

Class

11

CSC 240 Computer Science III

Comparators, Anonymous Classes & Lambdas

Dr. Si Chen (schen@wcupa.edu)



Tue / Thu · 9:30 – 10:45 AM · Anderson Hall 315
Fall 2026 · Week 6 · Thursday, October 1, 2026
wcu.ninja/csc240

Warm-up: Comparable, one more time

- Since Class 9, `Shape` implements `Comparable<Shape>`, so `Arrays.sort(shapes)` orders shapes by area, **smallest first**.
- **Q1:** Your boss wants the **largest** shape first. What do you change?
- **Q2:** Now she also wants them grouped **by class name** (all Circles, then Rectangles...). Can you add a second `compareTo`?
- **Q3:** What if `Shape` came from a library and you can't edit it at all?
- Today: how to sort and filter the same objects in **many** ways without touching the class. This is the last section of Chapter 10.

Not on the Oct 8 midterm. It will be on the final, and Java code uses it everywhere.



TODAY

One Class, Many Orders

Comparable gives a class ONE natural order, built into the class. A Comparator is a separate object that holds ANOTHER order. We'll write the same Comparator three ways, each shorter than the last: a named class, an anonymous class, and a lambda.

Step 1: a named Comparator class

```
import java.util.Comparator;

public class LargestFirst
    implements Comparator<Shape> {
    @Override
    public int compare(Shape a, Shape b) {
        return Double.compare(b.area(), a.area());
    }
}

// Main.java
Arrays.sort(shapes); // compareTo
Arrays.sort(shapes, new LargestFirst()); // compare

// Natural:      [Triangle 6, Rectangle 20, Circle 28.27]
// LargestFirst: [Circle 28.27, Rectangle 20, Triangle 6]
```

- `Comparator<T>` (java.util) is **just another interface**, like `Resizable` or `MegaEvolvable`.
- Its one method is `compare(a, b)`. Negative means a goes first, positive means b goes first, zero means tie.
- To get largest first, **swap** the arguments:
`compare(b.area(), a.area())`.
- `Arrays.sort(array, comparator)` and `list.sort(comparator)` take the order as a **parameter**.
- `Shape.java` was **not changed**.

Class 11 zip → 01_comparator_class/

Comparable vs. Comparator

Comparable<T> (java.lang)

- Implemented **by the class itself**:
`class Shape implements Comparable<Shape>`
- One method: `this.compareTo(other)`
- Exactly **one** order: the "natural" one
- Used automatically: `Arrays.sort(a)`, `Collections.sort(list)`
- You need to be able to edit the class

Comparator<T> (java.util)

- Implemented by **a separate object**
- One method: `compare(a, b)`
- **As many orders as you want**: largest first, by name, by perimeter...
- Passed in: `Arrays.sort(a, cmp)`, `list.sort(cmp)`
- Works even on classes you can't edit (like `String`)

Step 2: an anonymous inner class

```
Arrays.sort(shapes, new Comparator<Shape>() {
    @Override
    public int compare(Shape a, Shape b) {
        return Double.compare(b.area(), a.area());
    }
});

// a second order - still no new file
Comparator<Shape> byName = new Comparator<Shape>() {
    @Override
    public int compare(Shape a, Shape b) {
        return a.getClass().getSimpleName()
            .compareTo(b.getClass().getSimpleName());
    }
};
Arrays.sort(shapes, byName);
// [Circle 28.27, Circle 3.14, Rectangle 20, Triangle 6]
```

- `new Comparator<Shape>() { ... }` **declares a class with no name and creates one object of it**, right where it's used.
- `LargestFirst.java` is gone. The order is written right next to the sort that uses it.
- It still is a real class. After `javac`, look for `Main$1.class` and `Main$2.class`.
- This also works for any interface (or to extend a class): `new MegaEvolvable() { ... }` would need all 4 methods.

Class 11 zip → 02_anonymous_class/

Functional interface = exactly ONE abstract method

Functional ✓

- `Comparator<T>`: `compare(a, b)`
- `Comparable<T>`: `compareTo(o)`
- `Resizable` (Class 8): `resize(factor)`
- `Runnable`: `run()`
- `ActionListener` (Swing buttons): `actionPerformed(e)`
- Add `@FunctionalInterface` and the compiler checks the rule for you

Not functional ✗

- `MegaEvolvable`: **4** abstract methods (`megaEvolve`, `revertMega`, `isMegaEvolved`, `getMegaFormName`)
- `Payable` (HW2): **2** methods, `getPaymentAmount()` and `getPayee()`
- Why the rule matters: if there's only one method, Java knows **which** method your code is. So you can skip the name and write just the body...

Step 3: a lambda expression

anonymous class (Step 2)

```
Arrays.sort(shapes,  
    new Comparator<Shape>() {  
        @Override  
        public int compare(Shape a,  
                           Shape b) {  
            return Double.compare(  
                b.area(), a.area());  
        }  
    });
```

lambda (Step 3)

```
Arrays.sort(shapes,  
    (a, b) -> Double.compare(  
        b.area(), a.area()));  
  
// parameters -> body  
// a, b are Shapes (inferred)  
// one-expression body is  
// returned automatically
```

Same object, same behavior. The compiler writes the class for you. Class 11 zip → 03_lambda/

Lambda syntax, all the shapes it comes in

```
// two parameters, one expression
(a, b) -> Double.compare(a.area(), b.area())

// one parameter: parentheses optional
s -> s.area() > 10

// types written out (allowed, rarely needed)
(Shape a, Shape b) -> a.getClass().getSimpleName()
                    .compareTo(b.getClass().getSimpleName())

// block body: { } and an explicit return
(a, b) -> {
    double diff = b.area() - a.area();
    return (int) Math.signum(diff);
}

// no parameters
() -> System.out.println("hello")
```

- A lambda is an **object** of a functional interface. You can store it in a variable:
`Comparator<Shape> byName = (a, b) -> ...;`
- Expression body: no `return` and no `;` inside.
- Block body `{ ... }`: write statements; `return` a value if the method returns one.
- The parameter count must match the interface method.

ArrayList methods that take lambdas

```
ArrayList<Shape> list = new ArrayList<>(Arrays.asList(shapes));
list.add(new Rectangle(2, 2));

list.sort((a, b) -> Double.compare(a.area(), b.area()));
// [Circle 3.14, Rectangle 4, Triangle 6, Rectangle 20, Circle 28.27]

list.removeIf(s -> s.area() < 10);
// [Rectangle 20, Circle 28.27]

list.forEach(s -> {
    s.draw();
    System.out.printf("  area = %.1f%n", s.area());
});
// Drawing a rectangle
//   area = 20.0
// Drawing a circle
//   area = 28.3
```

- `sort(Comparator)`: puts the list in order
- `removeIf(test)`: removes every element where the lambda returns **true**
- `forEach(action)`: runs the lambda once per element
- Each one takes a lambda as a parameter. The list does the looping for you.

Class 11 zip → 03_lambda/

The one rule: "effectively final"

```
double limit = 25;
list.removeIf(s -> s.area() > limit);    // OK

limit = 30;    // now limit changes...
// error: local variables referenced from a
// lambda expression must be final or
// effectively final

int count = 0;
list.forEach(s -> count++);                // error, same rule

// fix: a plain loop when you need to update a local
for (Shape s : list) count++;
```

- A lambda may **read** a local variable only if that variable never changes after it gets its value ("effectively final").
- A lambda may **never** change a local variable.
- Fields and method calls are fine. The rule is only about **local** variables.
- To count or add things up, a regular **for** loop is the simple answer.

You've already been using lambdas

```
// PokedexGUI.java (Class 9 capstone)

searchBox.addActionListener(e -> rebuild());

megaBtn.addActionListener(e -> {
    ...
});

sendBtn.addActionListener(
    e -> PokemonWorldClient.send(p, frame));
```

- `ActionListener` is a functional interface. Its one method is `actionPerformed(ActionEvent e)`.
- `e -> rebuild()` means: "when this button is clicked, run `rebuild()`."
- Before Java 8, every button needed an anonymous class, 5 lines of boilerplate each.
- Now you can read every line of the Pokédex GUI you've been running.

Class 9 zip → 05_mega_world_starter/PokedexGUI.java

In-class: your own functional interface

```
@FunctionalInterface
public interface ShapeFilter {
    boolean keep(Shape s);
}

static ArrayList<Shape> select(
    ArrayList<Shape> shapes, ShapeFilter f) {
    ArrayList<Shape> result = new ArrayList<>();
    for (Shape s : shapes)
        if (f.keep(s)) result.add(s);
    return result;
}

// worked example
select(shapes, s -> s.area() > 10)
// [Rectangle 20, Circle 28.27]
```

- **TODO 1:** only the Circles (`instanceof`)
- **TODO 2:** area between 4 and 20, inclusive
- **TODO 3:** area $\geq$ a local variable `min = 5`
- **TODO 4:** TODO 1 again as an **anonymous class**, then compare the length
- **TODO 5:** sort largest first with `shapes.sort(...)`
- `select` is written once. Each question is just a different lambda.

Class 11 zip → 04_shape_filter_starter/

Practice: sort the Pokédex with one-liners

```
// worked example: strongest attackers first
dex.sort((a, b) -> b.getAttack() - a.getAttack());
top("Top 3 by ATK:", dex, 3);
// Dragonite   ATK=134
// Pinsir      ATK=125
// Gyarados    ATK=125

// TODO 1: 3 fastest           (Mewtwo is #1)
// TODO 2: A-Z by name         (Abra is #1)
// TODO 3: best DEF, not legendary,
//           on a COPY + removeIf (Onix is #1)
// TODO 4: forEach: print Fire types
// TODO 5: why won't count++ compile?
```

- Same `pokemon_data.csv` and `Pokemon` class as the capstone.
- Each TODO is **one line** with a lambda.
- Getters return `int`, so `b.getAttack() - a.getAttack()` works as a compare result.
- Strings already have `compareTo`: `a.getName().compareTo(b.getName())`.
- Finish at home if needed. The self-check answers are in the comments.

Class 11 zip → `05_pokedex_lambdas_starter/` (run from inside the folder)

Check yourself

- 1. Which of these are functional interfaces: `Resizable` (Class 8), `Payable` (HW2: `getPaymentAmount()` and `getPayee()`), `MegaEvolvable`?
- 2. Write a lambda `Comparator<String>` that orders strings **shortest first**.
- 3. After this runs, what's in the list? `list = [3, 8, 1, 9]` (`ArrayList<Integer>`),
`list.removeIf(n -> n % 2 == 1);`
- 4. Why doesn't this compile? `int total = 0; dex.forEach(p -> total += p.getHp());`
- 5. An anonymous class `new Comparator<Shape>() {...}` written in Main. What file does `javac` create for it?

Check yourself: answers

- 1. Only `Resizable` (one abstract method). `Payable` has two and `MegaEvolvable` has four, so you can't write a lambda for either.
- 2. `(a, b) -> a.length() - b.length()` (or `Integer.compare(a.length(), b.length())`)
- 3. `[8].removeIf` deletes every element where the test is true, which here means the odd numbers.
- 4. `total` is a local variable that changes, so it isn't effectively final. Use a `for` loop instead.
- 5. `Main$1.class`. The class has no name in your code, but the compiler still makes a real class.

Wrap-up & next week

- **Comparator** = an order kept outside the class. **Anonymous class** = declare and create a class in one expression. **Lambda** = a short way to write a functional interface with exactly one method.
- **HW2 due Tue Oct 6, 11:59 PM** on D2L. Lambdas are not needed (or expected) in HW2.
- **Midterm Thu Oct 8**, Chapters 9–10 (Classes 3–9), **online at wcu.ninja/csc240/midterm, mostly programming**. Today's material is **not** on it. Try the practice exam first: wcu.ninja/csc240/midterm/practice.
- Review: Tuesday's Class 10 slides, the practice exam (timed), and HW1/HW2.

Sources

- Gaddis, **Starting Out with Java**, 4th ed., Ch. 10: anonymous inner classes, functional interfaces, and lambda expressions.
- Java SE 17 API: `java.util.Comparator`, `java.lang.Comparable`, `java.util.ArrayList` (`sort`, `removeIf`, `forEach`), `java.lang.FunctionalInterface`.
- Shape / Circle / Rectangle / Triangle: same classes as Class 9. Pokédex data and PokedexGUI from the Class 9 capstone.
- All outputs shown were verified with OpenJDK 17.