

Class

8

CSC 240 Computer Science III

Chapter 10: Abstract Classes & Interfaces

Dr. Si Chen (schen@wcupa.edu)



Tue / Thu · 9:30 – 10:45 AM · Anderson Hall 315
Fall 2026 · Week 5 · Tuesday, September 22, 2026
wcu.ninja/csc240

Warm-up: predict the output

- Quick, out loud, no writing - look back at Thursday's `Shape s = new Circle(); s.draw();`.
- Q1: why does `s.roll()` not compile, even though `s` really is a `Circle` underneath?
- Q2: `Shape s = new Circle();` - which constructor runs first, `Shape()` or `Circle()`?
- Q3: `Circle()`'s first line is `super();`. What would change if you deleted that line?

TODAY

From "Can Override" to "Must Override"

Every Shape subclass so far chose to override draw(). Today: what if the parent class should never be usable on its own at all - and what if a capability needs to reach across totally unrelated classes?

Recap: reference type vs. actual object type

```
Shape s = new Circle();  
s.draw();           // OK  
  
// s.roll();  
// compile error - Shape  
// has no roll() method  
  
Circle c = (Circle) s;  
c.roll();           // OK now
```

- `draw()` resolves to Circle's version - dynamic binding, same as Thursday.
- `s.roll()` won't compile: the **reference type** (Shape) decides what you're allowed to call, not the object underneath.
- Cast to Circle to unlock it. You'll do this exact move again with `MegaEvolvable` and `Resizable` today.

Full code: 01_review_reference_vs_object/

Recap: constructors are not overridden

```
Shape s = new Circle();  
  
// prints, in this order:  
Shape constructor  
Circle constructor  
  
// Circle() begins with:  
super();
```

- Construction always runs top of the chain **down**: Shape() first, then Circle() - opposite of how method calls resolve.
- `super()`; triggers the parent constructor. Omit it and Java inserts an identical implicit call - the rule is “first line, no arguments” either way.
- Keep this order in mind today: an abstract class still HAS a constructor that runs, even though you can never new one directly.

Full code: 02_review_constructor_chain/

Problem: what does new Shape() even mean?

```
public class Shape {  
    public void draw() {  
        System.out.println(  
            "Drawing a shape");  
    }  
}  
  
Shape s = new Shape();  
s.draw();  
// ...but what IS a shape  
// with no size or sides?
```

- Every subclass so far (Circle, Rectangle, Triangle) overrides draw() to be useful - a bare Shape.draw() was always a placeholder nobody should really call.
- Same story for `area()`: there's no formula for “the area of a generic shape.”
- Today's fix: make Shape unable to be built directly, and FORCE every subclass to supply its own draw() and area().

This was last class's Shape.java.

Abstract classes: must override

```
public abstract class Shape {  
  
    public abstract void draw();  
    public abstract double area();  
  
    public void describe() {  
        System.out.printf(  
            "area %.2f", area());  
    }  
}
```

- **abstract** on the class: you can never write `new Shape()` - only `new Circle()`, `new Rectangle()`, ...
- **abstract** on a method: no body, just a signature ending in `;`. Any subclass that skips it won't compile.
- `describe()` is a normal, concrete method living right next to the abstract ones - abstract classes can freely mix both kinds.

Full code: 03_abstract_shape/

In-class: read the compiler's error

```
Shape[] shapes = {  
    new Circle(3),  
    new Rectangle(4, 5),  
    new Triangle(6, 2)  
};  
  
for (Shape s : shapes) {  
    s.draw();  
    s.describe();  
}  
  
// now uncomment this line:  
// Shape s2 = new Shape();
```

- Run Main.java as-is first - it works, because every element in the array is a concrete subclass.
- Now uncomment `Shape s2 = new Shape();` and recompile.
- Read the exact error out loud to your neighbor - that sentence IS the definition of “abstract class.”

Starter: 03_abstract_shape/

Interfaces: a capability, not a family

```
public interface Resizable {  
    void resize(double factor);  
}  
  
public class Circle extends Shape  
    implements Resizable {  
    ...  
    public void resize(double f) {  
        radius *= f;  
    }  
}
```

- An interface is a pure CONTRACT: signatures only, no bodies, no fields - just a promise every implementer keeps.
- A class extends only **one** superclass, but can implement **many** interfaces at once.
- Remember [MegaEvolvable](#) from Thursday's Pokemon bonus? Same idea - Glaceon and Sylveon “can-do” mega evolution without touching the extends chain at all.

Full code: 04_interface_resizable/

Abstract class vs. interface

abstract class Shape

- **is-a**: Circle IS-A Shape, permanently, one family.
- **single** inheritance: a class extends only one class.
- can hold fields, a constructor, AND concrete methods like describe().
- use it for: a real family of types sharing state and code.

interface Resizable

- **can-do**: any unrelated class can opt in, no family required.
- **multiple**: implements Resizable, Comparable, ... all at once.
- no fields, no constructor, no bodies (in this course) - signatures only.
- use it for: an optional capability, like MegaEvolvable.

In-class: give Vaporeon the MegaEvolvable capability

```
public class Vaporeon
    extends Eevee {

    // TODO 1: implements
    //   MegaEvolvable
    // TODO 2: boolean field
    //   megaEvolved
    // TODO 3: megaEvolve()
    //   hp += 50; def += 30;
    // TODO 4: revertMega()
    // TODO 5: isMegaEvolved()
}
```

- [Glacéon.java](#) in the same folder already implements [MegaEvolvable](#) - use it as your model.
- [Demo.java](#) compiles and runs the whole time; Vaporeon just gets silently skipped by instanceof MegaEvolvable until you finish.
- Stuck?
[06_pokemon_megaevolve_solution/](#) has the finished version.

Starter: 05_pokemon_megaevolve_exercise_starter/

Full circle: Resizable and MegaEvolvable are the same idea

```
interface Resizable {  
    void resize(double f);  
}  
  
interface MegaEvolvable {  
    void megaEvolve();  
    void revertMega();  
    boolean isMegaEvolved();  
}
```

- Same relationship, two different domains: Circle “can resize”, Vaporeon “can mega evolve” - neither has anything to do with being a Shape or a Pokemon.
- This is the real payoff of interfaces: bolt a capability onto classes from completely different families, as long as they agree to the contract.
- `instanceof` + cast unlocks it either way - same pattern in [04_interface_resizable](#) and just now with [MegaEvolvable](#).

Full code: [class08_abstract_interfaces/](#)

Before next class (Thu, Sep 24)

- Read Gaddis Chapter 10 (abstract classes section). Curious about interfaces in more depth? That's Chapter 12 - today was just the shape of it.
- Homework 2 (Chapter 10) will be assigned next class, Thursday Sep 24.
- Course site: wcu.ninja/csc240 - Class 8 code (review + abstract classes/interfaces + Pokemon exercise) is posted.

Sources

- Gaddis, Starting Out with Java, 4th ed., Ch. 10 (abstract classes) and Ch. 12 (interfaces preview).
- Shape/Circle/Rectangle/Triangle (code/class08_abstract_interfaces/): today's primary example set for abstract classes and interfaces.
- Vaporeon.java, Glaceon.java, Sylveon.java, MegaEvolvable.java: built on the CSC 240 Fall 2026 Pokedex capstone and Class 7 Pokemon extension.
- Java SE 17 API: abstract classes and interfaces (Java Language Specification §8.1.1.1, §9).