Code of Library Management System

Let's write the code for the designed classes in different languages.

We'll cover the following

We've gone over the different aspects of the library management system and observed the attributes attached to the problem using various UML diagrams. Let us now explore the more practical side of things, where we will work on implementing the library management system using multiple languages. This is usually the last step in an object-oriented design interview process.

We have chosen the following languages to write the skeleton code of the different classes present in the library management system:

- Java
- C#
- Python
- C++
- JavaScript

Library management

In this section, we will provide the skeleton code of the classes designed in the class diagram lesson.

> **Note:** For simplicity, we are not defining getter and setter functions. The reader can assume that all class attributes are private and accessed through their respective public getter methods and modified only through their public method functions.

Enumerations

First, we will define all the enumerations required in the library management system. According to the class diagram, there are four enumerations used in the system: `BookFormat`, `BookStatus`, `ReservationStatus`, and `AccountStatus`. The code to implement these enumerations is as follows:`

> **Note:** JavaScript does not support enumerations, so we will be using the `Object.freeze()` method as an alternative that freezes an object and prevents further modifications.

```
const BookFormat = Object.freeze({
    HARDCOVER,
    PAPERBACK,
    AUDIOBOOK,
    EBOOK,
    NEWSPAPER,
    MAGAZINE,
    JOURNAL

});

const BookStatus  = Object.freeze({
    AVAILABLE,
    RESERVED,
    LOANED,
    LOST

});

const ReservationStatus = Object.freeze({
    WAITING,
    PENDING,
    CANCELED,
    NONE

});

const AccountStatus = Object.freeze({
    ACTIVE,
    CLOSED,
    CANCELED
```

Enum definitions

Address and person

This section contains the code for `Address` and `Person` classes where the `Person` class is composed of an `Address` class. The implementation of these classes can be found below:

```
class Address {
    #streetAddress;
    #city;
    #state;
    #zipCode;
    #country;
}

class Person {
```

```
    #name;
    #address;
    #email;
    #phone;
}
```

The Address and Person classes

User

The `User` is an abstract class that represents the various people or actors that can interact with the system. Since there are two types of users, the librarian and the library member, the user can either be a `Librarian` or a `Member`. The implementation of the mentioned classes is shown below:

```
class User {
    #id;
    #password;
    #status;
    #person;
    #card

    constructor(id, password, status, person, card) {
        if (this.constructor == User) {
            throw new Error("Abstract classes can't be instantiated.");
        }
    }

    resetPassword();
}

class Librarian extends User {
    constructor(id, password, status, person, card) {
        this.#id = id;
        this.#password = password;
        this.#status = status;
        this.#person = person;
        this.#card = card;
    }

    addBookItem(bookItem);
    blockMember(member);
    unBlockMember(member);
    resetPassword() {};
}
```

User and its child classes

Book reservation, book lending and fine

This component shows the implementation of `BookReservation`, `BookLending`, and `Fine classes`. These classes will be responsible for managing reservations against books, managing reservations, and calculating fine on books. The code is shown below:

```
class BookReservation {
```

```
    #itemId;
    #creationDate;
    #status;
    #memberId;

    constructor(itemId, creationDate, status, memberId) {
        this.#itemId = itemId;
        this.#creationDate = creationDate;
        this.#status = status;
        this.#memberId = memberId;
    }
    fetchReservationDetails(bookItemId);
}

class BookLending {
    #itemId;
    #creationDate;
    #dueDate;
    #returnDate;
    #memberId;

    constructor(itemId, creationDate, dueDate, returnDate, memberId) {
        this.#itemId = itemId;
        this.#creationDate = creationDate;
        this.#dueDate = dueDate;
        this.#returnDate = returnDate;
        this.#memberId = memberId;
    }

    lendBook(bookItemId, memberId);
```

The BookReservation, BookLending and Fine classes

Book and rack

The `Book` is an abstract class and `BookItem` represents each copy of the book. For example, if there are two copies of the same book then there would only be one `Book` object and two `BookItem` objects. The code to implement these classes is as follows:

```
class Book {
    #isbn;
    #title;
    #subject;
    #publisher;
    #language;
    #numberOfPages;
    #bookFormat;
    #authors;
    constructor(isbn, title, subject, publisher, language, numberOfPages, bookFormat) {
        this.#isbn = isbn;
        this.#title = title;
        this.#subject = subject;
        this.#publisher = publisher;
        this.#language = language;
        this.#numberOfPages = numberOfPages;
        this.#bookFormat = bookFormat;
        this.#authors = new Array();
    }
}

class BookItem extends Book {
    #barcode;
    #isReferenceOnly;
    #borrowed;
```

```
    #borrowed;
    #dueDate;
    #price;
    #status;
    #dateOfPurchase;
    #publicationDate;
    #placedAt;
```

The Book, BookItem and Rack classes

Notification

The `Notification` class is another abstract class responsible for sending notifications to the users, with the `PostalNotification` and `EmailNotification` classes as its child classes. The implementation of this class can be found below:

```
class Notification {
    #notificationId;
    #creatioDate;
    #content;
    constructor() {
        if (this.constructor == Notification) {
            throw new Error("Abstract classes can't be instantiated.");
        }
    }

    sendNotification();
}

class PostalNotification extends Notification {
    #address;

    constructor(notificationId, creationDate, content, address) {
        this.#notificationId = notificationId;
        this.#creationDate = creationDate;
        this.#content = content;
        this.#address = address;
    }
}

class EmailNotification extends Notification {
    #email;

    constructor(notificationId, creationDate, content, email) {
        this.#notificationId = notificationId;
        this.#creationDate = creationDate;
        this.#content = content;
```

Notification and its derived classes

Search and catalog

The `Search` is an interface used in the efficient searching of library books by various methods, and the `Catalog` class is used to implement the search interface to help in book searching. The code to perform this functionality is presented below:

```
class Search {
    searchByTitle(title);
    searchByAuthor(author);
    searchBySubject(subject);
    searchByPublicationDate(publishDate);
```

```
}

class Catalog extends Search {
    #bookTitles;
    #bookAuthors;
    #bookSubjects;
    #bookPublicationDates;

    constructor(){
        this.bookTitles = new Map();
        this.bookAuthors = new Map();
        this.bookSubjects = new Map();
        this.bookPublicationDates = new Map();
    }
    searchByTitle(title);
    searchByAuthor(author);
    searchBySubject(subject);
    searchByPublicationDate(publishDate);
}
```

The Search interface and the Catalog class

Library

The final class of LMS is the `Library` class which will be a Singleton class, meaning the entire system will have only one instance of this class. The implementation of this class can be found below:

```
class Library {
    #name;
    #address;

    getAddress();

    constructor(name, address) {
        this.name = name;
        this.address = address;

        // The Library is a singleton class that ensures it will have only one active instance at a time
        var library = null;
    }

    // Created a static method to access the singleton instance of Library
    getInstance() {
        if (library == null) {
            library = new Library;
        }
        return library;
    }
}
```

The Library class

Wrapping up

We've explored the complete design of a library management system in this chapter. We've looked at how a basic library management system can be visualized using various UML diagrams and designed using object-oriented principles and design patterns.

Back

Activity Diagram for th...

Next

Getting Ready: Amazo...

Mark as Completed