The Simple 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 & hiding details the user doesn't need.
Procedural abstraction: a method's name says WHAT it does; callers needn't know HOW.
Data abstraction: use data by its name & properties, not by how it's stored inside.
Change inside only: make a method faster w/o telling users if its signature stays.
Parameters make a method reusable: deposit(double amt) works for any amount.

Designing a Class

Spec says	Becomes
Noun it has	instance var
Verb it does	method
Shared by all	static var

Instance variable: each object has its own value.
Class variable: 1 copy shared by all objects.

Impact of Program Design (3.2)

System reliability: the program works as expected; test it w/ normal, boundary & bad inputs.
Society: programs affect society, economy & culture; can cause unintended harm.
Intellectual property: reuse open-source code per its license; other code needs permission.

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

✓ FRQ 2 spec: nouns → instance vars; verbs → methods. Plan both before you write code.

3.3 Anatomy of a Class

Class Structure

Class = blueprint; **object** = 1 instance made w/ new. Each object gets its own instance vars.
Parts: public class BankAccount { ... } holds instance vars (data), constructors (setup) & methods (behavior), in that order.

Member	Access
class	public
instance vars	private
constructors	public
helper methods	private

Access Modifiers

public: usable from any class. **private**: usable only by code inside the class that declares it.
Per class: inside BankAccount, code may use other.balance of another BankAccount.

Encapsulation

Data encapsulation: keep instance vars private; outside code uses public methods to read or alter them, so the class guards its data.
Outside access: acct.balance in another class → compile error b/c private; use acct.getBalance().
Default values: unassigned fields: int → 0, double → 0.0, boolean → false, objects (String) → null.

⚠ FRQ 2 rubric gives a point for private instance vars; making them public loses that point.

✓ Order: private fields → constructor(s) → public methods → private helpers; easy to grade.

3.4 Constructors

Basics

Constructor: same name as the class, NO return type (not even void); runs on new.
Ex: public Dog(String n, int a) { name = n; age = a; } → called by new Dog("Rex", 3).
Job: give every instance var a starting value; copy each param into its field or it's lost.
Strings are immutable (can't change), so storing a String param directly in a field is safe.

Default & Overloaded

No constructor written → Java adds a no-arg one; fields get defaults (0, 0.0, false, null).
Write any constructor → no free no-arg one; new Dog() won't compile unless written.
Overloading: several constructors w/ different param lists; Java picks 1 by the args passed.

Call	Matches
new Pt()	Pt()
new Pt(3, 4)	Pt(int x, int y)
new Pt("3")	compile error

Traps

void constructor: public void Dog(...) is a METHOD, so new Dog(...) never runs it.
Re-declared field: int age = a; in the constructor makes a LOCAL var, so field age stays at 0.

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

✓ FRQ 2: the constructor sets every field the spec needs, incl. counters that start at 0.

3.5 Writing Methods

Method Header

Header: public double getBalance() → access, return type, name, (params) in that order.

Kind	Ex
accessor (getter)	getName()
mutator (setter)	setAge(int a)
void	printInfo()

Accessor: returns a field's value; doesn't alter state. **Mutator**: 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.
return ends the method at once; code after it in the same block is unreachable (compile error).
void: returns nothing; int x = obj.printInfo(); won't compile b/c there's no value to store.

Calling Methods

Overloaded methods: same name, different param lists; a different return type alone isn't enough.
Inside the class: call another method by name: deposit(5); means this.deposit(5).
Outside the class: objRef.method(args). Args must match the param types & their order.
Ex: boolean withdraw(double amt) { if (amt > bal) return false; bal -= amt; return true; }

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

✓ Read the return type 1st: "returns true if..." → boolean, & every branch must return a value.

3.6 Passing & Returning References

Primitive vs. Reference Params

Param	Caller sees change?
int, double...	no (copy of value)
obj, mutated	YES (same object)
obj, reassigned	no (local ref only)

Always pass by value: Java copies the arg; for objects the copy is the REFERENCE (address).
Mutate through ref: p.setX(9) in the method → caller's object changes; both refs share 1 object.
Reassign ref: p = new Pt(0, 0); alters only the local p; the caller still has the old object.

Example Trace

Ex: void m(Pt p, int n) { p.setX(5); n = 5; } → m(a, k) leaves a.x = 5 & k unchanged.

Returning References

Return an object: returns a reference, so the caller gets an alias to the same object, not a copy.
Strings are immutable (can't be changed): s = s + "!" makes a new String; caller's is unchanged.
Leak risk: a getter that returns a private array lets outside code alter private data.

Good Practice

Don't alter a mutable object param (one w/ setters) unless the spec asks you to update it.

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

✓ MCQ trace: draw a box for each object & an arrow for each reference; a param is a new arrow.

3.7 Class Variables & Methods

static Variables

Class (static) variable: 1 copy shared by ALL objects; it belongs to the class, not to 1 object.
Ex: private static int count = 0; constructor does count++; → count = # of objects made so far.
static final constant: public static final int MAX = 100; can't be reassigned; NAMED IN CAPS.

	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 instance access: a static method can't use instance vars or instance methods; compile error.
Instance methods CAN use static vars & call static methods of their class; not the reverse.

Patterns & Traps

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

⚠ Setting count = 0; in the constructor resets the shared total each time a new object is made.

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

3.8–3.9 Scope, Access & this

Scope

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

Variable	Usable in
instance var	whole class
parameter	its method
local var	rest of its block
for-loop var	that loop only

Local vars can't be public/private & get no default value; assign one before using them.
Shadowing: a param named like a field hides it: set(int x) { x = x; } leaves field x as is.

this Keyword

this = the current object: the one whose method or constructor is running (the object before the dot).
Fix shadowing: this.x = x; → left side is the field, right side is the param w/ the same name.
Pass self: other.addFriend(this); sends the current object as an argument to another method.
Static context: this can't be used in a static method b/c there's no current object.

Access Recap

private: visible only in its own class's code. **public**: visible anywhere the object is used.

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

✓ When a param & field share a name, write this.field on the left side of the assignment.

FRQ 2: Class Design Playbook

The Task

FRQ 2 (Class Design, 7 pts): write a whole class from a spec & a table of example method calls.
Example table: each row = a call & its result; reveals hidden fields (counters).

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("AI", 3), the constructor takes (String, int).
Return type from use: true/false → boolean; text → String; used in math → int/double.

Hidden State

Extra fields: "returns # of calls so far" needs a counter field that you add yourself.

Point Savers

return, don't print: if the spec says returns, println instead of return earns no return point.
No extra params: methods already see their own instance vars, so don't pass fields in.

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

✓ The Java Quick Reference lists String & Math methods; spend your time on logic, not syntax.