**Comparable** does not directly support lambdas because **Comparable** is designed to be implemented **within a class** to define the "natural ordering" of that class's objects. Lambdas are generally used for **functional interfaces**, which have a **single abstract method** (SAM), such as Comparator.

**Why Comparable Doesn't Use Lambdas:**

* **Comparable<T>** is an interface that classes implement to define how they should be **compared** to other objects of the same type. The class that implements Comparable must provide the comparison logic in the compareTo(T o) method.
* The purpose of **Comparable** is to define the natural ordering *once*, and it is tied to the object itself. Therefore, it requires the class to override the compareTo() method to define that order.

Lambdas, on the other hand, are typically used to provide **flexible, external logic** at runtime—this is more in line with what the Comparator interface does.

**Example of Comparable:**

When you implement Comparable, you write the compareTo() method inside the class to provide natural ordering:

java

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

public class Book implements Comparable<Book> {

private String title;

private String isbn;

// Constructor

public Book(String title, String isbn) {

this.title = title;

this.isbn = isbn;

}

// Implement compareTo to define natural order based on ISBN

@Override

public int compareTo(Book other) {

return this.isbn.compareTo(other.isbn);

}

@Override

public String toString() {

return "Book{" + "title='" + title + '\'' + ", isbn='" + isbn + '\'' + '}';

}

}

Here, compareTo() defines the natural ordering based on isbn.

**Why Comparator Supports Lambdas:**

* **Comparator** allows us to provide custom comparison logic externally, which is where lambdas become useful.
* You can pass lambdas to the Comparator because it is a functional interface (SAM interface), meaning it has only one abstract method: compare(T o1, T o2).

Here’s how you use lambdas with Comparator:

**Example with Comparator (Using Lambda):**

java

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

List<Book> books = new ArrayList<>();

books.add(new Book("Effective Java", "978-0134685991"));

books.add(new Book("Head First Java", "978-0596009205"));

books.add(new Book("Clean Code", "978-0132350884"));

// Sort using a lambda with Comparator based on title

Collections.sort(books, (book1, book2) -> book1.title.compareTo(book2.title));

books.forEach(System.out::println);

**Key Points:**

1. **Comparable** does **not** use lambdas because it’s meant to be implemented directly inside the class to provide a natural order, not a flexible order. The compareTo() method must be overridden in the class.
2. **Comparator** is designed for external and flexible ordering, and **supports lambdas** because it is a functional interface.

So, while Comparable doesn't use lambdas, you can achieve flexible sorting with Comparator using lambdas.