CS1010X: Programming Methodology I
Chapter 14: Introduction to Java

Lim Dillion
dillionlim@u.nus.edu

2025/26 Special Term 2
National University of Singapore

Chapter 14: Introduction to Java

Learning Objectives

By the end of this chapter, students should be able to:

- **Explain** why Java is widely used for teaching and software development (portability via the JVM, strong tooling, and a class-based design).
- **Describe** the Java program life cycle: *write* → *compile to bytecode* → *execute on the JVM*.
- **Distinguish** compile-time errors from runtime errors, and identify common causes of each.
- **Write** a valid Java program with a correct `main` method and understand file/class naming rules.
- **Use** primitive types and understand numeric promotions, integer division, and explicit casting.
- **Apply** basic operators and control-flow constructs (`if/else`, `switch`, loops, `break/continue`).
- **Work** with arrays, console input using `Scanner`, and formatted output using `printf`.
- **Use** the `String` class effectively, including common methods and escape sequences.

14.1 Why Java?

Java is frequently used as an introductory language for a few practical reasons:

- **A clear object-oriented model.** Most non-trivial Java programs are organized around classes and objects.
- **Strong static typing.** Variables have declared types, enabling early detection of many errors during compilation.
- **Portability.** Java is compiled to *bytecode* and executed on a Java Virtual Machine (JVM), supporting the idea of *"Write Once, Run Everywhere"*.
- **Industrial ecosystem.** The Java ecosystem has mature tooling (build systems, IDEs, profilers, testing frameworks).

Learning a New Language

A new language can be challenging in two distinct ways:

- **Syntax differences:** braces vs indentation, semicolons, declarations, etc.
- **Computation model differences:** static typing vs dynamic typing; compilation vs interpretation; object-oriented vs functional patterns.

If you understand the *algorithmic idea*, you can usually translate it across languages once the syntax and model are clear.

14.2 Java Program Life Cycle: From Source to Execution

At a high level, Java programs follow a **compile-then-run** workflow:

- **Write:** Create a `.java` source file.
- **Compile:** Use `javac` to compile source into `.class` bytecode.
- **Execute:** Use `java` to run bytecode on the JVM.

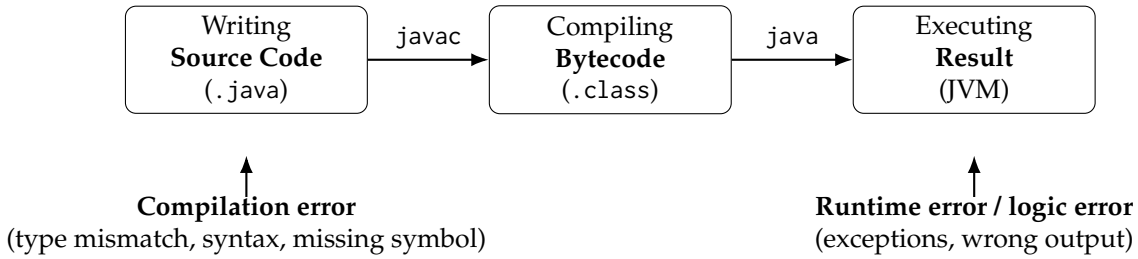


Figure 14.1: Java workflow: write → compile → execute.

14.2.1 Compile-Time vs Runtime Errors

- **Compile-time errors** are detected by `javac` before the program runs (e.g. syntax errors, undefined variables, incompatible types).
- **Runtime errors** occur during execution (e.g. division by zero, array index out of bounds, null pointer usage).

14.2.2 Structure of a Java Program

Java is class-based: code lives inside **classes**. A minimal program:

```
1 public class Hello {
2     public static void main(String[] args) {
3         System.out.println("Hello, world!");
4     }
5 }
```

14.2.3 File and Class Naming

- If a class is `public`, its name must match the file name: `Hello.java` contains `public class Hello`.
- Compile with `javac Hello.java`, run with `java Hello`.

14.2.4 Method Declaration Anatomy

A method header typically contains access control, modifiers, return type, name, and parameters:

```
1 public static float expt(float x, int n) {
2     ...
3 }
```

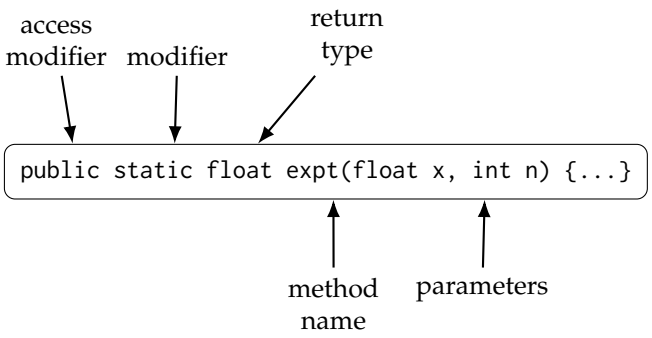


Figure 14.2: Reading a Java method signature.

14.2.5 The `main` Method

A Java application begins execution at:

```
1 public static void main(String[] args) { ... }
```

Key points:

- `public`: accessible by the JVM launcher.
- `static`: callable without creating an object.
- `void`: no return value required.
- `String[] args`: command-line arguments.

14.2.6 Static vs Dynamic Typing

- **Java is statically typed:** variables are declared with types; many errors are caught at compile time.
- **Python is dynamically typed:** values carry types at runtime; variables are names bound to objects.

14.3 Primitive Types in Java

14.3.1 Types in Java: Primitive vs Reference

Java types fall into two broad categories:

- **Primitive types**: store numeric/boolean values directly (e.g. `int`, `double`, `boolean`).
- **Reference types**: store references to objects (e.g. `String`, arrays, user-defined classes).

A key idea in Java is **static typing**: the compiler knows the declared (compile-time) type of each variable and checks whether assignments and operations are type-correct before running.

Primitive types include:

Kind	Types	Size (bits)
Boolean	`boolean`	1
Character	`char`	16
Integral	`byte`, `short`, `int`, `long`	8, 16, 32, 64
Floating-point	`float`, `double`	32, 64

Table 14.1: Primitive types and common sizes (overview).

14.3.2 Primitive Values Are Copied (No Sharing)

Unlike reference types, primitive variables do not alias each other: assignments copy the value.

```
int i = 1000;
int j = i;      // copies the value
i = i + 1;      // changes i only, j remains 1000
```

14.3.3 Integer and Floating-Point Literals

By default:

- An integer literal (e.g. `2030`) is of type `int`.
- A floating-point literal (e.g. `20.30`) is of type `double`.

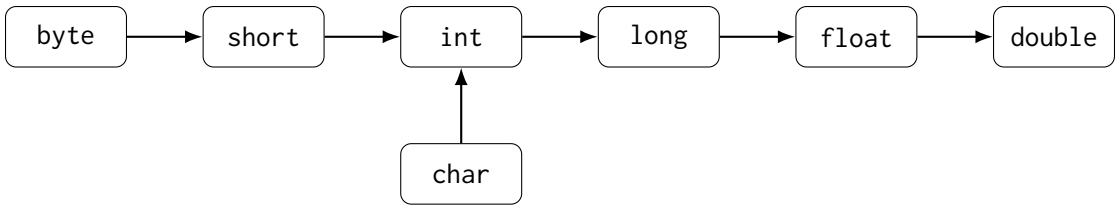
Use suffixes to force a different type:

- `L` for `long`: `2030L`
- `f` for `float`: `20.30f`

14.3.4 Primitive Subtypes and Conversions

Java allows certain **widening primitive conversions** implicitly (no cast required).

`byte <: short <: int <: long <: float <: double`
`char <: int`



An arrow $A \rightarrow B$ means a value of type A can be used where B is expected (widening). E.g. `int` can widen to `double`.

Figure 14.3: Primitive widening relationships.

Why is `long` a subtype of `float`?

At first glance, this feels counter-intuitive: `long` is 64-bit but `float` is 32-bit. The key is **range**, not bit-width. A `float` can represent very large magnitudes (with limited precision), so every `long` value can be represented as a `float` value (possibly losing precision).

```
float add(float x) { return x + x; }

long x = 9223372036854775807L;
float y = add(x);           // allowed (widening)
```

In contrast, not every `float` value can fit into a `long` without loss or overflow, so narrowing is not implicit:

```
long add(long x) { return x + x; }

float x = 3.4e+38f;
// long y = add(x);         // compile error (narrowing)
long y = add((long) x);     // explicit cast required
```

14.3.5 Widening vs Narrowing

- **Widening (promotion)**: implicit conversion to a wider type (e.g. `int` to `double`). Usually safe in terms of range (but precision may still change).
- **Narrowing (demotion)**: conversion to a smaller type requires an explicit cast (e.g. `double` to `int`). This can lose information.

Common Mistake: Confusing "written for" vs "used by"

It is easy to read code incorrectly and assume conversions work both ways. Java only allows implicit assignment when the right-hand side can widen to the left-hand side.

```
double d = 5.0;
int i = 5;
d = i;  // ok: int <: double (widening)
i = d;  // error: double </: int (narrowing needs a cast)
```

14.4 Operators and Numeric Promotion

14.4.1 Arithmetic Operators and Division Pitfalls

Java supports the standard arithmetic operators: `+`, `-`, `*`, `/`, `%`. Importantly, **integer division truncates**.

```
double d;
int i;
i = 12;
d = i / 10;      // d becomes 1.0, not 1.2 (integer division first)
```

Fix it by forcing floating-point computation:

```
d = i / 10.0;      // 1.2, or
d = (double) i / 10;  // 1.2
```

14.4.2 Binary Numeric Promotion (Result Type of Expressions)

When operands of an arithmetic operator have different numeric types, Java applies **binary numeric promotion** to decide a common type.

A practical rule-of-thumb:

1. If either operand is `double`, convert the other to `double`.
2. Else if either is `float`, convert the other to `float`.
3. Else if either is `long`, convert the other to `long`.
4. Else convert both to `int`.

14.4.3 Logical Operators

For boolean expressions:

- `&&` (AND), `||` (OR): **short-circuit** operators
- `!` (NOT)

Java also has `&`, `|`, `^` which can act as boolean operators (non-short-circuit) or bitwise operators for integer types. Prefer `&&` and `||` for boolean control flow.

14.4.4 Increment and Decrement Operators

Java provides shorthand operators for adding or subtracting 1 from a variable: `++` (increment) and `--` (decrement).

The position of the operator relative to the variable determines *when* the change happens during an expression evaluation.

- **Pre-Increment (++x):** Increments the variable **first**, then uses the *new* value in the expression.
- **Post-Increment (x++):** Uses the *current* value in the expression **first**, then increments the variable afterwards.

```java
int a = 5;
int b = 5;

// Post-increment: Use 5, then increment a to 6
int x = a++;
// Result: x = 5, a = 6

// Pre-increment: Increment b to 6, then use 6
int y = ++b;
// Result: y = 6, b = 6
```

Standalone Usage

If the operator is used as a standalone statement (not part of a larger assignment or expression), there is effectively no difference between the two.

```java
x++;  // x becomes x + 1
++x;  // x becomes x + 1 (same result here)
```

Syntax Error: ++x++

You cannot chain increment operators like ++x++. This results in a **compilation error** ("Unexpected type").

Why?

1. Java parses this as ++(x++).
2. The inner part, x++, evaluates to a **value** (e.g., if $x = 5$, it returns 5).
3. The outer part tries to increment that result. However, the increment operator requires a **variable** (a storage location), not a raw number. You cannot write ++5.

```java
int x = 5;
// ++x++;    // ERROR: required: variable, found: value
x++; x++;    // Correct way to increment twice
```

14.5 Flow Control

Flow control statements determine *which* statements execute and *how many times* they execute. In Java, the condition in an `if` or loop must be a `boolean` expression (Java does **not** treat integers like `0/1` as false/true).

14.5.1 `if / else` and the Ternary Operator

Use `if` to execute a block only when a condition holds. Use `else if` for additional mutually-exclusive branches, and `else` as a fallback.

```
if (condition) { statements }
else if (condition) { statements }
else { statements }
```

Java evaluates the conditions from top to bottom:

- The **first** condition that evaluates to `true` has its block executed.
- All remaining branches are skipped.
- If none are true and an `else` exists, the `else` block executes.

Common pitfalls.

- **Forgetting braces**: Java allows a single statement without braces, but this is error-prone. Prefer braces for clarity.
- **Using assignment instead of comparison**: = assigns; use == to compare primitives.
- **Non-boolean conditions**: `if (time)` is invalid in Java (unlike in some other languages like Python).

The ternary operator ?:

The ternary operator is an **expression** (it produces a value), and is often used as a concise alternative to an `if/else` when selecting between two values.

```
result = (condition) ? valueIfTrue : valueIfFalse;
```

When to use it.

- Good for short, readable value selection (e.g. choosing a label).
- Avoid nesting ternaries; readability drops quickly.

14.5.2 `switch`

A `switch` selects one branch among many based on a **selector expression**. In introductory Java, the selector is commonly an `int`, `char`, or `String`. Each `case` labels a possible value; `default` is the fallback when no case matches.

```
switch (expr) {
    case V1:
        statements;
        break;
    ...
    default:
        statements;
}
```

By default, execution **falls through** from one case to the next until a `break` is reached:

- `break;` exits the `switch`.
- If `break;` is omitted, subsequent case blocks may run even if they do not match (sometimes intentional, often a bug).

Therefore, include `break;` at the end of each case unless fall-through is explicitly desired (and documented with a comment).

14.5.3 Loops: `while`, `do-while`, `for`

Loops repeatedly execute a block while a condition remains true.

`while`

A `while` loop checks the condition **before** executing the loop body, so the body may run **zero times**.

```
while (condition) { statements }
```

Use `while` when the number of iterations is not known in advance (e.g. "keep reading input until EOF", "repeat until valid input").

`do-while`

A `do-while` loop executes the body **first**, then checks the condition, so the body runs **at least once**.

```
do { statements } while (condition);
```

Use `do-while` when the body must run at least once (e.g. show a menu once before asking whether to repeat).

`for`

A `for` loop is the most common loop for "counting" iterations. It groups initialization, condition checking, and update in one line.

```
for (init; condition; update) { statements }
```

Execution order.

1. Execute `init` once.
2. Check `condition`; if false, stop.
3. Execute the body.
4. Execute `update`.
5. Repeat from step 2.

Enhanced `for` (for-each). When iterating over every element in an array (or other iterable), Java provides:

```
for (ElementType x : array) { statements }
```

This avoids manual indexing and off-by-one errors, but you cannot easily access the index without extra bookkeeping.

14.5.4 `break` and `continue`

`break`

`break;` immediately terminates the **nearest** enclosing loop or `switch`. Execution resumes at the first statement after that loop/switch.

`continue`

`continue;` skips the remainder of the current loop iteration and proceeds to the next iteration:

- In a `for` loop: jumps to the `update` step, then re-checks the condition.
- In a `while/do-while`: jumps to the condition check.

Use `continue` sparingly; excessive use can make control flow harder to follow. Often, a well-structured `if/else` block is clearer.

For example, a typical loop structure when processing an array is:

```
for (int i = 0; i < nums.length; i++) {
    if (nums[i] == 0) break;
    if (nums[i] % 2 != 0) continue;
    sum += nums[i];
}
```

This pattern shows three ideas:

- **Early termination** (`break`) when a sentinel value (like 0) is encountered.
- **Skipping unwanted elements** (`continue`) for filtering (e.g. ignoring odd numbers).
- **Accumulation** of the remaining elements (e.g. summing evens).

14.6 Arrays (vs Python Lists/Tuples)

In Java, an **array** is a **mutable** object that stores a **fixed number of elements** of the **same type** in a contiguous, indexable sequence. This contrasts with:

- **Python list:** dynamic-sized, can hold mixed types (at runtime), rich built-in operations.
- **Python tuple:** fixed-sized (conceptually immutable), can hold mixed types (at runtime).

Java arrays behave more like a *typed, fixed-length container*. The compiler enforces element type consistency.

14.6.1 Key Properties of Java Arrays

1. **Fixed size.** Once created, an array's length cannot change. If you need a resizable sequence, you typically use `ArrayList<T>` instead.
2. **Homogeneous element type.** If an array is declared as `int[]`, every element is an `int`. For reference types, e.g. `String[]`, every element is a reference to a `String` object (or `null`).
3. **Indexed access with bounds checking.** Elements are accessed with `arr[i]`. Java does runtime bounds checking; invalid indices throw `ArrayIndexOutOfBoundsException`.
4. **4. `length` is a field.** Arrays expose their size via the public final field `arr.length` (not a method, unlike Python's `len(arr)`).

14.6.2 Declaring, Creating, and Initializing Arrays

Syntax patterns. There are two equivalent declaration styles:

```
ElementType[] name;    ElementType name[];
```

Creating an array requires specifying its length or using an initializer list:

```
name = new ElementType[n];    ElementType[] name = {v1, v2, . . . };
```

Default values. When you allocate with `new`, Java initializes all elements to a default:

- numeric primitives (`byte, short, int, long, float, double`) → `0` / `0.0`
- `char` → `\u0000`
- `boolean` → `false`
- reference types (e.g. `String`) → `null`

This differs from Python, where an uninitialized variable has no default value unless you explicitly assign one.

14.6.3 Basic Operations

Reading and writing elements. Similarly to Python, the syntax for reading and writing elements is:

`arr[i]` reads element at index i `arr[i] = value;` writes element at index i

Iteration. There are two common styles:

- **Index-based** loop: best when you need the index (position), or need to write back into the array.

```
for (int i = 0; i < arr.length; i++) { ... arr[i] ... }
```

- **Enhanced for-loop (for-each)**: best for read-only traversal.

```
for (ElementType x : arr) { ... x ... }
```

Important note on mutation with for-each. In a for-each loop, `x` is a *copy of the element value* for primitives, so assigning to `x` does **not** modify the array.

For reference types, `x` is a copy of the reference; mutating the referenced object is possible, but reassigning `x` does not change the array slot.

14.6.4 Multidimensional Arrays

Java represents a "2D array" as an **array of arrays**. This means it can be:

- **Rectangular:** every row has the same length.
- **Jagged:** rows can have different lengths.

`ElementType[][] grid;` `grid[r][c]` accesses row *r*, column *c*

Jagged arrays. A jagged array is valid because each row is itself an independent array:

```
int[][] grid = { {1,2,3,4}, {5,6,7} };
```

Here, `grid.length` is the number of rows, and `grid[r].length` is the number of columns in row `r`.

Hence, comparing Java arrays to Python lists / tuples:

Feature	Java Array	Python List/Tuple
Size	Fixed at creation (`length`)	List: dynamic, Tuple: fixed
Element types	Single static type (compile-time enforced)	Can mix at runtime
Indexing	Bounds-checked, throws exception	Bounds-checked, raises exception
Common resizable alternative	`ArrayList<T>`	Built-in list
Multidimensional	Array of arrays (jagged allowed)	Nested lists/tuples

14.7 Basic Input and Output (I/O)

Java provides robust libraries for standard I/O operations, primarily found in the `java.util` and `java.io` packages.

14.7.1 Standard Input: `Scanner`

For reading input from the console (Standard Input), the `java.util.Scanner` class is the most common tool. It parses primitive types and strings from an input stream.

- **Setup:** Instantiate a `Scanner` object wrapping `System.in`.
- **Methods:** Use typed methods like `nextInt()`, `nextDouble()`, and `nextLine()`.
- **Common Pitfall:** Mixing token-based reads (like `nextInt`) with line-based reads (`nextLine`) can lead to skipped input because integer reads leave the newline character in the buffer.

```
Scanner sc = new Scanner(System.in);
System.out.print("Enter age: ");
int age = sc.nextInt(); // Reads integer token
```

14.7.2 Standard Output: `System.out`

Java uses the `System.out` stream to write to the console.

- `print()`: Writes text without a newline.
- `println()`: Writes text followed by a newline.
- `printf()` / `String.format()`: Provides C-style formatted output using format specifiers.

You can control output precision, alignment, and types using placeholders.

```
// %s = string, %d = integer, %.2f = float (2 d.p.), %% = literal %
String s = String.format("Score: %s %d %.2f%% %n", "Player1", 7,
    98.123); // Output: "Score: Player1 7 98.12% \n"
```

1. Common Conversion Characters (Verbs)

These are common verbs used in string formatting to represent various primitives.

Type	Specifiers
`byte, short, int, long`	`%d` (decimal), `%x` (hex), `%o` (octal)
`float, double`	`%f` (fixed-point), `%e` (scientific), `%g` (general)
`char`	`%c` (character), `%C` (uppercase)
`boolean`	`%b` (true/false), `%B` (TRUE/FALSE)
`String / Object`	`%s` (string representation)
`Other`	`%n` (platform newline), `%%` (literal %)

2. Width, Precision, and Flags Syntax: `%[flags][width][.precision]conversion`

- **Width & Alignment:**
  - `%8s`: Min width 8, right-aligned (default).
  - `%-8s`: Min width 8, **left-aligned**.
- **Padding & Grouping:**
  - `%05d`: Zero-pad to width 5 (e.g., `00042`).
  - `%+d`: Always show sign (e.g., `+42`).
  - `%,d`: Use locale-specific grouping separators (e.g., `1,000`).
- **Precision (Floating Point):**
  - `%.3f`: Round to 3 decimal places.
  - `%8.2f`: Total width 8, 2 decimal places.

14.8 The `String` Class

In Java, `String` is a **reference type** that represents a sequence of characters. A key characteristic of Java Strings is that they are **immutable**. Once created, their value cannot be changed. Any method that appears to modify a string actually returns a new String object.

14.8.1 Common Operations

Java provides a rich API for manipulating strings.

- `length()`: Returns the number of characters.
- `toUpperCase()` / `toLowerCase()`: Returns a *new* string with case converted.
- `indexOf(str)`: Returns the index of the first occurrence of a substring (or -1 if not found).
- `charAt(i)`: Returns the character at index i.

```
String txt = "Hello World";
System.out.println(txt.length());          // 11
System.out.println(txt.toUpperCase());     // "HELLO WORLD"
System.out.println(txt.indexOf("World")); // 6
System.out.println(txt.charAt(0));         // 'H'

// Concatenation
String firstName = "John";
String lastName = "Doe";
System.out.println(firstName + " " + lastName); // "John Doe"
```

14.8.2 Escape Sequences

To include special characters in a String literal, use the backslash (\).

Sequence	Name	Example Output
`\n`	Newline	Line break
`\t`	Tab	Indentation
`\"`	Double Quote	Prints "
`\\`	Backslash	Prints \

14.8.3 Concatenation Behavior

The + operator is overloaded. If *any* operand is a String, Java converts the other operands to Strings and concatenates them.

```
String x = "10";
int a = 20;

System.out.println(x + x);  // "1010" (String + String)
System.out.println(a + a);  // 40 (int + int)
System.out.println(x + a);  // "1020" (String + int -> String)
```

Critical: Comparing Strings

Never use == to compare the *contents* of two Strings.

- `s1 == s2`: Checks if they point to the **same memory location** (Reference Equality).
- `s1.equals(s2)`: Checks if they contain the **same characters** (Value Equality).

```
String s1 = "Hello";
String s2 = new String("Hello");

System.out.println(s1 == s2);       // false (different objects)
System.out.println(s1.equals(s2));  // true (same content)
```