The Ultimate AP® Computer Science A Unit 4 Cheat Sheet

4.1–4.3 Data, Ethics & Arrays

Data Ethics & Data Sets

Ethics: collecting data → privacy, security & consent risks. Keep only needed fields; protect personal info.

Bias: data set that under-represents a group → biased results & algorithms. Conclusions only fit the population sampled.

Data set = collection of related values; code finds patterns & answers questions. Limits: missing or messy data.

Clean data before use: drop or flag missing values, fix formats; note what the data can't tell you.

Create & Access

Declare: int[] nums; (null until created). **Create**: new int[5] → 5 slots of 0, or **initializer list** {3, 7, 1} (length 3).

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

Fixed size: arr.length (field, no parens) vs str.length() vs list.size(). Valid **index**: 0 → length − 1.

Out of range: arr[arr.length] or arr[−1] → **ArrayIndexOutOfBoundsException** at run time, not compile.

Alias: int[] b = a; → both name the SAME array; b[0] = 9 also alters a[0]. Real copy = new array + loop.

Arrays of Objects

new Student[3] → 3 null refs, NOT 3 objects; fill each w/ new Student(...). Method call on null → **NullPointerException**.

Loop bound: use i < arr.length, never a hard-coded size.

⚠ A method gets a copy of the array REFERENCE: it can alter elements, but arr = new int[3] inside it can't.

✓ MCQ: draw the array as boxes w/ indexes and update a box after every line you trace; no mental math.

4.4–4.5 Array Traversals & Algorithms

Traversals

Indexed for: for (int i = 0; i < arr.length; i++) → read or write arr[i], skip, go backward, compare neighbors.

Enhanced for: for (int x : arr) → x is a COPY of each value; x = 0 doesn't alter arr. Objects: can call mutators on each.

Backward: i = arr.length − 1; i >= 0; i−−. **Neighbors**: stop at i < arr.length − 1, then use arr[i] & arr[i + 1].

Standard Algorithms

Goal	Pattern
Sum / average	total += x; then divide
Min / max	start at arr[0], not 0
Count	if (cond) count++
Any / all	return early on hit/miss
Duplicates	nested loop, j = i + 1
Reverse	swap i & len − 1 − i
Shift / rotate	save end value first

Swap: int temp = arr[i]; arr[i] = arr[j]; arr[j] = temp; skip temp → both slots end up w/ the same value.

All-match: return false inside the loop on the first fail; return true only AFTER the loop ends. **Any-match**: flip both.

Average: (double) total / arr.length; int / int truncates (7 / 2 = 3, not 3.5). Cast before dividing, not after.

Count runs: for (int i = 0; i < n; i++) runs n times; nested n × m loops → body runs n·m times; i += 2 → n/2.

Copy: int[] c = new int[a.length]; then c[i] = a[i] in a loop. Changing c then leaves a alone (no alias).

Default fill: new double[4] → {0.0, 0.0, 0.0, 0.0}.

⚠ Off-by-one: i <= arr.length → out of bounds. Max starting at 0 fails on an all-negative array; start at arr[0].

✓ FRQ: write the loop header first, then test it on a 1-element & empty 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** (file may be missing).

Open: Scanner in = new Scanner(new File("data.txt")); reads tokens split by spaces or line breaks (whitespace).

Scanner method	Returns
hasNext()	true if more to read
next()	next token (String)
nextInt() / nextDouble()	next token as number
nextLine()	rest of current line
close()	frees the file

Loop: while (in.hasNext()) { String w = in.next(); ... } then in.close(); count unknown → store in ArrayList.

split: "Ana,17".split(",") → {"Ana", "17"}; Integer.parseInt(parts[1]) → 17. Parts stay Strings.

Records: one line per object → split each line, parse fields, then list.add(new Student(name, age)).

Missing file → FileNotFoundException (a kind of IOException), so the header must say throws.

⚠ nextInt() leaves the line break; a nextLine() right after returns "" (rest of the old line). Read whole lines, then split.

Wrapper Classes

Integer & **Double** wrap int & double as objects so they fit in an ArrayList (objects only). Both are immutable.

Autoboxing: int → Integer automatically (list.add(5)).
Unboxing: Integer → int (int x = list.get(i) + 1 works).

Integer.MIN_VALUE / MAX_VALUE: safe start values for max & min. **parseInt(s)**, **parseDouble(s)**: String → number.

Double d = 3; fails: int can't box to Double; use 3.0.

⚠ Unboxing null → NullPointerException. Compare Integer objects w/ .equals() or after unboxing; == compares refs.

4.8 ArrayList Methods

Create

ArrayList<String> list = new ArrayList<String>(); → empty, size 0, grows & shrinks. Needs import java.util.ArrayList;

Type param must be a class: ArrayList<Integer>, never <int>. Adding the wrong type → compile error.

Methods (on the Java Quick Ref)

Method	Does
size()	# of elements
add(obj)	append; returns true
add(i, obj)	insert; shifts right
get(i)	returns element at i
set(i, obj)	replace; returns OLD
remove(i)	delete; shifts left

Valid index: 0 → size() − 1; add(i, obj) also allows i = size() (adds at end). Else **IndexOutOfBoundsException**.

remove(i) returns the removed element. ArrayList<Integer>: list.remove(2) removes INDEX 2, not the value 2.

Array vs ArrayList

Feature	Array	ArrayList
Size	fixed	dynamic
Length	arr.length	list.size()
Read	arr[i]	list.get(i)
Write	arr[i] = x	list.set(i, x)
Holds	prims or objs	objects only

Print: System.out.println(list) → [a, b, c] via each element's toString(); an empty list prints [] (not null).

⚠ Trace add & remove one step at a time: each one shifts every later index by 1, so old indexes go stale.

✓ Use an ArrayList when the count is unknown up front: reading a file, filtering, collecting matches.

4.9–4.10 ArrayList Traversals & Algorithms

Traversals

Indexed: for (int i = 0; i < list.size(); i++) → list.get(i); use when you add, remove, or need the index itself.

Enhanced for: for (String s : list) → read only; add or remove inside → **ConcurrentModificationException**.

Build a result: ArrayList<String> out = new ArrayList<String>(); add matches; return out (empty if none).

Removing While Looping

Forward bug: remove(i) shifts the next element into slot i, then i++ skips it → 2 matches in a row, 1 survives.

Fix 1: loop backward (i = list.size() − 1; i >= 0; i−−). **Fix 2**: i++ only when nothing was removed (while loop).

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

FRQ 3: Data Analysis w/ ArrayList

Typical task: class holds an ArrayList of objects → traverse, call getters, compute or filter, return a value or new list.

Elements are objects: list.get(i).getScore() calls the method on element i; compare Strings w/ .equals(), not ==.

Average over a list: sum via getters, then (double) sum / list.size(); check size() > 0 first to avoid ÷0.

Max object: best = list.get(0), then compare getters each pass; return best (or its index if asked).

⚠ size() changes as you add or remove; don't save it in a variable before a loop that alters the list.

✓ FRQ 3: use the getters & method names the class gives you; private fields of another class are off-limits.

4.11–4.13 2D Arrays

Create & Access

int[][] g = new int[3][4]; → 3 rows, 4 cols, all 0. **Initializer**: {{1, 2}, {3, 4}} → 2 rows of 2. String[][] → all null.

Rows = g.length. **Cols** = g[0].length. Access g[row][col]; row always comes first, like reading a table.

Array of arrays: g[r] is a 1D int[] row → pass it to any 1D array method (e.g. sum(g[r])) or loop over it alone.

Rectangular: AP 2D arrays have equal-length rows, so g[0].length is the col count for every row.

Traversals

Order	Outer loop	Inner loop
Row-major	rows (r)	cols (c)
Column-major	cols (c)	rows (r)

Row-major: for (int r = 0; r < g.length; r++) for (int c = 0; c < g[0].length; c++) → g[r][c], left to right.

Column-major: outer c < g[0].length, inner r < g.length; body still g[r][c], never g[c][r]. Reads top to bottom.

Enhanced: for (int[] row : g) for (int v : row) → row-major, read only; v = 0 changes nothing in g.

Algorithms

Row sum: reset sum inside the outer loop. **Col sum**: fix c, loop r. **Diagonal**: g[i][i] (square grids only).

Neighbors: bounds first: r > 0 before g[r − 1][c]; c < g[0].length − 1 before g[r][c + 1]. Use && short-circuit.

Count / find: nested loops + counter, or return true (or the row) the moment g[r][c] matches; else false after both loops.

Copy a 2D array: new int[g.length][g[0].length], then copy each g[r][c]; row = g[r] only aliases a row.

⚠ Swapping g.length & g[0].length → wrong answer or out of bounds on any non-square grid; test w/ 2 rows × 3 cols.

✓ FRQ 4: sketch a 3×4 grid, label r & c, trace one row by hand before writing code; then check edge rows & cols.

4.14–4.15 Searching & Sorting

Searching

Linear search: check each element in order; any data, sorted or not; worst case n checks; return index or −1.

Binary search: SORTED data only; check middle → discard half; ~$\log_2 n$ checks (1,000 items → ~10).

Binary steps: low = 0, high = n − 1, mid = (low + high) / 2; target < arr[mid] → high = mid − 1, else low = mid + 1.

Stop: found at mid, or low > high → not found (−1). **Trace**: list low, mid & high each pass in a small table.

Sorting

Sort	How	Note
Selection	swap min to front	~n^2 compares
Insertion	shift into place	fast if near-sorted
Merge	split, sort, merge	recursive

Selection: after pass k, the first k elements are the k smallest, in final spots. At most n − 1 swaps; compares ≈ $n^2/2$.

Insertion: after pass k, the first k + 1 values are sorted among themselves, not final; inner while shifts bigger ones right.

Merge sort: fastest of the 3 on big data (≈ n log n) but needs a temp array; same work sorted or not.

Best case: insertion on already-sorted data → n − 1 compares, no shifts. Selection still compares ≈ $n^2/2$ times.

Statement count: MCQs ask how many times a line runs → add up inner-loop runs per outer pass.

Selection: {5, 2, 8, 1} → {1, 2, 8, 5} → {1, 2, 8, 5} (no change) → {1, 2, 5, 8}; n − 1 passes for n items.

Insertion: {5, 2, 8, 1} → {2, 5, 8, 1} → {2, 5, 8, 1} → {1, 2, 5, 8}; the last pass shifts 3 values right to free slot 0.

⚠ After 2 passes: Selection front = 2 smallest overall; Insertion front = first 3 values, just sorted.

✓ Binary search on unsorted data can miss the target: check the stem for the word sorted before choosing it.

4.16–4.17 Recursion

Recursion Basics

Recursive method calls itself. Needs a **base case** (stops, no call) + a **recursive call** that moves toward it.

No base case or no progress (e.g. f(n) calls f(n) or f(n + 1)) → infinite recursion → **StackOverflowError**.

Each call gets its own params & locals; returns unwind in reverse order of the calls (last called, first done).

Recursion vs loops: any recursive method can be rewritten w/ a loop; recursion fits split-in-half problems.

Tracing

Example: int fact(int n) { if (n <= 1) return 1; return n * fact(n − 1); } → base case comes first, then the call.

Call	Returns
fact(4)	4 * fact(3) = 24
fact(3)	3 * fact(2) = 6
fact(2)	2 * fact(1) = 2
fact(1)	1 (base case)

Print before vs after the recursive call → output in forward vs reversed order (e.g. digits of 123 → 1 2 3 or 3 2 1).

Recursive Search & Sort

Recursive binary search: params low & high; base: low > high → −1, or arr[mid] == target → mid.

Else branch: target < arr[mid] → recurse on (low, mid − 1); else on (mid + 1, high). Each call halves the range.

Merge sort: split in half → mergeSort each half → merge the 2 sorted halves via a temp array. Base: 1 element.

void recursion: base case just returns; no value.

⚠ Trace w/ a call stack: one line per call going down, then fill in returns from the bottom up; never skip a level.

✓ Two recursive calls per method → draw a call tree and count its nodes to count calls (fib(4) → 9 calls).