|  |  |
| --- | --- |
| Collection | Focus: Java collections - List(Arraylist,Linkedlist),Queue and Set Collection Interface - Iterator,Iterable,Collection interfaces:  Here are two scenario-based questions related to Java Collections, focusing on List, Queue, Set, and the Collection interface including Iterator, Iterable, and Collection interfaces.  **Question 1: Managing a Library System**  **Scenario:** You are developing a library management system that needs to keep track of books and their availability. Each book has a title, author, and unique identifier (ISBN).   * **Part A:** Which Java Collection would you use to store the collection of books, and why? Consider factors like ordering, uniqueness, and retrieval speed. * **Part B:** Implement a method to add a new book to the collection. Include the use of the Iterator to check if the book already exists before adding it to avoid duplicates. – **need answer** |
|  | **Employee Management System(List, streams,Filters)**  **Context:** You are developing an employee management system for a company that maintains a list of employees. Each employee has attributes like name, age, department, and salary.  **Business Case Question**  **Question:**   * **Part A:** You need to retrieve a list of all employees who work in the "Sales" department and have a salary greater than $50,000. Describe how you would use Java Streams to filter this data. Include a brief code snippet to demonstrate the filtering process. * **Part B:** Additionally, you want to implement a functional interface called EmployeeFilter that has a method test(Employee employee) to filter employees based on custom criteria. Show how you would use this interface along with Streams to achieve the same goal as in Part A. Provide a code example demonstrating this implementation. |
| use of static nested classes and inner classes in Java. | Great question! Using separate classes, static nested classes, and inner classes each has its own purpose and advantages, depending on the context and requirements of your application. Here are some reasons for using static nested classes and inner classes instead of separate classes:  **Reasons to Use Static Nested Classes**   1. **Logical Grouping:**    * Static nested classes are used when a class is logically related to its enclosing class. For example, Category in the Product class is a category of products, and it makes sense to group them together. 2. **Encapsulation:**    * It helps encapsulate the nested class. If a class is not meant to be used outside its enclosing class, making it a static nested class can restrict its visibility and use, promoting better encapsulation. 3. **Simplicity and Clarity:**    * Using a static nested class can make the code easier to read and maintain, as it visually indicates that the nested class is closely related to the enclosing class. It reduces clutter in the top-level namespace. 4. **No Need for Instance Access:**    * Since static nested classes do not require an instance of the enclosing class to be created, they can be more efficient in certain scenarios, especially when you don’t need to access instance variables or methods.   **Reasons to Use Inner Classes**   1. **Access to Instance Variables:**    * Inner classes can access the instance variables and methods of the enclosing class. This is particularly useful when you want the nested class to directly manipulate or utilize the enclosing class's data. 2. **Enhanced Encapsulation:**    * Similar to static nested classes, inner classes can also help encapsulate functionality that is specific to the enclosing class. This enhances encapsulation by preventing outside classes from directly interacting with the inner class. 3. **Event Handling:**    * Inner classes are commonly used in GUI programming (e.g., Swing in Java) for event handling. They can directly reference the outer class's instance variables (like UI components) without needing extra parameters. 4. **Implementation of Interfaces:**    * An inner class can implement interfaces that need to use the instance variables of the outer class, making it convenient to handle callbacks and events in a cohesive manner.   **When to Use Separate Classes**   1. **Reusability:**    * If a class is intended to be reused across different contexts or classes, it is better to implement it as a separate top-level class. This makes it easier to maintain and test. 2. **Simplicity:**    * If the class does not logically belong to the enclosing class, or if it is complex and can stand on its own, using a separate class can simplify the structure and increase clarity. 3. **Avoiding Confusion:**    * Using too many nested classes can lead to confusion regarding their roles and relationships. In cases where separation is beneficial for understanding the code, separate classes are preferable.   E.G: Swing, GUI |
| Static and nested class | Questions:  **Static Nested Class:** Useful for grouping related constants or methods without requiring an instance of the enclosing class.   **Inner Class:** Allows direct access to the instance variables and methods of the enclosing class, providing a way to create complex relationships between classes. |
| Exceptions | **1. Checked Exceptions**  **Example: IOException**  **Example: SQLException**  **Unchecked Exceptions**  **Example: NullPointerException**  **Example: ArrayIndexOutOfBoundsException**  **Example: IllegalArgumentException**  **3. Custom Exceptions**  **Example: UserDefinedException**  **Example: CustomUncheckedException** |
| Wrapper | **. Unboxing (Automatic Conversion from Wrapper to Primitive)**   * **Purpose**: Converts a wrapper class object (Boolean, Integer, Double) to its corresponding primitive type (boolean, int, double). * **Explanation**: When assigning an Integer or Double object to a primitive int or double, the unboxing feature of Java automatically converts the object to its primitive equivalent.   int agePrimitive = voterAge; // Unboxing Integer to int  double ratingPrimitive = preferenceRating; // Unboxing Double to double  **valueOf() methods**: These are factory methods used to convert strings to their corresponding wrapper objects (Integer, Double). They handle parsing and return null or throw exceptions when the input is invalid.  Boolean.TRUE |
| Wrapper | Question: Write a Java program that uses Boolean, Double, and Integer wrapper classes to store the voter's agreement status, age, and preference rating (out of 10). Demonstrate how to handle null values and convert these wrapper types back to their primitive equivalents. |
| Object Reference Variables and Primitive Variables | 5. Differentiate Between Object Reference Variables and Primitive Variables  Scenario:  In your role as a software engineer, you are tasked with optimizing the memory usage of a real-time stock trading application. You have to decide between using primitive types and object references to store frequently updated numerical values like stock prices and volumes.  Question:  Explain the difference between object reference variables and primitive variables in terms of memory usage and performance. In which cases would you prefer to use primitive types over object references in this stock trading application, and why? |
| Static | You are designing a class to track user sessions in a web application. Each user can have one active session at a time, but there should be a static counter that tracks the total number of sessions created across all users. Question: Explain how you would use the static keyword to keep track of the total number of sessions, and the this keyword to refer to the current user's session. Write a Java program demonstrating the use of both static and this. |
| Final | You are creating a library management system. The base class LibraryItem has common properties like title and author. You are developing a subclass Book that inherits from LibraryItem, but you want to ensure that no further classes can inherit from Book. Question: Write a Java program that demonstrates the use of the super keyword to call the base class constructor from the Book class. Also, explain how and why you would use the final keyword to prevent inheritance of the Book class. |
| Final |  **Super Keyword**:   * The super keyword is used in the Book class constructor to call the constructor of the LibraryItem base class. This ensures that the title and author properties are initialized in the base class. * Additionally, super.displayDetails() is used to call the displayDetails() method from the base class to display the title and author before displaying the ISBN of the book.    **Final Keyword**:   * The final keyword is used before the Book class declaration to prevent any further classes from inheriting from it. This is useful when you want to stop the inheritance chain at a specific point. * In this scenario, we use final on the Book class to ensure no other class can extend it, which might be necessary to maintain the integrity of certain types of objects or business rules in your system.   **Why Use final:**   * In cases where you want to ensure that certain classes should not be extended (for example, to prevent modification of specific behaviors), the final keyword is essential. This helps to protect the functionality and structure of the class from unintended modifications by future subclasses. |
| Interface | Loose Coupling Using Interface Scenario: You are building a payment gateway system that should support multiple payment methods like credit cards, debit cards, and PayPal. You want to implement loose coupling so that future payment methods can be added without changing the existing codebase. Question: Explain how you would use interfaces to design a payment processing system. Write Java code that implements an interface OnlinePayment. Implement a CreditCardProcessor class that inherits from PaymentProcessor and uses the OnlinePayment interface. Show how this setup ensures loose coupling and flexibility.  make the system more dynamic by accepting input from the user, we can use the Scanner class to take input from the console for the payment type, card number, email, and the payment amount. Based on the input, we can then process the payment.  Loose Coupling Using Interface Scenario:  1. You are building a payment gateway system that should support multiple payment methods like credit cards,  debit cards, and PayPal.  2. You want to implement loose coupling so that future payment methods can be added without changing the  existing codebase.  3. Question: Explain how you would use interfaces to design a payment processing system.  Write Java code that implements an interface OnlinePayment.  Implement a CreditCardProcessor class that inherits from PaymentProcessor and uses the OnlinePayment interface.  Show how this setup ensures loose coupling and flexibility.  4. Make the system more dynamic by accepting input from the user,  we can use the Scanner class to take input from the console for the payment type, card number, email,  and the payment amount. Based on the input, we can then process the payment. |
|  | focused on creating test cases, writing test methods, and running tests using JUnit 5:  **Business Case Scenario: Online Learning Platform**  **Background:** You are part of a development team working on an online learning platform that allows users to enroll in courses, take quizzes, and track their learning progress. Your team has recently implemented a feature that allows instructors to create quizzes for their courses. This feature needs to be thoroughly tested to ensure it functions correctly before it goes live.  **Scenario:** The product owner has tasked your team with ensuring the quiz creation feature is robust and bug-free. To achieve this, you need to develop a suite of unit tests using JUnit 5.  **Tasks:**   1. **Creating Test Cases:** Identify the key functionalities of the quiz creation feature that need to be tested. This includes aspects like quiz title validation, question addition, time limits, and scoring mechanisms. Outline at least five test cases that cover both positive and negative scenarios. 2. **Writing Test Methods:** For each identified test case, write a test method in JUnit 5 that will verify the expected outcomes. Ensure to utilize JUnit 5 annotations like @Test, @BeforeEach, and @AfterEach as appropriate. 3. **Running Tests:** After implementing your test methods, run the test suite. Discuss how you would handle any failing tests and the steps you would take to debug the issues.   **Questions:**   1. What are the five test cases you would create for the quiz creation feature? Explain the rationale behind each one. 2. Provide an example of a JUnit 5 test method for one of the test cases you identified. 3. After running the tests, you find that two tests have failed. What steps will you take to investigate and resolve these failures?  * Encourages to think critically about software testing processes and gives them an opportunity to apply JUnit 5 concepts in a real-world context.   **1. Identified Test Cases:**  Here are five test cases for the quiz creation feature:   * **Test Case 1: Quiz Title Validation**   + **Objective:** Ensure the quiz title is required and adheres to length constraints (e.g., 5 to 100 characters).   + **Positive Scenario:** Valid title is accepted.   + **Negative Scenario:** Title is empty or exceeds 100 characters. * **Test Case 2: Question Addition**   + **Objective:** Verify that questions can be added to the quiz.   + **Positive Scenario:** A valid question is added successfully.   + **Negative Scenario:** Attempting to add a question with invalid formats (e.g., empty question text) fails. * **Test Case 3: Time Limit Setting**   + **Objective:** Check that the time limit for the quiz can be set correctly.   + **Positive Scenario:** A valid time limit (e.g., 1 to 120 minutes) is accepted.   + **Negative Scenario:** Setting a negative or excessively high time limit results in an error. * **Test Case 4: Scoring Mechanism**   + **Objective:** Ensure the scoring mechanism calculates scores correctly based on correct answers.   + **Positive Scenario:** Correct answers receive appropriate points.   + **Negative Scenario:** Incorrect answers do not add to the score. * **Test Case 5: Quiz Visibility Settings**   + **Objective:** Test the visibility settings to determine if quizzes can be set as public or private.   + **Positive Scenario:** Quiz is marked as public and is accessible by users.   + **Negative Scenario:** A private quiz is not accessible to non-enrolled users.   **2. Example of a JUnit 5 Test Method:**  Here’s a sample JUnit 5 test method for **Test Case 1: Quiz Title Validation**:  java  Copy code  import org.junit.jupiter.api.BeforeEach;  import org.junit.jupiter.api.Test;  import static org.junit.jupiter.api.Assertions.\*;  public class QuizTest {  private Quiz quiz;  @BeforeEach  public void setUp() {  quiz = new Quiz();  }  @Test  public void testQuizTitleValidation\_ValidTitle() {  String validTitle = "Java Basics Quiz";  quiz.setTitle(validTitle);  assertEquals(validTitle, quiz.getTitle());  }  @Test  public void testQuizTitleValidation\_EmptyTitle() {  Exception exception = assertThrows(IllegalArgumentException.class, () -> {  quiz.setTitle("");  });  assertEquals("Quiz title cannot be empty", exception.getMessage());  }  @Test  public void testQuizTitleValidation\_TooLongTitle() {  String longTitle = "This title is way too long and should exceed the maximum length allowed for a quiz title which is set to be 100 characters.";  Exception exception = assertThrows(IllegalArgumentException.class, () -> {  quiz.setTitle(longTitle);  });  assertEquals("Quiz title cannot exceed 100 characters", exception.getMessage());  }  }  **3. Steps for Investigating and Resolving Failing Tests:**  If two tests fail after running the test suite, I would take the following steps:   1. **Review Test Output:**    * Check the test report for details about the failing tests, including error messages and stack traces. 2. **Identify the Cause:**    * Look at the specific assertions that failed. For instance, if a title validation test failed, verify the logic in the setTitle method of the Quiz class. 3. **Debugging:**    * Use debugging tools to step through the code where the failure occurred, examining variable states and flow. 4. **Reproduce the Issue:**    * Try to reproduce the failure outside of the test to ensure it’s a code issue, not a problem with the test itself. 5. **Make Corrections:**    * Based on the findings, correct the underlying code or test logic as needed. 6. **Re-run the Tests:**    * After making changes, re-run the test suite to verify that the issues are resolved and that no new tests fail. 7. **Document Changes:**    * Document the findings and changes made in the codebase to keep a record for future reference. |
|  |  |