Module 9 of 20

Interfaces

Contracts without state, default/static/private methods and the diamond rule, functional interfaces, Comparable and Iterable, and sealed interfaces.

An interface is a type made only of promises. It is how Java gets multiple inheritance of type, how unrelated classes share a capability, how lambdas get a type, and — since Java 8 — how published APIs evolve without breaking their implementors.

This module covers interfaces from the contract up: declaring and implementing them and programming to the interface type; default, static and private methods and the exact rules for resolving conflicting defaults; functional interfaces as the target type of every lambda, with the java.util.function toolkit; Comparable and Iterable as the two interfaces every value type should consider, plus the guidelines for designing your own; and sealed interfaces, which close a hierarchy so the compiler can check it.

By the end you implement interfaces idiomatically, resolve a default-method diamond, choose the right functional interface, and model a closed set of alternatives with sealed types and records.

CONTENTS

1	Interfaces — contracts without implementation	13 min
2	Default, static and private interface methods	13 min
3	Functional interfaces and the shape of a lambda	13 min
4	Comparable, Iterable and designing with interfaces	14 min
5	Sealed interfaces — closed sets of types	12 min
✓	Checkpoint — Interfaces	22 min

Exercises in this module compile on OpenJDK 18. Reference solutions and the answer key follow each lesson.

LESSON 1 · 13 MIN

Interfaces — contracts without implementation

An interface is a **type** defined purely by what it can do: a set of method signatures with no state and (originally) no bodies. A class that `implements` an interface promises to provide those methods, and any code written against the interface type works with every implementation, present and future. Interfaces are how Java achieves multiple inheritance of *type*, and "program to an interface" is the most repeated design advice in the language for good reason.

Declaring and implementing

```
public interface Shape {
    double area();                          // implicitly public abstract
    double perimeter();
    int SIDES_UNKNOWN = -1;                 // implicitly public static final —
}

public class Circle implements Shape {
    private final double r;
    public Circle(double r) { this.r = r; }
    @Override public double area()      { return Math.PI * r * r; }      // Ml
    @Override public double perimeter() { return 2 * Math.PI * r; }
}
```

Rules:

- Interface methods without a body are `public abstract`, whether you write the words or not. An implementing class must declare them `public` — the most common compile error when implementing an interface ("attempting to assign weaker access privileges").
- Fields in an interface are `public static final` constants. Interfaces cannot have instance fields; there is no state to inherit.
- Interfaces have no constructors and cannot be instantiated: `new Shape()` is an error. `Shape s = new Circle(1)` is the whole point.
- A class implements **any number** of interfaces: `class Duck implements Swimmer, Flyer, Comparable<Duck>`.

An interface can `extends` other interfaces (one or several): `interface SortedSet<E> extends Set<E>`.

A class that does not implement every abstract method must be declared `abstract`.

The type, not the class

```
List<String> names = new ArrayList<>();      // declare with the interface; c
Set<Integer> ids = new HashSet<>();
Map<String, Integer> counts = new HashMap<>();
Comparable<Version> c = version;
```

Declaring variables, fields, parameters and return types as interfaces keeps the choice of implementation local: swapping `ArrayList` for `LinkedList`, or `HashMap` for `TreeMap`, changes one line. Methods that take `List<String>` accept every list; methods that take `ArrayList<String>` accept one class. The JDK's collections are the model: `List`, `Set`, `Map`, `Queue` are interfaces; the classes are details.

Polymorphism through interfaces

```
double total(List<Shape> shapes) {
    double sum = 0;
    for (Shape s : shapes) sum += s.area();     // dispatches to Circle.area,
    return sum;
}
```

Exactly the dynamic dispatch of Module 8, without a class hierarchy: `Circle` and `Square` need share no parent, no fields, nothing but the promise `Shape`. Unrelated classes can share a capability — a `String` and a `LocalDate` are both `Comparable`; a `Thread` and a lambda are both `Runnable`.

Interfaces the JDK expects you to implement

Interface	Method(s)	Enables
`Comparable<T>`	`int compareTo(T o)`	`Collections.sort`, `TreeMap`, `Arrays.sort` on your type
`Comparator<T>`	`int compare(T a, T b)`	Alternative orderings, passed to `sort`
`Runnable`	`void run()`	Threads, executors
`Callable<V>`	`V call()`	Tasks that return a value
`Iterable<T>`	`Iterator<T> iterator()`	For-each over your type
`AutoCloseable`	`void close()`	Try-with-resources
`Supplier<T>`, `Function<T,R>`, `Predicate<T>`, `Consumer<T>`	one method each	Lambdas (lesson 3, Module 11)

`Comparable` is the one to know cold: implement it on any value type with a natural order, using `Integer.compare` / `Double.compare` per field, most significant first, consistent with `equals` (if `compareTo` returns 0, `equals` should be true — `BigDecimal` famously breaks this).

Interfaces as roles

An interface names a **role** a class can play: `Closeable`, `Serializable`, `Comparable`, `Iterable`. Naming follows suit — adjectives (`-able`) or nouns describing a capability (`Shape`, `Repository`, `EventListener`). A class can play many roles; a class hierarchy could not express that.

Marker interfaces

`Serializable`, `Cloneable`, `RandomAccess` have **no methods**. They tag a class so that library code can test `instanceof` and behave accordingly. Annotations (`@FunctionalInterface`, `@Deprecated`) are the modern mechanism for most tagging; marker interfaces survive where the marker must be a *type* (a method parameter can require `Serializable`).

Constants in interfaces — a smell

Putting constants in an interface so classes can `implements` it to "inherit" them (`class Foo implements Constants`) pollutes the class's public API with unrelated constants. Use a `final` class with `private` constructor, or `static import`. The constant-interface pattern is *Effective Java* item 22.

Access and nesting

Top-level interfaces are `public` or package-private, like classes. Interfaces may be nested in classes and other interfaces (implicitly `static`). Members of an interface are always `public` — except the `private` helper methods added in Java 9 (next lesson).

Evolving an interface

Adding an abstract method to a published interface breaks every implementation. That is why Java 8 added **default methods** (next lesson): `List.sort`, `Collection.stream` and `Iterable.forEach` were added to interfaces implemented by thousands of classes without breaking one. Design interfaces small and stable; add capabilities as new interfaces or defaults.

Interview angle

"Can an interface have fields?" Only `public static final` constants.

"Why must implementing methods be public?" Interface methods are implicitly public; an override cannot narrow access.

"Can a class implement two interfaces with the same method signature?" Yes — one implementation satisfies both (if return types agree).

"Why declare `List<String> x = new ArrayList<>()`?" To depend on the contract, not the class.

"What is a marker interface?" An empty interface used as a type tag: `Serializable`.

Key takeaways

An interface is a contract of public abstract methods (plus constants); a class implements any number of them.

Program to the interface: declare variables and parameters as `List`, `Map`, `Shape`; construct concrete classes.

Unrelated classes share behaviour through interfaces — polymorphism without a hierarchy.

Know `Comparable`, `Comparator`, `Runnable`, `Iterable`, `AutoCloseable` and the functional interfaces.

Keep interfaces small; evolve them with default methods, not new abstract ones.

EXERCISE 1

Shapes through an interface

Declare `interface Shape` with `double area()` and `String name()`. Implement it in three **unrelated** classes `Circle`, `Rect` and `Triangle` (no shared parent). Read an integer `n` and `n` lines `circle <r>`, `rect <w> <h>` or `tri <b> <h>`; put them in a `List<Shape>`, print each as `<name>=<area with 2 decimals>`, and finish with `total=<sum with 2 decimals>`.

Example: `2` then `circle 1`, `tri 4 3` →

```
circle=3.14
triangle=6.00
total=9.14
```

INPUT 1

```
2
circle 1
tri 4 3
```

OUTPUT 1

```
circle=3.14
triangle=6.00
total=9.14
```

INPUT 2

```
3
rect 2 2
rect 0.5 0.5
circle 0
```

OUTPUT 2

```
rect=4.00
rect=0.25
circle=0.00
total=4.25
```

REFERENCE SOLUTION

```java
import java.util.*;

interface Shape {
    double area();
    String name();
}

class Circle implements Shape {
    private final double r;
    Circle(double r) { this.r = r; }
    public double area() { return Math.PI * r * r; }
    public String name() { return "circle"; }
}

class Rect implements Shape {
    private final double w, h;
    Rect(double w, double h) { this.w = w; this.h = h; }
    public double area() { return w * h; }
    public String name() { return "rect"; }
}

class Triangle implements Shape {
    private final double b, h;
    Triangle(double b, double h) { this.b = b; this.h = h; }
    public double area() { return 0.5 * b * h; }
    public String name() { return "triangle"; }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<Shape> shapes = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            String kind = in.next();
            switch (kind) {
                case "circle" -> shapes.add(new Circle(in.nextDouble()));
                case "rect" -> shapes.add(new Rect(in.nextDouble(), in.nextDouble()));
                default -> shapes.add(new Triangle(in.nextDouble(), in.nextDouble()));
            }
        }
        double total = 0;
        for (Shape s : shapes) {
            System.out.printf("%s=%.2f%n", s.name(), s.area());
            total += s.area();
        }
        System.out.printf("total=%.2f%n", total);
```

```
    }
}
```

EXERCISE 2

Greeters behind one contract

Declare `interface Greeter { String greet(String name); }` and three implementations: `Formal` (`Good day, <name>.`), `Casual` (`Hey <name>!`) and `Shouty` (`<NAME>!!!` in upper case). Write `static Greeter pick(String style)` returning the right one (`casual` by default). Read `n` lines `<style> <name>` and print `pick(style).greet(name)`. The caller only ever sees the interface.

Example: `3` then `formal Ada`, `shouty Linus`, `whatever Grace` →

```
Good day, Ada.
LINUS!!!
Hey Grace!
```

INPUT 1

```
3
formal Ada
shouty Linus
whatever Grace
```

OUTPUT 1

```
Good day, Ada.
LINUS!!!
Hey Grace!
```

INPUT 2

```
1
casual bob
```

OUTPUT 2

```
Hey bob!
```

REFERENCE SOLUTION

```java
import java.util.*;

interface Greeter {
    String greet(String name);
}
class Formal implements Greeter {
    public String greet(String name) { return "Good day, " + name + "."; }
}
class Casual implements Greeter {
    public String greet(String name) { return "Hey " + name + "!"; }
}
class Shouty implements Greeter {
    public String greet(String name) { return name.toUpperCase() + "!!!"; }
}

public class Main {
    static Greeter pick(String style) {
        return switch (style) {
            case "formal" -> new Formal();
            case "shouty" -> new Shouty();
            default -> new Casual();
        };
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            String style = in.next(), name = in.next();
            System.out.println(pick(style).greet(name));
        }
    }
}
```

QUIZ

1. A method declared in an interface without a body is implicitly...

A. `private`

B. `public abstract`

C. `protected`

D. `static`

2. Fields declared in an interface are…

A. Instance fields shared by implementors

B. `public static final` constants

C. Private

D. Not allowed at all

3. `List<String> xs = new ArrayList<>();` is preferred over `ArrayList<String> xs = …` because…

A. It is faster

B. Code depends on the `List` contract, so the implementation can change in one place

C. `ArrayList` is deprecated

D. It uses less memory

4. How many interfaces can a class implement?

A. One

B. Two

C. Any number

D. One per package

5. A marker interface such as `Serializable`…

A. Has one abstract method

B. Has no methods and tags a class as a type

C. Cannot be implemented

D. Is deprecated

ANSWER KEY · INTERFACES — CONTRACTS WITHOUT IMPLEMENTATION

1. B That is why the implementing class must declare it `public` — an override cannot reduce visibility.

2. B Interfaces carry no state. A field is a constant; abusing this as a way to "inherit" constants is an anti-pattern.

3. B Programming to the interface keeps the choice of class local to construction.

4. C Multiple inheritance of *type* is allowed; only class inheritance is single.

5. B Library code tests `instanceof Serializable`; the interface exists so the tag is a type a parameter can require.

LESSON 2 · 13 MIN

Default, static and private interface methods

Until Java 8 an interface was pure signatures. Then `default` methods gave interfaces bodies, static methods gave them utilities, and Java 9 added private helpers. The change was made so the collections could gain `stream()`, `forEach` and `sort` without breaking existing implementations — and it changed how interfaces are designed. This lesson covers the three kinds of concrete interface method and the one genuinely tricky rule: what happens when two defaults collide.

Default methods

```
public interface Greeter {
    String name();                                  // abstract

    default String greet() {                        // concrete, inherited
        return "Hello, " + name() + "!";
    }
}

public class Bot implements Greeter {
    public String name() { return "bot"; }          // greet() comes for fr
}

public class LoudBot implements Greeter {
    public String name() { return "LOUDBOT"; }
    @Override public String greet() { return Greeter.super.greet().toUpperCase
}
```

A `default` method has a body and is inherited by every implementing class that does not override it. It can call the interface's abstract methods — the template-method idea, but without state. `Greeter.super.greet()` is the syntax for reaching the interface's default from an override.

What defaults are for:

- **Evolving published interfaces**: `Collection.stream()`, `Iterable.forEach(Consumer)`, `List.sort(Comparator)`, `Map.getOrDefault`, `Map.computeIfAbsent`,

`Comparator.reversed()` — all added in Java 8 as defaults so existing classes kept compiling.

Convenience methods derived from the abstract ones: `Comparator.thenComparing`, `Predicate.and/or/negate`, `Function.andThen`.

Optional operations with a sensible default: `Iterator.remove()` defaults to throwing `UnsupportedOperationException`.

What they are not for: state. An interface still has no instance fields, so a default method can only compute from the abstract methods it calls. If shared code needs fields, that is an abstract class.

The diamond: resolving conflicts

Because a class can implement several interfaces, two can supply defaults with the same signature. Three rules resolve every case:

A class wins over an interface. If the class (or a superclass) defines the method, that is used; defaults are ignored. `Object`'s methods therefore can never be defaulted (`default String toString()` in an interface is a compile error).

A more specific interface wins. If `B extends A` and both define `default m()`, a class implementing both gets `B.m()`.

Otherwise the class must override and choose explicitly:

```
interface Swimmer { default String move() { return "swim"; } }
interface Flyer   { default String move() { return "fly"; } }

class Duck implements Swimmer, Flyer {
    @Override public String move() {
        return Swimmer.super.move() + " and " + Flyer.super.move();   // pick
    }
}
```

Without the override, `class Duck implements Swimmer, Flyer` fails: "types Swimmer and Flyer are incompatible; class Duck inherits unrelated defaults for move()". The compiler never guesses.

Static methods in interfaces

```
public interface Validator<T> {
    boolean isValid(T value);

    static <T> Validator<T> notNull() { return v → v ≠ null; }          // fa
    static <T> Validator<T> all(List<Validator<T>> vs) { … }
}
Validator<String> v = Validator.notNull();
```

Static interface methods are called on the interface name (`Validator.notNull()`), never on an instance and — unlike class statics — **not inherited** by implementing classes (`Bot.notNull()` does not compile). They exist for factories and utilities that belong with the type: `Comparator.comparing`, `List.of`, `Map.entry`, `Predicate.not`, `Function.identity`.

Private interface methods (Java 9)

```
public interface Logger {
    void write(String line);
    default void info(String msg)  { write(format("INFO", msg)); }
    default void error(String msg) { write(format("ERROR", msg)); }
    private String format(String level, String msg) {                    // s
        return "[" + level + "] " + msg;
    }
}
```

`private` (and `private static`) interface methods let default methods share code without exposing helper methods as part of the interface's API. They cannot be abstract and cannot be called from outside the interface.

Interfaces versus abstract classes, revisited

With defaults, interfaces can do most of what abstract classes do — except hold state, declare constructors, and have non-public members. The guidance:

> Interface with defaults when the shared code is a *derivation* of the abstract methods (`Comparator.reversed`).
>
> Abstract class when the shared code needs *fields* (a counter, a cache, a buffer) or a controlled constructor.

Both together (an interface plus an `AbstractX` skeleton) for library types others implement.

Defaults and `equals` / `hashCode`

An interface can *declare* `boolean equals(Object o)` abstractly to document a contract (`List` and `Set` do), but cannot provide a default for it — rule 1 above, since every class inherits `Object.equals`. Equality stays the class's responsibility.

Design cautions

A default method is part of the API forever; changing its behaviour changes every implementor that did not override it.

Defaults that call other defaults form dependency chains implementors cannot see; document which defaults call which (the fragile base class problem, interface edition).

Do not add defaults merely to save typing in one implementation — that is what an abstract class or a helper is for.

Interview angle

"Why were default methods added?" To evolve interfaces (collections gained `stream`) without breaking implementations.

"Two interfaces, same default — what happens?" Compile error unless the class overrides and picks with `X.super.m()`.

"Can an interface define `toString` as a default?" No — class methods (`Object`'s) always win.

"Are static interface methods inherited?" No; call them on the interface.

"What are private interface methods for?" Sharing code among defaults without exposing it.

Key takeaways

`default` methods give interfaces inherited bodies computed from their abstract methods; no state.

Conflict resolution: class beats interface; more specific interface beats less; otherwise the class must override (`Iface.super.m()`).

`static` interface methods are factories/utilities on the interface name, not inherited.

`private` interface methods share code between defaults.

Interfaces still cannot hold fields or constructors — that remains the abstract class's job.

EXERCISE 1

Resolve the diamond

Two interfaces, `Swimmer` and `Flyer` , each declare `default String move()` returning `"swim"` and `"fly"` , plus `default int speed()` returning 2 and 10. `Duck` implements both and must resolve the conflicts: `move()` returns `Swimmer.super.move() + "+" + Flyer.super.move()` and `speed()` returns the larger of the two. Also add a static method `Flyer.maxAltitude()` returning 1000 and print it via the interface name.

Read an integer `n` and print `n` lines: for each, the duck's `move()` , its `speed()` and `Flyer.maxAltitude()` joined by spaces.

Example: `2` →

```
swim+fly 10 1000
swim+fly 10 1000
```

INPUT 1

```
2
```

OUTPUT 1

```
swim+fly 10 1000
swim+fly 10 1000
```

REFERENCE SOLUTION

```java
import java.util.*;

interface Swimmer {
    default String move() { return "swim"; }
    default int speed() { return 2; }
}

interface Flyer {
    default String move() { return "fly"; }
    default int speed() { return 10; }
    static int maxAltitude() { return 1000; }
}

class Duck implements Swimmer, Flyer {
    @Override
    public String move() { return Swimmer.super.move() + "+" + Flyer.super.move(); }

    @Override
    public int speed() { return Math.max(Swimmer.super.speed(), Flyer.super.speed())
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Duck d = new Duck();
        for (int i = 0; i < n; i++) {
            System.out.println(d.move() + " " + d.speed() + " " + Flyer.maxAltitude(
        }
    }
}
```

EXERCISE 2

Default methods do the work

Declare `interface Shape` with abstract `double area()` and `String name()`, a **default** `String describe()` returning `<name()> with area <area() to two decimals>`, a default `boolean isLargerThan(Shape other)`, and a **static** `Shape largest(List<Shape> shapes)`. Implement `Circle` and `Square` as records. Read `n` shapes (`circle r` / `square s`), print each `describe()`, then `largest=<largest(shapes).name()>`. No implementation repeats the describe logic.

Example: `2` then `circle 1`, `square 2` →

```
circle with area 3.14
square with area 4.00
largest=square
```

INPUT 1

```
2
circle 1
square 2
```

OUTPUT 1

```
circle with area 3.14
square with area 4.00
largest=square
```

INPUT 2

```
1
circle 2
```

OUTPUT 2

```
circle with area 12.57
largest=circle
```

REFERENCE SOLUTION

```
import java.util.*;

interface Shape {
    double area();
    String name();
    default String describe() { return String.format(Locale.ROOT, "%s with area %.2f
    default boolean isLargerThan(Shape other) { return area() > other.area(); }
    static Shape largest(List<Shape> shapes) {
        Shape best = shapes.get(0);
        for (Shape s : shapes) if (s.isLargerThan(best)) best = s;
        return best;
    }
}
record Circle(double r) implements Shape {
    public double area() { return Math.PI * r * r; }
    public String name() { return "circle"; }
}
record Square(double s) implements Shape {
    public double area() { return s * s; }
    public String name() { return "square"; }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<Shape> shapes = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            String kind = in.next();
            shapes.add(kind.equals("circle") ? new Circle(in.nextDouble()) : new Squ
        }
        for (Shape s : shapes) System.out.println(s.describe());
        System.out.println("largest=" + Shape.largest(shapes).name());
    }
}
```

QUIZ

1. Why were default methods added to Java?

A. To replace abstract classes

B. To let interfaces such as `Collection` gain methods (`stream`) without breaking existing implementors

C. For performance

D. To allow interface fields

2. A class implements two interfaces that both provide `default void m()` . The class does not override `m` . Result?

A. The first interface wins

B. Compile error: the class must override `m` and may call `X.super.m()`

C. Runtime error

D. Both run

3. A superclass defines `m()` ; an interface the class implements has `default m()` . Which is used?

A. The interface's default

B. The superclass's method — classes win over interfaces

C. Compile error

D. Whichever is declared first

4. Static methods declared in an interface are…

A. Inherited by implementing classes

B. Called on the interface name and not inherited

C. Implicitly default

D. Not allowed

5. Private interface methods (Java 9) exist to...

A. Hide the abstract methods

B. Share code between default methods without exposing it as API

C. Store state

D. Replace static methods

ANSWER KEY · DEFAULT, STATIC AND PRIVATE INTERFACE METHODS

1. B Adding an abstract method breaks every implementation; a default arrives with a body they all inherit.

2. B The compiler refuses to guess between unrelated defaults. The class resolves it explicitly.

3. B Rule 1 of resolution: a method from a class always beats an interface default — which is also why `Object` methods cannot be defaulted.

4. B `Comparator.comparing(...)`, `List.of(...)` — factories that belong with the type but not with every implementor.

5. B A private helper can be called only from within the interface's own default and static methods.

LESSON 3 · 13 MIN

Functional interfaces and the shape of a lambda

A **functional interface** is an interface with exactly one abstract method. That single method is what a lambda expression or a method reference *implements* — every lambda in Java has a functional interface as its type. Understanding this link is what makes lambdas, streams and most modern Java APIs readable rather than magical. This lesson is the interface side; Module 11 covers the lambda syntax and captures in full.

One abstract method

```
@FunctionalInterface
public interface Validator<T> {
    boolean isValid(T value);                       // the single abstract meth

    default Validator<T> and(Validator<T> other) {  // defaults do not count
        return v -> isValid(v) && other.isValid(v);
    }
    static <T> Validator<T> notNull() { return v -> v != null; }   // statics
}
```

Default and static methods do not count toward the "one"; nor do abstract redeclarations of `Object`'s public methods (`Comparator` declares `boolean equals(Object)` and is still functional). `@FunctionalInterface` is optional but recommended: it makes adding a second abstract method a compile error, protecting every lambda that implements the interface.

Lambdas implement the SAM

```
Validator<String> nonEmpty = s -> !s.isEmpty();             // a lambda: an ins
Validator<String> shortish = s -> s.length() < 10;
Validator<String> both = nonEmpty.and(shortish);
both.isValid("hello");                                      // true
```

The compiler looks at the target type (`Validator<String>`), finds its single abstract method (`boolean isValid(String)`), checks that the lambda's parameter count and body fit, and

creates an object implementing the interface. The lambda *is* the implementation of `isValid`. Nothing else in the language is happening — no special "function type".

Before Java 8 the same thing was an **anonymous class**:

```
Validator<String> nonEmpty = new Validator<String>() {
    @Override public boolean isValid(String s) { return !s.isEmpty(); }
};
```

Lambdas are shorter, do not create a class file per use, and treat `this` differently (Module 11). Anonymous classes remain necessary for interfaces with more than one abstract method and for abstract classes.

The `java.util.function` toolkit

Java ships a family of general-purpose functional interfaces so you rarely write your own:

Interface	Method	Meaning
`Supplier<T>`	`T get()`	produce a value
`Consumer<T>`	`void accept(T t)`	use a value
`Function<T, R>`	`R apply(T t)`	transform
`Predicate<T>`	`boolean test(T t)`	test
`UnaryOperator<T>`	`T apply(T t)`	`Function<T, T>`
`BiFunction<T, U, R>`	`R apply(T t, U u)`	two inputs
`BinaryOperator<T>`	`T apply(T a, T b)`	`BiFunction<T, T, T>` — reduce steps
`BiConsumer<T, U>`, `BiPredicate<T, U>`		two-argument forms
`Runnable`	`void run()`	no input, no output
`Callable<V>`	`V call() throws Exception`	no input, a result, may throw
`Comparator<T>`	`int compare(T a, T b)`	ordering

Primitive specialisations avoid boxing: `IntPredicate`, `IntFunction<R>`, `ToIntFunction<T>`, `IntUnaryOperator`, `IntBinaryOperator`, `DoubleSupplier`, `LongConsumer`, and so on. Streams use them heavily (`mapToInt`, `IntStream.map`).

Choosing: count the inputs and whether there is an output. One in, boolean out → `Predicate`. One in, one out → `Function`. Nothing in, one out → `Supplier`. One in, nothing out → `Consumer`. Two in, one out → `BiFunction` (or `BinaryOperator` when all three types match).

Composition with default methods

The toolkit interfaces carry defaults that build new functions from old:

```
Predicate<String> isEmpty = String::isEmpty;
Predicate<String> notEmpty = isEmpty.negate();             // or Predicate.not
Predicate<String> shortNonEmpty = notEmpty.and(s → s.length() < 5);

Function<Integer, Integer> twice = x → x * 2;
Function<Integer, Integer> plusOne = x → x + 1;
twice.andThen(plusOne).apply(5);        // 11: (5*2)+1
twice.compose(plusOne).apply(5);        // 12: (5+1)*2

Comparator<String> byLength = Comparator.comparing(String::length);
Comparator<String> byLengthThenAlpha = byLength.thenComparing(Comparator.natur
```

`Comparator` is the richest: `comparing`, `comparingInt`, `thenComparing`, `reversed`, `nullsFirst`, `naturalOrder` — Module 14 uses all of them.

Writing your own

Prefer the standard interfaces; a custom one earns its place when the name carries meaning (`Validator`, `EventHandler`, `RetryPolicy`), when it needs a checked exception (`Callable`-like), or when it needs more than two parameters. Keep it to one abstract method, annotate it, and add composing defaults if they read naturally.

Method references

A method reference is a lambda that just calls one method: `String::isEmpty` for `s → s.isEmpty()`, `Integer::parseInt` for `s → Integer.parseInt(s)`, `System.out::println` for `x → System.out.println(x)`, `ArrayList::new` for `() → new ArrayList<>()`. Four kinds, all in Module 11; here the point is that they too need a functional-interface target type.

Functional interfaces and generics

`Function<T, R>` is generic in both directions; the compiler infers `T` and `R` from context — `map(String::length)` on a `Stream<String>` infers `Function<String, Integer>`. When inference fails ("cannot infer type arguments"), give the lambda's parameter a type: `(String s) → s.length()`, or assign to a typed variable first.

Interview angle

"What is a functional interface?" One abstract method; the target type of lambdas and method references.

"Does a default method break it?" No — only abstract methods count.

"What is `@FunctionalInterface` for?" A compile-time guarantee that the interface stays lambda-compatible.

"Name four from `java.util.function`." `Supplier`, `Consumer`, `Function`, `Predicate` (and `BiFunction`, `UnaryOperator`).

"`Runnable` vs `Callable`?" `Callable` returns a value and may throw checked exceptions.

Key takeaways

One abstract method = functional interface = a lambda's type; `@FunctionalInterface` enforces it.

`Supplier` / `Consumer` / `Function` / `Predicate` and their Bi-/primitive forms cover most needs; pick by arity and output.

Defaults compose: `and`, `or`, `negate`, `andThen`, `compose`, `thenComparing`, `reversed`.

Write a custom one for a meaningful name, a checked exception, or three-plus parameters.

EXERCISE 1

Validators from lambdas

Declare `@FunctionalInterface interface Validator<T>` with `boolean isValid(T value)` and a default `and(Validator<T> other)`. Build three validators for strings with lambdas: non-empty, length ≤ 8, and letters only (`s.chars().allMatch(Character::isLetter)`), combined with `and`. Read an integer `n` and `n` tokens; print `<token> ok` or `<token> rejected` for each, then `accepted=<count>`.

Example: `3` then `java Java17 verylongword` →

```
java ok
Java17 rejected
verylongword rejected
accepted=1
```

INPUT 1

```
3
java Java17 verylongword
```

OUTPUT 1

```
java ok
Java17 rejected
verylongword rejected
accepted=1
```

INPUT 2

```
2
abcdefgh abcdefghi
```

OUTPUT 2

```
abcdefgh ok
abcdefghi rejected
accepted=1
```

REFERENCE SOLUTION

```
import java.util.*;

@FunctionalInterface
interface Validator<T> {
    boolean isValid(T value);

    default Validator<T> and(Validator<T> other) {
        return v -> isValid(v) && other.isValid(v);
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Validator<String> nonEmpty = s -> !s.isEmpty();
        Validator<String> shortEnough = s -> s.length() <= 8;
        Validator<String> lettersOnly = s -> s.chars().allMatch(Character::isLetter)
        Validator<String> rule = nonEmpty.and(shortEnough).and(lettersOnly);
        int accepted = 0;
        for (int i = 0; i < n; i++) {
            String token = in.next();
            boolean ok = rule.isValid(token);
            if (ok) accepted++;
            System.out.println(token + (ok ? " ok" : " rejected"));
        }
        System.out.println("accepted=" + accepted);
    }
}
```

EXERCISE 2

Your own functional interface

Declare `@FunctionalInterface interface IntOp { int apply(int x); default IntOp then(IntOp next) { … } }` where `then` returns an op that applies this then `next`. Read an integer `k` and `k` step names — `double`, `inc`, `square`, `negate` — and build one `IntOp` by chaining lambdas with `then` in order. Then read `n` integers and print `<x> → <result>` for each.

Example: `2` then `inc square`, then `3` then `1 2 -3` →

```
1 → 4
2 → 9
-3 → 4
```

INPUT 1

```
2
inc square
3
1 2 -3
```

OUTPUT 1

```
1 → 4
2 → 9
-3 → 4
```

INPUT 2

```
0
2
5 -5
```

OUTPUT 2

```
5 → 5
-5 → -5
```

REFERENCE SOLUTION

```java
import java.util.*;

@FunctionalInterface
interface IntOp {
    int apply(int x);
    default IntOp then(IntOp next) {
        return x -> next.apply(apply(x));
    }
}

public class Main {
    static IntOp named(String step) {
        return switch (step) {
            case "double" -> x -> x * 2;
            case "inc" -> x -> x + 1;
            case "square" -> x -> x * x;
            default -> x -> -x;
        };
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int k = in.nextInt();
        IntOp op = x -> x;
        for (int i = 0; i < k; i++) op = op.then(named(in.next()));
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            int x = in.nextInt();
            System.out.println(x + " -> " + op.apply(x));
        }
    }
}
```

QUIZ

1. A functional interface has…

A. No methods

B. Exactly one abstract method (defaults and statics allowed)

C. Only default methods

D. One method of any kind

2. `@FunctionalInterface` causes…

A. Lambdas to run faster

B. A compile error if the interface gains a second abstract method

C. Automatic default methods

D. Nothing — it is documentation only

3. Which interface fits "take a `String`, return an `Integer`"?

A. `Predicate<String>`

B. `Function<String, Integer>`

C. `Supplier<Integer>`

D. `Consumer<String>`

4. `Runnable` vs `Callable<V>`?

A. Identical

B. `Callable` returns a value and may throw checked exceptions

C. `Runnable` returns a value

D. `Callable` is for threads only

5. `twice.andThen(plusOne).apply(5)` where `twice = x → x * 2` and `plusOne = x → x + 1` gives…

A. 12

B. 11

C. 10

D. 6

ANSWER KEY · FUNCTIONAL INTERFACES AND THE SHAPE OF A LAMBDA

1. B The single abstract method is what a lambda implements; extra defaults and statics do not count.

2. B It protects every lambda that targets the interface from a future incompatible change.

3. B One input, one output of a different type is `Function<T, R>`. `Predicate` returns boolean; `Supplier` takes nothing; `Consumer` returns nothing.

4. B `void run()` versus `V call() throws Exception` — the executor framework accepts both.

5. B `andThen` applies `twice` first, then `plusOne`: (5 × 2) + 1. `compose` would do the reverse.

LESSON 4 · 14 MIN

Comparable, Iterable and designing with interfaces

Two interfaces from the JDK teach more about interface design than any abstract discussion: `Comparable`, which gives a type a natural order, and `Iterable`, which lets it be looped over with for-each. Implementing both correctly is expected of any value type you write, and both illustrate how a small contract unlocks a large amount of library behaviour. The second half of the lesson turns those lessons into guidelines for interfaces of your own.

`Comparable<T>`: natural order

```
public final class Version implements Comparable<Version> {
    private final int major, minor, patch;

    @Override
    public int compareTo(Version o) {
        int c = Integer.compare(major, o.major);
        if (c ≠ 0) return c;
        c = Integer.compare(minor, o.minor);
        if (c ≠ 0) return c;
        return Integer.compare(patch, o.patch);
    }
}
```

`compareTo` returns negative, zero or positive for less-than, equal, greater-than. The contract:

Antisymmetric: `sgn(a.compareTo(b)) == -sgn(b.compareTo(a))`.

Transitive.

Consistent with equals (strongly recommended): `a.compareTo(b) == 0` iff `a.equals(b)`. `TreeSet` and `TreeMap` use `compareTo` *instead of* `equals`, so an inconsistency means a `TreeSet` and a `HashSet` disagree about which elements are duplicates. `BigDecimal` (`2.0` vs `2.00`) is the standard cautionary example.

Throws `NullPointerException` for a null argument (it is not meant to handle null).

Idioms: compare fields most-significant first; use `Integer.compare` / `Long.compare` / `Double.compare`, never subtraction (overflow) or `==` on

doubles; for strings `compareTo` or `compareToIgnoreCase`; for a chain of fields, `Comparator.comparing(...).thenComparing(...)` inside `compareTo` reads well:

```
private static final Comparator<Version> ORDER =
    Comparator.comparingInt((Version v) → v.major).thenComparingInt(v → v.mi
@Override public int compareTo(Version o) { return ORDER.compare(this, o); }
```

What implementing it buys: `Collections.sort(list)`, `Arrays.sort(array)`, `list.sort(null)`, `Collections.max/min`, `TreeSet` / `TreeMap` keys, `stream().sorted()`, `PriorityQueue` ordering — all without passing a comparator.

`Comparator<T>`: other orders

A type has one natural order but many useful ones. `Comparator` is an external ordering passed to the sort:

```
list.sort(Comparator.comparing(Person::lastName).thenComparing(Person::firstNa
list.sort(Comparator.comparingInt(Person::age).reversed());
list.sort(Comparator.comparing(Person::nickname, Comparator.nullsLast(Comparat
```

Implement `Comparable` for *the* order (if there is one obvious order); supply `Comparator`s for the rest. A class with no obvious order (`Employee`?) should implement neither and let callers pass comparators.

`Iterable<T>`: for-each over your type

```
public class Range implements Iterable<Integer> {
    private final int lo, hi;
    public Range(int lo, int hi) { this.lo = lo; this.hi = hi; }

    @Override
    public Iterator<Integer> iterator() {
        return new Iterator<>() {
            private int next = lo;
            @Override public boolean hasNext() { return next < hi; }
            @Override public Integer next() {
                if (!hasNext()) throw new NoSuchElementException();
                return next++;
            }
        };
    }
}

for (int i : new Range(0, 5)) System.out.print(i);     // 01234
```

The for-each loop desugars to `for (Iterator<T> it = x.iterator(); it.hasNext(); ) { T v = it.next(); … }`. Implementing `Iterable` means providing an `Iterator` — an object with `hasNext()` / `next()` (and optionally `remove()`). Rules: `next()` without a preceding successful `hasNext()` must throw `NoSuchElementException`, not return garbage; each call to `iterator()` returns a fresh, independent iterator; `Iterable` also gives you `forEach(Consumer)` and `spliterator()` as defaults, so `StreamSupport.stream(range.spliterator(), false)` opens streams.

Lazy iteration is the point: `Range` never materialises a list. Iterators over trees, files and generated sequences all follow this shape.

Guidelines for your own interfaces

Small. One to five methods. `Comparable` has one; `Iterable` has one. A twenty-method interface is implemented rarely and mocked painfully. Split by role (**interface segregation**): `Readable` and `Writable`, not `ReadWritable`.

Named for the capability. `Closeable`, `Runnable`, `Repository<T>`, `Shape`. Avoid `I` prefixes and `Impl` suffixes; `ArrayList implements List`, not `ListImpl implements IList`.

Stable. Adding abstract methods breaks implementors; add default methods or a new interface (`SortedSet extends Set`).

Documented contracts. What `compareTo(null)` does, whether `iterator()` is repeatable, whether `close()` is idempotent — the Javadoc is the real interface.

Depend on interfaces at boundaries. Parameters and return types of public APIs should be interfaces (`List`, not `ArrayList`; `Map`, not `HashMap`) so callers and implementations can change independently.

Provide a skeleton when the interface has many methods derivable from a few (`AbstractList`), or defaults when they derive from the abstract ones.

One method → consider functional. If it is a single operation, a functional interface lets callers use lambdas.

Interfaces and testing

An interface at a boundary — `PaymentGateway`, `Clock`, `Repository` — is what makes the code that uses it testable: the test supplies a fake implementation (`InMemoryRepository`, a fixed `Clock`) and exercises the logic without a network or a database. This is the practical reason for "program to an interface" in application code: not flexibility for its own sake, but substitutability where it matters. `java.time.Clock` is the JDK's own example.

Anti-patterns

Header interfaces: one interface per class, mirroring every method, implemented once. Noise without substitutability.

Constant interfaces (Module 9 lesson 1).

God interfaces implemented by throwing `UnsupportedOperationException` from half the methods — the Liskov violation in interface form. (`Collection` does exactly this for optional operations and is widely regarded as a compromise, not a model.)

Leaking implementation types through interface methods (`ArrayList<T> items()` on an interface).

Interview angle

"`Comparable` vs `Comparator`?" Natural order inside the class vs external orderings passed to sorts; one vs many.

"Why must `compareTo` be consistent with `equals`?" Sorted collections use `compareTo` to decide duplicates.

"How does for-each work on your class?" Implement `Iterable`; the loop calls `iterator()`, then `hasNext` / `next`.

"What is interface segregation?" Small, role-specific interfaces so implementors depend only on what they use.

Key takeaways

`Comparable.compareTo`: most significant field first, `Integer.compare`, consistent with `equals`; unlocks sorting and sorted collections.

`Comparator` for every other ordering; build with `comparing` / `thenComparing` / `reversed`.

`Iterable.iterator()` returning a fresh `Iterator` gives for-each and `forEach`; `next()` past the end throws.

Design interfaces small, capability-named, stable, documented, and use them at boundaries — that is also what makes code testable.

EXERCISE 1

A countdown you can loop over

Write `class Countdown implements Iterable<Integer>` that iterates from a start value down to 1 (an empty sequence if the start is below 1). Its `iterator()` must return a fresh iterator each call, and `next()` past the end must throw `NoSuchElementException`.

Read an integer `n` and `n` start values. For each, iterate with a for-each loop and print the values space-separated (an empty line for an empty sequence), then a second line `sum=<sum>` computed by a **second** for-each over the same object.

Example: `2` then `3 0` →

```
3 2 1
sum=6

sum=0
```

INPUT 1

```
2
3 0
```

OUTPUT 1

```
3 2 1
sum=6

sum=0
```

INPUT 2

```
1
5
```

OUTPUT 2

```
5 4 3 2 1
sum=15
```

REFERENCE SOLUTION

```java
import java.util.*;

class Countdown implements Iterable<Integer> {
    private final int start;
    Countdown(int start) { this.start = start; }

    @Override
    public Iterator<Integer> iterator() {
        return new Iterator<>() {
            private int current = start;
            @Override public boolean hasNext() { return current >= 1; }
            @Override public Integer next() {
                if (!hasNext()) throw new NoSuchElementException();
                return current--;
            }
        };
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            Countdown c = new Countdown(in.nextInt());
            StringBuilder sb = new StringBuilder();
            for (int v : c) {
                if (sb.length() > 0) sb.append(' ');
                sb.append(v);
            }
            System.out.println(sb);
            long sum = 0;
            for (int v : c) sum += v;
            System.out.println("sum=" + sum);
        }
    }
}
```

EXERCISE 2

Ranges you can loop and sort

Write `record Range(int from, int to) implements Iterable<Integer>, Comparable<Range>`: iterating yields `from` up to but excluding `to`; the natural order is by length ascending, then by `from`. Read `n` ranges, print each in sorted order as `[from,to) size=<len> sum=<sum of its elements via for-each>`.

Example: `3` then `5 8`, `0 2`, `10 12` →

```
[0,2) size=2 sum=1
[10,12) size=2 sum=21
[5,8) size=3 sum=18
```

INPUT 1

```
3
5 8
0 2
10 12
```

OUTPUT 1

```
[0,2) size=2 sum=1
[10,12) size=2 sum=21
[5,8) size=3 sum=18
```

INPUT 2

```
1
3 3
```

OUTPUT 2

```
[3,3) size=0 sum=0
```

REFERENCE SOLUTION

```java
import java.util.*;

record Range(int from, int to) implements Iterable<Integer>, Comparable<Range> {
    int size() { return to - from; }
    public Iterator<Integer> iterator() {
        return new Iterator<>() {
            int next = from;
            public boolean hasNext() { return next < to; }
            public Integer next() { return next++; }
        };
    }
    public int compareTo(Range o) {
        int c = Integer.compare(size(), o.size());
        return c != 0 ? c : Integer.compare(from, o.from);
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<Range> ranges = new ArrayList<>();
        for (int i = 0; i < n; i++) ranges.add(new Range(in.nextInt(), in.nextInt()))
        Collections.sort(ranges);
        for (Range r : ranges) {
            long sum = 0;
            for (int x : r) sum += x;
            System.out.println("[" + r.from() + "," + r.to() + ") size=" + r.size()
        }
    }
}
```

QUIZ

1. Why should `compareTo` be consistent with `equals`?

A. The compiler requires it

B. `TreeSet` / `TreeMap` use `compareTo` to decide duplicates and would disagree with `HashSet`

C. For speed

D. It is not recommended

2. `Comparable` versus `Comparator`?

A. Same thing

B. `Comparable` is the type's own natural order; `Comparator` is an external, additional ordering

C. `Comparator` is for primitives

D. `Comparable` is deprecated

3. For-each over an object requires it to implement…

A. `Collection`

B. `Iterable`

C. `List`

D. `Iterator`

4. `Iterator.next()` called when `hasNext()` is false must…

A. Return null

B. Throw `NoSuchElementException`

C. Return the last element

D. Restart

5. Interface segregation suggests…

A. One huge interface per module

B. Small, role-specific interfaces so implementors depend only on what they use

C. No interfaces

D. Interfaces with only default methods

ANSWER KEY · COMPARABLE, ITERABLE AND DESIGNING WITH INTERFACES

1. B Sorted collections never call `equals`; elements comparing as 0 are treated as the same.

2. B One natural order lives in the class; any number of comparators can be passed to sorts and sorted collections.

3. B The loop calls `iterator()` and then `hasNext` / `next`. Arrays are handled specially by the compiler.

4. B That is the `Iterator` contract; returning a value would silently corrupt loops that forgot to check.

5. B `Readable` and `Writable` rather than one `ReadWritable` that half the implementors must partially reject.

LESSON 5 · 12 MIN

Sealed interfaces — closed sets of types

An ordinary interface is open: anyone, anywhere, can implement it. That is right for `Comparable` and wrong for "the result of a parse is either a success with a value or a failure with a message" — a set of alternatives that is *complete* by design. Java 17's **sealed interfaces** express exactly that: the interface names its permitted implementations, the compiler knows the set is closed, and (with records and pattern matching) Java gains the algebraic data types that functional languages have had for decades.

Declaring a sealed interface

```
public sealed interface Shape permits Circle, Square, Triangle { }

public record Circle(double radius) implements Shape { }
public record Square(double side) implements Shape { }
public record Triangle(double base, double height) implements Shape { }
```

Rules:

- `sealed` requires `permits` listing every direct implementor — unless they are all in the same source file, in which case `permits` may be omitted and inferred.
- Each permitted type must directly implement the interface and be `final` (records and enums are implicitly final), `sealed` (continuing the closed hierarchy), or `non-sealed` (reopening that branch to anyone).
- Permitted types must be in the same module, or the same package if in the unnamed module.
- A sealed interface can have everything an interface has: abstract methods, defaults, statics.

Attempting `class Hexagon implements Shape` elsewhere fails: "class is not allowed to extend sealed class".

Why: exhaustiveness

The compiler now knows every `Shape` is a `Circle`, a `Square` or a `Triangle`. In Java 21 a `switch` with type patterns can prove it has covered them all:

```
static double area(Shape s) {
    return switch (s) {
        case Circle c   → Math.PI * c.radius() * c.radius();
        case Square q   → q.side() * q.side();
        case Triangle t → 0.5 * t.base() * t.height();
    };                                  // no default — exhaustive, checked l
}
```

Add a fourth shape and *every* such switch stops compiling until it handles the new case. That is the safety enums provide for constants, extended to types that carry data. On JDK 17–20 (and the JDK 18 runner behind these exercises), pattern switch is a preview feature, so the same logic is written with `instanceof` patterns — the sealing still documents and restricts the hierarchy, and the exhaustive switch arrives with Java 21.

Modelling alternatives: results, states, expressions

Sealed interfaces plus records model "one of these" data cleanly:

```
sealed interface ParseResult permits Ok, Err { }
record Ok(int value) implements ParseResult { }
record Err(String message) implements ParseResult { }

static ParseResult parse(String s) {
    try { return new Ok(Integer.parseInt(s)); }
    catch (NumberFormatException e) { return new Err("not a number: " + s); }
}
```

No exceptions for expected failures, no null, no `boolean success` field next to nullable ones — the type says a result is exactly one of two shapes, and the caller must handle both. The same technique models state machines (`Idle | Running | Done | Failed`), expression trees (`Num | Add | Mul`), and events. In interviews, "how would you model a result that can succeed or fail?" is answered with this.

Sealed classes versus sealed interfaces

`sealed` works on classes too (Module 8), but interfaces are the common choice: records cannot extend classes (they extend `Record`), and records are the natural leaves of a sealed hierarchy. Reach for a sealed *class* only when the alternatives share mutable state or a constructor.

Interaction with records

Records (Module 10) are final, immutable, with generated `equals` / `hashCode` / `toString` — everything a data-carrying alternative wants. A sealed interface of records is Java's tuple-and-variant type. Java 21 adds *record patterns* to destructure them in a switch: `case Circle(double r) → …`.

When *not* to seal

The set of implementations is genuinely open (plugins, user-defined strategies, `Comparable`).

Implementations live in other modules you do not control.

You want the classic OO extensibility: adding a type without touching existing code. Sealing trades that for the other axis — adding an *operation* (a new switch) without touching the types. Pick the axis that changes more often (the "expression problem").

Reflection and tooling

`Shape.class.isSealed()` and `getPermittedSubclasses()` expose the set at run time; frameworks use them for serialisation of variant types. `non-sealed` shows up in Javadoc as a deliberate reopening — document why.

Interview angle

"What does `sealed` give that `final` and `abstract` do not?" A *closed but non-empty* set of subtypes the compiler can enumerate — exhaustive switches.

"Which modifiers must a permitted subtype have?" `final`, `sealed` or `non-sealed`.

"How do you model success-or-failure without exceptions?" A sealed interface with `Ok` and `Err` records.

"Enums vs sealed types?" Enums are a fixed set of *instances*; sealed types are a fixed set of *types* whose instances carry data.

Key takeaways

`sealed interface X permits A, B, C` closes the set of implementors; each is `final`, `sealed` or `non-sealed`.

Closed sets enable compiler-checked exhaustive switches (Java 21 pattern switch; `instanceof` chains on JDK 17–20).

Sealed interface + records = algebraic data types: results, states, expression trees.

Seal when the alternatives are fixed by design; leave open when extension by others is the point.

EXERCISE 1

Parse results without exceptions

Declare `sealed interface ParseResult permits Ok, Err` with `record Ok(long value)` and `record Err(String message)` implementing it. Write `static ParseResult parse(String s)` returning `Ok` for a valid `long` and `Err("not a number: " + s)` otherwise (catch `NumberFormatException`).

Read an integer `n` and `n` tokens. For each, use `instanceof` patterns on the result: print `ok <value>` or `err <message>`. Finally print `sum=<sum of all Ok values>`.

Example: `3` then `12 x7 -5` →

```
ok 12
err not a number: x7
ok -5
sum=7
```

INPUT 1

```
3
12 x7 -5
```

OUTPUT 1

```
ok 12
err not a number: x7
ok -5
sum=7
```

INPUT 2

```
2
9223372036854775807 9223372036854775808
```

OUTPUT 2

```
ok 9223372036854775807
err not a number: 9223372036854775808
sum=9223372036854775807
```

REFERENCE SOLUTION

```java
import java.util.*;

sealed interface ParseResult permits Ok, Err { }
record Ok(long value) implements ParseResult { }
record Err(String message) implements ParseResult { }

public class Main {
    static ParseResult parse(String s) {
        try {
            return new Ok(Long.parseLong(s));
        } catch (NumberFormatException e) {
            return new Err("not a number: " + s);
        }
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        long sum = 0;
        for (int i = 0; i < n; i++) {
            ParseResult r = parse(in.next());
            if (r instanceof Ok ok) {
                sum += ok.value();
                System.out.println("ok " + ok.value());
            } else if (r instanceof Err err) {
                System.out.println("err " + err.message());
            }
        }
        System.out.println("sum=" + sum);
    }
}
```

EXERCISE 2

Ok or Err

Declare `sealed interface Result permits Ok, Err` with `record Ok(int value)` and `record Err(String message)`. Write `static Result parse(String token)` that returns `Ok` for an integer and `Err("not a number: <token>")` otherwise, and `Err("out of range: <token>")` for values

outside `[-1000, 1000]`. Read `n` tokens; for each print `ok <value>` or `err <message>` using pattern `instanceof`, then `sum=<sum of the Ok values>`.

Example: `4` then `5 x 2000 -7` →

```
ok 5
err not a number: x
err out of range: 2000
ok -7
sum=-2
```

INPUT 1

```
4
5 x 2000 -7
```

OUTPUT 1

```
ok 5
err not a number: x
err out of range: 2000
ok -7
sum=-2
```

INPUT 2

```
2
1000 -1001
```

OUTPUT 2

```
ok 1000
err out of range: -1001
sum=1000
```

REFERENCE SOLUTION

```java
import java.util.*;

sealed interface Result permits Ok, Err { }
record Ok(int value) implements Result { }
record Err(String message) implements Result { }

public class Main {
    static Result parse(String token) {
        try {
            int v = Integer.parseInt(token);
            if (v < -1000 || v > 1000) return new Err("out of range: " + token);
            return new Ok(v);
        } catch (NumberFormatException e) {
            return new Err("not a number: " + token);
        }
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        long sum = 0;
        for (int i = 0; i < n; i++) {
            Result r = parse(in.next());
            if (r instanceof Ok ok) {
                sum += ok.value();
                System.out.println("ok " + ok.value());
            } else if (r instanceof Err err) {
                System.out.println("err " + err.message());
            }
        }
        System.out.println("sum=" + sum);
    }
}
```

QUIZ

1. A permitted implementor of a sealed interface must be…

A. Abstract

B. `final`, `sealed` or `non-sealed`

C. In another package

D. A class, never a record

2. The main benefit of sealing a hierarchy is…

A. Faster dispatch

B. The compiler knows every subtype, enabling exhaustive pattern switches

C. Smaller class files

D. Automatic `equals`

3. Why pair sealed interfaces with records?

A. Records are required by the syntax

B. Records are final, immutable data carriers with generated equals/hashCode/toString — ideal variant leaves

C. Classes cannot implement interfaces

D. Records are faster

4. When should you **not** seal an interface?

A. When implementations are a closed set

B. When other modules or plugins should be able to implement it

C. When using records

D. Never

5. Enums versus sealed types?

A. Same feature

B. Enums are a fixed set of instances; sealed types are a fixed set of types whose instances carry data

C. Sealed types cannot have methods

D. Enums are newer

ANSWER KEY · SEALED INTERFACES — CLOSED SETS OF TYPES

1. B Each branch declares whether the hierarchy closes, continues closed, or reopens. Records and enums are implicitly final.

2. B Adding a variant then fails to compile at every switch that does not handle it — the safety enums give constants, for data-carrying types.

3. B A sealed interface of records is Java's algebraic data type: `Ok(value) | Err(message)`.

4. B Sealing trades open extension by type for safe extension by operation; choose it when the variants are fixed by design.

5. B `Color.RED` is one object; every `new Circle(r)` is a distinct instance of one of the permitted types.

CHECKPOINT · 22 MIN

Checkpoint — Interfaces

This checkpoint covers interface basics, default/static/private methods and conflict resolution, functional interfaces, `Comparable` and `Iterable`, and sealed interfaces.

How it works. Twelve questions and two programs; 70% on the questions and both programs accepted clears the module.

Before you start, make sure you can answer:

- What an interface may contain, and why implementing methods must be `public`.
- The three rules for resolving two defaults with the same signature.
- What makes an interface functional, and what `@FunctionalInterface` enforces.
- Why `compareTo` should agree with `equals`, and which collections rely on it.
- What `Iterable` requires, and what `next()` must do past the end.
- What `sealed … permits` buys, and the modifiers a permitted type needs.

The programs are an `Iterable` type with a natural order, and a strategy-style interface implemented by several small classes and a lambda.

EXERCISE 1

Ordered, iterable playlist

Write `final class Track implements Comparable<Track>` with a title and a duration in seconds; natural order is by duration ascending, then title. Write `class Playlist implements Iterable<Track>` that stores tracks in insertion order and offers `sorted()` returning a **new** `List<Track>` in natural order (`Collections.sort` on a copy).

Read an integer `n` and `n` lines `<title> <seconds>` (titles have no spaces). Print the playlist in insertion order via for-each, one per line as `<title> (<seconds>s)`, then a line `--`, then the sorted order in the same format.

Example: `3` then `intro 30`, `epic 300`, `bridge 30` →

```
intro (30s)
epic (300s)
bridge (30s)
--
bridge (30s)
intro (30s)
epic (300s)
```

INPUT 1

```
3
intro 30
epic 300
bridge 30
```

OUTPUT 1

```
intro (30s)
epic (300s)
bridge (30s)
--
bridge (30s)
intro (30s)
epic (300s)
```

INPUT 2

```
1
solo 1
```

OUTPUT 2

```
solo (1s)
--
solo (1s)
```

REFERENCE SOLUTION

```
import java.util.*;

final class Track implements Comparable<Track> {
    private final String title;
    private final int seconds;

    Track(String title, int seconds) {
        this.title = title;
        this.seconds = seconds;
    }

    @Override
    public int compareTo(Track o) {
        int c = Integer.compare(seconds, o.seconds);
        return c != 0 ? c : title.compareTo(o.title);
    }

    @Override
    public String toString() { return title + " (" + seconds + "s)"; }
}

class Playlist implements Iterable<Track> {
    private final List<Track> tracks = new ArrayList<>();

    void add(Track t) { tracks.add(t); }

    @Override
    public Iterator<Track> iterator() { return tracks.iterator(); }

    List<Track> sorted() {
        List<Track> copy = new ArrayList<>(tracks);
        Collections.sort(copy);
        return copy;
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Playlist p = new Playlist();
        for (int i = 0; i < n; i++) p.add(new Track(in.next(), in.nextInt()));
        for (Track t : p) System.out.println(t);
        System.out.println("--");
        for (Track t : p.sorted()) System.out.println(t);
    }
}
```

EXERCISE 2

Pricing strategies

Declare `@FunctionalInterface interface Pricing { long price(long baseCents, int qty); }`. Provide three implementations: a class `Flat` (base × qty), a class `Bulk` (10% off the flat price when qty ≥ 10, integer division), and a **lambda** for `Member` pricing (flat price minus 200 cents per item, never below 0). Read an integer `n` and `n` lines `<strategy> <baseCents> <qty>` where strategy is `flat`, `bulk` or `member`; look up the `Pricing` in a `Map<String, Pricing>` and print the price.

Example: `3` then `flat 1000 2`, `bulk 1000 10`, `member 100 3` →

```
2000
9000
0
```

INPUT 1

```
3
flat 1000 2
bulk 1000 10
member 100 3
```

OUTPUT 1

```
2000
9000
0
```

INPUT 2

```
2
bulk 1000 9
member 500 4
```

OUTPUT 2

```
9000
1200
```

REFERENCE SOLUTION

```java
import java.util.*;

@FunctionalInterface
interface Pricing {
    long price(long baseCents, int qty);
}

class Flat implements Pricing {
    public long price(long baseCents, int qty) { return baseCents * qty; }
}

class Bulk implements Pricing {
    public long price(long baseCents, int qty) {
        long flat = baseCents * qty;
        return qty ⩾ 10 ? flat * 90 / 100 : flat;
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Map<String, Pricing> strategies = new HashMap<>();
        strategies.put("flat", new Flat());
        strategies.put("bulk", new Bulk());
        strategies.put("member", (base, qty) -> Math.max(0, base * qty - 200L * qty)
        for (int i = 0; i < n; i++) {
            String name = in.next();
            long base = in.nextLong();
            int qty = in.nextInt();
            System.out.println(strategies.get(name).price(base, qty));
        }
    }
}
```

EXERCISE 3

Listeners you can unsubscribe

Declare `@FunctionalInterface interface Listener { void on(String event); }` and `class Bus` with `Runnable subscribe(Listener l)` — it registers the listener and returns a `Runnable` that removes it again — and `void publish(String event)` which calls every current listener in subscription order. Read `n` commands: `sub <name>` subscribes a listener that prints `<name> got <event>`; `unsub <name>` runs that listener's returned `Runnable`; `pub <event>` publishes. Print `listeners=<count>` at the end.

Example: `5` then `sub a`, `sub b`, `pub hello`, `unsub a`, `pub bye` →

```
a got hello
b got hello
b got bye
listeners=1
```

INPUT 1

```
5
sub a
sub b
pub hello
unsub a
pub bye
```

OUTPUT 1

```
a got hello
b got hello
b got bye
listeners=1
```

INPUT 2

```
3
pub nobody
sub x
unsub x
```

OUTPUT 2

```
listeners=0
```

REFERENCE SOLUTION

```
import java.util.*;

@FunctionalInterface
interface Listener {
    void on(String event);
}

class Bus {
    private final List<Listener> listeners = new ArrayList<>();
    Runnable subscribe(Listener l) {
        listeners.add(l);
        return () -> listeners.remove(l);
    }
    void publish(String event) {
        for (Listener l : new ArrayList<>(listeners)) l.on(event);
    }
    int count() { return listeners.size(); }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Bus bus = new Bus();
        Map<String, Runnable> unsubscribers = new HashMap<>();
        for (int i = 0; i < n; i++) {
            String cmd = in.next(), arg = in.next();
            switch (cmd) {
                case "sub" -> unsubscribers.put(arg, bus.subscribe(event -> System.o
                case "unsub" -> { Runnable r = unsubscribers.remove(arg); if (r != r
                default -> bus.publish(arg);
            }
        }
        System.out.println("listeners=" + bus.count());
    }
}
```

CHECKPOINT QUESTIONS

1. Implementing an interface method with package-private access…

A. Is fine

B. Is a compile error — interface methods are public and access cannot be reduced

C. Makes the method default

D. Is allowed in the same package

2. `interface A { default void m(){} } interface B extends A { default void m(){} } class C implements A, B {}` — `new C().m()` calls…

A. A's

B. B's — the more specific interface wins

C. Compile error

D. Neither

3. Which cannot be declared in an interface?

A. A `default` method

B. A `static` method

C. An instance field

D. A `private static` method

4. `Comparator<T>` declares `boolean equals(Object)` abstractly and yet is functional because…

A. It has no other methods

B. Redeclared public `Object` methods do not count toward the single abstract method

C. `equals` is a default

D. It is an exception in the compiler

5. Which functional interface matches `(int a, int b) → a + b` for `Integer`s?

A. `Function<Integer, Integer>`

B. `BinaryOperator<Integer>`

C. `Supplier<Integer>`

D. `Predicate<Integer>`

6. `Collections.sort(list)` on a `List<Track>` requires…

A. `Track` to override `equals`

B. `Track` to implement `Comparable<Track>`

C. A `Comparator` always

D. `Track` to be final

7. A `TreeSet<BigDecimal>` given `2.0` and `2.00` contains…

A. Two elements

B. One element — `compareTo` treats them as equal even though `equals` does not

C. Zero

D. It throws

8. An `Iterable` whose `iterator()` returns the **same** iterator object every time…

A. Is fine

B. Breaks a second for-each loop, which finds it already exhausted

C. Is faster

D. Is required

9. A sealed interface with `permits` omitted…

A. Is a compile error

B. Is allowed when all implementors are in the same source file; they are inferred

C. Permits everything

D. Permits nothing

10. Modelling `Ok(value) | Err(message)` as a sealed interface of records avoids…

A. Generics

B. Exceptions for expected failures and nullable fields for the missing half

C. Interfaces

D. Records

11. Depending on interfaces at boundaries (`Clock` , `Repository`) primarily improves…

A. Run-time speed

B. Testability and the ability to swap implementations

C. Memory use

D. Compile time

12. `default String toString() { … }` inside an interface is…

A. Valid

B. A compile error — `Object` methods cannot be defaulted

C. Valid only with `@Override`

D. Ignored

ANSWER KEY · CHECKPOINT — INTERFACES

1. B "attempting to assign weaker access privileges; was public" — add `public`.

2. B Rule 2 of default resolution: a subinterface's default overrides its parent's. No override in `C` is required.

3. C Interfaces hold no per-instance state. Constants are the only fields.

4. B Every implementation already inherits `equals` from `Object`, so the redeclaration does not need a lambda body.

5. B Two inputs and an output all of the same type is `BinaryOperator<T>` — `BiFunction<T, T, T>`.

6. B The no-comparator sort uses the natural order; without `Comparable` it is a compile error.

7. B Sorted collections use `compareTo`; this is the classic `compareTo` / `equals` inconsistency.

8. B Each `iterator()` call must start from the beginning; keep the cursor in the iterator, not the iterable.

9. B Same-file implementors are found by the compiler; across files the list must be explicit.

10. B The type states that a result is exactly one shape or the other, and the caller must handle both.

11. B A test supplies a fake implementation; production supplies the real one; the logic between them does not change.

12. B Class methods always beat interface defaults, so such a default could never be selected; the compiler rejects it.