Chapter 9: Methods

Computer Fundamentals and Introduction to Programming

Definition | Parameters | Return | Scope | Examples

Sercan KÜLCÜ · Giresun University

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/computer/slides/Chapter_09_Methods.html

INTRODUCTION

Learning Outcomes

Defining

Write method definitions and calls.

Data flow

Use parameters and return values.

Scope

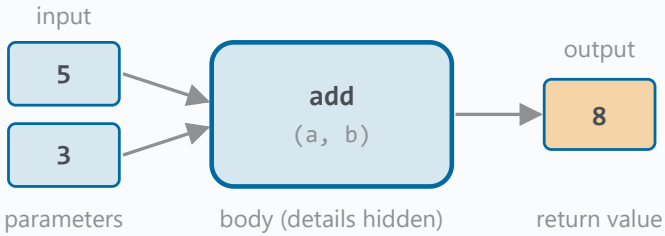
Explain where local variables live.

Design

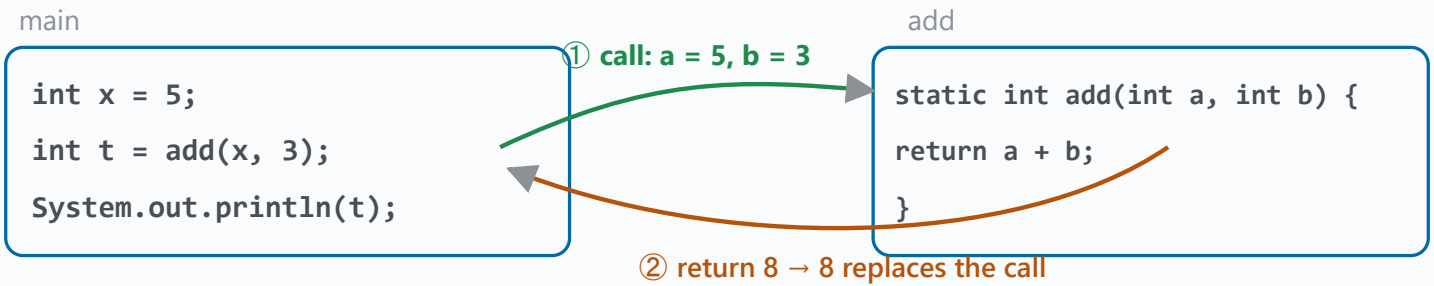
Build short, focused, testable methods.

What Is a Function?

- In Java, functions are defined inside a class as **methods**.
- A method names a task and makes it a reusable unit.
- Input arrives through **parameters**; the result comes back as the **return value**.
- Example: `Math.sqrt(16)` → 4.0; we don't need to know how it works.



How a Method Runs



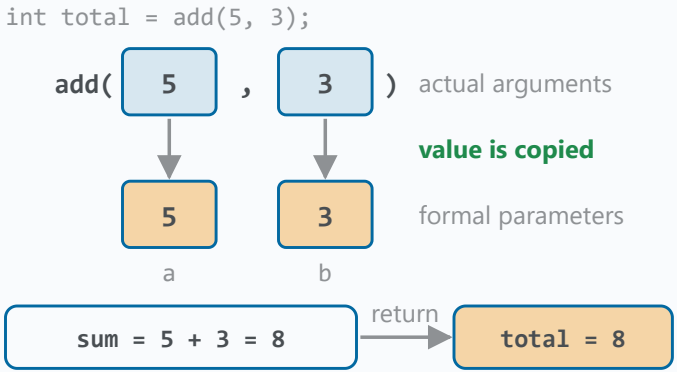
Evaluate the arguments → Copy into parameters → Run the body → Return a value → Resume the caller

- Each call works with its own parameters and local variables.

Method Definition and Call

```
static int add(int a, int b) {
    int sum = a + b;
    return sum;
}

int total = add(5, 3);
System.out.println(total);
```



- The return type is int; a and b are the formal parameters.
- 5 and 3 are the actual arguments.
- The returned value can be assigned to a variable or used in another expression.

Code Trace: `add(5, 3)`



main is running. The right-hand side comes first: `add(5, 3)` is called.

Step 1 / 6

```
1 static int add(int a, int b) {
2     int sum = a + b;
3     return sum;
4 }
5
6 // in main:
▶ int total = add(5, 3);
8 System.out.println(total);
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

Code Trace: `add(5, 3)`



A new **frame** opens: the arguments 5 and 3 are copied into parameters **a** and **b**.

Step 2 / 6

```
▶ static int add(int a, int b) {
2     int sum = a + b;
3     return sum;
4 }
5
6 // in main:
7 int total = add(5, 3);
8 System.out.println(total);
```

VARIABLES

`a = 5` `b = 3`

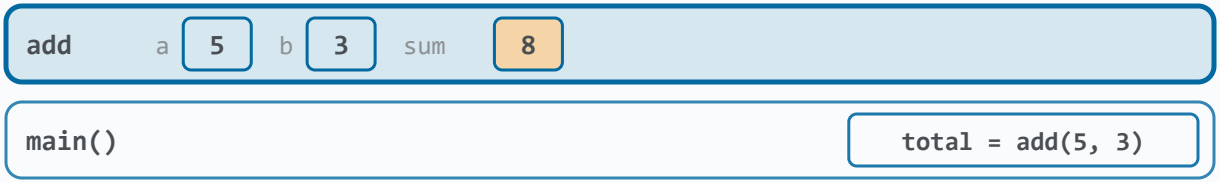
CALL STACK

`add(5, 3)`

`main()`

OUTPUT

Code Trace: `add(5, 3)`



Local variable **sum = 5 + 3 = 8**; it lives only in add's frame.

Step 3 / 6

```
1 static int add(int a, int b) {
▶     int sum = a + b;
3     return sum;
4 }
5
6 // in main:
7 int total = add(5, 3);
8 System.out.println(total);
```

VARIABLES

`a = 5` `b = 3` `sum = 8`

CALL STACK

`add(5, 3)`

`main()`

OUTPUT

Code Trace: `add(5, 3)`



`return sum`: 8 goes back to the caller and add's frame closes.

Step 4 / 6

```
1 static int add(int a, int b) {
2     int sum = a + b;
▶     return sum;
4 }
5
6 // in main:
7 int total = add(5, 3);
8 System.out.println(total);
```

VARIABLES

`a = 5` `b = 3` `sum = 8`

CALL STACK

`add(5, 3)`

`main()`

OUTPUT

Code Trace: `add(5, 3)`



The returned value **replaces the call**: `total = 8`. `a`, `b` and `sum` are gone.

Step 5 / 6

```
1 static int add(int a, int b) {
2     int sum = a + b;
3     return sum;
4 }
5
6 // in main:
▶ int total = add(5, 3);
8 System.out.println(total);
```

VARIABLES

`total = 8`

CALL STACK

`main()`

OUTPUT

Code Trace: `add(5, 3)`



8 is printed.

Step 6 / 6

```
1 static int add(int a, int b) {
2     int sum = a + b;
3     return sum;
4 }
5
6 // in main:
7 int total = add(5, 3);
▶ System.out.println(total);
```

VARIABLES

`total = 8`

CALL STACK

`main()`

OUTPUT

```
8
```

The Call Stack: when a method calls a method



main calls `avg(9, 7)`. For now the stack has one frame.

Step 1 / 9

```
1 static int add(int a, int b) {
2     return a + b;
3 }
4 static int avg(int x, int y) {
5     int t = add(x, y);
6     return t / 2;
7 }
▶ int o = avg(9, 7);  // in main
9 System.out.println(o);
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

The Call Stack: when a method calls a method



The **avg** frame is pushed on top of main: x = 9, y = 7.

Step 2 / 9

```
1 static int add(int a, int b) {
2     return a + b;
3 }
▶ static int avg(int x, int y) {
5     int t = add(x, y);
6     return t / 2;
7 }
8 int o = avg(9, 7);  // in main
9 System.out.println(o);
```

VARIABLES

x = 9 y = 7

CALL STACK

avg(9, 7)

main()

OUTPUT

The Call Stack: when a method calls a method



avg calls a method too: `add(x, y)` → the **values** of x and y (9, 7) are sent.

Step 3 / 9

```
1 static int add(int a, int b) {
2     return a + b;
3 }
4 static int avg(int x, int y) {
▶     int t = add(x, y);
6     return t / 2;
7 }
8 int o = avg(9, 7);  // in main
9 System.out.println(o);
```

VARIABLES

x = 9 y = 7

CALL STACK

avg(9, 7)

main()

OUTPUT

The Call Stack: when a method calls a method



add is on top: 3 frames. Only the top one runs.

Step 4 / 9

```
▶ static int add(int a, int b) {
2     return a + b;
3 }
4 static int avg(int x, int y) {
5     int t = add(x, y);
6     return t / 2;
7 }
8 int o = avg(9, 7);  // in main
9 System.out.println(o);
```

VARIABLES

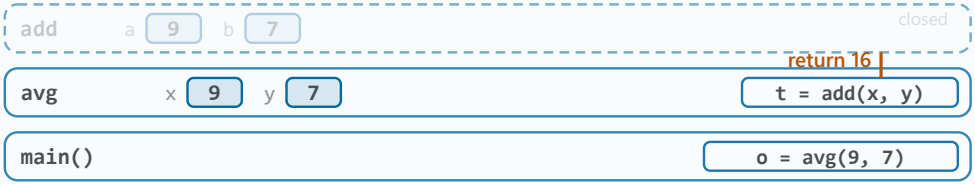
a = 9 b = 7

CALL STACK

- add(9, 7)
- avg(9, 7)
- main()

OUTPUT

The Call Stack: when a method calls a method



return 16: add's frame closes (last in, first out).

Step 5 / 9

```
1 static int add(int a, int b) {
▶       return a + b;
3 }
4 static int avg(int x, int y) {
5     int t = add(x, y);
6     return t / 2;
7 }
8 int o = avg(9, 7);  // in main
9 System.out.println(o);
```

VARIABLES

a = 9 b = 7

CALL STACK

- add(9, 7)
- avg(9, 7)
- main()

OUTPUT

The Call Stack: when a method calls a method



16 replaces the call: **t = 16**. avg resumes where it left off.

Step 6 / 9

```
1 static int add(int a, int b) {
2     return a + b;
3 }
4 static int avg(int x, int y) {
▶     int t = add(x, y);
6     return t / 2;
7 }
8 int o = avg(9, 7);  // in main
9 System.out.println(o);
```

VARIABLES

x = 9 y = 7 t = 16

CALL STACK

avg(9, 7)

main()

OUTPUT

The Call Stack: when a method calls a method



return t / 2 → 16 / 2 = 8 (integer division). avg's frame closes.

Step 7 / 9

```
1 static int add(int a, int b) {
2     return a + b;
3 }
4 static int avg(int x, int y) {
5     int t = add(x, y);
▶     return t / 2;
7 }
8 int o = avg(9, 7);  // in main
9 System.out.println(o);
```

VARIABLES

x = 9 | y = 7 | t = 16

CALL STACK

avg(9, 7)

main()

OUTPUT

The Call Stack: when a method calls a method



Back in main: **o = 8**. x, y, t, a and b no longer exist.


```
1 static int add(int a, int b) {
2     return a + b;
3 }
4 static int avg(int x, int y) {
5     int t = add(x, y);
6     return t / 2;
7 }
▶ int o = avg(9, 7);  // in main
9 System.out.println(o);
```

VARIABLES

o = 8

CALL STACK

main()

OUTPUT

The Call Stack: when a method calls a method



8 is printed. Frames behave like a stack: last opened, first closed.

Step 9 / 9

```
1 static int add(int a, int b) {
2     return a + b;
3 }
4 static int avg(int x, int y) {
5     int t = add(x, y);
6     return t / 2;
7 }
8 int o = avg(9, 7);  // in main
▶ System.out.println(o);
```

VARIABLES

o = 8

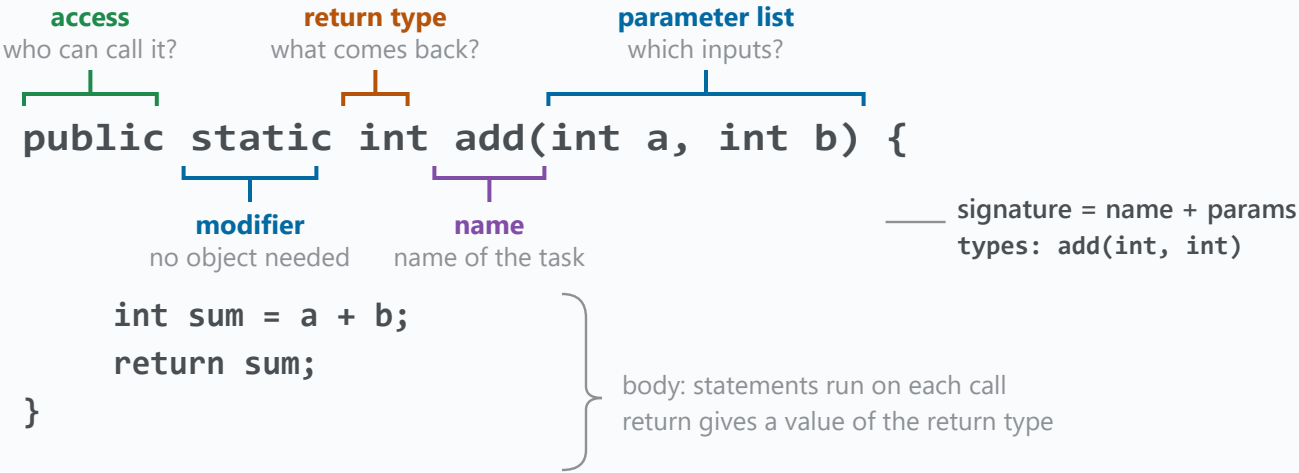
CALL STACK

main()

OUTPUT

```
8
```

Parts of a Method Definition



In these examples, methods live in the same class as `main` and are `static`.

BASICS

The `main` Signature and `static`

`public static void main(String[] args)`
`public` : so the JVM can call it from outside
`static` : callable without creating an object
`void` : returns no value
`main` : the name the JVM looks for
`String[] args` : command-line arguments

```
void greet() { }
public static void main(String[]
args) {
    greet();   // error
}
```

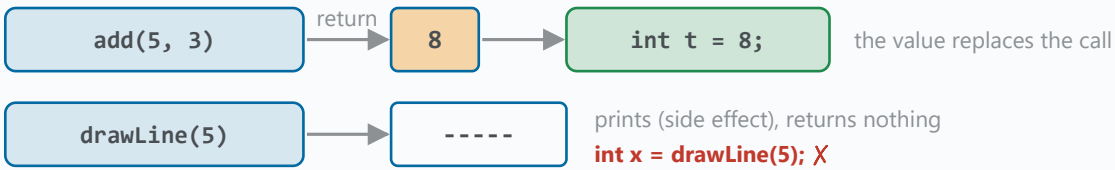
```
error: non-static method greet() cannot be referenced from a static context
```

- A **static** method belongs to the class and is called without an object. A non-static method belongs to an object.
- A static method cannot directly use a non-static method or field; the fix is to make that method `static` too.
- When the JVM starts a program there are no objects yet, so `main` must be static.

void Methods

```
static void drawLine(int n) {
    for (int i = 0; i < n; i++)
        System.out.print("-");
    System.out.println();
}
// in main:
drawLine(5);           // -----
int x = drawLine(5);   // ERROR: void has no value
```

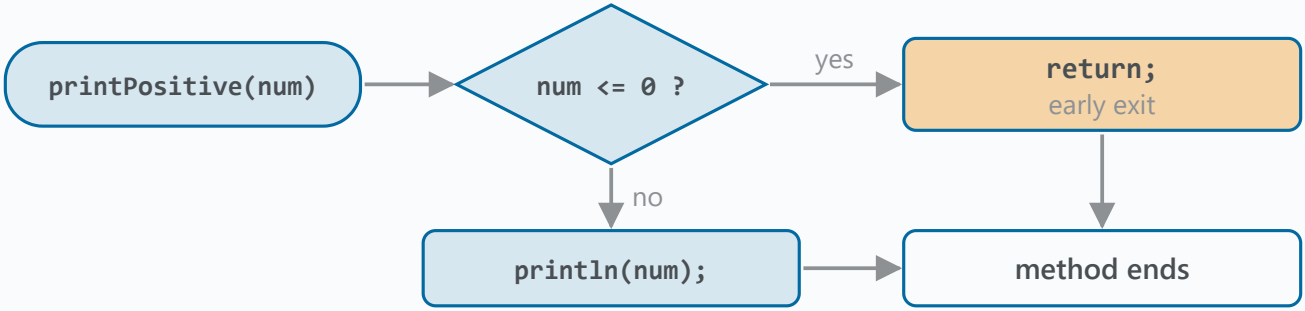
- `void` : the method returns no value to the call; its work is a side effect (such as printing).
- So a void call cannot be part of an assignment or expression.
- `return;` (with no value) can exit a void method early.



Early Return

```
static void printPositive(int num) {
    if (num <= 0) return;
    System.out.println(num);
}
```

- Handling special cases first can reduce nesting.
- In a value-returning method, every path must return a compatible value.



printPositive(-3) → no output

printPositive(7) → 7

Code Trace: `printPositive(-3)` and `(7)`



First call: argument −3.

Step 1 / 8

```
1 static void printPositive(int num) {
2     if (num <= 0) return;
3     System.out.println(num);
4 }
5 // in main:
▶ printPositive(-3);
7 printPositive(7);
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

Code Trace: `printPositive(-3)` and `(7)`



The frame opens: **num = −3**.

Step 2 / 8

```
▶ static void printPositive(int num) {
2     if (num <= 0) return;
3     System.out.println(num);
4 }
5 // in main:
6 printPositive(-3);
7 printPositive(7);
```

VARIABLES

`num = −3`

CALL STACK

`printPositive(−3)`

`main()`

OUTPUT

Code Trace: `printPositive(-3)` and `(7)`



−3 <= 0 → **true** → `return;` The method ends at once; println is never reached.

Step 3 / 8

```
1 static void printPositive(int num) {
▶     if (num <= 0) return;
3     System.out.println(num);
4 }
5 // in main:
6 printPositive(-3);
7 printPositive(7);
```

VARIABLES

`num = -3`

CALL STACK

`printPositive(-3)`

`main()`

OUTPUT

Code Trace: `printPositive(-3)` and `(7)`

main() printPositive(7)

main continues on the next line: second call, argument 7.

Step 4 / 8

```
1 static void printPositive(int num) {
2     if (num <= 0) return;
3     System.out.println(num);
4 }
5 // in main:
6 printPositive(-3);
▶ printPositive(7);
```

VARIABLES

—

CALL STACK

main()

OUTPUT

Code Trace: `printPositive(-3)` and `(7)`



A brand-new frame: **num = 7** (the previous call's `num` is long gone).

Step 5 / 8

```
▶ static void printPositive(int num) {
2     if (num <= 0) return;
3     System.out.println(num);
4 }
5 // in main:
6 printPositive(-3);
7 printPositive(7);
```

VARIABLES

`num = 7`

CALL STACK

`printPositive(7)`

`main()`

OUTPUT

Code Trace: `printPositive(-3)` and `(7)`



7 <= 0 → **false** → no early return; carry on.

Step 6 / 8

```
1 static void printPositive(int num) {
▶     if (num <= 0) return;
3     System.out.println(num);
4 }
5 // in main:
6 printPositive(-3);
7 printPositive(7);
```

VARIABLES

`num = 7`

CALL STACK

`printPositive(7)`

`main()`

OUTPUT

Code Trace: `printPositive(-3)` and `(7)`



7 is printed.

Step 7 / 8

```
1 static void printPositive(int num) {
2     if (num <= 0) return;
      System.out.println(num);
4 }
5 // in main:
6 printPositive(-3);
7 printPositive(7);
```

VARIABLES

`num = 7`

CALL STACK

`printPositive(7)`

`main()`

OUTPUT

```
7
```

Code Trace: `printPositive(-3)` and `(7)`



The closing brace is reached: a void method returns here on its own.

Step 8 / 8

```
1 static void printPositive(int num) {
2     if (num <= 0) return;
3     System.out.println(num);
▶ }
5 // in main:
6 printPositive(-3);
7 printPositive(7);
```

VARIABLES

`num = 7`

CALL STACK

`printPositive(7)`

`main()`

OUTPUT

```
7
```

Compile Errors: `return` and Type

```
static int check(int n) {
    if (n > 0) {
        return 1;
    }
}
```

`error: missing return statement` (at the closing `}`)

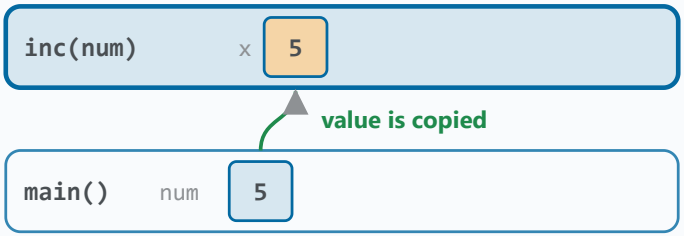
```
static int f() {
    return 2.5;
}
```

`error: incompatible types: possible lossy conversion from double to int`

- In a non-void method **every path** must end with return; here the if's false branch is missing. Fix: add `return 0;` at the end.
- The value must fit the return type: `return (int) 2.5;` compiles (returns 2), or make the return type `double`.
- `return 1;` in a `void` method → `error: incompatible types: unexpected return value`

Parameter Passing

- Java always passes arguments **by value**.
- For a primitive, a **copy** of the value goes into the parameter: two separate boxes in memory.
- Even if the method changes its copy, the caller's variable stays the same.
- For objects (e.g. arrays), a copy of the reference is passed.



Pass by Value: `inc(num)`



In main, **num = 5**: one box in main's frame.

Step 1 / 6

```
1 static void inc(int x) {
2     x++;
3 }
4 // in main:
▶ int num = 5;
6 inc(num);
7 System.out.println(num);   // 5
```

VARIABLES

num = 5

CALL STACK

main()

OUTPUT

Pass by Value: `inc(num)`



`inc(num)` : what is sent is **the value, not the variable** (5).

Step 2 / 6

```
1 static void inc(int x) {
2     x++;
3 }
4 // in main:
5 int num = 5;
▶ inc(num);
7 System.out.println(num);   // 5
```

VARIABLES

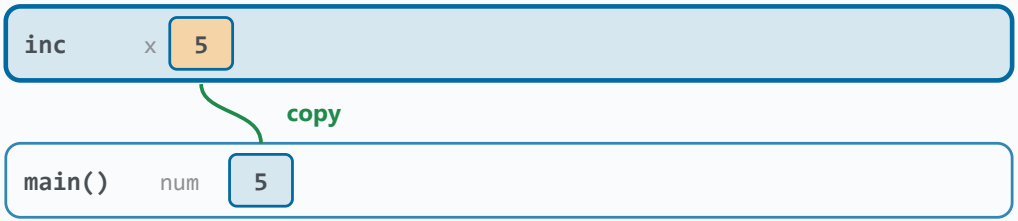
`num = 5`

CALL STACK

`main()`

OUTPUT

Pass by Value: `inc(num)`



A **separate box** in the new frame: x = 5 (a copy).

Step 3 / 6

```
▶ static void inc(int x) {
2     x++;
3 }
4 // in main:
5 int num = 5;
6 inc(num);
7 System.out.println(num);   // 5
```

VARIABLES

x = 5

CALL STACK

inc(5)

main()

OUTPUT

Pass by Value: `inc(num)`



x++ → x = 6. Only inc's box changed; `num` in main is still 5.

Step 4 / 6

```
1 static void inc(int x) {
▶     x++;
3 }
4 // in main:
5 int num = 5;
6 inc(num);
7 System.out.println(num);   // 5
```

VARIABLES

x = 6

CALL STACK

inc(5)

main()

OUTPUT

Pass by Value: `inc(num)`



The method ends; its frame and x (6) are discarded. Nothing is sent back (void).

Step 5 / 6

```
1 static void inc(int x) {
2     x++;
▶ }
4 // in main:
5 int num = 5;
6 inc(num);
7 System.out.println(num);   // 5
```

VARIABLES

`x = 6`

CALL STACK

`inc(5)`

`main()`

OUTPUT

Pass by Value: `inc(num)`



5 is printed. To change it we would write `static int inc(int x) { return x + 1; }` and `num = inc(num);`

Step 6 / 6

```
1 static void inc(int x) {
2     x++;
3 }
4 // in main:
5 int num = 5;
6 inc(num);
▶ System.out.println(num);   // 5
```

VARIABLES

`num = 5`

CALL STACK

`main()`

OUTPUT

```
5
```

The Pass-by-Value Pitfall: `swap(x, y)`



In main, **x = 3** and **y = 5**: two separate boxes in main's frame.

Step 1 / 7

```
1 static void swap(int a, int b) {
2     int t = a;
3     a = b;
4     b = t;
5 }
6 // in main:
▶ int x = 3, y = 5;
8 swap(x, y);
9 System.out.println(x + " " + y);   // 3 5
```

VARIABLES

x = 3 y = 5

CALL STACK

main()

OUTPUT

The Pass-by-Value Pitfall: `swap(x, y)`



`swap(x, y)` : what is sent is **the values, not the variables** (3 and 5).

Step 2 / 7

```
1 static void swap(int a, int b) {
2     int t = a;
3     a = b;
4     b = t;
5 }
6 // in main:
7 int x = 3, y = 5;
▶ swap(x, y);
9 System.out.println(x + " " + y);   // 3 5
```

VARIABLES

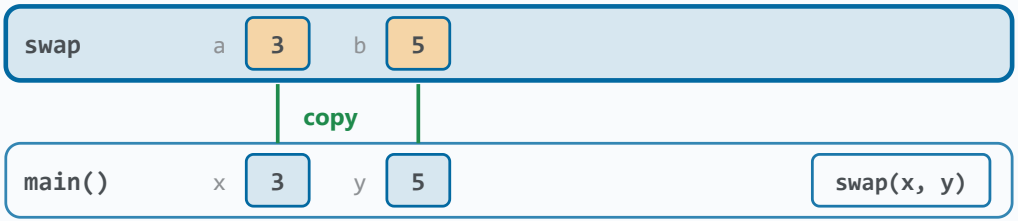
x = 3 y = 5

CALL STACK

main()

OUTPUT

The Pass-by-Value Pitfall: `swap(x, y)`



swap's frame has **two new boxes**: a = 3 and b = 5 (copies). They are not linked to x and y.

Step 3 / 7

```
▶ static void swap(int a, int b) {
2     int t = a;
3     a = b;
4     b = t;
5 }
6 // in main:
7 int x = 3, y = 5;
8 swap(x, y);
9 System.out.println(x + " " + y);   // 3 5
```

VARIABLES

a = 3 b = 5

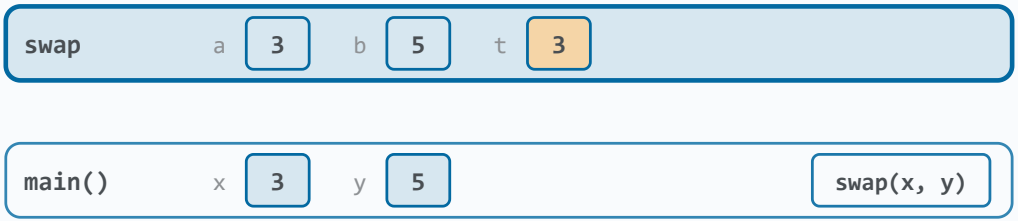
CALL STACK

swap(3, 5)

main()

OUTPUT

The Pass-by-Value Pitfall: `swap(x, y)`



t = a → t = 3: the temporary box keeps a's value.

Step 4 / 7

```
1 static void swap(int a, int b) {
▶     int t = a;
3     a = b;
4     b = t;
5 }
6 // in main:
7 int x = 3, y = 5;
8 swap(x, y);
9 System.out.println(x + " " + y);   // 3 5
```

VARIABLES

a = 3 | b = 5 | t = 3

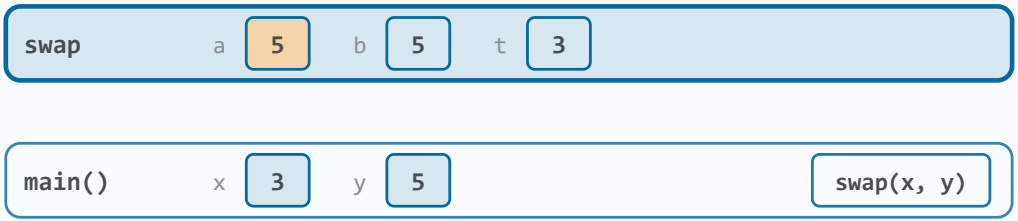
CALL STACK

swap(3, 5)

main()

OUTPUT

The Pass-by-Value Pitfall: `swap(x, y)`



a = b → a = 5. The old value of a now lives only in t.

Step 5 / 7

```
1 static void swap(int a, int b) {
2     int t = a;
▶     a = b;
4     b = t;
5 }
6 // in main:
7 int x = 3, y = 5;
8 swap(x, y);
9 System.out.println(x + " " + y);   // 3 5
```

VARIABLES

a = 5 | b = 5 | t = 3

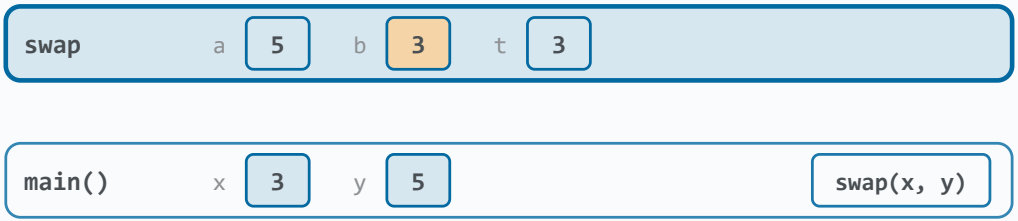
CALL STACK

swap(3, 5)

main()

OUTPUT

The Pass-by-Value Pitfall: `swap(x, y)`



b = t → b = 3. swap's **copies** are exchanged (a = 5, b = 3), but x and y in main were never touched.

Step 6 / 7

```
1 static void swap(int a, int b) {
2     int t = a;
3     a = b;
▶     b = t;
5 }
6 // in main:
7 int x = 3, y = 5;
8 swap(x, y);
9 System.out.println(x + " " + y);   // 3 5
```

VARIABLES

a = 5 | b = 3 | t = 3

CALL STACK

swap(3, 5)

main()

OUTPUT

The Pass-by-Value Pitfall: `swap(x, y)`



When swap ends, its frame (a, b, t) is discarded. main still has **x = 3, y = 5**: only the copies were swapped. A real swap needs a returned result or an array.

Step 7 / 7

```
1 static void swap(int a, int b) {
2     int t = a;
3     a = b;
4     b = t;
5 }
6 // in main:
7 int x = 3, y = 5;
8 swap(x, y);
▶ System.out.println(x + " " + y);   // 3 5
```

VARIABLES

x = 3 y = 5

CALL STACK

main()

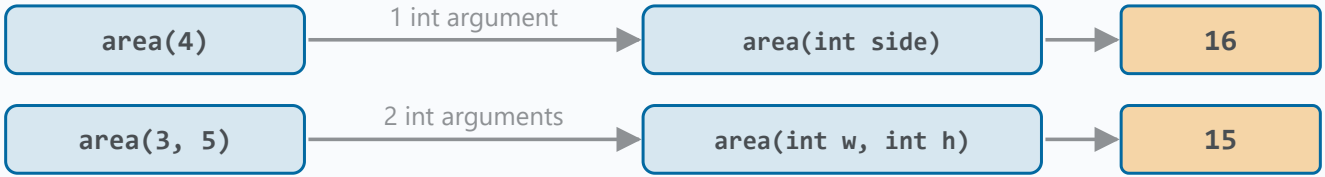
OUTPUT

```
3 5
```

Method Overloading

```
static int area(int side) {
    return side * side;
}
static int area(int w, int h) {
    return w * h;
}
// ERROR: only the return type differs
static double area(int side) { … }
```

- Same name, **different parameter list** (number, type or order).
- Changing only the return type is not enough: the call can't tell which one is meant.
- The compiler picks the matching signature from the call's arguments.
- A familiar example: `println(int)`, `println(double)`, `println(String)`.



The compiler checks argument count and types, not the return type.

Code Trace: which `area` does the compiler pick?



`area(4)` : the **compiler** looks at the arguments: one int.

Step 1 / 7

```
1 static int area(int side) {
2     return side * side;
3 }
4 static int area(int w, int h) {
5     return w * h;
6 }
▶ System.out.println(area(4));      // main
8 System.out.println(area(3, 5));   // main
```

VARIABLES

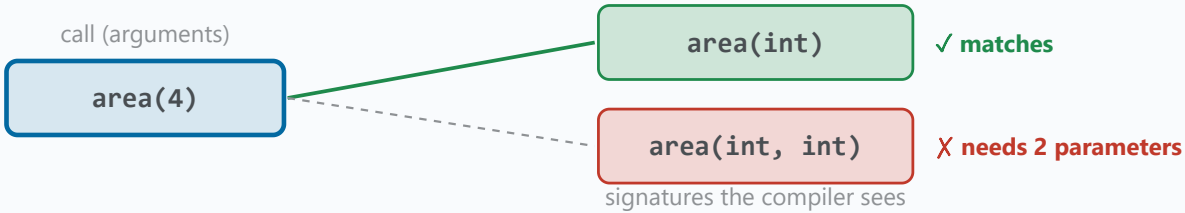
—

CALL STACK

`main()`

OUTPUT

Code Trace: which `area` does the compiler pick?



Two methods share the name; **area(int)**, whose parameter count and type match, is chosen. This is decided at compile time, before the program runs.

Step 2 / 7

```
1 static int area(int side) {
2     return side * side;
3 }
4 static int area(int w, int h) {
5     return w * h;
6 }
▶ System.out.println(area(4));      // main
8 System.out.println(area(3, 5));   // main
```

VARIABLES

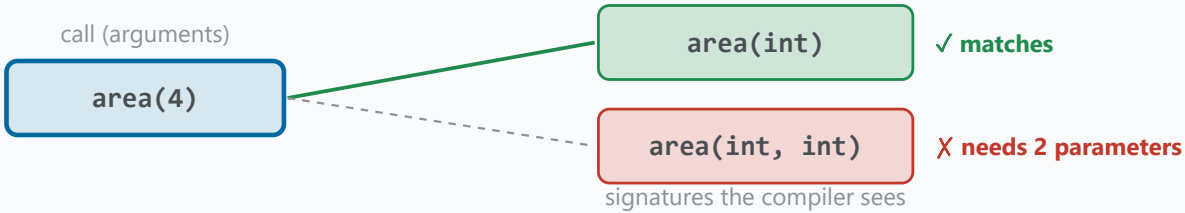
—

CALL STACK

`main()`

OUTPUT

Code Trace: which `area` does the compiler pick?



At run time area(int) is called: side = 4 → return 16.

Step 3 / 7

```
▶ static int area(int side) {
▶     return side * side;
3 }
4 static int area(int w, int h) {
5     return w * h;
6 }
7 System.out.println(area(4));      // main
8 System.out.println(area(3, 5));   // main
```

VARIABLES

`side = 4`

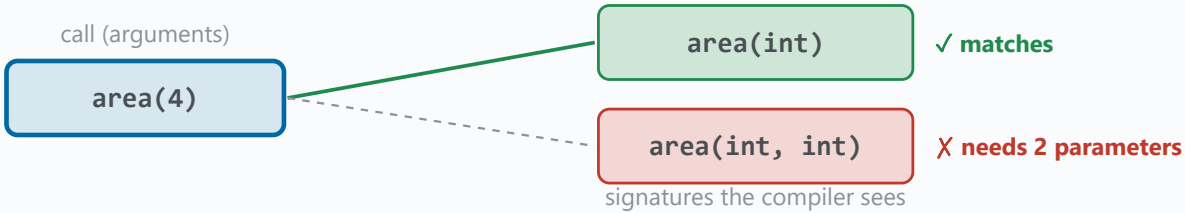
CALL STACK

`area(4)`

`main()`

OUTPUT

Code Trace: which `area` does the compiler pick?



16 is printed.

Step 4 / 7

```
1 static int area(int side) {
2     return side * side;
3 }
4 static int area(int w, int h) {
5     return w * h;
6 }
▶ System.out.println(area(4));      // main
8 System.out.println(area(3, 5));   // main
```

VARIABLES

—

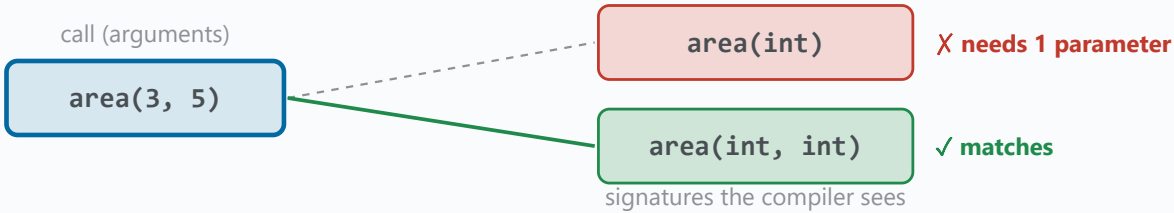
CALL STACK

`main()`

OUTPUT

```
16
```

Code Trace: which `area` does the compiler pick?



`area(3, 5)` : 2 int arguments → **area(int, int)** is chosen.

Step 5 / 7

```
1 static int area(int side) {
2     return side * side;
3 }
4 static int area(int w, int h) {
5     return w * h;
6 }
7 System.out.println(area(4));      // main
▶ System.out.println(area(3, 5));   // main
```

VARIABLES

—

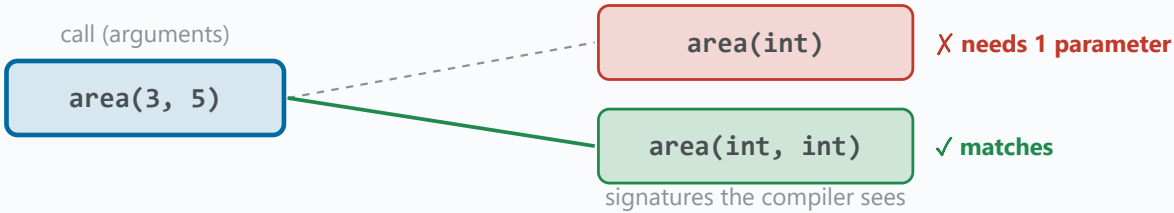
CALL STACK

`main()`

OUTPUT

```
16
```

Code Trace: which `area` does the compiler pick?



w = 3, h = 5 → return 15.

Step 6 / 7

```
1 static int area(int side) {
2     return side * side;
3 }
▶ static int area(int w, int h) {
▶     return w * h;
6 }
7 System.out.println(area(4));      // main
8 System.out.println(area(3, 5));   // main
```

VARIABLES

`w = 3` `h = 5`

CALL STACK

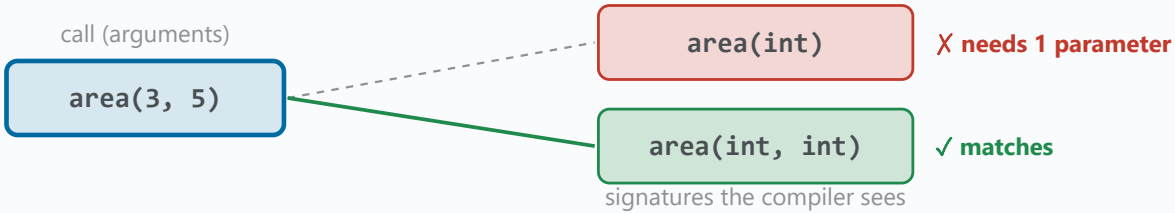
`area(3, 5)`

`main()`

OUTPUT

```
16
```

Code Trace: which `area` does the compiler pick?



15 is printed. `area(2.5)` would match no signature → **compile error** (double → int is not automatic).

Step 7 / 7

```
1 static int area(int side) {
2     return side * side;
3 }
4 static int area(int w, int h) {
5     return w * h;
6 }
7 System.out.println(area(4));      // main
▶ System.out.println(area(3, 5));   // main
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

```
16
15
```

Variable-Length Arguments (varargs)

```
static int add(int... s) {
    int t = 0;
    for (int v : s) t += v;
    return t;
}
add(1, 2, 3)   // 6
add()          // 0
add(new int[]{4, 5})   // 9
```

- `int... s` : any number of arguments; inside the method **s is an int[] array**.
- The compiler packs the call's values into an array; with no arguments you get an **empty array** (s.length == 0).
- The varargs parameter must come **last**, and a method can have only one.
- A familiar example: `printf(String, Object...)` .

Variable-Length Arguments: `add(1, 2, 3)`



`add(1, 2, 3)` : the compiler packs the three values into the array **new int[]{1, 2, 3}**.

Step 1 / 8

```
1 static int add(int... s) {
2     int t = 0;
3     for (int v : s) t += v;
4     return t;
5 }
6 // in main:
▶ System.out.println(add(1, 2, 3));
8 System.out.println(add());
```

VARIABLES

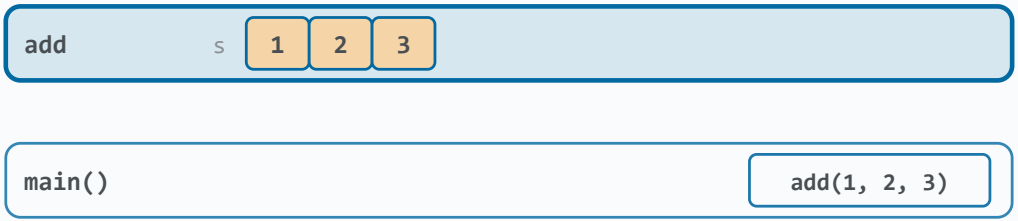
—

CALL STACK

`main()`

OUTPUT

Variable-Length Arguments: `add(1, 2, 3)`



s is an **int[]** array: s = {1, 2, 3}, s.length = 3.

Step 2 / 8

```
▶ static int add(int... s) {
2     int t = 0;
3     for (int v : s) t += v;
4     return t;
5 }
6 // in main:
7 System.out.println(add(1, 2, 3));
8 System.out.println(add());
```

VARIABLES

`s = [1, 2, 3]`

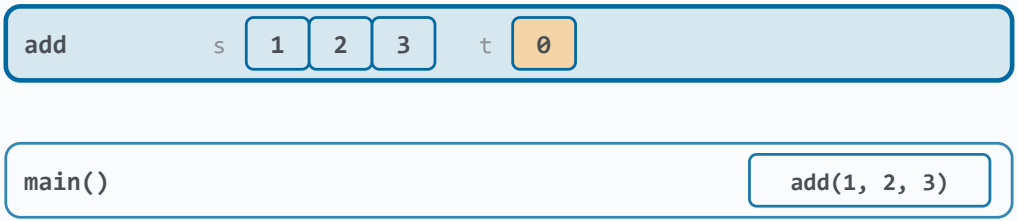
CALL STACK

`add(1, 2, 3)`

`main()`

OUTPUT

Variable-Length Arguments: `add(1, 2, 3)`



t = 0: a local box for the total.

Step 3 / 8

```
1 static int add(int... s) {
▶     int t = 0;
3     for (int v : s) t += v;
4     return t;
5 }
6 // in main:
7 System.out.println(add(1, 2, 3));
8 System.out.println(add());
```

VARIABLES

`s = [1, 2, 3]` `t = 0`

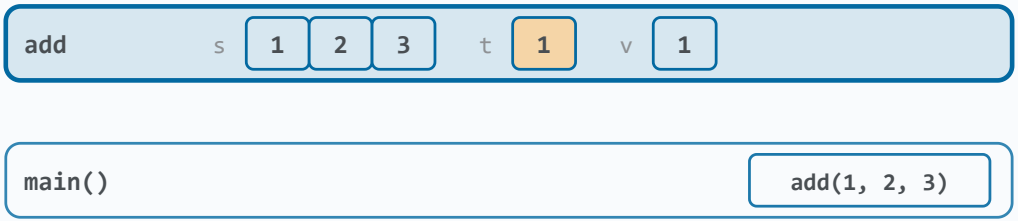
CALL STACK

`add(1, 2, 3)`

`main()`

OUTPUT

Variable-Length Arguments: `add(1, 2, 3)`



Enhanced for, pass 1: **v = 1** → t = 0 + 1 = **1**.

Step 4 / 8

```
1 static int add(int... s) {
2     int t = 0;
▶     for (int v : s) t += v;
4     return t;
5 }
6 // in main:
7 System.out.println(add(1, 2, 3));
8 System.out.println(add());
```

VARIABLES

`s = [1, 2, 3]` `t = 1` `v = 1`

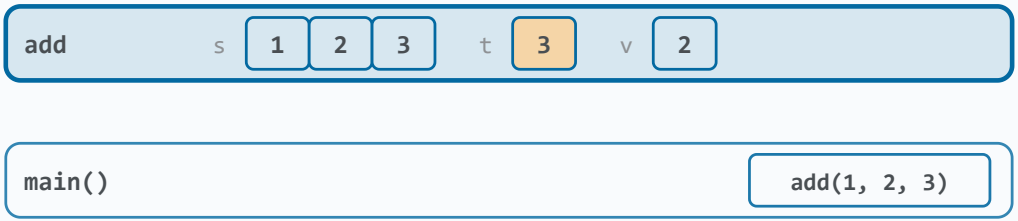
CALL STACK

`add(1, 2, 3)`

`main()`

OUTPUT

Variable-Length Arguments: `add(1, 2, 3)`



Pass 2: **v = 2** → t = 1 + 2 = **3**.

Step 5 / 8

```
1 static int add(int... s) {
2     int t = 0;
▶     for (int v : s) t += v;
4     return t;
5 }
6 // in main:
7 System.out.println(add(1, 2, 3));
8 System.out.println(add());
```

VARIABLES

`s = [1, 2, 3]` `t = 3` `v = 2`

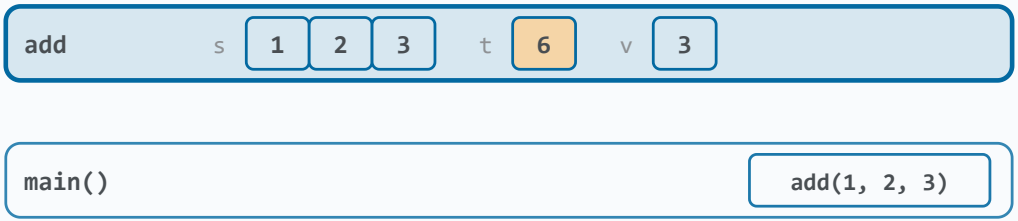
CALL STACK

`add(1, 2, 3)`

`main()`

OUTPUT

Variable-Length Arguments: `add(1, 2, 3)`



Pass 3: **v = 3** → t = 3 + 3 = **6**. No elements left.

Step 6 / 8

```
1 static int add(int... s) {
2     int t = 0;
▶     for (int v : s) t += v;
4     return t;
5 }
6 // in main:
7 System.out.println(add(1, 2, 3));
8 System.out.println(add());
```

VARIABLES

`s = [1, 2, 3]` `t = 6` `v = 3`

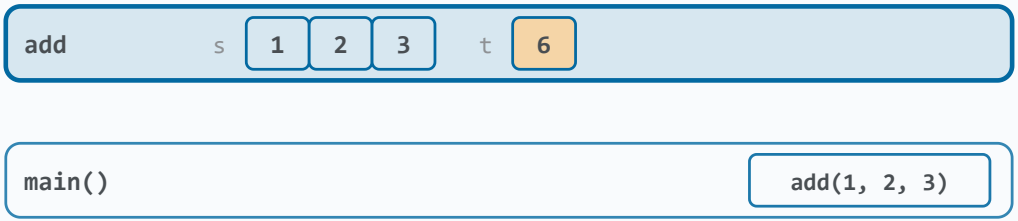
CALL STACK

`add(1, 2, 3)`

`main()`

OUTPUT

Variable-Length Arguments: `add(1, 2, 3)`



return t → returns 6. v went out of scope with the loop.

Step 7 / 8

```
1 static int add(int... s) {
2     int t = 0;
3     for (int v : s) t += v;
▶     return t;
5 }
6 // in main:
7 System.out.println(add(1, 2, 3));
8 System.out.println(add());
```

VARIABLES

`s = [1, 2, 3]` `t = 6`

CALL STACK

`add(1, 2, 3)`

`main()`

OUTPUT

Variable-Length Arguments: `add(1, 2, 3)`

`main()`

6 is printed. add's frame (s, t) is gone.

Step 8 / 8

```
1 static int add(int... s) {
2     int t = 0;
3     for (int v : s) t += v;
4     return t;
5 }
6 // in main:
▶ System.out.println(add(1, 2, 3));
8 System.out.println(add());
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

```
6
```

Empty Call: `add()`



`add()` : no arguments → the compiler passes an **empty array** (new int[0]), not null.

Step 1 / 6

```
1 static int add(int... s) {
2     int t = 0;
3     for (int v : s) t += v;
4     return t;
5 }
6 // in main:
7 System.out.println(add(1, 2, 3));
▶ System.out.println(add());
```

VARIABLES

—

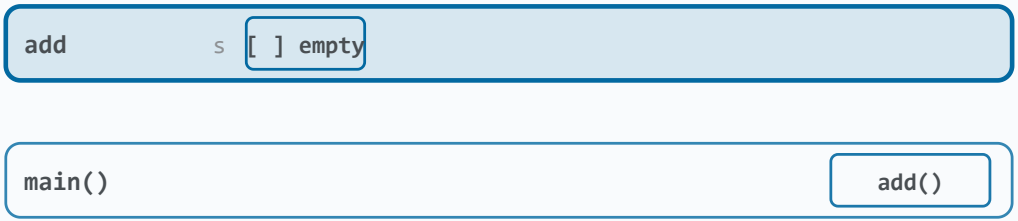
CALL STACK

`main()`

OUTPUT

```
6
```

Empty Call: `add()`



s is an empty array: s.length = 0.

Step 2 / 6

```
▶ static int add(int... s) {
2     int t = 0;
3     for (int v : s) t += v;
4     return t;
5 }
6 // in main:
7 System.out.println(add(1, 2, 3));
8 System.out.println(add());
```

VARIABLES

`s = []`

CALL STACK

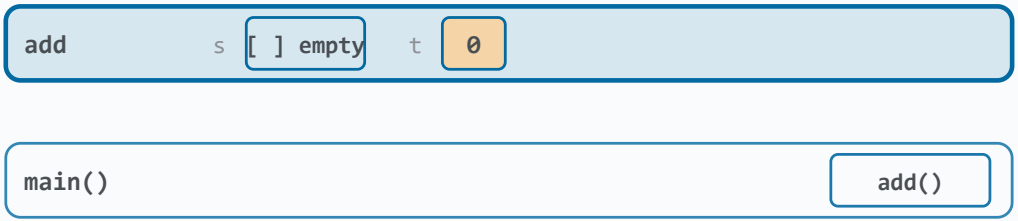
`add()`

`main()`

OUTPUT

```
6
```

Empty Call: `add()`



t = 0.

Step 3 / 6

```
1 static int add(int... s) {
▶     int t = 0;
3     for (int v : s) t += v;
4     return t;
5 }
6 // in main:
7 System.out.println(add(1, 2, 3));
8 System.out.println(add());
```

VARIABLES

`s = []` `t = 0`

CALL STACK

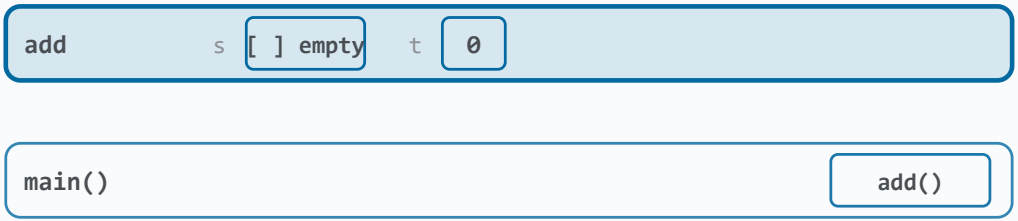
`add()`

`main()`

OUTPUT

```
6
```

Empty Call: `add()`



The array is empty: the body **never runs**, t stays 0.

Step 4 / 6

```
1 static int add(int... s) {
2     int t = 0;
▶     for (int v : s) t += v;
4     return t;
5 }
6 // in main:
7 System.out.println(add(1, 2, 3));
8 System.out.println(add());
```

VARIABLES

`s = []` `t = 0`

CALL STACK

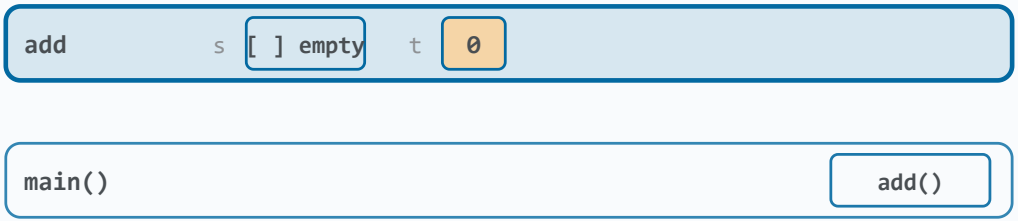
`add()`

`main()`

OUTPUT

```
6
```

Empty Call: `add()`



return t → returns 0.


```
1 static int add(int... s) {
2     int t = 0;
3     for (int v : s) t += v;
  ▶   return t;
5 }
6 // in main:
7 System.out.println(add(1, 2, 3));
8 System.out.println(add());
```

VARIABLES

`s = []` `t = 0`

CALL STACK

`add()`

`main()`

OUTPUT

```
6
```

Empty Call: `add()`

`main()`

0 is printed. An empty call is not an error; the result is 0, the identity for addition.

Step 6 / 6

```
1 static int add(int... s) {
2     int t = 0;
3     for (int v : s) t += v;
4     return t;
5 }
6 // in main:
7 System.out.println(add(1, 2, 3));
▶ System.out.println(add());
```

VARIABLES

—

CALL STACK

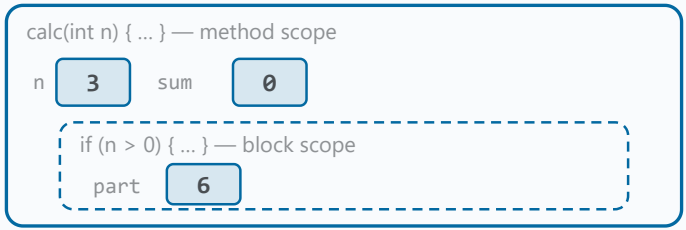
`main()`

OUTPUT

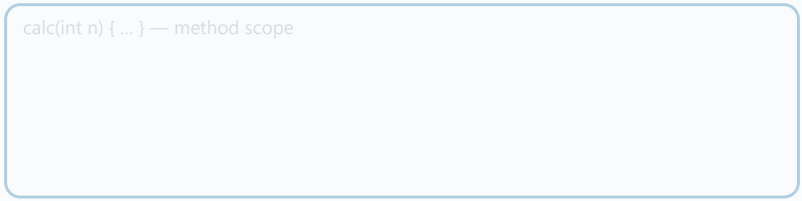
```
6
0
```

Local Scope

- Parameters and variables declared inside a method are **local**.
- A variable declared in a block (`{ }`) is visible only in that block and blocks nested in it.
- Two local variables with the same name cannot be declared in the same scope.
- Another method's local variables are not accessible.



Scope Example: `calc(3)`



main calls `calc(3)`.

Step 1 / 9

```
1 static int calc(int n) {
2     int sum = 0;
3     if (n > 0) {
4         int part = n * 2;
5         sum += part;
6     }
7     return sum;
8 }
▶ System.out.println(calc(3));  // main
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

Scope Example: `calc(3)`



The method scope opens: parameter **n = 3**.

Step 2 / 9

```
▶ static int calc(int n) {
2     int sum = 0;
3     if (n > 0) {
4         int part = n * 2;
5         sum += part;
6     }
7     return sum;
8 }
9 System.out.println(calc(3));  // main
```

VARIABLES

`n = 3`

CALL STACK

`calc(3)`

`main()`

OUTPUT

Scope Example: `calc(3)`



`sum` is in method scope: visible until the end of the method.

Step 3 / 9

```
1 static int calc(int n) {
▶     int sum = 0;
3     if (n > 0) {
4         int part = n * 2;
5         sum += part;
6     }
7     return sum;
8 }
9 System.out.println(calc(3));  // main
```

VARIABLES

`n = 3` `sum = 0`

CALL STACK

`calc(3)`

`main()`

OUTPUT

Scope Example: `calc(3)`



3 > 0 → **true**: enter the if block; a new inner scope opens.

Step 4 / 9

```
1 static int calc(int n) {
2     int sum = 0;
▶     if (n > 0) {
4         int part = n * 2;
5         sum += part;
6     }
7     return sum;
8 }
9 System.out.println(calc(3));  // main
```

VARIABLES

`n = 3` `sum = 0`

CALL STACK

`calc(3)`

`main()`

OUTPUT

Scope Example: `calc(3)`



part = 3 × 2 = 6: exists only inside the if block.

Step 5 / 9

```
1 static int calc(int n) {
2     int sum = 0;
3     if (n > 0) {
▶         int part = n * 2;
5         sum += part;
6     }
7     return sum;
8 }
9 System.out.println(calc(3));  // main
```

VARIABLES

`n = 3` `sum = 0` `part = 6`

CALL STACK

`calc(3)`

`main()`

OUTPUT

Scope Example: `calc(3)`



The inner block can reach `sum` in the outer scope: sum = 0 + 6 = **6**.

Step 6 / 9

```
1 static int calc(int n) {
2     int sum = 0;
3     if (n > 0) {
4         int part = n * 2;
▶         sum += part;
6     }
7     return sum;
8 }
9 System.out.println(calc(3));  // main
```

VARIABLES

`n = 3` `sum = 6` `part = 6`

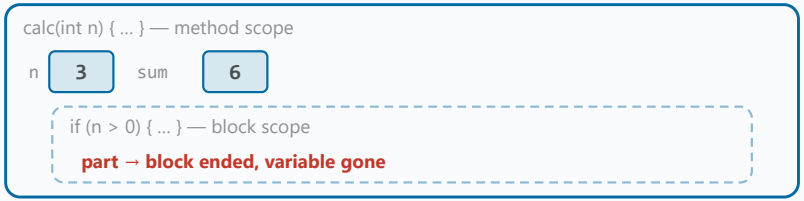
CALL STACK

`calc(3)`

`main()`

OUTPUT

Scope Example: `calc(3)`



The block closed: **part is gone**. Writing `part` here would be a compile error.

Step 7 / 9

```
1 static int calc(int n) {
2     int sum = 0;
3     if (n > 0) {
4         int part = n * 2;
5         sum += part;
▶     }
7     return sum;
8 }
9 System.out.println(calc(3));  // main
```

VARIABLES

`n = 3` `sum = 6`

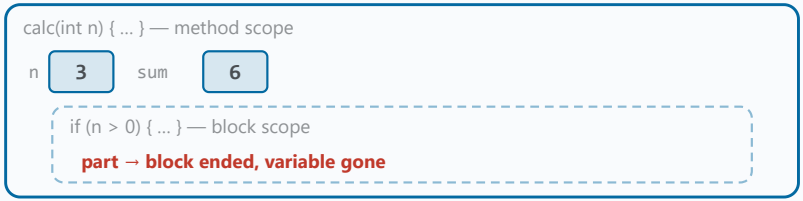
CALL STACK

`calc(3)`

`main()`

OUTPUT

Scope Example: `calc(3)`



return 6: the method scope closes too (`n` and `sum` are discarded).

Step 8 / 9

```
1 static int calc(int n) {
2     int sum = 0;
3     if (n > 0) {
4         int part = n * 2;
5         sum += part;
6     }
▶     return sum;
8 }
9 System.out.println(calc(3));  // main
```

VARIABLES

`n = 3` `sum = 6`

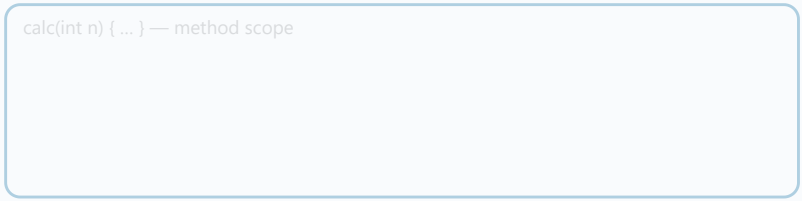
CALL STACK

`calc(3)`

`main()`

OUTPUT

Scope Example: `calc(3)`



The returned 6 becomes println's argument and is printed.

Step 9 / 9

```
1 static int calc(int n) {
2     int sum = 0;
3     if (n > 0) {
4         int part = n * 2;
5         sum += part;
6     }
7     return sum;
8 }
▶ System.out.println(calc(3));  // main
```

VARIABLES

—

CALL STACK

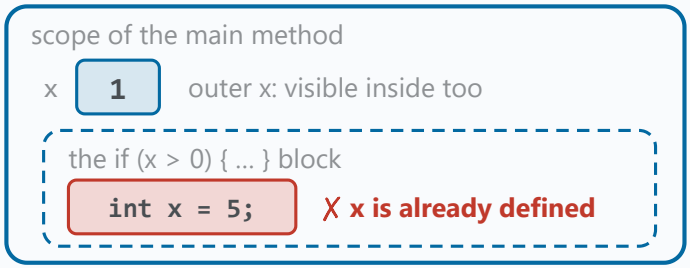
`main()`

OUTPUT

```
6
```

Scope: Variables with the Same Name

```
int x = 1;
if (x > 0) {
    int x = 5;    // error
}
```




```
error: variable x is already defined in method main(String[])
```

- An inner block cannot reuse the name of a local variable from the outer scope; a local does **not shadow** another local.
- Parameters are local too: `int a = 3;` inside `add(int a, int b)` gives the same error.
- A variable is gone when its block ends: after `for (int i …) { }` you may write `int i = 7;`.
- A **local** variable with the same name as a static field compiles; the local one is visible and the field is unchanged.

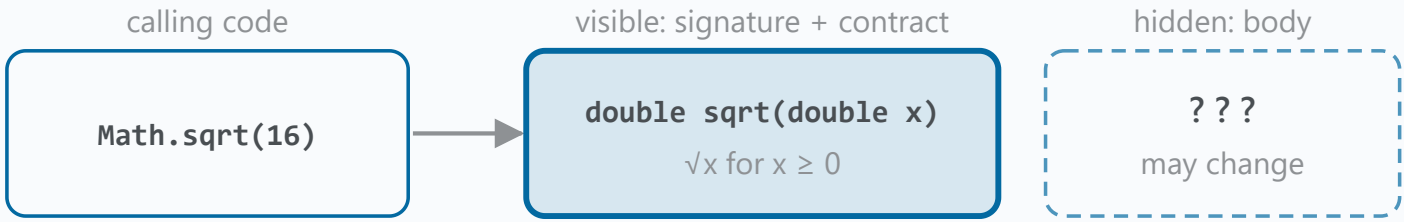
Common Library Methods: Math

Call	Result	Return type	Note
`Math.abs(-7)`	7	int	absolute value (overloaded: int, long, double…)
`Math.max(3, 9)`	9	int	returns the larger one
`Math.sqrt(16)`	4.0	double	square root; always a double
`Math.pow(2, 10)`	1024.0	double	power; needs (int) to assign to an int
`Math.round(2.5)`	3	long	rounds to the nearest integer

- Check the documentation for parameters, return type and special cases (e.g. `Math.sqrt(-1)` → NaN).

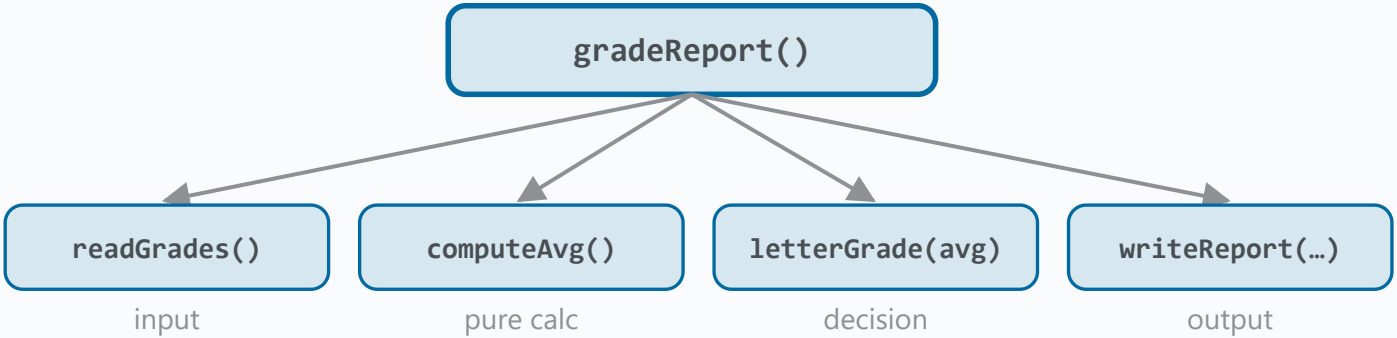
DESIGN

Abstraction



- A method hides implementation details from the calling code.
- A well-named method says **what** it does; **how** stays in the body.
- As long as the signature stays the same, callers are unaffected by changes to the body.

Decomposition



- A large task is split into subtasks, each with one responsibility.
- For each subtask, define its input, output and error behavior.
- A top-level method combines them in a sensible order; it reads like a table of contents.

Good Method Design

Principle	Question
Single task	Can the method be described in one sentence?
Meaningful name	Does the name state the result or the action?
Few parameters	Are all the inputs necessary?
Clear result	Are the return value and side effects clear?
Testability	Can the method be tested on its own?

EXAMPLE

Contract First

- The method name should state clearly what it does.
- Define the valid range of each parameter.
- Know what the return value means and how errors behave.

```
// Task  : returns the number of decimal digits.
// Input : any int (may be negative).
// Output: 1..10; 1 for 0, 3 for -507.
static int digitCount(int num) { … }
```



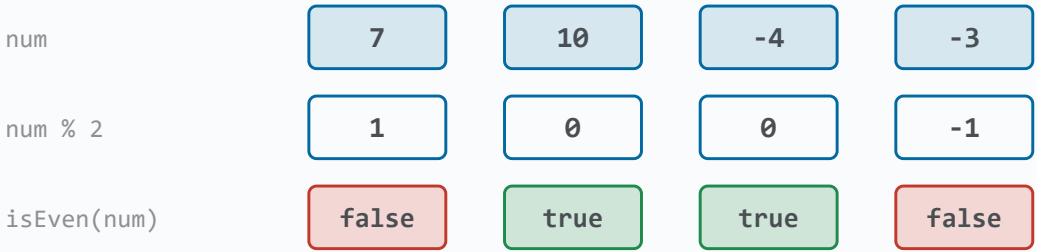
example: `0 → 1` `-507 → 3` `2147483647 → 10` `-2147483648 → 10`

EXAMPLE

Even / Odd

```
static boolean isEven(int num) {
    return num % 2 == 0;
}
```

- A boolean result can be used directly in decisions.
- The method returns the result instead of printing it.



In Java the remainder of a negative number can be negative (-3 % 2 = -1), so we test == 0.

EXAMPLES · STEP THROUGH THE CODE

Code Trace: `isEven(7)`

`main()` `if (isEven(7))`

To evaluate the if condition, `isEven(7)` is called first.

Step 1 / 5

```
1 static boolean isEven(int num) {
2     return num % 2 == 0;
3 }
4 // in main:
▶ if (isEven(7)) System.out.println("even");
6 else System.out.println("odd");
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

EXAMPLES · STEP THROUGH THE CODE

Code Trace: `isEven(7)`



Frame opens: num = 7.

Step 2 / 5

```
▶ static boolean isEven(int num) {
2     return num % 2 == 0;
3 }
4 // in main:
5 if (isEven(7)) System.out.println("even");
6 else System.out.println("odd");
```

VARIABLES

num = 7

CALL STACK

isEven(7)

main()

OUTPUT

EXAMPLES · STEP THROUGH THE CODE

Code Trace: `isEven(7)`



7 % 2 = 1 → 1 == 0 → **false**. return false.

Step 3 / 5

```
1 static boolean isEven(int num) {
▶     return num % 2 == 0;
3 }
4 // in main:
5 if (isEven(7)) System.out.println("even");
6 else System.out.println("odd");
```

VARIABLES

`num = 7`

CALL STACK

`isEven(7)`

`main()`

OUTPUT

Code Trace: `isEven(7)`

`main()` `if (false)`

The return value replaces the call: `if (false)` → the then branch is skipped.

Step 4 / 5

```
1 static boolean isEven(int num) {
2     return num % 2 == 0;
3 }
4 // in main:
▶ if (isEven(7)) System.out.println("even");
6 else System.out.println("odd");
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

Code Trace: `isEven(7)`

`main()`

The else branch prints **odd**. The method only computed; the caller decides what to print.

Step 5 / 5

```
1 static boolean isEven(int num) {
2     return num % 2 == 0;
3 }
4 // in main:
5 if (isEven(7)) System.out.println("even");
▶ else System.out.println("odd");
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

```
odd
```

EXAMPLE

Counting Digits

```
static int digitCount(int num) {
    long rest = Math.abs((long) num);
    if (rest == 0) return 1;
    int count = 0;
    while (rest > 0) {
        count++;
        rest /= 10;
    }
    return count;
}
System.out.println(digitCount(-507));
```

- `(long)` converts first: `Math.abs(-2147483648)` would stay negative as an int.
- Early return for 0: 1 digit.
- The result is **returned**, not printed; the caller can use it freely.

Code Trace: `digitCount(-507)`



For println's argument, `digitCount(-507)` is called first.

Step 1 / 11

```
1  static int digitCount(int num) {
2      long rest = Math.abs((long) num);
3      if (rest == 0) return 1;
4      int count = 0;
5      while (rest > 0) {
6          count++;
7          rest /= 10;
8      }
9      return count;
10 }
▶  System.out.println(digitCount(-507));
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

Code Trace: `digitCount(−507)`



Frame opens: num = −507.

Step 2 / 11

```
▶ static int digitCount(int num) {
2     long rest = Math.abs((long) num);
3     if (rest == 0) return 1;
4     int count = 0;
5     while (rest > 0) {
6         count++;
7         rest /= 10;
8     }
9     return count;
10 }
11 System.out.println(digitCount(-507));
```

VARIABLES

`num = −507`

CALL STACK

`digitCount(−507)`

`main()`

OUTPUT

Code Trace: `digitCount(−507)`



Math.abs((long) −507) → **rest = 507**. (A library method is a call too: it returns a value.)

Step 3 / 11

```
1  static int digitCount(int num) {
▶      long rest = Math.abs((long) num);
3      if (rest == 0) return 1;
4      int count = 0;
5      while (rest > 0) {
6          count++;
7          rest /= 10;
8      }
9      return count;
10 }
11 System.out.println(digitCount(-507));
```

VARIABLES

`num = −507` `rest = 507`

CALL STACK

`digitCount(−507)`

`main()`

OUTPUT

Code Trace: `digitCount(-507)`



507 == 0 → false: no early return.

Step 4 / 11

```
1  static int digitCount(int num) {
2      long rest = Math.abs((long) num);
▶      if (rest == 0) return 1;
4      int count = 0;
5      while (rest > 0) {
6          count++;
7          rest /= 10;
8      }
9      return count;
10 }
11 System.out.println(digitCount(-507));
```

VARIABLES

`num = -507` `rest = 507`

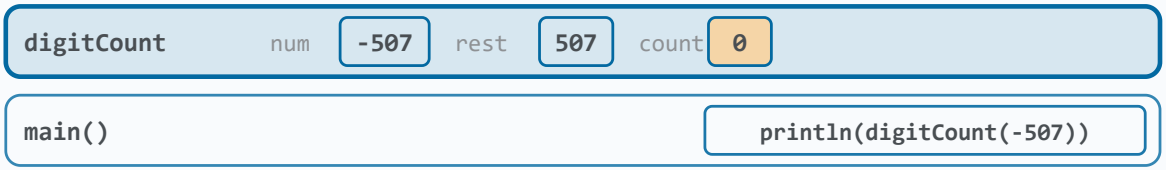
CALL STACK

`digitCount(-507)`

`main()`

OUTPUT

Code Trace: `digitCount(-507)`



Counter count = 0.

Step 5 / 11

```
1  static int digitCount(int num) {
2      long rest = Math.abs((long) num);
3      if (rest == 0) return 1;
▶      int count = 0;
5      while (rest > 0) {
6          count++;
7          rest /= 10;
8      }
9      return count;
10 }
11 System.out.println(digitCount(-507));
```

VARIABLES

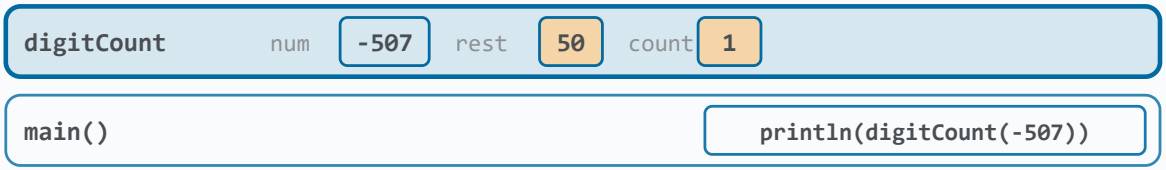
`num = -507` `rest = 507` `count = 0`

CALL STACK

- `digitCount(-507)`
- `main()`

OUTPUT

Code Trace: `digitCount(-507)`



Pass 1: 507 > 0 → count = 1, rest = 507 / 10 = 50.

Step 6 / 11

```
1  static int digitCount(int num) {
2      long rest = Math.abs((long) num);
3      if (rest == 0) return 1;
4      int count = 0;
▶      while (rest > 0) {
▶          count++;
▶          rest /= 10;
8      }
9      return count;
10 }
11 System.out.println(digitCount(-507));
```

VARIABLES

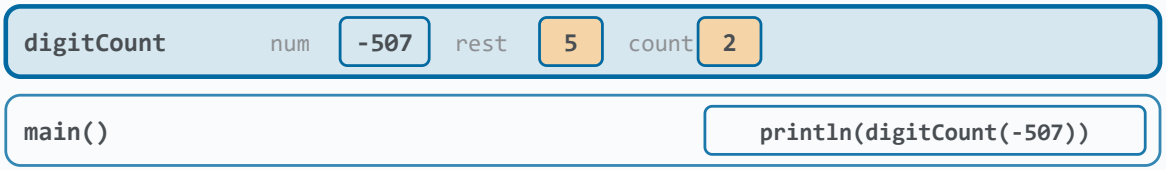
`num = -507` `rest = 50` `count = 1`

CALL STACK

- `digitCount(-507)`
- `main()`

OUTPUT

Code Trace: `digitCount(-507)`



Pass 2: 50 > 0 → count = 2, rest = 50 / 10 = 5.

Step 7 / 11

```
1  static int digitCount(int num) {
2      long rest = Math.abs((long) num);
3      if (rest == 0) return 1;
4      int count = 0;
▶      while (rest > 0) {
▶          count++;
▶          rest /= 10;
8      }
9      return count;
10 }
11 System.out.println(digitCount(-507));
```

VARIABLES

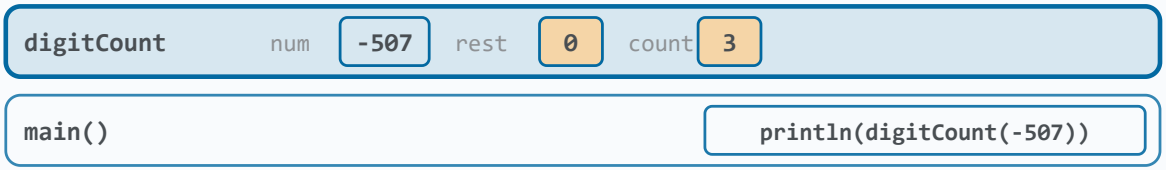
`num = -507` `rest = 5` `count = 2`

CALL STACK

- `digitCount(-507)`
- `main()`

OUTPUT

Code Trace: `digitCount(-507)`



Pass 3: 5 > 0 → count = 3, rest = 5 / 10 = 0.

Step 8 / 11

```
1  static int digitCount(int num) {
2      long rest = Math.abs((long) num);
3      if (rest == 0) return 1;
4      int count = 0;
▶      while (rest > 0) {
▶          count++;
▶          rest /= 10;
8      }
9      return count;
10 }
11 System.out.println(digitCount(-507));
```

VARIABLES

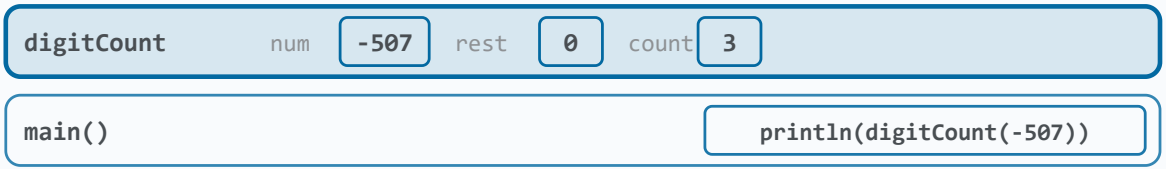
`num = -507` `rest = 0` `count = 3`

CALL STACK

- `digitCount(-507)`
- `main()`

OUTPUT

Code Trace: `digitCount(-507)`



0 > 0 → false: the loop ends.

Step 9 / 11

```
1  static int digitCount(int num) {
2      long rest = Math.abs((long) num);
3      if (rest == 0) return 1;
4      int count = 0;
▶      while (rest > 0) {
6          count++;
7          rest /= 10;
8      }
9      return count;
10 }
11 System.out.println(digitCount(-507));
```

VARIABLES

`num = -507` `rest = 0` `count = 3`

CALL STACK

- `digitCount(-507)`
- `main()`

OUTPUT

Code Trace: `digitCount(-507)`



return 3: the frame closes.

Step 10 / 11

```
1  static int digitCount(int num) {
2      long rest = Math.abs((long) num);
3      if (rest == 0) return 1;
4      int count = 0;
5      while (rest > 0) {
6          count++;
7          rest /= 10;
8      }
▶      return count;
10 }
11 System.out.println(digitCount(-507));
```

VARIABLES

`num = -507` `rest = 0` `count = 3`

CALL STACK

- `digitCount(-507)`
- `main()`

OUTPUT

Code Trace: `digitCount(-507)`

`main()` `println(3)`

3 replaces the call: `println(3)` → **3** is printed.

Step 11 / 11

```
1  static int digitCount(int num) {
2      long rest = Math.abs((long) num);
3      if (rest == 0) return 1;
4      int count = 0;
5      while (rest > 0) {
6          count++;
7          rest /= 10;
8      }
9      return count;
10 }
▶  System.out.println(digitCount(-507));
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

```
3
```

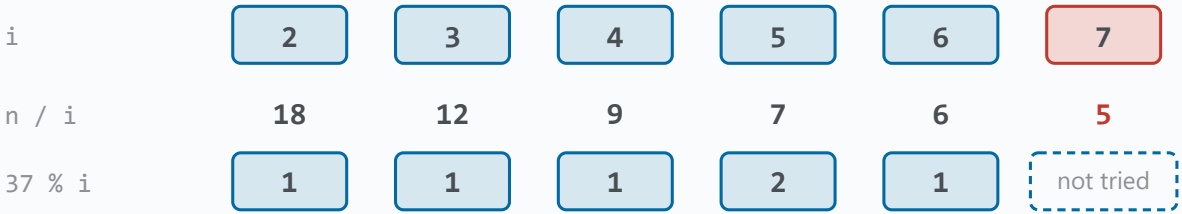
EXAMPLE

Is It Prime?

```
static boolean isPrime(int n) {
    if (n < 2) return false;
    for (int i = 2; i <= n / i; i++)
        if (n % i == 0) return false;
    return true;
}
```

- Return early as soon as the first divisor is found.

isPrime(37): only i with i ≤ n / i are tried (√37 ≈ 6.08)



No remainder is 0; for i = 7, 7 > 37 / 7 = 5 → the loop ends → return true

Code Trace: `isPrime(15)` and `isPrime(7)`

`main()` `println(isPrime(15))`

`isPrime(15)` is called.

Step 1 / 11

```
1  static boolean isPrime(int n) {
2      if (n < 2) return false;
3      for (int i = 2; i <= n / i; i++)
4          if (n % i == 0) return false;
5      return true;
6  }
▶ System.out.println(isPrime(15));   // main
8  System.out.println(isPrime(7));    // main
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

Code Trace: `isPrime(15)` and `isPrime(7)`



n = 15: 15 < 2 → false.

Step 2 / 11

```
1 static boolean isPrime(int n) {
▶       if (n < 2) return false;
3       for (int i = 2; i <= n / i; i++)
4           if (n % i == 0) return false;
5       return true;
6   }
7   System.out.println(isPrime(15));   // main
8   System.out.println(isPrime(7));    // main
```

VARIABLES

`n = 15`

CALL STACK

`isPrime(15)`

`main()`

OUTPUT

Code Trace: `isPrime(15)` and `isPrime(7)`



2 <= 15/2 = 7 → true; 15 % 2 = 1 → not a divisor, i++.

Step 3 / 11

```
1 static boolean isPrime(int n) {
2     if (n < 2) return false;
▶     for (int i = 2; i <= n / i; i++)
▶         if (n % i == 0) return false;
5     return true;
6 }
7 System.out.println(isPrime(15));   // main
8 System.out.println(isPrime(7));    // main
```

VARIABLES

`n = 15` `i = 2`

CALL STACK

`isPrime(15)`

`main()`

OUTPUT

Code Trace: `isPrime(15)` and `isPrime(7)`



3 <= 5 → true; 15 % 3 = 0 → **return false**: the remaining passes never run (early return).

Step 4 / 11

```
1 static boolean isPrime(int n) {
2     if (n < 2) return false;
▶     for (int i = 2; i <= n / i; i++)
▶         if (n % i == 0) return false;
5     return true;
6 }
7 System.out.println(isPrime(15));   // main
8 System.out.println(isPrime(7));    // main
```

VARIABLES

`n = 15` `i = 3`

CALL STACK

`isPrime(15)`

`main()`

OUTPUT

Code Trace: `isPrime(15)` and `isPrime(7)`



The returned **false** goes to println and is printed.

Step 5 / 11

```
1 static boolean isPrime(int n) {
2     if (n < 2) return false;
3     for (int i = 2; i <= n / i; i++)
4         if (n % i == 0) return false;
5     return true;
6 }
▶ System.out.println(isPrime(15));  // main
8 System.out.println(isPrime(7));   // main
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

```
false
```

Code Trace: `isPrime(15)` and `isPrime(7)`

`main()` `println(isPrime(7))`

`isPrime(7)` is called.

Step 6 / 11

```
1 static boolean isPrime(int n) {
2     if (n < 2) return false;
3     for (int i = 2; i <= n / i; i++)
4         if (n % i == 0) return false;
5     return true;
6 }
7 System.out.println(isPrime(15));   // main
▶ System.out.println(isPrime(7));    // main
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

```
false
```

Code Trace: `isPrime(15)` and `isPrime(7)`



n = 7: 7 < 2 → false.

Step 7 / 11

```
1 static boolean isPrime(int n) {
▶       if (n < 2) return false;
3       for (int i = 2; i <= n / i; i++)
4           if (n % i == 0) return false;
5       return true;
6   }
7   System.out.println(isPrime(15));   // main
8   System.out.println(isPrime(7));    // main
```

VARIABLES

`n = 7`

CALL STACK

`isPrime(7)`

`main()`

OUTPUT

```
false
```

Code Trace: `isPrime(15)` and `isPrime(7)`



2 <= 7/2 = 3 → true; 7 % 2 = 1 → not a divisor, i++.

Step 8 / 11

```
1  static boolean isPrime(int n) {
2      if (n < 2) return false;
▶      for (int i = 2; i <= n / i; i++)
▶          if (n % i == 0) return false;
5      return true;
6  }
7  System.out.println(isPrime(15));   // main
8  System.out.println(isPrime(7));    // main
```

VARIABLES

`n = 7` `i = 2`

CALL STACK

`isPrime(7)`

`main()`

OUTPUT

```
false
```

Code Trace: `isPrime(15)` and `isPrime(7)`



3 <= 7/3 = 2 → **false**: no divisor up to √7.

Step 9 / 11

```
1 static boolean isPrime(int n) {
2     if (n < 2) return false;
▶     for (int i = 2; i <= n / i; i++)
4         if (n % i == 0) return false;
5     return true;
6 }
7 System.out.println(isPrime(15));  // main
8 System.out.println(isPrime(7));   // main
```

VARIABLES

`n = 7` `i = 3`

CALL STACK

`isPrime(7)`

`main()`

OUTPUT

```
false
```

Code Trace: `isPrime(15)` and `isPrime(7)`



The loop finished without an early return → **return true**.

Step 10 / 11

```
1 static boolean isPrime(int n) {
2     if (n < 2) return false;
3     for (int i = 2; i <= n / i; i++)
4         if (n % i == 0) return false;
▶       return true;
6 }
7 System.out.println(isPrime(15));  // main
8 System.out.println(isPrime(7));   // main
```

VARIABLES

`n = 7`

CALL STACK

`isPrime(7)`

`main()`

OUTPUT

```
false
```

Code Trace: `isPrime(15)` and `isPrime(7)`



The returned **true** goes to println and is printed.

Step 11 / 11

```
1 static boolean isPrime(int n) {
2     if (n < 2) return false;
3     for (int i = 2; i <= n / i; i++)
4         if (n % i == 0) return false;
5     return true;
6 }
7 System.out.println(isPrime(15));  // main
▶ System.out.println(isPrime(7));   // main
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

```
false
true
```

gcd: Euclid's algorithm (48, 18)



Euclid's algorithm finds the greatest common divisor (GCD): gcd(a, b) = gcd(b, a % b); when b = 0 the answer is a.

Step 1 / 11

```
1 static int gcd(int a, int b) {
2     while (b != 0) {
3         int r = a % b;
4         a = b;
5         b = r;
6     }
7     return Math.abs(a);
8 }
▶ System.out.println(gcd(48, 18));  // main
```

VARIABLES

—

CALL STACK

main()

OUTPUT

gcd: Euclid's algorithm (48, 18)



The frame opens: a = 48, b = 18.

Step 2 / 11

```
▶ static int gcd(int a, int b) {
2     while (b != 0) {
3         int r = a % b;
4         a = b;
5         b = r;
6     }
7     return Math.abs(a);
8 }
9 System.out.println(gcd(48, 18));  // main
```

VARIABLES

a = 48 b = 18

CALL STACK

gcd(48, 18)

main()

OUTPUT

gcd: Euclid's algorithm (48, 18)



Condition: 18 != 0 → **true**.


```
1 static int gcd(int a, int b) {
▶       while (b != 0) {
3           int r = a % b;
4           a = b;
5           b = r;
6       }
7       return Math.abs(a);
8   }
9   System.out.println(gcd(48, 18));  // main
```

VARIABLES

a = 48 b = 18

CALL STACK

gcd(48, 18)

main()

OUTPUT

gcd: Euclid's algorithm (48, 18)



r = 48 % 18 = 12; a = 18, b = 12. (gcd(48, 18) = gcd(18, 12))

Step 4 / 11

```
1 static int gcd(int a, int b) {
2     while (b != 0) {
          int r = a % b;
          a = b;
          b = r;
6     }
7     return Math.abs(a);
8 }
9 System.out.println(gcd(48, 18));  // main
```

VARIABLES

a = 18 b = 12 r = 12

CALL STACK

gcd(48, 18)

main()

OUTPUT

gcd: Euclid's algorithm (48, 18)



Condition: 12 != 0 → **true**.

Step 5 / 11

```
1 static int gcd(int a, int b) {
▶       while (b != 0) {
3           int r = a % b;
4           a = b;
5           b = r;
6       }
7       return Math.abs(a);
8   }
9   System.out.println(gcd(48, 18));  // main
```

VARIABLES

a = 18 b = 12

CALL STACK

gcd(48, 18)

main()

OUTPUT

gcd: Euclid's algorithm (48, 18)



r = 18 % 12 = 6; a = 12, b = 6. (gcd(18, 12) = gcd(12, 6))

Step 6 / 11

```
1 static int gcd(int a, int b) {
2     while (b != 0) {
          int r = a % b;
          a = b;
          b = r;
6     }
7     return Math.abs(a);
8 }
9 System.out.println(gcd(48, 18));  // main
```

VARIABLES

a = 12 b = 6 r = 6

CALL STACK

gcd(48, 18)

main()

OUTPUT

gcd: Euclid's algorithm (48, 18)



Condition: 6 != 0 → **true**.


```
1 static int gcd(int a, int b) {
▶       while (b != 0) {
3           int r = a % b;
4           a = b;
5           b = r;
6       }
7       return Math.abs(a);
8   }
9   System.out.println(gcd(48, 18));  // main
```

VARIABLES

a = **12** b = **6**

CALL STACK

gcd(48, 18)

main()

OUTPUT

gcd: Euclid's algorithm (48, 18)



r = 12 % 6 = 0; a = 6, b = 0. (gcd(12, 6) = gcd(6, 0))

Step 8 / 11

```
1 static int gcd(int a, int b) {
2     while (b != 0) {
          int r = a % b;
          a = b;
          b = r;
6     }
7     return Math.abs(a);
8 }
9 System.out.println(gcd(48, 18));  // main
```

VARIABLES

a = 6 b = 0 r = 0

CALL STACK

gcd(48, 18)

main()

OUTPUT

gcd: Euclid's algorithm (48, 18)



Condition: 0 != 0 → **false**.


```
1 static int gcd(int a, int b) {
▶       while (b != 0) {
3           int r = a % b;
4           a = b;
5           b = r;
6       }
7       return Math.abs(a);
8   }
9   System.out.println(gcd(48, 18));  // main
```

VARIABLES

a = 6 b = 0

CALL STACK

gcd(48, 18)

main()

OUTPUT

gcd: Euclid's algorithm (48, 18)



return Math.abs(6) → 6. Math.abs keeps the result positive even for negative input.

Step 10 / 11

```
1 static int gcd(int a, int b) {
2     while (b != 0) {
3         int r = a % b;
4         a = b;
5         b = r;
6     }
▶     return Math.abs(a);
8 }
9 System.out.println(gcd(48, 18));  // main
```

VARIABLES

a = 6 b = 0

CALL STACK

gcd(48, 18)

main()

OUTPUT

gcd: Euclid's algorithm (48, 18)



6 is printed: 48 = 6 × 8, 18 = 6 × 3. Only 3 passes, much faster than trying every number from 1 to 18.

Step 11 / 11

```
1 static int gcd(int a, int b) {
2     while (b != 0) {
3         int r = a % b;
4         a = b;
5         b = r;
6     }
7     return Math.abs(a);
8 }
▶ System.out.println(gcd(48, 18));  // main
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

```
6
```

EXAMPLES · STEP THROUGH THE CODE

lcm(12, 18): a method calling a method



main calls `lcm(12, 18)`.

Step 1 / 7

```
1 static int lcm(int a, int b) {
2     if (a == 0 || b == 0) return 0;
3     return Math.abs(a / gcd(a, b) * b);
4 }
5 // gcd: Euclid's algorithm
▶ System.out.println(lcm(12, 18));  // main
```

VARIABLES

—

CALL STACK

main()

OUTPUT

lcm(12, 18): a method calling a method



lcm frame: a = 12, b = 18.

Step 2 / 7

```
▶ static int lcm(int a, int b) {
2     if (a == 0 || b == 0) return 0;
3     return Math.abs(a / gcd(a, b) * b);
4 }
5 // gcd: Euclid's algorithm
6 System.out.println(lcm(12, 18));  // main
```

VARIABLES

a = 12 b = 18

CALL STACK

lcm(12, 18)

main()

OUTPUT

lcm(12, 18): a method calling a method



a and b are nonzero, so line 2 is passed. To evaluate the expression, lcm calls **gcd(a, b)**: 3 frames on the stack.

Step 3 / 7

```
1 static int lcm(int a, int b) {
2     if (a == 0 || b == 0) return 0;
▶     return Math.abs(a / gcd(a, b) * b);
4 }
5 // gcd: Euclid's algorithm
6 System.out.println(lcm(12, 18));  // main
```

VARIABLES

a = 12 b = 18

CALL STACK

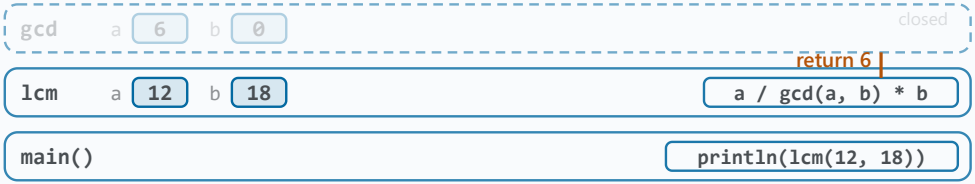
gcd(12, 18)

lcm(12, 18)

main()

OUTPUT

lcm(12, 18): a method calling a method



gcd runs its own loop: (12, 18) → (18, 12) → (12, 6) → (6, 0) → return **6**. The names match, but gcd's a and b are separate boxes.

Step 4 / 7

```
1 static int lcm(int a, int b) {
2     if (a == 0 || b == 0) return 0;
▶     return Math.abs(a / gcd(a, b) * b);
4 }
5 // gcd: Euclid's algorithm
6 System.out.println(lcm(12, 18));  // main
```

VARIABLES

a = 6 b = 0

CALL STACK

- gcd(12, 18)
- lcm(12, 18)
- main()

OUTPUT

lcm(12, 18): a method calling a method



6 replaces the call: 12 / 6 * 18 = 2 * 18 = **36**. Dividing first keeps a * b from overflowing.

Step 5 / 7

```
1 static int lcm(int a, int b) {
2     if (a == 0 || b == 0) return 0;
▶     return Math.abs(a / gcd(a, b) * b);
4 }
5 // gcd: Euclid's algorithm
6 System.out.println(lcm(12, 18));  // main
```

VARIABLES

a = 12 b = 18

CALL STACK

lcm(12, 18)

main()

OUTPUT

lcm(12, 18): a method calling a method



return 36: lcm's frame closes.

Step 6 / 7

```
1 static int lcm(int a, int b) {
2     if (a == 0 || b == 0) return 0;
▶     return Math.abs(a / gcd(a, b) * b);
4 }
5 // gcd: Euclid's algorithm
6 System.out.println(lcm(12, 18));  // main
```

VARIABLES

a = 12 b = 18

CALL STACK

lcm(12, 18)

main()

OUTPUT

lcm(12, 18): a method calling a method

main() println(36)

36 is printed. Calling another method keeps the code short: no duplication.

Step 7 / 7

```
1 static int lcm(int a, int b) {
2     if (a == 0 || b == 0) return 0;
3     return Math.abs(a / gcd(a, b) * b);
4 }
5 // gcd: Euclid's algorithm
▶ System.out.println(lcm(12, 18));  // main
```

VARIABLES

—

CALL STACK

main()

OUTPUT

```
36
```

EXAMPLES · STEP THROUGH THE CODE

Reuse: `max3(4, 9, 7)`



main wants the largest of three numbers.

Step 1 / 9

```
1 static int max(int a, int b) {
2     if (a > b) return a;
3     return b;
4 }
5 static int max3(int a, int b, int c) {
6     return max(max(a, b), c);
7 }
▶ System.out.println(max3(4, 9, 7));  // main
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

Reuse: `max3(4, 9, 7)`



max3 frame: a = 4, b = 9, c = 7.

Step 2 / 9

```
1 static int max(int a, int b) {
2     if (a > b) return a;
3     return b;
4 }
▶ static int max3(int a, int b, int c) {
6     return max(max(a, b), c);
7 }
8 System.out.println(max3(4, 9, 7));  // main
```

VARIABLES

a = 4 b = 9 c = 7

CALL STACK

- max3(4, 9, 7)
- main()

OUTPUT

Reuse: `max3(4, 9, 7)`



In a nested call the **inner** one is evaluated first: `max(4, 9)`.

Step 3 / 9

```
1 static int max(int a, int b) {
2     if (a > b) return a;
3     return b;
4 }
5 static int max3(int a, int b, int c) {
▶     return max(max(a, b), c);
7 }
8 System.out.println(max3(4, 9, 7));  // main
```

VARIABLES

`a = 4` `b = 9` `c = 7`

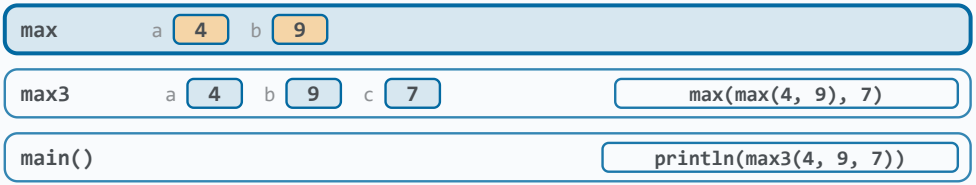
CALL STACK

`max3(4, 9, 7)`

`main()`

OUTPUT

Reuse: `max3(4, 9, 7)`



New frame (a = 4, b = 9): 4 > 9 → false.

Step 4 / 9

```
▶ static int max(int a, int b) {
▶     if (a > b) return a;
3     return b;
4 }
5 static int max3(int a, int b, int c) {
6     return max(max(a, b), c);
7 }
8 System.out.println(max3(4, 9, 7));  // main
```

VARIABLES

`a = 4` `b = 9`

CALL STACK

- `max(4, 9)`
- `max3(4, 9, 7)`
- `main()`

OUTPUT

EXAMPLES · STEP THROUGH THE CODE

Reuse: `max3(4, 9, 7)`



return b → 9.

Step 5 / 9

```
1 static int max(int a, int b) {
2     if (a > b) return a;
▶     return b;
4 }
5 static int max3(int a, int b, int c) {
6     return max(max(a, b), c);
7 }
8 System.out.println(max3(4, 9, 7));  // main
```

VARIABLES

a = 4 b = 9

CALL STACK

- max(4, 9)
- max3(4, 9, 7)
- main()

OUTPUT

Reuse: `max3(4, 9, 7)`



The expression simplifies to `max(9, 7)`. The same method is called a second time.

Step 6 / 9

```
1 static int max(int a, int b) {
2     if (a > b) return a;
3     return b;
4 }
5 static int max3(int a, int b, int c) {
▶     return max(max(a, b), c);
7 }
8 System.out.println(max3(4, 9, 7));  // main
```

VARIABLES

a = 4 | b = 9 | c = 7

CALL STACK

- max3(4, 9, 7)
- main()

OUTPUT

Reuse: `max3(4, 9, 7)`



New frame (a = 9, b = 7): 9 > 7 → true → **return a** → 9.

Step 7 / 9

```
1 static int max(int a, int b) {
▶     if (a > b) return a;
3     return b;
4 }
5 static int max3(int a, int b, int c) {
6     return max(max(a, b), c);
7 }
8 System.out.println(max3(4, 9, 7));  // main
```

VARIABLES

a = 9 b = 7

CALL STACK

- max(9, 7)
- max3(4, 9, 7)
- main()

OUTPUT

Reuse: `max3(4, 9, 7)`



max3 returns 9 as well.

Step 8 / 9

```
1 static int max(int a, int b) {
2     if (a > b) return a;
3     return b;
4 }
5 static int max3(int a, int b, int c) {
▶     return max(max(a, b), c);
7 }
8 System.out.println(max3(4, 9, 7));  // main
```

VARIABLES

`a = 4` `b = 9` `c = 7`

CALL STACK

- `max3(4, 9, 7)`
- `main()`

OUTPUT

Reuse: `max3(4, 9, 7)`



9 is printed. The comparison logic lives in one place (max), so a bug is fixed once.

Step 9 / 9

```
1 static int max(int a, int b) {
2     if (a > b) return a;
3     return b;
4 }
5 static int max3(int a, int b, int c) {
6     return max(max(a, b), c);
7 }
▶ System.out.println(max3(4, 9, 7));  // main
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

```
9
```

Methods returning double: temperature conversion



`fahrenheit(25)` : the int argument 25 is widened to **25.0** for the double parameter.

Step 1 / 8

```
1 static double fahrenheit(double c) {
2     return c * 9 / 5 + 32;
3 }
4 static double celsius(double f) {
5     return (f - 32) * 5 / 9;
6 }
▶ double f = fahrenheit(25);          // main
8 System.out.println(f);
9 System.out.println(celsius(f));
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

Methods returning double: temperature conversion



c = 25.0.

Step 2 / 8

```
▶ static double fahrenheit(double c) {
2     return c * 9 / 5 + 32;
3 }
4 static double celsius(double f) {
5     return (f - 32) * 5 / 9;
6 }
7 double f = fahrenheit(25);          // main
8 System.out.println(f);
9 System.out.println(celsius(f));
```

VARIABLES

c = 25.0

CALL STACK

fahrenheit(25.0)

main()

OUTPUT

Methods returning double: temperature conversion



25.0 × 9 = 225.0; / 5 = 45.0; + 32 = **77.0** → return.

Step 3 / 8

```
1 static double fahrenheit(double c) {
▶     return c * 9 / 5 + 32;
3 }
4 static double celsius(double f) {
5     return (f - 32) * 5 / 9;
6 }
7 double f = fahrenheit(25);          // main
8 System.out.println(f);
9 System.out.println(celsius(f));
```

VARIABLES

`c = 25.0`

CALL STACK

`fahrenheit(25.0)`

`main()`

OUTPUT

Methods returning double: temperature conversion



f = 77.0 (double).

Step 4 / 8

```
1 static double fahrenheit(double c) {
2     return c * 9 / 5 + 32;
3 }
4 static double celsius(double f) {
5     return (f - 32) * 5 / 9;
6 }
▶ double f = fahrenheit(25);          // main
8 System.out.println(f);
9 System.out.println(celsius(f));
```

VARIABLES

f = 77.0

CALL STACK

main()

OUTPUT

Methods returning double: temperature conversion



77.0 is printed (with .0, since it is a double).

Step 5 / 8

```
1 static double fahrenheit(double c) {
2     return c * 9 / 5 + 32;
3 }
4 static double celsius(double f) {
5     return (f - 32) * 5 / 9;
6 }
7 double f = fahrenheit(25);          // main
▶ System.out.println(f);
9 System.out.println(celsius(f));
```

VARIABLES

`f = 77.0`

CALL STACK

`main()`

OUTPUT

```
77.0
```

Methods returning double: temperature conversion



`celsius(f)` : the value of f (77.0) is copied.

Step 6 / 8

```
1 static double fahrenheit(double c) {
2     return c * 9 / 5 + 32;
3 }
4 static double celsius(double f) {
5     return (f - 32) * 5 / 9;
6 }
7 double f = fahrenheit(25);          // main
8 System.out.println(f);
▶ System.out.println(celsius(f));
```

VARIABLES

`f = 77.0`

CALL STACK

`celsius(77.0)`

`main()`

OUTPUT

```
77.0
```

Methods returning double: temperature conversion



(77.0 − 32) × 5 / 9 = 45.0 × 5 / 9 = **25.0**. With `5 / 9 * (f - 32)`, integer division would make 5 / 9 equal **0**!

Step 7 / 8

```
1 static double fahrenheit(double c) {
2     return c * 9 / 5 + 32;
3 }
4 static double celsius(double f) {
▶     return (f - 32) * 5 / 9;
6 }
7 double f = fahrenheit(25);          // main
8 System.out.println(f);
9 System.out.println(celsius(f));
```

VARIABLES

`f = 77.0`

CALL STACK

`celsius(77.0)`

`main()`

OUTPUT

```
77.0
```

Methods returning double: temperature conversion



25.0 is printed: the two methods are inverses (a nice check for testing).

Step 8 / 8

```
1 static double fahrenheit(double c) {
2     return c * 9 / 5 + 32;
3 }
4 static double celsius(double f) {
5     return (f - 32) * 5 / 9;
6 }
7 double f = fahrenheit(25);          // main
8 System.out.println(f);
▶ System.out.println(celsius(f));
```

VARIABLES

f = 77.0

CALL STACK

main()

OUTPUT

```
77.0
25.0
```

EXAMPLE

Pure Methods and Side Effects

- A method that returns the same result for the same input and changes no outside state is easier to test.
- Computing and printing can live in separate methods.
- When a side effect is needed, name it clearly and keep it contained.

Pure

```
static int square(int x) { return x * x; }
```

square(4) is always 16; usable anywhere.

Side effect

```
static void printSquare(int x) { System.out.println(x * x); }
```

Only prints; the result can't be used in another computation.

SUMMARY

Chapter Summary

- A method names a task, takes input through parameters and can return a result.
- Java always passes arguments by value.
- Overloading reuses one name with different parameter lists.
- Scope determines where a variable can be accessed.
- A method can call another (lcm → gcd, max3 → max); each call opens its own frame on the stack.
- A method that returns its result instead of printing it is easier to test and reuse.

SUMMARY

Review Questions

1. What is the difference between a parameter and an argument?
2. When is a void method appropriate?
3. Why doesn't `num` change after calling `inc(num)`?
4. Which part must differ in overloading? Why can't `int f(int)` and `double f(int)` coexist?
5. Why does `isEven` return a boolean instead of printing?

SUMMARY

Exercises

1. Write `static boolean isLeap(int year)` (the 4/100/400 rule) and test it with 1900, 2000, 2024.
2. Write `static long factorial(int n)` and use it in `static long choose(int n, int r)` = n! / (r! (n − r)!).
3. Write `static int reverse(int num)` and a `static boolean isPalindrome(int num)` that calls it.
4. `static double avg(int a, int b, int c)`: why can `(a + b + c) / 3` give a wrong result? Fix it.
5. Overloading: add `max(double, double)`; which one does `max(3, 4.5)` call?
6. Write a `main` that uses `isPrime` to list the twin primes in 1..100 (5–7, 11–13 …).

References

- Deitel and Deitel, Java: How to Program, Early Objects, 11th ed., Pearson, 2018.
- Liang, Introduction to Java Programming and Data Structures, 13th ed., Pearson, 2024.
- Oracle, The Java Tutorials: **Defining Methods**, **Passing Information to a Method**.
- Oracle, Java Language Specification SE 21: **Chapter 8, Classes** (§8.4 Method Declarations).
- Sedgewick and Wayne, Computer Science: An Interdisciplinary Approach, Addison-Wesley, 2016: **2.1 Static Methods**.
- Downey and Mayfield, Think Java, 2nd ed., O'Reilly, 2019 (**free online edition**).

The End

Chapter 9: Methods

Previous: Chapter 8: Loops · Next: Chapter 10: Arrays