GUVI | HCL
Skill Up. Level Up

Introduction to Java & Setup

Programming Foundation

Presented by
Nikhil Nair

Website
www.guvi.com

Meet & Greet

- Firstly, about me…
- Now, your turn!
- Pick any 2:
    - Prior experience with programming?
    - A non-technology hobby
    - What would you like from this course?

How do we interact?

- Quick replies: reactions via emoji, 1–3 word chat bursts
- “R” = repeat; “.” = done typing; “?” = question
- Time-boxed activities; gentle cold-calls to rotate voices
- Parking-lot for deep questions; notes & code shared after class

How We'll Work (Expectations for Me & You)

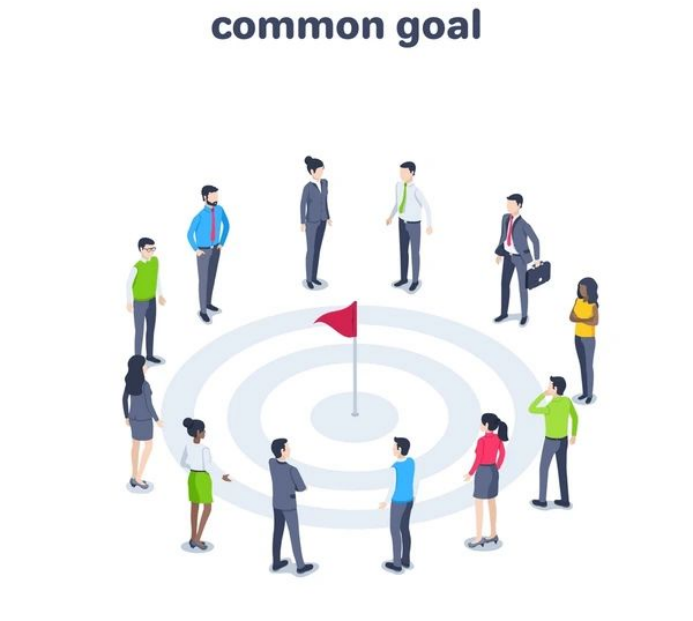
- From me
  - Reasoning-first teaching with live coding; concise, complete explanations
  - Code + notes published to the Github repository
  - Clear pacing (time-boxed segments) and a parking-lot for deep dives
  - Ensure your doubts are clarified
- From you
  - Participate: quick reactions/short chat replies; ask at least one question per session
  - Self-study commitment: 1 hour per lecture hour (target = ~9 hrs/week)
  - Use LLMs to understand the why; don't submit code generated by LLMs
  - Tools ready from next session: JDK 21, IntelliJ (free) or editor, GitHub account

Capability Roadmap (What You'll Be Able to Build)

- M1: Foundations (Java + Core DSA)
    - Write Java applications, use appropriate data structures & algorithms and reason about complexity
    - Build: CLI utilities and algorithmic solutions you can defend in interviews
- M2: Data & Backend (DB + Spring Boot + Kafka)
    - Model data, expose REST APIs, and move events through a message bus
    - Build: a data-backed API with auth, persistence, and background processing
- M3: Systems & Delivery (System Design + DevOps)
    - Design for scale/reliability and ship with observability and CI
    - Build: a deployable service with configs, logs/metrics, and a scaling story
- M4: Experience & Integration (Frontend + Spring + Gen-AI + Interview)
    - Stitch UI with your API, integrate an LLM feature responsibly, tell the story
    - Build: an end-to-end application (UI + API + DB + AI feature) and a portfolio narrative

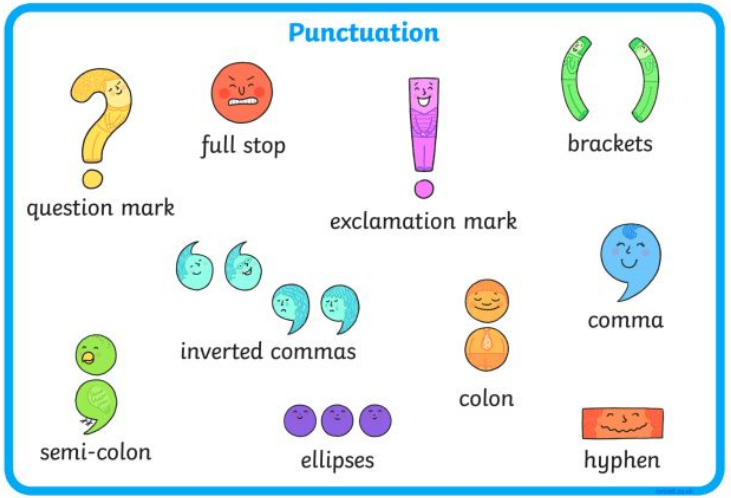
Tonight’s Destination

- How everyday language becomes precise, step-by-step instructions
- Who does what: JVM / JRE / JDK (clear mental picture)
- The shape of a tiny Java program and the main entry point
- Names that compile: identifiers, keywords, comments, conventions
- Variables & primitive types: declare, initialize, avoid common slips



What is a language?

- A language is a system of agreed-upon **symbols** (letters, words, sounds, gestures, or code tokens) and rules for combining them so a community can create and understand messages
- In everyday speech we also rely on **context** to fill gaps; people still understand us when details are missing
- A **programming language** reduces that reliance on context: symbols and rules must be explicit so a computer can follow them without guessing



Turning the precision dial

From a vague request to a precise procedure

- **Problem we're solving**: Consider a simple instruction like "Make chai"; it hides quantities, order, tools, and what to do when something goes wrong.
- **How we tighten it**: state the inputs/tools (ingredients, pot), the measurements (e.g., 200 ml water, 2 tsp tea), the order of steps, the decisions when conditions differ (no milk -> make black tea), and the finish condition.
- **Result**: a numbered list that can be followed without guessing; this is an algorithm (a precise procedure, not yet code).

Word order vs Java’s grammar

From a vague request to a precise procedure

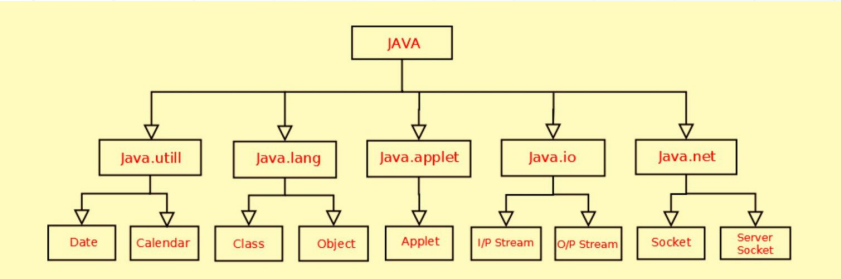
- In everyday languages, word order changes while meaning stays:
    - English: I drink chai (subject-verb-object)
    - Hindi: मैं चाय पीता/पीती हूँ (verb appears at the end)
    - Malayalam: ഞാൻ ചായ കുടിക്കുന്നു (verb appears at the end)
- In Java, the order inside a statement is fixed. A variable declaration must be: type name = value;
    - int age = 28;
    - age int = 28; (Java compiler cannot parse this)
- Changing order or punctuation in code changes the meaning or prevents the program from compiling

Scripts vs. Source Code

- A script (e.g., Devanagari, Bengali, Tamil, Kannada) is a writing system: the same sentence can be written in different scripts without changing the message for a human reader
- **Source code** is text written in a programming language and saved in a file (for Java, a .java file). The text follows strict rules so that tools can parse it consistently
- Your **editor** saves the file as plain text (typically UTF-8). Compilers read those bytes and look for language tokens; they do not infer missing context.
- **Takeaway**: scripts are for human readability across cultures; source code must be precise so both humans and tools interpret it the same way every time.

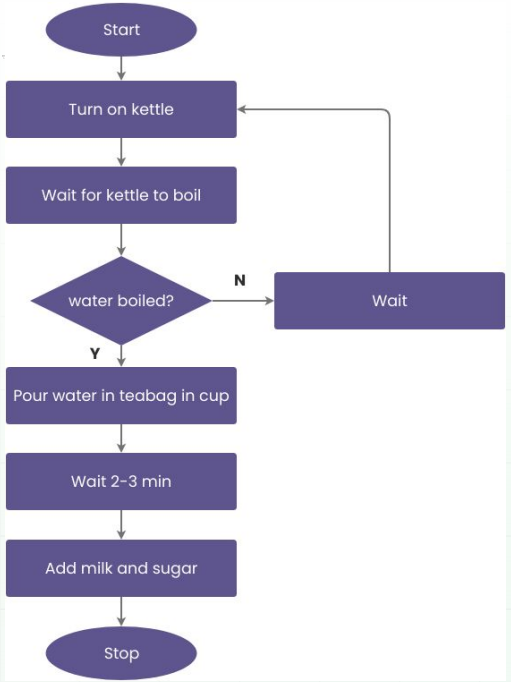
Reusing Vocabulary: Libraries

- A **library** is a collection of well-tested functions and types that solve common problems (files, dates, data structures, networking).
- Java ships with a large standard library (java.util, java.time, java.nio.file, etc.), so we can express ideas directly instead of hand-crafting low-level code.
- **Reusing** libraries improves clarity (“read this file as a string”), **reliability** (battle-tested code), and **security** (fewer home-grown mistakes).
- **Example** we’ll meet in an upcoming lecture:
    - Files.readString(Path.of("notes.txt")) – says exactly what we mean, in one line.



The Chai Algorithm

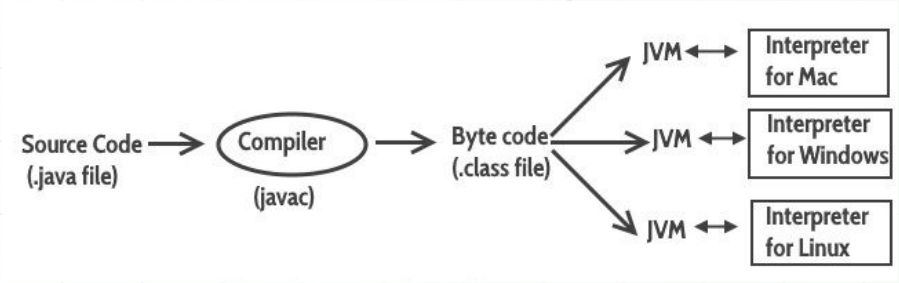
- **Definition**: An algorithm is a step-by-step procedure that takes inputs, follows clear, unambiguous steps in a fixed order, and terminates with an output
- **Why we need it**: everyday instructions hide quantities and choices; an algorithm exposes them so anyone (or a program) can follow the same path.
- **Chai example** (condensed):
    - Inputs: 200 ml water, 2 tsp tea, 100 ml milk, sugar (optional), pot, heat source, strainer.
    - Steps: boil water, add tea (2 min), add milk (1 min), strain to two cups
    - Decisions: no milk? skip that step; no sugar? continue.
    - Stop when poured; output = two cups of chai



From Ideas to Execution: Compiler & Runtime

How source code becomes something the computer can run

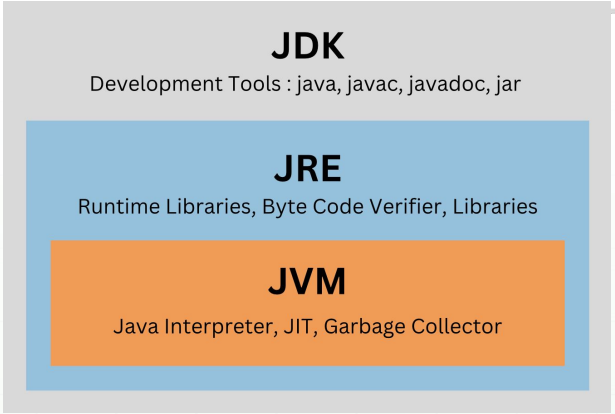
- You write source code (text in a .java file)
- The **compiler** (javac) checks rules and translates source into bytecode (.class)
- **Bytecode** is platform-neutral instructions designed for the Java Virtual Machine
- At run time, the JVM loads bytecode and executes it with help from standard libraries.
- **Two error moments**: compile-time (rules broken) vs run-time (program starts but fails while running)

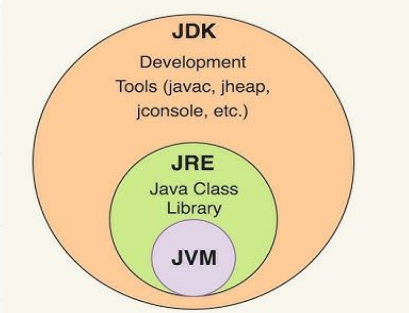


JVM, JRE, JDK — Who Does What

Tools vs. runtime vs. engine—clear mental picture

- **JDK** (Java Development Kit): everything developers use. Compiler, JShell, javadoc, and the runtime. Install this for development
- **JRE** (Java Runtime Environment): the class libraries and pieces a program needs at run time
- **JVM** (Java Virtual Machine): the engine that loads and executes bytecode on your OS
- Relationship: **JDK** = JRE + tools; the JVM sits inside the runtime and actually runs your code.





IntelliJ + JDK 21: First Run

- Open IntelliJ -> New Project -> Java, set SDK = JDK 21 -> Finish.
- In src, create App (Java class).
- Paste and run:

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

- If IntelliJ complains about SDK, set it via File → Project Structure → SDKs = JDK 21

Anatomy of a Java File

What lives in a .java file (and why)

- A Java file typically defines **one class**. If a class is marked public, the filename must match the class name.
- Statements end with ; and blocks use {} - punctuation is part of the grammar.
- You may see package and import lines at the top; they organize code and bring library names into scope (we'll use them soon, not today).
- A minimal, valid file can be as small as one class with a main method.

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

The main Method: The Program’s Front Door

Why public static void main(String[] args) looks the way it does

- **public**: the launcher can call it from outside your code.
- **static**: it runs without creating an object first.
- **void**: nothing is returned to the launcher.
- **String[] args**: optional inputs from the command line (like parameters to your program).
- You may also see **String... args** (same meaning, different syntax). You can have multiple classes with main; you choose which one to run.

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

Identifiers: Naming Things Clearly

Rules for valid names + conventions that keep code readable

- **Rules**: start with a letter, _, or $; then letters, digits, _ or $. Case-sensitive.
    - Cannot be a reserved keyword or the literals true, false, null.
- **Conventions**:
    - Classes/Interfaces: PascalCase (eg: BankAccount)
    - Methods/Variables: camelCase (eg: computeTotal)
    - Constants: UPPER_SNAKE with static final (MAX_USERS)
    - Packages: lowercase.dotted (com.example.app)
    - Clarity wins: prefer timeoutMins over t; avoid clever abbreviations; keep scope small when possible

Reserved Keywords: why some names are off-limits

Reserved words are part of Java's grammar, not your naming pool

- **Keywords**: class, public, static, if, for, return, try, catch and several others are reserved by the language; you can't use them as identifiers.
- **Literals**: true, false, null are values, not keywords; they're also not usable as identifiers.
- **Context/feature words** used in declarations (treat as reserved in practice): var (local inference), record, sealed, etc
- **Special case**: the single underscore _ is reserved (since Java 9) and cannot be used as a variable name
- **Rule of thumb**: if the compiler already uses a word to shape code, don't try to use it to name things
- Official Reference on Reserved Keywords

Comments: explain intent, not the obvious

Good comments say why this code exists or what must stay true

- **Use comments for intent and constraints**: decisions, trade-offs, invariants, units/assumptions, links to specifications or JIRA tickets
- **Avoid narrating the obvious** as it simply adds noise. Eg: i = i + 1; // increment i
- Prefer **Javadoc** on public APIs (classes/methods) to document usage: /** explains behavior, params, returns, throws */.
- **Keep comments honest**: update or delete when code changes; a wrong comment is worse than none
- **Practical tags**: TODO(username, date): ... for short-term work

```
// Good: Why 5s? Matches upstream timeout; avoids half-open sockets.
private static final Duration CONNECT_TIMEOUT = Duration.ofSeconds(5);

// Weak: obvious
count = count + 1; // increment count
```

Conventions that keep teams fast (and bugs down)

A few shared habits make code easier to read and harder to break

- **Layout & braces**: put { on the same line; indent consistently (2 or 4 spaces); no tab/space mixing.
- **Line length & wrapping**: aim ~100–120 chars; break method chains sensibly.
- **Imports**: avoid wildcard imports (import java.util.*;); keep imports ordered and unused ones removed.
- **Constants & magic numbers**: prefer static final named constants over raw numbers/strings in code.
- **Naming recap**: Classes/Interfaces PascalCase; methods/variables camelCase; constants UPPER_SNAKE; packages lowercase.dotted.
- **Initialize near first use**; keep variable scope as small as practical.
- **Prefer clarity over cleverness**: choose readable code over a one-liner that saves 2 characters.

```
// Before
if(flag==true){doThing(x,y,z);}

// After
if (flag) {
  doThing(x, y, z);
}
```

Variables & Primitive Types

Variables store values; the type defines what fits and how it behaves

- A variable is a named storage location; its type decides the kind of value it can hold and the operations that make sense.
- Java's primitive types (8):
    - **Whole numbers:** byte (very small), short (small), int (default), long (large counts/time)
    - **Decimals:** float (limited precision), double (default for decimals)
    - **Single text unit:** char (UTF-16 code unit; use String for words/sentences)
    - **Logic:** boolean (true / false)
- **Good fits**
    - Use **int** for most counters/indexes; switch to **long** when values can exceed ~2.1 billion.
    - Use **double** for measurements (distance, averages); avoid for money - we'll use *BigDecimal* later.
    - Use **char** for a single code unit; some emoji need two chars (we'll keep such text in String).
    - Use **boolean** for conditions; don't store 1/0 as "true/false".

Initialization, Literals, and Ranges

Make variables valid, pick correct literals, and respect limits

- **Definite assignment** (locals): a local variable must be given a value before use; fields get defaults, locals don't.
- **Literals**:
    - Integers can be decimal, binary (0b1010), or hex (0xFF); use _ for readability (1_000_000)
    - Add L for long values that exceed int range; add f for float decimals
    - char uses single quotes: 'A', '\u20B9'; strings use double quotes
    - **Logic:** boolean (true / false)
- **Ranges & overflow:**
    - Integers can store a minimum value of ~-2.1B and maximum value of ~2.1B
    - If the assigned value exceeds it, Java will wrap it without a warning
    - Use long or detect overflow when needed

```
Byte Overflow: 127 + 1 = -128
Byte Undeflow: -128 - 1 = 127

Short Overflow: 32767 + 1 = -32768
Short Undeflow: -32768 - 1 = 32767

Integer Overflow: 2147483647 + 1 = -2147483648
Integer Undeflow: -2147483648 - 1 = 2147483647

Long Overflow: 9223372036854775807 + 1 = -9223372036854775808
Long Undeflow: -9223372036854775808 - 1 = 9223372036854775807
```

Type Compatibility: Promotion vs. Narrowing

How do we combine numbers of different sizes safely and store the result in a data type that can hold it?

- **Step 1: What happens before the math**
    - When you use very small integer types (byte, short) in an expression, Java first treats them as int for the calculation.
- **Step 2: Safe, automatic widening**
    - If the two sides have different sizes (e.g., int with long, or int with double), Java automatically converts the smaller to the larger type before computing
    - This is safe because the larger type can represent everything the smaller one can
- **Step 3: Risky shrinking (requires your decision)**
    - Converting a larger type to a smaller one (for example, long to int, double to int) is not automatic
    - You must write a **cast**, and the value can change (big numbers clip; decimals lose the fraction
- **Step 3: The result must fit where you store it**
    - Even if the expression runs in a large type, **assigning** it to a smaller variable can still lose data.
    - Choose a big enough destination type, or convert **before** you compute so the math happens in a bigger “bucket”

GUVI | HCL
Skill Up. Level Up

Wrap-Up & What's Next

You can now explain the "why" behind Java's first steps

- Natural language to **precise procedure** (algorithm) without guessing
- **How code runs**: source to compiler to bytecode to JVM at runtime
- Clear roles: **JDK** (tools), **JRE** (runtime pieces), **JVM** (executes bytecode)
- First file anatomy, the **main** entry point, and valid names
- **Primitives & variables**: declare, initialize, pick safe types, avoid overflow
- **Tomorrow**: control flow (if/switch/loops), method basics, and short practice sets.