Introduction to Java

Today's Topics

- Object-oriented essentials in Java
  - Classes vs. objects; fields, methods, visibility
  - Inheritance, interfaces, and composition (quick tour)
- Primitive types, control flow, and arrays
  - Value vs. reference semantics
  - Loops and branching patterns you will actually use
- Useful standard classes
  - `Math`, `String`, `StringBuilder`, basic regex
- I/O overview
  - `JOptionPane` dialogs, console I/O, simple file read/write
- Writing your own classes
  - Constructors, `toString()`, `equals()`, basic testing

Goal

Leave with a working mental model and code patterns you can reuse this week.

Java at a Glance

- **Platform independent:** Write once, run anywhere (JVM and bytecode).
- **Object-oriented:** Organize behavior and data together.
- **Batteries included:** Large standard library (`java.util`, `java.io`, `java.time`, Swing).
- **Ecosystem:** Build CLIs, GUIs, web servers, Android apps.
- **Tooling:** `javac`, `java`, IDEs, build tools (`maven`/`gradle`).

Why it matters

Understanding bytecode and the JVM explains performance, portability, and tooling choices.

From Source to Running Program

1. Edit `Model.java` (source) in a text editor or IDE.
2. Compile with `javac` to produce `Model.class` (bytecode).
3. Run with `java Model`; the JVM loads classes and executes.

Pro tip

Keep one `public` class per file; file name must match the class name.

Common gotcha

Package declarations must match the folder structure (e.g., `package a.b;` implies path `a/b/Model.java`).

Minimal Program

```
// Hello, Java --- minimal console example
public class HelloJava {
  public static void main(String[] args) {
    System.out.println("Hello, Java!");
  }
}
```

- Compile: `javac HelloJava.java` then run: `java HelloJava`
- `main` signature is exact: `public static void main(String[] args)`
- `System.out.println` vs `System.out.print` (newline vs none)

Try it

Modify the string; add another print; recompile; re-run. Observe the quick iteration loop.

Classes and Objects

- A **class** is a recipe (fields and methods); an **object** is an instance.
- One public class per `.java` file; name matches the class.
- `import` makes other packages available; `java.lang` is auto-imported.
- `main` gives an entry point for standalone execution.
- **Design rules:**
  - Prefer composition over inheritance for code reuse.
  - Keep fields `private`; expose minimal `public` API.

Import and main

```
import javax.swing.*; // import Swing GUI classes

public class HelloWorld {
  public static void main(String[] args) {
    // entry point of a standalone program
  }
}
```

- Use specific imports for clarity: `import javax.swing.JOptionPane;`
- Wildcard imports are fine in small demos but avoid in libraries.

Why it matters

Understanding imports prevents name clashes and improves readability.

References vs. Primitives

- **Primitives:** `byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`
  - Stored by value; fast; no methods on them directly.
- **References:** variables hold addresses to objects.
  - Methods act on object state through references.
  - Two references can point to the same object.
- **Implication:** Mutating an object via one reference is visible via others.

Gotcha

== compares references for objects; use `equals()` for content.

Primitive Type Ranges (overview)

Type	Typical Range / Notes
`byte`	-128 . . . 127 (8-bit)
`short`	-32768 . . . 32767 (16-bit)
`int`	32-bit signed (use for counters, indexes)
`long`	64-bit signed (timestamps, big counters)
`float`	approx. 6–7 decimal digits (graphics, large arrays)
`double`	approx. 15 decimal digits (default for real numbers)
`char`	UTF-16 code unit (be careful with emoji/surrogates)
`boolean`	`true`/`false`

Pro tip

Prefer `int` and `double` unless you have a reason otherwise.

Operators (selected)

- **Arithmetic:** `+ - * / %`
- **Unary:** `+ - ! ~` **Inc/Dec:** `++ --`
- **Comparison:** `< <= > >= == !=`
- **Logical:** `&&` (and), `||` (or), `!` (not)
- **Bitwise:** `& | ^« » »>`
- **Ternary:** `cond ? a : b`
- **Assignment:** `=, +=, -=, *=, /=, . . .`

Gotcha

`==` for floating point is fragile; compare with an epsilon.

Type Conversion (Widening)

- Mixed numeric ops widen to the larger type automatically.
- Assigning smaller to larger is allowed (e.g., `int` to `long`).
- Narrowing casts require explicit `(type)` and may lose data.

```
int n = 7;
long big = n;                 // widening OK
double d = n * 2.5;           // int promoted to double
int bad = (int) 2.9;          // 2: truncation!
```

Pro tip

Keep expressions in double when mixing ints and decimals.

Declaring and Setting Variables

```
int square;
square = n * n;

double cube = n * (double) square;

final double TAX_RATE = 0.0625; // constants via 'final'
```

- Initialize close to first use; keep scope minimal.
- Use `final` for constants; name them in ALL_CAPS.

Referencing and Creating Objects

```
String greeting = "hello"; // string literal uses string pool
java.awt.Point p = new java.awt.Point(2, 3); // constructor
```

- Literals may reuse interned strings; `new` always creates.
- Two references can point to the same object; mutate carefully.

Gotcha

`null` reference throws `NullPointerException` on dereference.

Control Flow

- **Blocks:** `{ ... }` create local scope.
- **Branching:** `if/else`, `switch` (supports `String`).
- **Loops:** `for`, `while`, `do-while`, enhanced `for`.
- **Break/continue:** use sparingly; keep loops readable.

Pattern

Prefer enhanced `for (T x : xs)` to avoid off-by-one errors.

Methods and Parameters

- `static` = class method; instance methods operate on the object state.
- Java is pass-by-value: object reference is copied (not the object).
- Return early to simplify logic; keep methods small and focused.

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

static void update(java.awt.Point p) { p.x++; } // caller sees change
```

Gotcha

Reassigning the parameter (e.g., p = new Point()) will not affect caller's reference.

The Math Class

```
double r = Math.sqrt(2.0);
double a = Math.pow(3, 4);   // 81.0
int m = Math.max(10, 20);    // 20
double rnd = Math.random();  // [0.0, 1.0)
```

- Common: abs, ceil, floor, round, min, max
- For random with distributions/control: use java.util.Random.

Strings and Builders

- `String` is immutable; concatenations create new objects.
- Use `StringBuilder` (single-thread) or `StringBuffer` (legacy, synchronized) for loops.
- Helpful methods: `substring`, `indexOf`, `startsWith`, `format`.

```
String s = "abc";
s = s.toUpperCase();     // creates a new String

StringBuilder sb = new StringBuilder();
for (int i = 0; i < 3; i++) sb.append("Hi ");
String result = sb.toString(); // "Hi Hi Hi "
```

Comparing Objects (Strings)

```
String a = "foo";
String b = new String("foo");

boolean sameObject = (a == b);      // typically false
boolean sameText   = a.equals(b);   // true
```

- Use `equalsIgnoreCase` when case-insensitive is desired.
- For ordering: `compareTo` (<0, 0, >0).

Gotcha

Never call methods on a potential null without checking: `"foo".equals(x)` is safe.

Splitting Text

- Legacy: `StringTokenizer` (works, but dated).
- Modern: `String.split(regex)` or `java.util.regex.Pattern`.

```
String[] parts = "a b  c".split("\\s+"); // ["a","b","c"]
// trim non-empty tokens
```

Pro tip

Precompile a regex `Pattern` if reused in loops for performance.

Wrapper Classes

- Wrap primitives as objects: `Integer`, `Double`, `Boolean`, `Character`, etc.
- Autoboxing/unboxing: `int` $\leftrightarrow$ `Integer` seamlessly in many contexts.
- Use wrappers for generics: `List<Integer> nums = new ArrayList<>();`

Gotcha

Avoid excessive boxing/unboxing in tight loops — creates garbage and slows code.

Defining Your Own Classes

- Keep fields `private`; expose minimal `public` methods.
- Provide constructors that establish invariants.
- Override `toString`, `equals`, and `hashCode` when appropriate.
- Keep methods short; write unit tests for behavior.

Design tip

Prefer immutable value objects where feasible (clearer reasoning, thread-safe by default).

Example: Person (excerpt)

```
// javadoc omitted for brevity
public class Person {
  private String givenName;
  private String familyName;
  private String idNumber;
  private int birthYear;

  private static final int VOTE_AGE = 18;
  private static final int SENIOR_AGE = 65;

  public Person(String first, String family,
                String id, int birth) {
    this.givenName = first;
    this.familyName = family;
    this.idNumber = id;
    this.birthYear = birth;
  }

  public Person(String id) { this.idNumber = id; }

  public void setGivenName(String given) { this.givenName = given; }
```

Arrays

```
float[] grades = new float[numStudents];
for (int i = 0; i < grades.length; i++) {
  total += grades[i];
}
System.out.printf("Average = %6.2f%n", total / numStudents);
```

- Enhanced loop: `for (float g : grades) total += g;`
- Arrays know their length; bounds are checked at runtime.

Gotcha

Array indices start at 0; ArrayIndexOutOfBoundsException on mistakes.

Simple GUI I/O with JOptionPane

```
String input = javax.swing.JOptionPane
                  .showInputDialog("Enter your name:");
javax.swing.JOptionPane.showMessageDialog(
  null, "Hello " + input
);
```

- Blocking modal dialogs; good for tiny demos, not full apps.
- For console I/O, consider Scanner on System.in.

Streams and Console I/O

- Console: `System.in` (bytes), `System.out` (bytes to text via println).
- Readers/Writers: character streams (`BufferedReader`, `PrintWriter`).
- Modern option: `java.nio.file.Files` for small files (`readString`, `writeString`).

Pro tip

Use try-with-resources to auto-close streams.

Reading a File (BufferedReader)

```
import java.io.*;
import java.util.StringTokenizer;

try (BufferedReader rdr =
        new BufferedReader(new FileReader(args[0]))) {
  String line;
  while ((line = rdr.readLine()) != null) {
    StringTokenizer tok = new StringTokenizer(line);
    while (tok.hasMoreTokens()) {
      String token = tok.nextToken();
      // process token
    }
  }
} catch (IOException e) {
  e.printStackTrace();
}
```

Alternative (modern)

Files.lines(Path) streams lines; remember to close it.

Splitting with String.split (Alternative)

```
String[] tokens = line.split("\\s+"); // regex: whitespace
for (String t : tokens) {
  // process token; consider t.trim()
}
```

- Use `split("\s*")` for simple CSV-like parsing.
- For real CSV/TSV, prefer a parser library.

Writing a File (PrintStream)

```
try (PrintStream ps = new PrintStream(args[0])) {
  ps.print("Hello");
  ps.print(i + 3);
  ps.println(" and goodbye.");
  ps.printf("%2d %12d%n", i, 1 << i);
} catch (FileNotFoundException e) {
  e.printStackTrace();
}
```

Alternative

Files.writeString(Path.of(name), text) for quick one-liners.

Questions?