The Ultimate AP® Computer Science A Unit 1 Cheat Sheet

1.1–1.2 Programs, Variables & Types

Algorithms & Compiling

Algorithm: step-by-step process to solve a problem. **Compiler**: checks syntax + translates Java source → bytecode; won't run w/ any compile error.

Error	When found	Example
Syntax	Compile time	Missing ;
Logic	Wrong output	+ instead of *
Run-time	Crash (exception)	int ÷ 0, null

Primitive Types (AP subset)

Type	Holds	Example
int	Whole numbers	int n = -7;
double	Decimals	double d = 2.5;
boolean	true / false	boolean ok = true;

Reference type: variable stores a *reference* (address) to an object, e.g., **String**, any class. Default value **null**.
Literal: fixed value in code (**5**, **2.5**, **true**, **"hi"**). **Keywords** (int, class, new, public) can't be variable names.
Strongly typed: Java checks types at compile time → mismatches caught before the program runs.

Declaring Variables

Declare: **type name;** → **initialize**: **name = value;** Both at once: **int score = 0;** (1 type per declaration).
Names: start w/ letter, case-sensitive, no spaces/keywords. Convention: **camelCase** (totalScore, numStudents).
final: constant, can't be reassigned once set. **final double TAX = 0.08;** (name in ALL CAPS by convention).
double d = 7; is legal: stores **7.0** (int widened automatically).
Using a local variable before assigning it → compile error (local vars get no default value, unlike fields).

⚠ A variable's type is fixed once declared. int x = 3.5; won't compile — a double can't go into an int without a cast.

✓ MCQ "which won't compile?" → check types first: mismatched assignment is the #1 compile error.

1.3–1.4 Expressions, Output & Assignment

Arithmetic Operators

+ − * / %. Precedence: *** / %** before **+ −**; left to right at the same level. Use ( ) to override the default order.
Integer division: int / int → int, decimal *truncated* (toward 0). **7 / 2 = 3**, **−7 / 2 = −3**. If either operand is double → double result: **7 / 2.0 = 3.5**, **7.0 / 2 = 3.5**, **(7 + 0.0) / 2 = 3.5**.
Modulus %: remainder. **17 % 5 = 2**, **4 % 7 = 4**. Even test: **n % 2 == 0**. Last digit: **n % 10**. Drop last digit: **n / 10**.
Divide int by 0 → **ArithmeticException** at run time. **n % 0** also crashes. Both compile fine, then fail when run.
Precedence trace: **5 + 10 % 3 * 2** → 10 % 3 = 1 → 1 * 2 = 2 → 5 + 2 = **7** (% then *, left to right; + last).

Expression	Result
10 / 4	2
10 / 4.0	2.5
10 % 4	2
2 + 3 * 4	14
(2 + 3) * 4	20
1 / 2 * 4.0	0.0

Output & Strings in Expressions

System.out.print(x): no newline after. **System.out.println(x)**: adds a newline after the printed output.
String concatenation: + w/ any String → String. Evaluated left to right: **"A" + 1 + 2 = "A12"** but **1 + 2 + "A" = "3A"**. **println(3 + 4 + "x")** → **7x** (adds the ints first). **println("x" + 3 * 4)** → **x12** (* before +). **"x" + 3 + 4** → **x34**.
Escape sequences: **\"** quote, **** backslash, **\n** newline.

Assignment & Input

= assigns the right side to the left variable: **x = x + 1;** is legal (right side evaluated first). **Input**: Scanner etc. reads user data; the input code itself isn't tested.

⚠ 1 / 2 * 4.0 = 0.0, not 2.0. Integer division happens first (1 / 2 = 0) before the double ever appears.

1.5–1.6 Casting, Range & Compound Ops

Casting

Widening (automatic): int → double. **double d = 5;** stores 5.0.
Narrowing (needs cast): **(int) 3.99 = 3**. Casting *truncates* toward 0, never rounds. **(int) −3.99 = −3**.
Round to nearest int: **(int)(x + 0.5)** for x ≥ 0; **(int)(x − 0.5)** for x < 0. e.g., (int)(2.6 + 0.5) = (int) 3.1 = 3.
Cast precedence: cast binds tighter than * /. **(double) 7 / 2 = 3.5** but **(double)(7 / 2) = 3.0** (cast came too late).

Expression	Result
(int) 9.7	9
(double) 9 / 2	4.5
(double)(9 / 2)	4.0
(int)(2.6 + 0.5)	3
(int) -2.7	-2

Range & Precision

int range: **Integer.MIN_VALUE** to **Integer.MAX_VALUE** (−2,147,483,648 to 2,147,483,647). Exceed → **overflow**: wraps around (MAX + 1 = MIN), no error or warning at all.
double round-off: decimals stored approximately. **0.1 + 0.2** ≠ exactly 0.3. Don't compare doubles w/ == .

Compound Assignment

Operator	Same as
x += 3;	x = x + 3;
x -= 3;	x = x - 3;
x *= 3;	x = x * 3;
x /= 3;	x = x / 3;
x %= 3;	x = x % 3;
x++; / x--;	x = x ± 1;

⚠ On AP, use x++ as its own statement only. Tracing code? Update the variable after each line — don't skip steps.

1.7–1.10 APIs, Comments & Methods

APIs & Libraries

Library: collection of prewritten classes. **API**: documentation of what classes + methods do and how to call them.
Package: group of related classes (java.lang auto-imported).
Attributes = data (instance variables). **Behaviors** = methods. A **class** defines both for all of its objects.

Comments

// single line. **/* ... */** multi-line. **/** ... */** Javadoc (documents classes & methods). All are ignored by the compiler.
Precondition: what must be true before a call (method doesn't check it). **Postcondition**: what's true after it runs. FRQs state these: trust them, don't re-check.

Method Signatures

Signature = method name + parameter list (types, order).
public static double area(double r): return type double, 1 double param r; header shows public & static.

Term	Meaning
Parameter	Variable in header
Argument	Value passed in call
void	Returns nothing
Non-void	Must return a value
Overloading	Same name, diff. params

Call by value: arguments are *copied* into parameters. Changing a primitive param inside the method doesn't change the caller's variable; the copy dies w/ the call.

Calling Class (static) Methods

ClassName.methodName(args): e.g., **Math.sqrt(16)**. No object needed. A non-void call's value must be *used* (stored, printed, or in an expression) or it's lost.

⚠ Args must match the signature in number, order, and type. Math.sqrt("16") won't compile (String ≠ double).

✓ Overload Qs: compare parameter TYPES/COUNT only. Return type alone can't distinguish overloads.

1.11 The Math Class

Key Methods (on Quick Ref)

Call	Returns	Example
Math.abs(x)	|x| (int/double)	abs(-4) → 4
Math.pow(b, e)	b^e as double	pow(2,3) → 8.0
Math.sqrt(x)	√x as double	sqrt(9) → 3.0
Math.random()	0.0 ≤ r < 1.0	0.7312...

Static: call on class, no object. **pow** + **sqrt** always return **double** → cast to store in int: **int s = (int) Math.sqrt(n);**
Math.abs keeps the type: int in → int out; double in → double out. **abs(−7) = 7**, **abs(−2.5) = 2.5**, **abs(0) = 0**.

Random Integers

Formula: **(int)(Math.random() * range) + min**, where **range = max − min + 1** (the count of possible values).

Goal	Code
0–9	(int)(Math.random() * 10)
1–6 (die)	(int)(Math.random() * 6) + 1
5–15	(int)(Math.random() * 11) + 5
-3 to 3	(int)(Math.random() * 7) - 3

Cast placement: **(int) Math.random() * 10** always = 0 (cast happens first → 0 * 10). Wrap the product.

Common Uses

Distance: **Math.sqrt(Math.pow(x2 − x1, 2) + Math.pow(y2 − y1, 2))**. **Gap** between 2 values: **Math.abs(a − b)**.
Square check: compare **(int) Math.sqrt(n)** squared to n.
Round to 2 decimals: **(int)(x * 100 + 0.5) / 100.0** (divide by 100.0, not by 100, or the result stays an int).

⚠ Math.random() never returns 1.0, so * 10 gives 0–9.999... → (int) gives 0–9, never 10. Need +1 for 1–10.

✓ Range check: plug min (0.0) and near-max (0.999) into the formula to find the smallest and largest outputs.

1.12–1.13 Objects & Instantiation

Classes vs. Objects

Class: blueprint (defines attributes + behaviors). **Object**: specific **instance** of a class w/ its own state. Many objects per class; each has separate copies of the data.
State: current values of an object's attributes. **Behavior**: what it can do (its methods). Methods can change state.

Creating Objects

new + **constructor**: **Dog d = new Dog("Rex", 3);** Constructor has same name as class, no return type; sets initial state.
Overloaded constructors: same name, diff. param lists. **No-arg constructor**: **new Dog()** → default state.
Arguments must match a constructor signature in number, order, and type, or code won't compile.

References & Memory

Reference variable holds an address, not the object. **Dog d2 = d;** → both point to the *same* object (**aliasing**). A change made via d2 is seen via d too; no copy is ever made.
null: reference points to no object. **Dog d = null;** then **d.bark()** → **NullPointerException** at run time.
Reassign: **d = new Dog();** points d at a new object; old one w/ no references is garbage-collected.

	Primitive	Reference
Stores	Actual value	Address
Copy (=)	Copies value	Shares object
Default	0 / false	null
Compare ==	Values	Same object?

Strings Are Objects

String is a class: **new String("hi")** works, but literals (**"hi"**) are the norm. An unassigned String field is **null**, not "".

⚠ Declaring Dog d; does NOT create an object. Only new does. Calling a method on an unassigned reference fails.

✓ Trace aliasing w/ boxes + arrows: draw each object once, then point every reference variable at it.

1.14–1.15 Instance Methods & Strings

Instance Methods

objectName.method(args): needs an object. Static: **ClassName.method()**. Calling on **null** → NullPointerException. Use the return value.

String Basics

String: object, **immutable**: methods return a *new* String; original unchanged. **s.substring(1);** alone does nothing; use **s = s.substring(1);** to keep the result. Same for **+=**.
Index: 0 to **length() − 1**. "HELLO": H = 0, O = 4 (length 5). Bad index → **StringIndexOutOfBoundsException**.
String literal: **String s = "hi";** (no **new** needed). **Empty string ""**: length 0, not null; methods work on it.

Call (s = "APCSA")	Result
s.length()	5
s.substring(1, 3)	"PC" (end excl.)
s.substring(2)	"CSA"
s.substring(4, 5)	"A" (1 char)
s.indexOf("CS")	2
s.indexOf("Z")	-1 (not found)
s.equals("apcsa")	false (case)
"cat".compareTo("dog")	negative (c < d)

substring(a, b): length = **b − a**. One char at i: **s.substring(i, i + 1)**. **substring(s.length())** = "" (legal).
compareTo: < 0 if s comes first alphabetically, 0 if equal, > 0 if after. Uppercase sorts before lowercase.
indexOf returns the *first* match only. **"Hi " + obj** calls obj's **toString()** automatically (Unit 3 details).

⚠ Use .equals() to compare String contents. == only checks if both refer to the SAME object in memory.

✓ Substring trace: write index numbers under every letter before evaluating; end index is excluded.

Code Tracing & Exam Strategy

Exam Format

Section I: 42 MCQ, 90 min, 55%. **Section II**: 4 FRQs, 90 min, 45%: methods/control, class design, ArrayList data analysis, 2D array. Java Quick Reference given.
Unit 1 = 15–25% of exam. Its skills (types, Strings, Math, method calls) appear inside every FRQ.
MCQ pacing: ~2 min/Q. Trace on scratch paper; skip long traces and come back to them at the end.

Top Unit 1 Traps

Trap	Fix
int / int	Truncates → cast first
s1 == s2	Use s1.equals(s2)
substring end	End index excluded
(int) before *	Cast whole expr.
Ignored return	s = s.substring(1);
"" + 1 + 2	"12", not "3"
null.method()	NullPointerException
Math.pow → int	Cast: (int) Math.pow(...)

Tracing Method

1. Make a table: one column per variable. **2.** Update after every line. **3.** Note type (int vs. double) on each value. **4.** For output, track print vs. println & spaces between items.

FRQ Tips

Use given methods: if part (a) wrote a method, call it in part (b). **Match the header**: return the declared type.
Partial credit: write code even if unsure. Minor syntax slips (missing ;) usually aren't penalized; wrong logic is.

⚠ Don't print when the method says return. Printing instead of returning loses the point, even w/ right value.

✓ Quick Ref lists String + Math methods: learn their exact behavior (return types, end index), not just names.