Daffodil International University
Faculty of Science & Information Technology
Department of Software Engineering
Finalterm Examination, Summer 2026
Course Code: SE216; Course Title: Object Oriented Programming
Sections & Teachers: UKD(A,B,C), RJM(D,E,H), MBH(F), MSN(G), AJE(I), MIH(J)

Time: 2 Hours | Marks: 40

Answer ALL Questions

[The figures in the right margin indicate the full marks and corresponding course outcomes. All portions of each question must be answered sequentially.]

1.	*a)*	**Generate** a Java class called Fest with a main() method that does the following, in order: 1. Create a list called list containing 5 values: A, B, C, D, E. 2. A new item, "X", needs to be placed into the list so that it sits exactly between "B" and "C", without disturbing the rest of the order. 3. The value "D" has been recorded incorrectly. Correct it to "Z" in-place, without removing and re-adding it. 4. The item currently sitting second in the list is no longer needed. Get rid of it using its position, not its value. 5. Create a second list called junk containing A and X. Update list so that every value also present in junk is gone from it. 6. Store how many items remain in the list, letting Java figure out the type of that value on its own instead of stating it explicitly. Then print "Non-empty: " + <that value> if there's at least one item left, otherwise print "Empty". 7. Two more items, "M" and "N", arrived together and both need to go in at once, starting right at the beginning of the list (index 0), keeping their own order relative to each other. 8. The team is asked to check for an item called "Q": a value that was never part of any list here. Before trying to find its position, first check whether it's even present. Print "Not found: Q" if it isn't there then do NOT let the code attempt to locate a position for something that doesn't exist. 9. Put the remaining items in the list into alphabetical order. 10. Print list, then empty it out completely, then print it once more to confirm nothing remains.	[10]	CLO -3 *Level -3*
	b)	**Solve** the following scenario using appropriate UML diagrams, illustrating the association relationships. • A Professor teaches at a university. The professor has a name of type String. The Professor performs an action teachCourse() of type void. • A Professor is associated with a Course, where one professor can teach multiple courses, and each course is taught by one professor. • A Department contains multiple Professors, but professors can continue to exist even if the department is removed or reorganized. The Department has a departmentName of type String. • Each Department must have at least one Course. If a department is removed, all of its courses are removed as well. A Course has a courseCode of type String.	[3]	
	c)	i) You are building a system to manage product stock records, split across two packages: shop: contains the core product class run: contains the class that creates and tests product objects **Build** a class called Product inside the shop package that stores: • A product name (String) • A quantity (int) Requirements: 1. Both fields must be fully inaccessible directly from outside the class, even from classes in other packages. 2. Provide controlled access so other packages can read and update the values - but the setter for quantity must reject any value below 0 (print an error message instead of applying the change).	[6+1]	

		3. In a separate class called ProductTest, placed in the run package, import the Product class from shop and: • Create a Product object with a name and starting quantity of 50. • Attempt to set the quantity directly to -10 (should be rejected). • Print the final product name and quantity using the getters. **Expected Output:** Error: Quantity cannot be negative. Update ignored. Final Name: Notebook Final Quantity: 50 ii) If the Product class itself (not just its fields) was declared without the public keyword, would ProductTest still compile? Why or why not?		
2.	*a)*	**Build** the full JAVA code for the below given scenario. • Create an interface Netty with one abstract method: zap(). • Create an interface Hero with one abstract method: bond(). • Create an interface Juicy that extends Netty, and adds one more abstract method: fuel(). • Create an abstract class Gizmo that implements Juicy. Gizmo should have: ➔ Two of its own abstract methods: wake() and sleep() ➔ One concrete method: report() • Then, create a subclass Boom that extends Gizmo and implements Hero, providing implementations for all necessary methods. • Finally, in a TesterClass, create an instance (object) of every class and invoke (call) every method on each object. **The output should display the corresponding messages for each method call, as follows:** Boom is waking up Boom zapped into WiFi Boom fueling up Boom bonded with Hero Boom's report is currently active Boom going to sleep	[12]	CLO-4 *Level -3*
	b)	You are developing a simple ATM withdrawal system that validates customer withdrawal requests and handles errors using custom exceptions and nested try-catch blocks. **Handle** the Exceptions Using the Given Requirements: 1. Create one custom exception class: • **InvalidAmountException:** Thrown when the withdrawal amount is less than or equal to 0, or when it is not a multiple of 100. 2. Create an ATM class that includes: • A balance field initialized to 5000. • A withdraw(double amount) method that uses a simple if / else if / else structure: • If the withdrawal amount is invalid (less than or equal to 0, or not a multiple of 100), throw an **InvalidAmountException.** • Else if the withdrawal amount is greater than the available balance, throw an **ArithmeticException.** • Otherwise, deduct the withdrawal amount from the balance and display a success message. 3. In the main method: Use an outer try-catch block to process an array of withdrawal requests (e.g., {-500, 200, 7000, 350}). Inside the loop, use an inner try-catch block to call the withdraw() method. Catch InvalidAmountException and ArithmeticException separately, and display the exception message using e.getMessage(). Use the outer catch block to handle any unexpected exceptions. 4. Add a finally block that prints "Transaction attempt finished." after each withdrawal attempt. 5. After all transactions have been processed, display the final account balance.	[8]	