The Simple AP® Computer Science A Unit 4 Cheat Sheet

4.1–4.3 Data, Ethics & Arrays

Data Ethics & Data Sets

Ethics: collecting data → privacy & security risks. Keep only needed info; protect it.
Bias: data that leaves out a group → biased results. Conclusions fit only the group sampled.
Data set: a collection of related values; code searches it for patterns & answers questions.

Create & Access

Array (fixed-size list of one type): new int[5] → 5 slots of 0; or {3, 7, 1} (length 3).

Element type	Default
int	0
double	0.0
boolean	false
String / object	null

Index (position) runs 0 → arr.length − 1. arr.length has no parens (it's a field, not a method).
Out of range: arr[arr.length] → **ArrayIndexOutOfBoundsException** at run time.
Alias: int[] b = a; → same array, so b[0] = 9 also alters a[0]. No copy; both names point to it.

Arrays of Objects

new Student[3] → 3 null refs, NOT 3 objects. Method call on null → **NullPointerException**.

⚠ A method gets the array's reference, so it CAN alter the elements in the caller's array.

✓ MCQ: draw the array as boxes w/ indexes and update a box after every line you trace.

4.4–4.5 Array Traversals & Algorithms

Traversals

Indexed for: for (int i = 0; i < arr.length; i++) → read or write arr[i]; can go backward or skip.
Enhanced for: for (int x : arr) → x is a COPY of each value, so x = 0 doesn't alter arr.
Neighbors: stop at i < arr.length − 1, then compare arr[i] & arr[i + 1] (no out of bounds).

Standard Algorithms

Goal	Pattern
Sum / average	total += x
Min / max	start at arr[0]
Count	if (cond) count++
Any / all	return early
Duplicates	nested loop, j = i + 1
Reverse	swap i & len−1−i

Swap (trade 2 values): temp = arr[i]; arr[i] = arr[j]; arr[j] = temp; skip temp → a value is lost.
All-match: return false on the 1st fail; return true only AFTER the loop has checked every item.
Average: (double) total / arr.length; int / int drops the decimal (7 / 2 = 3, not 3.5).
Copy: make a new array of the same length, then copy each arr[i] in a loop (b = a only aliases).

⚠ Off-by-one: i <= arr.length goes out of bounds. Start max at arr[0], not 0 (all-negative data).

✓ FRQ: write the loop header first, then test it on a 1-element array before moving on.

4.6–4.7 Text Files & Wrapper Classes

Reading Text Files

Setup: import java.io.*; & java.util.*; method header adds **throws IOException**, b/c the file may be missing or unreadable at run time.
Open: Scanner in = new Scanner(new File("data.txt")); reads words split by spaces or line breaks, one token at a time.

Scanner method	Returns
hasNext()	true if more
next()	next word
nextInt()	next int
nextLine()	rest of line
close()	frees the file

Loop: while (in.hasNext()) { ... in.next() ... } then in.close(); unknown count → ArrayList.
split: "Ana,17".split(",") → {"Ana", "17"}; Integer.parseInt("17") → 17 (String → int).

⚠ nextInt() leaves the line break, so a nextLine() right after returns "" (an empty String).

Wrapper Classes

Integer & **Double** wrap int & double as objects, so they can go in an ArrayList (objects only).
Autoboxing: int → Integer on its own (list.add(5)).
Unboxing: Integer → int on its own.
Double d = 3; won't compile; write 3.0.
Integer.parseInt(s): String → int value.

⚠ Unboxing null → NullPointerException. Compare Integer objects w/ .equals(), not ==.

4.8 ArrayList Methods

Create

ArrayList (resizable list): ArrayList<String> list = new ArrayList<String>(); → empty, size 0.
Type must be a class: ArrayList<Integer>, never ArrayList<int>, since int is a primitive, not a class.

Methods (on the Java Quick Ref)

Method	Does
size()	# of elements
add(obj)	adds at end
add(i, obj)	inserts; shifts right
get(i)	element at i
set(i, obj)	replaces; returns old
remove(i)	deletes; shifts left

Valid index: 0 → size() − 1. Else **IndexOutOfBoundsException** (crash at run time).
remove(i) also returns the item it removed.

Array vs ArrayList

Feature	Array	ArrayList
Size	fixed	grows
Length	arr.length	list.size()
Read	arr[i]	list.get(i)
Holds	any type	objects

⚠ Every add or remove shifts the later indexes by 1, so trace one step at a time on paper.

✓ Use an ArrayList when you don't know the count up front, e.g. data read from a file.

4.9–4.10 ArrayList Traversals & Algorithms

Traversals

Indexed: for (int i = 0; i < list.size(); i++) → list.get(i); use it when you add or remove.
Enhanced for: for (String s : list) → read only; add or remove inside → **ConcurrentModificationException**.

Removing While Looping

Bug: remove(i) slides the next item into slot i, then i++ skips it, so 2 matches in a row → 1 is missed.
Fix: loop backward (i = list.size() − 1; i >= 0; i−−), so the shift only moves items already checked.

Task	Pattern
Filter	new list; add keepers
Remove all X	backward + remove(i)
Count matches	for-each + counter
Insert in order	find 1st bigger; add(i, x)

FRQ 3: Data Analysis w/ ArrayList

Typical task: a class holds an ArrayList of objects → loop, call getters, compute or filter, return it.
Objects: list.get(i).getScore() calls a method on item i. Compare Strings w/ .equals(), not ==.
Build a result: make a new empty ArrayList, add each match, return it (empty if no matches).
Empty list: get(0) crashes, so check size() > 0 before reading the first item (index 0).

⚠ size() changes as you add or remove, so call it in the loop header, not once before the loop.

✓ FRQ 3: use the getters the class gives you; other classes' private fields are off-limits.

4.11–4.13 2D Arrays

Create & Access

2D array (grid of rows & cols): new int[3][4] → 3 rows, 4 cols, all 0. {{1, 2}, {3, 4}} → 2 × 2.
Rows = g.length. **Cols** = g[0].length. Access g[row][col], always row first, then column.
Row: g[r] is a 1D array, so any 1D array method or loop works on it (e.g. sum(g[r])).

Traversals

Order	Outer	Inner
Row-major	rows	cols
Col-major	cols	rows

Row-major: for (r = 0; r < g.length; r++) for (c = 0; c < g[0].length; c++) → g[r][c].
Column-major: swap the 2 loops (cols outside), but still write g[r][c], never g[c][r].
Enhanced: for (int[] row : g) for (int v : row) → row-major order, read only; v = 0 doesn't alter g.

Algorithms

Row sum: reset sum inside the outer loop.
Diagonal: g[i][i] (square grids only).
Neighbors: check bounds first, e.g. r > 0 before g[r − 1][c], so the edges don't crash.
Col sum: fix c, then loop r down the rows.
Count: nested loops + a counter; add 1 for every g[r][c] that matches the condition.

⚠ Mixing up g.length & g[0].length breaks any grid that isn't square (e.g. 2 rows × 3 cols).

✓ FRQ 4: sketch a small grid, label r & c, and trace one row by hand before you write code.

4.14–4.15 Searching & Sorting

Searching

Linear search: check items one by one; works on any data; return the index, or −1 if not found.
Binary search: SORTED data only; check middle → drop half each time (1,000 → ~10 checks).
Steps: mid = (low + high) / 2; target smaller → high = mid − 1; target bigger → low = mid + 1; stop when low > high (return −1 = not found).
Unsorted data → use linear search.

Sorting

Sort	How	Speed
Selection	min to front	~n^2
Insertion	shift in place	~n^2
Merge	split & merge	faster

Selection: after pass k, the first k items are the k smallest, already in their final spots.
Insertion: after pass k, the first k + 1 items are sorted among themselves, but not final yet.
Trace {5, 2, 8, 1}: Selection pass 1 → {1, 2, 8, 5}; Insertion pass 1 → {2, 5, 8, 1} (only 5 & 2 moved).
Merge sort: fastest of the 3 on big lists, but it needs an extra temp array to merge into.
Best case: Insertion on sorted data shifts nothing; Selection still compares ~n^2 times.

⚠ After 2 passes, Selection's front holds the 2 smallest; Insertion's holds the first 3, sorted.

✓ Binary search needs sorted data: check the question for the word sorted before using it.

4.16–4.17 Recursion

Recursion Basics

Recursive method: calls itself. Needs a **base case** (stops) + a call that moves toward it.
No base case (or no progress toward it) → endless calls → **StackOverflowError** at run time.
Each call has its own variables; returns happen in reverse order (last call finishes first).

Tracing

Example: fact(n): if (n <= 1) return 1; else return n * fact(n − 1); → base case is checked first.

Call	Returns
fact(3)	3 * fact(2) = 6
fact(2)	2 * fact(1) = 2
fact(1)	1 (base case)

Print before vs after the call → output in forward vs reverse order (1 2 3 vs 3 2 1).

Recursive Search & Sort

Base case: reachable for every valid input.
Recursive binary search: check mid, then call again on one half; stop when low > high (−1).
Merge sort: split in half → sort each half → merge the 2 sorted halves. Base case: 1 item.
Two calls per step (like fib) → call count grows fast; fib(4) makes 9 calls in total (draw the tree).

⚠ Trace w/ a call stack: one line per call going down, then fill in returns going up.

✓ Two calls per method → draw a call tree and count its nodes to count the calls.