The Ultimate AP® Computer Science A Unit 3 Cheat Sheet

3.1–3.2 Abstraction & Program Design

Abstraction

Abstraction: reduce complexity by focusing on the main idea; hide details the user of the code doesn't need.
Data abstraction: use data by name & properties, not storage (an account *has* a balance; callers never see it).
Procedural abstraction: name a process (a method) → callers know WHAT it does, not HOW. Split big tasks into small methods → reuse, less complexity.
Parameters generalize a method: deposit(double amt) works for any amount, so 1 method replaces many copies.
Change inside only: rewrite a method's body (e.g., faster) w/o telling users if its signature & purpose stay.

Designing a Class

Spec says	Becomes	Ex
Noun it has	instance var	balance
Verb it does	method	deposit()
Shared by all	static var	count
Never changes	static final	MAX

Instance variable: value unique to each object. **Class variable**: 1 copy shared by every object of the class.
Plan 1st: list attributes (data) & behaviors (methods) in plain words or a diagram before writing any code.

Impact of Program Design (3.2)

System reliability: program does its job as expected under stated conditions; test w/ normal, boundary & bad inputs.
Society: programs affect society, economy & culture; even helpful ones can cause unintended harm.
Intellectual property: reuse open-source code per its license; other code needs permission (often purchase).

⚠ Abstraction isn't secrecy: users need only a method's name, params & result, not its internal steps.

✓ FRQ 2 spec: nouns → instance vars; verbs → methods; values that never change → static final constants.

3.3 Anatomy of a Class

Class Structure

Class = blueprint; **object** = 1 instance made w/ new. Each object gets its own copy of every instance var.
Header: public class BankAccount { ... } holds 3 parts: instance vars, constructors & methods, usually in that order.

Member	Access	Why
class	public	usable anywhere
instance vars	private	encapsulation
constructors	public	others can build
methods	public/private	API vs. helper

Access Modifiers

public: usable from any class. **private**: usable only by code written inside the class that declares it.
Per class, not per object: inside BankAccount, other.balance is legal when other is another BankAccount.
Private helper method: used only inside the class to break up work; outside code can't call it, so it isn't in the API.

Encapsulation

Data encapsulation: keep instance vars private; outside code reads or alters them only through public methods.
Why: the class guards valid state, e.g., withdraw() can refuse amt > balance; outside code can't set balance to −5.
Outside access: acct.balance in another class → compile error b/c private; write acct.getBalance() instead.
Default values: unassigned instance vars: int → 0, double → 0.0, boolean → false, any object (String) → null.

Example

Skeleton: public class Dog { private String name; private int age; public Dog(...) {...} public int getAge() {...} }

⚠ FRQ 2 rubric awards a point for private instance vars; declaring them public (or w/ no modifier) loses it.

✓ Order: private fields → constructor(s) → public methods → private helpers; FRQ code is easier to grade.

3.4 Constructors

Basics

Constructor: same name as the class, NO return type (not even void); runs once per new to set the starting state.
Ex: public BankAccount(String o, double b) { owner = o; balance = b; } → called by new BankAccount("Al", 50.0).
Job: give every instance var a starting value; params are local → copy them into fields or they're lost at the end.

Default & Overloaded

No constructor written → Java supplies a no-arg default constructor; fields get defaults (0, 0.0, false, null).
Write any constructor → no free default; new BankAccount() won't compile unless you also write a no-arg one.
Overloading: several constructors w/ different param lists (number, types or order); Java picks 1 by the args.

Call	Matches
new Pt()	Pt()
new Pt(3, 4)	Pt(int x, int y)
new Pt(p)	Pt(Pt other)
new Pt("3")	error: no String one

Mutable Object Params

Aliasing risk: storing a passed mutable object (e.g., an array) stores the same reference; the caller's edits alter your object.
Fix: store a copy (new array, copy each element). Strings are immutable → safe to store directly in a field.

Traps

void constructor: public void Dog(...) is a METHOD, not a constructor, so new Dog(...) never runs it.
Re-declared field: int age = a; in the constructor makes a LOCAL var, so the field age stays at its default 0.

⚠ name = name; (param & field share a name) assigns the param to itself; field stays null. Use this.name = name.

✓ FRQ 2: the constructor must set every field the spec needs, incl. counters (0) & values copied from params.

3.5 Writing Methods

Method Header

Header: public double getBalance() → access, return type, name, (params). **Signature** = name + param list.

Kind	Returns	Ex
accessor	field value	getName()
mutator	usually void	setAge(int a)
void	nothing	printInfo()
non-void	declared type	isEmpty()

Accessor (getter): returns a field's value; doesn't alter state.
Mutator (setter): alters a field, usually void.

return

non-void: every path must reach return w/ a value of the declared type, or it won't compile ("missing return").
return ends the method at once; code after it in the same block is unreachable & is a compile error.
Return by value: return balance; sends a copy of the primitive; the caller changing that copy doesn't alter the field.
void: no value; return; alone exits early. Its call can't sit in an expression: int x = obj.printInfo(); won't compile.

Calling Methods

Inside the class: call another instance method by name: deposit(5); means the same as this.deposit(5).
Outside: objRef.method(args). Args must match param types & order; int → double widens OK, double → int doesn't.
Ex: public boolean withdraw(double amt) { if (amt > balance) { return false; } balance -= amt; return true; }
Overloaded methods: same name, different param lists; return type alone can't tell 2 methods apart.

⚠ acct.getBalance(); on its own line does nothing visible; store the result in a var, print it or use it in a test.

✓ Read the spec's return type 1st: "returns true if..." → boolean, & every branch must end in a return statement.

3.6 Passing & Returning References

Primitive vs. Reference Params

Param	Method gets	Caller sees?
int, double...	copy of value	no change
obj, mutated	copy of ref	YES
obj, reassigned	new local ref	no change

Always pass by value: Java copies the arg into the param; for objects, the copied value is the REFERENCE (address).
Mutate through ref: p.setX(9) inside the method → caller's object changes b/c both refs point to the same object.
Reassign ref: p = new Pt(0, 0); alters only the local p; the caller's var still points to the old object.

Example Trace

Ex: void m(Pt p, int n) { p.setX(5); n = 5; p = new Pt(); p.setX(7); } → m(a, k) leaves a.x = 5 & k unchanged.
Strings are immutable: s = s + "!" in a method builds a new String; the caller's String is never changed.

Returning References

Return an object: returns a reference, so the caller gets an alias to the same object, not a copy of it.
Leak risk: a getter that returns a private mutable field (e.g., an array) lets outside code alter private data.
Same-class params: a method may read other's private fields when other is the same class: balance > other.balance.

Good Practice

Don't alter a mutable object param unless the spec asks (e.g., a method meant to update or reset it).
null refs: calling a method on a param that holds null → NullPointerException at run time, not at compile time.

⚠ "Swap" via params fails: swap(int a, int b), or reassigning 2 object params, never swaps the caller's variables.

✓ MCQ trace: draw a box for each object & an arrow for each ref; a param is a new arrow to the same box.

3.7 Class Variables & Methods

static Variables

Class (static) variable: 1 copy shared by ALL objects; it belongs to the class, not to any single object.
Ex: private static int count = 0; the constructor does count++; → count = # of objects created so far in the program.
Access: ClassName.var when public (e.g., Math.PI); every object of the class sees the same value.
static final constant: public static final int MAX = 100; can't be reassigned; NAMED IN CAPS by convention.

	instance	static
copies	1 per object	1 per class
called via	obj.m()	Class.m()
uses this?	yes	no
can use	both kinds	static only

static Methods

Class method: header has static; call it as ClassName.method(), e.g., Math.sqrt(x); no object needed.
No instance access: a static method can't use instance vars or call instance methods directly; compile error.
Instance methods CAN read & alter static vars & call static methods of their class; the reverse isn't allowed.
Utility class: all-static helpers, e.g., public static double area(double r) in a Geometry class; never instantiated.

Patterns & Traps

Shared change: a.count++ & b.count++ hit the SAME var; after both, a.count & b.count each read 2.
Unique IDs: private static int nextId = 1; in the constructor: id = nextId; nextId++; → objects get ids 1, 2, 3 in creation order.

⚠ Resetting a static var in the constructor (count = 0;) wipes the shared total every time a new object is made.

✓ Ask "does this value belong to each object or to the whole class?" → instance var vs. static var.

3.8–3.9 Scope, Access & this

Scope

Scope = where a var can be used. **Local var**: declared in a method, constructor or block; exists only there.

Variable	Usable in
instance var	whole class
parameter	its method/constructor
local var	declaration → end of block
for-loop var	that loop only

Local vars can't be public/private; they get no default, so assign one before use or it won't compile.
Shadowing: a local var or param named like a field hides that field in the method; the bare name = the local.
Ex: private int x; void set(int x) { x = x; } → the param is assigned to itself; the field x is unchanged.

this Keyword

this = reference to the current object: the one whose method or constructor is running (the object before the dot).
Fix shadowing: this.x = x; → the left side is the field, the right side is the param of the same name; field gets its value.
Pass self: other.addFriend(this); sends the current object as an argument to another object's method.
Compare: this == other → same object; this.balance > other.balance compares the 2 objects' fields.
Static context: this can't appear in a static method b/c there's no current object; compile error.

Access Recap

private members: visible only in their own class's code.
public: visible wherever the object is.

⚠ int total = 0; inside a method, when total is already a field, makes a local copy; the field never updates.

✓ When names collide, write this.field on the left side; when they don't, this is optional (but always legal).

FRQ 2: Class Design Playbook

The Task

FRQ 2 (Class Design, 7 pts): write a whole class from a spec & a table of example calls; may use a given class.
Example table: each row = a call & its return value or state change; it reveals hidden fields (counters, last value).

Checklist

Part	Must have
header	public class Name
fields	private, right types
constructor	params match calls
methods	exact names & types
logic	if/loops per spec

Match the calls: if the table shows new Game("Al", 3), the constructor takes (String, int) in that order.
Return type from use: result shown as true/false → boolean; text → String; used in math → int or double.

Hidden State

Extra fields: a method that "returns # of calls so far" or "the last value" needs a field you add yourself.
Ex: guess(int g) returns "higher", "lower" or "correct"; target is set in the constructor; tries++ on each call.

Point Savers

return, don't print: if the spec says returns, System.out.println instead of return earns no return point.
No extra params: don't pass fields into your methods; instance methods already see their own instance vars.
Use given methods: call the provided class's methods as documented; don't rewrite them or reach into their fields.

⚠ Unrequested printing or other side effects can cost a point; write only what the prompt asks for.

✓ Java Quick Reference (in Bluebook) lists String & Math methods; spend your time on logic, not syntax lookups.