Module 11 of 20

Nested classes & lambdas

Static nested and inner classes, local and anonymous classes, lambda expressions, method references, and functions as values.

Java has four kinds of nested class and, since Java 8, lambda expressions — five ways to put a piece of behaviour exactly where it is needed. Each has precise rules about what it can see (captured variables, the enclosing instance, this) and each is the right tool for a different job.

This module covers static nested versus inner classes and the hidden outer reference that leaks memory; local and anonymous classes and the effectively-final rule; lambda syntax, target typing, captures and this; the four kinds of method reference; and the techniques functions-as-values make possible — composition, closure factories, strategy maps, callbacks and lazy suppliers.

By the end you choose the right nested form automatically, write and read lambdas and method references fluently, and use functions as first-class building blocks.

CONTENTS

1	Static nested and inner classes	13 min
2	Local and anonymous classes	11 min
3	Lambda expressions	15 min
4	Method references — the four kinds	11 min
5	Functions as values — composition, factories and strategies	13 min
✓	Checkpoint — Nested classes & lambdas	22 min

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

LESSON 1 · 13 MIN

Static nested and inner classes

A class can be declared inside another class. Java has four kinds of such *nested* classes — static nested, inner (non-static member), local and anonymous — and the first two differ in one crucial way: an **inner** class instance secretly holds a reference to an instance of the outer class; a **static nested** class does not. That reference is convenient, and it is also the cause of memory leaks and confusing `this` rules. This lesson makes the distinction sharp and gives the rule for which to use.

Static nested classes

```
public class LinkedStack<T> {
    private static class Node<T> {            // static nested: a helper type
        final T value;
        final Node<T> next;
        Node(T value, Node<T> next) { this.value = value; this.next = next; }
    }

    private Node<T> head;

    public void push(T v) { head = new Node<>(v, head); }
    public T pop() { T v = head.value; head = head.next; return v; }
}
```

A static nested class is an ordinary class that happens to live inside another, for scoping and packaging. It has **no reference to an outer instance**; it can be created without one (`new LinkedStack.Node<>(…)` from outside, if accessible, or just `new Node<>(…)` inside). It can access the outer class's `static` members and — because nesting is a trust relationship — the outer class's `private` static members. The outer class can access the nested class's private members too.

Use it for: helper types that belong to one class (`Map.Entry`, `HashMap.Node`, builders — `Pizza.Builder`), and for grouping small related types. This should be your **default** kind of nested class.

Inner classes (non-static member classes)

```
public class Outer {
    private int count = 0;

    public class Counter {                    // inner: every Counter belongs
        public void increment() { count++; }   // reads the OUTER instance's
        public Outer owner() { return Outer.this; }
    }

    public Counter counter() { return new Counter(); }    // created inside a
}

Outer o = new Outer();
Outer.Counter c = o.counter();
Outer.Counter c2 = o.new Counter();           // explicit outer instance — ra
c.increment();                                // o.count is now 1
```

An inner class instance carries a hidden field, `Outer.this`, pointing at the outer instance it was created from. Inside it, unqualified names resolve outward: `count` means `Outer.this.count`. `this` alone is the inner instance; `Outer.this` is the outer.

Consequences:

- Cannot be created without an outer instance (`new Counter()` from a static method fails: "an enclosing instance that contains Outer.Counter is required").
- Before Java 16, inner classes could not declare static members (except constants); since Java 16 they can.
- **Every inner instance keeps its outer instance alive.** A `Counter` stored in a long-lived collection prevents its `Outer` from being garbage-collected — the classic Java memory leak, historically in Android with inner-class listeners holding `Activity`s.

Use it when the inner object genuinely *is part of* the outer's state and needs its fields: an `Iterator` implementation inside a collection (`ArrayList.Itr` is an inner class reading `ArrayList`'s array and `modCount`), an event-handler that manipulates the enclosing component.

The rule

Effective Java item 24: **if a nested class does not need access to the enclosing instance, make it static.** Every non-static nested class costs a hidden field, a hidden constructor parameter, and a lifetime dependency. IDEs warn "inner class may be static" for exactly this reason. Start with `static`; remove it only when you need `Outer.this`.

Nested types of other kinds

Nested interfaces, enums and records are implicitly `static` — they can never be inner classes.

A nested class inside an **interface** is implicitly `public static`.

Nesting depth is unlimited; more than two levels is a design smell.

Access, visibility and naming

Nested classes may use all four access levels (top-level classes only two). A private nested class is visible only within the top-level class that contains it — including to sibling nested classes. From outside, the name is `Outer.Nested`; the compiled class file is `Outer$Nested.class`, which you will see in stack traces and `javap` output. Anonymous classes compile to `Outer$1.class`, `Outer$2.class`.

Shadowing outward

```
public class Outer {
    int x = 1;
    class Inner {
        int x = 2;
        void show(int x) {
            System.out.println(x);            // 3: the parameter
            System.out.println(this.x);       // 2: Inner's field
            System.out.println(Outer.this.x); // 1: Outer's field
        }
    }
}
```

Names resolve innermost first; qualify to reach outer levels. Avoid shadowing across nesting levels — it is legal and unreadable.

Inner classes and inheritance

An inner class can extend another class and be extended; a subclass of an inner class must also have an outer instance, which makes such hierarchies awkward. Interfaces implemented by inner classes (the `Iterator` case) are the common form; inheritance among inner classes is rare and should stay rare.

Interview angle

"Difference between a static nested class and an inner class?" An inner class instance holds a reference to an enclosing instance and can use its fields; a static nested class does not and can be created without one.

"How do you create an inner class instance from outside?" `outer.new Inner()`.

"Why can inner classes cause memory leaks?" The hidden `Outer.this` reference keeps the outer object reachable.

"Which should you prefer?" Static nested, unless the enclosing instance is genuinely needed.

"Can an inner class have static members?" Since Java 16, yes; before, only compile-time constants.

Key takeaways

Static nested: a class in another's namespace, no outer instance — the default choice.

Inner: carries `Outer.this`, reads outer fields directly, needs an outer instance, keeps it alive.

Nested interfaces/enums/records are always static; nested classes may be private.

`Outer.this.x` reaches the outer field when names shadow; `outer.new Inner()` creates with an explicit outer.

EXERCISE 1

A list with an inner iterator

Write `class IntList` backed by a growing `int[]` with `add(int)` and `size()`, a **static nested** class `Stats` (fields `min`, `max`, `sum`; a `toString` of `min=<m> max=<M> sum=<s>`) returned by

`stats()`, and an **inner** class `Cursor` (created by `cursor()`) with `hasNext()` / `next()` that walks the list reading the outer object's array and size directly.

Read an integer `n` and `n` integers, add them, print the elements via a `Cursor` space-separated, then print `stats()`. For an empty list print an empty line and `min=0 max=0 sum=0`.

Example: `3` then `4 -2 9` →

```
4 -2 9
min=-2 max=9 sum=11
```

INPUT 1

```
3
4 -2 9
```

OUTPUT 1

```
4 -2 9
min=-2 max=9 sum=11
```

INPUT 2

```
0
```

OUTPUT 2

```

min=0 max=0 sum=0
```

REFERENCE SOLUTION

```
import java.util.*;

class IntList {
    private int[] items = new int[4];
    private int size = 0;

    static class Stats {
        final long min, max, sum;
        Stats(long min, long max, long sum) { this.min = min; this.max = max; this.s
        @Override public String toString() { return "min=" + min + " max=" + max + '
    }

    class Cursor {
        private int index = 0;
        boolean hasNext() { return index < size; }
        int next() { return items[index++]; }
    }

    void add(int v) {
        if (size == items.length) items = Arrays.copyOf(items, size * 2);
        items[size++] = v;
    }

    int size() { return size; }
    Cursor cursor() { return new Cursor(); }

    Stats stats() {
        if (size == 0) return new Stats(0, 0, 0);
        long min = items[0], max = items[0], sum = 0;
        for (int i = 0; i < size; i++) {
            min = Math.min(min, items[i]);
            max = Math.max(max, items[i]);
            sum += items[i];
        }
        return new Stats(min, max, sum);
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        IntList list = new IntList();
        for (int i = 0; i < n; i++) list.add(in.nextInt());
        StringBuilder sb = new StringBuilder();
        IntList.Cursor c = list.cursor();
        while (c.hasNext()) {
            if (sb.length() > 0) sb.append(' ');
            sb.append(c.next());
```

```
        }
        System.out.println(sb);
        System.out.println(list.stats());
    }
}
```

EXERCISE 2

A linked list with a nested Node

Write `class IntList` with a **private static nested** `class Node { int value; Node next; }`, and methods `addLast(int)`, `addFirst(int)`, `int size()` and `String toString()` giving `[a, b, c]`. Read `n` commands `first <v>`, `last <v>` or `show` (prints the list and `size=<n>`). The node type is an implementation detail — nothing outside the list can name it.

Example: `4` then `last 2`, `last 3`, `first 1`, `show` →

```
[1, 2, 3] size=3
```

INPUT 1

```
4
last 2
last 3
first 1
show
```

OUTPUT 1

```
[1, 2, 3] size=3
```

INPUT 2

```
1
show
```

OUTPUT 2

```
[] size=0
```

REFERENCE SOLUTION

```
import java.util.*;

class IntList {
    private static class Node {
        int value;
        Node next;
        Node(int value, Node next) { this.value = value; this.next = next; }
    }
    private Node head;
    private int size;
    void addFirst(int v) { head = new Node(v, head); size++; }
    void addLast(int v) {
        if (head == null) { addFirst(v); return; }
        Node cur = head;
        while (cur.next != null) cur = cur.next;
        cur.next = new Node(v, null);
        size++;
    }
    int size() { return size; }
    @Override public String toString() {
        StringJoiner sj = new StringJoiner(", ", "[", "]");
        for (Node cur = head; cur != null; cur = cur.next) sj.add(String.valueOf(cur
        return sj.toString();
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        IntList list = new IntList();
        for (int i = 0; i < n; i++) {
            String cmd = in.next();
            switch (cmd) {
                case "first" -> list.addFirst(in.nextInt());
                case "last" -> list.addLast(in.nextInt());
                default -> System.out.println(list + " size=" + list.size());
            }
        }
    }
}
```

QUIZ

1. An inner (non-static nested) class instance holds…

A. Nothing special

B. A hidden reference to the enclosing instance (`Outer.this`)

C. A copy of the outer's fields

D. A static reference to the outer class

2. Which nested class should be your default?

A. Inner

B. Static nested — unless the enclosing instance is genuinely needed

C. Anonymous

D. Local

3. How do you instantiate an inner class from outside the outer class?

A. `new Outer.Inner()`

B. `outer.new Inner()`

C. `Outer.Inner.create()`

D. It is impossible

4. Nested interfaces, enums and records are…

A. Inner

B. Implicitly static

C. Not allowed

D. Private by default

5. Inside an inner class, `Outer.this.x` refers to…

A. The inner class's field `x`

B. The enclosing instance's field `x`

C. A static field

D. A compile error

ANSWER KEY · STATIC NESTED AND INNER CLASSES

1. B That reference is how it reads outer fields directly — and why it keeps the outer object alive.

2. B A static nested class has no hidden field, no lifetime dependency and can be created without an outer instance.

3. B The outer instance must be supplied; the syntax is `outerInstance.new Inner()`.

4. B They never carry an enclosing instance.

5. B Qualified `this` reaches outer levels when names shadow.

LESSON 2 · 11 MIN

Local and anonymous classes

Two more places a class can be declared: inside a method (a **local class**) and inline at the point of use, without a name (an **anonymous class**). Both can capture variables from the surrounding method, which is where the *effectively final* rule comes from, and both were the way Java passed behaviour around before lambdas. You will read anonymous classes in every codebase older than 2014 and still write them when a lambda cannot do the job.

Local classes

```
static List<String> longestWords(List<String> words) {
    final int limit = 5;

    class LengthFilter {                              // visible only inside t
        boolean keep(String w) { return w.length() > limit; }   // captures li
    }

    LengthFilter f = new LengthFilter();
    List<String> out = new ArrayList<>();
    for (String w : words) if (f.keep(w)) out.add(w);
    return out;
}
```

A local class is declared in a block and scoped to it. It may implement interfaces and extend classes, have constructors and multiple methods — a full class, just private to the method. Declared in an instance method it is an inner class (has `Outer.this`); in a static method it has no outer instance. Local records, enums and interfaces (Java 16) are implicitly static.

Use it when a method needs a small helper type with more than one method or with state — and the type is meaningless elsewhere.

Anonymous classes

```
Comparator<String> byLength = new Comparator<String>() {        // declares ANI
    @Override
    public int compare(String a, String b) {
        return Integer.compare(a.length(), b.length());
    }
};

Runnable task = new Runnable() {
    @Override public void run() { System.out.println("working"); }
};

Thread t = new Thread() {                                        // extending c
    @Override public void run() { … }
};
```

`new Type() { body }` creates a class with no name that extends `Type` (a class) or implements it (an interface), and immediately creates one instance. The body may override methods, add fields and methods (usable only inside the body — the outside knows only `Type`), and have instance initialiser blocks — but **no constructor** (there is no name to give it) and no static members except constants. Arguments in the parentheses go to the superclass constructor.

Like local classes, anonymous classes declared in instance code are inner classes with a hidden `Outer.this`; `this` inside the body refers to the anonymous instance, not the enclosing object — the main behavioural difference from lambdas.

Capturing variables: effectively final

Both local and anonymous classes can use the enclosing method's parameters and locals — but only if those variables are **effectively final** (never reassigned after initialisation):

```
int count = 0;
Runnable r = new Runnable() {
    public void run() { count++; }      // compile error: local variables refe
};
```

Why: the class instance may outlive the method (returned, stored, run on another thread), while the local lives in the method's stack frame. Java resolves this by **copying** the captured value into a hidden field of the instance at creation. A copy of a variable that could later change would be a lie, so the language forbids the change. To mutate captured state, use a

field of the enclosing object, a one-element array, or `AtomicInteger` / `AtomicReference` — each an object whose *reference* is effectively final while its contents change.

Fields of the enclosing instance are not copies — they are reached through `Outer.this` — so they can be freely read and written.

Where anonymous classes still win over lambdas

Situation	Why a lambda cannot
The type has **more than one abstract method**	Lambdas need a single abstract method
Extending an **abstract class** or a concrete class	Lambdas only implement interfaces
You need `this` **to mean the new object** (call its own other methods, pass itself as a listener)	In a lambda `this` is the enclosing instance
You need **instance state** in the object	A lambda has no fields
Overriding **several** methods of an interface (`Iterator` with `hasNext` / `next`)	One method only

`Iterator` implementations, `TimerTask` / `Thread` subclasses, `AbstractList` one-offs, and listeners with two callbacks are the everyday cases.

Reading old code

Pre-Java-8 code passes behaviour as anonymous classes everywhere: `new Comparator<…>() { … }`, `new Runnable() { … }`, `new ActionListener() { … }`. Mentally translate each one-method anonymous class to a lambda; that is precisely what the IDE's "convert to lambda" refactoring does, and it is safe whenever the body does not use `this` or declare fields.

Compiled form and identity

Each anonymous class becomes `Outer$1.class`, `Outer$2.class` … — one class file per occurrence, loaded at first use. `getClass().getName()` shows the `$1`;

`getClass().getSimpleName()` is empty. Two anonymous instances from the same site have the same class; instances from different sites have different classes even if the bodies are identical.

Interview angle

"Can an anonymous class have a constructor?" No; use an instance initialiser block, or pass arguments to the superclass constructor.

"Why must captured locals be effectively final?" They are copied into the instance; a later change would not be seen.

"When would you use an anonymous class instead of a lambda?" Multiple abstract methods, abstract classes, needing `this` as the object, needing state.

"What is a local class?" A named class declared inside a method, scoped to it.

Key takeaways

Local classes: named, method-scoped, full classes. Anonymous classes: declared and instantiated inline, extending a class or implementing an interface, no constructor.

Both capture enclosing locals by copy — hence effectively final; enclosing fields are shared via `Outer.this`.

In an anonymous class `this` is the anonymous object; in a lambda it is the enclosing instance.

Prefer lambdas for single-method interfaces; anonymous classes for everything else.

EXERCISE 1

An anonymous comparator and a counting iterator

Read an integer `n` and `n` words. Sort them with an **anonymous** `Comparator<String>` by length ascending, then alphabetically, and print them space-separated. Then build an anonymous `Iterator<String>` over the sorted list that yields only words longer than `k` (read `k` after the words) and print those, space-separated (empty line if none), followed by `count=<how many>` — count them with an `int[] counter = {0}` captured by the iterator.

Example: `4` then `pear fig apple kiwi`, `k = 3` →

```
fig kiwi pear apple
kiwi pear apple
count=3
```

INPUT 1

```
4
pear fig apple kiwi
3
```

OUTPUT 1

```
fig kiwi pear apple
kiwi pear apple
count=3
```

INPUT 2

```
3
a bb cc
5
```

OUTPUT 2

```
a bb cc

count=0
```

REFERENCE SOLUTION

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<String> words = new ArrayList<>();
        for (int i = 0; i < n; i++) words.add(in.next());
        int k = in.nextInt();

        words.sort(new Comparator<String>() {
            @Override
            public int compare(String a, String b) {
                int c = Integer.compare(a.length(), b.length());
                return c != 0 ? c : a.compareTo(b);
            }
        });
        System.out.println(String.join(" ", words));

        int[] counter = {0};
        Iterator<String> longOnes = new Iterator<String>() {
            private int i = 0;
            private void skip() { while (i < words.size() && words.get(i).length() <
            @Override public boolean hasNext() { skip(); return i < words.size(); }
            @Override public String next() {
                skip();
                if (i >= words.size()) throw new NoSuchElementException();
                counter[0]++;
                return words.get(i++);
            }
        };
        StringBuilder sb = new StringBuilder();
        while (longOnes.hasNext()) {
            if (sb.length() > 0) sb.append(' ');
            sb.append(longOnes.next());
        }
        System.out.println(sb);
        System.out.println("count=" + counter[0]);
    }
}
```

EXERCISE 2

Anonymous and local, side by side

Read `n` words. (1) Create an **anonymous** `Runnable` that prints `ran with <n> words`, and run it. (2) Inside `main`, declare a **local** class `Tally` with a counter and `void see(String w)` that increments

when the word starts with an upper-case letter; feed it every word and print `capitalised=<count>`. (3) Sort the words with an anonymous `Comparator<String>` by length descending, then alphabetically, and print them space-separated.

Example: `4` then `pear Apple fig Kiwi` →

```
ran with 4 words
capitalised=2
Apple Kiwi pear fig
```

INPUT 1

```
4
pear Apple fig Kiwi
```

OUTPUT 1

```
ran with 4 words
capitalised=2
Apple Kiwi pear fig
```

INPUT 2

```
1
z
```

OUTPUT 2

```
ran with 1 words
capitalised=0
z
```

REFERENCE SOLUTION

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<String> words = new ArrayList<>();
        for (int i = 0; i < n; i++) words.add(in.next());

        Runnable announce = new Runnable() {
            @Override public void run() { System.out.println("ran with " + n + " wor
        };
        announce.run();

        class Tally {
            int count;
            void see(String w) { if (!w.isEmpty() && Character.isUpperCase(w.charAt(
        }
        Tally tally = new Tally();
        for (String w : words) tally.see(w);
        System.out.println("capitalised=" + tally.count);

        words.sort(new Comparator<String>() {
            @Override public int compare(String a, String b) {
                int c = Integer.compare(b.length(), a.length());
                return c != 0 ? c : a.compareTo(b);
            }
        });
        System.out.println(String.join(" ", words));
    }
}
```

QUIZ

1. An anonymous class cannot declare…

A. Fields

B. Methods beyond the interface's

C. A constructor

D. An instance initialiser block

2. `int count = 0; Runnable r = new Runnable() { public void run() { count++; } };` fails because…

A. `count` is not a field

B. Captured locals must be effectively final — they are copied into the instance

C. `Runnable` cannot capture

D. `count++` is not a statement

3. Inside an anonymous class body, `this` refers to…

A. The enclosing instance

B. The anonymous object itself

C. The interface

D. null

4. When is an anonymous class required instead of a lambda?

A. Never

B. When the type has several abstract methods, is a class, or the object needs its own state or `this`

C. Only for `Runnable`

D. When the body is long

5. A local class declared inside an instance method…

A. Cannot use the enclosing object's fields

B. Is an inner class with access to `Outer.this` and effectively-final locals

C. Must be static

D. Is visible to the whole class

ANSWER KEY · LOCAL AND ANONYMOUS CLASSES

1. C It has no name to give a constructor; use an initialiser block or superclass constructor arguments.

2. B The object may outlive the method's frame; Java copies the value, so it forbids later changes.

3. B Unlike a lambda. Use `Outer.this` for the enclosing instance.

4. B Lambdas implement one interface method and have no fields; anonymous classes are full classes.

5. B It is scoped to the block, inner if the method is an instance method, and captures locals by copy.

LESSON 3 · 15 MIN

Lambda expressions

A lambda expression is an anonymous function: parameters, an arrow, a body. Its type is whatever functional interface the context expects, and it compiles to something lighter than an anonymous class. Lambdas are the syntax behind streams, comparators, executors, callbacks and most Java written since 2014. This lesson covers every form of the syntax, how the compiler assigns a type, what `this` and captured variables mean inside one, and the rules for keeping lambdas readable.

Syntax

```
() → 42                                  // no parameters, expression body
x → x * 2                                // one parameter, parentheses optio
(x, y) → x + y                           // two parameters
(int x, int y) → x + y                   // explicit types (all or none)
(var x, var y) → x + y                   // var for all (Java 11) — lets you
s → { return s.length(); }               // block body with return
s → { System.out.println(s); }           // block body, void
(a, b) → {                               // multi-statement block
    int diff = a.length() - b.length();
    return diff ≠ 0 ? diff : a.compareTo(b);
}
```

An **expression body** yields its value (or is a `void` statement expression like a method call). A **block body** needs `return` for a value. Parameter types are usually inferred; write them when inference fails or for clarity. The parameter names must not clash with locals in the enclosing scope — lambda parameters are *not* a new scope for shadowing (unlike anonymous class methods).

Target typing

A lambda has no type of its own. The compiler takes the **target type** from the context — an assignment, a method parameter, a return statement, a cast — finds the functional interface's single abstract method, and checks the lambda against it:

```
Function<String, Integer> len = s → s.length();          // target: Function<
Predicate<String> empty = s → s.isEmpty();               // same lambda shape
Runnable r = () → System.out.println("hi");              // target from the c
list.sort((a, b) → a.compareTo(b));                      // target from the p
return x → x + 1;                                        // target from the r
Object o = (Runnable) () → {};                           // a cast supplies t
Object bad = () → {};                                    // compile error: Ob
```

The same text can become a `Callable<Integer>` or a `Supplier<Integer>` depending on where it appears; ambiguity between overloads (`submit(Runnable)` vs `submit(Callable)`) is resolved by whether the body returns a value — and sometimes needs a cast to settle.

Captures and effectively final

```
int base = 10;
Function<Integer, Integer> addBase = x → x + base;      // captures base
base = 20;                                               // compile error at
```

Same rule as anonymous classes, same reason: captured locals are copied. Fields, `this`, and the contents of captured objects (arrays, lists, atomics) can be read and written freely. The idiom for a counter inside a lambda is `int[] count = {0}; … count[0]++` or an `AtomicInteger` — but if you find yourself doing that in a stream, the stream is probably the wrong tool (streams want stateless lambdas).

`this` inside a lambda

```
public class Greeter {
    private String name = "outer";
    void run() {
        Runnable lambda = () → System.out.println(this.name);          // '
        Runnable anon = new Runnable() {
            String name = "anon";
            public void run() { System.out.println(this.name); }       // '
        };
    }
}
```

A lambda introduces **no new `this`**; it is lexically part of the enclosing method. This is the single most important behavioural difference from anonymous classes, and it is what makes lambdas natural for callbacks that touch the enclosing object.

How lambdas compile

Not to a class file per lambda. `javac` emits a private synthetic method holding the body and an `invokedynamic` instruction; on first execution the JVM's `LambdaMetafactory` spins a small class implementing the interface. Non-capturing lambdas are typically a single shared instance; capturing ones allocate per evaluation. Practical consequences: lambdas start cheaper than anonymous classes, do not pollute the class file list, and their `toString` / `getClass` are unreadable (`Main$$Lambda$14/0x…`).

Exceptions in lambdas

A lambda may throw only what its target method declares. `Runnable.run` and `Function.apply` declare nothing, so a checked exception inside must be caught in the body (or wrapped in an unchecked one). `Callable.call() throws Exception` accepts anything. This is the reason many stream pipelines have a `try/catch` inside a `map` or a small helper that rethrows as unchecked (Module 12).

Readability rules

One line if possible. A lambda longer than three lines wants to be a named method (`this::process`) — the name documents it, and it can be tested.

Descriptive parameter names in longer lambdas (`(order, line) →`), single letters in trivial ones (`x → x * 2`).

No side effects in stream lambdas; use them in `forEach` only.

Prefer method references when the lambda only calls one method (next lesson).

Do not overload methods on functional interfaces with the same arity (`f(Runnable)` and `f(Callable)`): callers need casts.

Lambdas versus anonymous classes, summarised

	Lambda	Anonymous class
Implements	one abstract method of an interface	any class or interface, any number of methods
`this`	enclosing instance	the anonymous object
State	none (captures only)	fields allowed
Cost	invokedynamic; shared if non-capturing	a class file and an object per creation
Shadowing params	not allowed	allowed

Interview angle

"What is the type of a lambda?" Whatever functional interface the context requires — it has no intrinsic type.

"What does `this` refer to inside a lambda?" The enclosing instance.

"Can a lambda modify a captured local?" No; it must be effectively final. Use a field or a holder object.

"Are lambdas compiled to anonymous classes?" No — `invokedynamic` and a runtime-generated class; non-capturing ones are shared.

"Can a lambda throw a checked exception?" Only if the functional interface's method declares it.

Key takeaways

`(params) → expression | { block }`; types inferred from the target functional interface.

Captured locals must be effectively final; `this` is the enclosing object, not the lambda.

Checked exceptions inside need the SAM to declare them or a catch in the body.

Keep lambdas short and stateless; extract longer ones into named methods.

EXERCISE 1

Lambdas in four shapes

Read an integer `n` and `n` integers. Using **lambdas** assigned to typed variables, build: a `Predicate<Integer>` for `even`, a `Function<Integer, Integer>` `square`, a `BinaryOperator<Integer>` `max`, and a `Supplier<String>` that returns `"n=" + n` (capturing `n`). Print four lines: `evens=<count of evens>`, `squares=<sum of squares as long>`, `max=<max via the operator, or "none" if n is 0>`, and the supplier's value.

Example: `3` then `2 3 4` →

```
evens=2
squares=29
max=4
n=3
```

INPUT 1

```
3
2 3 4
```

OUTPUT 1

```
evens=2
squares=29
max=4
n=3
```

INPUT 2

```
0
```

OUTPUT 2

```
evens=0
squares=0
max=none
n=0
```

REFERENCE SOLUTION

```java
import java.util.*;
import java.util.function.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        int[] a = new int[n];
        for (int i = 0; i < n; i++) a[i] = in.nextInt();

        Predicate<Integer> even = x -> x % 2 == 0;
        Function<Integer, Integer> square = x -> x * x;
        BinaryOperator<Integer> max = (x, y) -> x > y ? x : y;
        Supplier<String> label = () -> "n=" + n;

        int evens = 0;
        long squares = 0;
        Integer best = null;
        for (int v : a) {
            if (even.test(v)) evens++;
            squares += square.apply(v);
            best = best == null ? v : max.apply(best, v);
        }
        System.out.println("evens=" + evens);
        System.out.println("squares=" + squares);
        System.out.println("max=" + (best == null ? "none" : best));
        System.out.println(label.get());
    }
}
```

EXERCISE 2

Predicate chains

Read `n` integers. Build `Predicate<Integer>` lambdas `even`, `positive` and `big` (`> 100`), then print counts of the numbers that satisfy: `even.and(positive)`, `even.or(big)`, `positive.negate()`, and `Predicate.not(big).and(even.negate())`. Format each as `<name>=<count>`.

Example: `6` then `-4 3 8 150 -7 0` →

```
evenAndPositive=2
evenOrBig=4
notPositive=3
smallAndOdd=2
```

INPUT 1

```
6
-4 3 8 150 -7 0
```

OUTPUT 1

```
evenAndPositive=2
evenOrBig=4
notPositive=3
smallAndOdd=2
```

INPUT 2

```
2
101 102
```

OUTPUT 2

```
evenAndPositive=1
evenOrBig=2
notPositive=0
smallAndOdd=0
```

REFERENCE SOLUTION

```
import java.util.*;
import java.util.function.*;

public class Main {
    static long count(List<Integer> xs, Predicate<Integer> p) {
        long c = 0;
        for (int x : xs) if (p.test(x)) c++;
        return c;
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<Integer> xs = new ArrayList<>();
        for (int i = 0; i < n; i++) xs.add(in.nextInt());
        Predicate<Integer> even = x -> x % 2 == 0;
        Predicate<Integer> positive = x -> x > 0;
        Predicate<Integer> big = x -> x > 100;
        System.out.println("evenAndPositive=" + count(xs, even.and(positive)));
        System.out.println("evenOrBig=" + count(xs, even.or(big)));
        System.out.println("notPositive=" + count(xs, positive.negate()));
        System.out.println("smallAndOdd=" + count(xs, Predicate.not(big).and(even.ne
    }
}
```

QUIZ

1. A lambda's type is determined by…

A. Its parameter names

B. The target functional interface of the context it appears in

C. Its return statement

D. The JVM at run time

2. Inside a lambda, `this` refers to…

A. The lambda object

B. The enclosing instance

C. The functional interface

D. Nothing — it is an error

3. `Object o = () -> {};` is…

A. Valid

B. A compile error — `Object` is not a functional interface

C. A `Runnable`

D. Valid with a warning

4. A lambda that calls a method throwing `IOException`, targeting `Function`, must…

A. Declare `throws`

B. Catch the exception inside (or wrap it as unchecked)

C. Use `Callable` only

D. Nothing — it compiles

5. Non-capturing lambdas compile to…

A. One anonymous class file each

B. An `invokedynamic` site whose instance is typically shared across evaluations

C. A static method only

D. Reflection calls

ANSWER KEY · LAMBDA EXPRESSIONS

1. B The same lambda text can be a `Predicate`, a `Function` or a `Callable` depending on where it is assigned or passed.

2. B A lambda introduces no new `this`; it is lexically part of the enclosing method.

3. B The target type must be a functional interface; cast to `Runnable` to supply one.

4. B `Function.apply` declares no checked exceptions, so the body cannot let one escape.

5. B `LambdaMetafactory` spins a class at first use; without captured state, one instance can be reused.

LESSON 4 · 11 MIN

Method references — the four kinds

When a lambda does nothing but call one existing method — `s → s.length()`, `x → Integer.parseInt(x)`, `() → new ArrayList<>()` — Java lets you name the method instead with `::`. A **method reference** is exactly as expressive as the lambda it replaces, slightly shorter, and often clearer because the reader sees a name rather than a pipeline of parameters. There are four kinds, and knowing which one you are looking at is what makes `String::compareToIgnoreCase` and `this::process` readable.

The four kinds

Kind	Syntax	Equivalent lambda	Example
Static	`Class::staticMethod`	`(args) → Class.staticMethod(args)`	`Integer::parseInt`, `Math::max`
Bound (a particular object)	`object::method`	`(args) → object.method(args)`	`System.out::println`, `this::process`, `str::equals`
Unbound (first argument is the receiver)	`Class::instanceMethod`	`(obj, args) → obj.method(args)`	`String::length`, `String::compareTo`
Constructor	`Class::new`, `Type[]::new`	`(args) → new Class(args)`	`ArrayList::new`, `int[]::new`, `Person::new`

```
Function<String, Integer> parse = Integer::parseInt;          // static: s → In
Consumer<Object> print = System.out::println;                 // bound: x → Sys
Function<String, Integer> length = String::length;            // unbound: s → s
Comparator<String> cmp = String::compareToIgnoreCase;         // unbound, two ar
Supplier<List<String>> make = ArrayList::new;                 // constructor: ()
Function<Integer, int[]> array = int[]::new;                  // array construct
BiFunction<String, Integer, Person> build = Person::new;      // picks the (Stri
```

The compiler matches the reference against the target interface's single method: the parameter count and types decide which kind it is and which overload it means. `String::compareTo` targeting `Comparator<String>` (two parameters) is unbound — the first parameter becomes the receiver, the second the argument.

Bound versus unbound

The subtle pair:

```
String prefix = "user_";
Predicate<String> startsWithPrefix = prefix::startsWith;      // BOUND: s → p
Predicate<String> isUser = s → s.startsWith(prefix);          // the lambda yo
BiPredicate<String, String> starts = String::startsWith;      // UNBOUND: (a,
```

`Class::method` with an instance method is unbound — the object comes as the first argument at call time. `variable::method` is bound — the object is captured when the reference is created, and its state at that moment is what the method sees. `System.out::println` is bound to the `PrintStream` that `System.out` held when you wrote it.

Overload resolution with references

A reference names a method, not an overload; the target type selects:

```
Function<String, Integer> a = Integer::valueOf;           // valueOf(String)
IntFunction<Integer> b = Integer::valueOf;                // valueOf(int)
Consumer<String> c = System.out::println;                 // println(String)
```

When two overloads fit equally, the reference is ambiguous and the compiler asks for a lambda with explicit parameter types.

`this::` and `super::`

```
list.forEach(this::log);                    // bound to the current instance
Function<String, String> f = super::describe;   // the superclass's implementa
```

`this::method` is the idiomatic way to pass an instance method as a callback and is what "extract a long lambda into a method" produces.

When to use which

Use a **method reference** when the lambda would be exactly one call with the parameters passed through in order. `x → foo(x)` → `this::foo`; `(a, b) → a.compareTo(b)` → `String::compareTo`.

Use a **lambda** when arguments are rearranged, transformed or partially fixed (`s → s.startsWith(prefix)`), when there is more than one call, or when the reference would be *less* clear (`x → x` beats `Function.identity()` for some readers; `Function.identity()` beats `x → x` in a `toMap`).

Avoid references that look like one kind and are another (`prefix::startsWith` above) — the lambda states the intent.

Readability examples

```
names.stream().map(String::toUpperCase).forEach(System.out::println);
people.sort(Comparator.comparing(Person::lastName).thenComparing(Person::first
Map<String, List<Person>> byCity = people.stream().collect(Collectors.groupin
List<Integer> sizes = lists.stream().map(List::size).toList();
Optional<Person> oldest = people.stream().max(Comparator.comparingInt(Person::
String[] arr = list.toArray(String[]::new);
```

Each reference reads as "the length", "the last name", "print" — the reader is spared the parameter plumbing.

Interview angle

"What are the kinds of method reference?" Static, bound instance, unbound instance, constructor.

"`String::length` — what does the first parameter become?" The receiver: unbound instance method reference.

"Is a method reference a lambda?" Semantically yes — it compiles the same way and needs a functional-interface target.

"When is `Class::method` ambiguous?" When several overloads match the target's parameter types; use a lambda with typed parameters.

Key takeaways

`Class::static`, `obj::instance` (bound: receiver captured now), `Class::instance` (unbound: receiver is the first argument), `Class::new`.

The target functional interface selects the kind and the overload.

Use references for straight pass-through calls; lambdas when arguments are reshaped or intent needs stating.

`this::method` is the callback idiom and the result of extracting a long lambda.

EXERCISE 1

Four references

Read an integer `n` and `n` words. Using **method references** wherever a lambda would only forward its arguments: parse a list of lengths with `String::length` (unbound), a `Comparator<String>` from `String::compareToIgnoreCase` (unbound, two args), an uppercase transform with `String::toUpperCase`, a `Supplier<List<String>>` from `ArrayList::new` (constructor), and print each result line with a bound `System.out::println` through `forEach`.

Output, in order: the lengths space-separated, the words sorted case-insensitively space-separated, the words upper-cased space-separated, and `copies=<size of a fresh list from the supplier after adding all words>`.

Example: `3` then `banana Apple cherry` →

```
6 5 6
Apple banana cherry
BANANA APPLE CHERRY
copies=3
```

INPUT 1

```
3
banana Apple cherry
```

OUTPUT 1

```
6 5 6
Apple banana cherry
BANANA APPLE CHERRY
copies=3
```

INPUT 2

```
1
z
```

OUTPUT 2

```
1
z
Z
copies=1
```

REFERENCE SOLUTION

```
import java.util.*;
import java.util.function.*;
import java.util.stream.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<String> words = new ArrayList<>();
        for (int i = 0; i < n; i++) words.add(in.next());

        Function<String, Integer> length = String::length;
        Comparator<String> ci = String::compareToIgnoreCase;
        UnaryOperator<String> upper = String::toUpperCase;
        Supplier<List<String>> fresh = ArrayList::new;
        Consumer<String> out = System.out::println;

        List<String> lines = new ArrayList<>();
        lines.add(words.stream().map(length).map(String::valueOf).collect(Collectors
        List<String> sorted = new ArrayList<>(words);
        sorted.sort(ci);
        lines.add(String.join(" ", sorted));
        lines.add(words.stream().map(upper).collect(Collectors.joining(" ")));
        List<String> copy = fresh.get();
        copy.addAll(words);
        lines.add("copies=" + copy.size());
        lines.forEach(out);
    }
}
```

EXERCISE 2

Constructor references and friends

Use each kind of method reference at least once. Read `n` words. With `Supplier<List<String>> make = ArrayList::new` create the list; with `Function<String, Integer> len = String::length` map to lengths; with `BiFunction<String, String, Boolean> starts = String::startsWith` test whether each word starts with the first word; with `Function<Integer, String> fmt =`

`Integer::toBinaryString`. Print `lengths=<list of lengths>`, `sharePrefix=<count of words starting with the first word>`, `binary=<binary of the total length>`.

Example: `3` then `ab abc cd` →

```
lengths=[2, 3, 2]
sharePrefix=2
binary=111
```

INPUT 1

```
3
ab abc cd
```

OUTPUT 1

```
lengths=[2, 3, 2]
sharePrefix=2
binary=111
```

INPUT 2

```
1
hello
```

OUTPUT 2

```
lengths=[5]
sharePrefix=1
binary=101
```

REFERENCE SOLUTION

```
import java.util.*;
import java.util.function.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Supplier<List<String>> make = ArrayList::new;
        List<String> words = make.get();
        for (int i = 0; i < n; i++) words.add(in.next());

        Function<String, Integer> len = String::length;
        BiFunction<String, String, Boolean> starts = String::startsWith;
        Function<Integer, String> fmt = Integer::toBinaryString;

        List<Integer> lengths = new ArrayList<>();
        int total = 0, share = 0;
        for (String w : words) {
            lengths.add(len.apply(w));
            total += len.apply(w);
            if (starts.apply(w, words.get(0))) share++;
        }
        System.out.println("lengths=" + lengths);
        System.out.println("sharePrefix=" + share);
        System.out.println("binary=" + fmt.apply(total));
    }
}
```

QUIZ

1. `String::length` used as a `Function<String, Integer>` is which kind of reference?

A. Static

B. Bound instance

C. Unbound instance — the argument becomes the receiver

D. Constructor

2. `prefix::startsWith` (where `prefix` is a `String` variable) as a `Predicate<String>` means…

A. `s → s.startsWith(prefix)`

B. `s → prefix.startsWith(s)` — bound: the receiver is `prefix`

C. A compile error

D. A static call

3. `ArrayList::new` targeting `Supplier<List<String>>` is…

A. Invalid

B. A constructor reference equivalent to `() → new ArrayList<>()`

C. A static method call

D. Bound to an existing list

4. `int[]::new` as a `Function<Integer, int[]>` produces…

A. An empty array

B. `n → new int[n]`

C. A compile error

D. A shared array

5. A method reference is ambiguous when…

A. The method is static

B. Several overloads match the target interface's parameter types

C. It is bound

D. It has no parameters

ANSWER KEY · METHOD REFERENCES — THE FOUR KINDS

1. C `Class::instanceMethod` takes the receiver as the first parameter: `s → s.length()`.

2. B A bound reference captures the object now and passes the argument to its method — often the reverse of what was intended.

3. B The target's parameter list (none) selects the no-arg constructor.

4. B Array constructor references take the length; `toArray(String[]::new)` is the common use.

5. B The compiler then asks for a lambda with explicit parameter types.

LESSON 5 · 13 MIN

Functions as values — composition, factories and strategies

Once behaviour can be stored in a variable, passed to a method and returned from one, a set of techniques opens up that have no equivalent in purely object-oriented code: functions that build functions, pipelines assembled from small pieces, strategies chosen at run time from a map, and callbacks. This lesson is a tour of those techniques as they appear in everyday Java — nothing academic, all of it in real codebases.

Higher-order methods

A method that takes or returns a function:

```
static <T> List<T> filter(List<T> xs, Predicate<T> keep) {
    List<T> out = new ArrayList<>();
    for (T x : xs) if (keep.test(x)) out.add(x);
    return out;
}
static <T, R> List<R> map(List<T> xs, Function<T, R> f) { … }
static <T> void retry(int times, Supplier<T> action) { … }
static <T> T withTiming(String label, Supplier<T> action) {
    long t0 = System.nanoTime();
    try { return action.get(); }
    finally { System.out.println(label + ": " + (System.nanoTime() - t0) / 1_(
}

List<String> longNames = filter(names, n -> n.length() > 5);
Config c = withTiming("load", () -> loadConfig(path));
```

The "wrap an action" shape — timing, retrying, logging, running under a lock, in a transaction — is where `Supplier` / `Runnable` parameters shine: the wrapper owns the setup and teardown, the caller supplies the middle.

Functions that return functions (factories and currying)

```
static Predicate<String> startsWith(String prefix) {
    return s → s.startsWith(prefix);              // the returned lambda capt
}
static Function<Integer, Integer> adder(int n) {
    return x → x + n;
}
static Comparator<Person> byField(String field) {
    return switch (field) {
        case "age" → Comparator.comparingInt(Person::age);
        case "name" → Comparator.comparing(Person::name);
        default → throw new IllegalArgumentException(field);
    };
}

Predicate<String> isTemp = startsWith("tmp_");
Function<Integer, Integer> plus10 = adder(10);
```

A method returning a lambda that captured its arguments is a **closure factory**. `adder(10)` is partial application; a function returning a function of one argument at a time is currying (`x → y → x + y`, type `Function<Integer, Function<Integer, Integer>>`) — rare in Java, but you should recognise it.

Composition

```
Function<String, String> trim = String::trim;
Function<String, String> lower = String::toLowerCase;
Function<String, String> normalise = trim.andThen(lower);             // trim f
Function<String, Integer> normalisedLength = normalise.andThen(String::length)

Predicate<String> notBlank = Predicate.not(String::isBlank);
Predicate<String> valid = notBlank.and(s → s.length() < 100).or(s → s.equals

Comparator<Person> order = Comparator.comparing(Person::lastName)
    .thenComparing(Person::firstName)
    .thenComparingInt(Person::age)
    .reversed();

UnaryOperator<String> pipeline = List.<UnaryOperator<String>>of(String::trim,
    .stream().reduce(UnaryOperator.identity(), (f, g) → s → g.apply(f.apply(
```

`andThen` / `compose` for functions, `and` / `or` / `negate` for predicates, `thenComparing` / `reversed` for comparators — the default methods on the functional interfaces are the composition vocabulary. Small named pieces composed at the use site read better than one large lambda.

The strategy pattern, with lambdas

```
Map<String, BinaryOperator<Long>> ops = Map.of(
    "+", Long::sum,
    "-", (a, b) → a - b,
    "*", (a, b) → a * b,
    "max", Math::max
);
long result = ops.getOrDefault(symbol, (a, b) → { throw new IllegalArgumentEx
```

Where Java 5 code declared an interface, four classes and a factory, a map of lambdas does the same in six lines. The interface still exists (`BinaryOperator`), the strategies are still swappable and testable — they are just not classes. Reach for named classes again when a strategy has state or configuration of its own.

Callbacks and event handling

```
button.onClick(e → save());
executor.submit(() → process(job));
CompletableFuture.supplyAsync(this::fetch).thenApply(this::parse).thenAccept(t
list.removeIf(item → item.isExpired());
map.computeIfAbsent(key, k → new ArrayList<>()).add(value);
map.merge(word, 1, Integer::sum);
```

Every "call me when" API takes a functional interface. The collection APIs' `removeIf` , `replaceAll` , `computeIfAbsent` , `merge` , `forEach` are lambdas replacing loops that used to be five lines with an iterator.

Lazy evaluation with `Supplier`

```
void log(Level level, Supplier<String> message) {
    if (isEnabled(level)) System.out.println(message.get());     // the string
}
log(DEBUG, () → "state=" + expensiveDump());

T orElseGet(Supplier<? extends T> other)     // Optional: computed only when e
```

Passing a `Supplier` instead of a value defers the work until the callee decides it is needed — the difference between `orElse(computeDefault())` (always computes) and `orElseGet(this::computeDefault)` (computes on demand).

Exceptions in functional code

Functional interfaces in `java.util.function` declare no checked exceptions. Options when the body throws one:

- Catch inside the lambda and handle or wrap: `s → { try { return parse(s); } catch (IOException e) { throw new UncheckedIOException(e); } }`.
- Write a small helper interface `ThrowingFunction<T, R, E extends Exception>` and an adapter that wraps to unchecked.
- Restructure so the throwing call happens outside the lambda.

The first is the everyday answer; Module 12 covers the exception side.

Keeping it readable

- Name intermediate functions (`Predicate<String> valid = …`) instead of nesting lambdas three deep.
- Prefer the standard interfaces so callers recognise the shapes.
- Do not build a function pipeline when a plain method with three statements is clearer — functional style is a tool, not a goal.
- Side-effect-free lambdas compose and test easily; lambdas that mutate shared state do not.

Interview angle

"What is a higher-order function?" One that takes or returns a function — `filter(list, predicate)`, `Comparator.comparing`.

"What is a closure in Java?" A lambda that captures variables from its enclosing scope; `adder(10)` returns one.

"How would you implement the strategy pattern today?" A functional interface and lambdas (or a map of them), classes only for stateful strategies.

"Why `orElseGet` over `orElse`?" The supplier is evaluated only when needed.

Key takeaways

Methods can take functions (`filter`, `withTiming`) and return them (`startsWith(prefix)`, `adder(n)`) — closures capture their arguments.

Compose with `andThen` / `compose`, `and` / `or` / `negate`, `thenComparing` / `reversed`.

Strategies, callbacks and collection operations (`removeIf`, `computeIfAbsent`, `merge`) are lambdas now.

`Supplier` defers work; checked exceptions inside lambdas must be caught or wrapped.

EXERCISE 1

Build a text pipeline

Read a first line containing step names separated by spaces, chosen from `trim`, `lower`, `upper`, `snake` (spaces → underscores) and `reverse`; then an integer `n` and `n` lines of text. Build the pipeline **once** as a single `Function<String, String>` by folding the steps with `andThen` (start from `Function.identity()`), then apply it to each line and print the result. Also write a factory `static Function<String, String> step(String name)` returning the lambda for a step name (throw `IllegalArgumentException` for an unknown name).

Example: steps `trim lower snake`, then `2` lines `Hello World` and `Java Rocks` →

```
hello_world
java_rocks
```

INPUT 1

```
trim lower snake
2
  Hello World
Java Rocks
```

OUTPUT 1

```
hello_world
java_rocks
```

INPUT 2

```
reverse upper
1
abc
```

OUTPUT 2

```
CBA
```

REFERENCE SOLUTION

```java
import java.util.*;
import java.util.function.*;

public class Main {
    static Function<String, String> step(String name) {
        return switch (name) {
            case "trim" -> String::trim;
            case "lower" -> String::toLowerCase;
            case "upper" -> String::toUpperCase;
            case "snake" -> s -> s.replace(' ', '_');
            case "reverse" -> s -> new StringBuilder(s).reverse().toString();
            default -> throw new IllegalArgumentException(name);
        };
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String[] names = in.nextLine().trim().split("\\s+");
        Function<String, String> pipeline = Function.identity();
        for (String name : names) pipeline = pipeline.andThen(step(name));
        int n = Integer.parseInt(in.nextLine().trim());
        for (int i = 0; i < n; i++) {
            System.out.println(pipeline.apply(in.nextLine()));
        }
    }
}
```

EXERCISE 2

A pipeline from tokens

Read an integer `k` and `k` step names, then lines of text until end of input. Build one `Function<String, String>` by composing the steps in order with `andThen`: `upper` (upper-case), `trim`, `reverse`, `repeat2` (the text twice), `exclaim` (append `!`). Store the named steps in a `Map<String, Function<String, String>>`. Apply the composed function to every line and print the result.

Example: `2` then `trim upper`, then lines `hello` and `ok` →

```
HELLO
OK
```

INPUT 1

```
2
trim upper
  hello
ok
```

OUTPUT 1

```
HELLO
OK
```

INPUT 2

```
3
reverse exclaim repeat2
abc
```

OUTPUT 2

```
cba!cba!
```

REFERENCE SOLUTION

```
import java.io.*;
import java.util.*;
import java.util.function.*;

public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        String[] first = in.readLine().trim().split("\\s+");
        int k = Integer.parseInt(first[0]);
        String[] steps = in.readLine().trim().split("\\s+");
        Map<String, Function<String, String>> named = new HashMap<>();
        named.put("upper", String::toUpperCase);
        named.put("trim", String::trim);
        named.put("reverse", s -> new StringBuilder(s).reverse().toString());
        named.put("repeat2", s -> s + s);
        named.put("exclaim", s -> s + "!");
        Function<String, String> pipeline = Function.identity();
        for (int i = 0; i < k; i++) pipeline = pipeline.andThen(named.get(steps[i]))
        String line;
        while ((line = in.readLine()) != null) System.out.println(pipeline.apply(lir
    }
}
```

QUIZ

1. A method that returns `s → s.startsWith(prefix)` given a `prefix` parameter returns…

A. A string

B. A closure that captured `prefix`

C. A compile error

D. A static method

2. `f.andThen(g).apply(x)` computes…

A. `f(g(x))`

B. `g(f(x))`

C. `f(x) + g(x)`

D. `g(x)` only

3. Implementing the strategy pattern with a `Map<String, BinaryOperator<Long>>` …

A. Loses type safety

B. Replaces one-class-per-strategy with lambdas while keeping strategies swappable

C. Requires reflection

D. Is slower than classes

4. Why prefer `optional.orElseGet(this::compute)` to `orElse(compute())`?

A. Shorter

B. The supplier runs only when the optional is empty

C. It handles null

D. No difference

5. A lambda in a stream that increments a captured `int[] counter`…

A. Is idiomatic

B. Compiles but signals the stream is the wrong tool — stream lambdas should be stateless

C. Is a compile error

D. Runs in parallel safely

ANSWER KEY · FUNCTIONS AS VALUES — COMPOSITION, FACTORIES AND STRATEGIES

1. B The lambda remembers the argument it was built with — a closure factory.

2. B `andThen` runs the receiver first; `compose` is the reverse.

3. B The interface stays; the implementations become values. Use classes again when a strategy has state.

4. B `orElse` evaluates its argument eagerly, every time.

5. B Mutable captured state breaks parallel streams and readability; use `count()` or a reduction instead.

CHECKPOINT · 22 MIN

Checkpoint — Nested classes & lambdas

This checkpoint covers static nested versus inner classes, local and anonymous classes, lambda syntax and semantics, the four kinds of method reference, and functions as values.

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 inner class instance holds that a static nested one does not, and why that leaks.
- Why captured locals must be effectively final.
- What `this` means inside a lambda versus inside an anonymous class.
- Which kind of reference `String::length` is, and what `prefix::startsWith` captures.
- How to return a function that remembers an argument.

The programs are a higher-order text pipeline built from composed functions, and an iterator implemented as an inner class alongside a lambda-based strategy table.

EXERCISE 1

Event bus with lambdas

Write `class EventBus` with `subscribe(String topic, Consumer<String> handler)` and `publish(String topic, String payload)` (calls every handler of that topic in subscription order; unknown topics are ignored). Handlers are kept in a `Map<String, List<Consumer<String>>>` built with `computeIfAbsent`.

Read an integer `n` and `n` commands: `sub <topic> <tag>` registers a lambda that prints `<tag>:<payload>`; `pub <topic> <payload>` publishes. Also count all deliveries in a field and print `deliveries=<count>` at the end.

Example: `4` then `sub news a`, `sub news b`, `pub news hello`, `pub sports x` →

```
a:hello
b:hello
deliveries=2
```

INPUT 1

```
4
sub news a
sub news b
pub news hello
pub sports x
```

OUTPUT 1

```
a:hello
b:hello
deliveries=2
```

INPUT 2

```
3
pub t x
sub t z
pub t y
```

OUTPUT 2

```
z:y
deliveries=1
```

REFERENCE SOLUTION

```java
import java.util.*;
import java.util.function.*;

class EventBus {
    private final Map<String, List<Consumer<String>>> handlers = new HashMap<>();
    private int deliveries = 0;

    void subscribe(String topic, Consumer<String> handler) {
        handlers.computeIfAbsent(topic, k -> new ArrayList<>()).add(handler);
    }

    void publish(String topic, String payload) {
        for (Consumer<String> h : handlers.getOrDefault(topic, List.of())) {
            h.accept(payload);
            deliveries++;
        }
    }

    int deliveries() { return deliveries; }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        EventBus bus = new EventBus();
        for (int i = 0; i < n; i++) {
            String cmd = in.next(), topic = in.next(), arg = in.next();
            if (cmd.equals("sub")) {
                String tag = arg;
                bus.subscribe(topic, payload -> System.out.println(tag + ":" + payl
            } else {
                bus.publish(topic, arg);
            }
        }
        System.out.println("deliveries=" + bus.deliveries());
    }
}
```

EXERCISE 2

Memoised function factory

Write `static <T, R> Function<T, R> memoize(Function<T, R> f)` that returns a function which caches results in a `HashMap<T, R>` (using `computeIfAbsent`) and counts cache misses in a captured `int[]` exposed through a second static method — simplest: make `memoize` return a small class or

record holding both the function and a `misses()` supplier. Then wrap a slow `Function<Integer, Long>` computing `n * n` (and incrementing nothing else).

Read an integer `n` and `n` integers; apply the memoised function to each and print the results space-separated, then `misses=<distinct inputs>`.

Example: `5` then `3 4 3 3 5` →

```
9 16 9 9 25
misses=3
```

INPUT 1

```
5
3 4 3 3 5
```

OUTPUT 1

```
9 16 9 9 25
misses=3
```

INPUT 2

```
1
100000
```

OUTPUT 2

```
10000000000
misses=1
```

REFERENCE SOLUTION

```java
import java.util.*;
import java.util.function.*;

public class Main {
    record Memo<T, R>(Function<T, R> fn, IntSupplier misses) { }

    static <T, R> Memo<T, R> memoize(Function<T, R> f) {
        Map<T, R> cache = new HashMap<>();
        int[] misses = {0};
        Function<T, R> memo = x -> cache.computeIfAbsent(x, key -> {
            misses[0]++;
            return f.apply(key);
        });
        return new Memo<>(memo, () -> misses[0]);
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Memo<Integer, Long> square = memoize(x -> (long) x * x);
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < n; i++) {
            if (i > 0) sb.append(' ');
            sb.append(square.fn().apply(in.nextInt()));
        }
        System.out.println(sb);
        System.out.println("misses=" + square.misses().getAsInt());
    }
}
```

EXERCISE 3

A command dispatcher

Build a `Map<String, Consumer<String[]>>` of commands, each a lambda operating on a shared `List<String>` `items`: `add <x>` appends, `del <x>` removes the first occurrence (printing `missing <x>` if absent), `show` prints the list, `count` prints `count=<n>`. An unknown command prints `unknown <cmd>`. Read `n` command lines and dispatch each by looking it up — no `switch` in the loop.

Example: `5` then `add a`, `add b`, `del c`, `show`, `count` →

```
missing c
[a, b]
count=2
```

INPUT 1

```
5
add a
add b
del c
show
count
```

OUTPUT 1

```
missing c
[a, b]
count=2
```

INPUT 2

```
3
fly x
count
show
```

OUTPUT 2

```
unknown fly
count=0
[]
```

REFERENCE SOLUTION

```
import java.io.*;
import java.util.*;
import java.util.function.*;

public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        int n = Integer.parseInt(in.readLine().trim());
        List<String> items = new ArrayList<>();
        Map<String, Consumer<String[]>> commands = new HashMap<>();
        commands.put("add", p -> items.add(p[1]));
        commands.put("del", p -> { if (!items.remove(p[1])) System.out.println("miss
        commands.put("show", p -> System.out.println(items));
        commands.put("count", p -> System.out.println("count=" + items.size()));
        for (int i = 0; i < n; i++) {
            String[] parts = in.readLine().trim().split("\\s+");
            Consumer<String[]> cmd = commands.get(parts[0]);
            if (cmd == null) System.out.println("unknown " + parts[0]);
            else cmd.accept(parts);
        }
    }
}
```

CHECKPOINT QUESTIONS

1. A static nested class can access the outer class's…

A. Instance fields directly

B. Static members, including private ones

C. Nothing

D. Only public members

2. Storing inner-class listener objects in a long-lived list risks…

A. Nothing

B. Keeping their enclosing instances alive — a memory leak

C. Thread-safety issues only

D. Compile errors

3. A local class declared in a **static** method…

A. Has an enclosing instance

B. Has no enclosing instance

C. Cannot capture locals

D. Is a compile error

4. Which lambda is invalid for `BiFunction<Integer, Integer, Integer>`?

A. `(a, b) → a + b`

B. `(Integer a, Integer b) → a * b`

C. `a → a + 1`

D. `(a, b) → { return a - b; }`

5. Lambda parameters…

A. May shadow enclosing locals

B. May not have the same name as an enclosing local variable

C. Must be typed

D. Must be final

6. `System.out::println` captures…

A. Nothing

B. The `PrintStream` object held by `System.out` at that moment

C. The `System` class

D. The string to print

7. `Comparator<String> c = String::compareTo;` — the reference kind is…

A. Static

B. Bound

C. Unbound: `(a, b) → a.compareTo(b)`

D. Constructor

8. Which is the best refactoring for a 15-line lambda in a stream?

A. Leave it

B. Extract it to a named method and use `this::method`

C. Convert to an anonymous class

D. Split into three lambdas inline

9. `Function<Integer, Function<Integer, Integer>> add = x → y → x + y;` — `add.apply(2).apply(3)` is…

A. A compile error

B. 5

C. A function

D. 23

10. `map.computeIfAbsent(key, k → new ArrayList<>()).add(v)` replaces…

A. A `merge`

B. The get-null-check-put-then-add sequence for a multimap

C. A `remove`

D. A sort

11. `Predicate.not(String::isBlank)` is equivalent to…

A. `String::isBlank`

B. `s → !s.isBlank()`

C. `s → s.isBlank()`

D. A compile error

12. Checked exceptions inside a `Runnable` lambda…

A. Propagate automatically

B. Must be caught in the body or wrapped, since `run()` declares none

C. Are converted to `RuntimeException`

D. Are ignored

ANSWER KEY · CHECKPOINT — NESTED CLASSES & LAMBDAS

1. B Nesting grants private access in both directions, but without an instance there are no instance fields to reach.

2. B Each inner instance references `Outer.this`; the outer cannot be collected while the listener is reachable.

3. B Inner-ness depends on the enclosing context; static context means no `Outer.this`.

4. C The SAM takes two arguments; a one-parameter lambda does not fit.

5. B Unlike anonymous-class methods, a lambda body shares the enclosing scope and cannot redeclare its names.

6. B It is a bound reference; if `System.setOut` changes the stream later, the reference still prints to the old one.

7. C Two target parameters, an instance method of one parameter: the first argument is the receiver.

8. B A named method documents intent and can be unit-tested.

9. B Currying: applying 2 returns a function that adds 2; applying that to 3 gives 5.

10. B The mapping function runs only when the key is absent, and the (existing or new) value is returned.

11. B `Predicate.not` (Java 11) negates a predicate given as a method reference, which cannot itself carry a `!`.

12. B The lambda may throw only what the functional interface's method declares.

CodeKairo · Java study plan · Module 11: Nested classes & lambdas. Solve the exercises online to track your progress and earn XP.