**Answer 1.a.**

The UserFitness class has low cohesion because it's trying to do too many things at once. It's managing user info like userName and weight, tracking workouts with things like caloriesBurned and logWorkout(), and also handling nutrition with caloriesConsumed and logMeal(). These responsibilities are pretty different from each other, so by putting them all into one class, it's like the class doesn't really have a clear focus. In a well-designed class, everything should be closely related, but here, the class is handling multiple tasks that don’t necessarily belong together, which is why the cohesion is low.

**Answer 1.b.**

To make the class more focused, I would split it into separate classes that each have one job. For example, I would create a UserProfile class just for storing and updating basic user info like name, age, weight, and height. Then, I'd make a WorkoutTracker class for handling workout stuff like steps, calories burned, and logging different workouts. Finally, I’d create a NutritionTracker class for tracking meals and calories consumed.

This way, each class has a specific responsibility and is more cohesive because it's only focused on one thing. It also makes the code easier to manage since all related stuff is in the same place instead of everything being mashed together.

Reference: <https://www.geeksforgeeks.org/cohesion-in-java/>

**Answer 2 - Part 1:**

Student should be an abstract class because it can't really exist on its own. Every student is at least a first, second, or third-year student. By making it abstract, we can define common things like name and ID, and methods like getYear() that are shared across all students, without allowing someone to create a general Student object. This fits better than an interface because we can include shared properties and some methods with actual implementations, which wouldn't be possible with just an interface.

**Answer 2.a.**

The method is flawed because it’s trying to remove elements from the list while looping over it. This causes issues with how the list is indexed, which can lead to skipping elements or errors. According to Riel’s heuristic, you shouldn’t modify a list while iterating through it. Instead, use an iterator to safely remove elements while looping through the list.

References: <https://www.geeksforgeeks.org/difference-between-abstract-class-and-interface-in-java/>

**Answer 3.**

No, the BankAccount class isn't well-encapsulated because the fields like accountHolderName, accountNumber, balance, and interestRate are all public, which allows them to be accessed and modified directly from outside the class. To improve this, you should make these fields private and use getter and setter methods to control how the fields are accessed and updated. This would ensure better protection of the data and avoid accidental changes from outside the class.

Reference : <https://www.geeksforgeeks.org/encapsulation-in-java/>

**Answer 4.a.**

The current structure uses inheritance for trim levels (Base, Sports, Luxury), which means once a car is created with a specific trim, it can’t change to another trim without creating a new object. Inheritance locks the car into a single trim, preventing dynamic changes.

**Answer 4.b.**

To fix this, we can use composition. Instead of using inheritance for trim levels, create a TrimLevel class or interface. Then, give Car a reference to a TrimLevel object, so you can change the trim by assigning a new TrimLevel to the car, and this allows us to make dynamic changes.

**Answer 5.**

If I have two classes with a lot of the same code, it’s an issue because I’d be repeating myself, which goes against good OOP practices. According to Arthur Riel’s heuristics, code duplication makes the program harder to maintain because if I need to make changes, I’d have to update both classes, which could lead to mistakes or bugs.

To solve this, I could use inheritance or composition. If the two classes share a lot of common functionality, I could refactor the shared code into a parent class and have both classes inherit from it. If inheritance isn’t the best fit, I could use composition by creating a separate class that handles the shared behavior, which both classes can use. This way, I reduce duplication and make my code easier to maintain.

**Answer 6.**

Putting database calls directly in classes like Car or BankAccount isn’t a good idea because it makes the class responsible for too many things. The class should only focus on what it’s supposed to do (like managing a car’s details), not how it gets saved in the database. If the database changes, you’d have to go back and change the class itself, which makes everything harder to update and manage.

Instead, it’s better to keep the database stuff separate so the class can focus on its main job, and the saving or loading of data is handled somewhere else. This keeps things cleaner and easier to work with.

**Answer 7.**

To create a new class hierarchy using inheritance, a few things need to be true:

* Is-A Relationship: The subclass should be a type of the superclass, like a Car being a type of Vehicle.
* Shared Features: Both the superclass and subclass should have some common methods or properties to avoid repeating the same code.
* New Features in Subclass: The subclass should add its own new things, like features that don't make sense for the superclass.
* Avoid Repeating Code: We should avoid writing the same code in multiple places.

Reference: <https://www.w3resource.com/java-exercises/java-inheritance-exercise-10.php>

**Answer 8.**

Inheritance is the design technique in object-oriented programming to implement is-a relationship between objects. Inheritance in Java is implemented using the extends keyword. For example, Cat is an Animal relationship in java programming will be implemented like below.

Composition is the design technique in object-oriented programming to implement a relationship between objects. Composition in java is achieved by using instance variables of other objects. For example, a person who has a Job is implemented like below in java object-oriented programming.

You’d choose inheritance when there’s a strong relationship and shared behavior between the classes. Composition is better when you want more flexibility and when the parts of a class can stand alone.

Reference: <https://www.digitalocean.com/community/tutorials/composition-vs-inheritance>

**Answer 9.**

Cohesion in Java is the Object-Oriented principle most closely associated with making sure that a class is designed with a single, well-focused purpose. In object-oriented design, cohesion refers to how a single class is designed.

High cohesion is when you have a class that does a well-defined job. Low cohesion is when a class does a lot of jobs that don’t have much in common.

High cohesion gives us better-maintaining facility and Low cohesion results in monolithic classes that are difficult to maintain, understand and reduce re-usability

Reference: <https://www.geeksforgeeks.org/cohesion-in-java/>