Class

2

# CSC 240 Computer Science III

## OOP Foundations · Design · Toward Chapter 9

Dr. Si Chen (schen@wcupa.edu)

Tue / Thu · 9:30 – 10:45 AM · Anderson Hall 315
Fall 2026 · Week 1 · Thursday, August 27, 2026
wcu.ninja/csc240

# Recap from Tuesday

- CSC 240 = **object-oriented design**: objects bundle **attributes + methods**; **encapsulation** hides data behind an interface.
- The shift: from **procedures operating on shared data → objects that own their data**.
- The diagnostic checked your **141/142 foundations** — today we make the shakiest ones solid.
- Roadmap: Ch. 9 (text) → 10 (inheritance) → 11 (exceptions) → 18 (generics) → 20 (linked lists).

PART 1

# OOP foundations, made solid

The mental model everything in CSC 240 is built on

# Class vs. object

## Concept

- A **class** is a blueprint: **fields** (state) + **methods** (behavior).
- An **object** is one concrete instance made with `new`.
- Many objects, one class — each with its **own field values**.
- From the diagnostic: `Bulbasaur` is the class; each captured one is an object.

## In Java

```
public class Dog {
  private String name;   // field
  public Dog(String n){  //
constructor
    name = n; }
  public String speak(){ // method
    return name + " woofs"; }
}
Dog d = new Dog("Rex");  // object
```

# References vs. values — the #1 source of bugs

```
int a = 5;
int b = a;      // b copies the VALUE 5
b = 99;
System.out.println(a);      // 5   (a
unchanged)

int[] x = {1, 2, 3};
int[] y = x;    // y copies the REFERENCE
(same array!)
y[0] = 99;
System.out.println(x[0]);   // 99  (x sees
it too)

Dog p = new Dog("Rex");
Dog q = p;      // q and p point to the
SAME Dog
```

- **Primitives** (`int`, `double`, `boolean`, `char`) hold a **value**.
- **Objects & arrays** hold a **reference** — an arrow to data on the heap.
- Copying a reference does **not** copy the object; both names share it.
- This is diagnostic Q3 — and why `copy()` and `equals()` matter.

*Draw it: variables are boxes; references are arrows to objects on the heap.*

# == vs .equals() and arrays recap

## Equality

- `==` on objects compares **references** (same object?).
- `.equals()` compares **contents** — if the class defines it.
- `String` literals can share storage, so `==` sometimes *looks* right, then fails.
- Rule: **use `.equals()` for objects**, `==` for primitives.

## Arrays

- `int[] a = new int[5];` — indices `0..4`.
- `a.length` is the size (no parentheses).
- Loop bound is `i < a.length` — **not** `<=` (off-by-one!).
- Arrays are objects → passed by reference into methods.

# Review: static methods (class-level utilities)

```
public class MathUtil {
    // static: called on the CLASS, no
object needed
    public static int square(int n) {
        return n * n;
    }
}

int r = MathUtil.square(5);          // 25
— no "new"
String s = Arrays.toString(nums);    //
library static method
double p = Math.pow(2, 10);          //
Math is all static
```

- `static` methods belong to the **class**, not to any object — call them as `Class.method(...)`.
- No `new` needed; used for **utility classes** like `Math` and `Arrays`.
- A static method can only touch **static fields**, not instance fields.
- You have used these already (`Math.pow`, `Arrays.toString`) — now you know why.

*Instance methods need an object; static methods do not. Know which you are writing.*

# Review: ArrayList — the resizable array

```
import java.util.ArrayList;

ArrayList<String> names = new
ArrayList<String>();
names.add("Ada");         // grows
automatically
names.add("Alan");
names.remove("Ada");      // shrinks
automatically
System.out.println(names.get(0));   //
Alan
System.out.println(names.size());   // 1

// <String> = the element type it will
hold
```

- An `ArrayList` is like an array that **grows and shrinks** automatically.
- `<String>` in angle brackets sets the **element type** (this is a *generic* — Ch. 18).
- It stores **objects only** — not primitives like `int` directly.
- That restriction leads straight into today's bridge: **wrapper classes**.

PART 2

# What makes 240 different: engineering discipline

You can make code work. Now: make it planned, correct, and tested.

# Design before you type

### Milestones

Break a project into checkpoints, each runnable.

M1: class compiles + fields.

M2: one method works end to end.

### Checklists

Turn the spec into concrete, checkable items.

Tick them off; nothing silently forgotten.

### Incremental build

Get a tiny version working, then grow it.

Never write 200 lines before running.

Compile & test after each small step.

# Test systematically — do not guess

## Testing

- Write **test cases** before or alongside the code.
- Cover the **normal**, **edge**, and **error** cases.
- For `setLevel`: 1, 15, 16, 31, 32, and a huge number.
- A test is a tiny `main` that prints expected vs. actual.

## Debugging

- When it breaks, use the **debugger**, not guesswork:
- inspect **object state**, **dynamic type**, the **call path**.
- Reproduce with the **smallest example** that still fails.
- Change **one thing at a time**; re-run; keep evidence.

# Reviewing AI-generated code — spot the design flaws

```
public class Account {
    public double
balance;                // flaw 1

    public boolean equals(Account a)
{   // flaw 2
        return this.balance == a.balance;
    }
}
// An AI produced this. It COMPILES. Is it
good design?
```

- **Flaw 1 — no encapsulation:** a `public` field lets anyone write `acct.balance = -999`. Make it `private` with methods.
- **Flaw 2 — broken equals:** wrong signature (`Account`, not `Object`), so it does **not** override `Object.equals` — collections ignore it. And `==` on `double` money is unsafe.
- It **compiles and looks right** — but the **design** is wrong. That is precisely what AI misses.
- Your CSC 240 job: **review, catch, and fix** these. Design judgment is the durable, human skill.

*AI is fluent, not thoughtful. Encapsulation and correct equals are your call, not the model's.*

# Design exercise: extend Bulbasaur

IN-CLASS ⏱ 8 min · discuss

- No IDE needed — sketch on paper, discuss with a neighbor.
- Recall: `Bulbasaur` has `id` (1/2/3) and `level`; `setLevel` evolves it.
- **1. Checklist:** write the requirement list for `feed(int xp)` — raises `level` by `xp`, then re-evaluates evolution.
- **2. Test cases:** which levels would you test `feed` with, and why (boundaries)?
- **3. equals():** two Bulbasaur objects are "equal" when… exactly what? Which fields?

Hints:

• feed should REUSE setLevel's evolution logic (don't duplicate — DRY).

• Boundary tests: crossing 16 and 32 (15→16, 31→32) and a no-evolve case.

• equals(): compare BOTH id and level.

These are the exact ideas Ch. 10 formalizes with overriding & Object.

PART 3

# Bridge to Chapter 9: wrapper classes

Why an ArrayList of numbers needs a little help — our first topic next week

# The problem, and the fix

## Why

- `ArrayList` stores **objects**, not primitives.
- `ArrayList<int>` is **illegal** — `int` is not an object.
- But we often want a list of numbers!
- How do we bridge primitives ↔ objects?

## Wrapper classes

- **Wrapper classes** "wrap" a primitive in an object:
- `int` → `Integer`, `double` → `Double`, `char` → `Character`, `boolean` → `Boolean`.
- `ArrayList<Integer> nums = new ArrayList<>();`
- `nums.add(5);` — Java **auto-boxes** the 5 into an `Integer`.

# Before next class (Thu, Sep 3)

- Install & verify **IntelliJ IDEA** — write a small class with a constructor, one method, and a `main` that tests it.
- Re-solve any **diagnostic** question that felt shaky — references & `.equals()` especially.
  - Read **Gaddis Chapter 9** (text processing) — our next class (**Thu, Sep 3**) begins it.
- Course site: **wcu.ninja/csc240**. **HW1** is assigned next class.