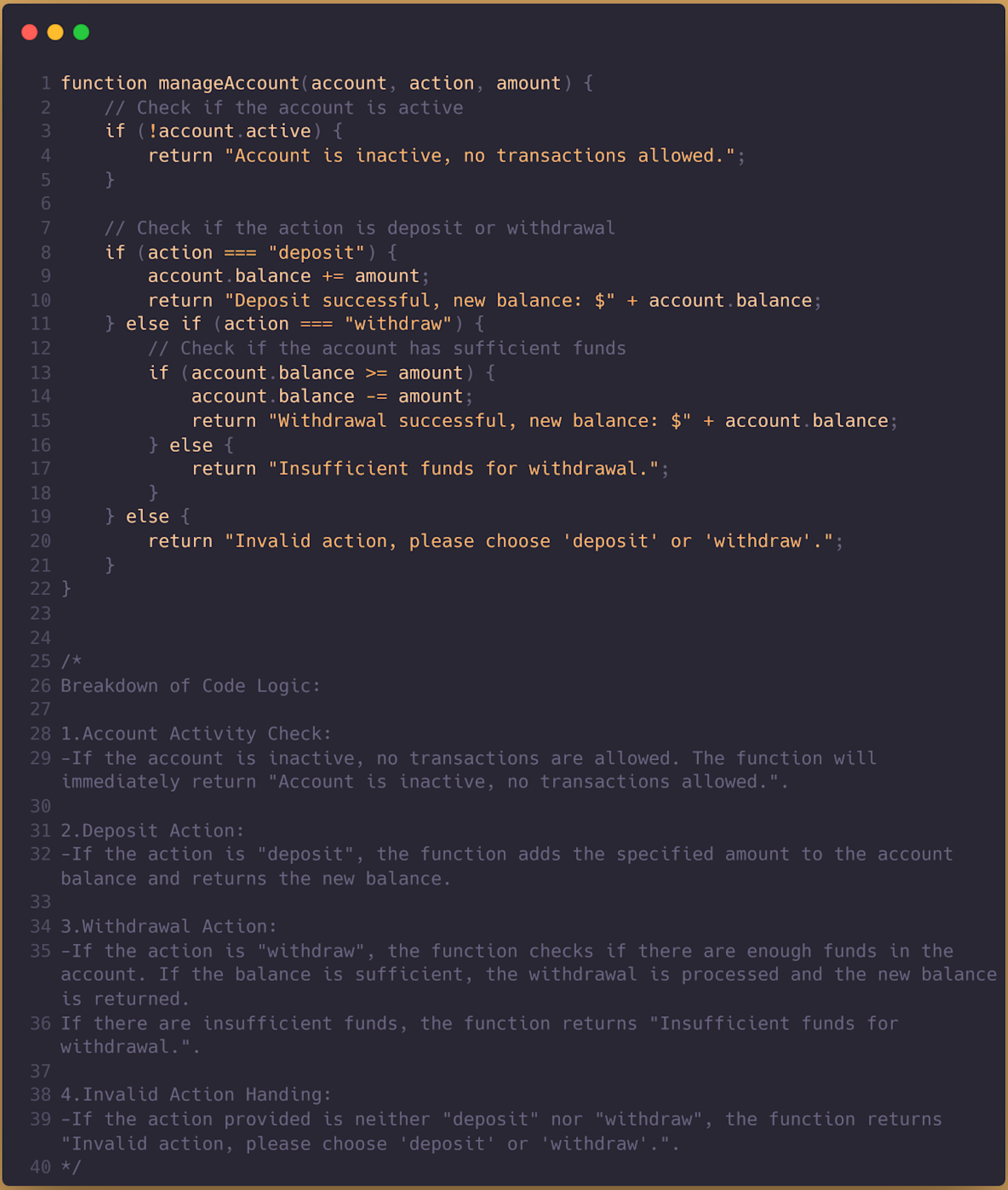


```javascript
function manageAccount(account, action, amount) {
    // Check if the account is active
    if (!account.active) {
        return "Account is inactive, no transactions allowed.";
    }

    // Check if the action is deposit or withdrawal
    if (action === "deposit") {
        account.balance += amount;
        return "Deposit successful, new balance: $" + account.balance;
    } else if (action === "withdraw") {
        // Check if the account has sufficient funds
        if (account.balance >= amount) {
            account.balance -= amount;
            return "Withdrawal successful, new balance: $" + account.balance;
        } else {
            return "Insufficient funds for withdrawal.";
        }
    } else {
        return "Invalid action, please choose 'deposit' or 'withdraw'.";
    }
}


/*
Breakdown of Code Logic:

1.Account Activity Check:
-If the account is inactive, no transactions are allowed. The function will immediately return "Account is inactive, no transactions allowed.".

2.Deposit Action:
-If the action is "deposit", the function adds the specified amount to the account balance and returns the new balance.

3.Withdrawal Action:
-If the action is "withdraw", the function checks if there are enough funds in the account. If the balance is sufficient, the withdrawal is processed and the new balance is returned.
If there are insufficient funds, the function returns "Insufficient funds for withdrawal.".

4.Invalid Action Handing:
-If the action provided is neither "deposit" nor "withdraw", the function returns "Invalid action, please choose 'deposit' or 'withdraw'.".
*/
```

I’m going to design a white box testing for bank management system using four techniques,

1.Code Coverage

2.Path Testing

3.Control Flow Testing

4.Data Flow Testing

The manageAccount(account, action, amount) function performs three essential checks:

1. It first verifies if the account is active.
2. Based on the action parameter, it proceeds with either a deposit or a withdrawal.
3. It handles any invalid actions that aren’t "deposit" or "withdraw."

1. Code Coverage Testing

To ensure complete code coverage, each line should be executed at least once by accounting for all possible conditions.

Test Cases for Code Coverage:

1. Active account, deposit action:
    a. Input: { active: true, balance: 100 }, "deposit", 50
    b. Expected Output: "Deposit successful, new balance: $150"

2. Inactive account:
    a. Input: { active: false, balance: 100 }, "deposit", 50
    b. Expected Output: "Account is inactive, no transactions allowed."

3. Active account, withdrawal with sufficient funds:
    a. Input: { active: true, balance: 100 }, "withdraw", 50
    b. Expected Output: "Withdrawal successful, new balance: $50"

4. Active account, withdrawal with insufficient funds:
    a. Input: { active: true, balance: 100 }, "withdraw", 150
    b. Expected Output: "Insufficient funds for withdrawal."

5. Active account, invalid action:
    a. Input: { active: true, balance: 100 }, "transfer", 50
    b. Expected Output: "Invalid action, please choose 'deposit' or 'withdraw'."

2. Path Testing

Path testing covers all possible paths the code can take. This function has five main paths.

Paths and Corresponding Test Cases:

1. Inactive account:
    a. Input: { active: false, balance: 100 }, "deposit", 50
    b. Expected Output: "Account is inactive, no transactions allowed."
2. Active account, valid deposit:
    a. Input: { active: true, balance: 100 }, "deposit", 50
    b. Expected Output: "Deposit successful, new balance: $150"
3. Active account, valid withdrawal with sufficient funds:
    a. Input: { active: true, balance: 100 }, "withdraw", 50
    b. Expected Output: "Withdrawal successful, new balance: $50"
4. Active account, withdrawal with insufficient funds:
    a. Input: { active: true, balance: 100 }, "withdraw", 150
    b. Expected Output: "Insufficient funds for withdrawal."
5. Active account, invalid action:
    a. Input: { active: true, balance: 100 }, "transfer", 50
    b. Expected Output: "Invalid action, please choose 'deposit' or 'withdraw'."

3. Control Flow Testing

Control Flow Testing ensures the conditions and branches flow as expected.

Control Flow Test Cases: The cases outlined above for code coverage and path testing also cover control flow. Each branch is checked, including:

1. Both true and false outcomes for account.active
2. The deposit and withdraw branches for action
3. The else branch for invalid actions

4. Data Flow Testing

This testing focuses on the data flow, especially ensuring that balance updates correctly for deposits and withdrawals.

Test Cases for Data Flow:

1. Deposit action data flow:
    a. Input: { active: true, balance: 100 }, "deposit", 50
    b. Expected Output: "Deposit successful, new balance: $150"
    c. Explanation: Ensures balance increases correctly with a deposit.
2. Withdrawal action data flow with sufficient funds:
    a. Input: { active: true, balance: 100 }, "withdraw", 50
    b. Expected Output: "Withdrawal successful, new balance: $50"
    c. Explanation: Ensures balance decreases correctly when funds are sufficient.
3. Withdrawal action data flow with insufficient funds:
    a. Input: { active: true, balance: 100 }, "withdraw", 150
    b. Expected Output: "Insufficient funds for withdrawal."
    c. Explanation: Verifies that balance remains unchanged when funds are insufficient.

5.Error Check

So reviewing the function manageAccount(account, action, amount), no significant issues or bugs were detected in its core functionality. The function is well-structured to handle the main operations expected in a simple banking management system.

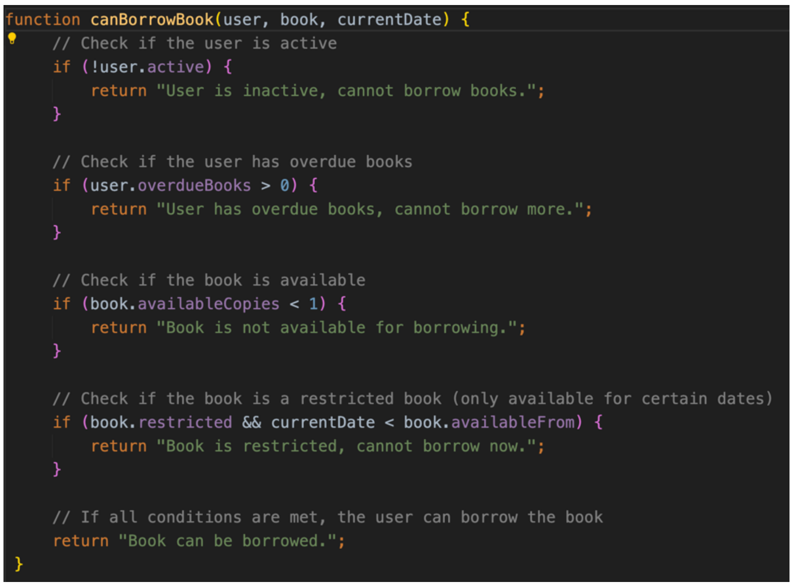
So In summary, I can say the manageAccount function effectively handles all expected scenarios for a basic account management system, including,

I. Inactive accounts

II. Valid deposit and withdrawal actions

III. Withdrawal attempts with insufficient funds

IV. Invalid actions



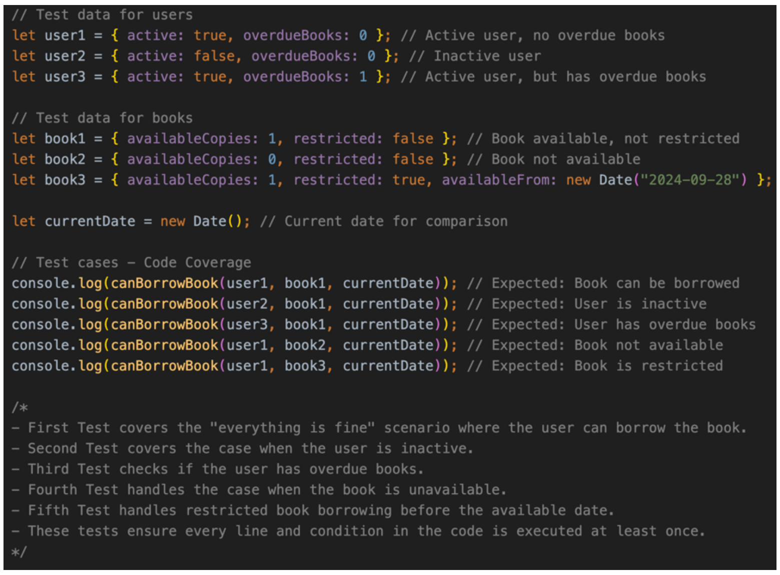
```javascript
function canBorrowBook(user, book, currentDate) {
    // Check if the user is active
    if (!user.active) {
        return "User is inactive, cannot borrow books.";
    }

    // Check if the user has overdue books
    if (user.overdueBooks > 0) {
        return "User has overdue books, cannot borrow more.";
    }

    // Check if the book is available
    if (book.availableCopies < 1) {
        return "Book is not available for borrowing.";
    }

    // Check if the book is a restricted book (only available for certain dates)
    if (book.restricted && currentDate < book.availableFrom) {
        return "Book is restricted, cannot borrow now.";
    }

    // If all conditions are met, the user can borrow the book
    return "Book can be borrowed.";
}
```



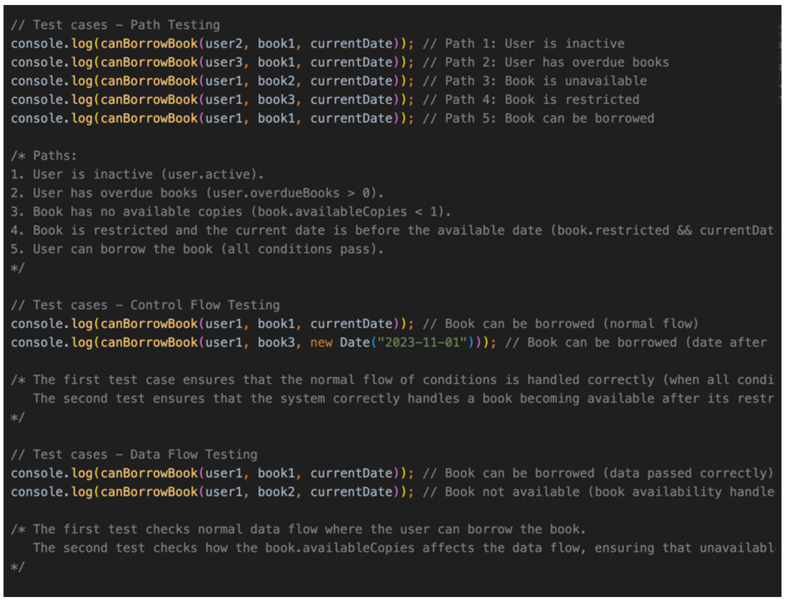
```javascript
// Test data for users
let user1 = { active: true, overdueBooks: 0 }; // Active user, no overdue books
let user2 = { active: false, overdueBooks: 0 }; // Inactive user
let user3 = { active: true, overdueBooks: 1 }; // Active user, but has overdue books

// Test data for books
let book1 = { availableCopies: 1, restricted: false }; // Book available, not restricted
let book2 = { availableCopies: 0, restricted: false }; // Book not available
let book3 = { availableCopies: 1, restricted: true, availableFrom: new Date("2024-09-28") };

let currentDate = new Date(); // Current date for comparison

// Test cases - Code Coverage
console.log(canBorrowBook(user1, book1, currentDate)); // Expected: Book can be borrowed
console.log(canBorrowBook(user2, book1, currentDate)); // Expected: User is inactive
console.log(canBorrowBook(user3, book1, currentDate)); // Expected: User has overdue books
console.log(canBorrowBook(user1, book2, currentDate)); // Expected: Book not available
console.log(canBorrowBook(user1, book3, currentDate)); // Expected: Book is restricted

/*
- First Test covers the "everything is fine" scenario where the user can borrow the book.
- Second Test covers the case when the user is inactive.
- Third Test checks if the user has overdue books.
- Fourth Test handles the case when the book is unavailable.
- Fifth Test handles restricted book borrowing before the available date.
- These tests ensure every line and condition in the code is executed at least once.
*/
```



```javascript
// Test cases - Path Testing
console.log(canBorrowBook(user2, book1, currentDate)); // Path 1: User is inactive
console.log(canBorrowBook(user3, book1, currentDate)); // Path 2: User has overdue books
console.log(canBorrowBook(user1, book2, currentDate)); // Path 3: Book is unavailable
console.log(canBorrowBook(user1, book3, currentDate)); // Path 4: Book is restricted
console.log(canBorrowBook(user1, book1, currentDate)); // Path 5: Book can be borrowed

/* Paths:
1. User is inactive (user.active).
2. User has overdue books (user.overdueBooks > 0).
3. Book has no available copies (book.availableCopies < 1).
4. Book is restricted and the current date is before the available date (book.restricted && currentDat
5. User can borrow the book (all conditions pass).
*/

// Test cases - Control Flow Testing
console.log(canBorrowBook(user1, book1, currentDate)); // Book can be borrowed (normal flow)
console.log(canBorrowBook(user1, book3, new Date("2023-11-01"))); // Book can be borrowed (date after

/* The first test case ensures that the normal flow of conditions is handled correctly (when all condi
   The second test ensures that the system correctly handles a book becoming available after its restr
*/

// Test cases - Data Flow Testing
console.log(canBorrowBook(user1, book1, currentDate)); // Book can be borrowed (data passed correctly)
console.log(canBorrowBook(user1, book2, currentDate)); // Book not available (book availability handle

/* The first test checks normal data flow where the user can borrow the book.
   The second test checks how the book.availableCopies affects the data flow, ensuring that unavailabl
*/
```

```
if (book.restricted && currentDate < book.availableFrom) {

return "Book is restricted, cannot borrow now.";

}
```

The issue of the code is this line which I given in upper section, The issue arises because the code assumes that every restricted book has a defined availableFrom date. If a restricted book doesn’t have an availableFrom date

(e.g., book.availableFrom is undefined), comparing currentDate < book.availableFrom will result in a runtime error or unexpected behavior.

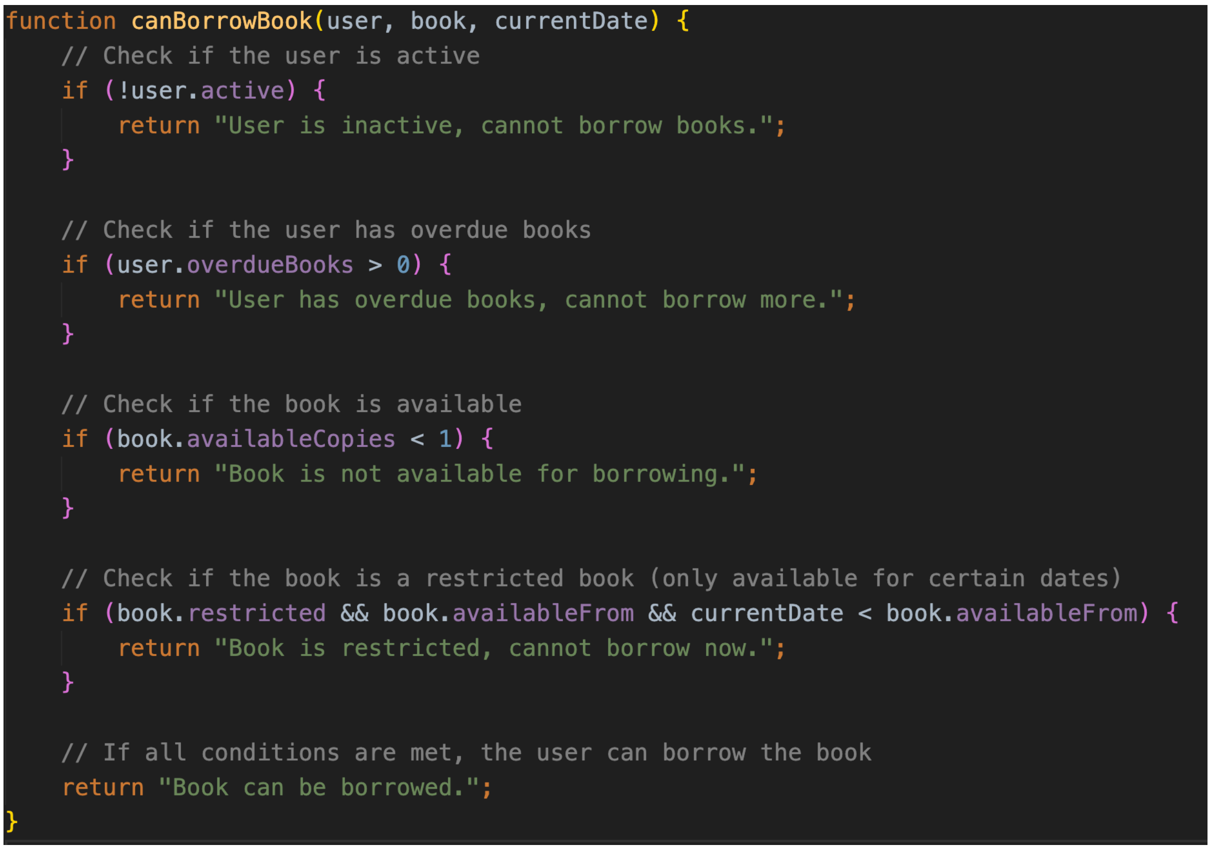
To prevent this, add an additional check to ensure book.availableFrom exists before comparing dates. So I am putting correct line,

```
if (book.restricted && book.availableFrom && currentDate < book.availableFrom) {

   return "Book is restricted, cannot borrow now.";

}
```

So adding book.availableFrom in the condition, I ensure that the code only performs the date comparison if availableFrom is defined. This avoids runtime errors when attempting to borrow a restricted book without a specified available date.



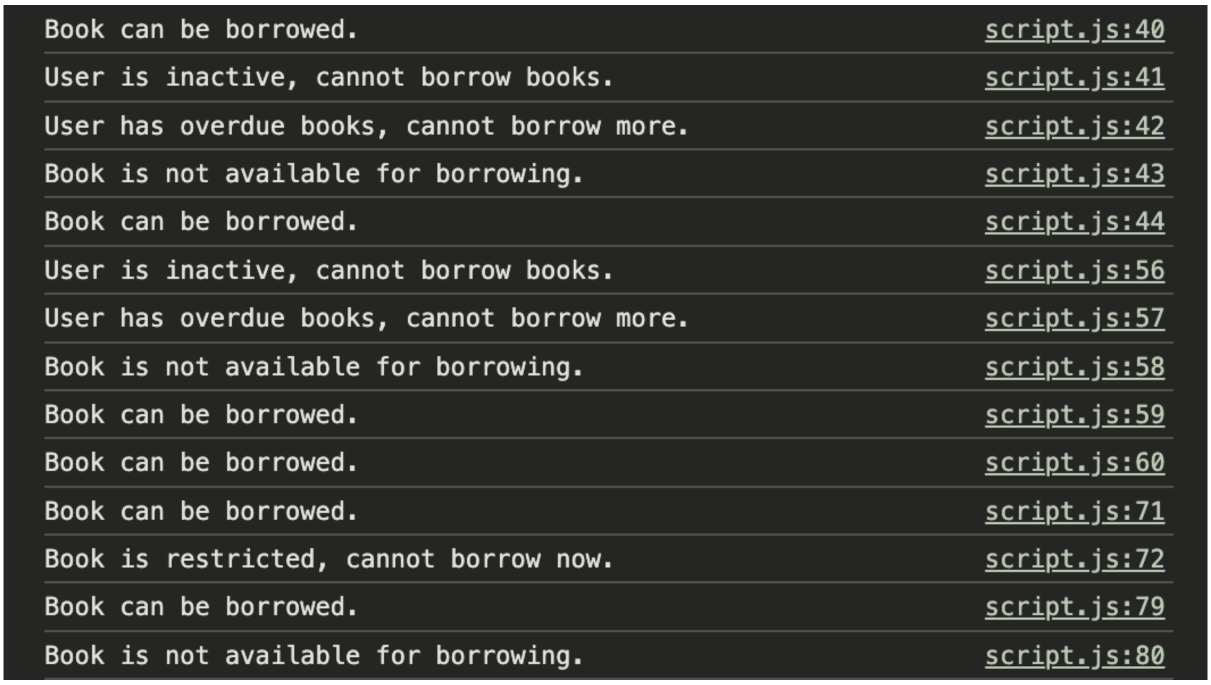
```javascript
function canBorrowBook(user, book, currentDate) {
    // Check if the user is active
    if (!user.active) {
        return "User is inactive, cannot borrow books.";
    }

    // Check if the user has overdue books
    if (user.overdueBooks > 0) {
        return "User has overdue books, cannot borrow more.";
    }

    // Check if the book is available
    if (book.availableCopies < 1) {
        return "Book is not available for borrowing.";
    }

    // Check if the book is a restricted book (only available for certain dates)
    if (book.restricted && book.availableFrom && currentDate < book.availableFrom) {
        return "Book is restricted, cannot borrow now.";
    }

    // If all conditions are met, the user can borrow the book
    return "Book can be borrowed.";
}
```



```
Book can be borrowed.                              script.js:40
User is inactive, cannot borrow books.             script.js:41
User has overdue books, cannot borrow more.        script.js:42
Book is not available for borrowing.               script.js:43
Book can be borrowed.                              script.js:44
User is inactive, cannot borrow books.             script.js:56
User has overdue books, cannot borrow more.        script.js:57
Book is not available for borrowing.               script.js:58
Book can be borrowed.                              script.js:59
Book can be borrowed.                              script.js:60
Book can be borrowed.                              script.js:71
Book is restricted, cannot borrow now.             script.js:72
Book can be borrowed.                              script.js:79
Book is not available for borrowing.               script.js:80
```