Chapter 3.2: Methods

Object Oriented Programming with Java

Definition and calls | Parameters and return values | The call stack | Overloading | Built-in Java methods | Scope | Abstraction and design

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/java/slides/Chapter_03_02_Methods.html

INTRODUCTION

Learning Outcomes

Defining and calling

Write a method with an access modifier, return type, name and parameter list; call it with arguments and use its return value.

Tracing execution

Show step by step how frames are pushed and popped on the call stack, and how arguments are copied **by value**.

Overloading

Tell same-named methods apart by their parameter lists; predict which version the compiler picks from the argument types.

Built-in methods

`System.out` and `Math`: use their methods correctly; know the commonly used classes and packages of the Java library.

Scope

Explain the scope and lifetime of local variables, class fields and nested blocks.

Design

Design good methods using abstraction, generalization, DRY and single responsibility; break a problem into methods.

What Is a Method?

A function is a piece of code that does one job, has a **name**, and can be used again and again. In Java, functions always live inside a class and are called **methods**.

- A method is defined by the **parameters** it takes and its **return type**.
- Methods make code more readable and reduce repetition.
- They encourage modular, well-organized code.
- They split a program into small, manageable pieces.

Analogy: a recipe

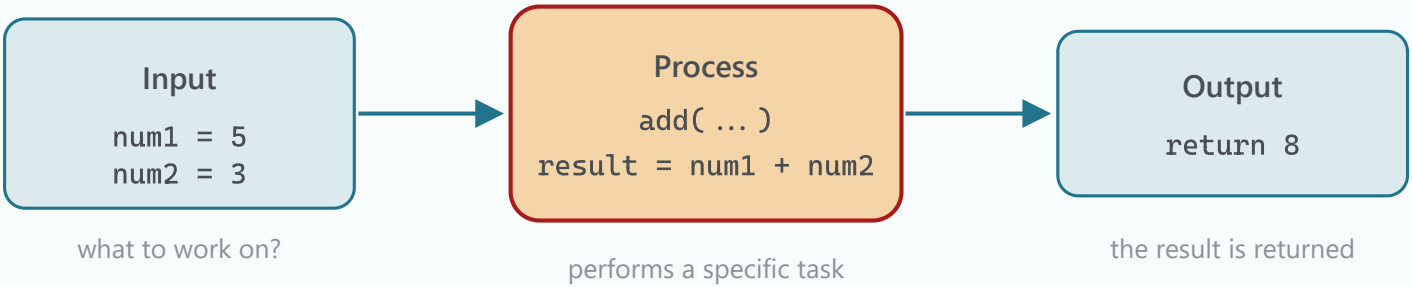
"Brew tea" is a recipe: written once, followed as often as you like. The ingredients are the **parameters**, and the tea you get is the **return value**. Whoever follows the recipe doesn't have to rethink every step.

The `main` we have been writing is also a method; `System.out.println(...)` is a call to a built-in **method**.

METHODS

Input – Process – Output

Every method can be seen in three stages:



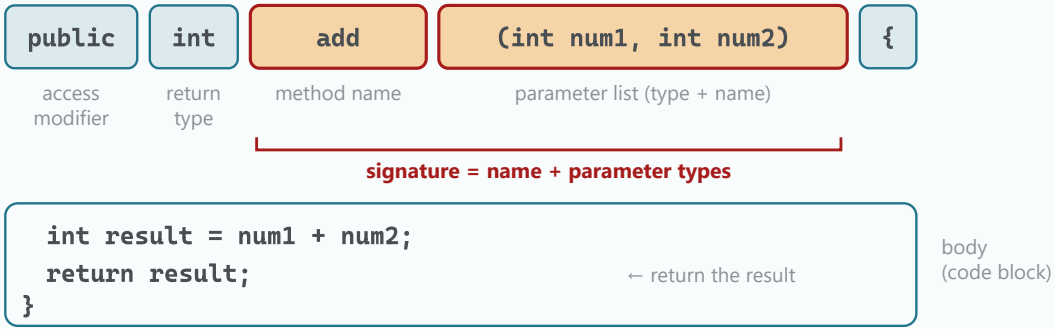
Input
The method receives data: **parameters**. They determine what the method works on.

Process
The method processes the input with the statements in its body and performs a specific task.

Output
The process produces a result, which `return` hands **back to the caller**.

Like f(x) = x^2 in math: x is the input, squaring is the process, x^2 is the output. Some methods have no input or no output (`void`).

Defining a Method



`public`	**Access modifier**: says who may call the method.
`int`	**Return type**: the type of the value the method gives back; if it returns nothing, `void`.
`add`	**Method name**: a verb or verb phrase starting with a lowercase letter (camelCase).
`(int num1, int num2)`	**Parameters**: the method's inputs; each has a type and a name.
`{ ... }`	**Code block** (body): the statements that run when the method is called.

The add method takes two input parameters, num1 and num2, adds them and returns the result.

Calling a Method

```java
public class AddExample {
    public static int add(int num1, int num2) {
        int result = num1 + num2;
        return result;
    }

    public static void main(String[] args) {
        int result = add(5, 3);
        System.out.println("Sum: " + result);
    }
}
```

OUTPUT

```
Sum: 8
```

- `add(5, 3)` is a **call**: control goes to the method and, when it finishes, comes back to where the call was made.
- In the call, `5` and `3` **are arguments**; in the definition, `num1`, `num2` are **parameters**.
- The call returns **8**; this value is assigned to the `result` variable.
- The number, order and types of the arguments must match the parameters.
- `static`: the method can be called directly from `main`, without creating an object.

Call and Return: Step by Step

```
1 static int add(int num1, int num2) {
2   int result = num1 + num2;
3   return result;
4 }
5
▶ public static void main(String[] args) {
7   int result = add(5, 3);
8   System.out.println("Sum: " + result);
9 }
```

Stack



A program always starts in its `main` method. The stack holds only the `main` frame.

Step 1 / 6

OUTPUT

VARIABLES

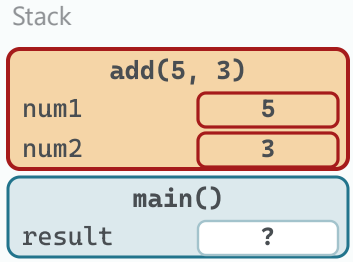
`args = []`

CALL STACK

`main()`

Call and Return: Step by Step

```
1 static int add(int num1, int num2) {
2   int result = num1 + num2;
3   return result;
4 }
5
6 public static void main(String[] args) {
▶   int result = add(5, 3);
8   System.out.println("Sum: " + result);
9 }
```



`add(5, 3)` is called: a new **frame** is pushed, and the arguments are **copied**.

Step 2 / 6

OUTPUT

VARIABLES

`num1 = 5` `num2 = 3`

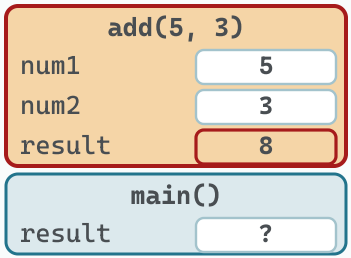
CALL STACK

- `add(5, 3)`
- `main()`

Call and Return: Step by Step

```
1 static int add(int num1, int num2) {
▶   int result = num1 + num2;
3   return result;
4 }
5
6 public static void main(String[] args) {
7   int result = add(5, 3);
8   System.out.println("Sum: " + result);
9 }
```

Stack



The body runs: `result = 5 + 3 = 8`. This `result` is add's **local** variable.

Step 3 / 6

OUTPUT

VARIABLES

`num1 = 5` `num2 = 3` `result = 8`

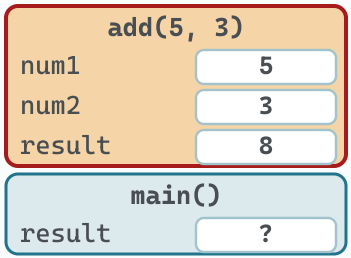
CALL STACK

- `add(5, 3)`
- `main()`

Call and Return: Step by Step

```
1 static int add(int num1, int num2) {
2   int result = num1 + num2;
▶   return result;
4 }
5
6 public static void main(String[] args) {
7   int result = add(5, 3);
8   System.out.println("Sum: " + result);
9 }
```

Stack



`return result;` sends 8 back to the caller and ends the method; the frame is **popped**.

Step 4 / 6

OUTPUT

VARIABLES

`num1 = 5` `num2 = 3` `result = 8`

`return = 8`

CALL STACK

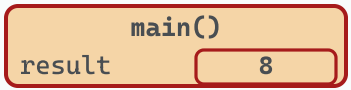
`add(5, 3)`

`main()`

Call and Return: Step by Step

```
1 static int add(int num1, int num2) {
2   int result = num1 + num2;
3   return result;
4 }
5
6 public static void main(String[] args) {
▶   int result = add(5, 3);
8   System.out.println("Sum: " + result);
9 }
```

Stack



Control returns to the calling line: the returned 8 is assigned to `main`'s `result` variable. The two `result` variables are different.

Step 5 / 6

OUTPUT

CALL STACK

`main()`

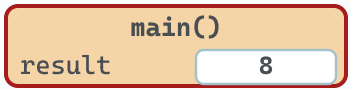
VARIABLES

`args = []` `result = 8`

Call and Return: Step by Step

```
1 static int add(int num1, int num2) {
2   int result = num1 + num2;
3   return result;
4 }
5
6 public static void main(String[] args) {
7   int result = add(5, 3);
▶   System.out.println("Sum: " + result);
9 }
```

Stack



`println` is also a method call: it prints **Sum: 8**.

Step 6 / 6

OUTPUT

```
Sum: 8
```

CALL STACK

`main()`

VARIABLES

`args = []` `result = 8`

DEFINITION AND CALLS

Sample Methods

```
public class SampleMethods {
    // adds two numbers
    public static int add(int num1, int num2) {
        return num1 + num2;
    }
    // computes a square
    public static double square(double num) {
        return num * num;
    }
    // prints hello world
    public static void helloWorld() {
        System.out.println("Hello, World!");
    }

    public static void main(String[] args) {
        System.out.println(add(2, 7));
        System.out.println(square(1.5));
        helloWorld();
    }
}
```

OUTPUT

```
9
2.25
Hello, World!
```

Method	Input	Output
`add`	2 × int	int
`square`	1 × double	double
`helloWorld`	none	none (void)

A value-returning method's result is used in an expression; a `void` method is called on its own as a statement.

DEFINITION AND CALLS

return and void

- `return expression;` hands the value to the caller and **ends the method at once**; the lines after it don't run.
- The returned value's type must be compatible with the declared return type (widening `int` → `double` is allowed).
- In a value-returning method, **every path** must end with a `return`; otherwise the compiler reports `missing return statement`.
- `void` methods return no value; a bare `return;` can be used to exit early.

```
public class EarlyReturn {
    static void printQuotient(int dividend, int divisor) {
        if (divisor == 0) {
            System.out.println("Cannot divide by zero!");
            return;                       // leave the method now
        }
        System.out.println(dividend / divisor);
    }
    public static void main(String[] args) {
        printQuotient(10, 2);
        printQuotient(7, 0);
    }
}
```

OUTPUT

```
5
Cannot divide by zero!
```

static Calls and Calls on an Object

static method

```
static int add(int num1, int num2) {
    return num1 + num2;
}
// directly, inside main:
int result = add(5, 3);
```

`static` methods belong to the class; no object is needed. From another class they are called as `ClassName.method()`, like `Math.sqrt(16)`.

object (instance) method

```
public class Calculator {
    int add(int num1, int num2) {
        return num1 + num2;
    }
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        int result = calc.add(5, 3);
        System.out.println(result);  // 8
    }
}
```

Methods that are not `static` are called on an **object**: first create the object with `new`, then call `calc.add(5, 3)`.

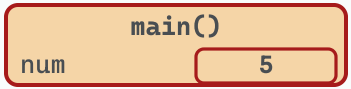
If the definition lacks `static`, then inside `main`, writing just `add(5, 3)` is a compile error:
`non-static method add(int,int) cannot be referenced from a static context`

Pass by Value

Can a method **change the variable it is given?**

```
1  static void increase(int num) {
2    num = num + 10;
3    System.out.println("increase: " + num);
4  }
5
6  public static void main(String[] args) {
▶    int num = 5;
8    increase(num);
9    System.out.println("main: " + num);
10 }
```

Stack



`main` : `num = 5` .

Step 1 / 5

OUTPUT

CALL STACK

`main()`

VARIABLES

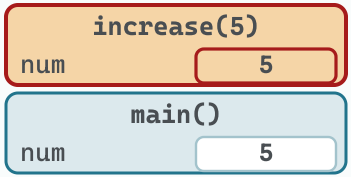
`num = 5`

Pass by Value

Can a method **change the variable it is given?**

```
1  static void increase(int num) {
2    num = num + 10;
3    System.out.println("increase: " + num);
4  }
5
6  public static void main(String[] args) {
7    int num = 5;
▶    increase(num);
9    System.out.println("main: " + num);
10 }
```

Stack



The call sends **a copy of the value** (5). increase's `num` is a separate box; sharing the same name means nothing.

Step 2 / 5

OUTPUT

VARIABLES

`num = 5`

CALL STACK

`increase(5)`

`main()`

Pass by Value

Can a method **change the variable it is given?**

```
1  static void increase(int num) {
▶    num = num + 10;
3    System.out.println("increase: " + num);
4  }
5
6  public static void main(String[] args) {
7    int num = 5;
8    increase(num);
9    System.out.println("main: " + num);
10 }
```

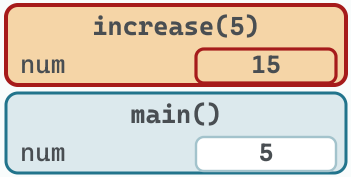
OUTPUT

```
```

VARIABLES

`num = 15`

Stack



Only the **copy** changes: increase's `num` becomes 15, main's is still 5.

Step 3 / 5

CALL STACK

`increase(5)`

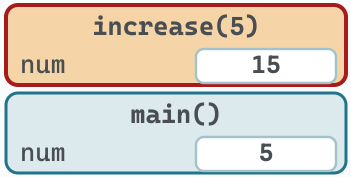
`main()`

Pass by Value

Can a method **change the variable it is given?**

```
1  static void increase(int num) {
2    num = num + 10;
▶    System.out.println("increase: " + num);
4  }
5
6  public static void main(String[] args) {
7    int num = 5;
8    increase(num);
9    System.out.println("main: " + num);
10 }
```

Stack



increase prints 15.

Step 4 / 5

OUTPUT

```
increase: 15
```

VARIABLES

`num = 15`

CALL STACK

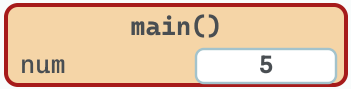
- `increase(5)`
- `main()`

Pass by Value

Can a method **change the variable it is given?**

```
1  static void increase(int num) {
2    num = num + 10;
3    System.out.println("increase: " + num);
4  }
5
6  public static void main(String[] args) {
7    int num = 5;
8    increase(num);
▶    System.out.println("main: " + num);
10 }
```

Stack



increase is done; its frame and copy are gone. main's `num` **5** remains: Java passes arguments **by value**.

Step 5 / 5

OUTPUT

```
increase: 15
main: 5
```

CALL STACK

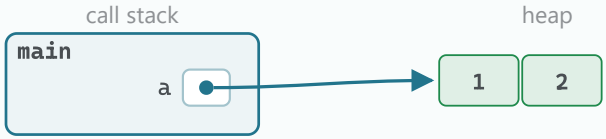
`main()`

VARIABLES

`num = 5`

A Copy of the Reference: Tracing the Code

```
 1 static void modify(int[] d, int x) {
 2     d[0] = 50;
 3     x = 50;
 4     d = new int[2];
 5     d[1] = 50;
 6 }
 7
 8 public static void main(String[] args) {
 ▶     int[] a = {1, 2};
10     int x = 1;
11     modify(a, x);
12     System.out.println(a[0] + " " + a[1] +
   " " + x);
13 }
```



`a` is a **reference**: the array object lives on the heap, and `a` holds its address.

Step 1 / 9

OUTPUT

CALL STACK

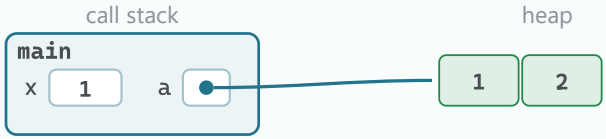
`main()`

VARIABLES

—

A Copy of the Reference: Tracing the Code

```
1  static void modify(int[] d, int x) {
2      d[0] = 50;
3      x = 50;
4      d = new int[2];
5      d[1] = 50;
6  }
7
8  public static void main(String[] args) {
9      int[] a = {1, 2};
▶      int x = 1;
11     modify(a, x);
12     System.out.println(a[0] + " " + a[1] +
   " " + x);
13 }
```



`x = 1` : the primitive value is stored right in the frame.

Step 2 / 9

OUTPUT

CALL STACK

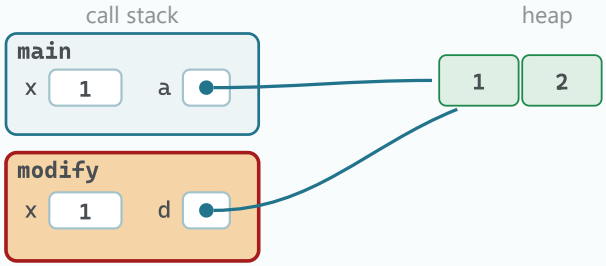
`main()`

VARIABLES

`x = 1`

A Copy of the Reference: Tracing the Code

```
▶ static void modify(int[] d, int x) {
2     d[0] = 50;
3     x = 50;
4     d = new int[2];
5     d[1] = 50;
6 }
7
8 public static void main(String[] args) {
9     int[] a = {1, 2};
10    int x = 1;
11    modify(a, x);
12    System.out.println(a[0] + " " + a[1] +
  " " + x);
13 }
```



The call: `d` gets `a`'s **reference copy**, `x` gets a copy of 1. `d` and `a` refer to the same array.

Step 3 / 9

OUTPUT

VARIABLES

`x = 1`

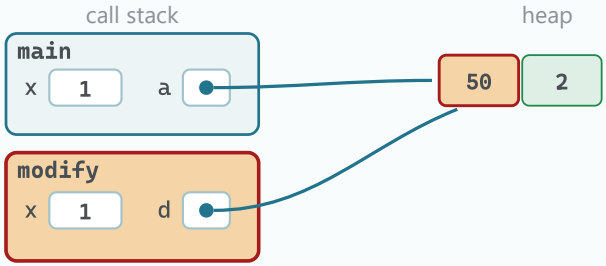
CALL STACK

`modify(a, x)`

`main()`

A Copy of the Reference: Tracing the Code

```
1  static void modify(int[] d, int x) {
▶      d[0] = 50;
3      x = 50;
4      d = new int[2];
5      d[1] = 50;
6  }
7
8  public static void main(String[] args) {
9      int[] a = {1, 2};
10     int x = 1;
11     modify(a, x);
12     System.out.println(a[0] + " " + a[1] +
   " " + x);
13 }
```



`d[0] = 50` : the shared array changes; `main` will see the change too.

Step 4 / 9

OUTPUT

VARIABLES

`x = 1`

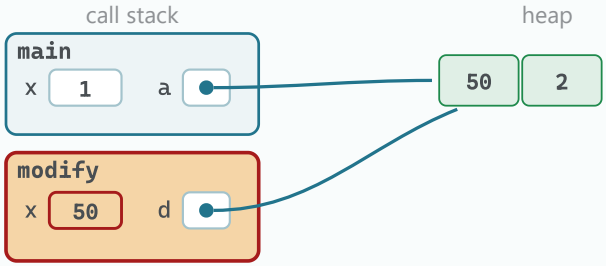
CALL STACK

`modify(a, x)`

`main()`

A Copy of the Reference: Tracing the Code

```
1  static void modify(int[] d, int x) {
2      d[0] = 50;
▶      x = 50;
4      d = new int[2];
5      d[1] = 50;
6  }
7
8  public static void main(String[] args) {
9      int[] a = {1, 2};
10     int x = 1;
11     modify(a, x);
12     System.out.println(a[0] + " " + a[1] +
   " " + x);
13 }
```



`x = 50` : only `modify` 's copy changes.

Step 5 / 9

OUTPUT

VARIABLES

`x = 50`

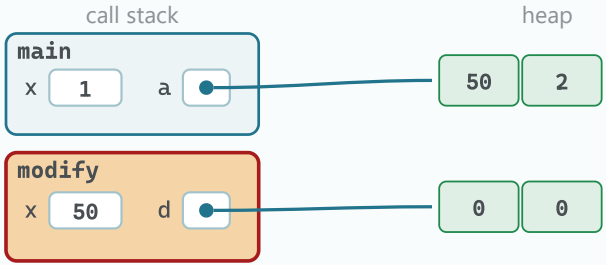
CALL STACK

`modify(a, x)`

`main()`

A Copy of the Reference: Tracing the Code

```
1  static void modify(int[] d, int x) {
2      d[0] = 50;
3      x = 50;
▶      d = new int[2];
5      d[1] = 50;
6  }
7
8  public static void main(String[] args) {
9      int[] a = {1, 2};
10     int x = 1;
11     modify(a, x);
12     System.out.println(a[0] + " " + a[1] +
   " " + x);
13 }
```



`d = new int[2]` : `d` now refers to a **new** array. `a` is unaffected; the link is broken.

Step 6 / 9

OUTPUT

VARIABLES

`x = 50`

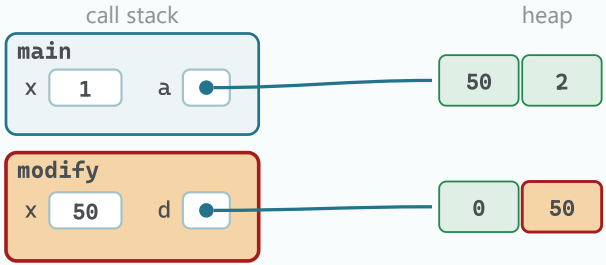
CALL STACK

`modify(a, x)`

`main()`

A Copy of the Reference: Tracing the Code

```
1  static void modify(int[] d, int x) {
2      d[0] = 50;
3      x = 50;
4      d = new int[2];
▶      d[1] = 50;
6  }
7
8  public static void main(String[] args) {
9      int[] a = {1, 2};
10     int x = 1;
11     modify(a, x);
12     System.out.println(a[0] + " " + a[1] +
   " " + x);
13 }
```



`d[1] = 50` is written to the new array; `main`'s array is unchanged.

Step 7 / 9

OUTPUT

VARIABLES

`x = 50`

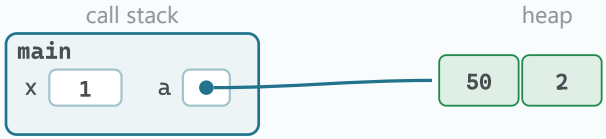
CALL STACK

`modify(a, x)`

`main()`

A Copy of the Reference: Tracing the Code

```
1  static void modify(int[] d, int x) {
2      d[0] = 50;
3      x = 50;
4      d = new int[2];
5      d[1] = 50;
6  }
7
8  public static void main(String[] args) {
9      int[] a = {1, 2};
10     int x = 1;
▶      modify(a, x);
12     System.out.println(a[0] + " " + a[1] +
   " " + x);
13 }
```



The method is done; its frame is gone. Nothing refers to the new array any more (the garbage collector will reclaim it).

Step 8 / 9

OUTPUT

CALL STACK

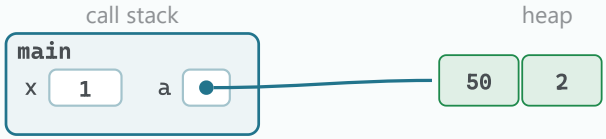
`main()`

VARIABLES

`x = 1`

A Copy of the Reference: Tracing the Code

```
1  static void modify(int[] d, int x) {
2      d[0] = 50;
3      x = 50;
4      d = new int[2];
5      d[1] = 50;
6  }
7
8  public static void main(String[] args) {
9      int[] a = {1, 2};
10     int x = 1;
11     modify(a, x);
▶      System.out.println(a[0] + " " + a[1] +
   " " + x);
13 }
```



`50 2 1` : the array's contents changed, `x` did not. Java passes everything by value; a reference is just a value that gets copied.

Step 9 / 9

OUTPUT

```
50 2 1
```

CALL STACK

`main()`

VARIABLES

`x = 1`

DEFINITION AND CALLS

Method Overloading

```
public class Overloading {
    static int sum(int a, int b) {
        return a + b;
    }
    static double sum(double a, double b) {
        return a + b;
    }
    static int sum(int a, int b, int c) {
        return a + b + c;
    }

    public static void main(String[] args) {
        System.out.println(sum(2, 3));
        System.out.println(sum(2.5, 1.2));
        System.out.println(sum(1, 2, 3));
    }
}
```

OUTPUT

```
5
3.7
6
```

Overloading: several methods with the same name can live in one class as long as their **parameter lists differ**.

- They can differ in the **number**, **type** or **order of parameters**.
- The compiler looks at a call's arguments and decides which method runs **at compile time**.
- Two methods that differ only in return type are **not overloads**: `method sum(int,int) is already defined` is what you get.
- `println` is overloaded too: `println(int)`, `println(String)`, `println(double)`...

Which sum Runs? Step by Step

```
1  static int sum(int a, int b) {
2      return a + b;
3  }
4  static double sum(double a, double b) {
5      return a + b;
6  }
7  static int sum(int a, int b, int c) {
8      return a + b + c;
9  }
10 public static void main(String[] args) {
▶      System.out.println(sum(4, 6));
12     System.out.println(sum(4, 2.5));
13     System.out.println(sum(7, 1, 2));
14     System.out.println(sum('A', 1));
15 }
```

sum(4, 6) — **arguments: int, int**

- `sum(int, int)` — ✓ exact match
- `sum(double, double)` — ✓ fits by widening
- `sum(int, int, int)` — ✗ wrong count

There are three methods named `sum`. The first call's arguments are `(int, int)`: the three-parameter version is ruled out; `(int, int)` **matches exactly**, so it is chosen.

Step 1 / 12

OUTPUT

CALL STACK

`main()`

Which sum Runs? Step by Step

```
1  static int sum(int a, int b) {
▶      return a + b;
3  }
4  static double sum(double a, double b) {
5      return a + b;
6  }
7  static int sum(int a, int b, int c) {
8      return a + b + c;
9  }
10 public static void main(String[] args) {
11     System.out.println(sum(4, 6));
12     System.out.println(sum(4, 2.5));
13     System.out.println(sum(7, 1, 2));
14     System.out.println(sum('A', 1));
15 }
```

sum(4, 6) — **arguments: int, int**

- `sum(int, int)` — `a = 4, b = 6`
- `sum(double, double)` — ✓ fits by widening
- `sum(int, int, int)` — ✗ wrong count

`sum(int, int)` runs: `a = 4, b = 6`.

Step 2 / 12

OUTPUT

CALL STACK

- `sum(int, int)`
- `main()`

Which sum Runs? Step by Step

```
1  static int sum(int a, int b) {
2      return a + b;
3  }
4  static double sum(double a, double b) {
5      return a + b;
6  }
7  static int sum(int a, int b, int c) {
8      return a + b + c;
9  }
10 public static void main(String[] args) {
▶      System.out.println(sum(4, 6));
12     System.out.println(sum(4, 2.5));
13     System.out.println(sum(7, 1, 2));
14     System.out.println(sum('A', 1));
15 }
```

sum(4, 6) — **arguments: int, int**

sum(int, int)	**returns 10**
sum(double, double)	✓ fits by widening
sum(int, int, int)	✗ wrong count

10 is returned and printed.

Step 3 / 12

OUTPUT

```
10
```

CALL STACK

main()

Which sum Runs? Step by Step

```
1  static int sum(int a, int b) {
2      return a + b;
3  }
4  static double sum(double a, double b) {
5      return a + b;
6  }
7  static int sum(int a, int b, int c) {
8      return a + b + c;
9  }
10 public static void main(String[] args) {
11     System.out.println(sum(4, 6));
▶      System.out.println(sum(4, 2.5));
13     System.out.println(sum(7, 1, 2));
14     System.out.println(sum('A', 1));
15 }
```

`sum(4, 2.5)` — **arguments: int, double**

- `sum(int, int)` — ✗ 2.5 is not an int
- **`sum(double, double)`** — ✓ 4 → 4.0
- `sum(int, int, int)` — ✗ wrong count

`(int, double)`: 2.5 to `int`? Not implicitly, so `(int, int)` is out. 4 is widened to 4.0 → `sum(double, double)`.

Step 4 / 12

OUTPUT

```
10
```

CALL STACK

`main()`

Which sum Runs? Step by Step

```
1  static int sum(int a, int b) {
2      return a + b;
3  }
4  static double sum(double a, double b) {
▶      return a + b;
6  }
7  static int sum(int a, int b, int c) {
8      return a + b + c;
9  }
10 public static void main(String[] args) {
11     System.out.println(sum(4, 6));
12     System.out.println(sum(4, 2.5));
13     System.out.println(sum(7, 1, 2));
14     System.out.println(sum('A', 1));
15 }
```

sum(4, 2.5) — **arguments: int, double**

- `sum(int, int)` — ✗ 2.5 is not an int
- **`sum(double, double)`** — `a = 4.0, b = 2.5`
- `sum(int, int, int)` — ✗ wrong count

`a = 4.0, b = 2.5` : the parameters are `double`.

Step 5 / 12

OUTPUT

```
10
```

CALL STACK

- **`sum(double, double)`**
- `main()`

Which sum Runs? Step by Step

```
1  static int sum(int a, int b) {
2      return a + b;
3  }
4  static double sum(double a, double b) {
5      return a + b;
6  }
7  static int sum(int a, int b, int c) {
8      return a + b + c;
9  }
10 public static void main(String[] args) {
11     System.out.println(sum(4, 6));
▶      System.out.println(sum(4, 2.5));
13     System.out.println(sum(7, 1, 2));
14     System.out.println(sum('A', 1));
15 }
```

sum(4, 2.5) — **arguments: int, double**

- `sum(int, int)` — ✗ 2.5 is not an int
- `sum(double, double)` — **returns 6.5**
- `sum(int, int, int)` — ✗ wrong count

6.5 is returned and printed.

Step 6 / 12

OUTPUT

```
10
6.5
```

CALL STACK

`main()`

Which sum Runs? Step by Step

```
1  static int sum(int a, int b) {
2      return a + b;
3  }
4  static double sum(double a, double b) {
5      return a + b;
6  }
7  static int sum(int a, int b, int c) {
8      return a + b + c;
9  }
10 public static void main(String[] args) {
11     System.out.println(sum(4, 6));
12     System.out.println(sum(4, 2.5));
▶      System.out.println(sum(7, 1, 2));
14     System.out.println(sum('A', 1));
15 }
```

sum(7, 1, 2) — **arguments: int, int, int**

- `sum(int, int)` — ✗ wrong count
- `sum(double, double)` — ✗ wrong count
- `sum(int, int, int)` — ✓ exact match

Three arguments: only the three-parameter version fits.

Step 7 / 12

OUTPUT

```
10
6.5
```

CALL STACK

`main()`

Which sum Runs? Step by Step

```
1  static int sum(int a, int b) {
2      return a + b;
3  }
4  static double sum(double a, double b) {
5      return a + b;
6  }
7  static int sum(int a, int b, int c) {
▶      return a + b + c;
9  }
10 public static void main(String[] args) {
11     System.out.println(sum(4, 6));
12     System.out.println(sum(4, 2.5));
13     System.out.println(sum(7, 1, 2));
14     System.out.println(sum('A', 1));
15 }
```

sum(7, 1, 2) — **arguments: int, int, int**

- `sum(int, int)` — ✗ wrong count
- `sum(double, double)` — ✗ wrong count
- `sum(int, int, int)` — `a = 7, b = 1, c = 2`

`a = 7, b = 1, c = 2` .

Step 8 / 12

OUTPUT

```
10
6.5
```

CALL STACK

- `sum(int, int, int)`
- `main()`

Which sum Runs? Step by Step

```
1  static int sum(int a, int b) {
2      return a + b;
3  }
4  static double sum(double a, double b) {
5      return a + b;
6  }
7  static int sum(int a, int b, int c) {
8      return a + b + c;
9  }
10 public static void main(String[] args) {
11     System.out.println(sum(4, 6));
12     System.out.println(sum(4, 2.5));
▶      System.out.println(sum(7, 1, 2));
14     System.out.println(sum('A', 1));
15 }
```

sum(7, 1, 2) — **arguments: int, int, int**

- `sum(int, int)` — ✗ wrong count
- `sum(double, double)` — ✗ wrong count
- `sum(int, int, int)` — **returns 10**

10 is returned and printed.

Step 9 / 12

OUTPUT

```
10
6.5
10
```

CALL STACK

`main()`

Which sum Runs? Step by Step

```
1  static int sum(int a, int b) {
2      return a + b;
3  }
4  static double sum(double a, double b) {
5      return a + b;
6  }
7  static int sum(int a, int b, int c) {
8      return a + b + c;
9  }
10 public static void main(String[] args) {
11     System.out.println(sum(4, 6));
12     System.out.println(sum(4, 2.5));
13     System.out.println(sum(7, 1, 2));
▶      System.out.println(sum('A', 1));
15 }
```

sum('A', 1) — **arguments: char, int**

`sum(int, int)`	✓ 'A' → 65
`sum(double, double)`	✓ fits by widening
`sum(int, int, int)`	✗ wrong count

`(char, int)` : `char` can widen to both `int` and `double` ; the compiler picks the **most specific** one, `(int, int)` .

Step 10 / 12

OUTPUT

```
10
6.5
10
```

CALL STACK

`main()`

Which sum Runs? Step by Step

```
1  static int sum(int a, int b) {
2      return a + b;
3  }
4  static double sum(double a, double b) {
5      return a + b;
6  }
7  static int sum(int a, int b, int c) {
8      return a + b + c;
9  }
10 public static void main(String[] args) {
11     System.out.println(sum(4, 6));
12     System.out.println(sum(4, 2.5));
13     System.out.println(sum(7, 1, 2));
14     System.out.println(sum('A', 1));
15 }
```

sum('A', 1) — arguments: char, int

sum(int, int)	a = 65, b = 1
sum(double, double)	✓ fits by widening
sum(int, int, int)	✗ wrong count

'A' → 65: a = 65, b = 1 .

Step 11 / 12

OUTPUT

```
10
6.5
10
```

CALL STACK

- sum(int, int)
- main()

Which sum Runs? Step by Step

```
1  static int sum(int a, int b) {
2      return a + b;
3  }
4  static double sum(double a, double b) {
5      return a + b;
6  }
7  static int sum(int a, int b, int c) {
8      return a + b + c;
9  }
10 public static void main(String[] args) {
11     System.out.println(sum(4, 6));
12     System.out.println(sum(4, 2.5));
13     System.out.println(sum(7, 1, 2));
▶      System.out.println(sum('A', 1));
15 }
```

sum('A', 1) — **arguments: char, int**

- **sum(int, int)** — **returns 66**
- **sum(double, double)** — ✓ fits by widening
- **sum(int, int, int)** — ✗ wrong count

66 is returned and printed. The choice is made at compile time, from the argument types alone.

Step 12 / 12

OUTPUT

```
10
6.5
10
66
```

CALL STACK

main()

Methods Calling Methods

```java
public class Nested {
    public static int square(int num) {
        return num * num;
    }
    public static int sum(int num1, int num2) {
        return num1 + num2;
    }
    public static int sumOfSquares(int num1, int num2) {
        return sum(square(num1), square(num2));
    }
    public static void main(String[] args) {
        int result = sumOfSquares(3, 4);
        System.out.println(result);
    }
}
```

OUTPUT

```
25
```

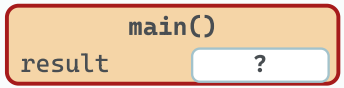
- A method **can call another method**.
- A complex task is split into small pieces: `sumOfSquares` delegates the work to `square` and `sum`.
- The code becomes readable and manageable; each piece can be tested on its own.
- `sum(square(num1), square(num2))` evaluates the **arguments** first (left to right), then makes the outer call.

$3^2 + 4^2 = 9 + 16 = \mathbf{25}$.

sumOfSquares(3, 4): The Call Stack

```
 1 static int square(int num) {
 2   return num * num;
 3 }
 4 static int sum(int num1, int num2) {
 5   return num1 + num2;
 6 }
 7 static int sumOfSquares(int num1,
 8                          int num2) {
 9   return sum(square(num1), square(num2));
10 }
11 public static void main(String[] args) {
▶   int result = sumOfSquares(3, 4);
13   System.out.println(result);
14 }
```

Stack



`main` calls `sumOfSquares(3, 4)`.

Step 1 / 9

OUTPUT

CALL STACK

`main()`

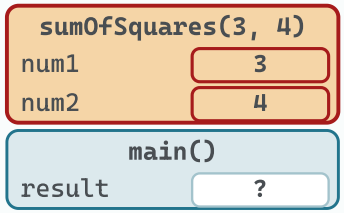
VARIABLES

`result = ?`

sumOfSquares(3, 4): The Call Stack

```
1  static int square(int num) {
2    return num * num;
3  }
4  static int sum(int num1, int num2) {
5    return num1 + num2;
6  }
7  static int sumOfSquares(int num1,
8                          int num2) {
▶    return sum(square(num1), square(num2));
10 }
11 public static void main(String[] args) {
12   int result = sumOfSquares(3, 4);
13   System.out.println(result);
14 }
```

Stack



`sum` needs its arguments before it can be called: first `square(num1)`.

Step 2 / 9

OUTPUT

VARIABLES

`num1 = 3` `num2 = 4`

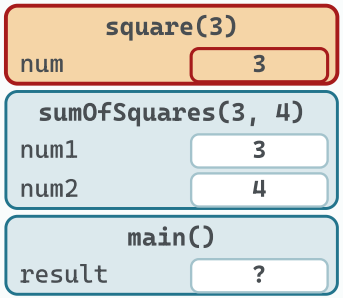
CALL STACK

- `sumOfSquares(3, 4)`
- `main()`

sumOfSquares(3, 4): The Call Stack

```
1  static int square(int num) {
▶    return num * num;
3  }
4  static int sum(int num1, int num2) {
5    return num1 + num2;
6  }
7  static int sumOfSquares(int num1,
8                          int num2) {
9    return sum(square(num1), square(num2));
10 }
11 public static void main(String[] args) {
12   int result = sumOfSquares(3, 4);
13   System.out.println(result);
14 }
```

Stack



A third frame is pushed: `square(3)` → 3 × 3 = **9** is returned.

Step 3 / 9

OUTPUT

VARIABLES

`num = 3` `return = 9`

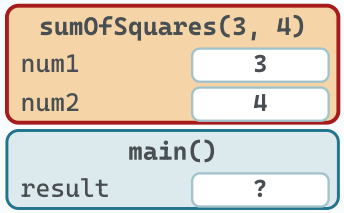
CALL STACK

- `square(3)`
- `sumOfSquares(3, 4)`
- `main()`

sumOfSquares(3, 4): The Call Stack

```
1  static int square(int num) {
2    return num * num;
3  }
4  static int sum(int num1, int num2) {
5    return num1 + num2;
6  }
7  static int sumOfSquares(int num1,
8                          int num2) {
▶    return sum(square(num1), square(num2));
10 }
11 public static void main(String[] args) {
12   int result = sumOfSquares(3, 4);
13   System.out.println(result);
14 }
```

Stack



square's frame is popped; the first argument (`arg1`) = 9. Next: `square(num2)` .

Step 4 / 9

OUTPUT

VARIABLES

`num1 = 3` `num2 = 4` `arg1 = 9`

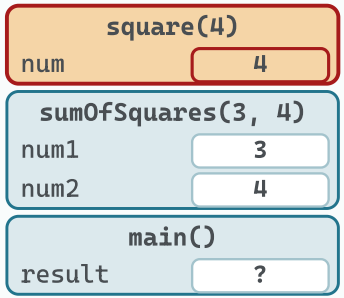
CALL STACK

- `sumOfSquares(3, 4)`
- `main()`

sumOfSquares(3, 4): The Call Stack

```
1  static int square(int num) {
▶    return num * num;
3  }
4  static int sum(int num1, int num2) {
5    return num1 + num2;
6  }
7  static int sumOfSquares(int num1,
8                          int num2) {
9    return sum(square(num1), square(num2));
10 }
11 public static void main(String[] args) {
12   int result = sumOfSquares(3, 4);
13   System.out.println(result);
14 }
```

Stack



`square(4)` → 4 × 4 = **16** is returned.

Step 5 / 9

OUTPUT

VARIABLES

`num = 4` `return = 16`

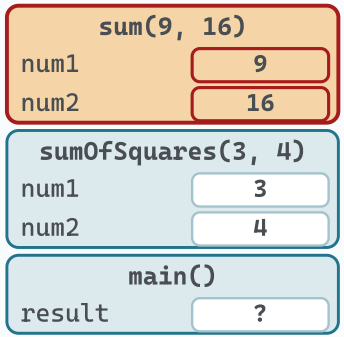
CALL STACK

- **square(4)**
- sumOfSquares(3, 4)
- main()

sumOfSquares(3, 4): The Call Stack

```
1  static int square(int num) {
2    return num * num;
3  }
4  static int sum(int num1, int num2) {
▶    return num1 + num2;
6  }
7  static int sumOfSquares(int num1,
8                          int num2) {
9    return sum(square(num1), square(num2));
10 }
11 public static void main(String[] args) {
12   int result = sumOfSquares(3, 4);
13   System.out.println(result);
14 }
```

Stack



The arguments are ready: `sum(9, 16)` is called → **25**. Here `num1` = 9; sumOfSquares's is still 3.

Step 6 / 9

OUTPUT

VARIABLES

`num1 = 9` `num2 = 16` `return = 25`

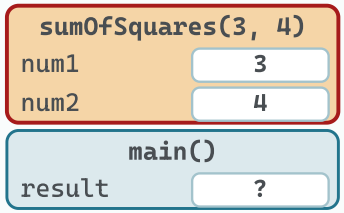
CALL STACK

- `sum(9, 16)`
- `sumOfSquares(3, 4)`
- `main()`

sumOfSquares(3, 4): The Call Stack

```
1  static int square(int num) {
2    return num * num;
3  }
4  static int sum(int num1, int num2) {
5    return num1 + num2;
6  }
7  static int sumOfSquares(int num1,
8                          int num2) {
▶    return sum(square(num1), square(num2));
10 }
11 public static void main(String[] args) {
12   int result = sumOfSquares(3, 4);
13   System.out.println(result);
14 }
```

Stack



sum's result becomes sumOfSquares's return value: **25** is returned.

Step 7 / 9

OUTPUT

VARIABLES

`num1 = 3` `num2 = 4` `return = 25`

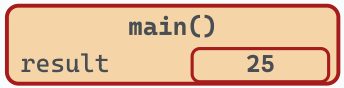
CALL STACK

- `sumOfSquares(3, 4)`
- `main()`

sumOfSquares(3, 4): The Call Stack

```
1  static int square(int num) {
2    return num * num;
3  }
4  static int sum(int num1, int num2) {
5    return num1 + num2;
6  }
7  static int sumOfSquares(int num1,
8                          int num2) {
9    return sum(square(num1), square(num2));
10 }
11 public static void main(String[] args) {
▶    int result = sumOfSquares(3, 4);
13   System.out.println(result);
14 }
```

Stack



Back in main: `result = 25`.

Step 8 / 9

OUTPUT

CALL STACK

`main()`

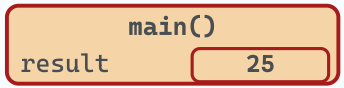
VARIABLES

`result = 25`

sumOfSquares(3, 4): The Call Stack

```
 1 static int square(int num) {
 2   return num * num;
 3 }
 4 static int sum(int num1, int num2) {
 5   return num1 + num2;
 6 }
 7 static int sumOfSquares(int num1,
 8                          int num2) {
 9   return sum(square(num1), square(num2));
10 }
11 public static void main(String[] args) {
12   int result = sumOfSquares(3, 4);
▶   System.out.println(result);
14 }
```

Stack



The screen shows **25**. At most 3 frames were on the stack at once.

Step 9 / 9

OUTPUT

```
25
```

CALL STACK

`main()`

VARIABLES

`result = 25`

The Most Common Methods (Java API)

Area	Classes	Example methods	Package
Console output	`System.out`	`print`, `println`, `printf`	java.lang
Math	`Math`	`sqrt`, `abs`, `max`, `pow`	java.lang
Text	`String`	`length`, `charAt`, `substring`, `equals`	java.lang
Arrays	`Arrays`	`sort`, `copyOf`, `binarySearch`, `equals`	java.util
User input	`Scanner`	`nextInt`, `nextLine`, `close`	java.util
Files	`File`, `FileWriter`	`createNewFile`, `write`, `close`	java.io
Databases	`DriverManager`, `Statement`	`getConnection`, `executeQuery`, `executeUpdate`	java.sql
Data structures	`ArrayList`	`add`, `remove`, `get`, `size`	java.util
Time	`LocalDateTime`, `DateTimeFormatter`	`now`, `ofPattern`, `format`	java.time
Web / graphics	`URI`, `URL`, `JFrame`	`URI.create`, `openConnection`, `setVisible`	java.net, javax.swing

`java.lang` is imported automatically; the others need an `import`, e.g. `import java.util.Scanner;`

BUILT-IN METHODS

Printing to the Screen

```
public class Printing {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
        System.out.print("One ");
        System.out.print("line");
        System.out.println();
        int age = 20;
        System.out.println("Age: " + age);
        System.out.printf("Pi is about %.2f%n", Math.PI);
    }
}
```

OUTPUT

```
Hello, World!
One line
Age: 20
Pi is about 3,14
```

- `System.out.println(x)` : prints x and moves **to the next line**.
- `System.out.print(x)` : prints x and stays on the same line.
- `+` joins text and values: `"Age: " + age` .
- `printf` : formatted output; `%.2f` 2 decimal places, `%d` integer, `%s` text, `%n` line break.
- `printf` uses the locale: on a Turkish system the decimal separator is a **comma** (3,14).

BUILT-IN METHODS

The Math Class

```
double root = Math.sqrt(16);
double absValue = Math.abs(-17.2);
double maximum = Math.max(3, 5);
System.out.println(root);
System.out.println(absValue);
System.out.println(maximum);
System.out.println(Math.pow(2, 10));
System.out.println(Math.round(2.6));
System.out.println(Math.min(-4, 7));
System.out.println(Math.max(3, 5));
```

OUTPUT

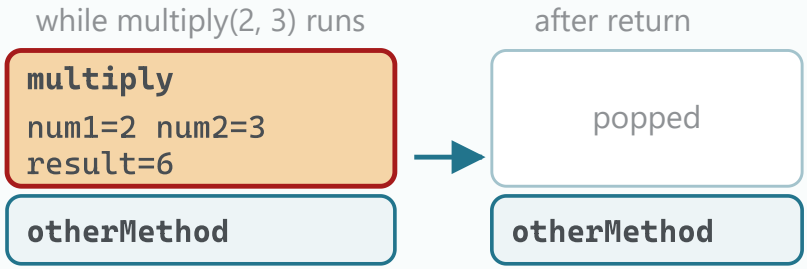
```
4.0
17.2
5.0
1024.0
3
-4
5
```

- The `Math` class holds math operations; its methods are `static`, so they are called as `Math.sqrt(16)`.
- `Math.max(3, 5)` returns `int` 5; stored in a `double` variable, it becomes **5.0**.
- `sqrt` and `pow` always return `double`.

Local Variables

```java
public int multiply(int num1, int num2) {
    int result = num1 * num2; // 'result' is a local variable
    return result;
}
public void otherMethod() {
    multiply(2, 3);
    System.out.println(result);  // ERROR
}
```

```
error: cannot find symbol
        System.out.println(result);
                           ^
  symbol:   variable result
```

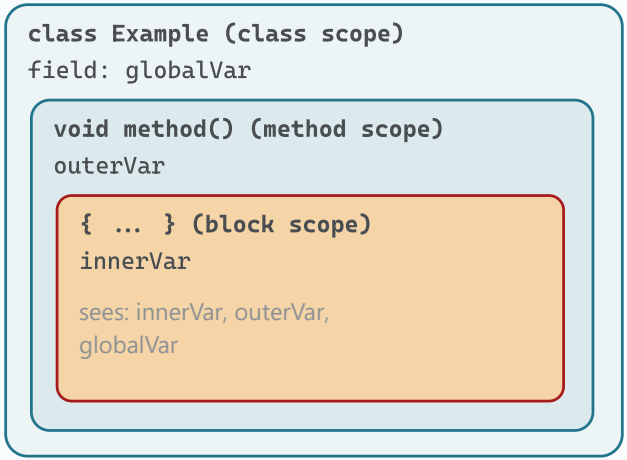


- **A local variable** is declared inside a method or block; parameters are local too.
- It exists only while the method runs; **it cannot be accessed from outside the method**.
- `result` exists only while `multiply` runs; it disappears when the method returns.
- The way to carry a value out is `return` : `int s = multiply(2, 3);`

Scope

Scope is the region of code where a variable is declared and can be used.

- Every variable has a scope: in Java, it is the **brace-delimited block**.
- Scope determines a variable's **accessibility** and **lifetime**.
- Local and global scopes have different access rules.
- In nested scopes, the inner scope can access the outer scope's variables; the outer scope cannot see the inner one.
- Using the same name in different scopes can lead to **name clashes**; misused scope causes bugs.



SCOPE

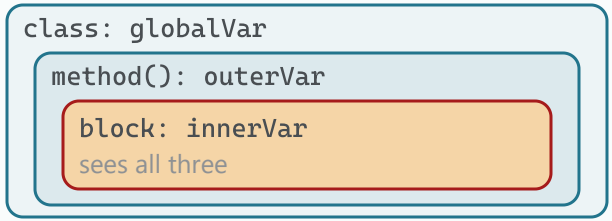
Local and Global Scope

```java
public class InnerScope {
    static int globalVar = 10; // global (class field)
    static void method() {
        int outerVar = 10;       // local scope
        {
            int innerVar = 5;
            // outerVar and innerVar are visible here
            System.out.println(outerVar + innerVar
                    + globalVar);
        }
        // innerVar does NOT exist here
    }
    public static void main(String[] args) {
        method();
    }
}
```

OUTPUT

```
25
```

10 + 5 + 10 = 25: the inner block sees both the method's and the class's variables.



Local scope

Inside a method or block; lives as long as that block runs.

Global scope

Accessible everywhere, lives for the whole program. In Java, class **fields** play this role.

Nested scope

A scope within a scope; the inner one can access the outer one's variables.

Name Clashes

Same name inside a block: error

```
public void method() {
    int outerVar = 10;
    {
        int outerVar = 5;  // ERROR
    }
}
```

```
error: variable outerVar is already
       defined in method method()
```

Java does **not allow** it.

A local variable shadows a field

```
public class Shadow {
    static int value = 10;  // class field
    static void show() {
        int value = 5;      // shadows the field
        System.out.println(value);
        System.out.println(Shadow.value);
    }
    public static void main(String[] args) {
        show();
        System.out.println(value);
    }
}
```

OUTPUT

```
5
10
10
```

The local `value` **shadows** the field with the same name; the field is reached as `Shadow.value`.

Shadowing: Tracing the Code

```
1  public class Shadow {
2      static int value = 10;  // class field
3      static void show() {
4          int value = 5;      // shadows the field
5          System.out.println(value);
6          System.out.println(Shadow.value);
7      }
8      public static void main(String[] args) {
▶          show();
10         System.out.println(value);
11     }
12 }
```

class field
Shadow.value = 10

main

The class field `value = 10` is created when the program starts and lives as long as it runs. `main` calls `show()`.

Step 1 / 6

OUTPUT

CALL STACK

main()

VARIABLES

`Shadow.value = 10`

Shadowing: Tracing the Code

```
1  public class Shadow {
2      static int value = 10;  // class field
▶      static void show() {
4          int value = 5;      // shadows the field
5          System.out.println(value);
6          System.out.println(Shadow.value);
7      }
8      public static void main(String[] args) {
9          show();
10         System.out.println(value);
11     }
12 }
```



`show` gets a new frame; it has no local variables yet.

Step 2 / 6

OUTPUT

VARIABLES

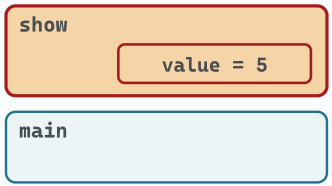
`Shadow.value = 10`

CALL STACK

- `show()`
- `main()`

Shadowing: Tracing the Code

```
1  public class Shadow {
2      static int value = 10;  // class field
3      static void show() {
▶          int value = 5;      // shadows the field
5          System.out.println(value);
6          System.out.println(Shadow.value);
7      }
8      public static void main(String[] args) {
9          show();
10         System.out.println(value);
11     }
12 }
```





The local `value = 5` is declared. Inside this method the name now means the local: the field is **shadowed**.

Step 3 / 6

OUTPUT

VARIABLES

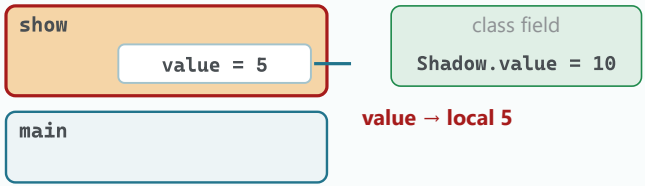
`Shadow.value = 10` `value (local) = 5`

CALL STACK

- `show()`
- `main()`

Shadowing: Tracing the Code

```
1  public class Shadow {
2      static int value = 10;  // class field
3      static void show() {
4          int value = 5;      // shadows the field
5          System.out.println(value);
6          System.out.println(Shadow.value);
7      }
8      public static void main(String[] args) {
9          show();
10         System.out.println(value);
11     }
12 }
```



`value` is looked up in the innermost scope first → the local 5 is printed.

Step 4 / 6

OUTPUT

```
5
```

VARIABLES

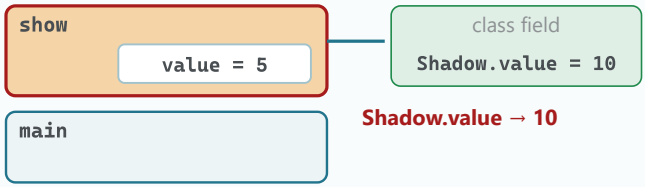
`Shadow.value = 10` `value (local) = 5`

CALL STACK

- `show()`
- `main()`

Shadowing: Tracing the Code

```
1  public class Shadow {
2      static int value = 10;  // class field
3      static void show() {
4          int value = 5;      // shadows the field
5          System.out.println(value);
▶          System.out.println(Shadow.value);
7      }
8      public static void main(String[] args) {
9          show();
10         System.out.println(value);
11     }
12 }
```



`Shadow.value` uses the class name to reach the field → 10.

Step 5 / 6

OUTPUT

```
5
10
```

CALL STACK

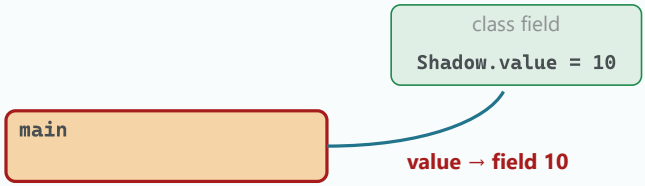
- `show()`
- `main()`

VARIABLES

`Shadow.value = 10` `value (local) = 5`

Shadowing: Tracing the Code

```
1  public class Shadow {
2      static int value = 10;  // class field
3      static void show() {
4          int value = 5;      // shadows the field
5          System.out.println(value);
6          System.out.println(Shadow.value);
7      }
8      public static void main(String[] args) {
9          show();
▶          System.out.println(value);
11     }
12 }
```



`show` has ended; the local `value` is gone. `main` has no local `value`: the name means the field → 10.

Step 6 / 6

OUTPUT

```
5
10
10
```

CALL STACK

`main()`

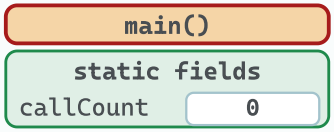
VARIABLES

`Shadow.value = 10`

When Are Variables Born, and When Do They Die?

```
1  static int callCount = 0;  // class field
2  static int multiply(int num1, int num2) {
3    callCount++;
4    int result = num1 * num2;
5    return result;
6  }
7  public static void main(String[] args) {
▶    int first = multiply(2, 3);
9    {
10     int second = multiply(first, 4);
11     System.out.println(second);
12   }
13   System.out.println(first + " " + callCount);
14 }
```

Memory



The class field `callCount` is created when the program starts and lives as long as it runs. `multiply(2, 3)` is being called.

Step 1 / 8

OUTPUT

CALL STACK

`main()`

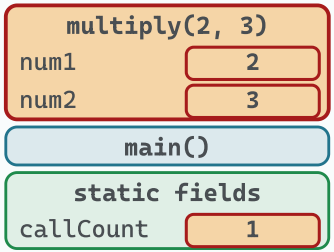
VARIABLES

`first = ?`

When Are Variables Born, and When Do They Die?

```
1  static int callCount = 0;  // class field
2  static int multiply(int num1, int num2) {
▶    callCount++;
4    int result = num1 * num2;
5    return result;
6  }
7  public static void main(String[] args) {
8    int first = multiply(2, 3);
9    {
10     int second = multiply(first, 4);
11     System.out.println(second);
12   }
13   System.out.println(first + " " + callCount);
14 }
```

Memory



The method can access the class field: `callCount` becomes 1.

Step 2 / 8

OUTPUT

VARIABLES

`num1 = 2` `num2 = 3`

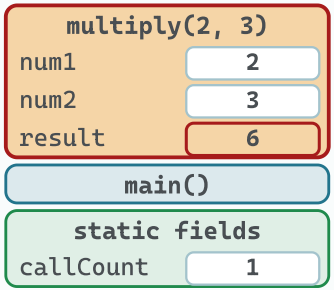
CALL STACK

- `multiply(2, 3)`
- `main()`

When Are Variables Born, and When Do They Die?

```
1  static int callCount = 0;  // class field
2  static int multiply(int num1, int num2) {
3    callCount++;
▶    int result = num1 * num2;
5    return result;
6  }
7  public static void main(String[] args) {
8    int first = multiply(2, 3);
9    {
10     int second = multiply(first, 4);
11     System.out.println(second);
12   }
13   System.out.println(first + " " + callCount);
14 }
```

Memory



The local `result` **is born** now: 6.

Step 3 / 8

OUTPUT

VARIABLES

`num1 = 2` `num2 = 3` `result = 6`

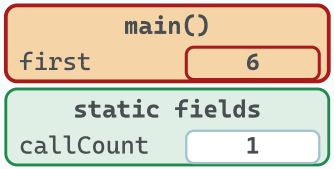
CALL STACK

- `multiply(2, 3)`
- `main()`

When Are Variables Born, and When Do They Die?

```
1  static int callCount = 0;  // class field
2  static int multiply(int num1, int num2) {
3    callCount++;
4    int result = num1 * num2;
5    return result;
6  }
7  public static void main(String[] args) {
▶    int first = multiply(2, 3);
9    {
10     int second = multiply(first, 4);
11     System.out.println(second);
12   }
13   System.out.println(first + " " + callCount);
14 }
```

Memory



multiply returned: `num1`, `num2`, `result` **are gone**. The value was assigned to `first`.

Step 4 / 8

OUTPUT

CALL STACK

`main()`

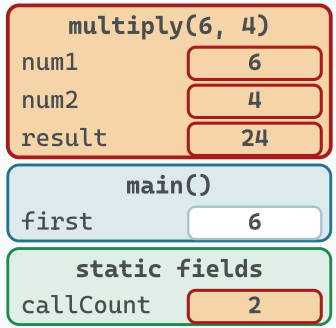
VARIABLES

`first = 6`

When Are Variables Born, and When Do They Die?

```
1  static int callCount = 0;  // class field
2  static int multiply(int num1, int num2) {
3    callCount++;
▶    int result = num1 * num2;
5    return result;
6  }
7  public static void main(String[] args) {
8    int first = multiply(2, 3);
9    {
10     int second = multiply(first, 4);
11     System.out.println(second);
12   }
13   System.out.println(first + " " + callCount);
14 }
```

Memory



New frame: `result` = 24, counter 2.

Step 5 / 8

OUTPUT

VARIABLES

`num1 = 6` `num2 = 4` `result = 24`

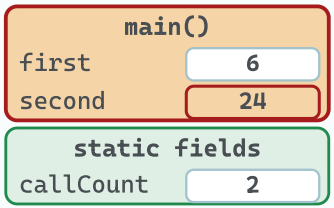
CALL STACK

- `multiply(6, 4)`
- `main()`

When Are Variables Born, and When Do They Die?

```
1  static int callCount = 0;  // class field
2  static int multiply(int num1, int num2) {
3    callCount++;
4    int result = num1 * num2;
5    return result;
6  }
7  public static void main(String[] args) {
8    int first = multiply(2, 3);
9    {
▶      int second = multiply(first, 4);
11     System.out.println(second);
12   }
13   System.out.println(first + " " + callCount);
14 }
```

Memory



`second` is a local variable that lives only inside the inner block.

Step 6 / 8

OUTPUT

CALL STACK

`main()`

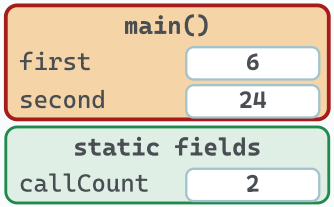
VARIABLES

`first = 6` `second = 24`

When Are Variables Born, and When Do They Die?

```
1  static int callCount = 0;  // class field
2  static int multiply(int num1, int num2) {
3    callCount++;
4    int result = num1 * num2;
5    return result;
6  }
7  public static void main(String[] args) {
8    int first = multiply(2, 3);
9    {
10     int second = multiply(first, 4);
▶      System.out.println(second);
12   }
13   System.out.println(first + " " + callCount);
14 }
```

Memory



Inside the block, `second` is accessible: **24** is printed.

Step 7 / 8

OUTPUT

```
24
```

CALL STACK

`main()`

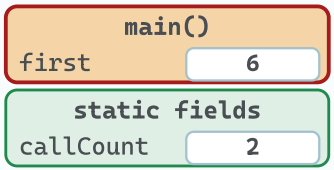
VARIABLES

`first = 6` `second = 24`

When Are Variables Born, and When Do They Die?

```
1  static int callCount = 0;  // class field
2  static int multiply(int num1, int num2) {
3    callCount++;
4    int result = num1 * num2;
5    return result;
6  }
7  public static void main(String[] args) {
8    int first = multiply(2, 3);
9    {
10     int second = multiply(first, 4);
11     System.out.println(second);
12   }
▶  System.out.println(first + " " + callCount);
14 }
```

Memory



The block ended, and `second` is gone: **6 2**.

Step 8 / 8

OUTPUT

```
24
6 2
```

CALL STACK

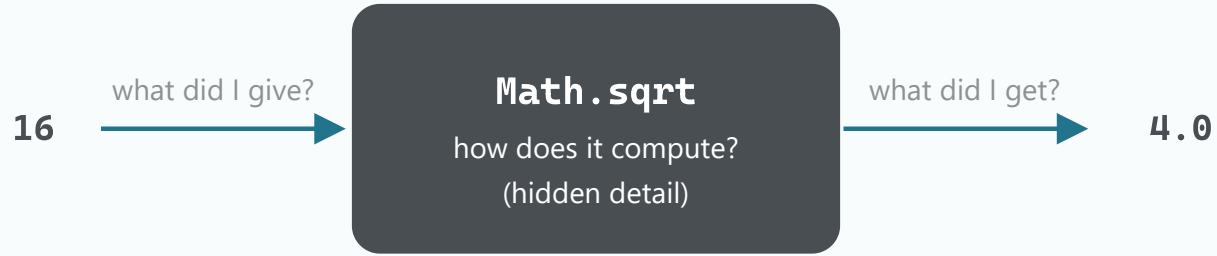
`main()`

VARIABLES

`first = 6`

Abstraction

Abstraction means **ignoring details to reduce complexity**.



Why?

It makes code more readable and simpler; it keeps large, complex programs understandable.

How?

It splits code into pieces that can be thought about separately. Whoever uses a method doesn't need to know its insides.

Everyday analogy: you don't need to know how the engine works to drive a car; the wheel and pedals are an "interface".

Procedural Abstraction

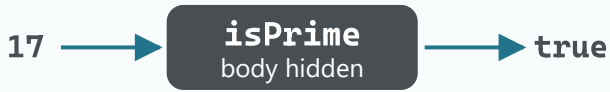
Abstraction over computations, **a.k.a. functional** or **procedural abstraction**.

- What matters is **what** a method computes, not **how** it computes it.
- As long as you know its job, you don't care how it does it.
- You don't worry about the method's implementation (its body).
- Even if the body changes (say, a faster algorithm), the calling code stays the same.

```
System.out.println("Hello");
```

The method prints the given text to the console; you don't need to know how (character encoding, OS calls...).

```
// the caller only knows the signature:
static boolean isPrime(int num)
// "returns true if num is prime"
if (isPrime(17)) { ... }
```



the user knows only the input and output

Generalization

Context:

- how a number relates to the other parts of an equation,
- how a word's meaning depends on its place and role in a sentence,
- the state and interactions inside a block of code.

The meaning of an expression can depend on the words around it.

Generalization:

- adapting a solution from one context to fit others,
- making a product more general and versatile.

In code, the tool for generalization is the **parameter**: fixed values become parameters.

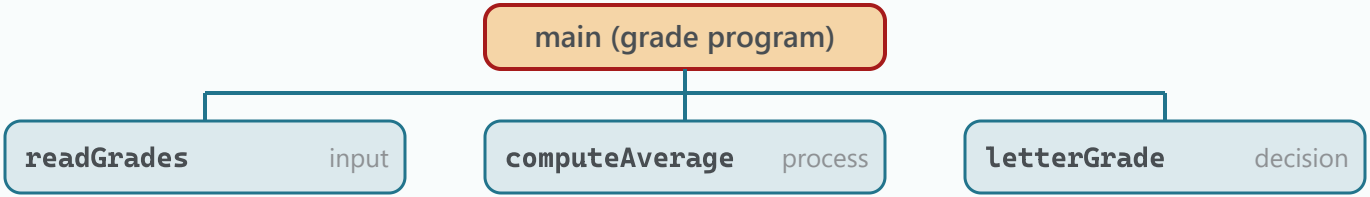
```java
public class Generalization {
    // specific: prints exactly 10 '-'
    static void drawLine() {
        System.out.println("----------");
    }
    // general: length and character as parameters
    static void drawLine(int length, char ch) {
        for (int i = 0; i < length; i++) {
            System.out.print(ch);
        }
        System.out.println();
    }
    public static void main(String[] args) {
        drawLine();
        drawLine(5, '*');
        drawLine(12, '=');
    }
}
```

OUTPUT

```
----------
*****
============
```

Decomposition

Breaking a problem into small pieces (decomposition) is a basic step in programming. Each piece becomes a method, so the code is organized, readable, **reusable** and maintainable.




```java
// readGrades and letterGrade are separate methods too
static double computeAverage(int[] grades) {
    int sum = 0;
    for (int n : grades) {
        sum += n;
    }
    return (double) sum / grades.length;
}
public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    int[] grades = readGrades(in, 3);
    double avg = computeAverage(grades);
    System.out.println("Average: " + avg + " -> " + letterGrade(avg));
}
```

OUTPUT (INPUT: 80 90 75)

```
Average:
81.66666666666667 ->
BB
```

`main` reads like a table of contents: you see what is done, and how it is done lives in the helper methods.

When to Write a Method?

A rule of thumb: **DRY** (*Don't Repeat Yourself*). The moment you feel like copying and pasting code, that code is a candidate for a method.

Repeated code

```
double avg1 = (70 + 80 + 90) / 3.0;
System.out.println(avg1);
double avg2 = (55 + 60 + 100) / 3.0;
System.out.println(avg2);
// if the formula changes, every copy
// must be fixed separately
```

With a method

```
static double average(int a, int b, int c) {
    return (a + b + c) / 3.0;
}
System.out.println(average(70, 80, 90));
System.out.println(average(55, 60, 100));
```

OUTPUT

```
80.0
71.66666666666667
```

This matters for reuse and easy maintenance. A good method does **one thing** and is easy to read.

Good Method Design

Single responsibility

Each method should do one thing. If you describe it with "and" (read *and* compute *and* print), split it.

Descriptive name

A verb that says what it does: `computeAverage`, `isPrime`, `readGrades`. `doStuff2` and similar names say nothing.

Parameters

Take the data you need as parameters; don't secretly depend on outside variables. Few parameters read better.

Return value

Return the result explicitly with `return`. A method that computes shouldn't print; let the caller do the printing.

Short and focused

Methods of a few lines that fit on one screen are easy to understand and test.

Call where needed

Call a method only where you need it; recomputing the same result again and again in a loop is wasted work.

Naming Variables

Each variable should represent one thing

- A variable should hold a single object or piece of data; using the same variable first for an age and then for a total is confusing.
- This makes code more readable and reusable, and reduces logic errors.

A good name

- Is descriptive and meaningful; reflects the nature of the data it stands for.
- Says what the variable holds and how it is used; meaningless or one-letter names hurt readability (the loop counter `i` and the mathematical `x` are exceptions).
- Java convention: **camelCase** (`studentCount`), classes capitalized (`Book`), constants `MAX_SIZE`.

```
int studentCount;       // a good name
int a;                   // a meaningless name
int age = 25;
String name = "Ahmet";
double average = 85.5;
String bookTitle;
```

Bad	Good
`int x1, x2;`	`int midtermGrade, finalGrade;`
`double d;`	`double circleArea;`
`boolean f;`	`boolean isPrime;`

Summary

Topic	Key points
Definition	`access [static] returnType name(parameters) { body }`; signature = name + parameter types
Call	arguments are **copied** into parameters (pass by value); each call pushes a new frame on the stack
return / void	`return` hands back a value and ends the method; `void` returns no value
static	static methods directly / `ClassName.method()`; otherwise `object.method()`
Overloading	same name, different parameter lists; chosen at compile time
Built-in methods	`System.out`, `Math` (`static`: `Math.sqrt(16)`); `java.lang` is automatic; others need `import`
Scope	a variable lives in the block that declares it; inner blocks see outer ones, not the reverse; fields live for the whole program
Design	abstraction (what, not how), generalization (parameters), decomposition, DRY, single responsibility

SUMMARY

Common Mistakes (1)

Calling a non-static method from main

`non-static method add(int,int) cannot be referenced from a static context`

Fix: make the method `static`, or create an object and call it on that.

Forgetting return on some paths

`missing return statement`

In an if/else if chain every case must end with a `return`; put a default `return` at the end.

Using the result of a void method

`incompatible types: void cannot be converted to int`

`int x = greet();` doesn't work: a void method produces no value.

Missing / extra arguments

`method add in class ... cannot be applied to given types; required: int,int found: int`

The number and types of arguments must match the signature.

SUMMARY

Common Mistakes (2)

Using a local variable outside its method

`cannot find symbol … variable result`

Return the value with `return`.

Same variable name in an inner block

`variable outerVar is already defined in method method()`

Expecting a swap method to work

`swap(int x, int y)` only swaps the copies; after `swap(arr[0], arr[2])` the array is unchanged. For array cells use `swap(int[] d, int i, int j)`.

Overloading by return type

`int f(int a)` and `double f(int a)` cannot coexist:
`method f(int) is already defined`
. Only the name and parameter types make up the signature.

Expecting the argument to change

`increase(num)` leaves `num` unchanged: the copy changes. Return the new value with `return`.

Forgetting Math's result type

`int k = Math.sqrt(16);` → `possible lossy conversion from double to int`
. `sqrt` and `pow` return `double`.

SUMMARY

Practice Problems (1)

```
static int f(int x) {
    x = x * 2;
    return x + 1;
}
public static void main(String[] args) {
    int a = 3;
    int b = f(a);
    System.out.println(a + " " + b);
    System.out.println(f(f(1)));
}
```

```
static int square(int n) { return n * n; }
static int cube(int n) { return n * square(n); }
// in main:
System.out.println(cube(2) + square(3));
```

1. What does the program on the left print? Why didn't `a` change?
2. What does the line on the right print? At most how many frames are on the stack while `cube(2)` runs (including main)?

▶ **Show answers**

SUMMARY

Practice Problems (2)

```
static int sum(int a, int b) { return a + b; }
static double sum(double a, double b) { return a + b; }
// in main:
System.out.println(sum(2, 3.0));
System.out.println(sum(2, 3));
System.out.println(Math.max(Math.abs(-8), Math.sqrt(49)));
```

1. In the first two `println` calls, which `sum` method runs, and what is printed?
2. Why does the last line not print `8` but `8.0`?

▶ **Show answers**

SUMMARY

Exercises

1. `boolean isEven(int num)` : return `true` if the number is even.
2. `int largest(int a, int b, int c)` : find the largest of three numbers with `Math.max` .
3. `int power(int base, int exp)` : compute the power with a loop (without `Math.pow`).
4. `int digitSum(int n)` : sum the digits of a number with a loop.
5. `double circleArea(double r)` — write it, then print the result with `printf` to 2 decimal places.
6. Overloaded `double area(double r)` (circle) and `double area(double width, double height)` (rectangle).

▶ **Answers / hints**

References

- Oracle, *The Java Tutorials*: "Defining Methods", "Passing Information to a Method or a Constructor".
- J. Gosling et al., *The Java Language Specification, Java SE 21 Edition* (Chapter 8.4 Method Declarations, 6.3 Scope).
- P. Deitel and H. Deitel, *Java: How to Program, Early Objects*, 11th ed., Pearson, 2017 (Chapter 6: Methods).
- K. Sierra and B. Bates, *Head First Java*, 3rd ed., O'Reilly, 2022.
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed., 2022.
- A. B. Downey and C. Mayfield, *Think Java*, 2nd ed., O'Reilly, 2019 (Chapter 4: Methods and Testing).

The End

Chapter 3.2: Methods

Previous: Chapter 3.1: Control Structures · Next: Chapter 3.3: Arrays