Class

7

CSC 240 Computer Science III

Chapter 10: Superclass/Subclass Relationships

Dr. Si Chen (schen@wcupa.edu)

Tue / Thu · 9:30 – 10:45 AM · Anderson Hall 315
Fall 2026 · Week 4 · Thursday, September 17, 2026
wcu.ninja/csc240

Recap from Tuesday

`extends` makes a **subclass** that automatically has every **public** member of its **superclass**.

`super(...)` must be the **first line** of a subclass constructor - it builds the superclass part first.

private members exist in every subclass object but are not name-accessible there; **protected** members ARE directly accessible inside subclass code; **public** members are open to everyone.

The IS-A test: "an Eevee IS A Pokemon" - if that reads naturally, inheritance is probably the right tool.

PART 3

Multi-level hierarchies

Inheritance chains: Jolteon extends Eevee extends Pokemon

Why the fun data matters: Eevee has nine forms

Tuesday: `Eevee extends Pokemon` - **one level**.

Today: `Jolteon extends Eevee` - now Jolteon inherits from Eevee, which itself inherits from Pokemon. Jolteon IS-A Eevee, and (transitively) IS-A Pokemon too.

Construction always runs **top of the chain down**: `Pokemon()` runs, then `Eevee(...)`, then `Jolteon(...)` - each `super(...)` call hands off to the level above it.

A real evolution line (`Eevee` -> `Vaporeon`/`Jolteon`/`Flareon`/...) is a perfect real-world example of exactly this multi-level shape.

Same chart from Class 5 - now read it as a class hierarchy, not just a game mechanic.

The three-level chain in code

```
public class Jolteon extends Eevee {

    public Jolteon(int id, String name,
            int hp, int attack,
            int defense, int speed) {
        // Eevee's constructor runs first,
        // which runs Pokemon's constructor
        // first. Chain goes top-down.
        super(id, name, hp, attack,
              defense, speed);
        useStone("Thunder Stone");
    }

    // Only Jolteon has this - not
    // Eevee, not Pokemon.
    public void thunderShock() {
        System.out.println(getName()
            + " used Thunder Shock!");
    }
}
```

`getName()` and `getHp()` come from **Pokemon** (two levels up); `useStone(...)` comes from **Eevee** (one level up); `thunderShock()` is brand new to **Jolteon**.

Nothing stops you from going a 4th, 5th level deep - but in practice, most real hierarchies stay shallow (2-3 levels) because very deep chains get hard to reason about.

Full listing: code/class07_inheritance/Jolteon.java.

PART 4

is-a vs. has-a

Two different ways classes relate to each other

A Trainer is NOT a kind of Pokemon

Trainer.java (composition)

```
public class Trainer {
  private String name;
  private ArrayList<Pokemon> team
      = new ArrayList<>();

  public void catchPokemon(
        Pokemon p) {
    team.add(p);
  }
}
```

is-a vs has-a

A `Trainer` does not `extends Pokemon` - a Trainer is not "a kind of" Pokemon at all.

Instead, a Trainer **HAS** Pokemon: it just holds a list of them as a field. This is called **composition** (the "has-a" relationship).

Quick test: does "X IS A Y" sound natural? "A Trainer is a Pokemon" - **no**. "A Trainer has Pokemon" - **yes**. That tells you composition is the right shape here, not inheritance.

Both `extends` (is-a) and "holds a field of that type" (has-a) are normal, everyday OOP tools - picking the right one for the relationship is the actual skill, not memorizing syntax.

PART 5

protected, one more time

Going deeper: when protected is (and is not) the right call

In-class: fix the access modifier

IN-CLASS 8 min

`WildPokemon.java` does **not compile**. It tries to use Pokemon's `id` and `name` fields directly inside a subclass method - but those are `private` in `Pokemon.java`.

Two valid fixes - discuss with your neighbor which is the better design:

(A) Change `id`/`name` in `Pokemon.java` to `protected`, so `WildPokemon` can use them directly.

(B) Leave them `private`, and have `WildPokemon` call the existing `getId()`/`getName()` instead.

Checklist: try compiling first and read the exact error · try fix (A) · revert · try fix (B) · decide which you would ship and why.

```
javac WildPokemon.java
Pokemon.java

// error: id has private
// access in Pokemon

// Fix A: loosen access
protected int id;

// Fix B: use the getter
"..." + getId() + "..."
```

protected: the honest tradeoffs

Pro: a subclass that needs to touch a field often (like `Eevee` bumping `hp` in a loop) avoids getter/setter overhead and reads cleanly.

Con: it weakens encapsulation for the WHOLE family of subclasses, forever - any future subclass can also reach in and set the field to something invalid, with no validation.

Middle ground: keep the field `private`, but add a `protected` method that changes it safely (e.g., a `protected void boostStats(int amount)` that validates before applying).

Default to `private`. Reach for `protected` only when you have a concrete subclass that needs it - not "just in case."

Coming next week (Sep 22, 24): constructors, overriding, Object

How `super(...)` interacts with **overloaded** constructors (a superclass with more than one).

Overriding: a subclass replacing a superclass method's behavior (you already saw a preview - `Eevee.toString()` calling `super.toString()`).

Every class secretly extends `Object` - where `toString()`, `equals()`, and `hashCode()` actually come from.

This is a preview, not new material for today - the goal is just so Thursday-to-Tuesday does not feel like a jump.

Before next class (Tue, Sep 22)

Read **Gaddis Chapter 10 §10.3** (constructors in inheritance, an early look at overriding).

Bring one more real-world **is-a** example AND one **has-a** example to class.

Check D2L for your **Quiz 1** and **HW1** feedback if you have not already.

Course site: **wcu.ninja/csc240** - Class 7 code zip is posted.

Sources

Gaddis, Starting Out with Java, 4th ed., Ch. 10 §10.1-10.3 (inheritance, superclass/subclass, protected, constructors preview).

Jolteon.java, Trainer.java, WildPokemon.java: built on the CSC 240 Fall 2026 Pokédex capstone and Class 6 inheritance demos.

Eevee evolution chart: continues the bridge slide from Class 5 (Dr. Si Chen, CSC 240 Spring 2019 Class 3 deck; species data from PokeAPI).

Java SE 17 API: Object, access control (Java Language Specification §6.6, §8.1.4 on superclasses).