**What is an exception in java?**

An exception in Java is an **event** that disrupts the normal flow of the program's instructions.

It is an **object** that is thrown at runtime when an error occurs.

Exceptions can be caused by a variety of reasons, such as user input errors, hardware failures, or resource exhaustion.

**Types of Exceptions**

1. **Checked Exceptions**:
   * These are exceptions that are checked at compile-time.
   * If a method might **throw** a checked exception, it must either handle it with **a try-catch** block or declare it with a **throws** clause.
   * Examples include IOException, SQLException, and ClassNotFoundException.
2. **Unchecked Exceptions**:
   * These are exceptions that are not checked at compile-time but at runtime.

They include:

* + **Runtime Exceptions**:
    1. These are subclasses of RuntimeException.
    2. They represent programming errors, such as logic errors or improper use of an API.
    3. Examples include NullPointerException, ArrayIndexOutOfBoundsException, and ArithmeticException.
  + **Errors**:
    1. These are subclasses of Error.
    2. They represent serious problems that an application should not try to catch.
    3. Examples include OutOfMemoryError and StackOverflowError.

**Exception Handling Mechanism**

Java provides a robust mechanism to handle exceptions, which includes the following keywords:

* **try**: The block of code where exceptions are expected.
* **catch**: The block of code that handles the exception.
* **finally**: A block that executes regardless of whether an exception is thrown or not, usually for cleanup code.
* **throw**: Used to explicitly throw an exception.
* **throws**: Used in method signatures to declare that a method might throw exceptions.

**Example**

public class ExceptionExample {

public static void main(String[] args) {

try {

int divideByZero = 5 / 0; // This will throw ArithmeticException

}

catch (ArithmeticException e) {

System.out.println("ArithmeticException caught: " + e.getMessage());

}

finally {

System.out.println("This is the finally block.");

}

}

}

In this example, the code inside the try block throws an ArithmeticException because of division by zero.

The catch block catches the exception and handles it by printing an error message.

The finally block executes regardless of whether an exception was thrown or caught.

**Common Exceptions**

Some common exceptions in Java include:

* NullPointerException
* ArrayIndexOutOfBoundsException
* FileNotFoundException
* IOException
* SQLException
* ClassNotFoundException

Handling exceptions properly is crucial for building robust and error-resistant applications.

**Exception classes and example code**

**1. NullPointerException**

Occurs when an application attempts to use an object reference that has the null value.

**Example 1:**

public class NullPointerExample1 {

public static void main(String[] args) {

String str = null;

try {

System.out.println(str.length());

} catch (NullPointerException e) {

System.out.println("Caught NullPointerException: " + e.getMessage());

}

}

}

**Example 2:**

public class NullPointerExample2 {

public static void main(String[] args) {

String[] array = null;

try {

System.out.println(array[0]);

} catch (NullPointerException e) {

System.out.println("Caught NullPointerException: " + e.getMessage());

}

}

}

**2. ArrayIndexOutOfBoundsException**

Occurs when an application attempts to access an array with an index that is out of bounds.

**Example 1:**

public class ArrayIndexOutOfBoundsExample1 {

public static void main(String[] args) {

int[] array = {1, 2, 3};

try {

System.out.println(array[3]);

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Caught ArrayIndexOutOfBoundsException: " + e.getMessage());

}

}

}

**Example 2:**

public class ArrayIndexOutOfBoundsExample2 {

public static void main(String[] args) {

String str = "Hello";

try {

System.out.println(str.charAt(5));

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Caught ArrayIndexOutOfBoundsException: " + e.getMessage());

}

}

}

**3. ArithmeticException**

Occurs when an exceptional arithmetic condition has occurred. For example, an integer "divide by zero" throws an instance of this class.

**Example 1:**

public class ArithmeticExample1 {

public static void main(String[] args) {

try {

int result = 10 / 0;

} catch (ArithmeticException e) {

System.out.println("Caught ArithmeticException: " + e.getMessage());

}

}

}

**Example 2:**

public class ArithmeticExample2 {

public static void main(String[] args) {

try {

int result = Integer.parseInt("abc");

} catch (NumberFormatException e) {

System.out.println("Caught NumberFormatException: " + e.getMessage());

}

}

}

**4. ClassCastException**

Occurs when an object is cast to a subclass that it is not an instance of.

**Example 1:**

public class ClassCastExample1 {

public static void main(String[] args) {

try {

Object obj = new Integer(100);

String str = (String) obj; // This will throw ClassCastException

} catch (ClassCastException e) {

System.out.println("Caught ClassCastException: " + e.getMessage());

}

}

}

**Example 2:**

public class ClassCastExample2 {

public static void main(String[] args) {

try {

Object obj = new String("Hello");

Integer num = (Integer) obj; // This will throw ClassCastException

} catch (ClassCastException e) {

System.out.println("Caught ClassCastException: " + e.getMessage());

}

}

}

**5. IllegalArgumentException**

Occurs when a method has been passed an illegal or inappropriate argument.

**Example 1:**

public class IllegalArgumentExample1 {

public static void main(String[] args) {

try {

Thread.sleep(-1000); // This will throw IllegalArgumentException

} catch (IllegalArgumentException | InterruptedException e) {

System.out.println("Caught IllegalArgumentException: " + e.getMessage());

}

}

}

**Example 2:**

public class IllegalArgumentExample2 {

public static void main(String[] args) {

try {

Integer.parseInt("abc"); // This will throw IllegalArgumentException

} catch (NumberFormatException e) {

System.out.println("Caught NumberFormatException: " + e.getMessage());

}

}

}

These examples cover various common exceptions encountered in Java, focusing on scenarios not involving file I/O or database operations.

**6. IllegalStateException**

Occurs when a method has been invoked at an illegal or inappropriate time.

**Example 1:**

import java.util.Iterator;

import java.util.ArrayList;

import java.util.List;

public class IllegalStateExample1 {

public static void main(String[] args) {

List<String> list = new ArrayList<>();

list.add("A");

list.add("B");

Iterator<String> iterator = list.iterator();

try {

iterator.remove(); // This will throw IllegalStateException

} catch (IllegalStateException e) {

System.out.println("Caught IllegalStateException: " + e.getMessage());

}

}

}

**Example 2:**

import java.util.Scanner;

public class IllegalStateExample2 {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

scanner.close();

try {

scanner.next(); // This will throw IllegalStateException

} catch (IllegalStateException e) {

System.out.println("Caught IllegalStateException: " + e.getMessage());

}

}

}

**7. IndexOutOfBoundsException**

Occurs when trying to access an index that is out of range.

**Example 1:**

import java.util.ArrayList;

import java.util.List;

public class IndexOutOfBoundsExample1 {

public static void main(String[] args) {

List<String> list = new ArrayList<>();

list.add("A");

try {

System.out.println(list.get(1)); // This will throw IndexOutOfBoundsException

} catch (IndexOutOfBoundsException e) {

System.out.println("Caught IndexOutOfBoundsException: " + e.getMessage());

}

}

}

**Example 2:**

public class IndexOutOfBoundsExample2 {

public static void main(String[] args) {

String str = "Hello";

try {

char ch = str.charAt(5); // This will throw IndexOutOfBoundsException

} catch (IndexOutOfBoundsException e) {

System.out.println("Caught IndexOutOfBoundsException: " + e.getMessage());

}

}

}

**8. NumberFormatException**

Occurs when an attempt is made to convert a string to a numeric type, but the string does not have the appropriate format.

**Example 1:**

public class NumberFormatExample1 {

public static void main(String[] args) {

try {

int num = Integer.parseInt("abc"); // This will throw NumberFormatException

} catch (NumberFormatException e) {

System.out.println("Caught NumberFormatException: " + e.getMessage());

}

}

}

**Example 2:**

public class NumberFormatExample2 {

public static void main(String[] args) {

try {

double d = Double.parseDouble("12.34.56"); // This will throw NumberFormatException

} catch (NumberFormatException e) {

System.out.println("Caught NumberFormatException: " + e.getMessage());

}

}

}

**9. StringIndexOutOfBoundsException**

Occurs when trying to access an invalid index in a string.

**Example 1:**

public class StringIndexOutOfBoundsExample1 {

public static void main(String[] args) {

String str = "Hello";

try {

char ch = str.charAt(10); // This will throw StringIndexOutOfBoundsException

} catch (StringIndexOutOfBoundsException e) {

System.out.println("Caught StringIndexOutOfBoundsException: " + e.getMessage());

}

}

}

**Example 2:**

public class StringIndexOutOfBoundsExample2 {

public static void main(String[] args) {

String str = "Hello";

try {

String substr = str.substring(2, 10); // This will throw StringIndexOutOfBoundsException

} catch (StringIndexOutOfBoundsException e) {

System.out.println("Caught StringIndexOutOfBoundsException: " + e.getMessage());

}

}

}

**10. UnsupportedOperationException**

Occurs when the requested operation is not supported.

**Example 1:**

import java.util.Collections;

import java.util.List;

import java.util.ArrayList;

public class UnsupportedOperationExample1 {

public static void main(String[] args) {

List<String> list = new ArrayList<>();

list.add("A");

list.add("B");

List<String> unmodifiableList = Collections.unmodifiableList(list);

try {

unmodifiableList.add("C"); // This will throw UnsupportedOperationException

} catch (UnsupportedOperationException e) {

System.out.println("Caught UnsupportedOperationException: " + e.getMessage());

}

}

}

**Example 2:**

import java.util.Arrays;

import java.util.List;

public class UnsupportedOperationExample2 {

public static void main(String[] args) {

List<String> list = Arrays.asList("A", "B");

try {

list.add("C"); // This will throw UnsupportedOperationException

} catch (UnsupportedOperationException e) {

System.out.println("Caught UnsupportedOperationException: " + e.getMessage());

}

}

}

These examples cover various common exceptions encountered in Java, focusing on different scenarios to illustrate when and why these exceptions might be thrown.

**What do you mean by user defined exceptions?**

**User-defined** exceptions in Java, also known as **custom** exceptions, are exceptions created by the programmer to handle specific error conditions that are not covered by Java's built-in exceptions.

Custom exceptions can make your code more readable and provide better error-handling capabilities tailored to the specific needs of your application.

**Creating a User-Defined Exception**

To create a user-defined exception, you need to extend the **Exception** class (or any of its subclasses) or **RuntimeException** class.

Here is a step-by-step guide to creating and using a custom exception.

**Steps to Create a User-Defined Exception**

1. **Define the Exception Class**: Create a new class that extends **Exception** (for checked exceptions) or **RuntimeException** (for unchecked exceptions).
2. **Constructor**: Define one or more constructors for your exception class. Typically, you will define at least one constructor that accepts a message string and optionally other constructors for additional data or cause chaining.

**Example: Creating a User-Defined Exception**

**Step 1: Define the Exception Class**

public class InvalidAgeException extends Exception {

public InvalidAgeException(String message) {

super(message);

}

}

**Step 2: Use the Custom Exception**

public class **UserDefinedExceptionExample** {

public static void main(String[] args) {

try {

checkAge(15);

} catch (InvalidAgeException e) {

System.out.println("Caught Exception: " + e.getMessage());

}

}

static void checkAge(int age) throws InvalidAgeException {

if (age < 18) {

throw new InvalidAgeException("Age must be at least 18.");

} else {

System.out.println("Age is valid.");

}

}

}

**In this example:**

* **InvalidAgeException** is a custom exception that extends the Exception class.
* The **checkAge** method throws an **InvalidAgeException** if the age is less than 18.
* The main method calls **checkAge** and handles the **InvalidAgeException** if it is thrown.

**Example 2: Creating and Using a RuntimeException**

**Step 1: Define the Exception Class**

public class **NegativeNumberException** extends RuntimeException {

public NegativeNumberException(String message) {

super(message);

}

}

**Step 2: Use the Custom Exception**

public class **UserDefinedRuntimeExceptionExample** {

public static void main(String[] args) {

try {

squareRoot(-9);

} catch (NegativeNumberException e) {

System.out.println("Caught Exception: " + e.getMessage());

}

}

static void squareRoot(int number) {

if (number < 0) {

throw new NegativeNumberException("Number must be non-negative.");

} else {

System.out.println("Square root is: " + Math.sqrt(number));

}

}

}

**In this example:**

* **NegativeNumberException** is a custom exception that extends **RuntimeException**.
* The **squareRoot** method throws a **NegativeNumberException** if the number is negative.
* The main method calls **squareRoot** and handles the **NegativeNumberException** if it is thrown.

**Benefits of User-Defined Exceptions**

* **Clarity**: Custom exceptions can provide more meaningful error messages specific to your application logic.
* **Modularity**: Encapsulating specific error conditions into custom exceptions keeps the code modular and easier to maintain.
* **Control**: Custom exceptions allow you to manage error conditions more precisely, including complex error handling logic that might not be adequately covered by standard exceptions.

Using user-defined exceptions is a good practice in Java to ensure that error conditions are handled in a way that is clear, specific, and maintainable.

**Example 01:**

**User-Defined Exception**

Now, let's create a custom exception called InsufficientFundsException and use it in a banking application:

**Custom Exception Class**

public class **InsufficientFundsException** extends Exception {

public InsufficientFundsException(String message) {

super(message);

}

}

**BankAccount Class**

public class **BankAccount** {

private double balance;

public BankAccount(double balance) {

this.balance = balance;

}

public void deposit(double amount) {

balance += amount;

System.out.println("Deposited: " + amount + ", New Balance: " + balance);

}

public void withdraw(double amount) throws InsufficientFundsException {

if (amount > balance) {

throw new InsufficientFundsException("Insufficient funds for withdrawal: " + amount);

}

balance -= amount;

System.out.println("Withdrew: " + amount + ", New Balance: " + balance);

}

public double getBalance() {

return balance;

}

}

**Main Class**

public class **BankingApplication** {

public static void main(String[] args) {

BankAccount account = new BankAccount(1000.00);

try {

account.deposit(500.00);

account.withdraw(200.00);

account.withdraw(1500.00); // This will throw InsufficientFundsException

}

catch (InsufficientFundsException e) {

System.out.println("Exception caught: " + e.getMessage());

}

System.out.println("Final Balance: " + account.getBalance());

}

}

**Example 02**

Let's create a more comprehensive example that combines both standard and user-defined exceptions.

We'll develop a more detailed banking application that includes operations such as deposit, withdrawal, and transfer of funds between accounts.

**Custom Exceptions**

We'll define two custom exceptions: InsufficientFundsException and InvalidAccountException.

**InsufficientFundsException**

public class **InsufficientFundsException** extends Exception {

public InsufficientFundsException(String message) {

super(message);

}

}

**InvalidAccountException**

public class **InvalidAccountException** extends Exception {

public InvalidAccountException(String message) {

super(message);

}

}

**BankAccount Class**

This class will handle basic operations and will use custom exceptions where appropriate.

public class **BankAccount** {

private String accountNumber;

private double balance;

public BankAccount(String accountNumber, double balance) {

this.accountNumber = accountNumber;

this.balance = balance;

}

public String getAccountNumber() {

return accountNumber;

}

public double getBalance() {

return balance;

}

public void deposit(double amount) {

balance += amount;

System.out.println("Deposited: " + amount + ", New Balance: " + balance);

}

public void withdraw(double amount) throws InsufficientFundsException {

if (amount > balance) {

throw new InsufficientFundsException("Insufficient funds for withdrawal: " + amount);

}

balance -= amount;

System.out.println("Withdrew: " + amount + ", New Balance: " + balance);

}

public void transfer(BankAccount targetAccount, double amount) throws InsufficientFundsException, InvalidAccountException {

if (targetAccount == null) {

throw new InvalidAccountException("Target account is invalid.");

}

this.withdraw(amount);

targetAccount.deposit(amount);

System.out.println("Transferred: " + amount + " from " + this.accountNumber + " to " + targetAccount.getAccountNumber());

}

}

**Main Application**

The main application will demonstrate various scenarios, including handling standard exceptions and custom exceptions.

public class **BankingApplication** {

public static void main(String[] args) {

// Create accounts

BankAccount account1 = new BankAccount("123456", 1000.00);

BankAccount account2 = new BankAccount("654321", 500.00);

try {

// Deposit money into account1

account1.deposit(200.00);

// Withdraw money from account1

account1.withdraw(150.00);

// Attempt to withdraw more than the balance in account2

account2.withdraw(1000.00); // This will throw InsufficientFundsException

}

catch (InsufficientFundsException e) {

System.out.println("Exception caught: " + e.getMessage());

}

try {

// Transfer money from account1 to account2

account1.transfer(account2, 500.00);

// Attempt to transfer money to a null account (invalid)

account1.transfer(null, 100.00); // This will throw InvalidAccountException

}

catch (InsufficientFundsException e) {

System.out.println("Exception caught: " + e.getMessage());

}

catch (InvalidAccountException e) {

System.out.println("Exception caught: " + e.getMessage());

}

// Final balances

System.out.println("Final Balance of account1: " + account1.getBalance());

System.out.println("Final Balance of account2: " + account2.getBalance());

}

}

**Example 03:**

let's create a comprehensive Library Management System that demonstrates the use of exceptions, both standard and user-defined.

**User-Defined Exceptions**

**BookNotFoundException**

public class **BookNotFoundException** extends Exception {

public BookNotFoundException(String message) {

super(message);

}

}

**MemberNotFoundException**

public class **MemberNotFoundException** extends Exception {

public MemberNotFoundException(String message) {

super(message);

}

}

**Book Class**

public class **Book** {

private String title;

private String author;

private String isbn;

private boolean isAvailable;

public Book(String title, String author, String isbn) {

this.title = title;

this.author = author;

this.isbn = isbn;

this.isAvailable = true;

}

public String getTitle() {

return title;

}

public String getAuthor() {

return author;

}

public String getIsbn() {

return isbn;

}

public boolean isAvailable() {

return isAvailable;

}

public void setAvailable(boolean available) {

isAvailable = available;

}

}

**Member Class**

import java.util.ArrayList;

import java.util.List;

public class **Member** {

private String name;

private String memberId;

private List<Book> borrowedBooks;

public Member(String name, String memberId) {

this.name = name;

this.memberId = memberId;

this.borrowedBooks = new ArrayList<>();

}

public String getName() {

return name;

}

public String getMemberId() {

return memberId;

}

public List<Book> getBorrowedBooks() {

return borrowedBooks;

}

public void borrowBook(Book book) {

borrowedBooks.add(book);

}

public void returnBook(Book book) {

borrowedBooks.remove(book);

}

}

**Library Class**

import java.util.ArrayList;

import java.util.List;

public class **Library** {

private List<Book> books;

private List<Member> members;

public Library() {

books = new ArrayList<>();

members = new ArrayList<>();

}

public void addBook(Book book) {

books.add(book);

}

public void addMember(Member member) {

members.add(member);

}

public Book findBookByIsbn(String isbn) throws BookNotFoundException {

for (Book book : books) {

if (book.getIsbn().equals(isbn)) {

return book;

}

}

throw new BookNotFoundException("Book with ISBN " + isbn + " not found.");

}

public Member findMemberById(String memberId) throws MemberNotFoundException {

for (Member member : members) {

if (member.getMemberId().equals(memberId)) {

return member;

}

}

throw new MemberNotFoundException("Member with ID " + memberId + " not found.");

}

public void borrowBook(String isbn, String memberId) throws BookNotFoundException, MemberNotFoundException {

Book book = findBookByIsbn(isbn);

Member member = findMemberById(memberId);

if (book.isAvailable()) {

book.setAvailable(false);

member.borrowBook(book);

System.out.println(member.getName() + " borrowed " + book.getTitle());

}

else {

System.out.println("Book is currently unavailable.");

}

}

public void returnBook(String isbn, String memberId) throws BookNotFoundException, MemberNotFoundException {

Book book = findBookByIsbn(isbn);

Member member = findMemberById(memberId);

if (member.getBorrowedBooks().contains(book)) {

book.setAvailable(true);

member.returnBook(book);

System.out.println(member.getName() + " returned " + book.getTitle());

}

else {

System.out.println(member.getName() + " did not borrow " + book.getTitle());

}

}

}

**Main Application**

public class **LibraryManagementSystem** {

public static void main(String[] args) {

Library library = new Library();

// Add books to the library

Book book1 = new Book("Effective Java", "Joshua Bloch", "9780134685991");

Book book2 = new Book("Clean Code", "Robert C. Martin", "9780132350884");

library.addBook(book1);

library.addBook(book2);

// Add members to the library

Member member1 = new Member("Alice", "M001");

Member member2 = new Member("Bob", "M002");

library.addMember(member1);

library.addMember(member2);

try {

// Member borrows a book

library.borrowBook("9780134685991", "M001");

// Member returns a book

library.returnBook("9780134685991", "M001");

// Attempt to borrow a book that does not exist

library.borrowBook("1234567890", "M002"); // This will throw BookNotFoundException

}

catch (BookNotFoundException e) {

System.out.println("Exception caught: " + e.getMessage());

}

catch (MemberNotFoundException e) {

System.out.println("Exception caught: " + e.getMessage());

}

try {

// Attempt to find a member that does not exist

library.findMemberById("M003"); // This will throw MemberNotFoundException

}

catch (MemberNotFoundException e) {

System.out.println("Exception caught: " + e.getMessage());

}

// Demonstrate standard exceptions within library logic

try {

book1 = null;

book1.getTitle(); // This will throw NullPointerException

}

catch (NullPointerException e) {

System.out.println("Null pointer exception: " + e.getMessage());

}

// Final borrowed books list

System.out.println("Borrowed books by " + member1.getName() + ": " + member1.getBorrowedBooks().size());

System.out.println("Borrowed books by " + member2.getName() + ": " + member2.getBorrowedBooks().size());

}

}

This example demonstrates a Library Management System that handles both standard exceptions and user-defined exceptions in various scenarios.

It shows how to handle book borrowing and returning, finding books and members, and handling exceptional situations such as missing books or members.

**Example 04:**

Let's expand the Library Management System example to include additional features like managing a catalog of books, tracking overdue books, and handling fine calculations for overdue books.

We'll also introduce more user-defined exceptions and incorporate more comprehensive exception handling.

**Custom Exceptions**

**BookNotFoundException**

public class **BookNotFoundException** extends Exception {

public BookNotFoundException(String message) {

super(message);

}

}

**MemberNotFoundException**

public class **MemberNotFoundException** extends Exception {

public MemberNotFoundException(String message) {

super(message);

}

}

**OverdueBookException**

public class **OverdueBookException** extends Exception {

public OverdueBookException(String message) {

super(message);

}

}

**Book Class**

import java.time.LocalDate;

public class **Book** {

private String title;

private String author;

private String isbn;

private boolean isAvailable;

private LocalDate dueDate;

public Book(String title, String author, String isbn) {

this.title = title;

this.author = author;

this.isbn = isbn;

this.isAvailable = true;

this.dueDate = null;

}

public String getTitle() {

return title;

}

public String getAuthor() {

return author;

}

public String getIsbn() {

return isbn;

}

public boolean isAvailable() {

return isAvailable;

}

public void setAvailable(boolean available) {

isAvailable = available;

}

public LocalDate getDueDate() {

return dueDate;

}

public void setDueDate(LocalDate dueDate) {

this.dueDate = dueDate;

}

}

**Member Class**

import java.util.ArrayList;

import java.util.List;

public class **Member** {

private String name;

private String memberId;

private List<Book> borrowedBooks;

public Member(String name, String memberId) {

this.name = name;

this.memberId = memberId;

this.borrowedBooks = new ArrayList<>();

}

public String getName() {

return name;

}

public String getMemberId() {

return memberId;

}

public List<Book> getBorrowedBooks() {

return borrowedBooks;

}

public void borrowBook(Book book) {

borrowedBooks.add(book);

}

public void returnBook(Book book) {

borrowedBooks.remove(book);

}

}

**Library Class**

import java.time.LocalDate;

import java.util.ArrayList;

import java.util.List;

public class **Library** {

private List<Book> books;

private List<Member> members;

public Library() {

books = new ArrayList<>();

members = new ArrayList<>();

}

public void addBook(Book book) {

books.add(book);

}

public void addMember(Member member) {

members.add(member);

}

public Book findBookByIsbn(String isbn) throws BookNotFoundException {

for (Book book : books) {

if (book.getIsbn().equals(isbn)) {

return book;

}

}

throw new BookNotFoundException("Book with ISBN " + isbn + " not found.");

}

public Member findMemberById(String memberId) throws MemberNotFoundException {

for (Member member : members) {

if (member.getMemberId().equals(memberId)) {

return member;

}

}

throw new MemberNotFoundException("Member with ID " + memberId + " not found.");

}

public void borrowBook(String isbn, String memberId) throws BookNotFoundException, MemberNotFoundException, OverdueBookException {

Book book = findBookByIsbn(isbn);

Member member = findMemberById(memberId);

if (!book.isAvailable()) {

if (book.getDueDate().isBefore(LocalDate.now())) {

throw new OverdueBookException("Book is overdue. Cannot borrow until it is returned.");

}

else {

System.out.println("Book is currently unavailable.");

return;

}

}

book.setAvailable(false);

book.setDueDate(LocalDate.now().plusWeeks(2));

member.borrowBook(book);

System.out.println(member.getName() + " borrowed " + book.getTitle() + ". Due date: " + book.getDueDate());

}

public void returnBook(String isbn, String memberId) throws BookNotFoundException, MemberNotFoundException, OverdueBookException {

Book book = findBookByIsbn(isbn);

Member member = findMemberById(memberId);

if (!member.getBorrowedBooks().contains(book)) {

System.out.println(member.getName() + " did not borrow " + book.getTitle());

return;

}

if (book.getDueDate().isBefore(LocalDate.now())) {

long overdueDays = LocalDate.now().toEpochDay() - book.getDueDate().toEpochDay();

double fine = overdueDays \* 1.5; // Assuming fine is $1.5 per day

System.out.println("Book is overdue by " + overdueDays + " days. Fine: $" + fine);

}

book.setAvailable(true);

book.setDueDate(null);

member.returnBook(book);

System.out.println(member.getName() + " returned " + book.getTitle());

}

public void listAvailableBooks() {

for (Book book : books) {

if (book.isAvailable()) {

System.out.println("Title: " + book.getTitle() + ", Author: " + book.getAuthor() + ", ISBN: " + book.getIsbn());

}

}

}

public void listBorrowedBooks() {

for (Book book : books) {

if (!book.isAvailable()) {

System.out.println("Title: " + book.getTitle() + ", Author: " + book.getAuthor() + ", ISBN: " + book.getIsbn() + ", Due Date: " + book.getDueDate());

}

}

}

}

**Main Application**

public class **LibraryManagementSystem** {

public static void main(String[] args) {

Library library = new Library();

// Add books to the library

Book book1 = new Book("Effective Java", "Joshua Bloch", "9780134685991");

Book book2 = new Book("Clean Code", "Robert C. Martin", "9780132350884");

Book book3 = new Book("Design Patterns", "Erich Gamma", "9780201633610");

library.addBook(book1);

library.addBook(book2);

library.addBook(book3);

// Add members to the library

Member member1 = new Member("Alice", "M001");

Member member2 = new Member("Bob", "M002");

library.addMember(member1);

library.addMember(member2);

try {

// Member borrows a book

library.borrowBook("9780134685991", "M001");

// Member returns a book

library.returnBook("9780134685991", "M001");

// Attempt to borrow a book that does not exist

library.borrowBook("1234567890", "M002"); // This will throw BookNotFoundException

} catch (BookNotFoundException e) {

System.out.println("Exception caught: " + e.getMessage());

} catch (MemberNotFoundException e) {

System.out.println("Exception caught: " + e.getMessage());

} catch (OverdueBookException e) {

System.out.println("Exception caught: " + e.getMessage());

}

try {

// Attempt to find a member that does not exist

library.findMemberById("M003"); // This will throw MemberNotFoundException

} catch (MemberNotFoundException e) {

System.out.println("Exception caught: " + e.getMessage());

}

// Demonstrate standard exceptions within library logic

try {

book1 = null;

book1.getTitle(); // This will throw NullPointerException

} catch (NullPointerException e) {

System.out.println("Null pointer exception: " + e.getMessage());

}

// List available books

System.out.println("\nAvailable books:");

library.listAvailableBooks();

// List borrowed books

System.out.println("\nBorrowed books:");

library.listBorrowedBooks();

// Final borrowed books list

System.out.println("\nBorrowed books by " + member1.getName() + ": " + member1.getBorrowedBooks().size());

System.out.println("Borrowed books by " + member2.getName() + ": " + member2.getBorrowedBooks().size());

}

}

**Example 05**

Let's create a comprehensive example for a Shopping Cart System that demonstrates the use of both standard and user-defined exceptions.

**Custom Exceptions**

**ProductNotFoundException**

public class ProductNotFoundException extends Exception {

public ProductNotFoundException(String message) {

super(message);

}

}

**InsufficientStockException**

public class InsufficientStockException extends Exception {

public InsufficientStockException(String message) {

super(message);

}

}

**UserNotFoundException**

public class UserNotFoundException extends Exception {

public UserNotFoundException(String message) {

super(message);

}

}

**Product Class**

public class Product {

private String productId;

private String name;

private double price;

private int stock;

public Product(String productId, String name, double price, int stock) {

this.productId = productId;

this.name = name;

this.price = price;

this.stock = stock;

}

public String getProductId() {

return productId;

}

public String getName() {

return name;

}

public double getPrice() {

return price;

}

public int getStock() {

return stock;

}

public void setStock(int stock) {

this.stock = stock;

}

}

**User Class**

import java.util.ArrayList;

import java.util.List;

public class User {

private String userId;

private String name;

private List<Product> cart;

public User(String userId, String name) {

this.userId = userId;

this.name = name;

this.cart = new ArrayList<>();

}

public String getUserId() {

return userId;

}

public String getName() {

return name;

}

public List<Product> getCart() {

return cart;

}

public void addToCart(Product product) {

cart.add(product);

}

public void removeFromCart(Product product) {

cart.remove(product);

}

}

**ShoppingCart Class**

import java.util.ArrayList;

import java.util.List;

public class ShoppingCart {

private List<Product> products;

private List<User> users;

public ShoppingCart() {

products = new ArrayList<>();

users = new ArrayList<>();

}

public void addProduct(Product product) {

products.add(product);

}

public void addUser(User user) {

users.add(user);

}

public Product findProductById(String productId) throws ProductNotFoundException {

for (Product product : products) {

if (product.getProductId().equals(productId)) {

return product;

}

}

throw new ProductNotFoundException("Product with ID " + productId + " not found.");

}

public User findUserById(String userId) throws UserNotFoundException {

for (User user : users) {

if (user.getUserId().equals(userId)) {

return user;

}

}

throw new UserNotFoundException("User with ID " + userId + " not found.");

}

public void addToCart(String userId, String productId, int quantity) throws ProductNotFoundException, UserNotFoundException, InsufficientStockException {

Product product = findProductById(productId);

User user = findUserById(userId);

if (product.getStock() < quantity) {

throw new InsufficientStockException("Insufficient stock for product: " + product.getName());

}

product.setStock(product.getStock() - quantity);

for (int i = 0; i < quantity; i++) {

user.addToCart(product);

}

System.out.println(user.getName() + " added " + quantity + " " + product.getName() + "(s) to their cart.");

}

public void removeFromCart(String userId, String productId) throws ProductNotFoundException, UserNotFoundException {

Product product = findProductById(productId);

User user = findUserById(userId);

if (!user.getCart().contains(product)) {

System.out.println(user.getName() + " does not have " + product.getName() + " in their cart.");

return;

}

user.removeFromCart(product);

product.setStock(product.getStock() + 1);

System.out.println(user.getName() + " removed " + product.getName() + " from their cart.");

}

public void checkout(String userId) throws UserNotFoundException {

User user = findUserById(userId);

double totalAmount = 0;

for (Product product : user.getCart()) {

totalAmount += product.getPrice();

}

user.getCart().clear();

System.out.println(user.getName() + " checked out. Total amount: $" + totalAmount);

}

public void listAvailableProducts() {

for (Product product : products) {

System.out.println("ID: " + product.getProductId() + ", Name: " + product.getName() + ", Price: " + product.getPrice() + ", Stock: " + product.getStock());

}

}

public void listUserCart(String userId) throws UserNotFoundException {

User user = findUserById(userId);

System.out.println(user.getName() + "'s cart:");

for (Product product : user.getCart()) {

System.out.println("Name: " + product.getName() + ", Price: " + product.getPrice());

}

}

}

**Main Application**

public class ShoppingCartSystem {

public static void main(String[] args) {

ShoppingCart shoppingCart = new ShoppingCart();

// Add products to the system

Product product1 = new Product("P001", "Laptop", 1000.00, 10);

Product product2 = new Product("P002", "Smartphone", 500.00, 20);

Product product3 = new Product("P003", "Headphones", 150.00, 15);

shoppingCart.addProduct(product1);

shoppingCart.addProduct(product2);

shoppingCart.addProduct(product3);

// Add users to the system

User user1 = new User("U001", "Alice");

User user2 = new User("U002", "Bob");

shoppingCart.addUser(user1);

shoppingCart.addUser(user2);

try {

// User adds products to their cart

shoppingCart.addToCart("U001", "P001", 1);

shoppingCart.addToCart("U001", "P002", 2);

shoppingCart.addToCart("U002", "P003", 3);

// User tries to add more products than available stock

shoppingCart.addToCart("U001", "P001", 10); // This will throw InsufficientStockException

} catch (ProductNotFoundException e) {

System.out.println("Exception caught: " + e.getMessage());

} catch (UserNotFoundException e) {

System.out.println("Exception caught: " + e.getMessage());

} catch (InsufficientStockException e) {

System.out.println("Exception caught: " + e.getMessage());

}

try {

// User removes product from their cart

shoppingCart.removeFromCart("U001", "P002");

// Attempt to remove a product not in the cart

shoppingCart.removeFromCart("U001", "P003");

} catch (ProductNotFoundException e) {

System.out.println("Exception caught: " + e.getMessage());

} catch (UserNotFoundException e) {

System.out.println("Exception caught: " + e.getMessage());

}

try {

// User checks out

shoppingCart.checkout("U001");

// List products available in the system

System.out.println("\nAvailable products:");

shoppingCart.listAvailableProducts();

// List products in a user's cart

System.out.println("\nAlice's cart:");

shoppingCart.listUserCart("U001"); // This will be empty as Alice checked out

System.out.println("\nBob's cart:");

shoppingCart.listUserCart("U002");

} catch (UserNotFoundException e) {

System.out.println("Exception caught: " + e.getMessage());

}

// Demonstrate standard exceptions within the system logic

try {

product1 = null;

product1.getName(); // This will throw NullPointerException

} catch (NullPointerException e) {

System.out.println("Null pointer exception: " + e.getMessage());

}

}

}

In this Shopping Cart System example, we have added the following features:

1. **Product Management:** Adding products to the system and checking their stock.
2. **User Management:** Adding users to the system.
3. **Cart Management:** Adding and removing products from the user's cart.
4. **Checkout:** Checking out the cart and calculating the total amount.
5. **Listing Products and Cart:** Listing available products and products in the user's cart.
6. **Exception Handling:** Handling user-defined exceptions (ProductNotFoundException, UserNotFoundException, InsufficientStockException) and standard exceptions (NullPointerException).

**Example 05**

Let's create a more complex example involving a **Restaurant Management System**. This example will utilize 10 classes, demonstrate the use of both standard and user-defined exceptions, and cover a variety of real-world scenarios.

**Overview of the System**

In this example, the Restaurant Management System includes features like managing a menu, handling reservations, processing orders, and managing staff. We will also include handling exceptions related to reservations, orders, and menu items.

**Custom Exceptions**

**ReservationNotFoundException**

public class ReservationNotFoundException extends Exception {

public ReservationNotFoundException(String message) {

super(message);

}

}

**TableNotAvailableException**

public class TableNotAvailableException extends Exception {

public TableNotAvailableException(String message) {

super(message);

}

}

**MenuItemNotFoundException**

public class MenuItemNotFoundException extends Exception {

public MenuItemNotFoundException(String message) {

super(message);

}

}

**InsufficientStockException**

public class InsufficientStockException extends Exception {

public InsufficientStockException(String message) {

super(message);

}

}

**InvalidOrderException**

public class InvalidOrderException extends Exception {

public InvalidOrderException(String message) {

super(message);

}

}

**1. MenuItem Class**

public class MenuItem {

private String id;

private String name;

private double price;

private int stock;

public MenuItem(String id, String name, double price, int stock) {

this.id = id;

this.name = name;

this.price = price;

this.stock = stock;

}

public String getId() {

return id;

}

public String getName() {

return name;

}

public double getPrice() {

return price;

}

public int getStock() {

return stock;

}

public void setStock(int stock) {

this.stock = stock;

}

}

**2. Order Class**

import java.util.ArrayList;

import java.util.List;

public class Order {

private String orderId;

private String tableId;

private List<MenuItem> items;

private double totalAmount;

public Order(String orderId, String tableId) {

this.orderId = orderId;

this.tableId = tableId;

this.items = new ArrayList<>();

this.totalAmount = 0;

}

public String getOrderId() {

return orderId;

}

public String getTableId() {

return tableId;

}

public List<MenuItem> getItems() {

return items;

}

public void addItem(MenuItem item, int quantity) throws InsufficientStockException {

if (item.getStock() < quantity) {

throw new InsufficientStockException("Not enough stock for " + item.getName());

}

item.setStock(item.getStock() - quantity);

for (int i = 0; i < quantity; i++) {

items.add(item);

}

totalAmount += item.getPrice() \* quantity;

}

public double getTotalAmount() {

return totalAmount;

}

}

**3. Table Class**

public class Table {

private String tableId;

private int seats;

private boolean isAvailable;

public Table(String tableId, int seats) {

this.tableId = tableId;

this.seats = seats;

this.isAvailable = true;

}

public String getTableId() {

return tableId;

}

public int getSeats() {

return seats;

}

public boolean isAvailable() {

return isAvailable;

}

public void setAvailable(boolean available) {

isAvailable = available;

}

}

**4. Reservation Class**

public class Reservation {

private String reservationId;

private String tableId;

private String customerName;

private int numberOfGuests;

private String reservationTime;

public Reservation(String reservationId, String tableId, String customerName, int numberOfGuests, String reservationTime) {

this.reservationId = reservationId;

this.tableId = tableId;

this.customerName = customerName;

this.numberOfGuests = numberOfGuests;

this.reservationTime = reservationTime;

}

public String getReservationId() {

return reservationId;

}

public String getTableId() {

return tableId;

}

public String getCustomerName() {

return customerName;

}

public int getNumberOfGuests() {

return numberOfGuests;

}

public String getReservationTime() {

return reservationTime;

}

}

**5. Staff Class**

public class Staff {

private String staffId;

private String name;

private String role;

public Staff(String staffId, String name, String role) {

this.staffId = staffId;

this.name = name;

this.role = role;

}

public String getStaffId() {

return staffId;

}

public String getName() {

return name;

}

public String getRole() {

return role;

}

}

**6. Menu Class**

import java.util.ArrayList;

import java.util.List;

public class Menu {

private List<MenuItem> menuItems;

public Menu() {

this.menuItems = new ArrayList<>();

}

public void addMenuItem(MenuItem item) {

menuItems.add(item);

}

public MenuItem findMenuItemById(String id) throws MenuItemNotFoundException {

for (MenuItem item : menuItems) {

if (item.getId().equals(id)) {

return item;

}

}

throw new MenuItemNotFoundException("Menu item with ID " + id + " not found.");

}

public void listMenuItems() {

for (MenuItem item : menuItems) {

System.out.println("ID: " + item.getId() + ", Name: " + item.getName() + ", Price: $" + item.getPrice() + ", Stock: " + item.getStock());

}

}

}

**7. ReservationManager Class**

import java.util.ArrayList;

import java.util.List;

public class ReservationManager {

private List<Reservation> reservations;

public ReservationManager() {

reservations = new ArrayList<>();

}

public void makeReservation(Reservation reservation) throws TableNotAvailableException {

for (Reservation res : reservations) {

if (res.getTableId().equals(reservation.getTableId()) && res.getReservationTime().equals(reservation.getReservationTime())) {

throw new TableNotAvailableException("Table " + reservation.getTableId() + " is already reserved at this time.");

}

}

reservations.add(reservation);

System.out.println("Reservation made for " + reservation.getCustomerName() + " at table " + reservation.getTableId());

}

public Reservation findReservationById(String reservationId) throws ReservationNotFoundException {

for (Reservation res : reservations) {

if (res.getReservationId().equals(reservationId)) {

return res;

}

}

throw new ReservationNotFoundException("Reservation with ID " + reservationId + " not found.");

}

}

**8. OrderManager Class**

import java.util.ArrayList;

import java.util.List;

public class OrderManager {

private List<Order> orders;

public OrderManager() {

orders = new ArrayList<>();

}

public void placeOrder(Order order) {

orders.add(order);

System.out.println("Order placed for table " + order.getTableId() + ". Total amount: $" + order.getTotalAmount());

}

public Order findOrderById(String orderId) throws InvalidOrderException {

for (Order order : orders) {

if (order.getOrderId().equals(orderId)) {

return order;

}

}

throw new InvalidOrderException("Order with ID " + orderId + " is invalid or does not exist.");

}

public void listOrders() {

for (Order order : orders) {

System.out.println("Order ID: " + order.getOrderId() + ", Table ID: " + order.getTableId() + ", Total Amount: $" + order.getTotalAmount());

}

}

}

**9. Restaurant Class**

import java.util.ArrayList;

import java.util.List;

public class Restaurant {

private List<Table> tables;

private Menu menu;

private ReservationManager reservationManager;

private OrderManager orderManager;

private List<Staff> staffList;

public Restaurant() {

tables = new ArrayList<>();

menu = new Menu();

reservationManager = new ReservationManager();

orderManager = new OrderManager();

staffList = new ArrayList<>();

}

public void addTable(Table table) {

tables.add(table);

}

public void addMenuItem(MenuItem item) {

menu.addMenuItem(item);

}

public void addStaff(Staff staff) {

staffList.add(staff);

}

public void makeReservation(String reservationId, String tableId, String customerName, int numberOfGuests, String reservationTime) throws ReservationNotFoundException, TableNotAvailableException {

Table table = findTableById(tableId);

if (!table.isAvailable()) {

throw new TableNotAvailableException("Table " + tableId + " is not available.");

}

Reservation reservation = new Reservation(reservationId, tableId, customerName, numberOfGuests, reservationTime);

reservationManager.makeReservation(reservation);

table.setAvailable(false); // Mark table as reserved

}

public void placeOrder(String orderId, String tableId, List<String> menuItemIds) throws MenuItemNotFoundException, InsufficientStockException, InvalidOrderException {

Order order = new Order(orderId, tableId);

for (String id : menuItemIds) {

MenuItem item = menu.findMenuItemById(id);

order.addItem(item, 1); // Assuming quantity is 1 for simplicity

}

orderManager.placeOrder(order);

}

public void checkout(String orderId) throws InvalidOrderException {

Order order = orderManager.findOrderById(orderId);

System.out.println("Order ID " + orderId + " checked out. Total Amount: $" + order.getTotalAmount());

}

public Table findTableById(String tableId) {

for (Table table : tables) {

if (table.getTableId().equals(tableId)) {

return table;

}

}

return null;

}

public void listAvailableTables() {

for (Table table : tables) {

if (table.isAvailable()) {

System.out.println("Table ID: " + table.getTableId() + ", Seats: " + table.getSeats());

}

}

}

public void listMenuItems() {

menu.listMenuItems();

}

public void listOrders() {

orderManager.listOrders();

}

public void listReservations() {

for (Reservation reservation : reservationManager.reservations) {

System.out.println("Reservation ID: " + reservation.getReservationId() + ", Table ID: " + reservation.getTableId() + ", Customer Name: " + reservation.getCustomerName() + ", Time: " + reservation.getReservationTime());

}

}

}

**10. Main Class**

import java.util.Arrays;

public class RestaurantManagementSystem {

public static void main(String[] args) {

Restaurant restaurant = new Restaurant();

// Add tables

restaurant.addTable(new Table("T1", 4));

restaurant.addTable(new Table("T2", 2));

restaurant.addTable(new Table("T3", 6));

// Add menu items

restaurant.addMenuItem(new MenuItem("M1", "Burger", 5.99, 50));

restaurant.addMenuItem(new MenuItem("M2", "Pizza", 8.99, 30));

restaurant.addMenuItem(new MenuItem("M3", "Pasta", 7.49, 20));

// Add staff members

restaurant.addStaff(new Staff("S1", "John", "Waiter"));

restaurant.addStaff(new Staff("S2", "Sarah", "Chef"));

try {

// Make a reservation

restaurant.makeReservation("R1", "T1", "Alice", 4, "2024-07-17 19:00");

// Place an order

restaurant.placeOrder("O1", "T1", Arrays.asList("M1", "M2"));

// Attempt to place an order with insufficient stock

restaurant.placeOrder("O2", "T1", Arrays.asList("M2", "M3", "M3", "M3", "M3", "M3", "M3")); // This will throw InsufficientStockException

} catch (TableNotAvailableException | MenuItemNotFoundException | InsufficientStockException | InvalidOrderException e) {

System.out.println("Exception caught: " + e.getMessage());

}

try {

// Check out an order

restaurant.checkout("O1");

// List available tables

System.out.println("\nAvailable tables:");

restaurant.listAvailableTables();

// List menu items

System.out.println("\nMenu items:");

restaurant.listMenuItems();

// List orders

System.out.println("\nOrders:");

restaurant.listOrders();

// List reservations

System.out.println("\nReservations:");

restaurant.listReservations();

} catch (InvalidOrderException e) {

System.out.println("Exception caught: " + e.getMessage());

}

}

}

**Summary of the System**

* **Classes:**
  + MenuItem: Represents items on the menu.
  + Order: Represents an order made at a table.
  + Table: Represents a table in the restaurant.
  + Reservation: Represents a reservation made for a table.
  + Staff: Represents a staff member.
  + Menu: Manages menu items.
  + ReservationManager: Manages reservations.
  + OrderManager: Manages orders.
  + Restaurant: Main class for managing the restaurant operations.
  + RestaurantManagementSystem: Entry point for the application.
* **User-Defined Exceptions:**
  + ReservationNotFoundException: For handling cases where a reservation is not found.
  + TableNotAvailableException: For handling table availability issues.
  + MenuItemNotFoundException: For handling cases where a menu item is not found.
  + InsufficientStockException: For handling insufficient stock of menu items.
  + InvalidOrderException: For handling invalid or non-existent orders.
* **Standard Exceptions:**
  + Handled during the try-catch block for operations like placing an order and making reservations.
* **Features:**
  + **Table Management:** Adding tables and listing available tables.
  + **Menu Management:** Adding menu items and listing them.
  + **Reservation Management:** Making and listing reservations.
  + **Order Management:** Placing orders, checking out, and listing orders.
  + **Staff Management:** Adding staff members.
  + **Exception Handling:** Managing both user-defined and standard exceptions.

This example covers a broad range of functionalities and exception handling scenarios, providing a robust foundation for a Restaurant Management System.

**Additional Ideas for Expanding**

If you want to further expand this system, consider adding:

* **Payment Processing:** Implement a class for processing payments.
* **Customer Feedback:** Add a feature for customers to leave feedback or reviews.
* **Loyalty Programs:** Implement a loyalty program for frequent customers.
* **Special Offers:** Add special offers or discounts for certain menu items.

This comprehensive example should provide a good starting point for understanding complex application structures and exception handling in Java.

**Example 06:**

After adding the following features for the example 05

* **Payment Processing:** Implement a class for processing payments.
* **Customer Feedback:** Add a feature for customers to leave feedback or reviews.
* **Loyalty Programs:** Implement a loyalty program for frequent customers.
* **Special Offers:** Add special offers or discounts for certain menu items.

Let’s expand the Restaurant Management System to include additional features like **Payment Processing**, **Customer Feedback**, **Loyalty Programs**, and **Special Offers**.

This will involve adding several new classes and extending existing ones.

Here’s a comprehensive implementation that includes all these features:

**1. PaymentProcessor Class**

This class handles the payment processing for orders.

public class PaymentProcessor {

public void processPayment(double amount) {

// Simulate payment processing

System.out.println("Processing payment of $" + amount);

}

}

**2. CustomerFeedback Class**

This class manages customer feedback for the restaurant.

public class CustomerFeedback {

private String feedbackId;

private String customerName;

private String feedback;

public CustomerFeedback(String feedbackId, String customerName, String feedback) {

this.feedbackId = feedbackId;

this.customerName = customerName;

this.feedback = feedback;

}

public String getFeedbackId() {

return feedbackId;

}

public String getCustomerName() {

return customerName;

}

public String getFeedback() {

return feedback;

}

@Override

public String toString() {

return "Feedback from " + customerName + ": " + feedback;

}

}

**3. LoyaltyProgram Class**

This class manages the loyalty points for customers.

import java.util.HashMap;

import java.util.Map;

public class LoyaltyProgram {

private Map<String, Integer> customerPoints;

public LoyaltyProgram() {

customerPoints = new HashMap<>();

}

public void addPoints(String customerId, int points) {

customerPoints.put(customerId, customerPoints.getOrDefault(customerId, 0) + points);

System.out.println("Added " + points + " points for customer ID " + customerId);

}

public int getPoints(String customerId) {

return customerPoints.getOrDefault(customerId, 0);

}

}

**4. SpecialOffer Class**

This class manages special offers or discounts for menu items.

public class SpecialOffer {

private String offerId;

private String description;

private double discountPercentage;

public SpecialOffer(String offerId, String description, double discountPercentage) {

this.offerId = offerId;

this.description = description;

this.discountPercentage = discountPercentage;

}

public String getOfferId() {

return offerId;

}

public String getDescription() {

return description;

}

public double getDiscountPercentage() {

return discountPercentage;

}

public double applyDiscount(double price) {

return price - (price \* discountPercentage / 100);

}

}

**5. Updated MenuItem Class**

We will add the ability to apply special offers to menu items.

public class MenuItem {

private String id;

private String name;

private double price;

private int stock;

private SpecialOffer specialOffer;

public MenuItem(String id, String name, double price, int stock) {

this.id = id;

this.name = name;

this.price = price;

this.stock = stock;

this.specialOffer = null; // No special offer by default

}

public String getId() {

return id;

}

public String getName() {

return name;

}

public double getPrice() {

return price;

}

public int getStock() {

return stock;

}

public void setStock(int stock) {

this.stock = stock;

}

public void setSpecialOffer(SpecialOffer offer) {

this.specialOffer = offer;

}

public SpecialOffer getSpecialOffer() {

return specialOffer;

}

public double getEffectivePrice() {

if (specialOffer != null) {

return specialOffer.applyDiscount(price);

}

return price;

}

}

**6. Updated Order Class**

We will adjust this class to handle special offers and payment processing.

import java.util.ArrayList;

import java.util.List;

public class Order {

private String orderId;

private String tableId;

private List<MenuItem> items;

private double totalAmount;

public Order(String orderId, String tableId) {

this.orderId = orderId;

this.tableId = tableId;

this.items = new ArrayList<>();

this.totalAmount = 0;

}

public String getOrderId() {

return orderId;

}

public String getTableId() {

return tableId;

}

public List<MenuItem> getItems() {

return items;

}

public void addItem(MenuItem item, int quantity) throws InsufficientStockException {

if (item.getStock() < quantity) {

throw new InsufficientStockException("Not enough stock for " + item.getName());

}

item.setStock(item.getStock() - quantity);

for (int i = 0; i < quantity; i++) {

items.add(item);

}

totalAmount += item.getEffectivePrice() \* quantity;

}

public double getTotalAmount() {

return totalAmount;

}

public void applyDiscount(double discount) {

totalAmount -= (totalAmount \* discount / 100);

}

}

**7. Updated Restaurant Class**

We will update this class to integrate the new features: processing payments, handling feedback, managing loyalty programs, and special offers.

import java.util.ArrayList;

import java.util.List;

public class Restaurant {

private List<Table> tables;

private Menu menu;

private ReservationManager reservationManager;

private OrderManager orderManager;

private List<Staff> staffList;

private LoyaltyProgram loyaltyProgram;

private List<CustomerFeedback> feedbackList;

private List<SpecialOffer> specialOffers;

private PaymentProcessor paymentProcessor;

public Restaurant() {

tables = new ArrayList<>();

menu = new Menu();

reservationManager = new ReservationManager();

orderManager = new OrderManager();

staffList = new ArrayList<>();

loyaltyProgram = new LoyaltyProgram();

feedbackList = new ArrayList<>();

specialOffers = new ArrayList<>();

paymentProcessor = new PaymentProcessor();

}

public void addTable(Table table) {

tables.add(table);

}

public void addMenuItem(MenuItem item) {

menu.addMenuItem(item);

}

public void addStaff(Staff staff) {

staffList.add(staff);

}

public void addSpecialOffer(SpecialOffer offer) {

specialOffers.add(offer);

for (MenuItem item : menu.menuItems) {

if (item.getId().equals(offer.getOfferId())) {

item.setSpecialOffer(offer);

}

}

}

public void makeReservation(String reservationId, String tableId, String customerName, int numberOfGuests, String reservationTime) throws ReservationNotFoundException, TableNotAvailableException {

Table table = findTableById(tableId);

if (table == null) {

throw new ReservationNotFoundException("Table " + tableId + " does not exist.");

}

if (!table.isAvailable()) {

throw new TableNotAvailableException("Table " + tableId + " is not available.");

}

Reservation reservation = new Reservation(reservationId, tableId, customerName, numberOfGuests, reservationTime);

reservationManager.makeReservation(reservation);

table.setAvailable(false); // Mark table as reserved

}

public void placeOrder(String orderId, String tableId, List<String> menuItemIds) throws MenuItemNotFoundException, InsufficientStockException, InvalidOrderException {

Order order = new Order(orderId, tableId);

for (String id : menuItemIds) {

MenuItem item = menu.findMenuItemById(id);

order.addItem(item, 1); // Assuming quantity is 1 for simplicity

}

orderManager.placeOrder(order);

// Apply special offers

for (SpecialOffer offer : specialOffers) {

if (offer.getOfferId().equals(menuItemIds.get(0))) {

order.applyDiscount(offer.getDiscountPercentage());

}

}

// Process the payment

paymentProcessor.processPayment(order.getTotalAmount());

}

public void checkout(String orderId) throws InvalidOrderException {

Order order = orderManager.findOrderById(orderId);

paymentProcessor.processPayment(order.getTotalAmount());

System.out.println("Order ID " + orderId + " checked out. Total Amount: $" + order.getTotalAmount());

}

public void leaveFeedback(String feedbackId, String customerName, String feedback) {

CustomerFeedback customerFeedback = new CustomerFeedback(feedbackId, customerName, feedback);

feedbackList.add(customerFeedback);

System.out.println("Feedback received from " + customerName + ": " + feedback);

}

public void addLoyaltyPoints(String customerId, int points) {

loyaltyProgram.addPoints(customerId, points);

}

public void listAvailableTables() {

for (Table table : tables) {

if (table.isAvailable()) {

System.out.println("Table ID: " + table.getTableId() + ", Seats: " + table.getSeats());

}

}

}

public void listMenuItems() {

menu.listMenuItems();

}

public void listOrders() {

orderManager.listOrders();

}

public void listReservations() {

for (Reservation reservation : reservationManager.reservations) {

System.out.println("Reservation ID: " + reservation.getReservationId() + ", Table ID: " + reservation.getTableId() + ", Customer Name: " + reservation.getCustomerName() + ", Time: " + reservation.getReservationTime());

}

}

public void listSpecialOffers() {

for (SpecialOffer offer : specialOffers) {

System.out.println("Offer ID: " + offer.getOfferId() + ", Description: " + offer.getDescription() + ", Discount: " + offer.getDiscountPercentage() + "%");

}

}

public void listFeedback() {

for (CustomerFeedback feedback : feedbackList) {

System.out.println(feedback);

}

}

public void listLoyaltyPoints(String customerId) {

int points = loyaltyProgram.getPoints(customerId);

System.out.println("Loyalty points for customer ID " + customerId + ": " + points);

}

private Table findTableById(String tableId) {

for (Table table : tables) {

if (table.getTableId().equals(tableId)) {

return table;

}

}

return null;

}

}

**8. Updated Main Class**

We will demonstrate the new features in the main class.

import java.util.Arrays;

public class RestaurantManagementSystem {

public static void main(String[] args) {

Restaurant restaurant = new Restaurant();

// Add tables

restaurant.addTable(new Table("T1", 4));

restaurant.addTable(new Table("T2", 2));

restaurant.addTable(new Table("T3", 6));

// Add menu items

restaurant.addMenuItem(new MenuItem("M1", "Burger", 5.99, 50));

restaurant.addMenuItem(new MenuItem("M2", "Pizza", 8.99, 30));

restaurant.addMenuItem(new MenuItem("M3", "Pasta", 7.49, 20));

// Add special offers

restaurant.addSpecialOffer(new SpecialOffer("M1", "Burger Deal", 10)); // 10% off for burgers

// Add staff members

restaurant.addStaff(new Staff("S1", "John", "Waiter"));

restaurant.addStaff(new Staff("S2", "Sarah", "Chef"));

try {

// Make a reservation

restaurant.makeReservation("R1", "T1", "Alice", 4, "2024-07-17 19:00");

// Place an order

restaurant.placeOrder("O1", "T1", Arrays.asList("M1", "M2"));

// Leave feedback

restaurant.leaveFeedback("F1", "Alice", "Great food and service!");

// Add loyalty points

restaurant.addLoyaltyPoints("C1", 100);

// List available tables

System.out.println("\nAvailable tables:");

restaurant.listAvailableTables();

// List menu items

System.out.println("\nMenu items:");

restaurant.listMenuItems();

// List special offers

System.out.println("\nSpecial offers:");

restaurant.listSpecialOffers();

// List orders

System.out.println("\nOrders:");

restaurant.listOrders();

// List reservations

System.out.println("\nReservations:");

restaurant.listReservations();

// List feedback

System.out.println("\nCustomer Feedback:");

restaurant.listFeedback();

// List loyalty points

System.out.println("\nLoyalty Points:");

restaurant.listLoyaltyPoints("C1");

// Checkout an order

restaurant.checkout("O1");

} catch (TableNotAvailableException | MenuItemNotFoundException | InsufficientStockException | InvalidOrderException | ReservationNotFoundException e) {

System.out.println("Exception caught: " + e.getMessage());

}

}

}

**Summary of Additions**

* **PaymentProcessor Class:** Manages the payment process for orders.
* **CustomerFeedback Class:** Manages customer feedback.
* **LoyaltyProgram Class:** Manages customer loyalty points.
* **SpecialOffer Class:** Manages special offers or discounts for menu items.
* **Updated MenuItem Class:** Now supports special offers.
* **Updated Order Class:** Can apply discounts and has a method for processing payments.
* **Updated Restaurant Class:** Includes methods for processing payments, handling feedback, managing loyalty points, and listing special offers.
* **Updated Main Class:** Demonstrates the use of new features.

This expanded version of the Restaurant Management System provides a richer set of features and a robust framework for understanding complex systems and exception handling in Java.

**Additional Features**

If you want to further expand the system, consider adding:

* **Online Ordering:** Allow customers to place orders online.
* **Table Management:** Add features for managing table layouts and configurations.
* **Advanced Reporting:** Generate detailed reports for sales, reservations, and feedback.
* **Employee Management:** Add features for managing staff schedules and payroll.

Feel free to adapt these ideas to fit your specific needs or use them as inspiration for your own projects!

**Example 07**

Let’s expand the Restaurant Management System with the following features:

1. **Online Ordering**
2. **Table Management**
3. **Advanced Reporting**
4. **Employee Management**

Here’s how you can implement these features with new and updated classes:

**1. Online Ordering Feature**

We'll add a class OnlineOrder to handle orders placed through an online platform.

**OnlineOrder Class**

public class OnlineOrder {

private String orderId;

private String customerId;

private List<MenuItem> items;

private double totalAmount;

public OnlineOrder(String orderId, String customerId) {

this.orderId = orderId;

this.customerId = customerId;

this.items = new ArrayList<>();

this.totalAmount = 0;

}

public String getOrderId() {

return orderId;

}

public String getCustomerId() {

return customerId;

}

public List<MenuItem> getItems() {

return items;

}

public void addItem(MenuItem item, int quantity) throws InsufficientStockException {

if (item.getStock() < quantity) {

throw new InsufficientStockException("Not enough stock for " + item.getName());

}

item.setStock(item.getStock() - quantity);

for (int i = 0; i < quantity; i++) {

items.add(item);

}

totalAmount += item.getEffectivePrice() \* quantity;

}

public double getTotalAmount() {

return totalAmount;

}

public void applyDiscount(double discount) {

totalAmount -= (totalAmount \* discount / 100);

}

}

**Updated Restaurant Class**

Add methods for online ordering in the Restaurant class.

import java.util.ArrayList;

import java.util.List;

public class Restaurant {

private List<Table> tables;

private Menu menu;

private ReservationManager reservationManager;

private OrderManager orderManager;

private List<Staff> staffList;

private LoyaltyProgram loyaltyProgram;

private List<CustomerFeedback> feedbackList;

private List<SpecialOffer> specialOffers;

private PaymentProcessor paymentProcessor;

private List<OnlineOrder> onlineOrders; // New list to manage online orders

public Restaurant() {

tables = new ArrayList<>();

menu = new Menu();

reservationManager = new ReservationManager();

orderManager = new OrderManager();

staffList = new ArrayList<>();

loyaltyProgram = new LoyaltyProgram();

feedbackList = new ArrayList<>();

specialOffers = new ArrayList<>();

paymentProcessor = new PaymentProcessor();

onlineOrders = new ArrayList<>(); // Initialize online orders list

}

// Other existing methods...

public void placeOnlineOrder(String orderId, String customerId, List<String> menuItemIds) throws MenuItemNotFoundException, InsufficientStockException {

OnlineOrder onlineOrder = new OnlineOrder(orderId, customerId);

for (String id : menuItemIds) {

MenuItem item = menu.findMenuItemById(id);

onlineOrder.addItem(item, 1); // Assuming quantity is 1 for simplicity

}

// Apply special offers

for (SpecialOffer offer : specialOffers) {

if (offer.getOfferId().equals(menuItemIds.get(0))) {

onlineOrder.applyDiscount(offer.getDiscountPercentage());

}

}

onlineOrders.add(onlineOrder);

paymentProcessor.processPayment(onlineOrder.getTotalAmount());

System.out.println("Online order placed. Order ID: " + orderId + ", Total Amount: $" + onlineOrder.getTotalAmount());

}

public void listOnlineOrders() {

for (OnlineOrder order : onlineOrders) {

System.out.println("Online Order ID: " + order.getOrderId() + ", Customer ID: " + order.getCustomerId() + ", Total Amount: $" + order.getTotalAmount());

}

}

}

**2. Table Management Feature**

Add functionality for managing table layouts and configurations.

**TableManagement Class**

public class TableManagement {

private List<Table> tables;

public TableManagement(List<Table> tables) {

this.tables = tables;

}

public void addTable(Table table) {

tables.add(table);

System.out.println("Added Table ID: " + table.getTableId() + ", Seats: " + table.getSeats());

}

public void removeTable(String tableId) {

tables.removeIf(table -> table.getTableId().equals(tableId));

System.out.println("Removed Table ID: " + tableId);

}

public void updateTable(String tableId, int newSeats) {

for (Table table : tables) {

if (table.getTableId().equals(tableId)) {

table.setSeats(newSeats);

System.out.println("Updated Table ID: " + tableId + " to Seats: " + newSeats);

}

}

}

public void listTables() {

for (Table table : tables) {

System.out.println("Table ID: " + table.getTableId() + ", Seats: " + table.getSeats() + ", Available: " + table.isAvailable());

}

}

}

**Updated Restaurant Class**

Integrate the TableManagement class into the Restaurant class.

public class Restaurant {

private TableManagement tableManagement; // New TableManagement class instance

private Menu menu;

private ReservationManager reservationManager;

private OrderManager orderManager;

private List<Staff> staffList;

private LoyaltyProgram loyaltyProgram;

private List<CustomerFeedback> feedbackList;

private List<SpecialOffer> specialOffers;

private PaymentProcessor paymentProcessor;

private List<OnlineOrder> onlineOrders; // Manage online orders

public Restaurant() {

List<Table> initialTables = new ArrayList<>();

tableManagement = new TableManagement(initialTables);

menu = new Menu();

reservationManager = new ReservationManager();

orderManager = new OrderManager();

staffList = new ArrayList<>();

loyaltyProgram = new LoyaltyProgram();

feedbackList = new ArrayList<>();

specialOffers = new ArrayList<>();

paymentProcessor = new PaymentProcessor();

onlineOrders = new ArrayList<>(); // Initialize online orders list

}

// Existing methods...

public void addTable(Table table) {

tableManagement.addTable(table);

}

public void removeTable(String tableId) {

tableManagement.removeTable(tableId);

}

public void updateTable(String tableId, int newSeats) {

tableManagement.updateTable(tableId, newSeats);

}

public void listTables() {

tableManagement.listTables();

}

// Online Order methods

public void placeOnlineOrder(String orderId, String customerId, List<String> menuItemIds) throws MenuItemNotFoundException, InsufficientStockException {

OnlineOrder onlineOrder = new OnlineOrder(orderId, customerId);

for (String id : menuItemIds) {

MenuItem item = menu.findMenuItemById(id);

onlineOrder.addItem(item, 1);

}

for (SpecialOffer offer : specialOffers) {

if (offer.getOfferId().equals(menuItemIds.get(0))) {

onlineOrder.applyDiscount(offer.getDiscountPercentage());

}

}

onlineOrders.add(onlineOrder);

paymentProcessor.processPayment(onlineOrder.getTotalAmount());

System.out.println("Online order placed. Order ID: " + orderId + ", Total Amount: $" + onlineOrder.getTotalAmount());

}

public void listOnlineOrders() {

for (OnlineOrder order : onlineOrders) {

System.out.println("Online Order ID: " + order.getOrderId() + ", Customer ID: " + order.getCustomerId() + ", Total Amount: $" + order.getTotalAmount());

}

}

// Other methods...

}

**3. Advanced Reporting Feature**

Add classes for generating reports.

**ReportGenerator Class**

import java.util.List;

public class ReportGenerator {

public static void generateSalesReport(List<OnlineOrder> onlineOrders, List<Order> orders) {

double totalSales = 0;

for (OnlineOrder order : onlineOrders) {

totalSales += order.getTotalAmount();

}

for (Order order : orders) {

totalSales += order.getTotalAmount();

}

System.out.println("Total Sales: $" + totalSales);

}

public static void generateReservationReport(List<Reservation> reservations) {

System.out.println("Reservation Report:");

for (Reservation reservation : reservations) {

System.out.println("Reservation ID: " + reservation.getReservationId() + ", Table ID: " + reservation.getTableId() + ", Customer Name: " + reservation.getCustomerName() + ", Time: " + reservation.getReservationTime());

}

}

public static void generateFeedbackReport(List<CustomerFeedback> feedbackList) {

System.out.println("Feedback Report:");

for (CustomerFeedback feedback : feedbackList) {

System.out.println(feedback);

}

}

}

**Updated Restaurant Class**

Add methods to generate reports.

import java.util.ArrayList;

import java.util.List;

public class Restaurant {

// Existing attributes

private ReportGenerator reportGenerator; // New ReportGenerator class instance

public Restaurant() {

// Initialize classes

List<Table> initialTables = new ArrayList<>();

tableManagement = new TableManagement(initialTables);

menu = new Menu();

reservationManager = new ReservationManager();

orderManager = new OrderManager();

staffList = new ArrayList<>();

loyaltyProgram = new LoyaltyProgram();

feedbackList = new ArrayList<>();

specialOffers = new ArrayList<>();

paymentProcessor = new PaymentProcessor();

onlineOrders = new ArrayList<>();

reportGenerator = new ReportGenerator(); // Initialize ReportGenerator

}

// Existing methods...

public void generateSalesReport() {

reportGenerator.generateSalesReport(onlineOrders, orderManager.getOrders());

}

public void generateReservationReport() {

reportGenerator.generateReservationReport(reservationManager.getReservations());

}

public void generateFeedbackReport() {

reportGenerator.generateFeedbackReport(feedbackList);

}

// Other methods...

}

**4. Employee Management Feature**

Add classes for staff scheduling and payroll.

**EmployeeManagement Class**

import java.util.HashMap;

import java.util.Map;

public class EmployeeManagement {

private Map<String, Staff> staffMap;

private Map<String, String> staffSchedules; // Staff ID to Schedule mapping

private Map<String, Double> staffSalaries; // Staff ID to Salary mapping

public EmployeeManagement() {

staffMap = new HashMap<>();

staffSchedules = new HashMap<>();

staffSalaries = new HashMap<>();

}

public void addStaff(Staff staff) {

staffMap.put(staff.getId(), staff);

System.out.println("Added Staff ID: " + staff.getId() + ", Name: " + staff.getName());

}

public void setSchedule(String staffId, String schedule) {

staffSchedules.put(staffId, schedule);

System.out.println("Set schedule for Staff ID: " + staffId + " to " + schedule);

}

public void setSalary(String staffId, double salary) {

staffSalaries.put(staffId, salary);

System.out.println("Set salary for Staff ID: " + staffId + " to $" + salary);

}

public void listStaff() {

for (Map.Entry<String, Staff> entry : staffMap.entrySet()) {

Staff staff = entry.getValue();

System.out.println("Staff ID: " + staff.getId() + ", Name: " + staff.getName() + ", Role: " + staff.getRole());

}

}

public void listSchedules() {

for (Map.Entry<String, String> entry : staffSchedules.entrySet()) {

System.out.println("Staff ID: " + entry.getKey() + ", Schedule: " + entry.getValue());

}

}

public void listSalaries() {

for (Map.Entry<String, Double> entry : staffSalaries.entrySet()) {

System.out.println("Staff ID: " + entry.getKey() + ", Salary: $" + entry.getValue());

}

}

}

**Updated Restaurant Class**

Add methods for employee management.

public class Restaurant {

private EmployeeManagement employeeManagement; // New EmployeeManagement class instance

public Restaurant() {

// Initialize classes

List<Table> initialTables = new ArrayList<>();

tableManagement = new TableManagement(initialTables);

menu = new Menu();

reservationManager = new ReservationManager();

orderManager = new OrderManager();

staffList = new ArrayList<>();

loyaltyProgram = new LoyaltyProgram();

feedbackList = new ArrayList<>();

specialOffers = new ArrayList<>();

paymentProcessor = new PaymentProcessor();

onlineOrders = new ArrayList<>();

reportGenerator = new ReportGenerator();

employeeManagement = new EmployeeManagement(); // Initialize EmployeeManagement

}

// Existing methods...

public void addStaff(Staff staff) {

employeeManagement.addStaff(staff);

}

public void setSchedule(String staffId, String schedule) {

employeeManagement.setSchedule(staffId, schedule);

}

public void setSalary(String staffId, double salary) {

employeeManagement.setSalary(staffId, salary);

}

public void listStaff() {

employeeManagement.listStaff();

}

public void listSchedules() {

employeeManagement.listSchedules();

}

public void listSalaries() {

employeeManagement.listSalaries();

}

// Other methods...

}

**Complete Example**

Here is a complete example including all classes with added features:

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

public class Main {

public static void main(String[] args) {

Restaurant restaurant = new Restaurant();

// Add tables

restaurant.addTable(new Table("T1", 4));

restaurant.addTable(new Table("T2", 2));

restaurant.addTable(new Table("T3", 6));

// Add menu items

restaurant.addMenuItem(new MenuItem("M1", "Burger", 5.99, 50));

restaurant.addMenuItem(new MenuItem("M2", "Pizza", 8.99, 30));

restaurant.addMenuItem(new MenuItem("M3", "Pasta", 7.49, 20));

// Add special offers

restaurant.addSpecialOffer(new SpecialOffer("M1", "Burger Deal", 10)); // 10% off for burgers

// Add staff members

restaurant.addStaff(new Staff("S1", "John", "Waiter"));

restaurant.addStaff(new Staff("S2", "Sarah", "Chef"));

// Set schedules and salaries

restaurant.setSchedule("S1", "9am - 5pm");

restaurant.setSalary("S1", 15.00);

restaurant.setSchedule("S2", "1pm - 9pm");

restaurant.setSalary("S2", 18.00);

// Add an online order

try {

restaurant.placeOnlineOrder("O2", "C1", List.of("M1", "M2"));

} catch (MenuItemNotFoundException | InsufficientStockException e) {

System.out.println("Exception caught: " + e.getMessage());

}

// List online orders

System.out.println("\nOnline Orders:");

restaurant.listOnlineOrders();

// Make a reservation

try {

restaurant.makeReservation("R1", "T1", "Alice", 4, "2024-07-17 19:00");

// Place an order

restaurant.placeOrder("O1", "T1", List.of("M1", "M2"));

// Leave feedback

restaurant.leaveFeedback("F1", "Alice", "Great food and service!");

// Add loyalty points

restaurant.addLoyaltyPoints("C1", 100);

// Generate reports

restaurant.generateSalesReport();

restaurant.generateReservationReport();

restaurant.generateFeedbackReport();

// List available tables

System.out.println("\nAvailable tables:");

restaurant.listTables();

// List menu items

System.out.println("\nMenu items:");

restaurant.listMenuItems();

// List special offers

System.out.println("\nSpecial offers:");

restaurant.listSpecialOffers();

// List orders

System.out.println("\nOrders:");

restaurant.listOrders();

// List reservations

System.out.println("\nReservations:");

restaurant.listReservations();

// List feedback

System.out.println("\nCustomer Feedback:");

restaurant.listFeedback();

// List loyalty points

System.out.println("\nLoyalty Points:");

restaurant.listLoyaltyPoints("C1");

// Checkout an order

restaurant.checkout("O1");

} catch (TableNotAvailableException | MenuItemNotFoundException | InsufficientStockException | InvalidOrderException | ReservationNotFoundException e) {

System.out.println("Exception caught: " + e.getMessage());

}

// List staff

System.out.println("\nStaff List:");

restaurant.listStaff();

// List staff schedules

System.out.println("\nStaff Schedules:");

restaurant.listSchedules();

// List staff salaries

System.out.println("\nStaff Salaries:");

restaurant.listSalaries();

}

}

**Summary of New Features**

1. **Online Ordering**:
   * OnlineOrder class for managing online orders.
   * placeOnlineOrder method and listOnlineOrders method in Restaurant class.
2. **Table Management**:
   * TableManagement class for managing tables.
   * Integrated into Restaurant class with methods for adding, removing, and updating tables.
3. **Advanced Reporting**:
   * ReportGenerator class for generating sales, reservation, and feedback reports.
   * generateSalesReport, generateReservationReport, and generateFeedbackReport methods in Restaurant class.
4. **Employee Management**:
   * EmployeeManagement class for managing staff schedules and payroll.
   * addStaff, setSchedule, setSalary, listStaff, listSchedules, and listSalaries methods in Restaurant class.

Feel free to expand or modify the features based on your specific requirements!

**Example 08:**

Let’s design a comprehensive **Library Management System**.

This system will manage book inventory, member registrations, book borrowing, and additional features like book reservations, fine calculation, and staff management.

The system will also include various user-defined exceptions and classes for extended functionality.

**1. Book Class**

Represents a book in the library.

public class Book {

private String bookId;

private String title;

private String author;

private boolean isAvailable;

public Book(String bookId, String title, String author) {

this.bookId = bookId;

this.title = title;

this.author = author;

this.isAvailable = true;

}

public String getBookId() {

return bookId;

}

public String getTitle() {

return title;

}

public String getAuthor() {

return author;

}

public boolean isAvailable() {

return isAvailable;

}

public void setAvailable(boolean available) {

isAvailable = available;

}

@Override

public String toString() {

return "Book ID: " + bookId + ", Title: " + title + ", Author: " + author + ", Available: " + isAvailable;

}

}

**2. Member Class**

Represents a library member.

public class Member {

private String memberId;

private String name;

public Member(String memberId, String name) {

this.memberId = memberId;

this.name = name;

}

public String getMemberId() {

return memberId;

}

public String getName() {

return name;

}

@Override

public String toString() {

return "Member ID: " + memberId + ", Name: " + name;

}

}

**3. BookLoan Class**

Represents a book loan transaction.

import java.time.LocalDate;

public class BookLoan {

private String loanId;

private String bookId;

private String memberId;

private LocalDate loanDate;

private LocalDate returnDate;

private boolean isReturned;

public BookLoan(String loanId, String bookId, String memberId, LocalDate loanDate) {

this.loanId = loanId;

this.bookId = bookId;

this.memberId = memberId;

this.loanDate = loanDate;

this.returnDate = null;

this.isReturned = false;

}

public String getLoanId() {

return loanId;

}

public String getBookId() {

return bookId;

}

public String getMemberId() {

return memberId;

}

public LocalDate getLoanDate() {

return loanDate;

}

public LocalDate getReturnDate() {

return returnDate;

}

public boolean isReturned() {

return isReturned;

}

public void setReturnDate(LocalDate returnDate) {

this.returnDate = returnDate;

this.isReturned = true;

}

@Override

public String toString() {

return "Loan ID: " + loanId + ", Book ID: " + bookId + ", Member ID: " + memberId + ", Loan Date: " + loanDate + ", Return Date: " + returnDate + ", Returned: " + isReturned;

}

}

**4. Reservation Class**

Represents a book reservation.

import java.time.LocalDate;

public class Reservation {

private String reservationId;

private String bookId;

private String memberId;

private LocalDate reservationDate;

public Reservation(String reservationId, String bookId, String memberId, LocalDate reservationDate) {

this.reservationId = reservationId;

this.bookId = bookId;

this.memberId = memberId;

this.reservationDate = reservationDate;

}

public String getReservationId() {

return reservationId;

}

public String getBookId() {

return bookId;

}

public String getMemberId() {

return memberId;

}

public LocalDate getReservationDate() {

return reservationDate;

}

@Override

public String toString() {

return "Reservation ID: " + reservationId + ", Book ID: " + bookId + ", Member ID: " + memberId + ", Reservation Date: " + reservationDate;

}

}

**5. Fine Class**

Represents a fine for overdue books.

public class Fine {

private String fineId;

private String memberId;

private double amount;

public Fine(String fineId, String memberId, double amount) {

this.fineId = fineId;

this.memberId = memberId;

this.amount = amount;

}

public String getFineId() {

return fineId;

}

public String getMemberId() {

return memberId;

}

public double getAmount() {

return amount;

}

@Override

public String toString() {

return "Fine ID: " + fineId + ", Member ID: " + memberId + ", Amount: $" + amount;

}

}

**6. Staff Class**

Represents a staff member of the library.

public class Staff {

private String staffId;

private String name;

private String role;

public Staff(String staffId, String name, String role) {

this.staffId = staffId;

this.name = name;

this.role = role;

}

public String getStaffId() {

return staffId;

}

public String getName() {

return name;

}

public String getRole() {

return role;

}

@Override

public String toString() {

return "Staff ID: " + staffId + ", Name: " + name + ", Role: " + role;

}

}

**7. Library Class**

Manages books, members, loans, reservations, fines, and staff.

import java.time.LocalDate;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

public class Library {

private List<Book> books;

private List<Member> members;

private List<BookLoan> bookLoans;

private List<Reservation> reservations;

private List<Fine> fines;

private List<Staff> staffList;

private Map<String, Integer> bookReservationCount; // Book ID to Reservation Count mapping

public Library() {

books = new ArrayList<>();

members = new ArrayList<>();

bookLoans = new ArrayList<>();

reservations = new ArrayList<>();

fines = new ArrayList<>();

staffList = new ArrayList<>();

bookReservationCount = new HashMap<>();

}

public void addBook(Book book) {

books.add(book);

}

public void addMember(Member member) {

members.add(member);

}

public void addStaff(Staff staff) {

staffList.add(staff);

}

public void loanBook(String loanId, String bookId, String memberId, LocalDate loanDate) throws BookNotAvailableException, MemberNotFoundException {

Book book = findBookById(bookId);

if (book == null) {

throw new BookNotAvailableException("Book not found");

}

if (!book.isAvailable()) {

throw new BookNotAvailableException("Book is not available");

}

BookLoan bookLoan = new BookLoan(loanId, bookId, memberId, loanDate);

bookLoans.add(bookLoan);

book.setAvailable(false);

}

public void returnBook(String loanId, LocalDate returnDate) throws LoanNotFoundException, BookNotAvailableException {

BookLoan bookLoan = findLoanById(loanId);

if (bookLoan == null) {

throw new LoanNotFoundException("Loan not found");

}

if (bookLoan.isReturned()) {

throw new BookNotAvailableException("Book already returned");

}

bookLoan.setReturnDate(returnDate);

Book book = findBookById(bookLoan.getBookId());

if (book != null) {

book.setAvailable(true);

}

calculateFine(bookLoan);

}

public void reserveBook(String reservationId, String bookId, String memberId, LocalDate reservationDate) throws BookNotAvailableException {

Book book = findBookById(bookId);

if (book == null || !book.isAvailable()) {

throw new BookNotAvailableException("Book is not available for reservation");

}

Reservation reservation = new Reservation(reservationId, bookId, memberId, reservationDate);

reservations.add(reservation);

bookReservationCount.put(bookId, bookReservationCount.getOrDefault(bookId, 0) + 1);

}

public void createFine(String fineId, String memberId, double amount) {

Fine fine = new Fine(fineId, memberId, amount);

fines.add(fine);

}

public void listBooks() {

for (Book book : books) {

System.out.println(book);

}

}

public void listMembers() {

for (Member member : members) {

System.out.println(member);

}

}

public void listLoans() {

for (BookLoan loan : bookLoans) {

System.out.println(loan);

}

}

public void listReservations() {

for (Reservation reservation : reservations) {

System.out.println(reservation);

}

}

public void listFines() {

for (Fine fine : fines) {

System.out.println(fine);

}

}

public void listStaff() {

for (Staff staff : staffList) {

System.out.println(staff);

}

}

private Book findBookById(String bookId) {

for (Book book : books) {

if (book.getBookId().equals(bookId)) {

return book;

}

}

return null;

}

private BookLoan findLoanById(String loanId) {

for (BookLoan loan : bookLoans) {

if (loan.getLoanId().equals(loanId)) {

return loan;

}

}

return null;

}

private void calculateFine(BookLoan bookLoan) {

if (bookLoan.getReturnDate() != null) {

LocalDate dueDate = bookLoan.getLoanDate().plusWeeks(2); // Assuming a 2-week loan period

if (bookLoan.getReturnDate().isAfter(dueDate)) {

long daysLate = bookLoan.getReturnDate().toEpochDay() - dueDate.toEpochDay();

double fineAmount = daysLate \* 1.00; // $1 per day late

createFine("F" + fines.size(), bookLoan.getMemberId(), fineAmount);

}

}

}

public void generateReport() {

System.out.println("\nLibrary Report:");

listBooks();

listMembers();

listLoans();

listReservations();

listFines();

listStaff();

}

}

**8. Custom Exceptions**

Define custom exceptions for various error scenarios.

public class BookNotAvailableException extends Exception {

public BookNotAvailableException(String message) {

super(message);

}

}

public class MemberNotFoundException extends Exception {

public MemberNotFoundException(String message) {

super(message);

}

}

public class LoanNotFoundException extends Exception {

public LoanNotFoundException(String message) {

super(message);

}

}

**9. Library Management System Test**

Here’s how you can test the functionality of the library management system:

import java.time.LocalDate;

public class Main {

public static void main(String[] args) {

Library library = new Library();

// Add books

library.addBook(new Book("B1", "Java Programming", "John Doe"));

library.addBook(new Book("B2", "Data Structures", "Jane Smith"));

library.addBook(new Book("B3", "Design Patterns", "Emily Davis"));

// Add members

library.addMember(new Member("M1", "Alice"));

library.addMember(new Member("M2", "Bob"));

// Add staff

library.addStaff(new Staff("S1", "Carol", "Librarian"));

library.addStaff(new Staff("S2", "Dave", "Assistant"));

// Loan a book

try {

library.loanBook("L1", "B1", "M1", LocalDate.now());

library.loanBook("L2", "B2", "M2", LocalDate.now());

} catch (BookNotAvailableException | MemberNotFoundException e) {

System.out.println("Exception caught: " + e.getMessage());

}

// Return a book and calculate fine

try {

library.returnBook("L1", LocalDate.now().plusDays(5)); // Assume returning late

} catch (LoanNotFoundException | BookNotAvailableException e) {

System.out.println("Exception caught: " + e.getMessage());

}

// Reserve a book

try {

library.reserveBook("R1", "B3", "M1", LocalDate.now().plusDays(1));

} catch (BookNotAvailableException e) {

System.out.println("Exception caught: " + e.getMessage());

}

// Create a fine

library.createFine("F1", "M1", 5.00);

// Generate a report

library.generateReport();

}

}

**10. Advanced Features**

**Online Ordering**:

Implement an online catalog and reservation system for checking book availability and making reservations.

public class OnlineCatalog {

private Library library;

public OnlineCatalog(Library library) {

this.library = library;

}

public void viewAvailableBooks() {

for (Book book : library.getBooks()) {

if (book.isAvailable()) {

System.out.println(book);

}

}

}

public void reserveBookOnline(String reservationId, String bookId, String memberId, LocalDate reservationDate) {

try {

library.reserveBook(reservationId, bookId, memberId, reservationDate);

System.out.println("Book reserved successfully.");

} catch (BookNotAvailableException e) {

System.out.println("Exception caught: " + e.getMessage());

}

}

}

**Table Management**:

Manage book genres, book locations in the library.

public class BookLocation {

private String bookId;

private String location;

public BookLocation(String bookId, String location) {

this.bookId = bookId;

this.location = location;

}

public String getBookId() {

return bookId;

}

public String getLocation() {

return location;

}

@Override

public String toString() {

return "Book ID: " + bookId + ", Location: " + location;

}

}

**Advanced Reporting**:

Generate detailed reports.

public class ReportGenerator {

public void generateBookAvailabilityReport(List<Book> books) {

System.out.println("Book Availability Report:");

for (Book book : books) {

System.out.println(book);

}

}

public void generateReservationReport(List<Reservation> reservations) {

System.out.println("Reservation Report:");

for (Reservation reservation : reservations) {

System.out.println(reservation);

}

}

public void generateFineReport(List<Fine> fines) {

System.out.println("Fine Report:");

for (Fine fine : fines) {

System.out.println(fine);

}

}

}

**Employee Management**:

Manage staff details.

public class EmployeeManagement {

private List<Staff> staffList;

public EmployeeManagement() {

staffList = new ArrayList<>();

}

public void addStaff(Staff staff) {

staffList.add(staff);

}

public void listStaff() {

for (Staff staff : staffList) {

System.out.println(staff);

}

}

}

**11. Complete Example**

Here is the complete example including all features:

import java.time.LocalDate;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

public class Library {

private List<Book> books;

private List<Member> members;

private List<BookLoan> bookLoans;

private List<Reservation> reservations;

private List<Fine> fines;

private List<Staff> staffList;

private Map<String, Integer> bookReservationCount;

private OnlineCatalog onlineCatalog;

private ReportGenerator reportGenerator;

private EmployeeManagement employeeManagement;

public Library() {

books = new ArrayList<>();

members = new ArrayList<>();

bookLoans = new ArrayList<>();

reservations = new ArrayList<>();

fines = new ArrayList<>();

staffList = new ArrayList<>();

bookReservationCount = new HashMap<>();

onlineCatalog = new OnlineCatalog(this);

reportGenerator = new ReportGenerator();

employeeManagement = new EmployeeManagement();

}

public void addBook(Book book) {

books.add(book);

}

public void addMember(Member member) {

members.add(member);

}

public void addStaff(Staff staff) {

staffList.add(staff);

employeeManagement.addStaff(staff);

}

public void loanBook(String loanId, String bookId, String memberId, LocalDate loanDate) throws BookNotAvailableException, MemberNotFoundException {

Book book = findBookById(bookId);

if (book == null) {

throw new BookNotAvailableException("Book not found");

}

if (!book.isAvailable()) {

throw new BookNotAvailableException("Book is not available");

}

BookLoan bookLoan = new BookLoan(loanId, bookId, memberId, loanDate);

bookLoans.add(bookLoan);

book.setAvailable(false);

}

public void returnBook(String loanId, LocalDate returnDate) throws LoanNotFoundException, BookNotAvailableException {

BookLoan bookLoan = findLoanById(loanId);

if (bookLoan == null) {

throw new LoanNotFoundException("Loan not found");

}

if (bookLoan.isReturned()) {

throw new BookNotAvailableException("Book already returned");

}

bookLoan.setReturnDate(returnDate);

Book book = findBookById(bookLoan.getBookId());

if (book != null) {

book.setAvailable(true);

}

calculateFine(bookLoan);

}

public void reserveBook(String reservationId, String bookId, String memberId, LocalDate reservationDate) throws BookNotAvailableException {

Book book = findBookById(bookId);

if (book == null || !book.isAvailable()) {

throw new BookNotAvailableException("Book is not available for reservation");

}

Reservation reservation = new Reservation(reservationId, bookId, memberId, reservationDate);

reservations.add(reservation);

bookReservationCount.put(bookId, bookReservationCount.getOrDefault(bookId, 0) + 1);

}

public void createFine(String fineId, String memberId, double amount) {

Fine fine = new Fine(fineId, memberId, amount);

fines.add(fine);

}

public void listBooks() {

for (Book book : books) {

System.out.println(book);

}

}

public void listMembers() {

for (Member member : members) {

System.out.println(member);

}

}

public void listLoans() {

for (BookLoan loan : bookLoans) {

System.out.println(loan);

}

}

public void listReservations() {

for (Reservation reservation : reservations) {

System.out.println(reservation);

}

}

public void listFines() {

for (Fine fine : fines) {

System.out.println(fine);

}

}

public void listStaff() {

employeeManagement.listStaff();

}

private Book findBookById(String bookId) {

for (Book book : books) {

if (book.getBookId().equals(bookId)) {

return book;

}

}

return null;

}

private BookLoan findLoanById(String loanId) {

for (BookLoan loan : bookLoans) {

if (loan.getLoanId().equals(loanId)) {

return loan;

}

}

return null;

}

private void calculateFine(BookLoan bookLoan) {

if (bookLoan.getReturnDate() != null) {

LocalDate dueDate = bookLoan.getLoanDate().plusWeeks(2); // Assuming a 2-week loan period

if (bookLoan.getReturnDate().isAfter(dueDate)) {

long daysLate = bookLoan.getReturnDate().toEpochDay() - dueDate.toEpochDay();

double fineAmount = daysLate \* 1.00; // $1 per day late

createFine("F" + fines.size(), bookLoan.getMemberId(), fineAmount);

}

}

}

public void generateReport() {

System.out.println("\nLibrary Report:");

reportGenerator.generateBookAvailabilityReport(books);

reportGenerator.generateReservationReport(reservations);

reportGenerator.generateFineReport(fines);

}

public OnlineCatalog getOnlineCatalog() {

return onlineCatalog;

}

public static void main(String[] args) {

Library library = new Library();

// Add books

library.addBook(new Book("B1", "Java Programming", "John Doe"));

library.addBook(new Book("B2", "Data Structures", "Jane Smith"));

library.addBook(new Book("B3", "Design Patterns", "Emily Davis"));

// Add members

library.addMember(new Member("M1", "Alice"));

library.addMember(new Member("M2", "Bob"));

// Add staff

library.addStaff(new Staff("S1", "Carol", "Librarian"));

library.addStaff(new Staff("S2", "Dave", "Assistant"));

// Loan a book

try {

library.loanBook("L1", "B1", "M1", LocalDate.now());

library.loanBook("L2", "B2", "M2", LocalDate.now());

} catch (BookNotAvailableException | MemberNotFoundException e) {

System.out.println("Exception caught: " + e.getMessage());

}

// Return a book and calculate fine

try {

library.returnBook("L1", LocalDate.now().plusDays(5)); // Assume returning late

} catch (LoanNotFoundException | BookNotAvailableException e) {

System.out.println("Exception caught: " + e.getMessage());

}

// Reserve a book

try {

library.reserveBook("R1", "B3", "M1", LocalDate.now().plusDays(1));

} catch (BookNotAvailableException e) {

System.out.println("Exception caught: " + e.getMessage());

}

// Create a fine

library.createFine("F1", "M1", 5.00);

// Generate a report

library.generateReport();

// Use Online Catalog

library.getOnlineCatalog().viewAvailableBooks();

library.getOnlineCatalog().reserveBookOnline("R2", "B2", "M2", LocalDate.now());

// Generate updated report

library.generateReport();

}

}

**Summary of New Features Implemented**

1. **Online Ordering**: OnlineCatalog class allows online reservations and viewing available books.
2. **Table Management**: Added BookLocation class for managing book locations.
3. **Advanced Reporting**: ReportGenerator class for generating detailed reports.
4. **Employee Management**: EmployeeManagement class for managing staff details and schedules.

This example covers a wide range of features and functionality suitable for a comprehensive library management system, similar in scale and complexity to the earlier Restaurant Management System example.