Module 5 of 20

Methods

Declaring and calling methods, static vs instance, pass-by-value, overload resolution, varargs, recursion, and scope.

Methods are how Java code is organised, reused and reasoned about. The mechanics look simple and hide three of the most-asked interview questions in the language: is Java pass-by-value or by-reference, which overload does the compiler pick, and what happens when recursion goes too deep.

This module answers all three precisely — with the stack-frame model that makes them obvious — and covers the rest of the method toolkit: static versus instance context, varargs and their pitfalls, recursion with memoization and backtracking, and the scope and lifetime rules behind shadowing and effectively-final variables.

By the end you can explain the argument-copy model with a diagram, predict overload resolution, and decide when recursion is the right tool.

CONTENTS

1	Defining and calling methods	13 min
2	Java is pass-by-value — always	14 min
3	Overloading and how the compiler picks	14 min
4	Varargs — methods that take any number of arguments	10 min
5	Recursion — methods that call themselves	16 min
6	Scope, lifetime and static context	12 min
✓	Checkpoint — Methods	22 min

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

LESSON 1 · 13 MIN

Defining and calling methods

A method is a named block of code with inputs and, optionally, an output. Every line of Java you run lives in one. This lesson covers the anatomy of a method declaration, the difference between `static` and instance methods (the single most common compile error for beginners), how calls work, and the conventions that make a method good rather than merely legal.

Anatomy

```
public static long factorial(int n) {
    long result = 1;
    for (int i = 2; i ⩽ n; i++) result *= i;
    return result;
}
```

Part	Here	Meaning
Access modifier	`public`	Who may call it (`public` , `protected` , package-private, `private`)
`static`	`static`	Belongs to the class, not to an object
Return type	`long`	The type of the value handed back; `void` for none
Name	`factorial`	lowerCamelCase, a verb or verb phrase
Parameters	`(int n)`	Typed, comma-separated; the *formal* parameters
Body	`{ … }`	Statements; a non-`void` method must `return` a value on every path

The **signature** is the name plus the parameter types — `factorial(int)` . The return type and parameter *names* are not part of it, which matters for overloading (lesson 3). Two methods in one class cannot have the same signature.

Optional extras: `throws IOException` after the parameters (Module 12), type parameters `<T>` before the return type (Module 13), `final` / `abstract` / `synchronized` modifiers (later modules), and annotations like `@Override` above.

Calling

```
long f = factorial(5);              // same class, static: just the name
long g = Main.factorial(5);         // qualified with the class
int len = "hello".length();         // instance method: on an object
double r = Math.sqrt(2.0);          // static method of another class
```

The *arguments* (`5`) are evaluated left to right and copied into the parameters. A call to a non-`void` method is an expression with a value; you may ignore the value (`list.add(x);` returns a `boolean` nobody reads), which is fine for methods with side effects and a bug for pure ones (`s.trim();`).

`static` versus instance

```
public class Counter {
    private int count;                          // instance field

    public void increment() { count++; }        // instance method: needs a Cou
    public int get() { return count; }

    public static Counter zero() { return new Counter(); }   // static: no Cou
}

Counter c = Counter.zero();     // static call on the class
c.increment();                  // instance call on an object
c.get();                        // 1
```

A **static** method belongs to the class. It has no `this` , so it cannot touch instance fields or call instance methods without an object. An **instance** method runs against a particular object (`c`), whose fields it reads through `this` . The error every beginner meets:

```
error: non-static method increment() cannot be referenced from a static contex
```

means you called an instance method from `main` (which is static) without an object. Either create an object (`new Counter().increment()`), or make the method static if it does not use instance state. Utility methods that only transform their arguments — `Math.max`, `Integer.parseInt`, your `isPalindrome(String)` — should be static. Behaviour that depends on an object's state should be an instance method on that object (Module 7).

Return

```
static int sign(int x) {
    if (x > 0) return 1;
    if (x < 0) return -1;
    return 0;                  // without this line: "missing return statement"
}

static void log(String msg) {
    if (msg == null) return;   // void methods may return early with a bare ret
    System.out.println(msg);
}
```

A non-`void` method must return on every path; the compiler checks. `return` ends the method immediately — code after it in the same block is unreachable. A method returns exactly one value; to return two, return an object (a record — Module 10 — or a small class), an array, or design it as two methods.

Parameters

Parameters are local variables initialised by the caller. Reassigning a parameter inside the method is legal (`n = n * 2`) but changes only the local copy — the caller's variable is untouched (next lesson). Declaring parameters `final` documents that you will not reassign them.

A method with no parameters still needs the parentheses: `int size()`. Calling it also needs them: `list.size()`, not `list.size` (that would be a field access).

Naming and size

Verbs for actions (`compute`, `send`, `parse`), `is` / `has` / `can` for booleans, `get` / `set` for accessors, `to` for conversions (`toString`, `toList`).

One job per method. If you need "and" to describe it, split it.

Short: a method that fits on a screen is one a reviewer can hold in their head. Extract helpers freely — a call costs nothing after the JIT inlines it.

Parameters: three or fewer is comfortable; more suggests a parameter object.

Methods that call methods

```
static boolean isValid(String input) {
    return isNonEmpty(input) && isNumeric(input) && inRange(Integer.parseInt(
}
```

Composition is the point. Small, well-named methods make the top-level one read like a sentence, and each helper can be tested alone. In interviews, writing `isValid` first and filling in helpers after is a strong way to structure a solution.

The call stack

Each call pushes a **frame** holding the method's parameters and locals; `return` pops it. Frames live on the thread's stack, which is why locals vanish when a method ends and why deep recursion overflows it (lesson 5). A stack trace (`at Main.factorial(Main.java:12)`) is a printout of these frames, innermost first — read the top lines to find where an exception was thrown, then down to see who called it.

Interview angle

"What is a method signature?" Name plus parameter types — not the return type.

"Can a static method access an instance variable?" Not without an object reference; there is no `this`.

"Why is `main` static?" The JVM calls it before any object of the class exists.

"Can you return multiple values?" Not directly — return a record/object/array.

Key takeaways

A method is `modifiers returnType name(params) { body }`; its signature is name + parameter types.

`static` = belongs to the class, no `this`; instance = runs on an object. "Non-static from static context" means you forgot the object.

Non-void methods return on every path; `return` ends the method.

Small, verb-named, single-purpose methods composed together.

EXERCISE 1

A prime helper

Write `static boolean isPrime(long n)` (trial division up to the square root; numbers below 2 are not prime). Read an integer `k`, then `k` integers; for each print `<n> prime` or `<n> composite`, then a last line `primes=<count>`.

Input: `k`, then `k` integers (each fits in a `long`). **Output:** `k + 1` lines.

INPUT 1

```
5
2 9 17 1 25
```

OUTPUT 1

```
2 prime
9 composite
17 prime
1 composite
25 composite
primes=2
```

INPUT 2

```
3
1000000007 4 0
```

OUTPUT 2

```
1000000007 prime
4 composite
0 composite
primes=1
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    static boolean isPrime(long n) {
        if (n < 2) return false;
        if (n % 2 == 0) return n == 2;
        for (long d = 3; d * d <= n; d += 2) {
            if (n % d == 0) return false;
        }
        return true;
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int k = in.nextInt();
        int primes = 0;
        for (int i = 0; i < k; i++) {
            long n = in.nextLong();
            boolean p = isPrime(n);
            if (p) primes++;
            System.out.println(n + (p ? " prime" : " composite"));
        }
        System.out.println("primes=" + primes);
    }
}
```

EXERCISE 2

gcd, lcm and a helper for each

Write `static long gcd(long a, long b)` (Euclid), `static long lcm(long a, long b)` (`a / gcd * b` — divide first to avoid overflow) and `static boolean coprime(long a, long b)`. Read an integer `n` and `n` pairs; print `gcd(a,b)=<g> lcm(a,b)=<l> coprime=<true|false>` for each. Every method has one job and a name that says it.

Example: `2` then `12 18`, `7 20` →

```
gcd(12,18)=6 lcm(12,18)=36 coprime=false
gcd(7,20)=1 lcm(7,20)=140 coprime=true
```

INPUT 1

```
2
12 18
7 20
```

OUTPUT 1

```
gcd(12,18)=6 lcm(12,18)=36 coprime=false
gcd(7,20)=1 lcm(7,20)=140 coprime=true
```

INPUT 2

```
1
100 10
```

OUTPUT 2

```
gcd(100,10)=10 lcm(100,10)=100
coprime=false
```

REFERENCE SOLUTION

```
import java.util.Scanner;

public class Main {
    static long gcd(long a, long b) {
        while (b ≠ 0) {
            long t = a % b;
            a = b;
            b = t;
        }
        return a;
    }
    static long lcm(long a, long b) {
        return a / gcd(a, b) * b;
    }
    static boolean coprime(long a, long b) {
        return gcd(a, b) == 1;
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            long a = in.nextLong(), b = in.nextLong();
            System.out.println("gcd(" + a + "," + b + ")=" + gcd(a, b) + " lcm(" + a
        }
    }
}
```

QUIZ

1. What makes up a method's *signature*?

A. Name, parameter types and return type

B. Name and parameter types

C. Name only

D. Return type and parameter names

2. "non-static method f() cannot be referenced from a static context" means…

A. `f` must be renamed

B. You called an instance method from static code (e.g. `main`) without an object

C. `f` is private

D. The class has no constructor

3. A non-void method with an `if` that returns in one branch and nothing after — result?

A. Returns 0 on the other path

B. Compile error: missing return statement

C. Returns null

D. Runtime exception

4. `s.trim();` as a statement on its own is…

A. A compile error

B. Legal but useless — the trimmed string is discarded

C. Trims `s` in place

D. Only legal for `StringBuilder`

5. Which method should be `static`?

A. One that reads the object's fields

B. One that only transforms its arguments, like `isPalindrome(String)`

C. Every method in `Main`

D. One that calls other instance methods

ANSWER KEY · DEFINING AND CALLING METHODS

1. B Two methods in a class may not share a name and parameter list; the return type and parameter names do not distinguish them.

2. B A static context has no `this`. Create an object and call `obj.f()`, or make `f` static if it needs no instance state.

3. B Every path through a non-void method must return a value; the compiler checks this statically.

4. B Ignoring a return value is allowed. For a pure method like `trim` it is a bug; for `list.add` it is normal.

5. B Utility methods that depend only on their inputs belong on the class; behaviour that depends on an object's state belongs on the object.

LESSON 2 · 14 MIN

Java is pass-by-value — always

"Is Java pass-by-value or pass-by-reference?" is asked in almost every Java interview, and the confident wrong answer — "primitives by value, objects by reference" — is what separates candidates who have memorised from those who understand. Java is **strictly pass-by-value**. What is passed for an object is the value of the *reference*. This lesson makes that precise and shows the three situations where it matters.

The rule

When you call a method, each argument expression is evaluated and its **value is copied** into the parameter. The parameter is a new local variable holding that copy.

> For a primitive, the value is the number/char/boolean itself.
>
> For a reference type, the value is the **reference** — the address of the object. The object is *not* copied; the *arrow pointing at it* is.

So the method gets its own arrow to the same object. Through that arrow it can **change the object's contents**. It cannot change **which object the caller's variable points to**.

Case 1: primitives — the callee cannot affect the caller

```
static void increment(int x) {
    x++;                    // changes the local copy
}
int n = 5;
increment(n);
System.out.println(n);      // 5
```

The method received a copy of 5. Nothing it does to `x` reaches `n`. To get a new value out, `return` it: `n = incremented(n);`.

Case 2: objects — mutation is visible, reassignment is not

```
static void fill(int[] arr) {
    arr[0] = 99;              // mutates the shared object — visible to the cal
    arr = new int[] {7};      // rebinds the LOCAL copy of the reference — invi
}
int[] data = {1, 2, 3};
fill(data);
System.out.println(data[0]); // 99
System.out.println(data.length); // 3 — still the original array
```

Line 1 follows the arrow and changes the array both variables point to. Line 2 points the *parameter* at a new array; `data` still points at the old one. This single example contains the whole story. If you can explain why `99` and why `3`, you understand Java's parameter passing.

The same applies to any object: a `StringBuilder` appended to inside a method shows the change; a `StringBuilder` parameter *reassigned* to a new one does not.

Case 3: strings and other immutables — nothing the callee does shows

```
static void shout(String s) {
    s = s.toUpperCase();      // new String, assigned to the local; caller unaj
}
String name = "ada";
shout(name);
System.out.println(name);     // ada
```

Strings are immutable, so there is no "mutate the object" option; the only thing the method can do is rebind its local, which is invisible. This is why "strings are passed by value" *feels* different from arrays — but the mechanism is identical; the difference is that `String` has no mutating methods.

The swap that cannot work

```
static void swap(int a, int b) { int t = a; a = b; b = t; }   // swaps two loc
static void swap(Integer a, Integer b) { … }                  // same: Integer
```

You cannot write a `swap` method for two variables in Java. You *can* swap two **elements of an array** (`swap(arr, i, j)`) or two **fields of an object**, because those are mutations through a shared reference. Interviewers ask this precisely because C++ programmers expect `&` references to exist.

`this` and fields are the escape hatch

A method that must "change the caller's variable" is usually a sign the value belongs in an object:

```
class Wallet { int balance; }
static void deposit(Wallet w, int amount) { w.balance += amount; }   // visibi
```

Or return the new value. Or, for several outputs, return a small object/record. Never a one-element array as an "out parameter" — it works, and reviewers hate it.

Why Java chose this

C++ offers both value and reference passing and it is a permanent source of bugs (aliasing, dangling references). Java's designers took C's model — everything by value — and, because objects are always accessed through references, mutation-through-reference gives most of the benefit of reference passing with none of the surprises. The one thing you lose is the ability to rebind a caller's variable, and that is a feature.

A mental model that never fails

Draw two columns: *variables* (boxes holding either a number or an arrow) and *objects* (blobs on the heap). Calling a method copies the box's contents into a new box. If the contents is an arrow, both boxes point at the same blob. `arr[0] = 99` scribbles on the blob. `arr = new int[]{7}` redraws the arrow in the *new* box. The caller's box is untouched in both cases; the blob is shared in the first.

Related gotchas

Returning a reference to internal state lets callers mutate your object: `getItems()` returning the actual `List` field. Return a copy or an unmodifiable view (Module 14).

Passing a mutable object to a method you do not control means it might change it. Defensive copies at boundaries (Module 7).

Autoboxed parameters are copies of references to immutable wrappers — nothing the callee does shows.

Interview angle

"Pass by value or by reference?" By value, always; for objects the value is the reference.

"Can a method change the object an argument refers to?" Its contents, yes; the caller's variable, no.

"Can you write `swap(int, int)`?" No. Swap array elements or object fields instead.

"`void f(String s) { s = "x"; }` — does the caller see it?" No; the local reference was rebound.

Key takeaways

Arguments are copied; for objects, the *reference* is copied, so both sides share one object.

Mutating through the reference is visible to the caller; reassigning the parameter is not.

Immutable arguments (strings, wrappers) can never be changed by a callee.

No `swap(a, b)` of variables — return values or mutate a shared container.

EXERCISE 1

What the caller sees

Implement three methods and report what the caller observes:

`static void bump(int x)` — does `x += 100`.

`static void fillFirst(int[] a)` — sets `a[0] = 100`, then does `a = new int[] {7}`.

`static void shout(StringBuilder sb)` — appends `!`, then does `sb = new StringBuilder("gone")`.

Read an integer `x`, an integer `n` followed by `n` integers for the array, and a word. Call the three methods, then print `x=<x>`, `array=<elements space-separated>` and `word=<contents of the`

`builder>`.

Example: `5`, `3 1 2 3`, `hi` → `x=5`, `array=100 2 3`, `word=hi!`

INPUT 1

```
5
3 1 2 3
hi
```

OUTPUT 1

```
x=5
array=100 2 3
word=hi!
```

INPUT 2

```
-1
1 9
java
```

OUTPUT 2

```
x=-1
array=100
word=java!
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    static void bump(int x) {
        x += 100;
    }

    static void fillFirst(int[] a) {
        a[0] = 100;
        a = new int[] {7};
    }

    static void shout(StringBuilder sb) {
        sb.append('!');
        sb = new StringBuilder("gone");
    }

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

        bump(x);
        fillFirst(a);
        shout(sb);

        StringBuilder arr = new StringBuilder();
        for (int i = 0; i < a.length; i++) {
            if (i > 0) arr.append(' ');
            arr.append(a[i]);
        }
        System.out.println("x=" + x);
        System.out.println("array=" + arr);
        System.out.println("word=" + sb);
    }
}
```

EXERCISE 2

Mutate or rebind?

Write three methods and call each from `main`: `static void fill(int[] a, int v)` sets every element to `v`; `static void replace(int[] a, int v)` assigns `a = new int[]{v}` (a rebind);

`static void append(StringBuilder sb, String s)` appends; and `static void shout(String s)` assigns `s = s.toUpperCase()`. Read `v` and a word `w`; start with `int[] a = {1, 2, 3}` and `StringBuilder sb = new StringBuilder(w)`. After each call print what the **caller** sees.

Example: `9 hi` →

```
after fill: [9, 9, 9]
after replace: [9, 9, 9]
after append: hi!
after shout: hi
```

INPUT 1

```
9 hi
```

OUTPUT 1

```
after fill: [9, 9, 9]
after replace: [9, 9, 9]
after append: hi!
after shout: hi
```

INPUT 2

```
0 Java
```

OUTPUT 2

```
after fill: [0, 0, 0]
after replace: [0, 0, 0]
after append: Java!
after shout: Java
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    static void fill(int[] a, int v) { Arrays.fill(a, v); }
    static void replace(int[] a, int v) { a = new int[]{v}; }
    static void append(StringBuilder sb, String s) { sb.append(s); }
    static void shout(String s) { s = s.toUpperCase(); }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int v = in.nextInt();
        String w = in.next();
        int[] a = {1, 2, 3};
        StringBuilder sb = new StringBuilder(w);
        fill(a, v);
        System.out.println("after fill: " + Arrays.toString(a));
        replace(a, v);
        System.out.println("after replace: " + Arrays.toString(a));
        append(sb, "!");
        System.out.println("after append: " + sb);
        shout(w);
        System.out.println("after shout: " + w);
    }
}
```

QUIZ

1. Java passes arguments…

A. Primitives by value, objects by reference

B. Everything by value — for objects, the value is the reference

C. Everything by reference

D. By value only for `final` parameters

2. `static void f(int[] a) { a = new int[]{1}; } int[] x = {5}; f(x);` — `x[0]` afterwards is…

A. 1

B. 5

C. 0

D. Compile error

3. `static void g(StringBuilder b) { b.append("!"); }` — visible to the caller?

A. No

B. Yes — the method mutated the object both references share

C. Only if `b` is returned

D. Only for `String`

4. Why can't you write `swap(int a, int b)` that swaps the caller's variables?

A. Java forbids two `int` parameters

B. The method receives copies; swapping them swaps the copies

C. It works if the parameters are `final`

D. It works with `Integer`

5. `static void h(String s) { s = s + "x"; }` — after `h(name)`, `name` is…

A. Changed

B. Unchanged — the local reference was rebound to a new string

C. null

D. Compile error

ANSWER KEY · JAVA IS PASS-BY-VALUE — ALWAYS

1. B The reference is copied into the parameter. The callee can mutate the shared object but cannot rebind the caller's variable.

2. B Reassigning the parameter changes only the local copy of the reference. `x` still points at the original array.

3. B Mutation through the copied reference reaches the one shared object. This is the difference from reassignment.

4. B Neither the primitives nor the (immutable) `Integer` references let the callee reach the caller's variables. Swap array elements or fields instead.

5. B Strings are immutable, so the only thing the callee can do is point its own variable somewhere else.

LESSON 3 · 14 MIN

Overloading and how the compiler picks

Two methods with the same name and different parameter lists are **overloads**. `System.out.println` has ten of them; `Math.max` has four; you will write your own to offer convenient variants of one operation. The compiler chooses among them at **compile time** by a three-phase rule that is precise, occasionally surprising, and a favourite interview topic.

What can be overloaded

```
static int    max(int a, int b)
static long   max(long a, long b)
static double max(double a, double b)
static int    max(int a, int b, int c)
```

Overloads differ in the **number or types of parameters** (the signature). They may not differ *only* in return type — `int f()` and `long f()` in one class is an error, because a call `f()` could not choose. Parameter names are irrelevant. Overloads may have different return types, modifiers and exceptions, as long as the parameter lists differ.

Overloading is resolved **statically**: the compiler looks at the declared types of the argument expressions, not what they hold at run time. That is the crucial difference from **overriding** (Module 8), which is resolved at run time by the receiver's actual class.

The three phases

Given a call, the compiler finds the applicable overloads in three phases and stops at the first phase that finds any:

1. **Exact match or primitive widening** / **reference subtyping** — no boxing, no varargs.
2. **With boxing**/**unboxing** allowed.
3. **With varargs** allowed.

Within a phase, the **most specific** method wins (an `int` parameter is more specific than `long`, `String` more specific than `Object`). If two are equally specific, the call is *ambiguous* and fails to compile.

```
static void f(long x)      { System.out.println("long"); }
static void f(Integer x)   { System.out.println("Integer"); }
static void f(Object x)    { System.out.println("Object"); }
static void f(int... x)    { System.out.println("varargs"); }

f(5);       // "long"    — phase 1: int widens to long; boxing (Integer) is pl
```

Widening beats boxing. Remove `f(long)` and the answer becomes `"Integer"` (phase 2, boxing, and `Integer` is more specific than `Object`). Remove that too and it is `"Object"` (still phase 2: box to `Integer`, then subtype to `Object`). Only with all three gone does varargs win.

Widening preferences

Among primitives, the compiler picks the *smallest* widening: with `f(long)`, `f(float)` and `f(double)` available, `f(5)` calls `f(long)`. `char` widens to `int` before anything else: `f('a')` with `f(int)` and `f(long)` calls `f(int)`.

`byte` / `short` / `char` never *narrow* to match: `f((short) 1)` cannot call `f(byte)`.

Reference types: most specific wins

```
static void g(Object o)  { … "Object" }
static void g(String s)  { … "String" }

g("hi");            // "String" — more specific
g(null);            // "String" — null converts to any reference; String is mo
Object o = "hi";
g(o);               // "Object" — the DECLARED type of o is Object; resolution
```

The last line is the one to remember: the argument's *static type* decides, not the object inside. And `g(null)` with `g(String)` and `g(Integer)` is **ambiguous** — neither is more specific than the other — and fails to compile; cast the null: `g((String) null)`.

Boxing pitfalls

```
static void h(long x)   { … }
static void h(Integer x){ … }
Integer boxed = 5;
h(boxed);           // "Integer" — exact match in phase 1 (no conversion neede
h(5);               // "long"    — widening (phase 1) beats boxing (phase 2)
```

A related classic: `List<Integer> list; list.remove(1)` calls `remove(int index)` — removing the element at index 1 — not `remove(Object o)`, because the exact primitive match wins. To remove the *value* 1: `list.remove(Integer.valueOf(1))`.

Overloading and `null`

`System.out.println(null)` is ambiguous (`char[]` and `String` both apply). `println((Object) null)` compiles. Passing bare `null` to an overloaded method is a smell for this reason.

Overloading versus defaults

Java has no default parameter values. Overloads *are* the way to offer defaults: the short version calls the full one.

```
void connect(String host)            { connect(host, 80, 30_000); }
void connect(String host, int port)  { connect(host, port, 30_000); }
void connect(String host, int port, int timeoutMs) { … the real work … }
```

One implementation, several entry points. This "telescoping" pattern is standard; beyond three or four parameters, a builder (Module 7) reads better.

Overloads and constructors

Constructors overload the same way, and `this(...)` chains them (Module 7). The same resolution rules apply to `new Foo(5)`.

Guidelines

Overloads should do the **same thing** for different inputs. `add(int)` and `add(String)` doing unrelated work is a naming failure.

Avoid overloads whose parameters differ only by boxing or widening (`f(int)` vs `f(Integer)`, `f(long)` vs `f(int)`) — callers cannot predict which they get. `Math.max`'s four are fine because they behave identically.

Avoid overloads that both accept `null`-able reference types at the same position.

Never rely on varargs to disambiguate.

`@Override`-style safety does not exist for overloads: a typo in a parameter type silently creates a new overload instead of an error. Tests catch it; the compiler does not.

Interview angle

"Can methods be overloaded by return type?" No.

"`f(5)` with `f(long)` and `f(Integer)` — which runs?" `f(long)`: widening before boxing.

"Is overloading resolved at compile time or run time?" Compile time, from the arguments' declared types.

"`list.remove(1)` on a `List<Integer>`?" Removes index 1 — exact match on `int`.

"What does `g(null)` do with `g(String)` and `g(Object)`?" Calls `g(String)`, the more specific.

Key takeaways

Overloads differ in parameter lists; return type alone does not count.

Resolution is static (declared types) in three phases: widening/subtyping → boxing → varargs; most specific wins; ties are errors.

Widening beats boxing beats varargs; `String` beats `Object`; `null` picks the most specific reference overload or is ambiguous.

Use overloads for defaults and same-behaviour variants; avoid boxing-only differences.

EXERCISE 1

Area overloads

Write three overloads of `area`: `area(int side)` for a square, `area(int w, int h)` for a rectangle, and `area(double radius)` for a circle (`Math.PI * r * r`). Read an integer `n`, then `n` lines of the

form `square 3`, `rect 3 4` or `circle 2.5`, call the matching overload, and print each area with two decimals.

Note that `area(3)` must call the `int` version and `area(2.5)` the `double` one — parse the arguments with the right types.

Example: `square 3` → `9.00`; `circle 1` → `3.14`.

INPUT 1

```
3
square 3
rect 3 4
circle 1
```

OUTPUT 1

```
9.00
12.00
3.14
```

INPUT 2

```
2
circle 2.5
square 100000
```

OUTPUT 2

```
19.63
10000000000.00
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    static double area(int side) { return (double) side * side; }
    static double area(int w, int h) { return (double) w * h; }
    static double area(double radius) { return Math.PI * radius * radius; }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            String shape = in.next();
            double a;
            switch (shape) {
                case "square" -> a = area(in.nextInt());
                case "rect" -> a = area(in.nextInt(), in.nextInt());
                default -> a = area(in.nextDouble());
            }
            System.out.printf("%.2f%n", a);
        }
    }
}
```

EXERCISE 2

Which overload?

Define four overloads named `describe`: `describe(int)`, `describe(long)`, `describe(double)` and `describe(Object)`, each returning its own name (`"int"`, `"long"`, `"double"`, `"Object"`). Read `n` tokens of the form `<kind>:<value>` where kind is `int`, `long`, `double`, `char`, `short`, `Integer` or `String`; declare a variable of exactly that type, call `describe` with it, and print `<token> → <chosen overload>`. Widening beats boxing; a `char` or `short` widens to `int`; an `Integer` is an `Object`.

Example: `4` then `int:5 char:a Integer:5 short:7` →

```
int:5 → int
char:a → int
Integer:5 → Object
short:7 → int
```

INPUT 1

```
4
int:5 char:a Integer:5 short:7
```

OUTPUT 1

```
int:5 → int
char:a → int
Integer:5 → Object
short:7 → int
```

INPUT 2

```
3
long:9 double:2.5 String:hey
```

OUTPUT 2

```
long:9 → long
double:2.5 → double
String:hey → Object
```

REFERENCE SOLUTION

```java
import java.util.Scanner;

public class Main {
    static String describe(int x) { return "int"; }
    static String describe(long x) { return "long"; }
    static String describe(double x) { return "double"; }
    static String describe(Object x) { return "Object"; }

    static String pick(String kind, String value) {
        switch (kind) {
            case "int": { int v = Integer.parseInt(value); return describe(v); }
            case "long": { long v = Long.parseLong(value); return describe(v); }
            case "double": { double v = Double.parseDouble(value); return describe(v
            case "char": { char v = value.charAt(0); return describe(v); }
            case "short": { short v = Short.parseShort(value); return describe(v); }
            case "Integer": { Integer v = Integer.valueOf(value); return describe(v)
            case "String": { String v = value; return describe(v); }
            default: return "?";
        }
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            String token = in.next();
            String[] parts = token.split(":");
            System.out.println(token + " → " + pick(parts[0], parts[1]));
        }
    }
}
```

QUIZ

1. Given `f(long)`, `f(Integer)` and `f(int...)`, the call `f(5)` invokes…

A. `f(Integer)`

B. `f(long)` — widening beats boxing

C. `f(int...)`

D. It is ambiguous

2. Can two methods differ only in return type?

A. Yes

B. No — the return type is not part of the signature

C. Only if one is static

D. Only in different packages

3. `Object o = "hi"; g(o);` with `g(Object)` and `g(String)` calls…

A. `g(String)` because the object is a String

B. `g(Object)` — overloads are chosen by the argument's declared type at compile time

C. Both

D. Compile error

4. `List<Integer> list; list.remove(1);` removes…

A. The element equal to 1

B. The element at index 1 — `remove(int)` is an exact match

C. Both

D. Nothing

5. `g(null)` with `g(String)` and `g(Integer)` is…

A. `g(String)`

B. `g(Integer)`

C. Ambiguous — a compile error

D. A `NullPointerException`

ANSWER KEY · OVERLOADING AND HOW THE COMPILER PICKS

1. B Resolution has three phases: widening/subtyping first, boxing second, varargs last. `int` → `long` is a phase-1 match.

2. B A call site could not choose between them. The compiler reports a duplicate method.

3. B Overload resolution is static. The run-time class is irrelevant; that is what *overriding* uses.

4. B The primitive `int` argument matches `remove(int index)` in phase 1; `remove(Object)` would need boxing. Use `remove(Integer.valueOf(1))` for the value.

5. C `null` converts to both and neither type is more specific than the other. Cast: `g((String) null)`.

LESSON 4 · 10 MIN

Varargs — methods that take any number of arguments

`String.format("%d %s", 3, "x")` takes two extra arguments; `Arrays.asList(1, 2, 3)` takes three; `Math.max` takes exactly two. The first two use **varargs** — a parameter declared with `...` that accepts zero or more values and arrives as an array. This lesson covers the syntax, the rules, the interaction with overloading, and the two pitfalls (`null` and generics) that produce warnings and bugs.

Syntax

```
static int sum(int... numbers) {          // numbers is an int[] inside
    int total = 0;
    for (int n : numbers) total += n;
    return total;
}

sum();               // 0     — an empty array
sum(1);              // 1
sum(1, 2, 3);        // 6
sum(new int[] {4, 5});   // 9  — an explicit array is also accepted
```

Inside the method a varargs parameter **is** an array: `numbers.length`, `numbers[i]`, for-each. The compiler packs the call's trailing arguments into a new array at each call site — `sum(1, 2, 3)` compiles to `sum(new int[] {1, 2, 3})`.

Rules:

- Only the **last** parameter may be varargs: `static void log(String level, Object... details)`.
- At most one per method.
- `T...` is equivalent to `T[]` in the signature — `sum(int...)` and `sum(int[])` are the *same* method and cannot both be declared.
- The main method may be `public static void main(String... args)`.

Common uses

```
System.out.printf("%s is %d%n", name, age);          // Object... after the fo
Arrays.asList("a", "b", "c");                          // T...
List.of(1, 2, 3);                                      // overloads up to 10 a
EnumSet.of(Day.MON, Day.TUE);
Collections.addAll(list, "x", "y", "z");
Path.of("home", "user", "file.txt");                   // String first, String
static <T> T firstNonNull(T... values)
```

They shine for "one or more of the same thing" APIs where forcing the caller to build an array would be noise.

Varargs and overload resolution

Varargs applicability is the **third and last phase** of overload resolution (previous lesson): a fixed-arity overload that fits, with or without boxing, always wins.

```
static void f(int a, int b)   { System.out.println("two"); }
static void f(int... a)       { System.out.println("varargs"); }

f(1, 2);      // "two"
f(1, 2, 3);   // "varargs"
f();          // "varargs"
```

Mixing a fixed overload and a varargs overload of the same type is a common and sensible pattern (`List.of` does it for performance: fixed-arity overloads avoid allocating an array for the common small cases).

Pitfall 1: passing `null`

```
static void print(String... items) { System.out.println(items.length); }

print();              // 0
print((String) null); // 1 — an array holding one null
print(null);          // NullPointerException: null is passed AS the array; it
```

A bare `null` matches the array type directly, so the parameter *is* null. Cast it (`(String) null`) to mean "one null element", and inside varargs methods consider guarding `if (items == null)`.

Pitfall 2: a single array argument

```
static void show(Object... items) { System.out.println(items.length); }

show(new String[] {"a", "b"});      // 2 — String[] is an Object[], passed as
show((Object) new String[] {"a", "b"});   // 1 — one element which is an array
show(new int[] {1, 2});             // 1 — int[] is not an Object[], so it bec
```

An array whose type matches the varargs array type is passed *as* the array, not wrapped. This is the source of "why does my `Object...` method see 2 elements". Cast to `Object` to force wrapping.

Pitfall 3: generics — heap pollution

```
static <T> List<T> listOf(T... items) { … }     // warning: possible heap poll
```

Varargs create an array, and arrays of generic types are unsafe (Module 13). The compiler warns at the declaration and at call sites. If your method only reads the array and never stores it or exposes it, annotate it `@SafeVarargs` (allowed on `static`, `final` and `private` methods and constructors) to promise that and silence the warning. `Arrays.asList`, `List.of` and `EnumSet.of` carry this annotation.

Cost

Every varargs call allocates an array — trivial in ordinary code, measurable in a hot loop called millions of times. Logging frameworks provide `log(String, Object)` and `log(String, Object, Object)` overloads before the `Object...` version for exactly this reason. Do the same if a varargs method is on a hot path; otherwise do not think about it.

Design notes

Put required parameters first, then the varargs: `static int max(int first, int... rest)` guarantees at least one value and avoids a runtime check for an empty array.

Varargs of a *single* type are clearest. `Object...` accepts anything and defers type errors to run time (`String.format` with mismatched specifiers).

Do not overload a varargs method with another varargs method of a different but convertible type (`f(int...)` and `f(long...)`); `f()` becomes ambiguous.

Interview angle

"What is a varargs parameter at run time?" An array; the compiler builds it at the call site.

"Where must it appear?" Last, and only once.

"`print(null)` to `print(String...)`?" The array itself is null — NPE on `.length`.

"Which wins, `f(int, int)` or `f(int...)` for `f(1, 2)`?" The fixed-arity one; varargs is the last resort.

Key takeaways

`T... name` is a `T[]` parameter that accepts zero or more values; last position only.

Fixed-arity overloads always beat varargs; varargs are phase three.

A bare `null` or a matching array is passed *as* the array — cast to `Object` / `(T) null` to wrap.

Generic varargs warn; `@SafeVarargs` when you only read the array.

Each call allocates an array; add small fixed-arity overloads on hot paths.

EXERCISE 1

Stats for any count

Implement `static int max(int first, int... rest)` and `static double mean(int first, int... rest)` — the leading parameter guarantees at least one value. Read an integer `n`, then `n` lines each containing one or more integers. For each line, split it, call both methods (pass the remaining values as an array), and print `max=<m> mean=<mean with 2 decimals>`.

Example line: `3 9 4` → `max=9 mean=5.33`

INPUT 1

```
3
3 9 4
7
-5 -2 -9
```

OUTPUT 1

```
max=9 mean=5.33
max=7 mean=7.00
max=-2 mean=-5.33
```

INPUT 2

```
1
2147483647 2147483647
```

OUTPUT 2

```
max=2147483647 mean=2147483647.00
```

REFERENCE SOLUTION

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

public class Main {
    static int max(int first, int... rest) {
        int m = first;
        for (int v : rest) if (v > m) m = v;
        return m;
    }

    static double mean(int first, int... rest) {
        long sum = first;
        for (int v : rest) sum += v;
        return (double) sum / (rest.length + 1);
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        int n = Integer.parseInt(br.readLine().trim());
        for (int i = 0; i < n; i++) {
            String[] t = br.readLine().trim().split("\\s+");
            int first = Integer.parseInt(t[0]);
            int[] rest = new int[t.length - 1];
            for (int j = 1; j < t.length; j++) rest[j - 1] = Integer.parseInt(t[j]);
            System.out.printf("max=%d mean=%.2f%n", max(first, rest), mean(first, re
        }
    }
}
```

EXERCISE 2

Varargs toolkit

Write `static String join(String sep, String... parts)` and `static int max(int first, int... rest)` (the required first parameter guarantees at least one value). Read an integer `n` and `n` commands: `join <sep> <k> <k words>` prints the words joined with `sep`; `max <k> <k ints>` prints the largest. Call `join` with the words spread from an array and `max` with the first value split off — that is what varargs make natural.

Example: `2` then `join - 3 a b c`, `max 4 3 9 2 7` →

```
a-b-c
9
```

INPUT 1

```
2
join - 3 a b c
max 4 3 9 2 7
```

OUTPUT 1

```
a-b-c
9
```

INPUT 2

```
2
join , 1 solo
max 1 -4
```

OUTPUT 2

```
solo
-4
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    static String join(String sep, String... parts) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < parts.length; i++) {
            if (i > 0) sb.append(sep);
            sb.append(parts[i]);
        }
        return sb.toString();
    }
    static int max(int first, int... rest) {
        int m = first;
        for (int x : rest) m = Math.max(m, x);
        return m;
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            String cmd = in.next();
            if (cmd.equals("join")) {
                String sep = in.next();
                int k = in.nextInt();
                String[] words = new String[k];
                for (int j = 0; j < k; j++) words[j] = in.next();
                System.out.println(join(sep, words));
            } else {
                int k = in.nextInt();
                int first = in.nextInt();
                int[] rest = new int[k - 1];
                for (int j = 0; j < k - 1; j++) rest[j] = in.nextInt();
                System.out.println(max(first, rest));
            }
        }
    }
}
```

QUIZ

1. Inside `static int sum(int... xs)`, what is `xs`?

A. A `List<Integer>`

B. An `int[]` built by the compiler at the call site

C. A single `int`

D. An `Iterator`

2. Where may a varargs parameter appear?

A. Anywhere

B. Only as the last parameter, and only once

C. Only first

D. Only in static methods

3. `print(null)` where `print(String... s)` — what happens?

A. `s` has one null element

B. `s` itself is null; `s.length` throws `NullPointerException`

C. Compile error

D. `s` is empty

4. With `f(int, int)` and `f(int...)` both declared, `f(1, 2)` calls…

A. `f(int...)`

B. `f(int, int)` — fixed arity beats varargs

C. Ambiguous

D. Neither

5. What does `@SafeVarargs` assert?

A. The method never throws

B. The method does not store or expose its generic varargs array unsafely, so the heap-pollution warning can be suppressed

C. The varargs may be null

D. The method is thread-safe

ANSWER KEY · VARARGS — METHODS THAT TAKE ANY NUMBER OF ARGUMENTS

1. B `T...` is `T[]` — `sum(1, 2)` compiles to `sum(new int[]{1, 2})`, and `sum()` passes an empty array.

2. B The compiler needs an unambiguous split between fixed and variable arguments.

3. B A bare `null` matches the array type directly. Cast it — `(String) null` — to pass a one-element array.

4. B Varargs applicability is the third and last resolution phase; any fixed-arity match wins first.

5. B Generic varargs create an array of a generic type; the annotation is the author's promise that this is used safely.

LESSON 5 · 16 MIN

Recursion — methods that call themselves

A recursive method solves a problem by solving a smaller version of the same problem and combining. It is the natural way to express anything tree-shaped, anything defined inductively, and most "try every possibility" searches. It is also the place where the call stack stops being an abstraction: every recursive call is a frame, and the JVM's default stack holds only a few thousand of them. This lesson gives you the discipline — base case, progress, trust — and the tools to keep recursion correct and fast.

The shape

```
static long factorial(int n) {
    if (n ⩽ 1) return 1;                    // base case: answered directly
    return n * factorial(n - 1);            // recursive case: smaller problem,
}
```

Every correct recursion has:

A base case — an input answered without recursing. Without one, the calls never stop.

Progress — every recursive call moves toward the base case (`n - 1`, a shorter string, a smaller subtree).

Trust — the recursive call is assumed to return the right answer for the smaller input; you only write the combining step. This is the mental leap: do not trace the whole recursion in your head; check the base case, check one step, and let induction do the rest.

What happens on the stack

`factorial(4)` pushes a frame (`n = 4`), which calls `factorial(3)` (a second frame), … down to `factorial(1)`, which returns 1. Then each frame completes its multiplication and pops: 1 → 2 → 6 → 24. At the deepest point, five frames are live. The depth of recursion is the amount of stack used.

The default thread stack is around 512 KB–1 MB; a simple frame is tens of bytes, so roughly **10 000–20 000 nested calls** fit before:

```
Exception in thread "main" java.lang.StackOverflowError
```

This is an `Error`, not an `Exception`; catching it is not the fix. A recursion over a list of a million elements, or over `n` down to 0 with `n = 100 000`, will overflow. Recursion depth must be *logarithmic* (binary search, balanced trees: depth ~30) or *small* (a few thousand); linear-depth recursion over big inputs must become a loop.

Classic examples

Fibonacci, naive — exponential time, since `fib(n - 2)` is computed twice at every level:

```
static long fib(int n) {
    if (n < 2) return n;
    return fib(n - 1) + fib(n - 2);       // ~2^n calls; fib(45) takes seconds
}
```

Fibonacci with memoization — remember answers; each is computed once:

```
static long[] memo = new long[91];
static long fib(int n) {
    if (n < 2) return n;
    if (memo[n] != 0) return memo[n];
    return memo[n] = fib(n - 1) + fib(n - 2);    // O(n) calls
}
```

Memoization turns overlapping-subproblem recursions into linear work; it is the bridge to dynamic programming.

Binary search — logarithmic depth, safe:

```
static int search(int[] a, int target, int lo, int hi) {
    if (lo > hi) return -1;
    int mid = lo + (hi - lo) / 2;
    if (a[mid] == target) return mid;
    return a[mid] < target ? search(a, target, mid + 1, hi) : search(a, targe
}
```

Reverse a string — linear depth; fine for short strings, overflows for long ones:

```
static String reverse(String s) {
    if (s.length() ⩽ 1) return s;
    return reverse(s.substring(1)) + s.charAt(0);
}
```

Power set / **permutations** / **subsets** — the "choose or skip" pattern, where recursion is genuinely the clearest tool:

```
static void subsets(int[] a, int i, List<Integer> current) {
    if (i == a.length) { System.out.println(current); return; }
    subsets(a, i + 1, current);                        // skip a[i]
    current.add(a[i]);
    subsets(a, i + 1, current);                        // take a[i]
    current.remove(current.size() - 1);                // undo — backtracking
}
```

Helper methods and accumulators

Public API rarely wants the extra parameters recursion needs. Wrap:

```
public static int search(int[] a, int target) { return search(a, target, 0, a
private static int search(int[] a, int target, int lo, int hi) { … }
```

An **accumulator** parameter carries the result down instead of combining on the way up:

```
static long factorial(int n, long acc) {
    if (n ⩽ 1) return acc;
    return factorial(n - 1, acc * n);      // tail call: nothing to do after
}
```

In languages with *tail-call optimisation* this runs in constant stack. **The JVM does not do tail-call optimisation** — a tail-recursive method still uses a frame per call. Write the loop.

Recursion versus iteration

Every recursion can be rewritten as a loop with an explicit stack; every loop can be written recursively. Choose by clarity and depth:

Prefer recursion	Prefer iteration
Trees, nested structures, divide and conquer	Linear scans, counting, accumulating
Backtracking / generate-all-combinations	Anything with depth ~n for large n
The recursive definition *is* the algorithm (GCD, Towers of Hanoi)	Performance-critical inner loops

A recursive `sum(list)` over a million elements is a `StackOverflowError`; a recursive tree traversal over a million-node balanced tree is depth 20 and fine.

Debugging recursion

Print the arguments on entry, indented by depth, to see the call tree.

Check the base case handles the smallest inputs *and* empty/zero.

Make sure every recursive path makes progress — a call with the same arguments is an infinite loop.

If the answer is wrong for `n = 2` but right for `n = 1`, the combining step is wrong.

`-Xss` raises the stack size if you truly need a few more frames; it is a workaround, not a design.

Mutual recursion and `static` context

Two methods can call each other (`isEven(n)` calls `isOdd(n - 1)`); the same base-case rules apply to the pair. Recursive methods are usually `static` (they depend on their arguments, not on an object) — an instance method can recurse too, on `this` or on other objects (a tree node calling `left.size()`).

Interview angle

"What happens without a base case?" `StackOverflowError`.

"Why is naive Fibonacci slow?" Exponential re-computation; memoize or iterate.

"Does Java optimise tail recursion?" No.

"When do you choose recursion?" Trees, divide-and-conquer, backtracking; not linear scans of large inputs.

"How deep can you recurse?" Thousands to low tens of thousands with the default stack.

Key takeaways

Base case, progress, trust — check the base case and one step; do not trace everything.

Each call is a stack frame; depth must be small or logarithmic. Linear-depth recursion over big inputs overflows — use a loop.

Memoize overlapping subproblems; the JVM has no tail-call optimisation.

Wrap the recursive worker in a clean public method; use accumulators and backtracking (undo after the call) for searches.

EXERCISE 1

Three recursions

Implement three recursive methods and dispatch on commands. Read an integer `n`, then `n` lines:

`sum <k>` → the sum of the decimal digits of `k` (k ≥ 0), recursively: `digitSum(k) = k % 10 + digitSum(k / 10)`, base case `k < 10`.

`pow <b> <e>` → `b^e` computed by **exponentiation by squaring** (e ≥ 0; the answer fits in a `long`): base case `e == 0`, otherwise square the half power and multiply by `b` once more if `e` is odd.

`rev <word>` → the word reversed recursively (base case: length ≤ 1).

Print one result per line.

INPUT 1

```
4
sum 12345
pow 2 10
rev abc
pow 3 0
```

OUTPUT 1

```
15
1024
cba
1
```

INPUT 2

```
3
sum 0
pow 7 13
rev a
```

OUTPUT 2

```
0
96889010407
a
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    static int digitSum(long k) {
        if (k < 10) return (int) k;
        return (int) (k % 10) + digitSum(k / 10);
    }

    static long power(long b, int e) {
        if (e == 0) return 1;
        long half = power(b, e / 2);
        long result = half * half;
        return (e % 2 == 1) ? result * b : result;
    }

    static String reverse(String s) {
        if (s.length() <= 1) return s;
        return reverse(s.substring(1)) + s.charAt(0);
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            String cmd = in.next();
            switch (cmd) {
                case "sum" -> System.out.println(digitSum(in.nextLong()));
                case "pow" -> System.out.println(power(in.nextLong(), in.nextInt()));
                default -> System.out.println(reverse(in.next()));
            }
        }
    }
}
```

EXERCISE 2

Towers of Hanoi

Read `n` (`1 ≤ n ≤ 8`). Print the moves that transfer `n` discs from peg `A` to peg `C` using peg `B`, one per line as `move disc <k> from <X> to <Y>` (disc 1 is the smallest), in the order the classic recursion produces them, then `moves=<total>`. The recursion: move `n-1` discs to the spare peg, move disc `n`, move the `n-1` back on top.

Example: `2` →

```
move disc 1 from A to B
move disc 2 from A to C
move disc 1 from B to C
moves=3
```

INPUT 1

```
2
```

OUTPUT 1

```
move disc 1 from A to B
move disc 2 from A to C
move disc 1 from B to C
moves=3
```

INPUT 2

```
1
```

OUTPUT 2

```
move disc 1 from A to C
moves=1
```

REFERENCE SOLUTION

```java
import java.util.Scanner;

public class Main {
    static int moves = 0;

    static void hanoi(int n, char from, char to, char via) {
        if (n == 0) return;
        hanoi(n - 1, from, via, to);
        System.out.println("move disc " + n + " from " + from + " to " + to);
        moves++;
        hanoi(n - 1, via, to, from);
    }

    public static void main(String[] args) {
        int n = new Scanner(System.in).nextInt();
        hanoi(n, 'A', 'C', 'B');
        System.out.println("moves=" + moves);
    }
}
```

QUIZ

1. A recursive method without a reachable base case results in…

A. An infinite loop that never uses memory

B. `StackOverflowError`

C. A compile error

D. `OutOfMemoryError`

2. Why is naive recursive Fibonacci exponential?

A. Java recursion is slow

B. Each call spawns two calls and the same subproblems are recomputed many times

C. `long` arithmetic is slow

D. It is linear

3. Does the JVM optimise tail recursion into a loop?

A. Yes, always

B. No — a tail call still uses a frame

C. Only with `-O2`

D. Only for static methods

4. A recursive binary search over a million elements has depth about…

A. 1 000 000

B. 20

C. 1000

D. 500 000

5. In backtracking, after the recursive call returns you must…

A. Return immediately

B. Undo the change made before the call so the next branch starts from the same state

C. Clear the whole structure

D. Call it again

ANSWER KEY · RECURSION — METHODS THAT CALL THEMSELVES

1. B Each call pushes a stack frame; with no base case the thread's stack is exhausted after a few thousand to tens of thousands of frames.

2. B `fib(n-2)` is computed inside both `fib(n-1)` and directly. Memoization stores each answer once and makes it linear.

3. B Java has no tail-call elimination. Linear-depth recursion over large inputs must be rewritten as iteration.

4. B Halving each time gives $\log_2(10^6) \approx 20$ frames — recursion is safe when depth is logarithmic.

5. B Add-recurse-remove is the pattern: the shared `current` list must look the same for every sibling branch.

LESSON 6 · 12 MIN

Scope, lifetime and static context

Scope is where a name can be used; *lifetime* is how long the thing it names exists. They are different — a field is in scope only inside its class but lives as long as its object; a local is in scope for a block and lives for one execution of it. Getting them straight explains the compile errors around shadowing and static context, and, later, why lambdas can only capture effectively-final variables.

Levels of scope

```
public class Inventory {                          // class scope: fields and meth
    private static int instances = 0;             // static field: class lifetime
    private final List<String> items = new ArrayList<>();   // instance field

    public int countLongerThan(int limit) {       // parameter: method scope, cal
        int count = 0;                            // local: method scope from her
        for (String item : items) {               // loop variable: loop scope
            int len = item.length();              // block local: one iteration
            if (len > limit) count++;
        }
        // len and item are gone here; count is not
        return count;
    }
}
```

Rules:

A **local** is in scope from its declaration to the end of its enclosing block. Not before its declaration, not after the block.

A **parameter** is in scope for the whole method body.

A **field** is in scope throughout the class (and, if not private, in subclasses and other classes via a reference), regardless of where in the class it is declared — methods above the field can use it.

Blocks nest; an inner block sees everything in outer blocks and the class.

No shadowing among locals

```
int total = 0;
for (int i = 0; i < 3; i++) {
    int total = i;          // error: variable total is already defined in me
}
```

Java forbids a local (or parameter) from redeclaring a name visible from an enclosing local scope of the same method. This removes a whole family of "which `total` is this" bugs that C allows. Sibling blocks *may* reuse names — two consecutive `for (int i …)` loops are fine, because neither `i` is in scope in the other.

Fields can be shadowed

```
public class Point {
    private int x, y;
    public Point(int x, int y) {
        this.x = x;               // this.x: the field; x: the parameter
        this.y = y;
    }
    public void move(int x) {
        x = x + 1;                // changes the PARAMETER; the field is unt
    }
}
```

A parameter or local **can** shadow a field. Inside that scope, the bare name means the local; `this.name` reaches the field. The constructor pattern `this.x = x` is idiomatic; the `move` method above is the trap — it compiles and does nothing. IDEs warn about "assignment to a parameter that shadows a field"; a good habit is to name parameters exactly like the fields *only* in constructors and setters, where `this.` is expected.

Lifetime: stack versus heap

A local or parameter lives in the current **stack frame**; it is created when its declaration executes and destroyed when the block ends (the frame is popped when the method returns). An object lives on the **heap** from `new` until no reference to it remains, at which point

the garbage collector may reclaim it. A field lives as long as the object (or, for `static`, as long as the class).

So a local *reference* can die while the object it pointed to lives on — if the object was stored in a field, added to a collection, or returned:

```
static List<String> build() {
    List<String> local = new ArrayList<>();      // the reference is a local
    local.add("x");
    return local;                                // the LIST survives; the var
}
```

And an object can outlive the method that made it but still be garbage as soon as nothing refers to it. Lifetime belongs to the object; scope belongs to the name.

Static context

A `static` method or a static initialiser has no `this`. From there you can use static fields and static methods directly, but instance members only through an explicit object:

```
public class App {
    private int counter;                         // instance
    private static int total;                    // static

    public static void main(String[] args) {
        total++;                                 // fine
        counter++;                               // error: non-static variabl
        new App().counter++;                     // fine: through an object
        run();                                   // error if run is an instan
    }
    void run() { counter++; total++; }           // instance methods may use
}
```

The rule in one line: **instance members need an instance**. The reverse is free — instance code may use static members. The error message names the problem exactly; the fix is either to create an object or to decide the member should have been static (it did not use instance state).

Static *initialisation* order can also bite: a static field initialised from a static method that reads another static field declared *later* sees its default value. Declare in dependency order.

Effectively final

Java 8 introduced a term you will meet with lambdas and inner classes (Module 11):

```
int base = 10;
Runnable r = () → System.out.println(base + 1);    // fine: base is effectivel
base = 20;                                          // now the lambda line is
```

A local is *effectively final* if it is never reassigned after initialisation. Lambdas and anonymous/local classes may capture only effectively-final locals — because they copy the value into the closure, and a copy of a variable that later changes would lie. The same rule applies to variables used inside a `try`-with-resources. If you need to mutate captured state, use a field, an array cell, or an `AtomicInteger`.

Initialisation order within a class

For an object: field initialisers and instance initialiser blocks run in textual order, then the constructor body. For a class: static field initialisers and static blocks run in textual order, once, on first use. Referencing a field before its initialiser has run reads its default (0/false/null) — the compiler catches some forward references, not all (a method call can hide one). Module 7 has the full sequence with inheritance.

Naming across scopes

Because fields and locals can collide, some teams prefix fields (`mCount`, `_count`, `count_`) or use `this.` consistently. The mainstream Java convention is plain names with `this.` in constructors and setters only, and short methods where shadowing has no room to hide. Whatever the convention, the compiler's rules are the ones above.

Interview angle

> *"Difference between scope and lifetime?"* Where a name is usable vs how long the value exists; a returned object outlives its local reference.
>
> *"Can a local shadow a field?"* Yes; a local cannot shadow another local.

"Why can't `main` call an instance method directly?" No `this` in a static context; create an object.

"What is effectively final?" Never reassigned after initialisation; required for lambda capture.

Key takeaways

Locals: declaration to end of block; parameters: the method; fields: the class. Locals cannot shadow locals; they can shadow fields (`this.` reaches the field).

Objects live on the heap as long as they are referenced — independent of the scope of any variable naming them.

Static context has no `this`: instance members need an instance.

Effectively final = never reassigned; lambdas capture only those.

EXERCISE 1

Fix the shadowing

The `Point` class below has two shadowing bugs: the constructor assigns the parameters to themselves, and `move` updates its parameters instead of the fields. Fix both using `this`, without renaming anything.

Then read a starting point `x y`, an integer `n`, and `n` moves `dx dy`; apply them and print `(<x>, <y>)`.

Example: `1 2`, `2`, `3 4`, `-1 0` → `(3, 6)`

INPUT 1

```
1 2
2
3 4
-1 0
```

OUTPUT 1

```
(3, 6)
```

INPUT 2

```
0 0
0
```

OUTPUT 2

```
(0, 0)
```

REFERENCE SOLUTION

```java
import java.util.*;

class Point {
    private int x, y;

    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    void move(int x, int y) {
        this.x = this.x + x;
        this.y = this.y + y;
    }

    public String toString() { return "(" + x + ", " + y + ")"; }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        Point p = new Point(in.nextInt(), in.nextInt());
        int n = in.nextInt();
        for (int i = 0; i < n; i++) p.move(in.nextInt(), in.nextInt());
        System.out.println(p);
    }
}
```

EXERCISE 2

Ids that survive the call

A `static int nextId` field lives for the whole program; a local lives for one call. Write `static int issue()` that returns `nextId++` (ids start at 1) and `static String label(String kind)` that declares a **local** `count` starting at 0, increments it, and returns `kind + "#" + issue() + " (local count " + count + ")"`. Read `n` kinds and print `label(kind)` for each, then `issued=<nextId - 1>`. The local count is 1 every time; the static id keeps climbing.

Example: `3` then `ticket ticket badge` →

```
ticket#1 (local count 1)
ticket#2 (local count 1)
badge#3 (local count 1)
issued=3
```

INPUT 1

```
3
ticket ticket badge
```

OUTPUT 1

```
ticket#1 (local count 1)
ticket#2 (local count 1)
badge#3 (local count 1)
issued=3
```

INPUT 2

```
1
pass
```

OUTPUT 2

```
pass#1 (local count 1)
issued=1
```

REFERENCE SOLUTION

```
import java.util.Scanner;

public class Main {
    static int nextId = 1;

    static int issue() {
        return nextId++;
    }

    static String label(String kind) {
        int count = 0;
        count++;
        return kind + "#" + issue() + " (local count " + count + ")";
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) System.out.println(label(in.next()));
        System.out.println("issued=" + (nextId - 1));
    }
}
```

QUIZ

1. `int t = 0; for (int i = 0; i < 3; i++) { int t = i; }` — result?

A. Compiles; inner `t` shadows the outer

B. Compile error: a local cannot redeclare a name from an enclosing local scope

C. Runtime error

D. `t` becomes 2

2. A method returns a `List` it created as a local variable. After the method returns…

A. The list is destroyed with the frame

B. The list lives on — the caller holds a reference; only the local *variable* is gone

C. The list becomes immutable

D. It is copied

3. What does *effectively final* mean?

A. Declared `final`

B. Never reassigned after initialisation, even without the keyword

C. A constant expression

D. A static field

4. Inside a static method, `this` is…

A. The class object

B. Not available — there is no instance

C. null

D. The first parameter

5. In `void set(int x) { x = x; }` the field `x` …

A. Is set to the parameter

B. Is untouched — both names refer to the parameter

C. Is set to 0

D. Causes a compile error

ANSWER KEY · SCOPE, LIFETIME AND STATIC CONTEXT

1. B Java forbids shadowing between locals in the same method. Only fields can be shadowed by locals or parameters.

2. B Scope belongs to the name; lifetime belongs to the object. Objects live while any reference reaches them.

3. B Lambdas and inner classes may capture only such locals, because they copy the value and a later change would go unseen.

4. B Static code runs without an object; instance fields and methods need an explicit reference.

5. B The parameter shadows the field, so the statement assigns the parameter to itself. `this.x = x` is the fix.

CHECKPOINT · 22 MIN

Checkpoint — Methods

This checkpoint covers method declarations and calls, static versus instance, pass-by-value, overloading and its resolution rules, varargs, recursion, and scope.

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 a method signature consists of, and what it excludes.
- What `fill(int[] a) { a[0] = 9; a = new int[1]; }` does to the caller's array.
- Which overload `f(5)` picks among `f(long)`, `f(Integer)` and `f(int...)`.
- What `print(null)` does to `print(String...)`.
- What every recursion needs, and what happens without it.
- Why `main` cannot call an instance method directly.

The programs are a recursion with a helper and an accumulator, and a small overloaded utility exercised through a fixed input format.

EXERCISE 1

Permutations by backtracking

Read one word of 1–6 **distinct** lowercase letters. Print every permutation of its letters, one per line, in lexicographic order. Generate them recursively with backtracking: pick an unused letter (in alphabetical order), recurse, un-pick. Sort the letters first so the output is ordered without sorting afterwards.

Example: `cab` →

```
abc
acb
bac
bca
cab
cba
```

INPUT 1

```
cab
```

OUTPUT 1

```
abc
acb
bac
bca
cab
cba
```

INPUT 2

```
a
```

OUTPUT 2

```
a
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    static char[] letters;
    static boolean[] used;
    static StringBuilder current = new StringBuilder();
    static StringBuilder out = new StringBuilder();

    static void permute() {
        if (current.length() == letters.length) {
            out.append(current).append('\n');
            return;
        }
        for (int i = 0; i < letters.length; i++) {
            if (used[i]) continue;
            used[i] = true;
            current.append(letters[i]);
            permute();
            current.deleteCharAt(current.length() - 1);
            used[i] = false;
        }
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        letters = in.next().toCharArray();
        Arrays.sort(letters);
        used = new boolean[letters.length];
        permute();
        System.out.print(out);
    }
}
```

EXERCISE 2

Recursive binary search with a helper

Read an integer `n`, then `n` integers in **non-decreasing** order, then an integer `q` and `q` query values. For each query print the index of the value in the array (any index if it occurs more than once) or `-1`.

Write a public wrapper `static int search(int[] a, int target)` that calls a private recursive helper `search(int[] a, int target, int lo, int hi)`. Compute the midpoint as `lo + (hi - lo) / 2`.

INPUT 1

```
5
1 3 5 7 9
3
7 4 1
```

OUTPUT 1

```
3
-1
0
```

INPUT 2

```
1
42
2
42 41
```

OUTPUT 2

```
0
-1
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    static int search(int[] a, int target) {
        return search(a, target, 0, a.length - 1);
    }

    private static int search(int[] a, int target, int lo, int hi) {
        if (lo > hi) return -1;
        int mid = lo + (hi - lo) / 2;
        if (a[mid] == target) return mid;
        if (a[mid] < target) return search(a, target, mid + 1, hi);
        return search(a, target, lo, mid - 1);
    }

    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();
        int q = in.nextInt();
        for (int i = 0; i < q; i++) System.out.println(search(a, in.nextInt()));
    }
}
```

EXERCISE 3

Every subset

Read an integer `n` and `n` distinct words. Print every subset of the words, one per line, as the words joined by spaces (the empty subset prints as `{}`), in the order produced by the recursion "for each word, first leave it out, then take it" — so the empty set comes first and the full set last. Finish with `subsets=<2^n>`.

Example: `2` then `a b` →

```
{}
b
a
a b
subsets=4
```

INPUT 1

```
2
a b
```

OUTPUT 1

```
{}
b
a
a b
subsets=4
```

INPUT 2

```
0
```

OUTPUT 2

```
{}
subsets=1
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    static String[] words;
    static int count = 0;

    static void subsets(int i, List<String> chosen) {
        if (i == words.length) {
            System.out.println(chosen.isEmpty() ? "{}" : String.join(" ", chosen));
            count++;
            return;
        }
        subsets(i + 1, chosen);                 // leave words[i] out
        chosen.add(words[i]);                   // take it
        subsets(i + 1, chosen);
        chosen.remove(chosen.size() - 1);       // undo, so the caller's list is unc
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        words = new String[n];
        for (int i = 0; i < n; i++) words[i] = in.next();
        subsets(0, new ArrayList<>());
        System.out.println("subsets=" + count);
    }
}
```

CHECKPOINT QUESTIONS

1. Which pair of methods can coexist in one class?

A. `int f(int a)` and `long f(int b)`

B. `int f(int a)` and `int f(long a)`

C. `void f(int a)` and `void f(int b)`

D. `static int f()` and `int f()`

2. `static void f(int[] a) { a[0] = 9; a = new int[]{1}; }` — after `f(x)` where `x = {0}`, `x[0]` and `x.length` are…

A. 1 and 1

B. 9 and 1

C. 0 and 1

D. 9 and 2

3. Overload resolution happens…

A. At run time, by the object's class

B. At compile time, by the arguments' declared types

C. At class-load time

D. At JIT time

4. `f(5)` with `f(double)` and `f(Integer)` — which runs?

A. `f(Integer)`

B. `f(double)` — primitive widening precedes boxing

C. Ambiguous

D. Neither

5. `show(new String[]{"a","b"})` to `show(Object... xs)` gives `xs.length` =

A. 1

B. 2 — a matching array is passed as the varargs array

C. 0

D. Compile error

6. How many stack frames does `factorial(5)` (recursing to `factorial(1)`) have live at its deepest point?

A. 1

B. 5

C. 10

D. None — recursion does not use the stack

7. Which recursion is safe on an input of one million elements?

A. Recursive linear sum over the array

B. Recursive binary search

C. Recursive string reversal by `substring`

D. Naive recursive Fibonacci

8. Memoization helps when…

A. The base case is missing

B. The same subproblems recur (overlapping subproblems)

C. Recursion is tail-recursive

D. Inputs are strings

9. A parameter named like a field…

A. Is a compile error

B. Shadows the field inside the method; `this.field` reaches the field

C. Automatically assigns the field

D. Is ignored

10. A lambda captures a local `int count` that is later incremented. The compiler…

A. Allows it

B. Rejects it: captured locals must be effectively final

C. Copies it silently

D. Makes it volatile

11. `public static void main(String... args)` is…

A. Invalid

B. A valid entry point — varargs is a `String[]` parameter

C. Valid only with `-ea`

D. Valid only in Java 21

12. To return two values from a method, idiomatic Java uses…

A. Two `return` statements

B. An out-parameter array

C. A small object or record holding both

D. A global variable

ANSWER KEY · CHECKPOINT — METHODS

1. B Only the second pair differs in parameter types. Return type, parameter names and `static` do not distinguish signatures.

2. B The mutation `a[0] = 9` reaches the shared array; the reassignment is local. `x` is the original one-element array with 9 in it.

3. B That is the defining difference between overloading (static) and overriding (dynamic dispatch).

4. B `int` widens to `double` in phase 1; boxing to `Integer` is phase 2.

5. B `String[]` is an `Object[]`, so it becomes the array itself. Cast to `Object` to make it a single element.

6. B Frames for n = 5, 4, 3, 2, 1 are all live until the base case returns; then they pop one by one.

7. B Only binary search has logarithmic depth (~20). The others have linear or exponential call depth/count.

8. B Storing each subproblem's answer turns exponential recomputation into linear work, as in Fibonacci.

9. B Legal and idiomatic in constructors and setters with `this.x = x`; a trap anywhere else.

10. B Captured variables are copied; mutation after capture would be invisible, so Java forbids it. Use a field or a holder.

11. B `String...` compiles to exactly `String[]`, so the launcher accepts it.

12. C A method returns one value; make it a value that holds both. Records (Module 10) make this a one-liner.