Neighborhood Library QA Partner Activity

Goal

Today you'll put on your *tester hats*!
Your job is to **test your partner's Neighborhood Library program** — not just to see if it works, but to explore *how it behaves when things go wrong*.

You'll practice:

- Writing clear test cases (inputs → expected outputs)
- Thinking about user behavior
- Seeing why defensive coding (like exceptions) matters

Step 1: Partner Up

Decide who will go first.

- **Tester A:** runs and tests their partner's program first.
- **Tester B:** observes, takes notes, and records results.
  Then switch roles halfway through.

Step 2: Happy Path Tests (everything should work)

#	Test	Steps to Reproduce	Expected Result	Actual Result
1	**View available books**	Run program → choose "Show Available Books."	Displays all books that are *not* checked out.	
2	**Check out a book**	Choose a book ID, enter your name.	Book shows as checked out to that name.	
3	**View checked-out books**	Return to main menu → choose "Show Checked Out Books."	Book appears in the checked-out list.	
4	**Check in a book**	Enter book ID to check in.	Book moves back to the available list.	

# Test	Steps to Reproduce	Expected Result	Actual Result
5 **Exit**	Choose “Exit.”	Program closes cleanly.	

Step 3: Edge Cases and Break Tests

#	Test	Steps to Reproduce	Expected Result	Actual Result
6	**Invalid menu choice**	Type a letter instead of a number at the menu.	Program handles it gracefully (no crash).	
7	**Nonexistent book ID**	Try checking out book ID 99 (or another that doesn’t exist).	Error message, no crash.	
8	**Empty name**	Try checking out but press *Enter* without typing a name.	Program asks again for a valid name.	
9	**Check out a book twice**	Try to check out a book that’s already checked out.	Program prevents duplicate checkout.	
10	**Rapid inputs**	Enter numbers quickly without waiting.	Still reads input correctly (no skipped lines).	

Step 4: Reflection

Discuss with your partner:

1. Which test caused the program to behave unexpectedly?
2. Did any test *crash* the program or freeze it?
3. What could the program do to handle bad input better?
4. **Question to think about:**
   How could we make our program more resilient to user input?
   (Hint: using try/catch, validating input, or checking conditions before acting.)