CS2001 W07-StacksQueues-Report

1. Introduction

In this assignment I implemented two fundamental data structures in Java - a DoubleStack (two stacks sharing one array) and a DoubleStackQueue (a FIFO queue built using two stacks). The goal was to achieve an efficient memory design (two stacks in one array) while strictly conforming to provided interfaces for stacks and queues. I focused on ensuring correctness in all boundary conditions (empty structures, full capacity) through proper exception handling, and I developed a thorough JUnit test suite to validate the implementation under normal and edge cases. The work emphasized interface adherence and robust design: my DoubleStack and DoubleStackQueue classes implement the given IStack, IDoubleStack, and IQueue interfaces exactly, and throw appropriate exceptions as specified. This report reflects on the design decisions, implementation details, testing strategy, and lessons learned in the process.

2. Design & Implementation

2.1 Core data structures

Shared array Doublestack: The DoubleStack is implemented with a single fixed-size array holding two distinct stacks as required by the specification. One stack (“first stack”) uses the array from the beginning (index 0 upwards), and the other (“second stack”) uses the array from the end (index n-1 downwards). By having the two stacks grow towards each other, they can share the array without conflict. As required in the specification, the stacks are capped at using at most half of the array, this effectively creates an isolation barrier in the middle of the array so the two stacks can’t overlap with each other and overwrite each other’s data.

Internally, the DoubleStack maintains two pointers for the tops of the stacks. The first stack’s bottom is at index 0 and grows right (increasing index), while the second stack’s bottom starts at index n-1 and grows left (decreasing index). Because we never allow either index to cross the midpoint boundary, the tops will never collide. On an odd-sized array, one slot in the exact middle remains unused as a buffer (e.g. with capacity 9, each stack maxes at 4 elements, leaving index 4 unused. I decided to take this approach to ensure isolation of the two LIFO structures at the cost of potentially not using one slot when N is odd (discussed further in Evaluation).

2.2 Interface and validation

Stack interface implementation: I implemented all methods of the IStack interface for both stack instances. My implementation closely follows the intended semantics of a stack. For push() the method checks that the stack is not full, throwing a StackOverflowException if no more elements can be pushed. The same respective runtime checks are performed for pop() and top() checking that the stack is not empty, throwing a StackEmptyException if it is. These checks are crucial for defensive design: they enforce the preconditions of each operation and ensure the stack cannot enter an invalid state (such as returning a bogus value when empty).

Queue interface implementation: DoubleStackQueue implements IQueue using the classic two-stack technique: an input stack for enqueue and an output stack for dequeue. Internally, I construct a DoubleStack with array length 2*Q, where Q is the logical queue capacity. Each internal stack therefore has room for Q items (half of 2*Q), satisfying the “each stack <= half of its array” requirement while letting the queue be full if size()==Q. enqueue(e) first checks size()==Q and throws QueueFullException if full; otherwise it pushes to the input stack. dequeue() throws a QueueEmptyException if both stacks are empty; if the output stack is empty but the input is not, it transfers all items from input to output (reversing order) and then pops from output. Internally impossible states (e.g., overflow during transfer under size guard) are surfaced as hard failures (IllegalStateException / AssertionError) rather than being misreported as queue-level errors.

2.3 DoubleStackQueue architecture

The DoubleStackQueue implements a FIFO queue using two stacks - an input stack for enqueues and an output stack for dequeues. When the output stack becomes empty, all elements from the input stack are transferred into it in reverse order, restoring FIFO order.
In the original design, each internal stack was restricted to half of the queue’s total array size. While this worked, it meant the queue could sometimes appear “full” even when there was unused space - for instance, if the output stack still contained a few items, the input stack could hit its half-capacity and prevent further enqueues.
To address this, I modified the internal design so that the queue’s DoubleStack now allocates an array of size 2 * Q, where Q is the logical queue capacity. Each internal stack (input and output) therefore has Q available slots, satisfying the requirement that

each stack uses at most half of its own array, while ensuring that the queue as a whole can hold up to exactly Q items.

I also have to highlight that the interface constrains me from throwing anything other than QueueEmptyException for the dequeue() method. So for cases where I need to throw StackEmptyException and StackOverflowException, I throw AssertionError and IllegalStateException respectively.

This design still follows the coursework specification because each stack individually uses no more than half of its internal array. It simply overprovisions the underlying storage to make fullness depend on total size rather than per-half limits.

The trade-off is slightly higher memory use (the array is twice as long), I acknowledge this and decided to prioritize functionality over explicit memory. I think this solution solves the problem more elegantly and in a much more predictable manner.

3. Testing

A comprehensive suite of unit tests was developed using JUnit 5 to verify the correctness and robustness of both the DoubleStack and DoubleStackQueue implementations. I added the tests in parallel with my development, adding tests at each new functionality to make sure it worked before I moved on. I decided not to follow a TDD process as I felt confident with Stacks and Queues from the start and wanted to go straight into the development but also because I wasn't sure which approach I would take with queues, as written above, my design changed halfway through which would've affected the tests I had written pre-development.

Each test targets a specific aspect of functionality - covering normal operation, boundary conditions, and exception handling.

DoubleStack tests

What is being tested	Name of the test methods	Pre-conditions	Pass/Fail
Factory returns a valid instance	factoryReturnsNonNullDoubleStackObject	None beyond factory availability	Pass

Both stacks start empty	bothStacksAreInitiallyEmpty	Fresh IDoubleStack created in @BeforeEach	Pass
LIFO on first stack	pushAndPopFirstStack	Fresh double stack; operate on first stack	Pass
LIFO on second stack	pushAndPopSecondStack	Fresh double stack; operate on second stack	Pass
Independence of stacks	stacksDoNotInterfere	Fresh double stack; push on both stacks	Pass
Each stack limited to floor(N/2)	cannotExceedHalfCapacity	Default size 10; try pushing 6th item	Pass
Empty ops throw on both stacks	popOnEmptyThrows	New double stack of size 8; both stacks empty	Pass
LIFO order on both sides	lifoBothSides	New double stack size 10; push sequences on both	Pass
clear() empties & allows reuse	clearResetsAndAllowsReuse	New double stack size 6; push then clear	Pass
Odd capacity split by floor	oddCapacityEachGetsFloorHalf	New double stack size 9; push 4 each, 5th overflows	Pass
Half-cap bound across many N (param.)	eachStackCappedAtFloorHalf (@ValueSource {1..10,17})	For each N, fill to N/2 then assert overflow on next	Pass

DoubleStackQueue tests

What is being tested	Name of the test methods	Pre-conditions	Pass/Fail
Factory returns a valid queue	factoryReturnsNonNullDoubleStackQueue	None beyond factory availability	Pass
Basic FIFO (even capacity)	fifoBasicEven	Q=10; enqueue 1..5 then dequeue all	Pass
Single transfer then straight pops	transferOncePerBatch	Q=10; enqueue 1..5; first dequeue triggers transfer	Pass
Interleaved enq/deq preserves FIFO	interleavedOps	Q=10; mixed operations A..E	Pass
Dequeue on empty throws	dequeueEmptyThrows	Q=6; queue empty	Pass
clear() empties & reuse works	clearAllowsReuse	Q=10; enqueue then clear; enqueue again	Pass

Fill exactly to capacity (even)	fullEvenCapacity	Q=10; enqueue 1..10; then dequeue in order	Pass
Fill exactly to capacity (odd)	fullCapacityWithoutDequeues	Q=9; enqueue 1..9; Q+1 throws; partial deq/enq cycle	Pass
Transfer only when output empty (observable)	transferOnlyWhenNeeded	Q=10; after two dequeues, output holds 3,4,5; later dequeue drains output before transfer	Pass
size() reflects both stacks	sizeReflectsBothStacks	Q=10; stage items across output & input; check size and order	Pass
Draining then dequeuing throws	drainThenThrow	Q=6; enqueue 5, drain all, then dequeue again	Pass
clear() idempotence	multipleClears	Q=10; clear twice; reuse after each	Pass

Enqueue stops at logical capacity even if output has items	enqueueStopsOnlyAtLogicalCapacityWhenOutputHasData	Q=10; leave 3 in output, then enqueue up to total Q	Pass
Long interleaving over many rounds	longInterleavingRemainsFifo	Q=6; 20 rounds of patterned enq/deq; always ends empty	Pass
Odd Q obeys logical capacity	sizeNeverExceedsLogicalCapacityWhenOddQ	Q=9; enqueue 1..9; 10th throws; then FIFO drain	Pass
null values allowed (policy)	enqueueNullIsAllowedAndDequeuedAsNull	Q=4; enqueue null then "X"; ensure null dequeues first	Pass
Capacity guard across many Q (param.)	fillExactlyToCapacityThenRejectNext (@ValueSource {1..11,17})	For each Q, enqueue exactly Q; next enqueue throws; FIFO drain	Pass

Output holds *k* then fill to Q (param.)	enqueueStopsAtLogicalCapacityWhenOutputHasK (@ValueSource {2,3,4,5})	Q=10; leave k in output, then enqueue Q−k; next enqueue throws; FIFO drain	Pass
Single transfer per full "wave" (param.)	singleTransferPerWaveObservable (@ValueSource {5,10})	Fill to Q; first dequeue triggers transfer; remaining dequeues continuous	Pass

4. Evaluation

This coursework was less about "coding a stack and a queue" and more about learning to choose, justify and then iterate on a design under real constraints. I started with a spec-faithful, fixed-split DoubleStack and a standard two-stack queue; along the way I uncovered edge cases, tightened my exception contracts, re-designed the queue's capacity model, and expanded the test suite from happy paths to differential and parameterised checks. Below I reflect on the decisions, criteria, alternatives, achievements, and what I would improve given more time.

Early on, my priority was correctness. I implemented the DoubleStack so that each internal stack could occupy exactly half of the array. While this matched the brief, it introduced a subtle but frustrating limitation: when the queue was built on top of this structure, it could appear full even when there was still space left unused in the middle of the array. This was because one stack could reach its half-capacity while the other still contained elements. blocking further enqueues. Initially, I didn't realise this was a design problem - it just felt like another "bug" to patch - but after stepping back, I recognised it as a consequence of a **design assumption** rather than an implementation error.

To solve this, I made what I now consider the most significant design decision of the project: allocating an internal array twice the logical capacity of the queue. This allowed

each stack to have up to Q slots internally while still ensuring that the queue's logical size never exceeded Q. It preserved the coursework requirement that each stack use no more than half of its own array, but removed the earlier "half-full stall" problem entirely. This decision simplified the fullness logic - the queue became full only when size() == capacity - and eliminated all the confusing corner cases that arose with odd capacities. The trade-off was a small increase in memory use, but in return I gained clarity, predictability, and a design that was easier to test and explain. It felt like the first time I had made a change not just to fix something, but to make the overall design cleaner and more principled.

Reflecting on this, I realise I could have different approaches. A common alternative is a dynamic-sharing design, where both stacks grow toward each other in a single array and stop only when their tops meet. This would have saved the extra space but required far more complex boundary management. Another option would have been to abandon the two-stack model entirely (provided the specification allowed it) and implement the queue as a circular buffer, which would make enqueue and dequeue operations trivially constant time without any transfer step. Both alternatives have clear advantages - better space efficiency and potentially higher performance - but they would have moved the design away from the conceptual goal of the exercise, which was to explore how abstract data types can be built from one another. In the end, I chose the path that best balanced clarity, compliance, and simplicity within the constraints of the assignment.

I also spent time refining smaller design details that improved the overall robustness. For example, I ensured that both pop() and clear() methods in the stack explicitly set array slots to null, preventing lingering references and potential memory leaks - a lesson drawn from reading *Effective Java.* These were small but meaningful refinements that showed me how clean interfaces depend as much on what they hide as on what they expose.

If I had more time and had the opportunity to delve outside the specification, I would like to explore a few directions. The most immediate would be implementing the dynamic-sharing DoubleStack idea properly, where the stacks share the entire array dynamically instead of dividing it upfront. This would achieve the same functionality with less memory overhead, though it would introduce more complex index management. Another natural extension would be to generalise the data structure using Java generics, allowing compile-time type checking and removing the need for casting. I would also like to experiment with a resizing strategy, where the queue grows automatically when full, as well as integrating assertions or design-by-contract style invariants to formally verify correctness during runtime.

Perhaps the most valuable part of this project was recognising that every piece of code represents a design decision - and that good design isn't just about what works, but about why it works that way. My first instinct was to follow the specification literally, but by the end I was thinking in terms of design criteria: clarity, testability and maintainability. I learned to question whether my solution truly captured the intended behaviour, and to iterate toward something conceptually sound rather than just operationally correct. In that sense, the project taught me not only about stacks and queues, but about the mindset of a software engineer - to reason, reflect and refine.

5. Conclusion

This project began as a straightforward implementation task but evolved into a deeper exploration of design, abstraction, and disciplined software reasoning. Through building and refining the DoubleStack and DoubleStackQueue, I learned how small structural choices - such as how space is divided or when data is transferred - can have large consequences for clarity, efficiency and testability

The progression from a fixed-half to a 2*Q internal design was particularly formative. It taught me to identify not only when something works, but when it works for the right reasons. Along the way, I strengthened my understanding of interface contracts, exception handling, and defensive programming, while learning to view testing as an essential tool for validation rather than an afterthought.

Ultimately, this coursework helped me develop a more reflective approach to software design: one that balances correctness with elegance, and specification with insight. The final solution is simple, predictable, and well tested - but more importantly, it represents a tangible step forward in how I think about solving problems in Computer Science.

6. References

[1] University of St Andrews (2025). CS2001-W04-Stacks, CS2001-W04-Queues
[2] (Gamma et al., 1994; Bloch, 2018) – Design patterns and best practices on object-oriented design and coding to interfaces. (Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. & Bloch, J. (2018). Effective Java (3rd ed.). AddisonWesley.)
[3] Baeldung. (2024). How to Implement a Queue Using Two Stacks? (Baeldung on Computer Science). Retrieved from https://www.baeldung.com/cs/queue-two-stacks-simulate .
[4] Baeldung. (2024) Circular Buffer. Retrieved from https://www.baeldung.com/cs/circular-buffer