

UNIT 1

Introduction to Java

(2 Hours) — Object Oriented Programming in Java (CACS2O4)

Java is one of the most widely used object-oriented programming languages in the world, powering everything from enterprise servers and Android applications to embedded devices. This unit introduces Java's origins, its place within the wider Internet and IT ecosystem, the working of the Java Virtual Machine, and the basic mechanics of writing, compiling, and running a Java program.

1.1 Definition and History of Java

Java is a high-level, class-based, object-oriented programming language that is designed to have as few implementation dependencies as possible. It is a general-purpose language intended to let developers write once, run anywhere (WORA) — compiled Java code can run on any platform that supports Java, without the need for recompilation.

History of Java

Java was developed by a team led by James Gosling at Sun Microsystems, beginning in 1991 under a project initially called the “Green Project.” The language was originally named Oak, after an oak tree that stood outside Gosling's office, and was later renamed Java — reportedly inspired by Java coffee. It was designed for programming consumer electronic devices, but the explosive growth of the World Wide Web in the mid-1990s made Java's platform independence extremely valuable for Internet-based applications, and Sun redirected the project towards the web.

Year	Milestone
1991	“Green Project” begins at Sun Microsystems; language first called Oak
1995	Publicly released and renamed Java (Java 1.0)
1997	Sun submits Java to standards bodies (ECMA, ISO) for standardization
2006	Sun releases most of Java as free and open-source software (OpenJDK)
2010	Oracle Corporation acquires Sun Microsystems and takes over Java
2017+	Oracle moves to a faster, time-based release cycle (new version every 6 months)

Note: Java's official mascot is Duke, and its slogan — coined by Sun — is “Write Once, Run Anywhere.”

1.2 The Internet and Java's Place in IT

The Internet is a global network of interconnected computer networks that allows information to be exchanged between machines located anywhere in the world. Before Java, software written for one type of computer (say, a Windows PC) generally could not run directly on another (say, a Unix workstation) without being recompiled for that machine's processor and operating system. This was a serious obstacle for the Internet, where a single web page might be viewed on thousands of different kinds of devices.

Java solved this problem by compiling programs into an intermediate, platform-neutral form called byte code, which any device running a Java Virtual Machine (JVM) can execute — regardless of the underlying hardware or operating system. This made Java a natural fit for the Internet age, and it quickly became central to server-side web applications, enterprise software, mobile development (notably Android), and embedded systems. Java's placement at the intersection of portability, security, and networking support is what secured its central role in modern Information Technology.

Why Java Suited the Internet

- ✓ Platform-independent byte code runs on any device with a JVM
- ✓ Built-in support for networking (sockets, URLs, RMI) from the very first release
- ✓ Strong security model — byte code is verified before execution (the “sandbox”)
- ✓ Automatic memory management (garbage collection) reduces a whole class of bugs

1.3 Applications and Applets

Java programs are broadly written in two forms depending on how and where they run: stand-alone applications and applets.

Applications

A Java application is a complete, self-contained program that runs directly on the Java Virtual Machine installed on a computer, independent of a web browser. It is started from a `main()` method and has full access to the local system's resources (files, network, and so on), subject to the security permissions granted to it.

Applets

An applet is a small Java program designed to be embedded inside a web page and run within a web browser (or an applet viewer). Applets do not have a `main()` method; instead they rely on a defined life cycle of callback methods (`init()`, `start()`, `stop()`, and `destroy()`) that the browser's

Java plug-in invokes automatically. For security reasons, applets traditionally run inside a restricted sandbox that limits access to the local file system and network.

Note: Applets have been deprecated and removed from modern Java releases and web browsers because of long-standing security concerns; modern Java web development instead relies on server-side frameworks, Java Web Start alternatives, or client technologies such as JavaFX and HTML5. Applets remain important to study because they illustrate Java's original design goals of portability and sandboxed security.

Aspect	Application	Applet
Entry point	main() method	init() / start() life-cycle methods
Execution environment	Runs directly on JVM / OS	Runs inside a browser or applet viewer
System access	Full access (as permitted by OS)	Restricted — runs in a security sandbox
Typical use	Desktop and server programs	Small embedded web programs (legacy)

1.4 Java Virtual Machine (JVM)

The Java Virtual Machine (JVM) is an abstract computing machine that provides a runtime environment in which Java byte code can be executed. It is the JVM — not the operating system directly — that loads, verifies, and runs compiled Java classes. Because every platform (Windows, Linux, macOS, and so on) has its own JVM implementation tailored to that platform's hardware and OS, the same byte code file can run unmodified on any of them, which is exactly what makes Java portable.

The JVM performs several key jobs: it loads class files, verifies that the byte code does not violate Java's security and memory-safety rules, interprets or compiles the byte code into native machine instructions (using a Just-In-Time, or JIT, compiler for performance), and manages memory automatically through garbage collection.

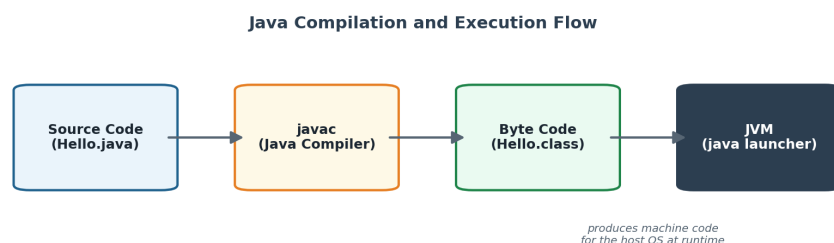


Figure 1.1 — Source code is compiled into byte code, which the JVM then executes on the host machine.

It is useful to distinguish three related terms that are often confused:

Term	Meaning
JDK (Java Development Kit)	Full development environment: compiler (javac), tools, and a JRE
JRE (Java Runtime Environment)	Everything needed to run Java programs: JVM plus core class libraries
JVM (Java Virtual Machine)	The engine that loads and executes byte code on a specific platform

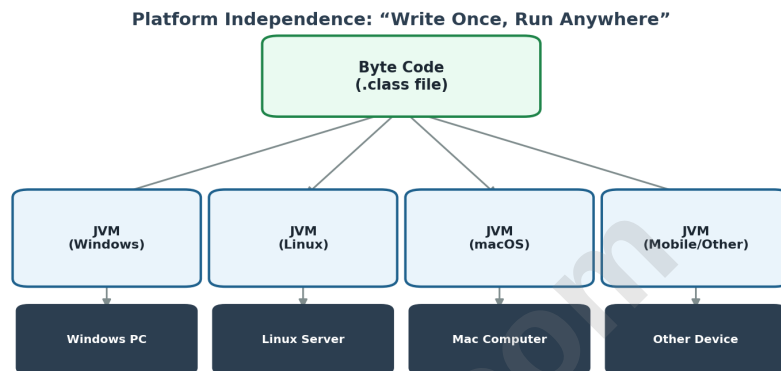


Figure 1.2 — The same byte code file runs unmodified on any platform that has a compatible JVM installed.

1.5 Byte Code — Not an Executable Code

When a Java source file (with extension `.java`) is compiled using the Java compiler, `javac`, the output is not native machine code that the processor can run directly. Instead, the compiler produces byte code — a set of platform-neutral instructions stored in a `.class` file. Byte code is an intermediate representation: it is highly compact and optimized for the JVM to interpret or Just-In-Time compile, but it is not directly executable by the underlying hardware the way, say, a compiled C program's machine code is.

This distinction is central to Java's design. Because byte code is not tied to any real processor's instruction set, it can be verified for safety and then executed by any JVM, on any hardware, without ever needing to be recompiled from source. Only the JVM's internal JIT compiler ultimately turns byte code into genuine machine code, and it does so at run time, on the machine where the program is actually running.

Byte Code vs. Machine Code

- ✓ Byte code is produced by javac from .java source files and stored in .class files
- ✓ Byte code is platform-independent; machine code is specific to one processor architecture
- ✓ The JVM interprets/JIT-compiles byte code into machine code only at run time
- ✓ Byte code is verified for safety by the JVM before it is ever executed

1.6 Procedure-Oriented vs. Object-Oriented Programming

Procedure-Oriented Programming (POP) structures a program as a sequence of instructions organized into functions or procedures, with data and functions kept largely separate. Global data is typically shared and manipulated freely by many functions across the program, which becomes harder to manage as programs grow larger.

Object-Oriented Programming (OOP), by contrast, organizes a program around objects — self-contained units that bundle data (fields) together with the behavior (methods) that operates on that data. Access to an object's internal data is normally controlled (encapsulated), and programs are built by modeling real-world entities and their interactions. Java is a purely object-oriented language in this sense — almost everything in a Java program lives inside a class.

Aspect	Procedure-Oriented Programming	Object-Oriented Programming
Basic unit	Function / procedure	Object (class instance)
Data handling	Data and functions are separate; data often global	Data and methods are bundled together (encapsulation)
Security	Less secure — data can be freely accessed	More secure — access controlled via access modifiers
Approach	Top-down decomposition into steps	Bottom-up composition of interacting objects
Reusability	Limited code reuse	High reuse via inheritance and composition
Examples	C, Pascal, FORTRAN	Java, C++, Python, C#

Note: Java also drops some POP-style features found in languages like C — for example, it has no standalone functions outside classes and no explicit pointers — precisely to enforce a consistently object-oriented, safer style of programming.

1.7 Setting Up Your Computer for the Java Environment

Before writing Java programs, a computer must have the Java Development Kit (JDK) installed, since it includes the compiler (javac) and the runtime (JRE/JVM) needed to compile and run programs.

1. Download the JDK for your operating system (e.g. Oracle JDK or an OpenJDK build such as Eclipse Temurin).
2. Run the installer, noting the installation directory.
3. Add the JDK's bin folder to the system PATH so that java and javac can be run from any directory.
4. Verify the installation from a terminal or command prompt.
5. Choose a text editor or IDE (VS Code, IntelliJ IDEA, Eclipse, NetBeans) to write code.

```
C:\Users\student> java -version
java version "21.0.2" 2024-01-16 LTS
C:\Users\student> javac -version
javac 21.0.2
```

Note: If these commands are not recognized, the PATH environment variable has not been set correctly and should be checked before proceeding.

1.8 Writing, Compiling, and Running a Simple Program

Every stand-alone Java application begins execution from a special method called main(). The example below is the traditional first program: it prints a short greeting to the console.

Example 1.1 — A First Java Program

Write the following code and save it exactly as HelloWorld.java — the file name must match the public class name.

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

Compiling and Running

Step 1 — Compile: from the folder containing the file, run `javac HelloWorld.java`. This invokes the Java compiler, which translates the source code into byte code and produces `HelloWorld.class`.

Step 2 — Run: execute `java HelloWorld` (note: no `.java` or `.class` extension). This starts the JVM, which loads `HelloWorld.class` and calls its `main()` method.

Output: Hello, World!

It is worth restating the three distinct stages a program passes through: writing the human-readable source code in a `.java` file; compiling that source into platform-neutral byte code using `javac`; and interpreting/running that byte code on the JVM using the `java` command, which converts it to native instructions at run time and executes them.

1.9 Handling Common Errors

Beginners typically encounter three broad categories of error while learning Java: compile-time errors (syntax mistakes caught by `javac` before the program can run), run-time errors (problems that occur while the program is executing, such as exceptions), and logical errors (the program runs and produces output, but the output is wrong because the logic itself is flawed).

Error / Message	Typical Cause	Fix
class HelloWorld is public, should be declared in a file named HelloWorld.java	File name does not match the public class name	Rename the file to match the class exactly (case-sensitive)
';' expected	Missing semicolon at the end of a statement	Add the missing semicolon
cannot find symbol	Misspelled variable/method name, or variable used before declaration	Check spelling and ensure the identifier is declared first
Error: Could not find or load main class	Running java on the wrong class name, or .class file missing	Recompile, and run java with the exact public class name
'javac' is not recognized...	JDK's bin folder is not in the system PATH	Add the JDK bin directory to PATH and reopen the terminal
NullPointerException (run-time)	Attempting to use an object reference that is null	Initialize the object before use; add a null check

Note: Reading the compiler's error message carefully — including the file name and line number it reports — is usually the fastest way to locate and fix a mistake.

Key Takeaways

- ✓ Java is a high-level, object-oriented, platform-independent language created by James Gosling and team at Sun Microsystems (1991-95), originally named Oak
- ✓ Java's byte code and JVM together give it the “write once, run anywhere” property, which suited it perfectly to the growing Internet
- ✓ Applications run stand-alone via `main()`; applets historically ran embedded in browsers within a security sandbox
- ✓ The JVM loads, verifies, and executes byte code; the JDK includes the compiler, and the JRE includes the JVM plus libraries
- ✓ Byte code (.class) is an intermediate, platform-neutral format — not directly executable machine code
- ✓ Java is object-oriented (data and methods bundled in objects) rather than procedure-oriented (data and functions kept separate)
- ✓ A Java program moves through three stages: write (.java) → compile with `javac` (.class byte code) → run with `java` on the JVM
- ✓ Common beginner errors fall into compile-time, run-time, and logical categories — reading compiler messages closely speeds up debugging

— *End of Unit 1* —