Chapter 4.3: Introduction to Programming

Computer Fundamentals and Introduction to Programming

Languages | Translation | Problem solving | Software process | Basic constructs

Sercan KÜLCÜ · Giresun University

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/computer/slides/Chapter_04_03_Introduction_to_Programming.html

Learning Outcomes

Concepts

Distinguish source code, machine language and programming languages.

Execution

Explain compiled, interpreted and hybrid execution.

Problem solving

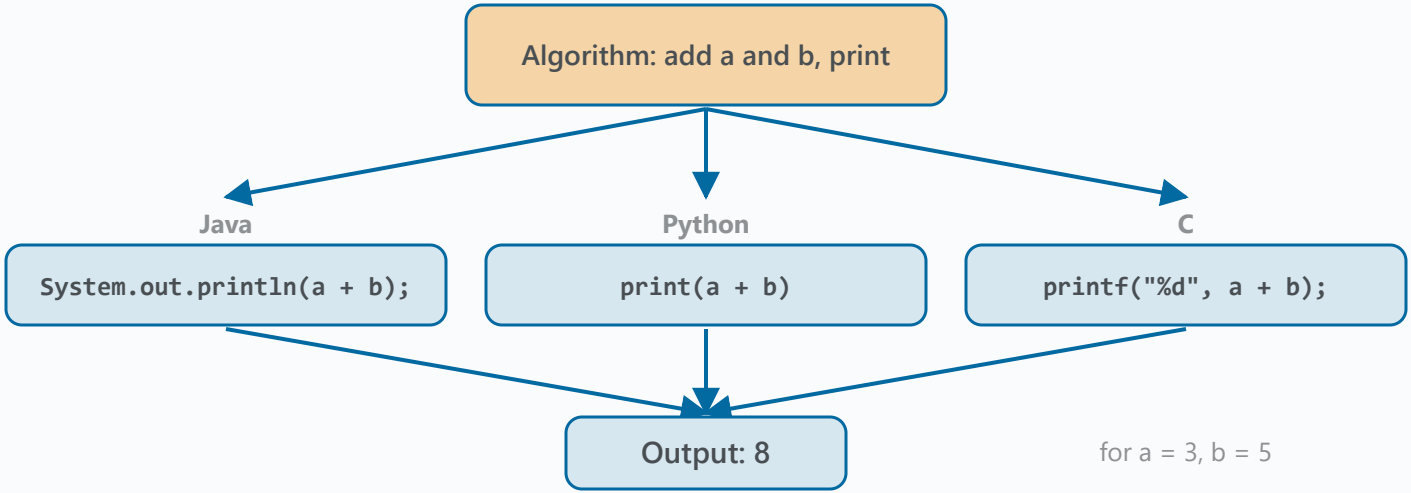
Follow the path from algorithm to program.

Constructs

Recognize variables, decisions, loops, functions and arrays.

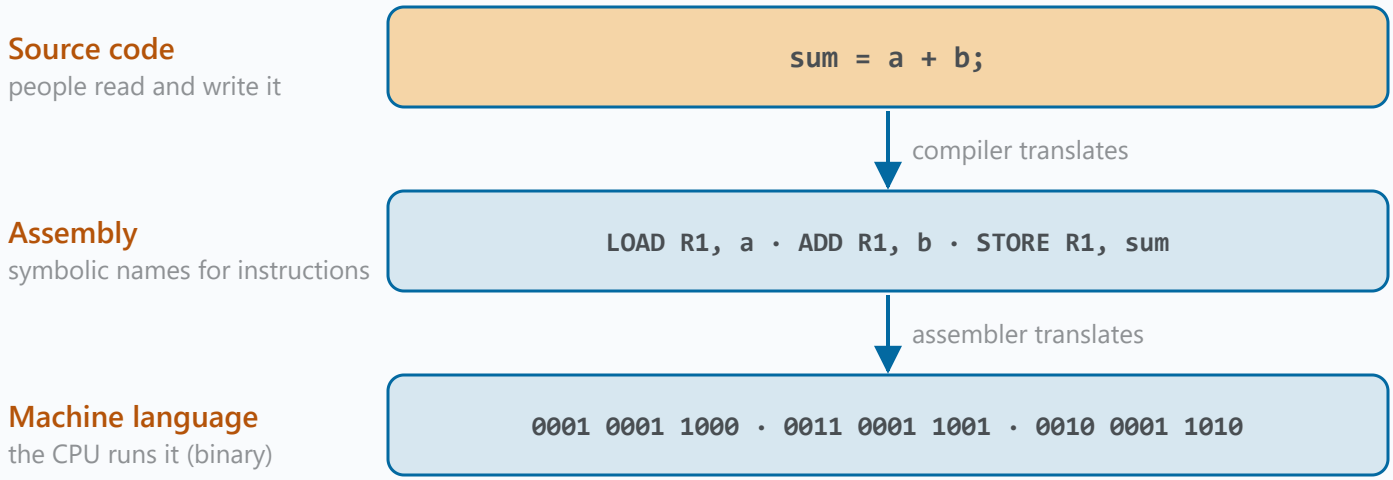
What Is a Programming Language?

- A formal language that lets people express the steps a computer should carry out.
- **Syntax** defines which constructs can be written; **semantics** defines what they do.
- The same problem can be solved in different languages; the choice depends on requirements and environment.



BASIC CONCEPTS

Source Code and Machine Language

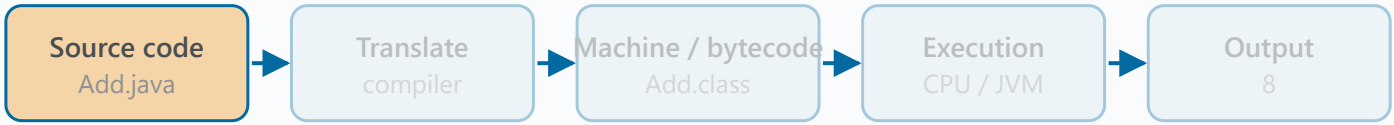


Illustrative machine code (real encodings depend on the CPU).

- Source code is the program text a developer can read and edit; machine language is the binary instructions the CPU executes.
- A compiler or interpreter bridges the gap: the programming language and the processor's language are not the same thing.

EXECUTION · STEP BY STEP

How Does a Program Run?

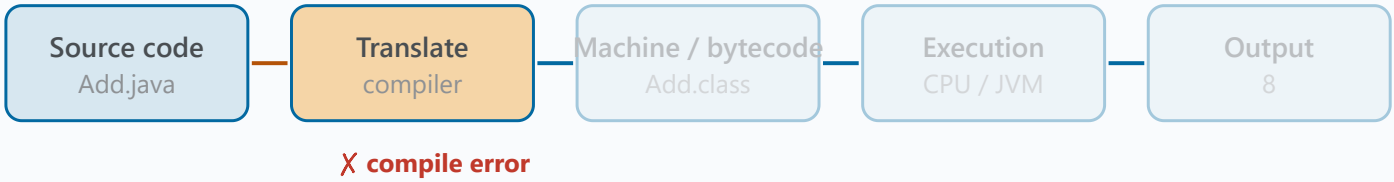


The programmer writes the **source code** in a text file. The computer cannot run this text directly.

Step 1 / 5

Kinds of errors: **compile-time** (a syntax rule), **runtime** (during execution), **logic** (a wrong result).

How Does a Program Run?

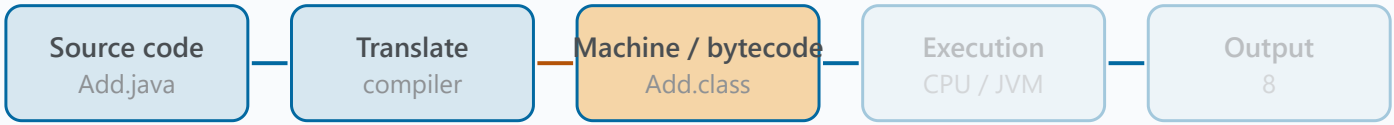


Translation: the compiler checks the syntax. A forgotten semicolon causes a **compile error** here, and the program never runs.

Kinds of errors: **compile-time** (a syntax rule), **runtime** (during execution), **logic** (a wrong result).

EXECUTION · STEP BY STEP

How Does a Program Run?

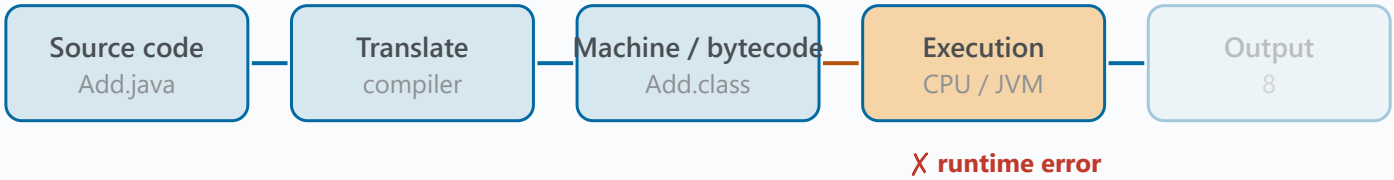


If there are no errors, **machine code** is produced, or, as in Java, **intermediate code** (bytecode).

Step 3 / 5

Kinds of errors: **compile-time** (a syntax rule), **runtime** (during execution), **logic** (a wrong result).

How Does a Program Run?

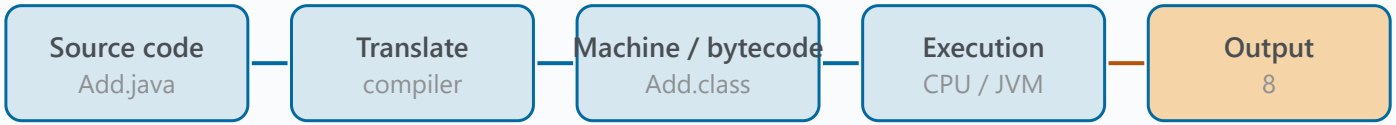


Execution: the instructions run on the CPU (in Java, via the JVM). **Runtime errors** such as division by zero show up here.

Kinds of errors: **compile-time** (a syntax rule), **runtime** (during execution), **logic** (a wrong result).

EXECUTION · STEP BY STEP

How Does a Program Run?



The **output** depends on the input (a = 3, b = 5) and on the program's logic. With a logic error, the program runs but gives a wrong result.

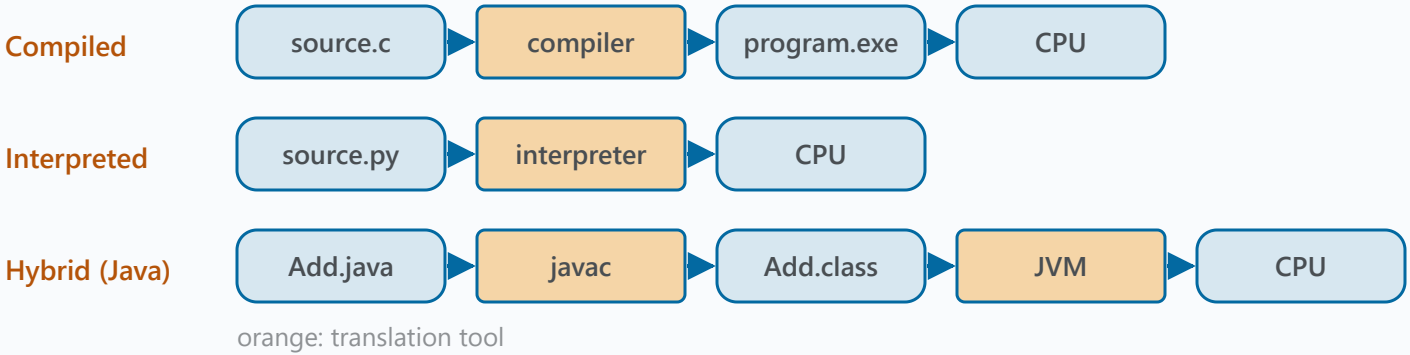
Step 5 / 5

Kinds of errors: **compile-time** (a syntax rule), **runtime** (during execution), **logic** (a wrong result).

EXECUTION

Ways to Run a Program

Approach	How it works	Example
Compiled	Translated ahead of time into the target machine code	C, C++, Rust
Interpreted	Statements are evaluated while the program runs	Python, JavaScript
Hybrid	Intermediate code runs on a virtual machine	Java, C#



orange: translation tool

EXECUTION · STEP BY STEP

Compiler and Interpreter Step by Step

Compiler: translate everything first, then run

1	`print "Hello"`	translating
2	`x ← 10`	translating
3	`print x / 0`	translating

console



The compiler reads the **whole** program from start to end; no line has run yet.

Step 1 / 7

A compiler catches syntax errors **before** the program runs; an interpreter starts at once and (in simple interpreters) sees an error only on reaching that line. Modern interpreters such as Python and JavaScript parse the whole file first and report syntax errors up front; Python also compiles to bytecode.

Compiler and Interpreter Step by Step

Compiler: translate everything first, then run

```
1  print "Hello"   ✓
2  x ← 10         ✓
3  print x / 0    ✓
```

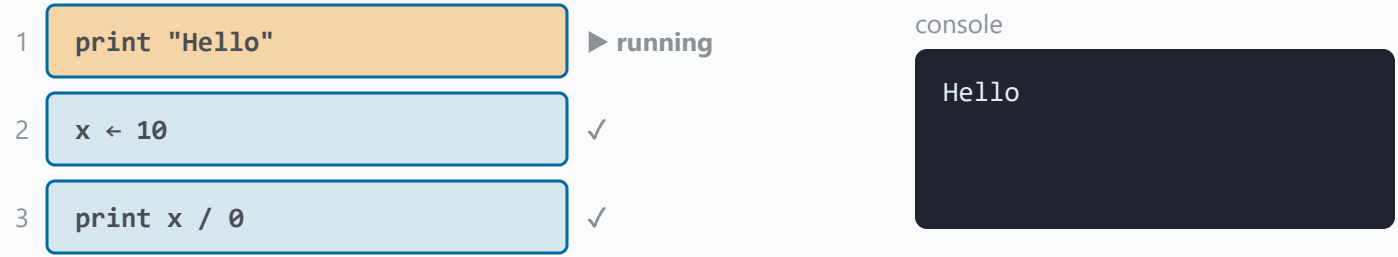
console

Dividing by zero on line 3 is not a syntax error; translation succeeds and an executable file is produced.

A compiler catches syntax errors **before** the program runs; an interpreter starts at once and (in simple interpreters) sees an error only on reaching that line. Modern interpreters such as Python and JavaScript parse the whole file first and report syntax errors up front; Python also compiles to bytecode.

Compiler and Interpreter Step by Step

Compiler: translate everything first, then run



Now execution starts: line 1 → output **Hello**.

A compiler catches syntax errors **before** the program runs; an interpreter starts at once and (in simple interpreters) sees an error only on reaching that line. Modern interpreters such as Python and JavaScript parse the whole file first and report syntax errors up front; Python also compiles to bytecode.

Compiler and Interpreter Step by Step

Compiler: translate everything first, then run

```
1  print "Hello"      ✓
2  x ← 10             ✓
3  print x / 0        ✗ error
```

console

```
Hello
ERROR
```

Line 3 causes a **runtime error** (division by zero), the kind of error a compiler cannot catch.

A compiler catches syntax errors **before** the program runs; an interpreter starts at once and (in simple interpreters) sees an error only on reaching that line. Modern interpreters such as Python and JavaScript parse the whole file first and report syntax errors up front; Python also compiles to bytecode.

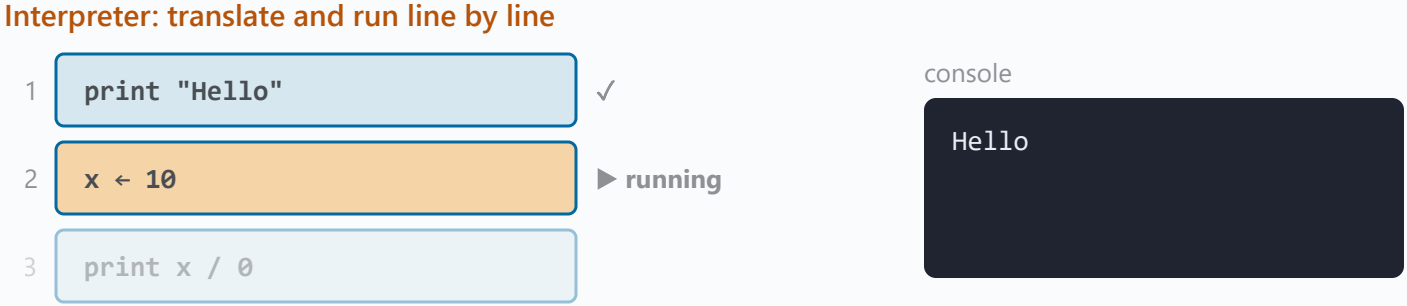
Compiler and Interpreter Step by Step



An interpreter works in a **translate-run** loop: line 1 is translated and run at once → Hello.

A compiler catches syntax errors **before** the program runs; an interpreter starts at once and (in simple interpreters) sees an error only on reaching that line. Modern interpreters such as Python and JavaScript parse the whole file first and report syntax errors up front; Python also compiles to bytecode.

Compiler and Interpreter Step by Step

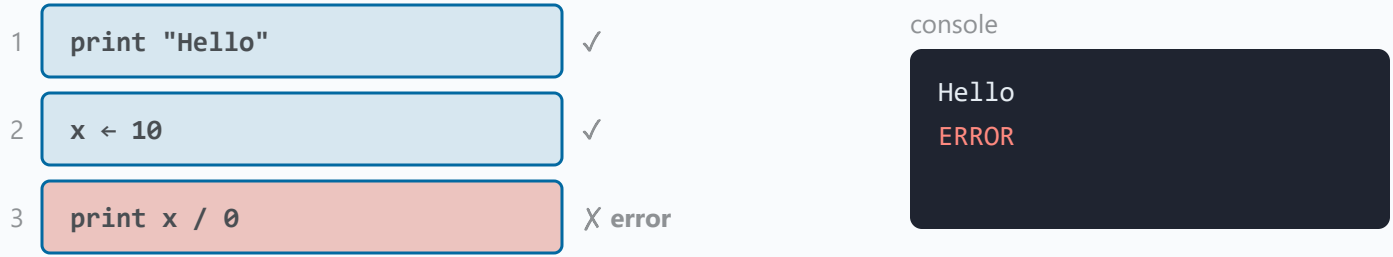


Line 2 is translated and run: **x = 10**. Line 3 has not even been read.

A compiler catches syntax errors **before** the program runs; an interpreter starts at once and (in simple interpreters) sees an error only on reaching that line. Modern interpreters such as Python and JavaScript parse the whole file first and report syntax errors up front; Python also compiles to bytecode.

Compiler and Interpreter Step by Step

Interpreter: translate and run line by line



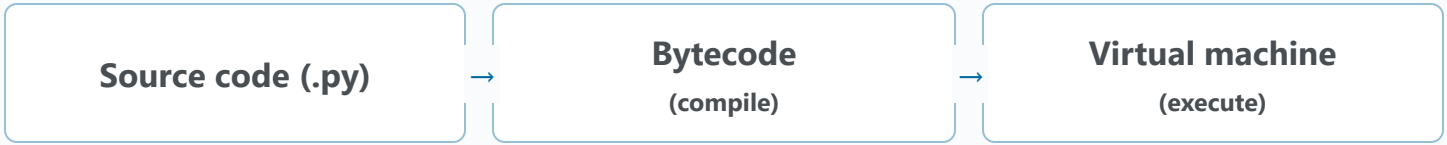
At line 3, an error. The earlier lines **have already run**. A simple line-by-line interpreter would notice even a syntax error only on reaching that line.

Step 7 / 7

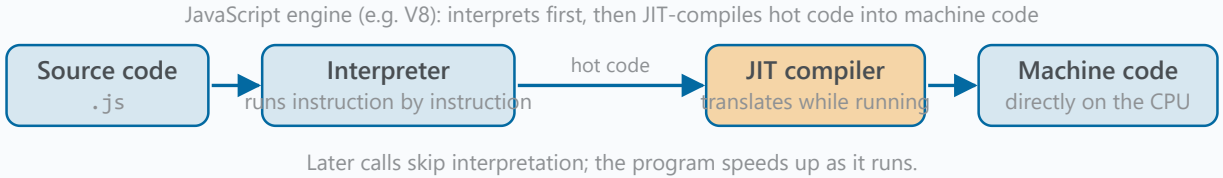
A compiler catches syntax errors **before** the program runs; an interpreter starts at once and (in simple interpreters) sees an error only on reaching that line. Modern interpreters such as Python and JavaScript parse the whole file first and report syntax errors up front; Python also compiles to bytecode.

EXECUTION

Why Is Python Called “Interpreted”?



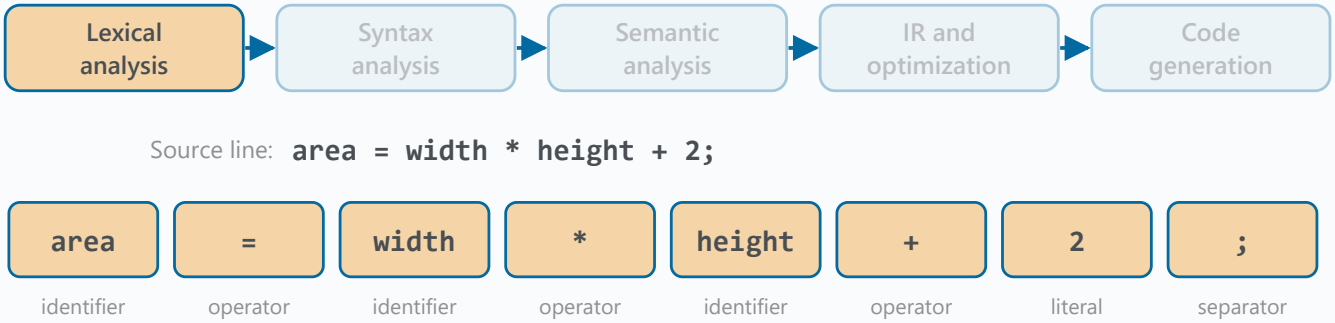
- **CPython** first compiles the source to **bytecode**, then executes it instruction by instruction on a virtual machine; no machine-code file is produced.
- The “interpreted” label comes from this way of executing; the compile step is hidden from the programmer.
- Modern **JavaScript** engines (e.g. V8) use a **JIT**: they translate frequently run code into machine code while the program runs.



Compiled versus interpreted is a property of the **implementation**, not the language; in practice most systems combine both.

EXECUTION · STEP BY STEP

Phases of a Compiler

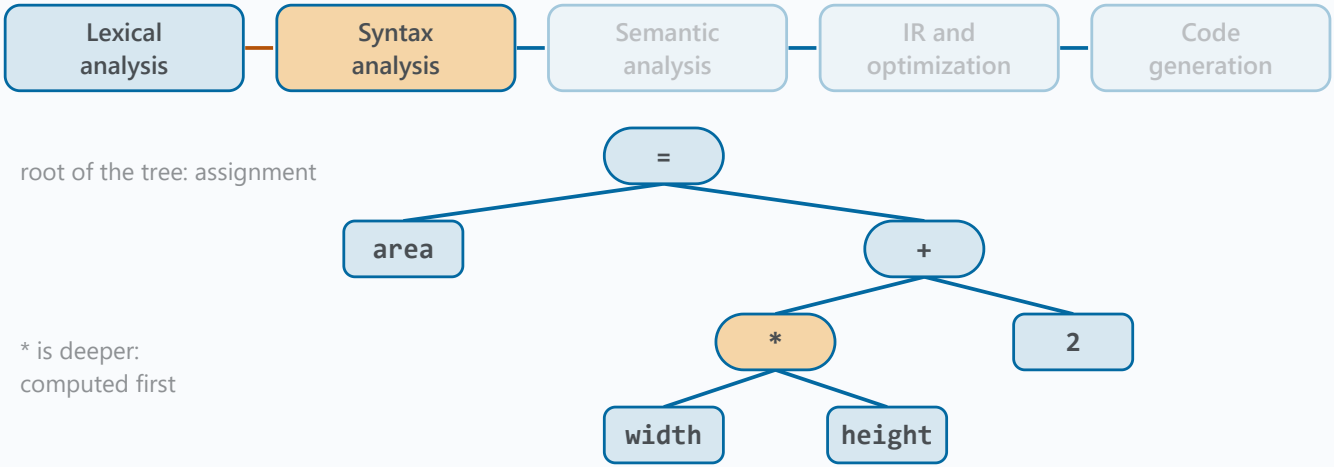


Lexical analysis: the character stream is split into meaningful pieces called **tokens**; whitespace and comments are dropped. A character the language lacks, such as `#`, or an unclosed string is caught here.

Step 1 / 5

The first three phases look for errors and report them with a line number; the last two produce the code that runs. No phase catches value-dependent errors such as division by zero.

Phases of a Compiler

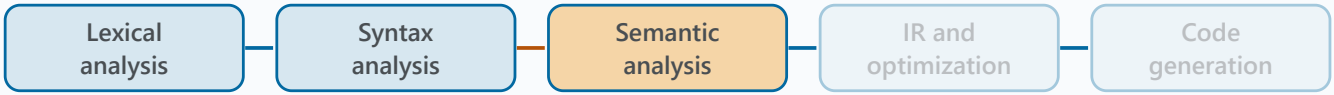


Syntax analysis: the tokens are arranged into a **syntax tree** by the language's grammar. A missing `;` or an unclosed parenthesis is caught at this phase.

Step 2 / 5

The first three phases look for errors and report them with a line number; the last two produce the code that runs. No phase catches value-dependent errors such as division by zero.

Phases of a Compiler



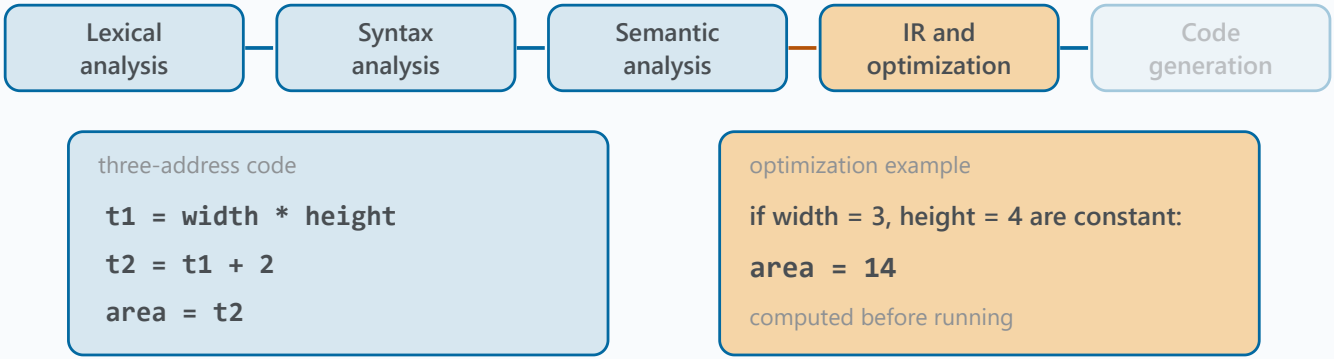

```
width                → declared · type int ✓
height               → declared · type int ✓
width * height + 2   → int × int + int → int ✓
area ← int           → area is int: types match ✓
```

if height were never declared → error: cannot find symbol

Semantic analysis: checks that names are declared and types are compatible. Undefined variables and type mismatches are found here.

The first three phases look for errors and report them with a line number; the last two produce the code that runs. No phase catches value-dependent errors such as division by zero.

Phases of a Compiler

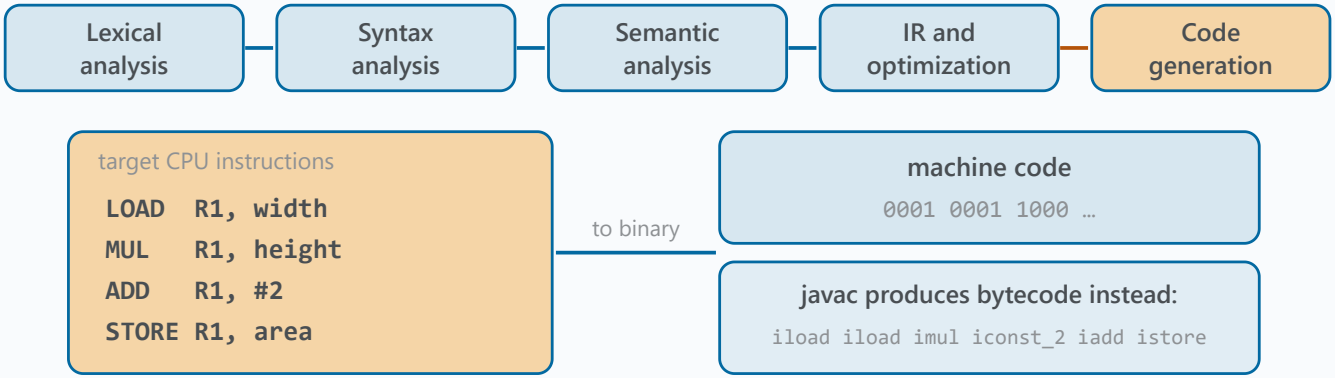


constant folding: operations whose result never changes are done at compile time

Intermediate code and optimization: the expression is broken into simple instructions that each do one operation; unnecessary work is removed and constant expressions are precomputed.

The first three phases look for errors and report them with a line number; the last two produce the code that runs. No phase catches value-dependent errors such as division by zero.

Phases of a Compiler

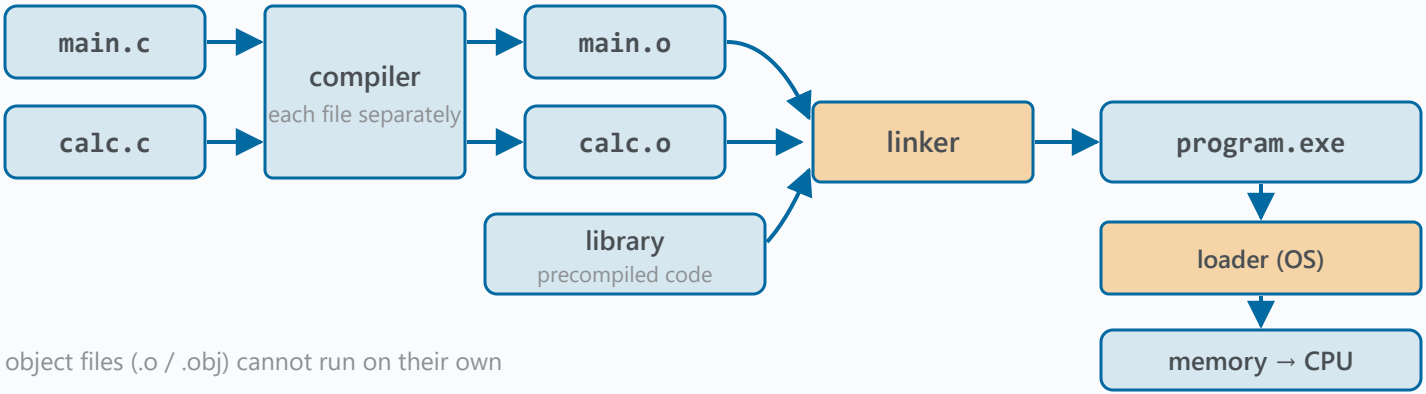


Code generation: instructions for the target CPU are chosen and encoded in binary. The Java compiler's target is not a real CPU but the JVM.

The first three phases look for errors and report them with a line number; the last two produce the code that runs. No phase catches value-dependent errors such as division by zero.

EXECUTION

Linker and Loader



object files (.o / .obj) cannot run on their own

- Large programs consist of many source files; the compiler turns each into a separate **object file**.
- The **linker** combines the object files and any library code used into one executable. If a called function is defined nowhere, it reports a **link error**.
- The **loader** is part of the operating system: it brings the file into memory, attaches shared libraries (.dll, .so) and starts execution at the first instruction.
- Java has no separate link step: the JVM's **class loader** loads .class files as they are needed.

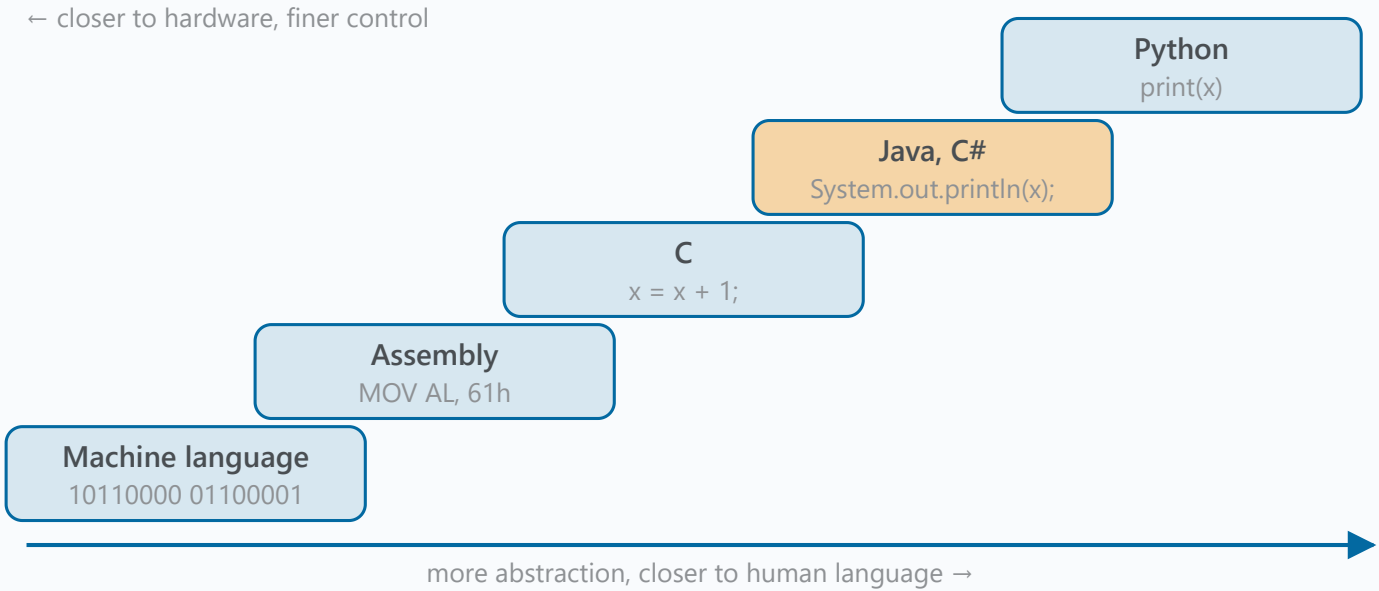
EXECUTION

Compile or Interpret?

Criterion	Compiled (C)	Interpreted (Python)	Hybrid (Java)
Speed	Usually fastest: direct machine code	Slower: statements are decoded at run time	Close to compiled, thanks to the JIT
Error reporting	Syntax and type errors before running	Type errors usually only when that line runs	Most at compile time, the rest while running
Distribution	A separate file for each OS and CPU	Source code + an interpreter installed on the target	One .class / .jar + a JVM on every platform
Development cycle	Recompile after every change	Save and run at once	Compile, then run
Startup	Starts at once, no extra software	The interpreter loads first	The JVM starts first

It is a trade-off: speed and early error checking on one side, quick experiments and flexibility on the other. That is why one language can have both a compiler and an interpreter.

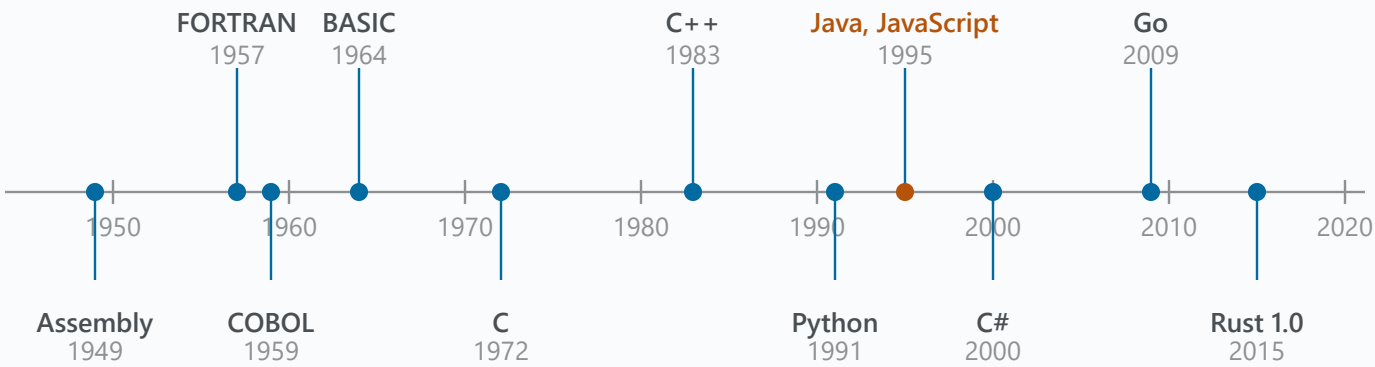
Language Levels



- **Low-level** languages are close to the hardware; **high-level** languages let you express a problem in human terms.
- Level describes the **degree of abstraction**, not whether a language is good or bad.

LANGUAGES

The Evolution of Programming Languages



1940–1970

Machine language and assembly; the first high-level languages, FORTRAN and COBOL.

1970–1990

C brought structured, portable systems programming; C++ added the object-oriented approach.

After 1990

Python, Java, JavaScript, C#: the web, managed runtimes, open-source ecosystems.

Application Areas

Web

Clients, servers and web services.

Mobile

Phone and tablet apps.

Data and AI

Analysis, modeling and automation.

Systems

Operating systems, drivers and embedded software.

Scientific computing

Simulation and engineering solutions.

Games

Game engines, graphics and interaction.

LANGUAGES

Kinds of Languages

General-purpose

Offer variables, decisions, loops and functions for any problem: Java, C, Python.

```
for (int i = 0; i < 3; i++)
```

Domain-specific

Focus on one job: SQL queries a database, an Excel formula computes in a cell.

```
SELECT name FROM student WHERE avg >= 60;
```

Scripting languages

Automate repetitive tasks, usually interpreted: Python, JavaScript, Bash, PowerShell.

```
Copy-Item *.txt backup\
```

Markup languages

HTML, XML and Markdown describe the structure of content; with no decisions or loops, they are not programming languages.

```
<h1>Title</h1>
```

In block-based languages such as Scratch, you drag blocks to build programs; since syntax errors are impossible, they are used to teach programming.

Programming Paradigms

- The imperative approach states step by step how the work is done.
- The object-oriented approach combines data and behavior in objects.
- The functional approach emphasizes transformations and immutable data.
- Java can combine several approaches.

One task, three approaches: the sum of the array grades = {70, 90, 80} (240)

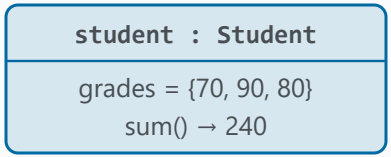
Imperative

```
int sum = 0;
for (int n : grades)
    sum += n;
```

sum: 0 → 70 → 160 → 240

state changes step by step

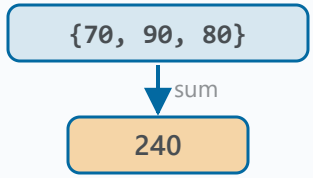
Object-oriented



`student.sum()`

data and behavior together

Functional

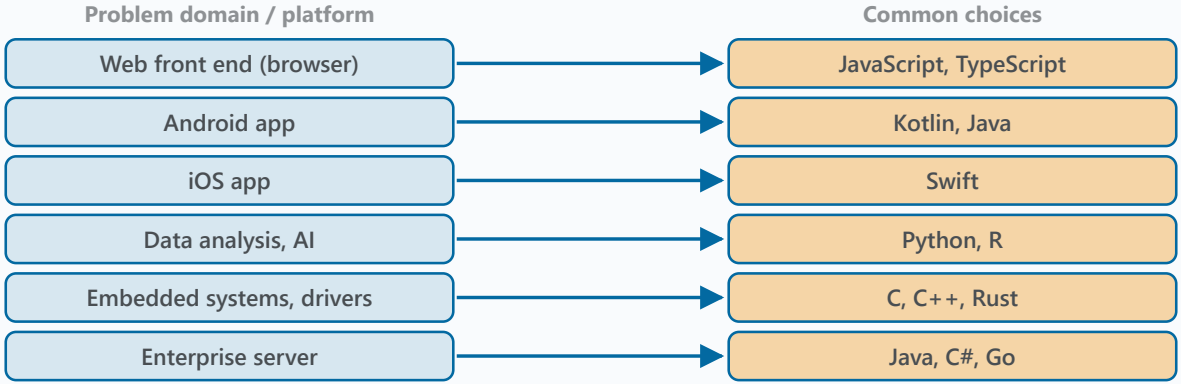


`Arrays.stream(grades).sum()`

a transformation; the array is unchanged

Choosing a Language

- Consider the problem domain and the target platform.
- Weigh the library ecosystem, the community and the maintenance cost.
- Think about performance, security, team experience and deployment together.



The final call depends on the library ecosystem, performance, security, team experience and maintenance cost.

Popularity alone is not enough to pick the right language.

Static and Dynamic Typing

Java: static

```
int x = 5;
x = "five";   // won't compile
```

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

Python: dynamic

```
x = 5
x = "five"    # fine
print("a" + 1)
```

```
TypeError: can only concatenate
str (not "int") to str
```

- With **static typing**, a variable's type is declared in the source and checked **at compile time**: Java, C, C#. Errors are found before the program ever runs.
- With **dynamic typing**, the type travels with the value and is checked **at run time**: Python, JavaScript. The same name can hold a number, then text; Python reports the error only when that line runs.
- Some languages convert types silently: in JavaScript `"5" * 2` → 10, but `"5" + 2` → "52".

From Pseudocode to Languages

Pseudocode	Java	Python
`read x`	`x = in.nextInt();`	`x = int(input())`
`print "Sum: ", t`	`System.out.println("Sum: " + t);`	`print("Sum:", t)`
`x ← x + 1`	`x = x + 1;`	`x = x + 1`
`if x > 0 then … else …`	`if (x > 0) { … } else { … }`	`if x > 0: … else: …`
`for each i ← 1 … n`	`for (int i = 1; i <= n; i++)`	`for i in range(1, n + 1):`
`while condition is true, repeat`	`while (cond) { … }`	`while cond:`

- Java marks blocks with braces, Python with **indentation**; in Python, wrong indentation changes the program's meaning.
- Java ends every statement with `;` and requires variable types to be declared; Python requires neither.
- Pseudocode uses ← for assignment so it is not confused with equality. In Java, `in` is a `Scanner` object reading from the keyboard.

PROBLEM SOLVING · STEP BY STEP

From Problem to Program



First pin down **what** is asked: the area of a rectangle. Write the input, output and constraint.

Step 1 / 5

- Each stage's output is the next stage's input.
- A badly defined problem can become a program that works correctly but does the wrong job.

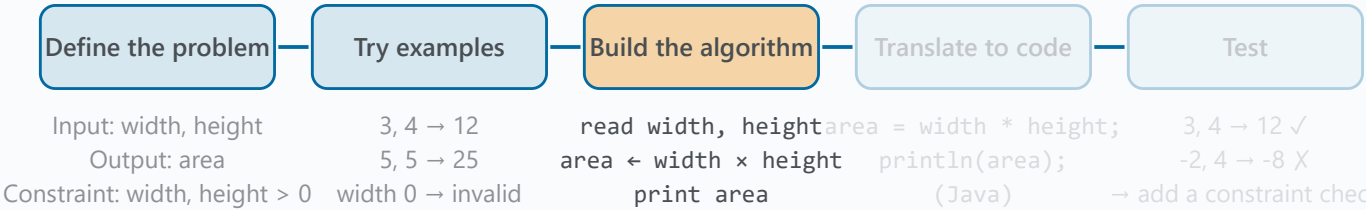
From Problem to Program



Solve a few **examples** by hand; consider the boundary case (0) too.

- Each stage's output is the next stage's input.
- A badly defined problem can become a program that works correctly but does the wrong job.

From Problem to Program

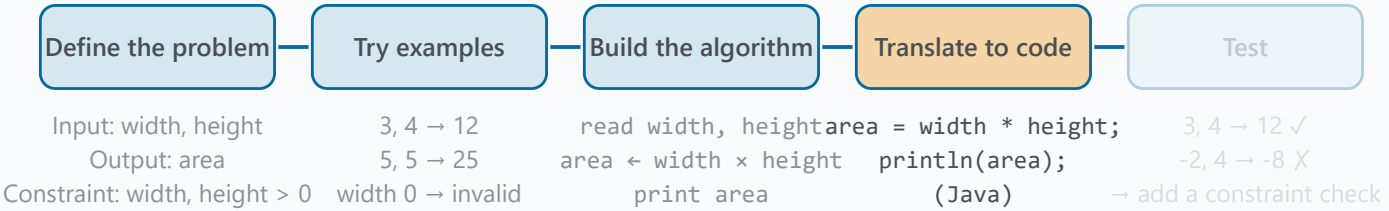


Write a language-independent **algorithm** (pseudocode).

Step 3 / 5

- Each stage's output is the next stage's input.
- A badly defined problem can become a program that works correctly but does the wrong job.

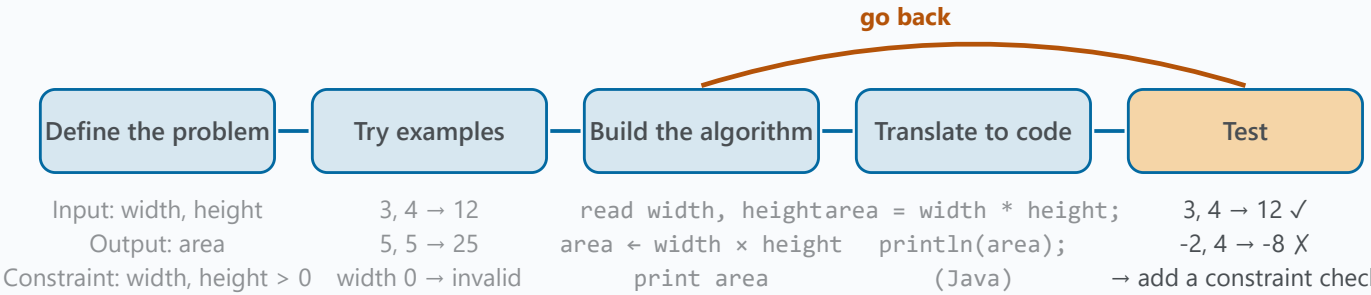
From Problem to Program



Translate the algorithm into the chosen language (Java).

- Each stage's output is the next stage's input.
- A badly defined problem can become a program that works correctly but does the wrong job.

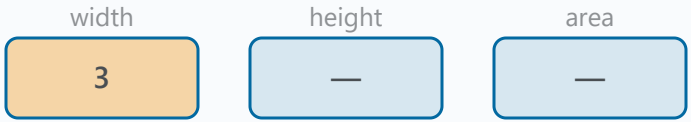
From Problem to Program



Test: a negative input gave a wrong result → orange arrow: go back to the algorithm and add the step "error unless width, height > 0".

- Each stage's output is the next stage's input.
- A badly defined problem can become a program that works correctly but does the wrong job.

Tracing the Algorithm: Rectangle Area



Run 1: read **width = 3**; it goes into the memory box named "width".

Step 1 / 9

```
▶ read width
2 read height
3 if width > 0 and height > 0 then
4     area ← width × height
5     print "Area: ", area
6 else
7     print "Invalid size"
```

VARIABLES

`width = 3`

OUTPUT

Tracing the Algorithm: Rectangle Area



Read **height = 4**.

Step 2 / 9

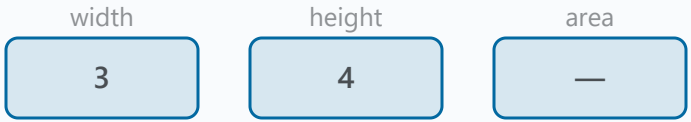
```
1 read width
▶ read height
3 if width > 0 and height > 0 then
4     area ← width × height
5     print "Area: ", area
6 else
7     print "Invalid size"
```

VARIABLES

`width = 3` `height = 4`

OUTPUT

Tracing the Algorithm: Rectangle Area



Condition: **3 > 0 and 4 > 0 → true**.

Step 3 / 9

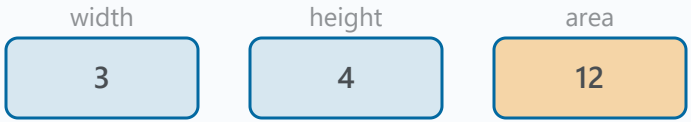
```
1 read width
2 read height
▶ if width > 0 and height > 0 then
4     area ← width × height
5     print "Area: ", area
6 else
7     print "Invalid size"
```

VARIABLES

`width = 3` `height = 4`

OUTPUT

Tracing the Algorithm: Rectangle Area



area = 3 × 4 = 12.

Step 4 / 9

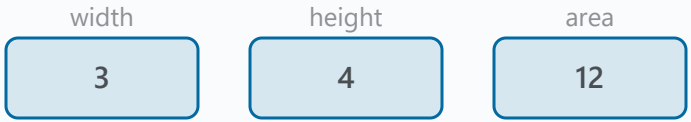
```
1 read width
2 read height
3 if width > 0 and height > 0 then
▶     area ← width × height
5     print "Area: ", area
6 else
7     print "Invalid size"
```

VARIABLES

`width = 3` `height = 4` `area = 12`

OUTPUT

Tracing the Algorithm: Rectangle Area



The result is printed. The "else" branch is skipped this time.

Step 5 / 9

```
1 read width
2 read height
3 if width > 0 and height > 0 then
4     area ← width × height
▶     print "Area: ", area
6 else
7     print "Invalid size"
```

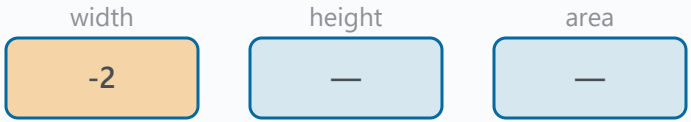
VARIABLES

`width = 3` `height = 4` `area = 12`

OUTPUT

```
Area: 12
```

Tracing the Algorithm: Rectangle Area



Run 2: read **width = -2**; it goes into the memory box named "width".

Step 6 / 9

```
▶ read width
2 read height
3 if width > 0 and height > 0 then
4     area ← width × height
5     print "Area: ", area
6 else
7     print "Invalid size"
```

VARIABLES

`width = −2`

OUTPUT

Tracing the Algorithm: Rectangle Area



Read **height = 4**.

Step 7 / 9

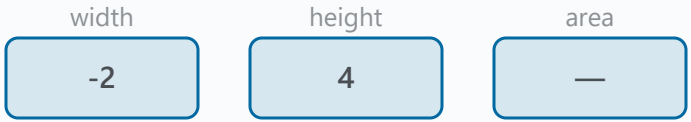
```
1 read width
▶ read height
3 if width > 0 and height > 0 then
4     area ← width × height
5     print "Area: ", area
6 else
7     print "Invalid size"
```

VARIABLES

`width = −2` `height = 4`

OUTPUT

Tracing the Algorithm: Rectangle Area



Condition: **-2 > 0 and 4 > 0** → **false** (width is negative) → take the else branch.

Step 8 / 9

```
1 read width
2 read height
▶ if width > 0 and height > 0 then
4     area ← width × height
5     print "Area: ", area
6 else
7     print "Invalid size"
```

VARIABLES

`width = -2` `height = 4`

OUTPUT

Tracing the Algorithm: Rectangle Area

width	height	area
-2	4	—

A warning is printed; the area is never computed. The bug found in testing is now fixed.

Step 9 / 9

```
1 read width
2 read height
3 if width > 0 and height > 0 then
4     area ← width × height
5     print "Area: ", area
6 else
▶     print "Invalid size"
```

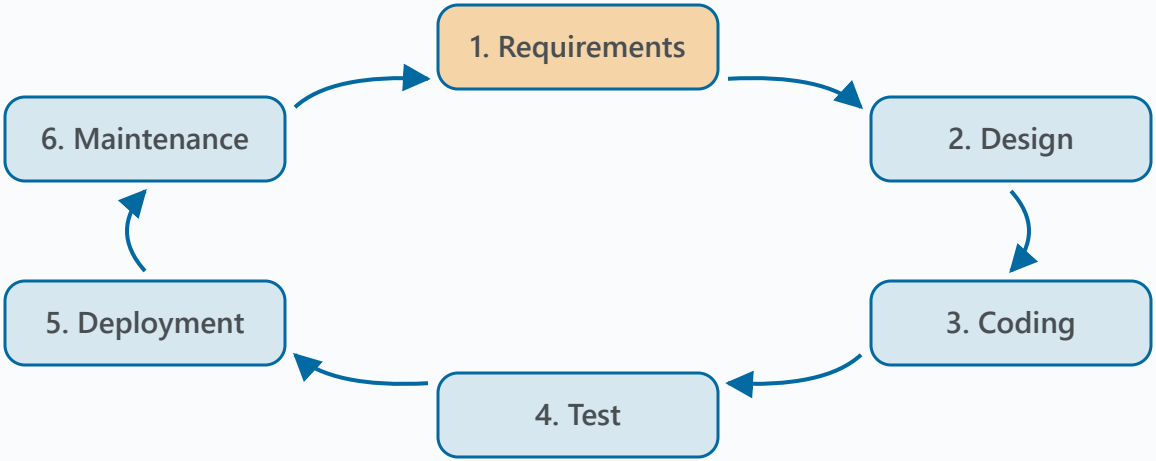
VARIABLES

`width = -2` `height = 4`

OUTPUT

```
Invalid size
```

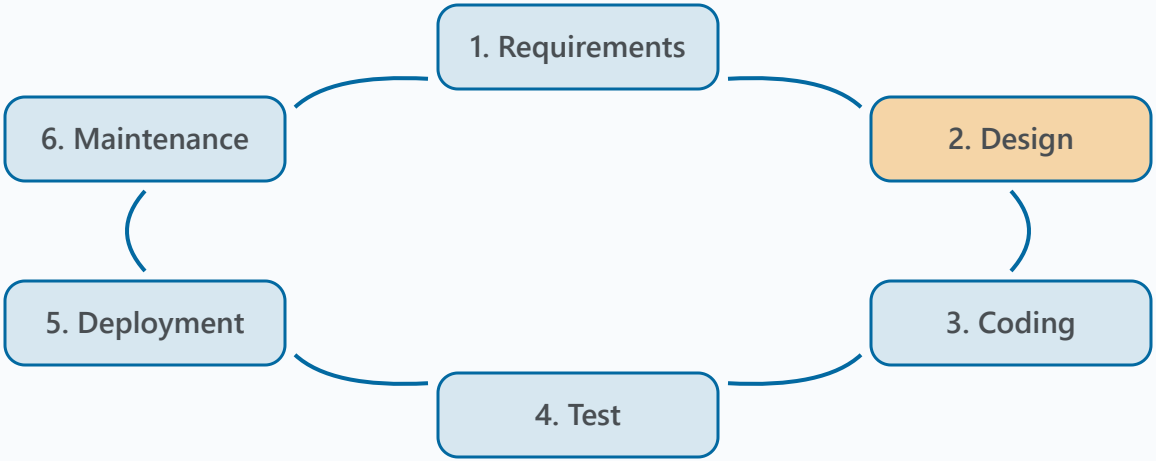
The Software Development Process



Requirements: what does the user want? E.g. "an app that computes a student's grade average".

- Depending on the project, the stages can run in sequence or iteratively; user feedback can change the requirements and the design.

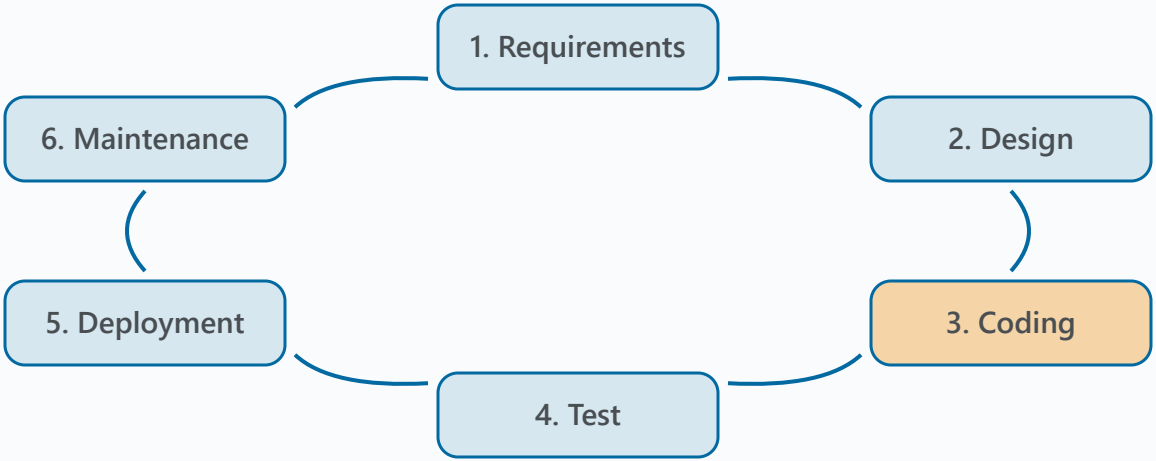
The Software Development Process



Design: plan the screens, data and algorithms (this is where flowcharts come in).

- Depending on the project, the stages can run in sequence or iteratively; user feedback can change the requirements and the design.

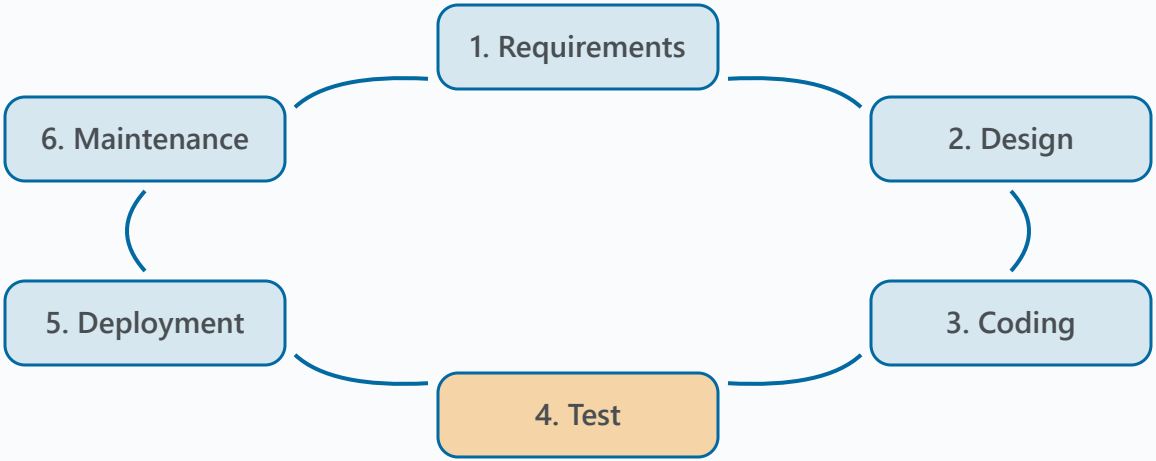
The Software Development Process



Coding: implement the design in code.

- Depending on the project, the stages can run in sequence or iteratively; user feedback can change the requirements and the design.

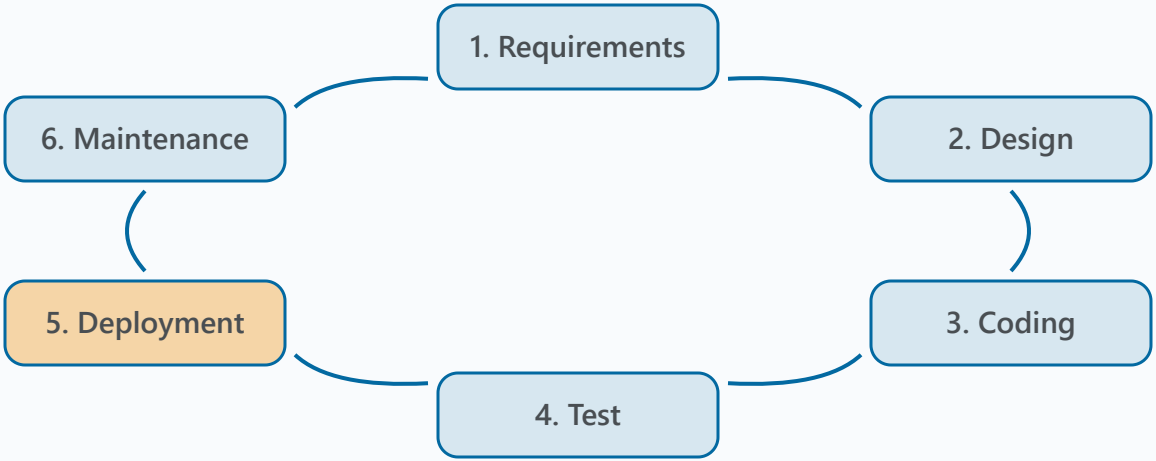
The Software Development Process



Testing: compare expected and actual results; if there are bugs, go back to coding.

- Depending on the project, the stages can run in sequence or iteratively; user feedback can change the requirements and the design.

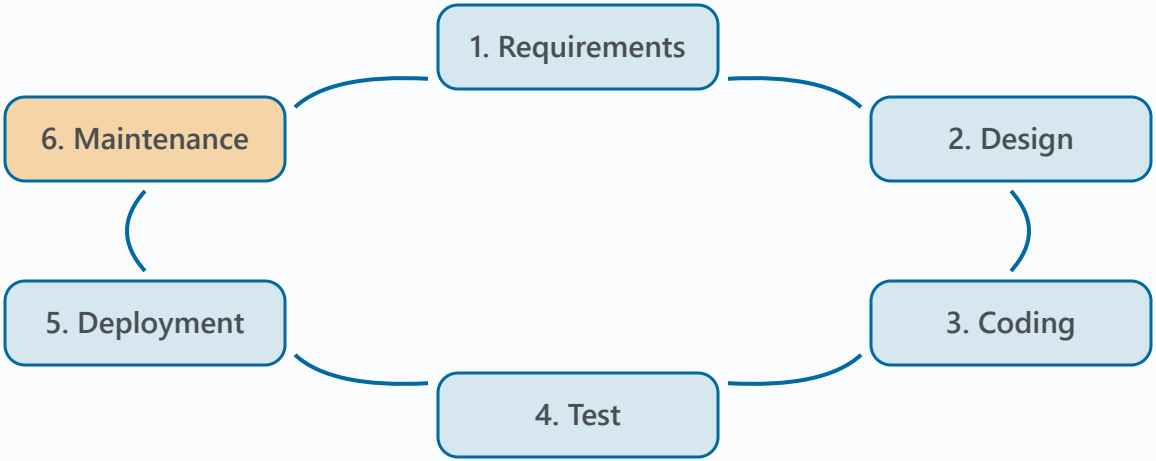
The Software Development Process



Deployment: deliver the program to users (installation, release).

- Depending on the project, the stages can run in sequence or iteratively; user feedback can change the requirements and the design.

The Software Development Process



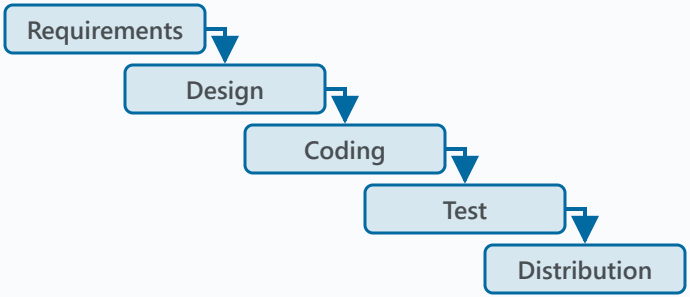
Maintenance: bugs are fixed and new requests arrive → the cycle restarts with new requirements.

- Depending on the project, the stages can run in sequence or iteratively; user feedback can change the requirements and the design.

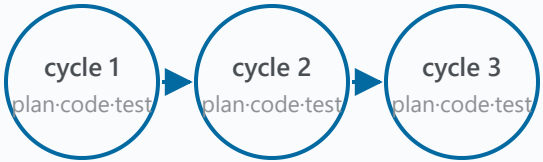
SOFTWARE

Development Approaches

Waterfall



Agile



a working version + feedback at the end of each cycle

Approach	Strength	Caution
Waterfall model	Clear stages and documents	Going back for changes is costly
Agile	Gets feedback in short cycles	Needs constant communication
Prototyping	Makes vague requirements visible	Don't mistake the prototype for the product

SOFTWARE

Development Tools

Code editor / IDE

Writing code, completion and project management.

Compiler / runtime

Translating and running the program.

Debugger

Inspecting program state step by step.

Version control

Change history and teamwork.

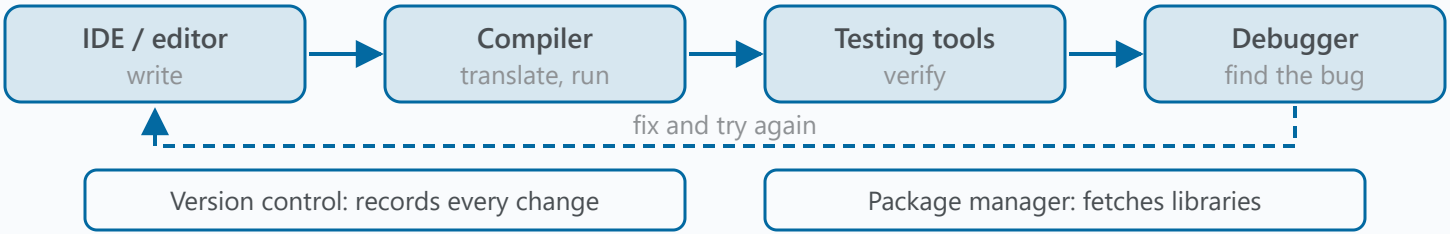
Testing tools

Checking expected behavior automatically.

Package manager

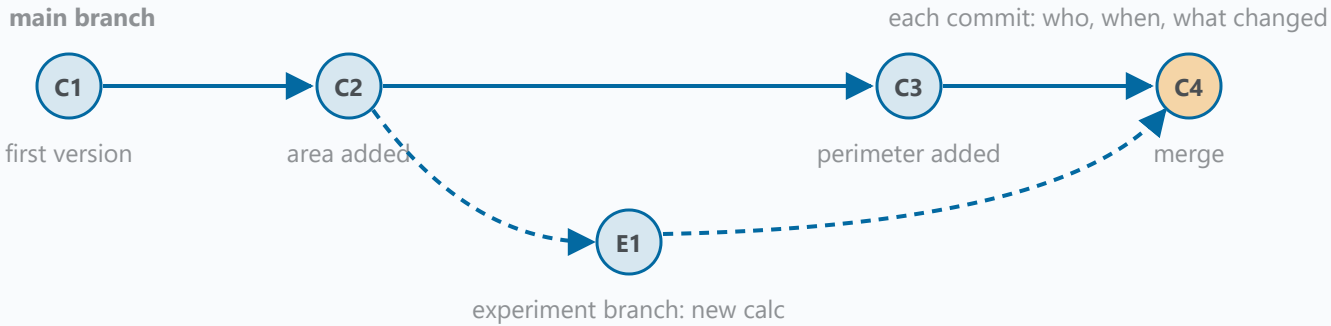
Managing dependencies.

One development round: each tool steps in at a different point of the cycle



SOFTWARE

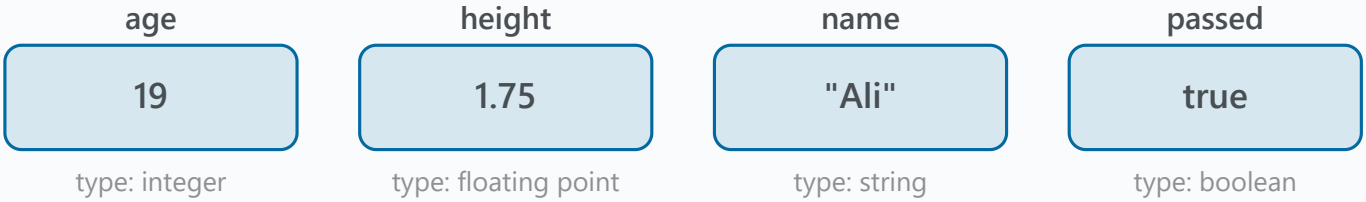
Version Control



- Each **commit** stores the files as they are at that moment, with who changed what and when. After a bad change you can go back to an earlier version.
- A **branch** lets you experiment without breaking the main code; when the work is done, the branch is **merged** into the main branch.
- Team members work on the same files; if two people change the same line, the tool flags the **conflict** and asks someone to choose.
- The most widely used tool is **Git**; GitHub and GitLab are services that host repositories online.

Variables and Data Types

- A variable is a **named box** in memory holding a value; the value can change while the program runs.
- A data type fixes how a value is represented and which operations apply to it.
- Meaningful names make a program's purpose visible.

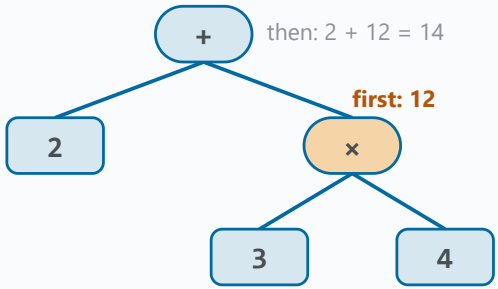


In Java: int, double, String, boolean.

Operator Precedence

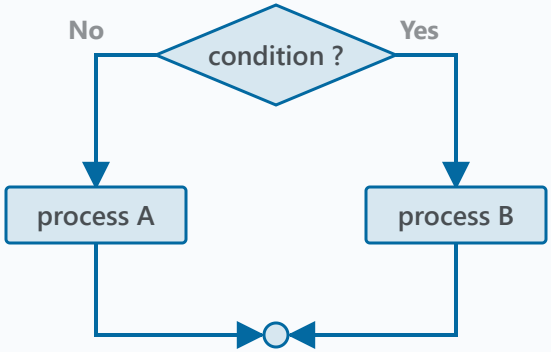
1. Parentheses
2. Multiplication, division, mod (remainder)
3. Addition, subtraction
4. Equal levels go left to right

Expression	Result
`2 + 3 × 4`	14
`(2 + 3) × 4`	20
`20 / 4 × 2`	10 (left to right)
`10 − 4 − 3`	3
`7 mod 3 + 1`	2

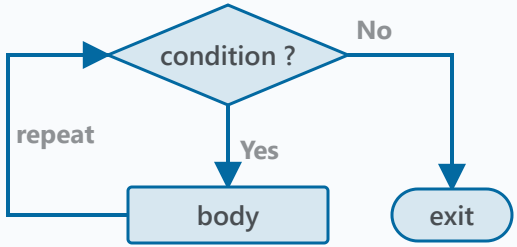


Forget the parentheses in a formula and the program gives no error, just a wrong result: `avg ← a + b / 2` divides b first. This is a **logic error**; only testing finds it.

Decisions and Loops



Decision: only one of two paths



Loop: an arrow going back

- A decision picks one of several paths by a condition; a loop repeats a group of steps while a condition holds.
- In both, it must be clear when the condition changes (otherwise: an infinite loop).

Loop Trace: An Infinite Loop

First version (no line 4)

i	condition	times the body ran
1	i ≤ 3	0

First version: line 4 is missing. i ← 1 sets up the loop variable.

Step 1 / 12

```
▶ i ← 1
2 while i ≤ 3 repeat
3     print i
4     i ← i + 1      // missing at first
5 print "done"
```

VARIABLES

i = 1

OUTPUT

Loop Trace: An Infinite Loop

First version (no line 4)

i	condition	times the body ran
1	1 ≤ 3 → true	0

Condition **1 ≤ 3** → **true**: enter the loop body.

Step 2 / 12

```
1 i ← 1
▶ while i ≤ 3 repeat
3     print i
4     i ← i + 1      // missing at first
5 print "done"
```

VARIABLES

`i = 1`

OUTPUT

Loop Trace: An Infinite Loop

First version (no line 4)

i	condition	times the body ran
1	i ≤ 3	1

print i → 1. The body is done; back to the condition.

Step 3 / 12

```
1 i ← 1
2 while i ≤ 3 repeat
▶     print i
4     i ← i + 1      // missing at first
5 print "done"
```

VARIABLES

`i = 1`

OUTPUT

```
1
```

Loop Trace: An Infinite Loop

First version (no line 4)

i	condition	times the body ran
1	1 ≤ 3 → true	1

The condition is again **1 ≤ 3 → true**: no line in the body changes i.

Step 4 / 12

```
1 i ← 1
▶ while i ≤ 3 repeat
3     print i
4     i ← i + 1      // missing at first
5 print "done"
```

VARIABLES

`i = 1`

OUTPUT

```
1
```

Loop Trace: An Infinite Loop

First version (no line 4)

i	condition	times the body ran
1	i ≤ 3	2

The same output again: 1. The state is identical to the previous round.

Step 5 / 12

```
1 i ← 1
2 while i ≤ 3 repeat
▶     print i
4     i ← i + 1      // missing at first
5 print "done"
```

VARIABLES

`i = 1`

OUTPUT

```
1
1
```

Loop Trace: An Infinite Loop

First version (no line 4)

i	condition	times the body ran
1	1 ≤ 3 → true	…

↻ ∞

Nothing ever makes the condition false, so the loop never ends: an **infinite loop**. A console program can only be stopped with Ctrl+C.

Step 6 / 12

```
1 i ← 1
▶ while i ≤ 3 repeat
3     print i
4     i ← i + 1      // missing at first
5 print "done"
```

VARIABLES

`i = 1`

OUTPUT

```
1
1
…
```

Loop Trace: An Infinite Loop

Fixed version

i	condition	times the body ran
1	i ≤ 3	0

Fixed version: **i ← i + 1** added to the body. Start again: i = 1.

Step 7 / 12

```
▶ i ← 1
2 while i ≤ 3 repeat
3     print i
4     i ← i + 1      // missing at first
5 print "done"
```

VARIABLES

i = 1

OUTPUT

Loop Trace: An Infinite Loop

Fixed version

i	condition	times the body ran
2	1 ≤ 3 → true	1

1 ≤ 3 is true: print 1, then i ← 2. The loop variable now advances each round.

Step 8 / 12

```
1 i ← 1
2 while i ≤ 3 repeat
▶     print i
▶     i ← i + 1      // missing at first
5 print "done"
```

VARIABLES

`i = 2`

OUTPUT

```
1
```

Loop Trace: An Infinite Loop

Fixed version

	condition	times the body ran
i = 3	2 ≤ 3 → true	2

2 ≤ 3 is true: print 2, i ← 3.

Step 9 / 12

```
1 i ← 1
2 while i ≤ 3 repeat
▶     print i
▶     i ← i + 1      // missing at first
5 print "done"
```

VARIABLES

`i = 3`

OUTPUT

```
1
2
```

Loop Trace: An Infinite Loop

Fixed version

	condition	times the body ran
i = 4	3 ≤ 3 → true	3

3 ≤ 3 is also true (equality): print 3, i ← 4.

Step 10 / 12

```
1 i ← 1
2 while i ≤ 3 repeat
▶     print i
▶     i ← i + 1      // missing at first
5 print "done"
```

VARIABLES

`i = 4`

OUTPUT

```
1
2
3
```

Loop Trace: An Infinite Loop

Fixed version

i	condition	times the body ran
4	4 ≤ 3 → false	3

Condition **4 ≤ 3** → **false**: leave the loop.

Step 11 / 12

```
1 i ← 1
▶ while i ≤ 3 repeat
3     print i
4     i ← i + 1      // missing at first
5 print "done"
```

VARIABLES

`i = 4`

OUTPUT

```
1
2
3
```

Loop Trace: An Infinite Loop

Fixed version

i	condition	times the body ran
4	i ≤ 3	3

The line after the loop: **done**. The body ran 3 times and the condition was checked 4 times.

Step 12 / 12

```
1 i ← 1
2 while i ≤ 3 repeat
3     print i
4     i ← i + 1      // missing at first
▶ print "done"
```

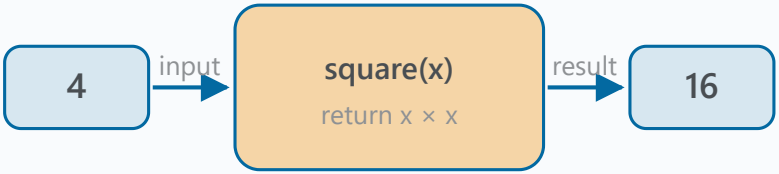
VARIABLES

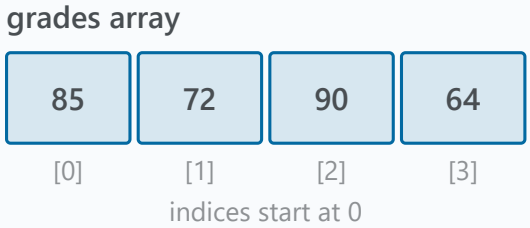
`i = 4`

OUTPUT

```
1
2
3
done
```

Functions and Arrays





- A function names a task; it can take input and return a result.
- An array stores many values of the same type, by index.
- These constructs help split a program into small, testable pieces.

Function Trace: Call and Return

The definition is skipped; execution starts at line 5.

area(3, 4) is about to be called

main program a = — b = —

call stack (the top frame is running)

A function **definition** is only recorded, not run. The program starts at line 5 and meets the call **area(3, 4)**.

Step 1 / 10

```
1 function area(width, height)
2     result ← width × height
3     return result
4 end
▶ a ← area(3, 4)
6 b ← area(2, 5)
7 print a + b
```

VARIABLES

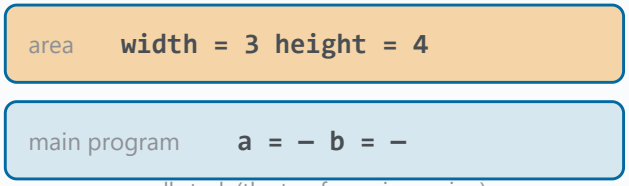
—

CALL STACK

main program

OUTPUT

Function Trace: Call and Return



call stack (the top frame is running)

A new frame opened.

arguments copied into parameters

A new frame for **area** is pushed onto the call stack: 3 → width, 4 → height.

Step 2 / 10

```
▶ function area(width, height)
2     result ← width × height
3     return result
4 end
5 a ← area(3, 4)
6 b ← area(2, 5)
7 print a + b
```

VARIABLES

`width = 3` `height = 4`

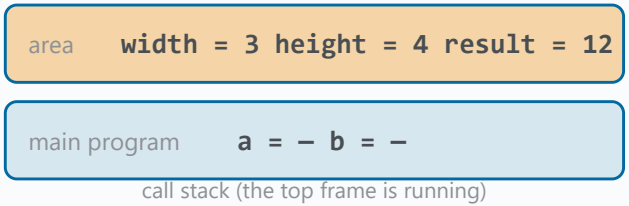
CALL STACK

`area(3, 4)`

`main program`

OUTPUT

Function Trace: Call and Return



call stack (the top frame is running)

result ← 3 × 4 = 12

The body runs: **result = 12**. This variable lives only in this frame.

Step 3 / 10

```
1 function area(width, height)
▶     result ← width × height
3     return result
4 end
5 a ← area(3, 4)
6 b ← area(2, 5)
7 print a + b
```

VARIABLES

`width = 3` `height = 4` `result = 12`

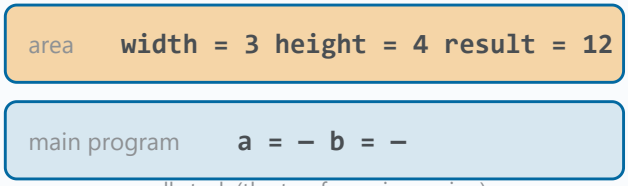
CALL STACK

`area(3, 4)`

`main program`

OUTPUT

Function Trace: Call and Return



call stack (the top frame is running)

return 12 ↓

the frame will be removed

return result: 12 goes back to the caller and the function ends.

Step 4 / 10

```
1 function area(width, height)
2     result ← width × height
▶     return result
4 end
5 a ← area(3, 4)
6 b ← area(2, 5)
7 print a + b
```

VARIABLES

`width = 3` `height = 4` `result = 12`

CALL STACK

`area(3, 4)`

`main program`

OUTPUT

Function Trace: Call and Return

a ← 12

main program a = 12 b = –

width, height, result are gone

call stack (the top frame is running)

The frame is popped; the returned value is assigned to **a**: a = 12.

Step 5 / 10

```
1 function area(width, height)
2     result ← width × height
3     return result
4 end
▶ a ← area(3, 4)
6 b ← area(2, 5)
7 print a + b
```

VARIABLES

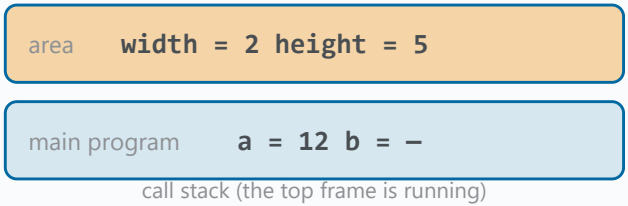
a = 12

CALL STACK

main program

OUTPUT

Function Trace: Call and Return



area(2, 5) called

same function, new frame

The same function is called with different arguments: a brand-new frame, width = 2, height = 5.

Step 6 / 10

```
▶ function area(width, height)
2     result ← width × height
3     return result
4 end
5 a ← area(3, 4)
6 b ← area(2, 5)
7 print a + b
```

VARIABLES

`width = 2` `height = 5`

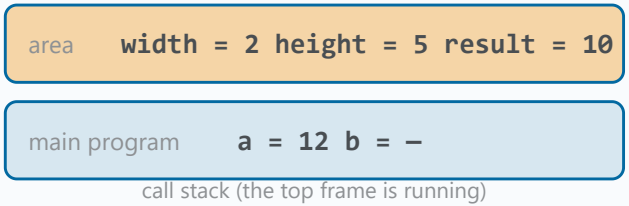
CALL STACK

`area(2, 5)`

`main program`

OUTPUT

Function Trace: Call and Return



call stack (the top frame is running)

result ← 2 × 5 = 10

result = 10. The 12 from the previous call is not visible in this frame.

Step 7 / 10

```
1 function area(width, height)
▶     result ← width × height
3     return result
4 end
5 a ← area(3, 4)
6 b ← area(2, 5)
7 print a + b
```

VARIABLES

width = 2 | height = 5 | result = 10

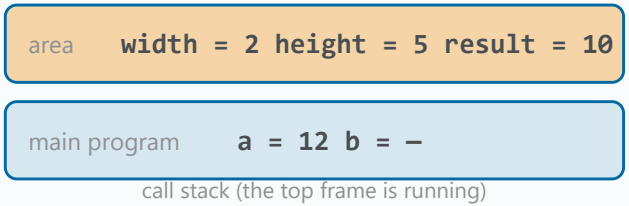
CALL STACK

area(2, 5)

main program

OUTPUT

Function Trace: Call and Return



call stack (the top frame is running)

return 10 ↓

10 is returned and the frame closes.

Step 8 / 10

```
1 function area(width, height)
2     result ← width × height
▶     return result
4 end
5 a ← area(3, 4)
6 b ← area(2, 5)
7 print a + b
```

VARIABLES

`width = 2` `height = 5` `result = 10`

CALL STACK

`area(2, 5)`

`main program`

OUTPUT

Function Trace: Call and Return

b ← 10

main program a = 12 b = 10

call stack (the top frame is running)

The returned value is assigned to **b**: b = 10.

Step 9 / 10

```
1 function area(width, height)
2     result ← width × height
3     return result
4 end
5 a ← area(3, 4)
▶ b ← area(2, 5)
7 print a + b
```

VARIABLES

a = 12 b = 10

CALL STACK

main program

OUTPUT

Function Trace: Call and Return

print 12 + 10 → 22

main program a = 12 b = 10

call stack (the top frame is running)

22 is printed. A function written once was used twice; the area formula lives in one place.

Step 10 / 10

```
1 function area(width, height)
2     result ← width × height
3     return result
4 end
5 a ← area(3, 4)
6 b ← area(2, 5)
▶ print a + b
```

VARIABLES

a = 12 b = 10

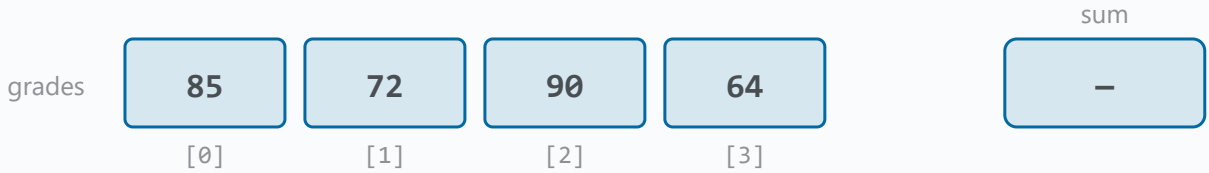
CALL STACK

main program

OUTPUT

22

Array Trace: Average of Grades



The array holds four grades under **one name**; cells are selected by the indices 0, 1, 2, 3.

Step 1 / 8

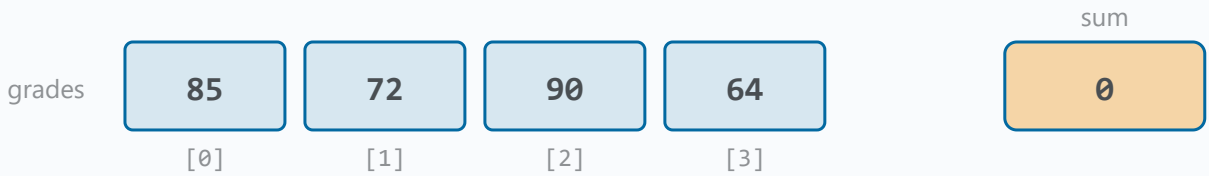
```
▶ grades ← [85, 72, 90, 64]
2 sum ← 0
3 for each i ← 0, 1, 2, 3 do
4     sum ← sum + grades[i]
5 avg ← sum / 4
6 print "Average: ", avg
```

VARIABLES

`grades = [85, 72, 90, 64]`

OUTPUT

Array Trace: Average of Grades



sum ← 0: the accumulator starts at zero. Without a starting value, the sum would be undefined.

Step 2 / 8

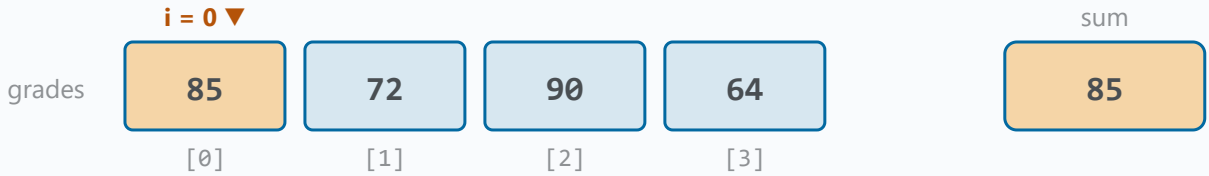
```
1 grades ← [85, 72, 90, 64]
▶ sum ← 0
3 for each i ← 0, 1, 2, 3 do
4     sum ← sum + grades[i]
5 avg ← sum / 4
6 print "Average: ", avg
```

VARIABLES

`sum = 0`

OUTPUT

Array Trace: Average of Grades



i = 0: grades[0] = 85 → sum = 0 + 85 = **85**.

Step 3 / 8

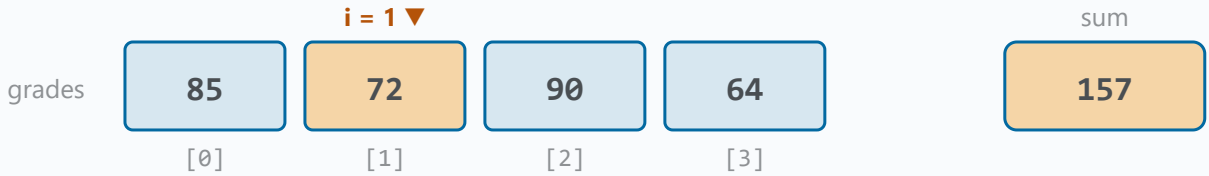
```
1 grades ← [85, 72, 90, 64]
2 sum ← 0
▶ for each i ← 0, 1, 2, 3 do
▶     sum ← sum + grades[i]
5 avg ← sum / 4
6 print "Average: ", avg
```

VARIABLES

`i = 0` `grades[i] = 85` `sum = 85`

OUTPUT

Array Trace: Average of Grades



i = 1: grades[1] = 72 → sum = 85 + 72 = **157**.

Step 4 / 8

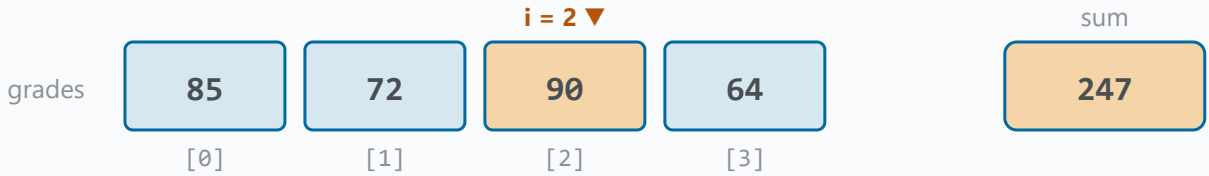
```
1 grades ← [85, 72, 90, 64]
2 sum ← 0
▶ for each i ← 0, 1, 2, 3 do
▶     sum ← sum + grades[i]
5 avg ← sum / 4
6 print "Average: ", avg
```

VARIABLES

`i = 1` `grades[i] = 72` `sum = 157`

OUTPUT

Array Trace: Average of Grades



i = 2: grades[2] = 90 → sum = 157 + 90 = **247**.

Step 5 / 8

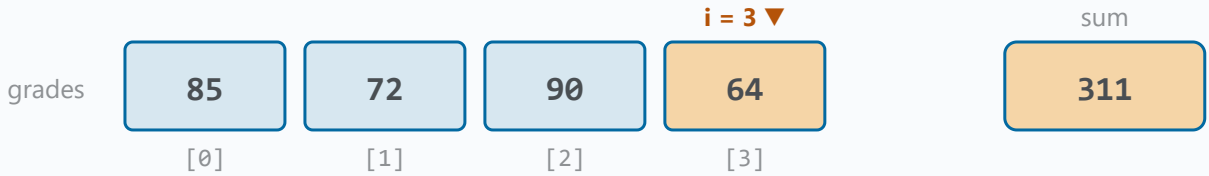
```
1 grades ← [85, 72, 90, 64]
2 sum ← 0
▶ for each i ← 0, 1, 2, 3 do
▶     sum ← sum + grades[i]
5 avg ← sum / 4
6 print "Average: ", avg
```

VARIABLES

`i = 2` `grades[i] = 90` `sum = 247`

OUTPUT

Array Trace: Average of Grades



i = 3: grades[3] = 64 → sum = 247 + 64 = **311**.

Step 6 / 8

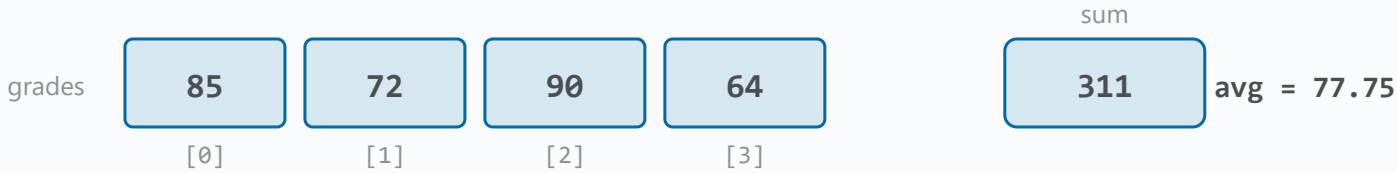
```
1 grades ← [85, 72, 90, 64]
2 sum ← 0
▶ for each i ← 0, 1, 2, 3 do
▶     sum ← sum + grades[i]
5 avg ← sum / 4
6 print "Average: ", avg
```

VARIABLES

`i = 3` `grades[i] = 64` `sum = 311`

OUTPUT

Array Trace: Average of Grades



The loop ends after 4 rounds. **avg = 311 / 4 = 77.75**. The last index is 3: length − 1.

Step 7 / 8

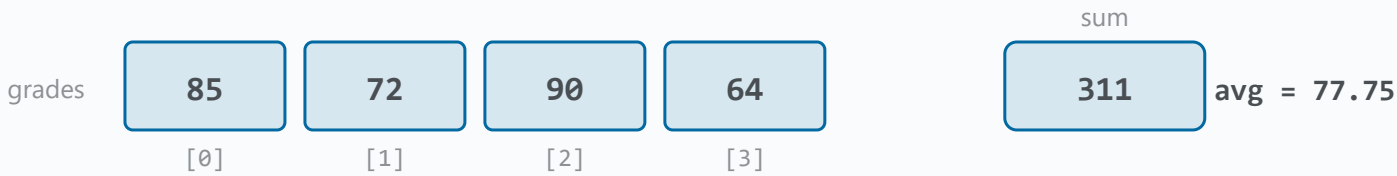
```
1 grades ← [85, 72, 90, 64]
2 sum ← 0
3 for each i ← 0, 1, 2, 3 do
4     sum ← sum + grades[i]
▶ avg ← sum / 4
6 print "Average: ", avg
```

VARIABLES

`sum = 311` `avg = 77.75`

OUTPUT

Array Trace: Average of Grades



The result is printed. Had the loop reached **grades[4]**, it would have gone past the end: a 4-element array has no cell 4.

Step 8 / 8

```
1 grades ← [85, 72, 90, 64]
2 sum ← 0
3 for each i ← 0, 1, 2, 3 do
4     sum ← sum + grades[i]
5 avg ← sum / 4
▶ print "Average: ", avg
```

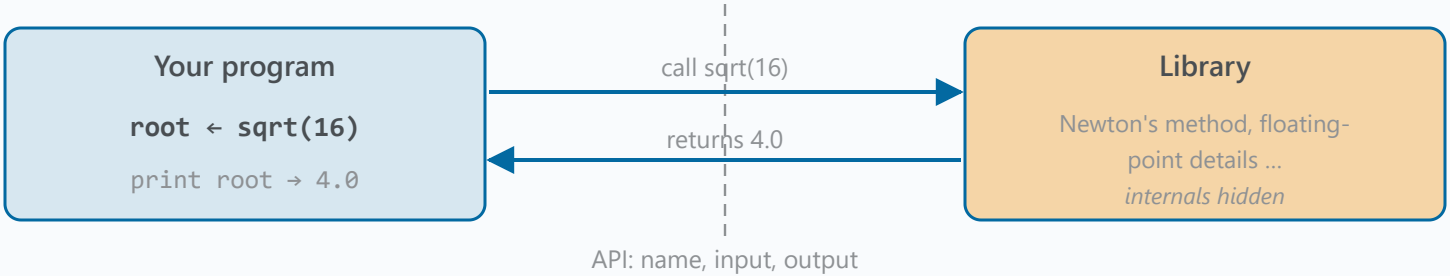
VARIABLES

`sum = 311` `avg = 77.75`

OUTPUT

```
Average: 77.75
```

Libraries and APIs

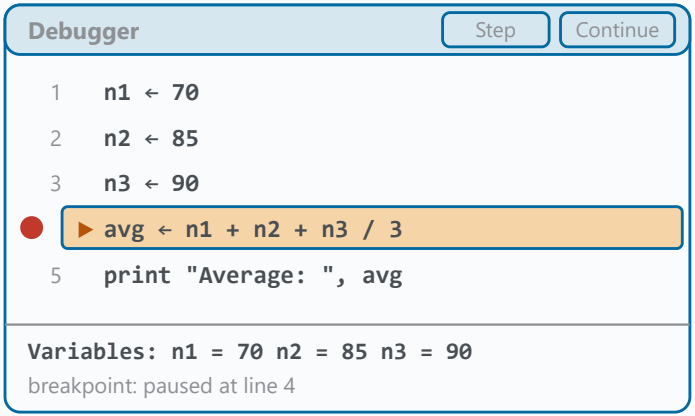


- A **library** is a collection of ready-made functions others have written and tested; it saves rewriting the same work in every program.
- An **API** (application programming interface) defines the functions a library or service exposes, with their inputs and return values. Callers need not know the internals.

Call	Language	Result
`Math.sqrt(16)`	Java	4.0
`Math.max(3, 8)`	Java	8
`len("Giresun")`	Python	7

Debugging Techniques

- **Reproduce the bug**: does the same input trigger it every time?
- **Print intermediate values**: print variables before and after the suspicious line.
- Set a **breakpoint**: the program pauses at that line so you can inspect variables.
- **Step** through: after each line, see which value changed.
- **Narrow it down**: disable half the code and find which half holds the bug.
- **Explain it out loud**: walking someone (or a toy on your desk) through the code line by line often reveals the bug.



Paused, the program shows that n3 / 3 = 30 will be computed: the average comes out as 185, but 81.67 is expected.

Finding a Bug by Tracing: The Largest Number

Test 1: 7, 12, 5



Test 1: inputs 7, 12, 5; expected output 12.

Step 1 / 10

```
▶ read a, b, c
2 max ← 0
3 if a > max then max ← a
4 if b > max then max ← b
5 if c > max then max ← c
6 print "Largest: ", max
```

VARIABLES

a = 7 | b = 12 | c = 5

OUTPUT

Finding a Bug by Tracing: The Largest Number

Test 1: 7, 12, 5

a	b	c	max
7	12	5	0

max ← **0**: a starting value for the largest.

Step 2 / 10

```
1 read a, b, c
▶ max ← 0
3 if a > max then max ← a
4 if b > max then max ← b
5 if c > max then max ← c
6 print "Largest: ", max
```

VARIABLES

a = 7 | b = 12 | c = 5 | max = 0

OUTPUT

Finding a Bug by Tracing: The Largest Number

Test 1: 7, 12, 5



7 > 0 is true: max = 7.

Step 3 / 10

```
1 read a, b, c
2 max ← 0
▶ if a > max then max ← a
4 if b > max then max ← b
5 if c > max then max ← c
6 print "Largest: ", max
```

VARIABLES

a = 7 | b = 12 | c = 5 | max = 7

OUTPUT

Finding a Bug by Tracing: The Largest Number

Test 1: 7, 12, 5



12 > 7 is true: max = 12.

Step 4 / 10

```
1 read a, b, c
2 max ← 0
3 if a > max then max ← a
▶ if b > max then max ← b
5 if c > max then max ← c
6 print "Largest: ", max
```

VARIABLES

`a = 7` `b = 12` `c = 5` `max = 12`

OUTPUT

Finding a Bug by Tracing: The Largest Number

Test 1: 7, 12, 5

a	b	c	max
7	12	5	12

5 > 12

false

5 > 12 is false: max stays. Output **12** ✓. One passing test does not prove a program correct.

Step 5 / 10

```
1 read a, b, c
2 max ← 0
3 if a > max then max ← a
4 if b > max then max ← b
▶ if c > max then max ← c
▶ print "Largest: ", max
```

VARIABLES

a = **7** b = **12** c = **5** max = **12**

OUTPUT

```
Largest: 12
```

Finding a Bug by Tracing: The Largest Number

Test 2: −5, −2, −9



Test 2: all negative, −5, −2, −9; expected output −2.

Step 6 / 10

```
▶ read a, b, c
2 max ← 0
3 if a > max then max ← a
4 if b > max then max ← b
5 if c > max then max ← c
6 print "Largest: ", max
```

VARIABLES

a = −5 b = −2 c = −9

OUTPUT

Finding a Bug by Tracing: The Largest Number

Test 2: −5, −2, −9

a	b	c	max
-5	-2	-9	0

max ← 0. This value comes from none of the inputs.

Step 7 / 10

```
1 read a, b, c
▶ max ← 0
3 if a > max then max ← a
4 if b > max then max ← b
5 if c > max then max ← c
6 print "Largest: ", max
```

VARIABLES

a = −5 | b = −2 | c = −9 | max = 0

OUTPUT

Finding a Bug by Tracing: The Largest Number

Test 2: −5, −2, −9

a	b	c	max
-5	-2	-9	0

`-5 > 0 · -2 > 0 · -9 > 0`

all three false

All three comparisons are false: no input exceeds 0, so max never changes.

Step 8 / 10

```
1 read a, b, c
2 max ← 0
▶ if a > max then max ← a
▶ if b > max then max ← b
▶ if c > max then max ← c
6 print "Largest: ", max
```

VARIABLES

`a = −5` `b = −2` `c = −9` `max = 0`

OUTPUT

Finding a Bug by Tracing: The Largest Number

Test 2: −5, −2, −9

a	b	c	max
-5	-2	-9	0

`output 0, expected -2`

✗ **logic error**

Output **0** ✗. The trace points to the culprit: the starting value on line 2 comes from outside the inputs.

Step 9 / 10

```
1 read a, b, c
2 max ← 0
3 if a > max then max ← a
4 if b > max then max ← b
5 if c > max then max ← c
▶ print "Largest: ", max
```

VARIABLES

`a = −5` `b = −2` `c = −9` `max = 0`

OUTPUT

```
Largest: 0
```

Finding a Bug by Tracing: The Largest Number

Fix: max ← a



Fix: **max ← a**. max = −5 → −2 > −5 true, max = −2 → −9 > −2 false. Output −2 ✓. Test 1 is rerun as well: still 12.

Step 10 / 10

```
1 read a, b, c
▶ max ← 0
3 if a > max then max ← a
4 if b > max then max ← b
5 if c > max then max ← c
6 print "Largest: ", max
```

VARIABLES

a = **−5** b = **−2** c = **−9** max = **−2**

OUTPUT

```
Largest: -2
```

Common Mistakes

Mistake	Example	What happens?
Missing or wrong initial value	`sum ← sum + x` without initializing `sum`	Java won't compile it; some languages compute with a garbage value
Off-by-one error	`for each i ← 1 … n − 1`	the last element is skipped, or one extra round runs
Infinite loop	the loop variable is not updated in the body	the program never ends
Division by zero	`avg ← sum / n`, n = 0	runtime error
Going past the end of an array	`grades[4]` in a 4-element array	runtime error
Forgetting parentheses	`avg ← a + b / 2`	wrong result (logic error)
Not validating input	accepting −3 as an age	meaningless results, errors in later steps

Boundary cases (0, an empty list, the smallest and largest values, negative numbers) expose most of these bugs.

The Cost of Software Bugs

Ariane 5 (1996)

A 64-bit floating-point velocity value **overflowed** when converted to a 16-bit integer. The navigation software shut down; about 37 seconds after liftoff the rocket veered off course and broke up within seconds.

Mars Climate Orbiter (1999)

One piece of software produced thrust data in imperial units (pound-force·seconds) while another expected metric units (newton·seconds). The probe came too close to Mars and was lost.

Therac-25 (1985–1987)

A timing bug in a radiotherapy machine's software bypassed a safety check when the operator typed quickly; patients received massive radiation overdoses.

- None of the three could be caught by a compiler: one was a boundary case, one a unit mismatch, one a timing-dependent logic error.
- The shared lesson: test, try boundary cases, state units explicitly, and have someone else review the code.

DEBUGGING

Readable Code

Works, but unreadable

```
a ← read()
b ← read()
c ← a*b/2
print c
```

The same program, readable

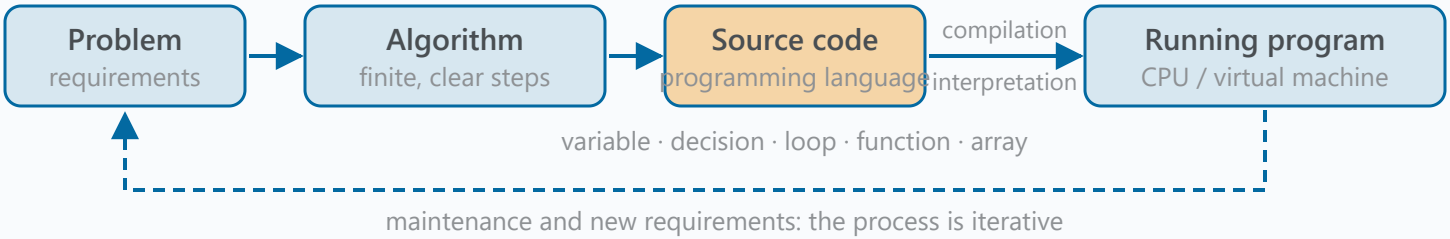
```
// triangle area = base × height / 2
base ← read()
height ← read()
area ← base × height / 2
print "Area: ", area
```

- **Meaningful names**: `area` instead of `c`; the reader learns the variable's role from its name.
- **Consistent indentation and spacing**: the bounds of nested blocks are visible at a glance.
- A named constant instead of a **magic number**: `VAT_RATE ← 0.20` instead of `0.20`; when the rate changes, you fix one line.
- **Move repetition into a function**: if the same formula appears in three places, every fix must be made three times.
- A comment explains **why** the code does something, not **what** it does. Code is written once and read many times.

SUMMARY

Chapter Summary

- A programming language expresses an algorithm in a form the computer can execute.
- Source code runs through compilation, interpretation or a hybrid process.
- Languages are classified by level, kind and paradigm.
- Software development is an iterative process from requirements to maintenance.
- Variables, decisions, loops, functions and arrays are the basic programming constructs.
- Bugs are found by testing with boundary cases and tracing the program step by step.



Review Questions

1. What is the difference between source code and machine language?
2. How do compiled and interpreted programs differ?
3. Why can an algorithm be designed independently of a programming language?
4. What is the purpose of software testing?
5. How do the roles of a decision and a loop differ?
6. What do the linker and the loader do to get a program running?
7. When is a type error detected under static and under dynamic typing?
8. How do the imperative and functional approaches express the sum of an array differently?
9. Why does a loop become an infinite loop?

SUMMARY

Exercises

1. Classify these as compile-time, runtime or logic errors: forgetting a semicolon; dividing the sum by n + 1 instead of n for an average; dividing by zero.
2. Write the input, output and constraints for the problem "compute the average of a student's three exam grades".
3. Write pseudocode for the same problem and trace it with (70, 85, 90); the result should be 81.67.
4. Add a perimeter calculation to the rectangle-area algorithm: perimeter = 2 × (width + height). What outputs do you expect for width = 3, height = 4?
5. For a phone app, name one task done in each stage of the software development process.
6. Classify Java, Python and C as compiled / interpreted / hybrid and draw their execution pipelines.

▶ **Hints**

References

- Sebesta, Concepts of Programming Languages, 12th ed., Pearson, 2019.
- Aho, Lam, Sethi and Ullman, Compilers: Principles, Techniques, and Tools, 2nd ed., Addison-Wesley.
- Sedgewick and Wayne, Computer Science: An Interdisciplinary Approach, Addison-Wesley, 2016.
- Downey and Mayfield, Think Java, 2nd ed., O'Reilly, 2019 (**free online edition**).
- Oracle, The Java Tutorials: **Getting Started**.

The End

Chapter 4.3: Introduction to Programming

Previous: Chapter 4.2: Flowcharts · Next: Chapter 5: Getting Started with Java