PROJECT: selectionSort — analysis of the algorithm and infrastructure of metrics

1) Summary
- Main error: premature termination of the external sorting loop due to the `swapped` flag. This leads to incorrect sorting for some inputs.
- Incorrect flag semantics: `swapped` actually means "a new minimum has been found", not "an exchange has been completed".
- Metrics: counting comparisons is correct; counting array accesses is generally plausible, but the methodology must be explicitly documented (accounting for reads and writes).
- Infrastructure: CSV without header, `Scanner` does not close, reverse array generation starts with `n` (not a bug, but it's worth it consciously).
- Tests: There is no test that will catch an early exit (e.g.., `[1, 3, 2]`).

2) Details of the identified problems
2.1) Premature exit from the external loop (critical error)
Fragment (simplified):
- Outer loop: `for (int i = 0; i < n - 1; i++) { ... }`
- Initialized inside `boolean swapped = false;`
- The inner loop updates the `minIndex` and sets `swapped = true` when a smaller element is found.
- After the inner loop is executed:
  - `if (minIndex != i) swap(...)`
  - `if (!swapped) break;`
Problem: for the selection sorting algorithm, the absence of a new minimum at position `i` does not mean that the tail of the array is sorted. An interrupt leaves the array partially unsorted.
An example where the algorithm breaks down: input `[1, 3, 2]`
- At `i = 0`, the minimum is already in place, `swapped` remains `false`, a `break` occurs, the result: `[1, 3, 2]` (unsorted).
Solution: remove the `if (!swapped) break;`. The classic selection sort is not adaptive and does not terminate the cycle prematurely.

2.2) Incorrect semantics of the `swapped` variable name
`swapped` is used as an indicator of "a new minimum has been found", not "an exchange has occurred". This is misleading and has led to erroneous optimization. Preferably rename it to `foundNewMinimum` or delete it completely along with the early release.

3) Metrics and correctness of calculations
3.1) Comparisons (`comparisons')
- Increment at each iteration of the inner loop — corresponds to one comparison `arr[j] < arr[minIndex]'. This is correct for the classical implementation.

3.2) Array accesses (`arrayAccesses')
- The inner loop increments the counter twice for each comparison — corresponds to two reads (`arr[j]` and `arr[minIndex]`).
- In the `swap`, the counter is incremented four times — it corresponds to two reads and two writes. This is a valid model if records are also considered "hits" (should be documented).

- Recommendation: explicitly clarify the methodology — whether only reads or writes are taken into account; add a comment to the code and description in `README.md `.

3.3) Potential differences in methodology
- The current model does not take into account "indirect" requests (for example, receiving `arr.length`), which is normal for practical benchmarking metrics, but it should be recorded in the description.

4) Infrastructure and utilities
4.1) CSV export (class `PerformanceTracker')
- A line is written without a title. Recommendation: when starting the file for the first time, add a header or document the format:
`algorithm,n,runtime_ns,comparisons,swaps,array_accesses'.
- The record format is correct (6 fields) and consistent with the metrics.

4.2) CLI (`BenchmarkRunner`)
- The `Scanner' won't close. Recommendation: use try-with-resources or explicitly call `scanner.close()` after reading the parameters.
- Generating the reverse array: `arr[i] = n - i` creates the range `[n, n-1, ..., 1]`. This is valid, but it differs from the frequently used `[n-1, ..., 0]`; it is important to just consciously use one of the definitions.
- UX: if the input type is selected incorrectly, the array is filled with zeros. Perhaps it is better to request input again or explicitly inform about the use of the default value.

5) Tests
- The test set is good in terms of coverage of boundary cases (empty, one element, already sorted, reverse, duplicates, negative, large random).
- There is a critical lack of a test that catches early termination: add a case like `int[] arr = {1, 3, 2};` And waiting `{1, 2, 3}`. This test will detect the current error.
- The `null` test expects a `NullPointerException'. The current implementation will indeed throw the NPE when accessing `arr.length`, which corresponds to the test.

6) Recommendations for corrections (briefly)
- Remove the early termination of the outer loop: remove the `if (!swapped) break;` block.
- (Optional) Delete/rename the `swapped` variable to avoid incorrect optimizations in the future.
- Add the `int[] arr' case to the tests = {1, 3, 2};` to prevent regressions.
- Document the methodology for calculating `arrayAccesses` (accounting for reads and writes), add a description of the CSV format, and, if necessary, a header to the file.
- Close the `Scanner` or use try-with-resources.

7) Impact on complexity and performance
- The classic selection sort has O(n2) comparisons; removing the early exit does not worsen the asymptotics (it is already O(n2)), but eliminates the logical error. At best, an early exit would save time on a sorted array, but for selection sort, correct adaptivity is usually not applied (unlike insertion sort). Maintaining correctness is more important than hypothetical economy.

8) The result
- After eliminating the early exit, the algorithm will be correct for all test-covered and typical inputs. The recommended small improvements improve the reliability of metrics and the usability of utilities.