log(myTasks.getAllTasks()); // Output: []

**Explanation:**

Here, the tasks array is private to the taskManager function, and we access or modify it through the closures created inside the returned object. Each method forms a closure around the tasks variable.

**Exercise 7: Timer with Start, Stop, and Reset**

Create a timer function that tracks the elapsed time. The timer should:

* Start counting time when start() is called.
* Stop counting when stop() is called.
* Reset the time to 0 when reset() is called.
* Return the current time with getTime().

Use closures to manage the timer's state (like the elapsed time and interval ID).

**Solution:**

function timer() {

let startTime = 0;

let elapsedTime = 0;

let intervalId = null;

return {

start: function() {

if (!intervalId) {

startTime = Date.now() - elapsedTime; // compensate for paused time

intervalId = setInterval(() => {

elapsedTime = Date.now() - startTime;

console.log(`Elapsed time: ${(elapsedTime / 1000).toFixed(2)} seconds`);

}, 1000);

console.log('Timer started!');

}

},

stop: function() {

if (intervalId) {

clearInterval(intervalId);

intervalId = null;

console.log('Timer stopped!');

}

},

reset: function() {

elapsedTime = 0;

startTime = 0;

clearInterval(intervalId);

intervalId = null;

console.log('Timer reset to 0!');

},

getTime: function() {

return (elapsedTime / 1000).toFixed(2); // return time in seconds

}

};

}

const myTimer = timer();

myTimer.start(); // Start the timer

setTimeout(() => myTimer.stop(), 5000); // Stop the timer after 5 seconds

setTimeout(() => {

console.log(`Final Time: ${myTimer.getTime()} seconds`); // Output elapsed time

}, 6000);

**Explanation:**

This example uses closures to keep track of the startTime, elapsedTime, and intervalId. The start(), stop(), and reset() methods access and modify these variables, forming closures around them.

**Exercise 8: Secure Bank Account with Deposit, Withdraw, and Statement**

Write a bankAccount function that:

* Allows the user to deposit money.
* Allows the user to withdraw money.
* Keeps track of transaction history.
* Provides a statement of all transactions.

Each transaction should be logged privately, and the statement should return a list of all transactions along with the account balance.

**Solution:**

function bankAccount(initialBalance) {

let balance = initialBalance;

let transactions = [];

function addTransaction(type, amount) {

transactions.push({ type, amount, date: new Date() });

}

return {

deposit: function(amount) {

if (amount > 0) {

balance += amount;

addTransaction('Deposit', amount);

console.log(`Deposited $${amount}. New balance: $${balance}`);

} else {

console.log('Invalid deposit amount!');

}

},

withdraw: function(amount) {

if (amount > 0 && amount <= balance) {

balance -= amount;

addTransaction('Withdraw', amount);

console.log(`Withdrew $${amount}. New balance: $${balance}`);

} else {

console.log('Invalid withdrawal amount or insufficient funds!');

}

},

getStatement: function() {

console.log('--- Statement ---');

transactions.forEach(transaction => {

console.log(`${transaction.date}: ${transaction.type} of $${transaction.amount}`);

});

console.log(`Current balance: $${balance}`);

}

};

}

const account = bankAccount(1000);

account.deposit(200);

account.withdraw(150);

account.getStatement(); // Shows all transactions and balance

**Explanation:**

In this example, balance and transactions are private, and the methods form closures around them. The addTransaction function is also private, ensuring that transaction logging is only done through the public methods.

**Exercise 9: Voting System with Multiple Candidates**

Create a votingSystem function that allows users to vote for candidates. It should:

* Allow the addition of new candidates.
* Allow voting for any candidate.
* Show the results with the vote counts of all candidates.

Use closures to maintain the vote count for each candidate.

**Solution:**

function votingSystem() {

let candidates = {};

return {

addCandidate: function(name) {

if (!candidates[name]) {

candidates[name] = 0;

console.log(`Candidate '${name}' added!`);

} else {

console.log(`Candidate '${name}' already exists!`);

}

},

vote: function(name) {

if (candidates[name] !== undefined) {

candidates[name]++;

console.log(`Vote cast for '${name}'!`);

} else {

console.log(`Candidate '${name}' does not exist!`);

}

},

getResults: function() {

console.log('--- Voting Results ---');

for (let candidate in candidates) {

console.log(`${candidate}: ${candidates[candidate]} votes`);

}

}

};

}

const election = votingSystem();

election.addCandidate('Alice');

election.addCandidate('Bob');

election.vote('Alice');

election.vote('Bob');

election.vote('Alice');

election.getResults(); // Output: Alice: 2 votes, Bob: 1 vote

**Explanation:**

The candidates and their vote counts are stored in the private candidates object. The closure ensures that only the methods provided can modify or access the vote counts.

**Exercise 10: Event Subscription System**

Create an eventManager function that allows event subscription and notification. It should:

* Allow subscription to events with on(eventName, callback).
* Trigger events using emit(eventName, data).
* Keep track of event subscribers privately.

**Solution:**

function eventManager() {

let events = {};

return {

on: function(eventName, callback) {

if (!events[eventName]) {

events[eventName] = [];

}

events[eventName].push(callback);

console.log(`Subscribed to event: ${eventName}`);

},

emit: function(eventName, data) {

if (events[eventName]) {

events[eventName].forEach(callback => callback(data));

} else {

console.log(`No subscribers for event: ${eventName}`);

}

}

};

}

const myEvents = eventManager();

myEvents.on('greet', name => console.log(`Hello, ${name}!`));

myEvents.on('greet', name => console.log(`How are you, ${name}?`));

myEvents.emit('greet', 'Alice'); // Output: Hello, Alice! How are you, Alice?

**Explanation:**

Here, events is a private object storing event subscribers. The on method allows subscribing, and the emit method triggers the callbacks associated with an event. The closure maintains access to the events object for all future calls.

**Exercise 11: To-Do List with Categories**

Create a toDoList function that manages tasks in multiple categories. Each category has its own set of tasks. The system should support the following:

* addCategory(name): Adds a new category.
* removeCategory(name): Removes a category.
* addTask(category, task): Adds a task to a specific category.
* removeTask(category, task): Removes a task from a specific category.
* getTasks(category): Returns all tasks for a specific category.
* clearTasks(category): Clears all tasks in a category.
* getAllCategories(): Returns all category names.
* getAllTasks(): Returns all tasks across all categories.

Each method will work with closures to keep tasks and categories private.

**Solution:**

function toDoList() {

let categories = {};

return {

addCategory: function(name) {

if (!categories[name]) {

categories[name] = [];

console.log(`Category '${name}' added!`);

} else {

console.log(`Category '${name}' already exists!`);

}

},

removeCategory: function(name) {

if (categories[name]) {

delete categories[name];

console.log(`Category '${name}' removed!`);

} else {

console.log(`Category '${name}' does not exist!`);

}

},

addTask: function(category, task) {

if (categories[category]) {

categories[category].push(task);

console.log(`Task '${task}' added to category '${category}'!`);

} else {

console.log(`Category '${category}' does not exist!`);

}

},

removeTask: function(category, task) {

if (categories[category]) {

const index = categories[category].indexOf(task);

if (index > -1) {

categories[category].splice(index, 1);

console.log(`Task '${task}' removed from category '${category}'!`);

} else {

console.log(`Task '${task}' not found in category '${category}'!`);

}

} else {

console.log(`Category '${category}' does not exist!`);

}

},

getTasks: function(category) {

if (categories[category]) {

return categories[category].slice();

} else {

console.log(`Category '${category}' does not exist!`);

return [];

}

},

clearTasks: function(category) {

if (categories[category]) {

categories[category] = [];

console.log(`All tasks cleared from category '${category}'!`);

} else {

console.log(`Category '${category}' does not exist!`);

}

},

getAllCategories: function() {

return Object.keys(categories);

},

getAllTasks: function() {

let allTasks = [];

for (let category in categories) {

allTasks.push(...categories[category]);

}

return allTasks;

}

};

}

const myToDo = toDoList();

myToDo.addCategory('Work');

myToDo.addCategory('Personal');

myToDo.addTask('Work', 'Finish project');

myToDo.addTask('Personal', 'Buy groceries');

console.log(myToDo.getTasks('Work')); // Output: ['Finish project']

console.log(myToDo.getAllCategories()); // Output: ['Work', 'Personal']

myToDo.removeTask('Personal', 'Buy groceries');

console.log(myToDo.getAllTasks()); // Output: ['Finish project']

**Explanation:**

In this example, we have **8 closures**:

1. addCategory()
2. removeCategory()
3. addTask()
4. removeTask()
5. getTasks()
6. clearTasks()
7. getAllCategories()
8. getAllTasks()

Each method forms a closure around the categories object, allowing interaction with category and task data privately.

**Exercise 12: Library System with Books and Authors**

Create a library function that manages books and authors. The system should:

* addAuthor(name): Add an author to the system.
* addBook(author, book): Add a book to a specific author.
* removeBook(author, book): Remove a book from an author.
* removeAuthor(name): Remove an author and all their books.
* getAuthorBooks(author): Get all books for a specific author.
* getAllAuthors(): Get a list of all authors.
* getAllBooks(): Get a list of all books across all authors.
* clearLibrary(): Remove all authors and books.

**Solution:**

function library() {

let authors = {};

return {

addAuthor: function(name) {

if (!authors[name]) {

authors[name] = [];

console.log(`Author '${name}' added!`);

} else {

console.log(`Author '${name}' already exists!`);

}

},

addBook: function(author, book) {

if (authors[author]) {

authors[author].push(book);

console.log(`Book '${book}' added to author '${author}'!`);

} else {

console.log(`Author '${author}' does not exist!`);

}

},

removeBook: function(author, book) {

if (authors[author]) {

const index = authors[author].indexOf(book);

if (index > -1) {

authors[author].splice(index, 1);

console.log(`Book '${book}' removed from author '${author}'!`);

} else {

console.log(`Book '${book}' not found for author '${author}'!`);

}

} else {

console.log(`Author '${author}' does not exist!`);

}

},

removeAuthor: function(name) {

if (authors[name]) {

delete authors[name];

console.log(`Author '${name}' and all their books removed!`);

} else {

console.log(`Author '${name}' does not exist!`);

}

},

getAuthorBooks: function(author) {

if (authors[author]) {

return authors[author].slice();

} else {

console.log(`Author '${author}' does not exist!`);

return [];

}

},

getAllAuthors: function() {

return Object.keys(authors);

},

getAllBooks: function() {

let allBooks = [];

for (let author in authors) {

allBooks.push(...authors[author]);

}

return allBooks;

},

clearLibrary: function() {

authors = {};

console.log('Library cleared!');

}

};

}

const myLibrary = library();

myLibrary.addAuthor('J.K. Rowling');

myLibrary.addBook('J.K. Rowling', 'Harry Potter');

myLibrary.addAuthor('J.R.R. Tolkien');

myLibrary.addBook('J.R.R. Tolkien', 'The Hobbit');

console.log(myLibrary.getAuthorBooks('J.K. Rowling')); // Output: ['Harry Potter']

myLibrary.removeBook('J.R.R. Tolkien', 'The Hobbit');

console.log(myLibrary.getAllBooks()); // Output: ['Harry Potter']

myLibrary.clearLibrary();

console.log(myLibrary.getAllAuthors()); // Output: []

**Explanation:**

In this example, we have **8 closures**:

1. addAuthor()
2. addBook()
3. removeBook()
4. removeAuthor()
5. getAuthorBooks()
6. getAllAuthors()
7. getAllBooks()
8. clearLibrary()

Each closure interacts with the authors object to manage data privately, ensuring encapsulation.

**Exercise 13: Shopping Cart with Discounts**

Create a shoppingCart function that manages a cart and applies discounts. It should:

* addItem(name, price): Add an item to the cart.
* removeItem(name): Remove an item from the cart.
* applyDiscount(code, percentage): Apply a discount code.
* removeDiscount(): Remove any applied discount.
* getTotal(): Return the total price of all items in the cart after discount.
* getItems(): Return all items in the cart.
* clearCart(): Clear the cart.
* getDiscount(): Return the current discount information.

**Solution:**

function shoppingCart() {

let items = [];

let discount = { code: null, percentage: 0 };

return {

addItem: function(name, price) {

items.push({ name, price });

console.log(`Item '${name}' added with price $${price}!`);

},

removeItem: function(name) {

const index = items.findIndex(item => item.name === name);

if (index > -1) {

items.splice(index, 1);

console.log(`Item '${name}' removed!`);

} else {

console.log(`Item '${name}' not found!`);

}

},

applyDiscount: function(code, percentage) {

if (percentage > 0 && percentage <= 100) {

discount = { code, percentage };

console.log(`Discount '${code}' applied with ${percentage}% off!`);

} else {

console.log('Invalid discount percentage!');

}

},

removeDiscount: function() {

discount = { code: null, percentage: 0 };

console.log('Discount removed!');

},

getTotal: function() {

let total = items.reduce((sum, item) => sum + item.price, 0);

total -= total \* (discount.percentage / 100);

return total.toFixed(2);

},

getItems: function() {

return items.slice();

},

clearCart: function() {

items = [];

console.log('Cart cleared!');

},

getDiscount: function() {

return discount;

}

};

}

const cart = shoppingCart();

cart.addItem('Shirt', 25);

cart.addItem('Jeans', 50);

cart.applyDiscount('SUMMER10', 10);

console.log(cart.getTotal()); // Output: 67.50

cart.removeDiscount();

console.log(cart.getTotal()); // Output: 75.00

cart.clearCart();

**Explanation:**

This exercise contains **8 closures**:

1. addItem()
2. removeItem()
3. applyDiscount()
4. removeDiscount()
5. getTotal()
6. getItems()
7. clearCart()
8. getDiscount()

The closures interact with items and discount to manage the shopping cart and apply discounts effectively.

**Exercise 14: Online Store with Inventory Management, Cart, and Promotions**

Create an online store system with the following features:

* **Inventory**:
  + addItem(name, price, stock): Add an item to the inventory.
  + removeItem(name): Remove an item from the inventory.
  + updateStock(name, stock): Update the stock quantity of an item.
  + getInventory(): Return all items in the inventory.
* **Shopping Cart**:
  + addToCart(name, quantity): Add an item to the cart if it exists in inventory and has enough stock.
  + removeFromCart(name): Remove an item from the cart.
  + getCartTotal(): Return the total price of items in the cart.
  + clearCart(): Clear the cart.
* **Promotions**:
  + applyPromo(code, percentage): Apply a promotion code for a percentage discount.
  + removePromo(): Remove the promotion code.
  + getTotalWithPromo(): Get the cart total after applying the promo.
* **Checkout**:
  + checkout(): Complete the purchase by reducing stock for each item in the cart.

**Solution**:

function onlineStore() {

let inventory = {};

let cart = [];

let promo = { code: null, percentage: 0 };

return {

// Inventory management

addItem: function(name, price, stock) {

if (!inventory[name]) {

inventory[name] = { price, stock };

console.log(`Item '${name}' added to inventory!`);

} else {

console.log(`Item '${name}' already exists!`);

}

},

removeItem: function(name) {

if (inventory[name]) {

delete inventory[name];

console.log(`Item '${name}' removed from inventory!`);

} else {

console.log(`Item '${name}' not found in inventory!`);

}

},

updateStock: function(name, stock) {

if (inventory[name]) {

inventory[name].stock = stock;

console.log(`Stock updated for '${name}'!`);

} else {

console.log(`Item '${name}' does not exist!`);

}

},

getInventory: function() {

return { ...inventory };

},

// Shopping cart management

addToCart: function(name, quantity) {

if (inventory[name]) {

if (inventory[name].stock >= quantity) {

let existingItem = cart.find(item => item.name === name);

if (existingItem) {

existingItem.quantity += quantity;

} else {

cart.push({ name, quantity });

}

console.log(`${quantity} of '${name}' added to the cart!`);

} else {

console.log(`Not enough stock for '${name}'!`);

}

} else {

console.log(`Item '${name}' not found in inventory!`);

}

},

removeFromCart: function(name) {

const index = cart.findIndex(item => item.name === name);

if (index > -1) {

cart.splice(index, 1);

console.log(`Item '${name}' removed from the cart!`);

} else {

console.log(`Item '${name}' not in the cart!`);

}

},

getCartTotal: function() {

let total = cart.reduce((sum, item) => sum + (inventory[item.name].price \* item.quantity), 0);

return total.toFixed(2);

},

clearCart: function() {

cart = [];

console.log('Cart cleared!');

},

// Promotions management

applyPromo: function(code, percentage) {

if (percentage > 0 && percentage <= 100) {

promo = { code, percentage };

console.log(`Promo code '${code}' applied with ${percentage}% discount!`);

} else {

console.log('Invalid promo percentage!');

}

},

removePromo: function() {

promo = { code: null, percentage: 0 };

console.log('Promo code removed!');

},

getTotalWithPromo: function() {

let total = cart.reduce((sum, item) => sum + (inventory[item.name].price \* item.quantity), 0);

total -= total \* (promo.percentage / 100);

return total.toFixed(2);

},

// Checkout process

checkout: function() {

cart.forEach(item => {

if (inventory[item.name]) {

inventory[item.name].stock -= item.quantity;

}

});

this.clearCart();

console.log('Purchase completed! Stock updated.');

}

};

}

// Example usage:

const store = onlineStore();

// Inventory management

store.addItem('Laptop', 1200, 10);

store.addItem('Mouse', 20, 50);

store.addItem('Keyboard', 60, 30);

// Cart management

store.addToCart('Laptop', 1);

store.addToCart('Mouse', 2);

// Promotions

store.applyPromo('SUMMER10', 10);

// Checkout

console.log('Total with promo:', store.getTotalWithPromo()); // Discounted total

store.checkout();

console.log('Remaining Inventory:', store.getInventory());

**Explanation:**

This system contains **12 closures**:

1. addItem()
2. removeItem()
3. updateStock()
4. getInventory()
5. addToCart()
6. removeFromCart()
7. getCartTotal()
8. clearCart()
9. applyPromo()
10. removePromo()
11. getTotalWithPromo()
12. checkout()

Each closure interacts with inventory, cart, and promo objects, allowing private management of store data and ensuring encapsulation.

**Exercise 15: Task Manager with User Access Control**

Create a taskManager function that manages tasks for multiple users. It should handle:

* **User Management**:
  + addUser(name): Add a user to the system.
  + removeUser(name): Remove a user and all their tasks.
  + getUsers(): Return a list of all users.
* **Task Management**:
  + addTask(user, task): Add a task for a specific user.
  + removeTask(user, task): Remove a task for a specific user.
  + getUserTasks(user): Get all tasks for a specific user.
  + clearUserTasks(user): Clear all tasks for a specific user.
* **Access Control**:
  + grantAccess(user, task): Grant access to a task for another user.
  + revokeAccess(user, task): Revoke access to a task from another user.
  + getSharedTasks(user): Return all tasks that the user has access to (granted by others).

**Solution**:

function taskManager() {

let users = {};

return {

// User management

addUser: function(name) {

if (!users[name]) {

users[name] = { tasks: [], sharedTasks: [] };

console.log(`User '${name}' added!`);

} else {

console.log(`User '${name}' already exists!`);

}

},

removeUser: function(name) {

if (users[name]) {

delete users[name];

console.log(`User '${name}' removed!`);

} else {

console.log(`User '${name}' does not exist!`);

}

},

getUsers: function() {

return Object.keys(users);

},

// Task management

addTask: function(user, task) {

if (users[user]) {

users[user].tasks.push(task);

console.log(`Task '${task}' added to user '${user}'!`);

} else {

console.log(`User '${user}' does not exist!`);

}

},

removeTask: function(user, task) {

if (users[user]) {

const index = users[user].tasks.indexOf(task);

if (index > -1) {

users[user].tasks.splice(index, 1);

console.log(`Task '${task}' removed from user '${user}'!`);

} else {

console.log(`Task '${task}' not found for user '${user}'!`);

}

} else {

console.log(`User '${user}' does not exist!`);

}

},

getUserTasks: function(user) {

if (users[user]) {

return [...users[user].tasks];

} else {

console.log(`User '${user}' does not exist!`);

return [];

}

},

clearUserTasks: function(user) {

if (users[user]) {

users[user].tasks = [];

console.log(`All tasks cleared for user '${user}'!`);

} else {

console.log(`User '${user}' does not exist!`);

}

},

// Access control

grantAccess: function(fromUser, task, toUser) {

if (users[fromUser] && users[toUser]) {

if (users[fromUser].tasks.includes(task)) {

users[toUser].sharedTasks.push(task);

console.log(`Task '${task}' from '${fromUser}' shared with '${toUser}'!`);

} else {

console.log(`Task '${task}' not found for user '${fromUser}'!`);

}

} else {

console.log(`Either '${fromUser}' or '${toUser}' does not exist!`);

}

},

revokeAccess: function(fromUser, task, toUser) {

if (users[fromUser] && users[toUser]) {

const index = users[toUser].sharedTasks.indexOf(task);

if (index > -1) {

users[toUser].sharedTasks.splice(index, 1);

console.log(`Access to task '${task}' revoked from user '${toUser}'!`);

} else {

console.log(`User '${toUser}' does not have access to task '${task}'!`);

}

} else {

console.log(`Either '${fromUser}' or '${toUser}' does not exist!`);

}

},

getSharedTasks: function(user) {

if (users[user]) {

return [...users[user].sharedTasks];

} else {

console.log(`User '${user}' does not exist!`);

return [];

}

}

};

}

**// Example usage:**

const manager = taskManager();

// User management

manager.addUser('Alice');

manager.addUser('Bob');

// Task management

manager.addTask('Alice', 'Complete Project');

manager.addTask('Alice', 'Review Code');

manager.addTask('Bob', 'Write Documentation');

// Access control

manager.grantAccess('Alice', 'Complete Project', 'Bob');

console.log('Bob\'s shared tasks:', manager.getSharedTasks('Bob')); // ['Complete Project']

manager.revokeAccess('Alice', 'Complete Project', 'Bob');

console.log('Bob\'s shared tasks after revocation:', manager.getSharedTasks('Bob')); // []

**Explanation:**

This exercise contains **10 closures**:

1. addUser()
2. removeUser()
3. getUsers()
4. addTask()
5. removeTask()
6. getUserTasks()
7. clearUserTasks()
8. grantAccess()
9. revokeAccess()
10. getSharedTasks()

These closures manage user tasks, task-sharing capabilities, and encapsulate user data within the taskManager object.