Chapter 2.1: Programming

Object Oriented Programming with Java

Programming languages | Compilation and interpretation | Paradigms

Syntax and semantics | Kinds of errors | Expressions and precedence

Variables and assignment | Static and dynamic typing | Strings and types

From algorithm to program

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_02_01_Programming.html

Learning Outcomes

Programming languages

Define a programming language; classify languages by abstraction level, application domain, execution model and paradigm.

How Java runs

Explain how source code is compiled by `javac` into bytecode and run on the JVM.

Syntax and semantics

Distinguish syntax, static semantics and semantics; recognize compile-time, run-time and logic errors.

Expressions

Evaluate an expression using operator precedence, left-to-right evaluation and integer division.

Variables and types

Declare variables, trace an assignment in two steps, and explain static vs. dynamic typing and mixed-type operations.

From algorithm to program

Tell expressions from statements, turn an algorithm into a Java program, and show why statement order matters.

INTRODUCTION

Goals of This Chapter

1

Building blocks

Learn the basic building blocks of a programming language: its **syntax** and its **semantics**.

2

Translating a recipe

Use these building blocks to turn a recipe for solving a problem into a form **the computer understands**: a program.

3

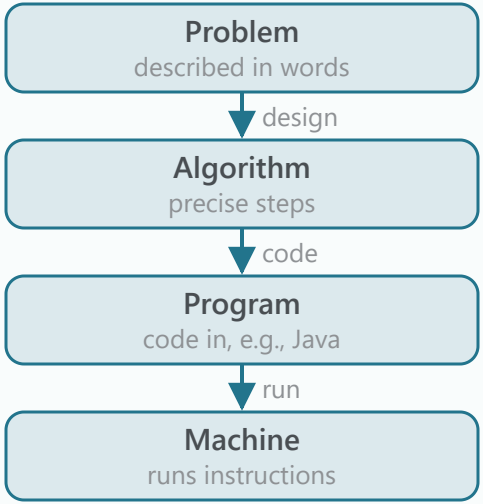
Computational thinking

Combine decomposition, pattern recognition, abstraction and algorithm design **techniques** to solve a problem.

This chapter introduces the concepts shared by every programming language, **with Java examples**.

What Is a Programming Language?

- **A programming language** is a notation for expressing the computations a machine can perform: an **artificial language**.
- It is used to write **programs** that control a machine's behavior:
  - **algorithms** stated clearly and precisely;
  - a **means of communication** between programmers.
- It gives computers, phones and embedded devices their **functionality**.
- By abstraction level: machine language and assembly are **low-level**; Java and Python are **high-level**.
- The choice of language **depends on the project's needs**.
- Examples: C, C++, Java, Python, Prolog, Haskell, Scala…



General-Purpose and Domain-Specific

- By application domain, languages are either **general-purpose** or **domain-specific**.

General-purpose

+ A broad core instruction set that supports a wide range of applications.
+ Java, C, C++, Python: desktop, web, mobile, embedded…

Domain-specific

- Focuses on one kind of application and makes those tasks easy.
- **MATLAB**: numerical computing; optimized matrix and vector operations. **SQL**: database queries.

MATLAB · MATRIX MULTIPLICATION

```
C = A * B;    % one line
```

JAVA · SAME JOB, LOOPS WRITTEN BY HAND

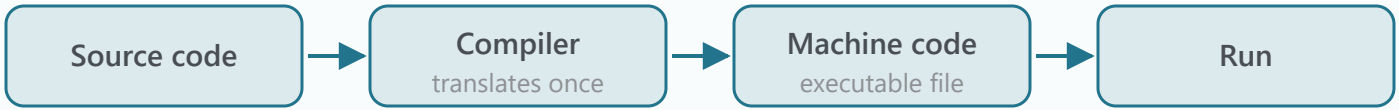
```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
            C[i][j] += A[i][k] * B[k][j];
```

PROGRAMMING LANGUAGES

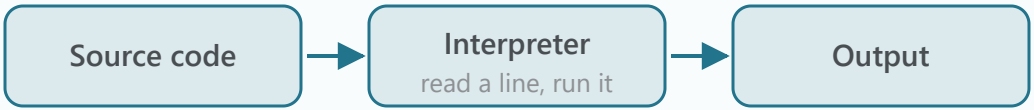
Compiled and Interpreted Languages

By how code is **executed**, languages are **compiled** or **interpreted**.

COMPILED LANGUAGE (C, C++, GO, RUST)



INTERPRETED LANGUAGE (PYTHON, JAVASCRIPT)



	Compiled	**Interpreted**
How?	The whole source is translated **before** it runs, into object / machine code.	The interpreter reads the source and executes statements **one by one**, on the fly.
Speed	Fast, optimized execution	Usually slower
Errors	Most are caught at compile time	Most surface only when that line runs

Java combines both: `javac` → **bytecode** (`.class`) → JVM (interpreter + JIT).

PROGRAMMING LANGUAGES

Programming Paradigms

A paradigm is the **basic approach** taken to solve a problem; languages can attack the same problem in different ways.

Imperative



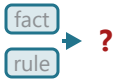
The programmer tells the machine, step by step, **how** to change its state. FORTRAN, BASIC, Pascal, C…

Functional



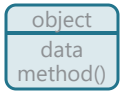
Treats computation as the **evaluation of mathematical functions** and avoids changing state. Lisp, Scheme, Haskell…

Logic



Describes the **logic** of a computation (facts and rules), not its control flow. Prolog

Object-oriented



Groups related **data and methods** into an **object**; instructions live with the state they work on. C++, Java, C#, Python…

Java is mainly **object-oriented** and imperative; since Java 8 it also has functional features (lambdas, streams). Languages that support several paradigms are called **multi-paradigm**.

Same Problem: Imperative vs. Functional

Problem: find the sum of the numbers 1 to 5.

```java
import java.util.stream.IntStream;

public class Paradigm {
    public static void main(String[] args) {
        // Imperative: change state step by step
        int sum = 0;
        for (int i = 1; i <= 5; i++) {
            sum = sum + i;
        }
        System.out.println("Imperative: " + sum);

        // Functional (Java 8+): say what you want
        int result = IntStream.rangeClosed(1, 5).sum();
        System.out.println("Functional: " + result);
    }
}
```

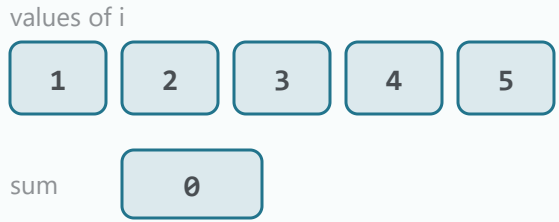
OUTPUT

```
Imperative: 15
Functional: 15
```

- **Imperative**: `sum` is **updated** at every step; you spell out “how”.
- **Functional**: “the sum of the range 1..5” **is requested**; no variable is updated.

Imperative vs. Functional: Code Trace

```
1 int sum = 0;
2 for (int i = 1; i <= 5; i++) {
3     sum = sum + i;
4 }
5 System.out.println("Imperative: " + sum);
6
7 int result = IntStream.rangeClosed(1, 5).sum();
8 System.out.println("Functional: " + result);
```



In the imperative version, the `sum` **state** variable starts at 0.

Step 1 / 10

VARIABLES

`sum = 0`

OUTPUT

Imperative vs. Functional: Code Trace

```
1 int sum = 0;
▶ for (int i = 1; i <= 5; i++) {
▶     sum = sum + i;
4 }
5 System.out.println("Imperative: " + sum);
6
7 int result = IntStream.rangeClosed(1, 5).sum();
8 System.out.println("Functional: " + result);
```

values of i

1	2	3	4	5

sum **1** **0 + 1 = 1**

`i = 1` : 1 <= 5 is true → sum = 0 + 1 = **1**. The state changes on every pass.

Step 2 / 10

VARIABLES

`sum = 1` `i = 1`

OUTPUT

Imperative vs. Functional: Code Trace

```
1 int sum = 0;
▶ for (int i = 1; i <= 5; i++) {
▶     sum = sum + i;
4 }
5 System.out.println("Imperative: " + sum);
6
7 int result = IntStream.rangeClosed(1, 5).sum();
8 System.out.println("Functional: " + result);
```

values of i



sum **3** **1 + 2 = 3**

`i = 2` : 2 <= 5 is true → sum = 1 + 2 = **3**.

Step 3 / 10

VARIABLES

`sum = 3` `i = 2`

OUTPUT

Imperative vs. Functional: Code Trace

```
1 int sum = 0;
▶ for (int i = 1; i <= 5; i++) {
▶     sum = sum + i;
4 }
5 System.out.println("Imperative: " + sum);
6
7 int result = IntStream.rangeClosed(1, 5).sum();
8 System.out.println("Functional: " + result);
```

values of i



sum **6** **3 + 3 = 6**

`i = 3` : 3 <= 5 is true → sum = 3 + 3 = **6**.

Step 4 / 10

VARIABLES

`sum = 6` `i = 3`

OUTPUT

Imperative vs. Functional: Code Trace

```
1 int sum = 0;
▶ for (int i = 1; i <= 5; i++) {
▶     sum = sum + i;
4 }
5 System.out.println("Imperative: " + sum);
6
7 int result = IntStream.rangeClosed(1, 5).sum();
8 System.out.println("Functional: " + result);
```

values of i



sum **10** **6 + 4 = 10**

`i = 4` : 4 <= 5 is true → sum = 6 + 4 = **10**.

Step 5 / 10

VARIABLES

`sum = 10` `i = 4`

OUTPUT

Imperative vs. Functional: Code Trace

```
1 int sum = 0;
▶ for (int i = 1; i <= 5; i++) {
▶     sum = sum + i;
4 }
5 System.out.println("Imperative: " + sum);
6
7 int result = IntStream.rangeClosed(1, 5).sum();
8 System.out.println("Functional: " + result);
```

values of i



sum **15** **10 + 5 = 15**

`i = 5` : 5 <= 5 is true → sum = 10 + 5 = **15**.

Step 6 / 10

VARIABLES

`sum = 15` `i = 5`

OUTPUT

Imperative vs. Functional: Code Trace

```
1 int sum = 0;
▶ for (int i = 1; i <= 5; i++) {
3     sum = sum + i;
4 }
5 System.out.println("Imperative: " + sum);
6
7 int result = IntStream.rangeClosed(1, 5).sum();
8 System.out.println("Functional: " + result);
```

values of i



sum



`i` becomes 6: 6 <= 5 is **false** → the loop ends.

Step 7 / 10

VARIABLES

`sum = 15` `i = 6`

OUTPUT

Imperative vs. Functional: Code Trace

```
1 int sum = 0;
2 for (int i = 1; i <= 5; i++) {
3     sum = sum + i;
4 }
▶ System.out.println("Imperative: " + sum);
6
7 int result = IntStream.rangeClosed(1, 5).sum();
8 System.out.println("Functional: " + result);
```

values of i



sum **15**

The result is printed: five passes add up to 15.

Step 8 / 10

VARIABLES

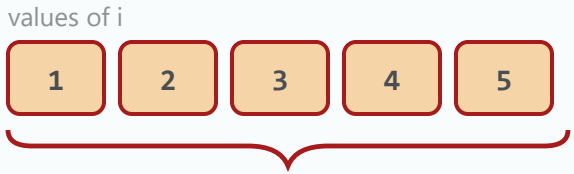
`sum = 15`

OUTPUT

```
Imperative: 15
```

Imperative vs. Functional: Code Trace

```
1 int sum = 0;
2 for (int i = 1; i <= 5; i++) {
3     sum = sum + i;
4 }
5 System.out.println("Imperative: " + sum);
6
▶ int result = IntStream.rangeClosed(1, 5).sum();
8 System.out.println("Functional: " + result);
```



Functional: `rangeClosed(1, 5)` generates the numbers 1…5, `sum()` adds them up. No variable is updated step by step.

Step 9 / 10

VARIABLES

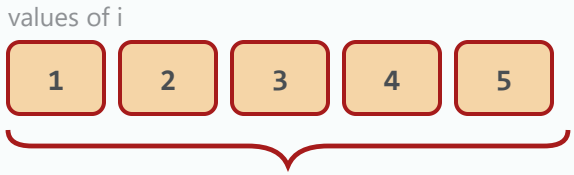
`sum = 15` `result = 15`

OUTPUT

```
Imperative: 15
```

Imperative vs. Functional: Code Trace

```
1 int sum = 0;
2 for (int i = 1; i <= 5; i++) {
3     sum = sum + i;
4 }
5 System.out.println("Imperative: " + sum);
6
7 int result = IntStream.rangeClosed(1, 5).sum();
▶ System.out.println("Functional: " + result);
```



Both give the same result; the difference is whether you write **how** to do it or **what** you want.

Step 10 / 10

VARIABLES

`sum = 15` `result = 15`

OUTPUT

```
Imperative: 15
Functional: 15
```

Logic and Object-Oriented

LOGIC · PROLOG

```
% facts
parent(ali, ayse).
parent(ayse, can).
% rule
grandparent(X, Z) :- parent(X, Y),
                     parent(Y, Z).
% query
?- grandparent(ali, can).
true.
```

Only **facts** and a **rule** are given; Prolog itself decides how to answer the query (the search order).

A `Rectangle` object holds both **data** (`width` , `height`) and the **method** (`area()`) that works on that data.

OBJECT-ORIENTED · JAVA

```
public class Rectangle {
    double width, height;           // data (state)

    Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    double area() {           // behavior
        return width * height;
    }

    public static void main(String[] args) {
        Rectangle d = new Rectangle(3, 4);
        System.out.println("Area: " + d.area());
    }
}
```

OUTPUT

```
Area: 12.0
```

Object-Oriented: Code Trace

```
1  public class Rectangle {
2      double width, height;
3      Rectangle(double width, double height) {
4          this.width = width;
5          this.height = height;
6      }
7      double area() {
8          return width * height;
9      }
10     public static void main(String[] args) {
▶           Rectangle d = new Rectangle(3, 4);
12         System.out.println("Area: " + d.area());
13     }
14 }
```

Stack Heap



`main` starts. The stack holds a `main` frame; `d` does not refer to an object yet.

Step 1 / 7

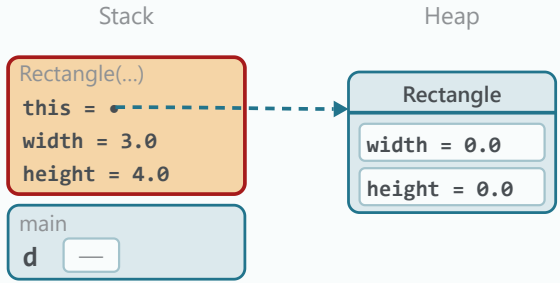
VARIABLES

`d = –`

OUTPUT

Object-Oriented: Code Trace

```
1  public class Rectangle {
2      double width, height;
▶      Rectangle(double width, double height) {
4          this.width = width;
5          this.height = height;
6      }
7      double area() {
8          return width * height;
9      }
10     public static void main(String[] args) {
11         Rectangle d = new Rectangle(3, 4);
12         System.out.println("Area: " + d.area());
13     }
14 }
```



`new` allocates the object on the heap; fields start at the default **0.0**. The constructor is called: 3 and 4 are widened to `double`.

Step 2 / 7

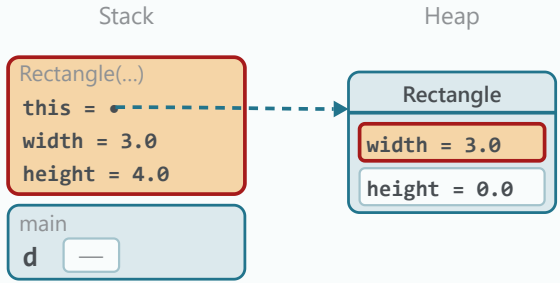
VARIABLES

`this = •` `width = 3.0` `height = 4.0`

OUTPUT

Object-Oriented: Code Trace

```
1  public class Rectangle {
2      double width, height;
3      Rectangle(double width, double height) {
▶          this.width = width;
5          this.height = height;
6      }
7      double area() {
8          return width * height;
9      }
10     public static void main(String[] args) {
11         Rectangle d = new Rectangle(3, 4);
12         System.out.println("Area: " + d.area());
13     }
14 }
```



`this.width` is the object's field; the `width` on the right is the parameter. The field becomes 3.0.

Step 3 / 7

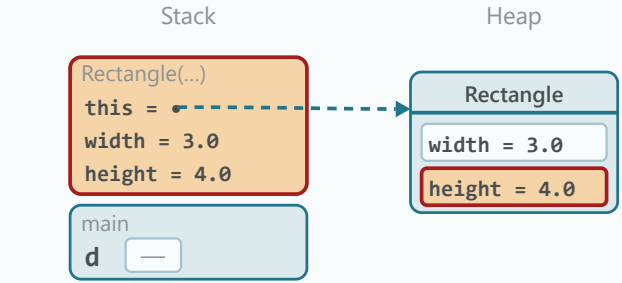
VARIABLES

`this = •` `width = 3.0` `height = 4.0`

OUTPUT

Object-Oriented: Code Trace

```
1  public class Rectangle {
2      double width, height;
3      Rectangle(double width, double height) {
4          this.width = width;
▶          this.height = height;
6      }
7      double area() {
8          return width * height;
9      }
10     public static void main(String[] args) {
11         Rectangle d = new Rectangle(3, 4);
12         System.out.println("Area: " + d.area());
13     }
14 }
```



`height` becomes 4.0. The constructor returns and its frame is popped.

Step 4 / 7

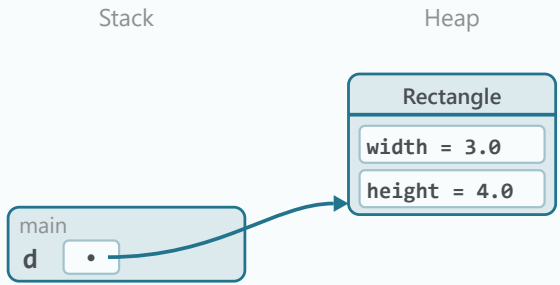
VARIABLES

`this = •` `width = 3.0` `height = 4.0`

OUTPUT

Object-Oriented: Code Trace

```
1  public class Rectangle {
2      double width, height;
3      Rectangle(double width, double height) {
4          this.width = width;
5          this.height = height;
6      }
7      double area() {
8          return width * height;
9      }
10     public static void main(String[] args) {
▶           Rectangle d = new Rectangle(3, 4);
12         System.out.println("Area: " + d.area());
13     }
14 }
```



`new` evaluates to the object's **reference**; `d` now refers to this object.

Step 5 / 7

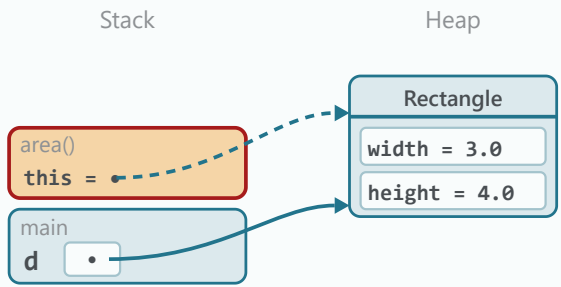
VARIABLES

`d = • (Rectangle)`

OUTPUT

Object-Oriented: Code Trace

```
1  public class Rectangle {
2      double width, height;
3      Rectangle(double width, double height) {
4          this.width = width;
5          this.height = height;
6      }
7      double area() {
▶          return width * height;
9      }
10     public static void main(String[] args) {
11         Rectangle d = new Rectangle(3, 4);
12         System.out.println("Area: " + d.area());
13     }
14 }
```



`d.area()` : `this` and `d` refer to the same object. `width * height` = 3.0 × 4.0 = **12.0** is returned.

Step 6 / 7

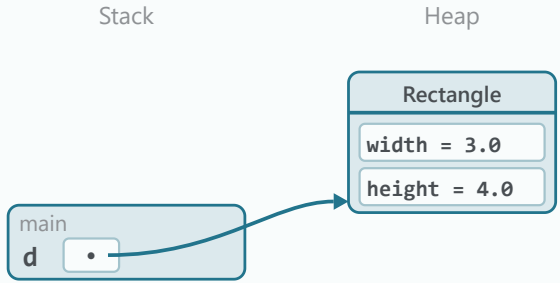
VARIABLES

`this = •`

OUTPUT

Object-Oriented: Code Trace

```
1  public class Rectangle {
2      double width, height;
3      Rectangle(double width, double height) {
4          this.width = width;
5          this.height = height;
6      }
7      double area() {
8          return width * height;
9      }
10     public static void main(String[] args) {
11         Rectangle d = new Rectangle(3, 4);
▶          System.out.println("Area: " + d.area());
13     }
14 }
```



The returned 12.0 is concatenated with the string: `Area: 12.0`. Data (width, height) and behavior (area) live in the same object.

Step 7 / 7

VARIABLES

`d = • (Rectangle)`

OUTPUT

```
Area: 12.0
```

Basic Elements and Syntax

Programming languages and natural languages have a lot in common.

	English (natural language)	Java (programming language)
Basic elements	words, punctuation marks	numbers, strings, simple operators (`+ - * /`), keywords
Syntax	"cat dog boy": valid words but no sentence structure → **invalid**	defines which sequences of characters and symbols are **well formed**

Syntax defines the **rules** of a language and affects how readable it is. The compiler rejects any sequence of symbols that breaks them:

✓ VALID

```
int x = 5;
```

✗ INVALID: WRONG ORDER

```
int = x 5;
```

```
Bad.java:3: error: not a statement
        int = x 5;
        ^
...
4 errors
```

Static Semantics

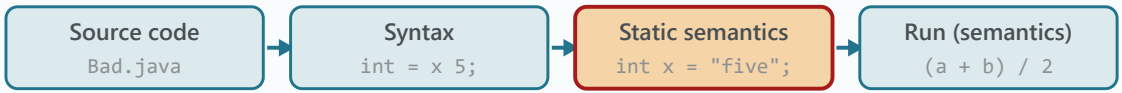
- **Static semantics**: checks whether a syntactically valid expression is **meaningful**, **before** the program runs.
- English: "I are big" matches the pattern `<noun> <verb> <noun>`, but "I" is singular and "are" plural → **invalid**.
- Java: `<value> <operator> <value>` is a valid pattern, but dividing a number by a string makes no sense → **static semantic error**.

```
double result = 2.3 / "abc";
```

```
error: bad operand types for binary operator '/'
  first type:  double
  second type: String
```

```
int x = "five";
```

```
error: incompatible types:
  String cannot be converted to int
```



Java **is statically typed**: the compiler checks the operand types of every operator, so these errors are caught before the program ever runs.

Semantics

- **Semantics**: what a syntactically valid expression **means**.
- Natural languages can be **ambiguous**: "They saw the man with the telescope."
  - They used the telescope, or
  - the man had the telescope.
- In a programming language every valid expression has **exactly one meaning**.
- But that meaning may not be what the programmer **intended**: a wrong operation gives an unexpected result.

```
int a = 7, b = 8;
int avg = (a + b) / 2;
System.out.println(avg);       // expected 7.5
System.out.println(a + b / 2);      // precedence
System.out.println((a + b) / 2.0);  // correct
```

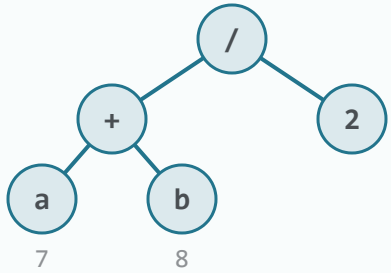
OUTPUT

```
7
11
7.5
```

The average of 7 and 8 is 7.5. The first line does **integer division**; in the second line `/` is applied before `+`. The compiler sees no error here.

Semantic Error: Code Trace

```
1 int a = 7, b = 8;
2 int avg = (a + b) / 2;
3 System.out.println(avg);        // not 7.5
4 System.out.println(a + b / 2);      //
  precedence
5 System.out.println((a + b) / 2.0);  // correct
```



`a` = 7, `b` = 8; both are `int`. The tree shows the structure of `(a + b) / 2`.

Step 1 / 7

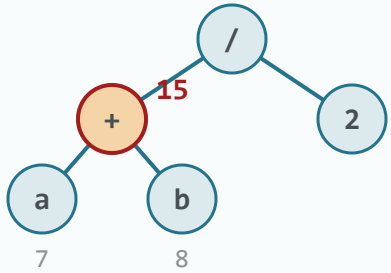
VARIABLES

`a = 7` `b = 8`

OUTPUT

Semantic Error: Code Trace

```
1 int a = 7, b = 8;
▶ int avg = (a + b) / 2;
3 System.out.println(avg);        // not 7.5
4 System.out.println(a + b / 2);      //
  precedence
5 System.out.println((a + b) / 2.0);  // correct
```



Parentheses first: `a + b` = 15 (int).

Step 2 / 7

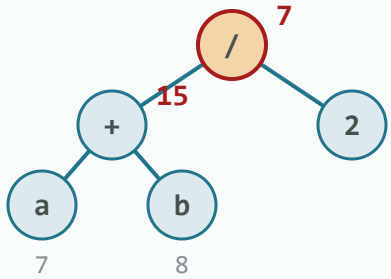
VARIABLES

`a = 7` `b = 8`

OUTPUT

Semantic Error: Code Trace

```
1 int a = 7, b = 8;
▶ int avg = (a + b) / 2;
3 System.out.println(avg);        // not 7.5
4 System.out.println(a + b / 2);      //
  precedence
5 System.out.println((a + b) / 2.0);  // correct
```



`15 / 2` : both operands are `int` → **integer division**, so the fraction is dropped: 7.

Step 3 / 7

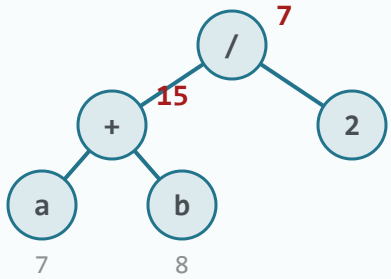
VARIABLES

`a = 7` `b = 8` `avg = 7`

OUTPUT

Semantic Error: Code Trace

```
1 int a = 7, b = 8;
2 int avg = (a + b) / 2;
▶ System.out.println(avg);        // not 7.5
4 System.out.println(a + b / 2);      //
  precedence
5 System.out.println((a + b) / 2.0);  // correct
```



7 is printed. The program ran without errors, but the meaning is not what we wanted.

Step 4 / 7

VARIABLES

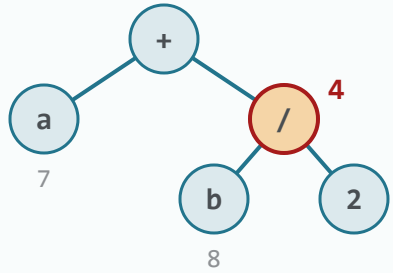
`a = 7` `b = 8` `avg = 7`

OUTPUT

```
7
```

Semantic Error: Code Trace

```
1 int a = 7, b = 8;
2 int avg = (a + b) / 2;
3 System.out.println(avg);        // not 7.5
▶ System.out.println(a + b / 2);      // precedence
5 System.out.println((a + b) / 2.0);  // correct
```



No parentheses: `/` has higher precedence than `+`. First `b / 2` = 4.

Step 5 / 7

VARIABLES

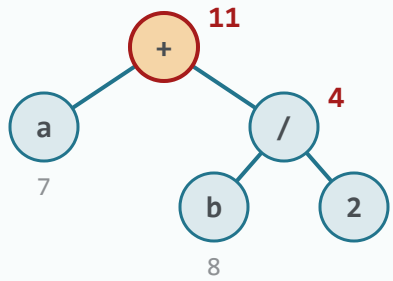
`a = 7` `b = 8` `avg = 7`

OUTPUT

```
7
```

Semantic Error: Code Trace

```
1 int a = 7, b = 8;
2 int avg = (a + b) / 2;
3 System.out.println(avg);        // not 7.5
▶ System.out.println(a + b / 2);      // precedence
5 System.out.println((a + b) / 2.0);  // correct
```



Then `7 + 4` = 11 is printed.

Step 6 / 7

VARIABLES

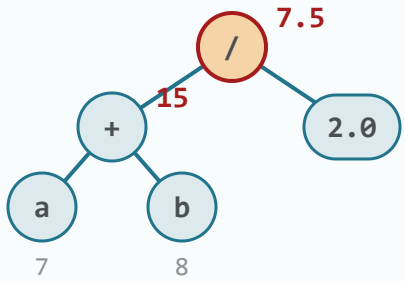
`a = 7` `b = 8` `avg = 7`

OUTPUT

```
7
11
```

Semantic Error: Code Trace

```
1 int a = 7, b = 8;
2 int avg = (a + b) / 2;
3 System.out.println(avg);        // not 7.5
4 System.out.println(a + b / 2);      //
  precedence
▶ System.out.println((a + b) / 2.0);  // correct
```



`2.0` is a `double`: 15 is widened to double too, `15 / 2.0` = **7.5**.

Step 7 / 7

VARIABLES

`a = 7` `b = 8` `avg = 7`

OUTPUT

```
7
11
7.5
```

What Can Go Wrong?

Error type	Detected when?	Java example	Symptom
Syntax	compile time (compiler)	`System.out.println("Hello")`	`';' expected`
Static semantics	compile time (type check)	`int x = "five";`	`incompatible types`
Run-time	while the program runs	`a / b` (when b is zero)	crash: `ArithmeticException`
Logic	not by the compiler; only by testing	`(a + b) / 2` for an average	wrong result, infinite loop

- **Syntax errors** break clearly defined rules, like spelling mistakes; the computer finds them **easily**.
- **Static semantic errors** are checked by language-specific rules before the program runs. In languages with weak checks, the errors that slip through make program behavior **unpredictable**.
- Even a program that is **flawless** in syntax and static semantics may not do what you expect: it can crash, loop **forever**, or give the wrong result.

Reading Error Messages

COMPILE ERROR

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello")
    }
}
```

```
Hello.java:3: error: ';' expected
        System.out.println("Hello")
                                   ^
1 error
```

- `Hello.java:3` : file and **line** number
- `error: ';' expected` : what was expected
- `^` mark: the error's **position**
- If there is an error, no `.class` file is produced; the program never runs.

RUN-TIME ERROR (EXCEPTION)

```
public class Divide {
    public static void main(String[] args) {
        int a = 10, b = 0;
        System.out.println(a / b);
    }
}
```

```
Exception in thread "main"
java.lang.ArithmeticException: / by zero
        at Divide.main(Divide.java:4)
```

- The code compiles, but at run time an **exception** is thrown.
- `/ by zero` : an integer cannot be divided by zero.
- `(Divide.java:4)` : the line where it happened.
- If the exception is not caught with `try` - `catch` , the program stops at this line.

When several errors are listed, start with the **first one**; the rest are often side effects of it.

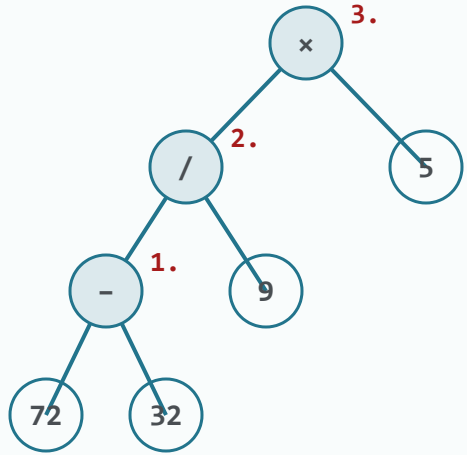
Evaluating an Expression Step by Step

The computer does not evaluate an expression in one go; it simplifies it **step by step** by precedence rules.

```
(72 - 32) / 9 * 5
```

First the **parentheses**: 72 − 32 = 40.

Step 1 / 8



Expression tree: lower operations run first; numbers give the order.

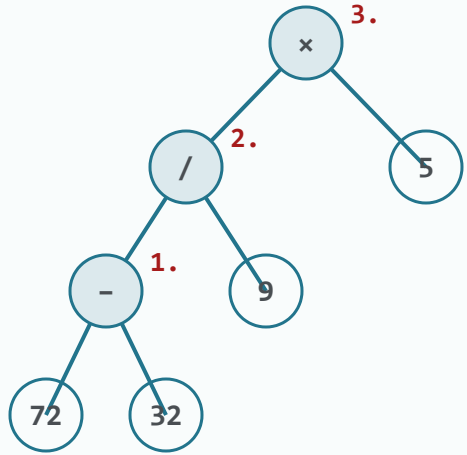
Evaluating an Expression Step by Step

The computer does not evaluate an expression in one go; it simplifies it **step by step** by precedence rules.

```
40 / 9 * 5
```

`/` and `*` have the same precedence → **left to right**: 40 / 9 first.

Step 2 / 8



Expression tree: lower operations run first; numbers give the order.

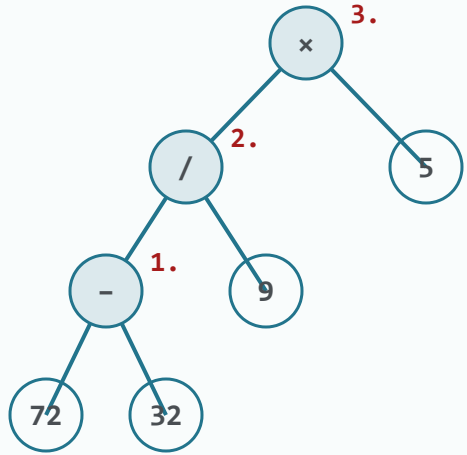
Evaluating an Expression Step by Step

The computer does not evaluate an expression in one go; it simplifies it **step by step** by precedence rules.

```
4 * 5      (40 / 9 = 4, remainder 4 dropped)
```

Both operands are `int` → **integer division**: 4, not 4.44….

Step 3 / 8



Expression tree: lower operations run first; numbers give the order.

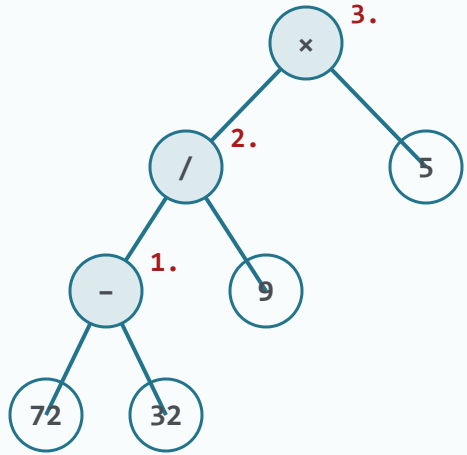
Evaluating an Expression Step by Step

The computer does not evaluate an expression in one go; it simplifies it **step by step** by precedence rules.

20

Result `20`. The mathematically expected value was 22.22…: a **semantic** (logic) error.

Step 4 / 8



Expression tree: lower operations run first; numbers give the order.

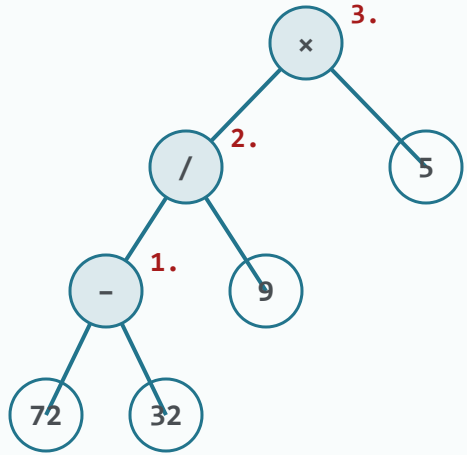
Evaluating an Expression Step by Step

The computer does not evaluate an expression in one go; it simplifies it **step by step** by precedence rules.

```
(72.0 - 32) / 9 * 5
```

With 72.0, in `double` − `int` the `int` is widened to `double` first; the result is 40.0.

Step 5 / 8



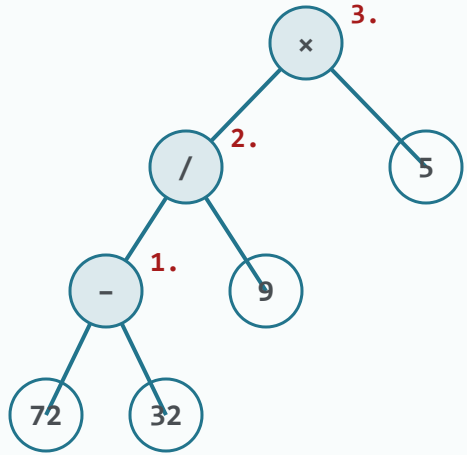
Expression tree: lower operations run first; numbers give the order.

Evaluating an Expression Step by Step

The computer does not evaluate an expression in one go; it simplifies it **step by step** by precedence rules.

```
40.0 / 9 * 5
```

40.0 / 9 is now **real division**: 4.444444444444445.



Expression tree: lower operations run first; numbers give the order.

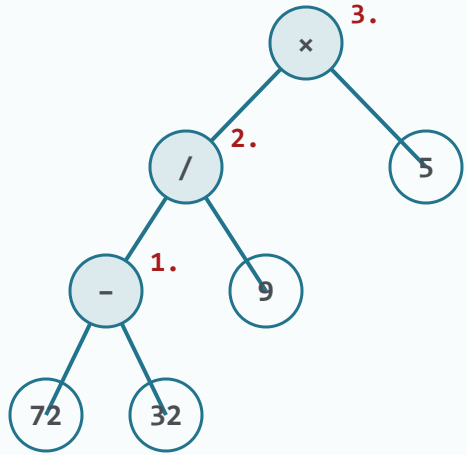
Evaluating an Expression Step by Step

The computer does not evaluate an expression in one go; it simplifies it **step by step** by precedence rules.

```
4.444444444444445 * 5
```

Multiplication: `double` × `int` → `double`.

Step 7 / 8



Expression tree: lower operations run first; numbers give the order.

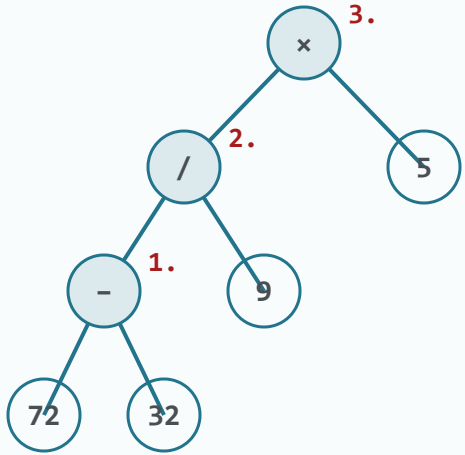
Evaluating an Expression Step by Step

The computer does not evaluate an expression in one go; it simplifies it **step by step** by precedence rules.

```
22.22222222222222
```

Correct result. If you need a fractional result, at least one operand must be a `double`.

Step 8 / 8



Expression tree: lower operations run first; numbers give the order.

Dynamic and Static Typing

Dynamically typed languages (Python, JavaScript)

- A variable's type is **not declared** in advance; it is set when a value is assigned.
- The type can change **at run time**.
- Flexible, but errors show up only **while running**.

Statically typed languages (Java, C, C#)

- A variable's type is declared **explicitly** and never changes.
- Incompatible operations, such as string with integer, are reported **at compile time**.
- Safer, and the compiler can generate faster code.

PYTHON

```
x = 5          # x is an int now
x = "five"      # now a str: no problem
print(x + 1)   # at run time:
# TypeError: can only concatenate
#   str (not "int") to str
```

JAVA

```
int x = 5;
x = "five";     // does not compile:
// error: incompatible types:
//   String cannot be converted to int
```

Java 10+: `var count = 3;` declares a variable without writing its type, but the type is still fixed at compile time (as `int`); Java stays statically typed.

Don't Compare Different Types

- Comparing different data types directly **causes errors**: you cannot compare an integer with a string.
- Languages define **rules** and **type conversions** for comparing types.

✗ DIRECT COMPARISON

```
int num = 5;
String text = "5";
boolean b = num == text;
```

```
error: bad operand types for binary operator '=='
  first type:  int
  second type: String
```

✓ CONVERT TO ONE TYPE FIRST

```
int num = 5;
String text = "5";
System.out.println(Integer.parseInt(text) == num);
System.out.println(text.equals(String.valueOf(num)));
System.out.println(text.equals(num));
```

```
true
true
false
```

`text.equals(num)` compiles but always returns `false`: a `String` is never equal to an `Integer`. Compare strings with `equals`, not `==`.

Operations on Different Types

Operations **behave differently** on different types: on strings, `+` means **concatenation**, not addition.

Expression	Java result	Type	Explanation
`3.0 + 4.0`	`7.0`	`double`	two floating-point numbers
`3 + 4`	`7`	`int`	two integers
`3 + 4.0`	`7.0`	`double`	`int` → `double` **widening** (automatic)
`"3" + "4"`	`"34"`	`String`	concatenation
`3 + "4"`	`"34"`	`String`	if one side is a string, the other is converted to a string. **Not an error** (Python raises `TypeError`)
`3 + true`	**does not compile**	—	`bad operand types for binary operator '+'`
`'a' + 1`	`98`	`int`	`char` is a numeric code ('a' = 97)

Mixing types without knowing the language's rules leads to **unexpected results** or errors. Java's rules are precise, and the compiler enforces them.

What Is a Program? Order Matters

A program is a **sequence of instructions**; the computer executes them **one at a time** and **in order**, so getting the order right matters.

```
▶ int a = 3, b = 5;
2 a = b;
3 b = a;
4 System.out.println(a + " " + b);
5
6 int c = 3, d = 5;
7 int temp = c;
8 c = d;
9 d = temp;
10 System.out.println(c + " " + d);
```



Goal: take `a` and `b` and **swap** their values.

Step 1 / 9

VARIABLES

`a = 3` `b = 5`

OUTPUT

What Is a Program? Order Matters

A program is a **sequence of instructions**; the computer executes them **one at a time** and **in order**, so getting the order right matters.

```
1  int a = 3, b = 5;
▶  a = b;
3  b = a;
4  System.out.println(a + " " + b);
5
6  int c = 3, d = 5;
7  int temp = c;
8  c = d;
9  d = temp;
10 System.out.println(c + " " + d);
```



`a = b` : 5 is written to a. Its old value 3 is **lost**!

Step 2 / 9

VARIABLES

`a = 5` `b = 5`

OUTPUT

What Is a Program? Order Matters

A program is a **sequence of instructions**; the computer executes them **one at a time** and **in order**, so getting the order right matters.

```
1  int a = 3, b = 5;
2  a = b;
▶ 3  b = a;
4  System.out.println(a + " " + b);
5
6  int c = 3, d = 5;
7  int temp = c;
8  c = d;
9  d = temp;
10 System.out.println(c + " " + d);
```



`b = a` : b gets a's **current** value (5).

Step 3 / 9

VARIABLES

`a = 5` `b = 5`

OUTPUT

What Is a Program? Order Matters

A program is a **sequence of instructions**; the computer executes them **one at a time** and **in order**, so getting the order right matters.

```
1  int a = 3, b = 5;
2  a = b;
3  b = a;
▶  System.out.println(a + " " + b);
5
6  int c = 3, d = 5;
7  int temp = c;
8  c = d;
9  d = temp;
10 System.out.println(c + " " + d);
```



Output `5 5` : the swap failed. The **order** of statements decides the result.

Step 4 / 9

VARIABLES

a = **5** b = **5**

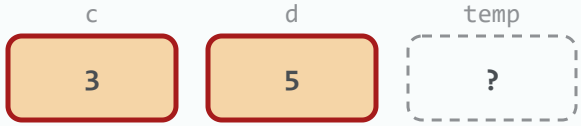
OUTPUT

```
5 5
```

What Is a Program? Order Matters

A program is a **sequence of instructions**; the computer executes them **one at a time** and **in order**, so getting the order right matters.

```
1  int a = 3, b = 5;
2  a = b;
3  b = a;
4  System.out.println(a + " " + b);
5
▶ int c = 3, d = 5;
7  int temp = c;
8  c = d;
9  d = temp;
10 System.out.println(c + " " + d);
```



The right way: a third, **temporary** variable.

Step 5 / 9

VARIABLES

`c = 3` `d = 5` `temp = –`

OUTPUT

```
5 5
```

What Is a Program? Order Matters

A program is a **sequence of instructions**; the computer executes them **one at a time** and **in order**, so getting the order right matters.

```
1  int a = 3, b = 5;
2  a = b;
3  b = a;
4  System.out.println(a + " " + b);
5
6  int c = 3, d = 5;
▶ int temp = c;
8  c = d;
9  d = temp;
10 System.out.println(c + " " + d);
```



First, c's value goes into `temp` and is **saved**.

Step 6 / 9

VARIABLES

`c = 3` `d = 5` `temp = 3`

OUTPUT

```
5 5
```

What Is a Program? Order Matters

A program is a **sequence of instructions**; the computer executes them **one at a time** and **in order**, so getting the order right matters.

```
1  int a = 3, b = 5;
2  a = b;
3  b = a;
4  System.out.println(a + " " + b);
5
6  int c = 3, d = 5;
7  int temp = c;
▶  c = d;
9  d = temp;
10 System.out.println(c + " " + d);
```



Now c can be overwritten: c = 5.

Step 7 / 9

VARIABLES

`c = 5` `d = 5` `temp = 3`

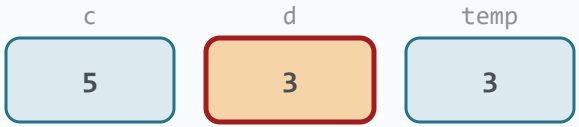
OUTPUT

```
5 5
```

What Is a Program? Order Matters

A program is a **sequence of instructions**; the computer executes them **one at a time** and **in order**, so getting the order right matters.

```
1  int a = 3, b = 5;
2  a = b;
3  b = a;
4  System.out.println(a + " " + b);
5
6  int c = 3, d = 5;
7  int temp = c;
8  c = d;
▶  d = temp;
10 System.out.println(c + " " + d);
```



d gets the saved old value of c (3).

Step 8 / 9

VARIABLES

`c = 5` `d = 3` `temp = 3`

OUTPUT

```
5 5
```

What Is a Program? Order Matters

A program is a **sequence of instructions**; the computer executes them **one at a time** and **in order**, so getting the order right matters.

```
1 int a = 3, b = 5;
2 a = b;
3 b = a;
4 System.out.println(a + " " + b);
5
6 int c = 3, d = 5;
7 int temp = c;
8 c = d;
9 d = temp;
▶ System.out.println(c + " " + d);
```



Output `5 3` : the swap worked.

Step 9 / 9

VARIABLES

`c = 5` `d = 3` `temp = 3`

OUTPUT

```
5 5
5 3
```

Expressions, Statements and Programs

Expression → value

Computes a value and returns it: `3 + 4` evaluates to 7.

Statement → effect

Has an effect: `pi = 3.14159;` assigns a value to a variable, `System.out.println(pi);` prints to the screen. In Java, statements end with `;`.

Expressions can nest

An expression can appear inside other expressions and statements: `double area = pi * r * r;`

Program = a sequence of statements

A program is many statements executed in order.

A statement cannot appear inside an expression. `println` prints (an effect) but **returns no value** (`void`):

```
double d = 3 + System.out.println(pi);
```

```
error: 'void' type not allowed here
```

```
3 + 4;      // value, no effect
```

```
error: not a statement
```

In Java, assignment also produces a value: `int y = (x = 4) + 1;` → x = 4, y = 5. It compiles, but it is hard to read, so avoid it.

Popular Programming Languages

- Programming languages offer **different strengths** for different tasks.
- Programmers choose **the best-fitting language** for the project's needs.
- Language popularity is ranked every year by surveys and indexes (Stack Overflow survey, TIOBE, IEEE Spectrum).
- The table shows some of the leading languages in 2023.

Language	Where it shines
Python	data analysis, AI, web development
Java	large-scale enterprise applications, Android
JavaScript	web browsers, web development
C#	Windows applications, game development
C++	games, systems programming, performance-critical software
Rust	safe and fast systems programming
Swift	app development for Apple platforms
Kotlin	Android app development (runs on the JVM)
Go (Golang)	simple and efficient; network and cloud applications

Summary

Topic	Key points
Kinds of languages	low / high level; general-purpose / domain-specific; compiled / interpreted
Java	`javac` → bytecode (`.class`) → JVM (interpreter + JIT); “write once, run anywhere”
Paradigms	imperative, functional, logic, object-oriented; Java: object-oriented + imperative
Syntax / semantics	syntax: form; static semantics: type checks at compile time; semantics: one precise meaning, but maybe not the one you intended
Kinds of errors	syntax and static semantics (compile time), run time (exceptions), logic (wrong result)
Expressions	parentheses → `* / %` → `+ -`, left to right; `int / int` is integer division; for powers use `Math.pow`
Variables and assignment	type + name; the right side is evaluated first, then stored; `=` is assignment, `==` is comparison
Types	Java is statically typed; `int` + `double` → `double` ; string + anything → concatenation
Program	algorithm first, then code; expressions produce values, statements have effects; statements run in order

SUMMARY

Common Mistakes

Forgetting the semicolon

`System.out.println("Hello")` → `';' expected`. javac's `^` marks the **end** of the line missing the ;.

Mixing up = and ==

`if (x = 100)` → `int cannot be converted to boolean`. Comparison is `==`.

Integer division

`(a + b) / 2` gives 7 as the average of 7 and 8. Real division needs `/ 2.0`.

** or ^ for powers

`2 ** 3` → `illegal start of expression`; `2 ^ 3` compiles but is XOR (1). Use `Math.pow(2, 3)`.

String + number order

`"Total: " + 2 + 3` → `Total: 23`. To add first: `"Total: " + (2 + 3)`.

Upper/lower case

`Num` declared, then `num` used → `cannot find symbol`. Java is case-sensitive.

Using a variable before assigning it

`int s; System.out.println(s);` → `variable s might not have been initialized`.

Decimal separator

On a computer with a Turkish locale, `nextDouble()` expects input like `7,5`; typing `7.5` throws `InputMismatchException`.

SUMMARY

Practice Problems

1. Evaluate: (a) `10 + 6 / 4 * 2` (b) `(10 + 6) / 4 * 2` (c) `10 + 6 / 4.0 * 2`
2. What do these print? `System.out.println("Total: " + 2 + 3);` and `System.out.println(2 + 3 + " total");`
3. After this code runs, what are `a` and `b`? `int a = 4; int b = a * 3; a = b - a; b = b / a;`
4. Name the error type: (a) `int n = 4.5;` (b) `System.out.println("Hello";` (c) for the area of a circle `area = 2 * pi * r;`
5. `double c = (f - 32) * 5 / 9;` — what is the result for `f = 100`? What if you declare `f` and `c` `int` instead?

▶ **Show answers**

SUMMARY

Exercises

1. Give the value and type of: `7 % 3`, `-7 / 2`, `1 / 3 * 3`, `1.0 / 3 * 3`, `Math.pow(2, 10)`.
2. Which of these are valid variable names? `total2`, `2total`, `_counter`, `student count`, `$price`, `int`, `averageGrade`
3. Write the algorithm, then the Java code, for a program that reads a Fahrenheit value from the user and converts it to Celsius.
4. Classify each as an expression or a statement: `x * 2`, `x = 5;`, `x > 3`, `System.out.println(x);`
5. `5 / 9 * (f - 32)` with `int f = 100` gives 0 — why? Fix it in two different ways.

▶ **Answers / hints**

References

- Oracle, *The Java Tutorials*: “Language Basics” (variables, operators, expressions, statements and blocks).
- J. Gosling et al., *The Java Language Specification, Java SE 21 Edition*, Oracle, 2023 (Chapter 3: Lexical Structure, Chapter 15: Expressions).
- P. Deitel and H. Deitel, *Java: How to Program, Early Objects*, 11th ed., Pearson, 2017 (Chapters 1–2).
- A. B. Downey and C. Mayfield, *Think Java: How to Think Like a Computer Scientist*, 2nd ed., O'Reilly, 2019 (Chapters 1–2).
- K. Sierra and B. Bates, *Head First Java*, 3rd ed., O'Reilly, 2022.
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed., Pearson, 2022 (Chapters 2–3).
- J. Guttag, *Introduction to Computation and Programming Using Python*, 3rd ed., MIT Press, 2021 (Chapter 2: syntax, static semantics and semantics).

The End

Chapter 2.1: Programming

Previous: Chapter 1.4: Flowcharts · Next: Chapter 2.2: Syntax Rules