ReviseHub

FREE SAMPLE

ReviseHub — sample notes

A free taste of the library: selected AI Engineering, Java and System Design notes. The full library has 958 notes across 11 subjects.

AI Engineering

Building AI apps end to end — LLMs, prompt engineering, RAG, agents, deployment and optimization.

What is Prompt Engineering?

Beginner · Prompt Engineering · Interview-oriented notes

Core idea: Prompt engineering is the engineering work of specifying intent, inputs, constraints, and outputs so a language model behaves reliably enough for the job. The model weights are identical across a good and bad prompt; only the framing changes.

Key points

- Not clever phrases. A weak prompt produces inconsistent formats, missed edge cases, and unsupported details that only surface with real inputs.
- API analogy: an LLM call is a set of signals — instructions, constraints, examples, retrieved docs, tool schemas, formatting cues. Prompt engineering is choosing them intentionally.
- SQL comparison: `SELECT * FROM users` is easy; a correct, performant, edge-case-safe query takes skill — prompts are the same.
- Prompt engineering removes ambiguity, states the task practically, and makes output easier for the model **and** downstream code.

Why it matters — quality, cost, reliability

- **Quality gap:** stronger models still perform better with explicit task/format/evidence/constraints. Production systems (support tickets, code gen, structured extraction) cannot tolerate vague answers.
- **Cost:** APIs charge by token; bloated/retried prompts cost more. A concise engineered prompt often improves results *and* lowers cost.
- **Reliability:** a prompt that works on 10 hand-picked examples may fail on partial requests, copied logs, mixed languages, malformed data, adversarial inputs. A good prompt defines output, handles edge cases, states what to do when data is missing, and stays parseable.

Illustrative classification at 100,000 items/day (simplified $3/1M-token rate):

Approach	Tokens/Request	Accuracy	Daily Cost ($3/1M)
Naive prompt	~500	78%	$150
Engineered prompt	~300	94%	$90
Engineered + few-shot	~450	97%	$135

Evolution (each step solves a failure mode)

- **Zero-shot (2020):** GPT-3 showed in-context learning from instructions alone; limits — misunderstood tasks, inconsistent formats, shallow answers.
- **Few-shot (2020):** show a few examples; helps the model infer the pattern. Still one of the most effective/reliable techniques.
- **Chain-of-thought (2022):** examples with intermediate reasoning improve math/logic/multi-hop tasks. Lesson: complex tasks benefit from decomposition and intermediate checks, not that users must see step-by-step reasoning.
- **Templates & pipelines (2023):** prompts become templated, parameterized, versioned, tested, reused (LangChain popularized this). Single prompts give way to multi-step pipelines.
- **Context engineering (2024+):** behavior depends on full context — system instructions, retrieved docs, history, tool definitions/results, user profile, external sources. **MCP (Model Context Protocol)** reflects giving models the right info/tools at the right time.
- **Automated optimization (2023+): DSPy** treats an LLM workflow as a program — declare inputs/outputs, supply

1. **Specificity beats cleverness:** say what you want, format, audience, constraints; remove unnecessary degrees of freedom.
2. **Decomposition beats complexity:** break multi-part tasks into numbered steps or chained calls (extract → identify → rank → report) so each stage is testable.
3. **Iteration is part of the process:** draft → test (representative + adversarial) → inspect failures → refine → repeat. Treat prompts as versioned, reviewed, observable code.
4. **Evaluation turns guessing into engineering:** test cases, a definition of "good", version comparisons, tracked failure modes. Without eval, prompt changes are opinions.

From prompt engineering to context engineering

- Production call: user message ~50 tokens, full context maybe 10,000+ tokens. The challenge becomes what to include/omit/order/retrieve at runtime.
- Context engineering questions: which docs, in what order, how to trim/summarize history, how to write tool descriptions, what goes in system vs user prompt.
- **Prompt engineering is the foundation; context engineering extends it** across the entire context window.

Interview angle

- Frame prompt engineering as *specification + reliability engineering*, not "prompt tricks."
- Be ready to justify investment with the quality/cost/reliability triad and the token-cost table.
- Know the timeline and *what failure each technique fixes* (zero-shot → format drift; CoT → reasoning; templates → reproducibility; context engineering → wrong/missing facts; DSPy → manual tuning).
- Pitfall: claiming a better model removes the need for prompts — it does not supply business rules, data contracts, or definitions of correct.
- Rule of thumb: "prompt engineering writes the instruction; context engineering assembles the evidence around it."

Anatomy of an Effective Prompt

Intermediate · Prompt Engineering · Interview-oriented notes

Core idea: An effective prompt is a small contract for the model call — what to do, what info to use, what limits to respect, what output shape to produce. Improve prompts by testing real failures, not guessing.

The five components

1. **Role / persona:** nudges language, priorities, detail level (e.g., "senior backend engineer" pulls in queues, delivery guarantees, fan-out, idempotency). It changes assumptions, not just tone.
2. **Task / instruction:** give an action, not a topic.

- Vague: "Tell me about database indexing."
- Clear: "Explain how B-tree indexes work in PostgreSQL. Cover structure, lookup traversal, and which query types benefit most."

3. **Context:** background info. Without it the model fills gaps with generic assumptions that may not match your product/users/data (e.g., "Upload failed. Error code 413." vs a nurse-friendly explanation).
4. **Format specification:** structure the output. In production this is not cosmetic — downstream code, UI, validators, analytics depend on response shape. Prefer provider structured output / schema validation.
5. **Constraints / guardrails:** length, on-topic, product/safety rules. Explicit behavior boundaries; **not** a security boundary, but give validators something to check.

> Add the part that fixes the failure you see. A quick internal query may need only a task; a customer-facing workflow usually needs task + context + format + constraints (+ examples).

System vs user prompts

- **System prompt:** stable behavior — role, boundaries, style defaults, non-negotiable rules. Application policy, not user-visible.
- **User prompt:** the specific request and request-specific data.

Put in System Prompt	Put in User Prompt
Role/persona	Specific question/task
Behavioral constraints	Request-specific context
Output format defaults	One-off format requirements
Constants across requests	Things that change per request
Safety guardrails	User data/documents

System prompt persistence: in stateless chat APIs the system prompt is sent every call. A 500-token system prompt across 20 turns = 10,000 repeated input tokens. Keep it clear, stable, short.

The specificity spectrum

- L1 too vague ("Write something about APIs") → any output.
- L2 slightly better ("Write a blog post about REST APIs") → format+topic, no audience/length/depth.
- L3 good for most use cases → audience, length, scope, example domain.
- L4 highly specific / production quality → adds sub-topics, tone, no-undefined-jargon rule.

and output is brittle.

Use Case	Recommended Level	Why
Creative writing	2–3	Room to be creative
Technical documentation	3–4	Accuracy and structure
Data extraction / parsing	4	Predictable structured output
Customer-facing chatbot	3–4	Consistent tone, natural
Code generation	4	Precision in requirements, flexible implementation

Writing clear instructions

- **Use imperative verbs.** "Explain…", "List the top 5…", "Compare…" beats "Can you help me with…", "I was wondering if…", "Maybe you could…".
- **Decompose complex tasks** into numbered steps — a checklist the model is less likely to skip.
- **Specify output format explicitly** — show a template rather than describe it; still enforce with parsers/validators.
- **Use delimiters for input data** (e.g., `---BEGIN REVIEW---` / `---END REVIEW---`) to mark the instruction/data boundary. Delimiters do **not** stop prompt injection; pair with instruction hierarchy, output validation, least-privilege tools.

Few-shot prompting

- **Zero-shot:** instructions only. **One-shot:** one example. **Few-shot:** two or more.
- Ticket-classification example: a fifth example classifying a login issue as `ACCOUNT` (not `TECHNICAL`) teaches local routing policy — identity/access issues go to the account team.
- **Examples help when:** business-specific format/scheme, consistent edge-case handling, style easier to show than describe, or a smaller/less capable model.
- **Examples hurt:** token cost (5 × 50 = 250 tokens per request), overly similar examples (literal copying), contradictory examples.
- Rule: start zero-shot, test, then add examples for *observed* failures — distinct edge cases, not repeats.

Iterative prompt development

1. **Draft** based on required behavior.
2. **Test** on multiple inputs, especially edge cases.
3. **Analyze** failures/drift.
4. **Refine** to fix specific failures.
5. **Repeat** until consistent.

- Product-description example: v1 failed on length (300+ vs 100–150 words), salesy tone, invented features → v2 constrains length/tone/facts; test edge cases (few features, many features, technical jargon). After launch, update prompt **and** eval set together.

Interview angle

- Present the five components as a checklist; name which component fixes which failure.
- Likely follow-ups: "Isn't role just flavor?" (no — changes assumptions), "How much detail is too much?" (specify only outcome/safety/parseability-critical details).
- Pitfall: stuffing all five parts into every prompt; bloating the system prompt; treating delimiters as security.
- Rule of thumb: **start zero-shot, add examples only for measured failures; specify outcome, leave harmless choices flexible; treat prompts as versioned code.**

Java

Full course theory plus interview Q&A banks.

Introduction to Java

Beginner · Java Basics · Interview-oriented notes

1. What Is Java?

Java is a high-level, class-based, object-oriented programming language designed to have as few implementation dependencies as possible.

It is best understood as three connected pieces:

1. The Java language — what the developer writes
2. The bytecode — the intermediate format the compiler produces
3. The JVM — the engine that executes the bytecode on a specific machine

A program written once runs on Windows, Linux and macOS without modification. This single idea drives almost every design decision in Java.

2. The Problem Java Was Built to Solve

Before Java, languages like C and C++ compiled directly to the machine code of one specific CPU and operating system.

This created three problems:

1. A program written for Windows had to be recompiled for Linux or Mac.
2. In many cases the code itself had to be modified, not just recompiled.
3. Manual memory management produced crashes and security holes.

The team led by James Gosling at Sun Microsystems began work on this problem in 1991 (the Green Project), originally calling the language Oak. In 1995 it was renamed Java (after Java coffee) and released. Oracle acquired Sun Microsystems in 2010 and maintains Java today.

Java's answer: compile to an intermediate format — bytecode — that a per-platform engine (the JVM) translates for its own machine.

3. The Journey of a Java Program

A Java program travels through four steps:

1. The developer writes source code in a `.java` file.
2. The compiler (`javac`) converts the source into a `.class` file containing bytecode.
3. The JVM loads the bytecode, verifies it, and translates it into machine code for the current CPU.
4. The machine code executes.

```
Hello.java → [javac] → Hello.class (bytecode) → [JVM] → machine code → execution
```

```
// Hello.java — source code written by the developer
class Hello {
 public static void main(String[] args) {
 System.out.println("Hello, Java!");
 }
}
```

The meaning of `class`, `main` and `String[] args` is covered in an earlier note and an earlier note. This snippet is shown only to make the journey concrete.

> Important javac never compiles for a chip. It compiles for the JVM. Only the JVM speaks to the actual machine.

---

4. What Is Bytecode?

Bytecode is the intermediate instruction set of the JVM, stored inside `.class` files. It is not tied to any CPU. Any conforming JVM can run the same `.class` file.

Aspect	Machine Code	Bytecode
Written for	A specific CPU (x86, ARM)	The JVM
Produced by	C/C++ compilers	`javac`
Portable	No	Yes
Executed by	The CPU directly	The JVM
Contains	Native CPU instructions	JVM instructions

---

5. Write Once, Run Anywhere (WORA)

WORA is the practical result of the bytecode design.

1. Compile once → one `.class` file.
2. Copy the `.class` file to any machine.
3. That machine's JVM executes it.

The key asymmetry:

- **Bytecode** — platform independent
- **JVM** — platform specific. A separate JVM build exists for each OS.

> Interviewer's perspective The classic question is "How is Java platform independent if the JVM itself is platform dependent?" The answer — Java achieves platform independence *through* a platform-dependent JVM. Each OS gets its own JVM, but all of them understand the same bytecode.

---

6. Compiled, Interpreted, or Both?

Java is a hybrid.

1. `javac` compiles `.java` source into `.class` bytecode.
2. The JVM interpreter starts executing bytecode directly.
3. The JIT (Just-In-Time) compiler watches which code runs often and translates those hot regions into native machine code at runtime.

Stage	Component	Input	Output
Compilation	`javac`	`.java` source	`.class` bytecode
Execution	Interpreter	Bytecode	Execution, line by line
Execution	JIT compiler	Hot bytecode regions	Native machine code at runtime

This is why "Java is interpreted" and "Java is compiled" are both half-truths. It is compiled to bytecode, and interpreted/JIT-compiled to machine code at runtime.

---

7. Is Java Purely Object-Oriented?

No. Java is object-oriented, not purely object-oriented.

Two things live outside the object model:

1. Primitive types — `int`, `double`, `char`, `boolean` and others are not objects.
2. Static members — static methods and variables belong to the class, not to objects.

Primitives exist for performance. Crisp interview answer:

> "Java is not purely object-oriented because primitive types and static members exist outside the object model. Everything else — including classes, methods, and interfaces — revolves around objects."

---

8. Key Features of Java

Each feature in one line:

- **Simple** — no pointer arithmetic, no manual memory allocation, familiar C-style syntax.
- **Object-oriented** — the program is a collection of interacting objects.
- **Platform independent** — one bytecode, many JVMs.
- **Portable** — bytecode moves across systems without source changes.
- **Architecture neutral** — primitive sizes are fixed; an `int` is 32 bits on every platform. (In C, `int` size varies by machine.)
- **Robust** — automatic garbage collection, compile-time type checking, no dangling pointers.
- **Secure** — no pointer arithmetic plus bytecode verification at class-load time.
- **Multithreaded** — threads are built into the language (`Thread`, `synchronized`).
- **High performance** — JIT compilation brings execution close to native speed.
- **Distributed** — networking libraries and remote invocation are built into the standard library.
- **Dynamic** — classes are loaded at runtime as needed.

Memory aid — group the features:

- Sentence for the rest: "Big Java Runs Secure Multithreaded Distributed High-performance Dynamic Applications" — Robust, Secure, Multithreaded, Distributed, High-performance, Dynamic.

---

9. Where Java Runs in the Real World

- **Enterprise backends** — banking, payment and trading systems (a large share of enterprise servers run Java; Spring is the dominant framework).
- **Android** — the app ecosystem began on Java.
- **Big data** — Hadoop, Kafka and Spark are built in Java or its ecosystem.
- **Developer tools** — IntelliJ IDEA, Eclipse and Android Studio.
- **Test automation** — Selenium bindings are widely written in Java.
- **Games** — Minecraft is the classic example.

> Interviewer's perspective When asked "Why learn Java in 2026?", answer with scale: enterprise backends, Android heritage and the big-data ecosystem keep Java among the most demanded languages.

---

10. Java vs C / C++

Aspect	Java	C / C++
Compilation target	Bytecode (JVM)	Machine code per platform
Portability	WORA via JVM	Recompile (often re-modify) per platform
Memory management	Automatic via Garbage Collector	Manual (`malloc` / `free` , `new` / `delete`)
Pointers	References only, no pointer arithmetic	Full pointer arithmetic
Multiple inheritance	Not for classes (interfaces instead)	Allowed for classes
Operator overloading	Not supported	Supported (C++)
Threading	Built into the language	Library-based
Runtime safety	Bytecode verification, GC	Programmer-managed

---

11. Myth vs Reality

Myth	Reality
"Java is slow"	JIT compilation makes it close to native for most workloads; only startup cost lags C++.
"Java and JavaScript are related"	Name similarity is marketing only. JavaScript is a different language.
"Java does not need compilation"	`javac` is always required; the output is bytecode, not an executable.
"GC means no memory bugs"	References held in static collections or caches can still leak objects.
"Java is dying"	Enterprise, big data and Android keep it a top language in demand.

12. Common Trap Questions

> Trap 1 — Inverted WORA "Java is platform independent because the JVM is platform independent." False. The JVM is platform *specific*; the bytecode is platform independent.

> Trap 2 — "int is an object" "Java is object-oriented, so `int` is an object." No. `int` is a primitive. `Integer` is its object wrapper.

> Trap 3 — "JVM runs .java files" No. The JVM executes `.class` bytecode files.

> Trap 4 — "Java has pointers like C++" No. Java has only references. Pointer arithmetic is not allowed — one of the reasons Java is called secure.

> Trap 5 — "Interpreted, therefore slow" Outdated. JIT compilation converts hot code paths to native machine code at runtime.

13. Points to Remember

1. Java compiles to bytecode, not machine code.
2. The bytecode is platform independent; the JVM is platform specific.
3. WORA works precisely because a JVM exists for every major platform.
4. Java is both compiled (`javac` → bytecode) and interpreted/JIT (JVM → machine code).
5. Java is object-oriented, not purely object-oriented: primitives and static members exist.
6. Primitive sizes are fixed across platforms — this is architecture neutrality.
7. Memory is managed by the Garbage Collector; there is no pointer arithmetic.
8. Security comes from the same design: no pointers + bytecode verification.
9. History: James Gosling, Sun Microsystems, 1995, originally Oak, now maintained by Oracle.
10. Next note: JVM, JDK, JRE and the first program — the natural interview follow-up.

14. Likely Interview Questions

Question	Crisp Answer
What is Java?	A high-level, class-based, object-oriented language that compiles to platform-independent bytecode executed by the JVM.
Why is Java platform independent?	Source compiles to bytecode; a JVM exists on each OS and runs the same bytecode.
Is the JVM platform dependent?	Yes — and that is exactly how Java achieves platform independence.
Compiled or interpreted?	Both. `javac` compiles to bytecode; the JVM interprets and JIT-compiles it.
What is bytecode?	The CPU-independent instruction set of the JVM, stored in `.class` files.
What is WORA?	Write Once, Run Anywhere — one compiled `.class` file runs on any machine with a JVM.
Is Java 100% object-oriented?	No. Primitives and static members live outside the object model.
Who created Java and when?	James Gosling's team at Sun Microsystems; released in 1995; originally named Oak.
Java vs C++ in memory handling?	Garbage collector vs manual management; no pointer arithmetic in Java.
Why is Java considered secure?	No pointer arithmetic + bytecode verification at class-load time.
Why is Java fast despite being interpreted?	JIT compiler converts hot bytecode into native machine code at runtime.

---

15. Final Summary

```
.java source → javac → .class bytecode → JVM (per OS) → machine code → execution
```

- Bytecode → portability
- JVM → the platform-specific engine
- JIT → performance at runtime
- Garbage Collector → robustness
- No pointers + verification → security

One-line takeaway: Java does not compile programs for a chip. It compiles them for the JVM — and the JVM adapts to every chip.

Write Your First Java Program

Beginner · Java Basics · Interview-oriented notes

JDK, JRE and JVM are the three layers a Java program passes through, and knowing exactly what each layer does is what separates "I write Java" from "I understand Java". This note covers what each of them is, how they nest inside each other, how to write, compile and run the first program, the errors beginners hit on day one, and the three Java editions.

1. The Three Layers of the Java Platform

When you install Java, you are really installing three nested components:

1. **JVM (Java Virtual Machine)** — the engine that executes bytecode on a specific machine.
2. **JRE (Java Runtime Environment)** — the JVM plus the standard class libraries (`java.lang` , `java.util` and the rest of `rt.jar` in older JDKs) that running programs need.
3. **JDK (Java Development Kit)** — the JRE plus developer tools (`javac` , `javap` , `jar` , `javadoc` , `jdb`).

The nesting is the whole story:

```
JDK ⊃ JRE ⊃ JVM
(dev tools) (libraries) (engine)
```

Memory aid — the nesting doll: the **D**evelopment kit holds the **R**untime which holds the **M**achine. You can run a program with the JRE, but you need the JDK to create one.

2. The JVM in Detail

The JVM does four jobs in sequence for every class it loads:

1. **Loading** — the class loader reads the `.class` file binary from disk/network.
2. **Verification** — the bytecode verifier rejects malformed or unsafe bytecode before any of it runs. This is half of Java's security claim.
3. **Execution** — the interpreter starts executing bytecode; the JIT compiler translates hot methods into native machine code.
4. **Memory services** — the JVM allocates objects in the heap and reclaims dead ones (Garbage Collection).

What the JVM is not: a compiler and not a text editor. Important

> Important The JVM speaks one language: bytecode. It never sees your `.java` source file. (Memory internals — stack, heap, method area — are covered in an earlier note.)

3. The JRE in Detail

The JRE = the JVM + the standard libraries + supporting files needed to run Java programs.

What it has: the `java` launcher, class libraries, JVM. What it does not have: `javac` . You can run any compiled program with a JRE, but you cannot compile one with it.

> Interviewer's perspective The classic formulation is: "JVM is an abstract specification + implementation." The JVM *spec* is written as a standard document; Sun/Oracle/other vendors ship JVM *implementations* (HotSpot, OpenJ9, GraalVM). When an interviewer asks "what is the JVM?", the layered answer — class loader, bytecode verifier, interpreter/JIT, GC — is what earns marks.

---

4. The JDK in Detail

The JDK = the JRE + development tools. It is what you install to write Java.

Tool	Purpose

- `java` — Launches the JVM on compiled classes (also runs single-file source, Java 11+)
- `jar` — Packages `.class` files and resources into a `.jar` archive (a ZIP with a manifest)
- `javadoc` — Generates HTML documentation from doc comments
- `jdb` — Simple command-line debugger
- `javap` — Disassembler — shows the bytecode inside a `.class` file

Two JDK families exist:

1. **Oracle JDK** — Oracle's commercial build (license-restricted for production since 2019).
2. **OpenJDK** — the open-source reference implementation; virtually all vendors (Eclipse Temurin, Corretto, Amazon, Microsoft) build on it.

> Important Since Java 11, Oracle does not ship a separate JRE distribution — the JDK is the standard install, and the JRE live on inside it as its runtime core.

---

5. JDK vs JRE vs JVM

Aspect	JDK	JRE	JVM
Full form	Java Development Kit	Java Runtime Environment	Java Virtual Machine
Contains	JRE + dev tools	JVM + libraries + launcher	Communication with the OS only
Who uses it	Developers	End users running Java apps	The JRE/user — it is the engine
Compiles code	Yes (`javac`)	No	No
Runs bytecode	Yes	Yes	Yes — actually does the executing
Ship with	Compiler, debugger, docs tools	Runtime classes	Specification + implementation

Reading the table: compilation flows downward (JDK → JRE → JVM executes). JVM never compiles your source; `javac` does, and the JVM runs the result.

---

6. JSE vs JEE vs JME

Java is delivered in three editions, each targeting a different runtime environment:

JSE	Java Standard Edition	Desktop and standalone applications	Core libraries, Swing/JavaFX, the Java taught in these notes
JEE	Java Enterprise Edition	Large web/server applications	Servlets, JSP, REST services; renamed **Jakarta EE** in 2017 (moved from Oracle to the Eclipse Foundation)
JME	Java Micro Edition	Embedded/resource-constrained devices	Older feature phones, set-top boxes, Java Card

The layering: JEE builds on top of JSE (every enterprise API uses the standard libraries underneath); JME was a separate, smaller profile.

> Interviewer's perspective "What is the difference between Java SE and Java EE?" — Java SE is the core language + standard libraries; Java EE (now Jakarta EE) is a set of specifications on top of SE for web/enterprise work (servlets, transactions, persistence). Spring Framework is the dominant successor in practice — most "Java" backend jobs today are Spring, i.e. the JEE-style space, but Spring does not require Jakarta EE itself.

While JME and applets are essentially historical, Java SE knowledge remains the substrate for every edition. That is why these notes focus on Java SE.

---

7. Writing the First Program

```
// Hello.java — the file name must match the public class name exactly
public class Hello {
 public static void main(String[] args) {
 System.out.println("Hello, Java!");
 }
}
```

Anatomy, piece by piece:

1. `public class Hello` — the container of code. A `.java` file may hold several classes, but only one may be `public`, and the file must be named after it (`Hello.java`).
2. `public static void main(String[] args)` — the entry point the JVM looks for. Exactly one method with this exact signature is required in the class you launch.
3. `System.out.println("Hello, Java!")` — `System` is a class in `java.lang`, `out` is its standard-output print stream, `println` is the method that prints one line.

> Interviewer's perspective "Does the JVM search the whole file for an entry point?" No — it looks for the specific signature `public static void main(String[] args)` in the **class you name** on the `java` command line. Rename `main` to `Main` or drop `String[]` and compilation still works but the launch fails.

---

8. Compiling and Running

```
$ javac Hello.java # compiles → Hello.class (bytecode) — produced for the JVM
$ java Hello # launches the JVM on the Hello class
Hello, Java!
```

```
$ java Hello.java # compiles in memory and runs in one step
```

What each stage did:

Stage	Command	Result
Compilation	`javac Hello.java`	`Hello.class` — bytecode, platform independent
Execution	`java Hello`	JVM loads, verifies and executes the bytecode
Modern single-file run	`java Hello.java`	One step — convenient for scripts and learning

> Important `javac` never compiles for a chip. It compiles for the JVM. Only the JVM speaks to the actual machine — exactly the design covered in an earlier note.

---

9. Common Beginner Errors

- **`javac: command not found`** — JDK not installed or not on PATH — Install a JDK (e.g., Eclipse Temurin/OpenJDK) and check PATH
- **`class Hello is public, should be declared in a file named Hello.java`** — File name and public class name mismatch — Rename file (or class) so they match exactly
- **`Could not find or load main class Hello`** — Wrong class path, wrong name case, or a declared package — Run from the right directory; check spelling/case; account for packages
- **`Main method not found in class Hello`** — The signature was broken (e.g., not static, wrong arg type) — Exactly: `public static void main(String[] args)`
- **`error: ';' expected`** — Syntax slip — Read the reported line; fix the syntax

Common thread: Java is case-sensitive and file-name sensitive.

---

10. Myth vs Reality

- **"The JVM compiles my source code"** — `javac` (a JDK tool) compiles source to bytecode; the JVM executes it.
- **"Running `java Hello` runs `Hello.java`"** — It runs `Hello.class` (bytecode) — unless you use single-file source mode.
- **"A JRE is enough for development"** — You need the JDK: only it has `javac`.
- **"JVM and JRE are the same thing"** — JRE = JVM + libraries + launcher; JVM is the engine inside it.
- **"The `main` signature can vary (like C)"** — Java is strict: `public static void main(String[] args)` or the JVM refuses to launch.
- **"Java EE is dead"** — Officially renamed Jakarta EE and alive (via its specs and Spring heritage); JEE-style thinking powers most backend work.

---

11. Common Trap Questions

> Trap 1 — Inverted nesting: "The JVM contains the JRE, which contains the JDK." Exactly backwards. JDK ⊃ JRE ⊃ JVM. "Which is installed to run a program?" A JRE suffices. "Which is installed to compile?" You need a JDK.

> Trap 2 — "javac is a JDK tool or a JRE tool?" It is a JDK tool; JREs ship no compiler.

> Trap 3 — "java command is part of JRE or JDK?" Both — it is the launcher present in both; `javac` is the part only the JDK has.

> Trap 4 — "Is java.exe the JVM?" It is the JVM **launcher**; it loads the JVM library and hands it your class to execute.

> Trap 5 — "JVM executes .jar files?" A `.jar` is an archive of `.class` files (plus a manifest); the JVM still executes the classes inside it — the unit of execution is bytecode, not the archive format.

> Trap 6 — "JEE, JSE, JME are different languages?" No — one language, three editions targeting different environments.

---

12. Points to Remember

1. JDK = JRE + development tools; JRE = JVM + standard libraries; JVM = the bytecode engine.
2. Only the JDK compiles code (`javac`); both JRE and JVM can run it.
3. Bytecode is the contract between compiler and runtime — the whole portability design of an earlier note.
4. The JVM performs: loading → bytecode verification → interpreter/JIT execution → GC memory services.
5. The JVM spec is a document; HotSpot, OpenJ9 and GraalVM are implementations of it.
6. Source file must be named after its `public` class; case and file-name sensitivity are strict.
7. The entry point is exactly `public static void main(String[] args)` — the launcher will not start anything else.
8. Since Java 11, `java Hello.java` runs source in one step (good for learning and scripts).
9. Editions: JSE = core/desktop; JEE → Jakarta EE = web/enterprise specs; JME = embedded (historical).
10. OpenJDK is the open-source reference; Oracle JDK is its commercial sibling.

---

13. Likely Interview Questions

- **What is the JVM?** — The Java Virtual Machine: loads bytecode, verifies it, and executes it (interpreting + JIT compiling), managing object memory with GC.
- **What is the JRE?** — The runtime environment: JVM + standard class libraries + the `java` launcher. Enough to run, not to compile.
- **What is the JDK?** — The development kit: JRE + tools (`javac` , `jar` , `javadoc` , `jdb` , `javap`). The only layer that compiles.
- **JDK vs JRE vs JVM nesting?** — JDK ⊃ JRE ⊃ JVM — each layer is contained within the next one out.
- **Does the JVM compile Java code?** — No. `javac` (in the JDK) compiles to bytecode; the JVM executes it. JIT compilation inside the JVM is a different thing (runtime optimization).
- **Why doesn't the JVM see `.java` files?** — The JVM executes `.class` bytecode; source files are irrelevant to it.
- **What happens when we run `java Hello` ?** — The launcher loads `Hello.class` , checks for `main` , starts the JVM: class load → verify → interpret/JIT-execute → GC-managed memory.
- **Difference between JSE and JEE?** — SE is the core platform + standard libraries; EE (Jakarta EE) is enterprise specifications layered on top of SE (servlets, persistence, etc.).
- **What happened to Java EE?** — Renamed Jakarta EE in 2017 after moving to the Eclipse Foundation; the ideas live on in modern frameworks.

applications.

- **Which command produces bytecode?** — `javac` — and `java Hello.java` (Java 11+) does the equivalent in memory at launch.
- **Can we run Java without a JDK?** — Yes — with a JRE — but only compiled programs. No JDK, no `javac`, no development.

---

14. Final Summary

```
Hello.java --javac (JDK tool)--> Hello.class (bytecode)
 --java (JRE/JVM)--> load → verify → execute (+JIT) → GC-managed memory
```

- JDK → development (compiles)
- JRE → runtime (runs)
- JVM → the engine (executes bytecode, manages memory)
- Bytecode → the platform-independent contract between them
- Editions → JSE (core), JEE → Jakarta EE (enterprise), JME (embedded, historical)

> **Golden rule One installs a JDK to build, a JRE to run, and the JVM does the actual machine communication — never the compiler.**

One-line takeaway: the developer talks to the JDK, the user talks to the JRE, and both meet in the JVM through bytecode.

HLD · System Design

System-design concepts, patterns, technologies and 17 full design problems.

URL Shortener

Advanced · Basics and Realtime Messaging · Interview-oriented notes

---

Requirements

- Functional: shorten a long URL to a unique short URL; redirect short URL to original; custom aliases (nice-to-have); expiration with default + custom TTL; basic click counts.
- Non-functional: 99.99% availability; p99 redirect < 50 ms; globally unique, collision-free codes; scale to 100:1 read/write; durability (mappings never lost).

Back-of-the-envelope estimation

- Assumptions: 10M new links/day, 100:1 read:write → ~1B redirects/day.
- Writes: `10,000,000 / 86,400` ≈ **115 QPS**; peak 3x ≈ **350 QPS**.
- Reads: `1,000,000,000 / 86,400` ≈ **11,500 QPS**; peak 3x ≈ **35,000 QPS**.
- Per record ≈ **300 bytes**: short code 7 B, long URL 200 B, user ID UUID 36 B, timestamps/metadata ~50 B.
- Storage: 3.65B URLs/year; ~~1.1 TB/year; **~~5.5 TB over 5 years**.
- Short code (Base62 a-z, A-Z, 0-9): 6 chars = 62^6 ≈ **56.8B**; 7 chars = 62^7 ≈ **3.5T** → 7 chars gives decades of headroom.

Core APIs

Endpoint	Purpose	Notes
`POST /shorten`	Create short URL	Body: `long_url` (req), `custom_alias`, `expires_at`, `user_id`. Errors: 409 (alias exists), 400 (invalid URL), 401 (unauth). Returns `short_url`, `long_url`, `expires_at`, `created_at`.
`GET /{short_code}`	Redirect	302 default; 301 optional; 410 if expired; 404 if missing.
`GET /analytics/{short_code}`	Click stats	Basic counts only.

High-level architecture

- Components: Clients, Load Balancer (SSL termination, rate limiting, failover routing), URL Generation Service (write path), Redirection Service (read path), Database, Redis cache.
- Design read and write paths separately so each scales independently (reads dominate 100:1).
- Write flow: client `POST /shorten` → LB → Write Service validates URL + generates code → stores `short_code → long_url` → returns short URL.
- Read flow: client `GET /abc123` → LB → Read Service → check Redis → on hit return; on miss query DB then populate cache → check expiry → return 302.

Data model

`short_code`	String (PK)	Partition key; holds custom aliases too (same key space).
`long_url`	String	Original destination.
`user_id`	String	Creator (optional).
`created_at`	Timestamp	Creation time.
`expires_at`	Timestamp	Expiry.
`click_count`	Integer	Redirect count.
`is_custom`	Boolean	Custom alias flag.

- Secondary index on `user_id` to list a user's links. Optional Users table (`user_id` PK, `email`, `created_at`, `api_key`).
- SQL vs NoSQL: billions of records, simple key-value lookups, no joins, high read throughput, high availability → **NoSQL (DynamoDB / Cassandra)**.

Key deep dives & decisions

- **Unique code generation:**

Strategy	Pros	Cons	Best for
Hashing (SHA-256/MD5, truncate to 48 bits/6 bytes, Base62 → ~8 chars)	Deterministic, stateless, dedupes identical URLs	Truncation causes inevitable collisions; resolution needs a DB lookup	Services where dedup matters
Global counter (Redis `INCR` /etcd/Zookeeper)	Guaranteed unique, simple, sub-ms, deterministic length	Central bottleneck, SPOF for writes, guessable IDs	Small/medium traffic
Distributed ID (Snowflake-like)	Decentralized, k-sortable, in-memory, ~4M IDs/s/worker	Clock skew, needs coordination service (ZooKeeper/etcd)	Large-scale distributed

- Collision resolution: re-hash with salt (`sha256(url+salt)`) or append suffix — both sacrifice the stateless benefit (require a lookup).
- Sharded counters: 1024 counters; 64-bit ID = 10-bit shard + 54-bit local counter (`global_id = (shard_id << 54) | local_counter`), ~18 quadrillion per shard. Obfuscate sequential IDs with a **Feistel cipher / XOR mask** (reversible).
- Snowflake 64-bit layout: **41-bit** ms since custom epoch (~69.7 years; epoch 2025 → ~2094), **10-bit** worker (1024 workers via ZooKeeper/etcd lease), **12-bit** sequence (4096 IDs/ms/worker = 4.09M/s; wait if exhausted).
- **Redirect code choice: 302 by default** (click tracking, updatable/expirable); 301 fastest after first visit but loses control/analytics; 307 also temporary.
- **Read-path "waterfall":** Layer 1 browser cache (301 only) → Layer 2 CDN/edge (Cloudflare/Akamai/Fastly, ~10-50 ms) → Layer 3 Redis/Memcached (sub-ms) → Layer 4 DB (DynamoDB/Cassandra/ScyllaDB). On DB fetch, write back to Redis with TTL ~24 h.
- **Custom aliases:** validate regex `^[a-zA-Z0-9_-]+$`, length 3-50, reserved-word blocklist (`/api`, `/admin`, `/login`, etc.) before any DB op; enforce uniqueness with a single **atomic conditional write** (DynamoDB `attribute_not_exists`, Cassandra `INSERT... IF NOT EXISTS`, or a UNIQUE/PK constraint) to avoid check-then-write races.

Scaling, bottlenecks, failure handling

- Multi-region: one primary region for writes, read replicas everywhere, latency-based DNS.
- DynamoDB Global Tables replicate in seconds; Cassandra RF=3 with `LOCAL_QUORUM` reads.
- Graceful degradation if DB down: serve from cache, queue writes for retry, return stale data over errors.

heavy) → **hybrid**: passive on read path + low-frequency cleanup job. Cache TTL must respect expiry: `TTL_cache = min(link.remaining_lifetime, default_cache_ttl)`.

- Click counting: (1) direct increment (simple, write-heavy, contention), (2) **buffered counting** (Redis counter flushed periodically — recommended at scale), (3) **event streaming** (Kafka/Kinesis → Flink/Spark → Cassandra/Druid/ClickHouse) for real-time analytics. Separate sync redirect from async click logging.

Interview angle

- Lead with requirements + estimation; the 100:1 ratio drives separating read/write services and caching.
- Likely follow-ups: hash vs counter vs Snowflake (recommend based on scale), 301 vs 302 trade-off, custom alias race conditions (atomic write), cache/expiry consistency.
- Pitfalls: claiming Base64 (uses `+` / `/`); ignoring collisions after truncation; forgetting cache TTL vs link expiry; making the counter a SPOF without sharding.
- Rule of thumb: 7-char Base62 handles trillions; use a distributed ID generator for large scale, 302 for analytics, and `min(remaining_lifetime, default_ttl)` for caches.