-damentals, that is.

In this chapter, we discuss the essential background knowledge to get us started on our journey to learn programming with Java.

We will cover:

- The components of computer systems
- Programming languages
- Programming environments
- Why study Java

But first, what are computers good for, anyway?

What is the purpose of a computer?

For a large part of my life I believed that computers were only good for scrolling social media. Computers were simply like a television to me: every website or app was a channel to flip through. I would occasionally text a friend through instant messenger, but I viewed computers primarily as a passive form of entertainment.

Sure, computers were useful on occasions when I had to type a paper for school. I could also use them to illegally download music and play games. That was cool. But eventually I had to ask myself: Why would serious adults use computers? What were they doing with them? Who were they emailing?

As I entered high school, I noticed some older students using computers in the media lab to draw pictures and edit photos. They were deftly switching between brushes and filters; transforming bland photos into fantastic scenes with ease.

In the band room, I saw musical keyboards hooked up to computers. A student played a few chords and the notes appeared on a digital music sheet in front of him. Then, he pressed the space bar on the computer keyboard and played back the music over the speakers.

In the journalism lab, students were writing and arranging articles on a screen that was laid out like a real newspaper. One student plugged a digital camera into the computer and dragged a photo from a Lacrosse game right into the center of the page. I was astounded. The software wasn't just a tool for writing; it was a powerful medium for sophisticated design and communication.

Of course, scattered throughout the campus were less-specialized computers where students entered data into Excel or perfected a PowerPoint.

Later, in college, I saw more rooms filled with even more specialized computers. Architecture students created 3D objects and building plans. Natural sciences students designed experiments using lab simulation software. Other students modelled complex calculations using black and white terminal screens; analyzing the composition of a far-away galaxy.

At some point during this process, I became interested in how all of these programs were created. I learned that programmers use their own specialized programs to write programs for other people.

> **The purpose of a computer is to help a person achieve whatever goal they want to achieve**. It doesn't matter if the goal is to cure cancer or watch eight hours of short-form videos, a computer **will** be there to help you achieve it!

However, because computers do whatever we ask, there is a risk that we become pacified by them. While learning to program is not a magic elixir to ward off the zombification process of mindless scrolling, it does unlock new ways of thinking about your computer. You come to view your computer as a powerful tool to help you build things.

The goal of this book is to help you understand and access the world inside your computer. While this is only the start of your programming journey, I hope you can leave this book with the foundational knowledge, tools, and confidence to use your computer to build whatever you can dream up. # 1.1 Computer Systems A computer system is comprised of hardware and software.

- **Hardware** can also be called a machine, device, or computer.
- **Software** can also be called a program, app, script, or tool.

A computer system has an **operating system**, which is the most important software on the machine.

Examples of computer systems

Here are some examples of computer systems that you may interact with (along with operating systems):

Device Category	Examples	Operating System(s)
Laptops	Apple MacBook Air M4 HP EliteBook 840 G11 Lenovo ThinkPad T14 G5	macOS Windows Linux
Desktops (Personal Computers)	Apple iMac Dell XPS Desktop Custom-built Gaming PC	macOS Windows Linux
Smartphones	Apple iPhone 16 Samsung Galaxy S25 Google Pixel 9 Pro	iOS Android
Tablets	Apple iPad mini A17 Pro Samsung Galaxy Tab S10 Amazon Fire HD 10	iPadOS Android Fire OS

Device Category	Examples	Operating System(s)
Smartwatches	Apple Watch Series 10 Samsung Galaxy Watch 7 Google Pixel Watch 3	watchOS Wear OS
Virtual Reality	Apple Vision Pro Meta Quest 3 Sony PlayStation VR2	visionOS Meta Horizon OS PlayStation OS
Smart TV / Streaming	Apple TV Google TV Roku TV Amazon Fire TV Edition	tvOS Android TV OS Roku OS Fire OS
Smart Appliances	Samsung Family Hub Refrigerator LG ThinQ Washer Google Nest Thermostat iRobot Roomba	Tizen OS customized Linux RTOS
Automotive Infotainment Systems	Systems with Android Auto Systems with Apple CarPlay Tesla Ford SYNC Android Automotive OS	Android iOS Linux-based QNX
Gaming Consoles	Sony PlayStation 5 Microsoft Xbox Series X/S Nintendo Switch Steam Deck	PlayStation OS Windows Horizon OS SteamOS (Linux-based)
Specialized and Embedded Systems	Medical Devices Industrial Control Systems ATMs, POS Systems, Routers, Drones Smart IoT Devices	RTOS (e.g., FreeRTOS, VxWorks) Linux Windows Embedded proprietary systems

Hardware

The core hardware of a computer falls into four categories: **Processor**, **I/O**, **Memory**, and **Disk**.

The **I/O** (Input/Output) category is further broken down into three sub-components: **Input Devices**, **Output Devices**, and **Networking Components**.

The following table provides more information about these components:

Category	Component	Function	Examples
Processor	Central Processing Unit (CPU)	The "brain" of the computer that directs and synchronizes all other components. It runs the operating system and applications, and manages tasks.	Intel Core i9, AMD Ryzen 9, Apple M4

Category	Component	Function	Examples
I/O	Input Device	A device that sends data to the computer.	Mouse, keyboard, touchscreen, microphone, webcam, scanner, Apple Pencil
I/O	Output Device	A device that presents data from the computer in an understandable format.	Display screen, VR headset, printer, speakers, headphones
I/O	Networking Components	Manages connections and handles data between devices.	Network Interface Card (NIC), Wi-Fi chip, Bluetooth, cellular modem, USB-C ports
Memory	Random Access Memory (RAM)	Volatile memory used to store data and instructions the CPU needs to access quickly. Data is erased when the power is off.	Crucial Pro 32GB DDR5, Kingston Fury 16GB, Apple Silicon Unified Memory 16GB
Disk	Storage Devices	Non-volatile memory for long-term data storage. Data is retained even when the power is off.	Hard Disk Drives (HDDs), Solid State Drives (SSDs), USB flash drives, optical discs

Can you think of how the four components of hardware apply to computer systems listed in the previous section?

Operating systems and software

The most important software running on a computer system is the **operating system (OS)**, which allows all other software to interact with the hardware. An OS is Sometimes referred to as a **platform**. The operating system determines which apps can run on the computer system.

Apps run on top of the operating system. You can run many apps at once in an operating system. Its job to determine which apps get precedence, and how to handle idle apps and background tasks.

Common operating systems include:

- Windows, macOS, and Linux for personal computers (laptops and desktops).
- Android and iOS for mobile devices.
- Linux and Windows for servers (non-personal computers).
- Linux for embedded devices.

Some apps are OS-exclusive. For example, iMessage and FaceTime only run on Apple operating systems like iOS for iPhone and macOS for MacBook.

All apps are OS-specific. For example, you can't install the macOS version of Google Chrome or Spotify on Windows. Each operating systems requires its own installer, libraries and supported programming languages.

At large or midsize software companies, development teams often focus on one operating system (or at least, a desktop team and a mobile team). In addition to the different user preferences, the underlying technologies of the operating systems differ. It takes time to become acquainted with the quirks of a particular platform. Each OS has its own libraries that determine certain elements of the app (for example, how to open and close it, what the menu bar or app settings window looks like). Additionally, each OS supports different programming languages. In fact, some OS **require** specific programming languages.

Real World App Example: Spotify

Spotify is an example of a multi-platform application. You can download it on Windows, macOS, Android, iOS, and more.

The same basic functionality is provided in every version: you log in, access your saved playlists, and play music. However, think about the differences between platforms.

- *How do you open and close the app?*
- *What is displayed on home screen when you open the app?*
- *How does the operating system indicate you are currently listening to music?*

Some of those differences are enforced by the operating system. For example, there is no button to close an app on mobile. Other differences are choices made by the application developer. A common **user experience** principle is to display fewer items on mobile than on desktop.

Spotify determined that mobile users (including users listening on a watch) prefer less choice in song selection if it means they can access their favorite music immediately. Mobile users also tend to accept lower-quality data (i.e., worse song quality) if it means that their songs will load more quickly and their music won't stop if their internet connection drops for a few moments.

Computer memory

Running programs are stored in a computer system's main memory component, RAM, which is measured in **gigabytes (GB)**. One gigabyte is **one billion bytes.** Modern smartphones generally range from 4GB in budget models to 16GB or more in high-end flagship phones. Laptops range from 8GB as a baseline up to to 128GB. With laptops, 16GB is widely considered the sweet spot for a smooth user experience.

When you open a program, it requests a large block of memory from the OS. The OS determines the priority of the request. If insufficient space is available, it could close background apps to accommodate the new request. When you close a program, the memory space is either totally deallocated or reduced to handle minimal background services. The OS uses **virtual memory mapping** to allocate dynamic and non-sequential memory blocks to the application. This is useful if the program needs more resources than it requested when it started.

The benefit of Java and other high-level languages is that you don't have to manage this memory process. It happens automatically.

As mentioned above, modern computer systems have multiple gigabytes of RAM. A laptop with 16GB of RAM has access to 16 billion bytes of memory.

Memory is stored as **bytes**, which consist of eight **bits**. Each bit is a binary digit, and it can have one of two possible values: a 0 or a 1.

With a single byte, you can represent one of **256** different values (28). A two-byte sequence can represent one of **65,536** different values (216). This value could be:

- A number, such as an integer from **0 to 65,535**.
- A character, using a character encoding standard like **UTF-16**, where each character (or code unit) is represented by two or more bytes. For instance, the two-byte sequence `0x0041` represents the character 'A'.
- An instruction or address. A two-byte sequence could specify the location of the next instruction to execute or the position of data in memory.

Each byte is stored at a **memory address location**, a unique identifier for a location in a computer's memory (RAM). It's a numerical value, typically represented in hexadecimal format. For example, `0x7FFC9F3E8B00`.

Software development

Computer software consists of code that uses logic to help a user perform at least one task.

Software could be as simple as a script that takes a snapshot of the weather conditions every hour and writes it to a file. Or it could be as complex as Uber, which coordinates 36 million trips per day and supports multiple activities (taxi, food deliver, courier services, and even car rentals).

However, in addition to being a computer program, Uber is also a company. A computer program doesn't need to be attached to a commercial enterprise. In fact, most computer programs have only one user: the person who programmed it.

A person who writes computer programs a **software developer** (or developer). Other titles include programmer or engineer.

The developer of a software application is often the only user because the software was built to solve a specific problem that the developer faced. Software should always solve a problem that you face. It doesn't matter how niche or insignificant the problem might seem to others, as long as you write software to solve a problem you face, learning to program will benefit you greatly.

Software development is also a progressive skill. Start with lower-complexity projects to build your confidence. You'll soon realize just how capable you are. Then you'll start looking for bigger problems to solve.

Software developers often share their programs for free on platforms like GitHub, enabling other developers to freely use and adjust the program to suit their needs. If you distribute your own software for free or improve the software of others, you are contributing to the open source community.

Of course, problem-solving skills are also desirable to employers, who often take special note of applicants who have built a diverse array of programs to solve problems of all sizes in personal and professional settings. # 1.2 Programming Languages Programming languages are used to write applications, including operating systems.

We can divide programming languages into two categories.

- **Low-level programming languages** interact closely with the hardware. Examples include Assembly, C, and C++.
- **High-level programming languages** interact with the operating system (rather than the hardware). Examples include Java, Python, C#, Swift, and JavaScript. These languages are more human-readable and provide many developer experience advantages over low-level languages.

Low-level languages

Low-level (or systems) programming languages interact with the physical components of the computer, like the CPU, memory, and I/O devices. They are "low" in the sense of being close to the hardware, not in the sense of "low complexity." Because of this, systems programmers must be aware of the underlying hardware architecture. Code written for one architecture (e.g., x86) may not be compatible with another (e.g., ARM) without significant modification.

As you could imagine, low-level languages are also the oldest programming languages.

- **Assembly languages** were invented in the **late 1940s and early 1950s**. When you write a program in an assembly language, it is translated into machine code by an **assembler**. Because each assembly language is specific to a particular procesor, the code written for one machine cannot run on another.
- The **C programming language** was devised in the early **1970s** by Dennis Ritchie at Bell Laboratories. C made significant improvements

over assembly by introducing higher-level datatypes and features that enhanced productivity and portability. C programs can compile to assembly code or directly to machine code by a C **compiler**.

- **C++** was invented by Bjarne Stroustrup at AT&T Bell Labs in **1979**. It originated as "C with Classes," designed to extend the C language with object-oriented programming features. The name was changed to C++ in 1983. Like C, it uses a compiler to generate machine code.

The C and C++ languages are human-readable but are full of opaque concepts and references. They often require specialized manuals to tell you the meaning of each value. Assembly languages are not human readable, but can include comments to help programmers keep track of the operations.

However, the major benefit to using low-level languages is that they are fast and efficient. If you need to execute millions of requests per second, highly optimized code can save you millions of dollars in computing resources.

Low-level code samples

Earlier we said that a computer's purpose is to help us achieve our goals. Let's imagine we're application developers at a bank. While we develop an app, we want to keep track of how many times we tried to run it. This helps with debugging, monitoring, and auditing. We can develop a small script that record events, errors, or just confirmation of its operations.

Assume we're free to use any programming language to develop this application logging script. Let's see how it might look in different low-level languages.

Let's start with the lowest level language.

Assembly

Specifically, Netwide Assembly (NASM) for Linux:

```
section .data
    filename db "application.log", 0
    message db "INFO: Operation completed successfully.", 0x0A
    msg_len equ $ - message

section .text
    global _start

_start:
    mov rax, 2
    lea rdi, [rel filename]
    mov rsi, 0o1002
    mov rdx, 0o644
    syscall
    mov rbx, rax

    mov rax, 1
    mov rdi, rbx
    lea rsi, [rel message]
    mov rdx, msg_len
    syscall

    mov rax, 3
    mov rdi, rbx
    syscall

    mov rax, 60
    xor rdi, rdi
    syscall
```

This code is incomprehensible without comments, but it does what we need: loads (or creates) a log file into memory, writes the message, and closes the file. It interacts directly with the operating system through system calls (`syscall`) and doesn't require any extra libraries.

> **Keep Exploring!** Paste the assembly code into your favorite LLM and ask it to explain each line.

C

Here's how the script might look written in C:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    FILE *file_ptr;
    time_t rawtime;
    struct tm *info;
    char timestamp[20];

    time(&rawtime);
    info = localtime(&rawtime);
    strftime(timestamp, sizeof(timestamp), "%Y-%m-%d %H:%M:%S", info);

    file_ptr = fopen("application.log", "a");
    if (file_ptr == NULL) {
        perror("Error opening log file");
        return EXIT_FAILURE;
    }
    fprintf(file_ptr, "[%s] INFO: Operation completed successfully.\n", timestamp);
    fclose(file_ptr);
    printf("Logged message to application.log\n");
    return EXIT_SUCCESS;
}
```

There are some obscure symbols and abbreviations, but it's at least human-readable. It does what the assembly code does, but it adds error handling in case the file can't be opened. It also prepends a timestamp to each message so we know exactly when the script was run.

Keep Exploring! Paste the C code into your favorite LLM and ask it to add comments to it.

One difference from Assembly is that C uses `#include` statements to bring portable libraries into your code. These library functions wrap the complex, OS-specific system calls. For example, when you call `fopen("application.log", "a")`, the `fopen` library function determines what underlying system calls are necessary for that specific operating system.

C++

Here's how the script might look written in C++:

```
#include <fstream>
#include <iostream>
#include <string>
#include <chrono>
#include <iomanip>

int main() {
    std::ofstream output_file("application.log", std::ios::app);
    if (!output_file.is_open()) {
        std::cerr << "Error: Could not open log file application.log\n";
        return 1;
    }

    auto now = std::chrono::system_clock::now();
    auto in_time_t = std::chrono::system_clock::to_time_t(now);

    output_file << "[" << std::put_time(std::localtime(&in_time_t), "%Y-%m-%d %H:%M:%S") <<
    output_file.close();
    std::cout << "Logged message to application.log\n";
    return 0;
}
```

The C++ code is similar to C, but achieves the same functionality in fewer lines of code. It also adds benefits in terms of safety and design.

C++, like Java, uses **objects** to represent data. You don't directly manipulate the raw data that comprises a file, for example. Instead, you interact with a **file object**. This object acts as a structured interface, providing a safer way to perform operations. This means there is less opportunity for programmers to accidentally misuse low-level pointers or memory, which can lead to crashes or security vulnerabilities in C or Assembly.

High-level languages

High-level programming languages interact with the operating system (rather than the hardware) either through an **interpreter** or a **compiled by a virtual machine**.

They are sometimes called **abstraction languages**, because they abstract away the lower-level details like memory management and system calls. They are extremely portable and provide many advantages for developer productivity.

Interpreted Languages

Interpreted languages, like **Python**, **Ruby**, and early versions of **JavaScript**, don't get fully compiled into machine code before execution. Instead, a program called an **interpreter** reads and executes the code line by line at runtime.

The interpreter itself is a program written in a lower-level language (like C or C++) that runs directly on the operating system. When you execute a Python script, for example, the Python interpreter starts up, reads your script, and translates each line into actions that the OS can understand (e.g., allocating memory, writing to a file, performing calculations).

Virtual Machine (VM) Based Languages

Languages like **Java** and **C#** are designed to run on a **virtual machine**. These languages are typically compiled into an **intermediate bytecode** rather than directly into machine code. The bytecode is then executed by a virtual machine. The VM acts as a runtime environment that translates the bytecode into native machine instructions for the underlying operating system.

The VM itself is an application that runs on the operating system. It requests resources from the OS, similar to how an interpreter does. The key difference is that the VM provides a consistent environment for the bytecode, abstracting away the specifics of the underlying OS.

JavaScript

JavaScript has had a unique execution journey that deserves separate mention.

JavaScript began primarily as a language for web browsers, where it was **interpreted**. The browser's JavaScript engine would parse and execute the code directly, line by line, allowing for dynamic, interactive web pages.

However, modern JavaScript environments, including current web browsers (like Chrome's V8 engine and Firefox's SpiderMonkey) and server-side runtimes like **Node.js**, now use a **hybrid execution model**. They typically start by parsing JavaScript into an intermediate form and running it in an **interpreter** for fast startup.

The engine then employs sophisticated **Just-In-Time (JIT) compilers** to translate frequently executed parts of the JavaScript code directly into highly optimized native machine code. This compilation happens during runtime, allowing modern JavaScript applications to achieve performance levels far exceeding what pure interpretation could offer.

High-level code samples

Java

We'll be seeing a lot of this! Here is the logger script in Java:

```java
import java.io.FileWriter;
import java.io.IOException;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class AppLogger {
    public static void main(String[] args) {
        String fileName = "application.log";
        String logMessage = "INFO: Operation completed successfully.";

        LocalDateTime now = LocalDateTime.now();
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
        String timestamp = now.format(formatter);

        try (FileWriter writer = new FileWriter(fileName, true)) {
            writer.write(String.format("[%s] %s%n", timestamp, logMessage));
            System.out.println("Logged message to " + fileName);
        } catch (IOException e) {
            System.err.println("An error occurred while writing to the log file: " + e.getMe
            e.printStackTrace();
        }
    }
}
```

Python

Here's the script in Python:

```python
import datetime

file_name = "application.log"
log_message = "INFO: Operation completed successfully."

current_time = datetime.datetime.now()
timestamp = current_time.strftime("%Y-%m-%d %H:%M:%S")

try:
    with open(file_name, "a") as file:
        file.write(f"[{timestamp}] {log_message}\n")
    print(f"Logged message to {file_name}")
except IOError as e:
    print(f"Error writing to log file {file_name}: {e}")
```

JavaScript

Here's the script in JavaScript:

```javascript
const fs = require('fs');

const fileName = 'application.log';
const logMessage = 'INFO: Operation completed successfully.';

const now = new Date();
const timestamp = now.getFullYear() + '-' +
                  String(now.getMonth() + 1).padStart(2, '0') + '-' +
                  String(now.getDate()).padStart(2, '0') + ' ' +
                  String(now.getHours()).padStart(2, '0') + ':' +
                  String(now.getMinutes()).padStart(2, '0') + ':' +
                  String(now.getSeconds()).padStart(2, '0');

const logEntry = `[${timestamp}] ${logMessage}\n`;

fs.appendFile(fileName, logEntry, (err) => {
    if (err) {
        console.error('Error writing to log file:', err);
    } else {
        console.log(`Logged message to ${fileName}`);
    }
});
```

1.3 Programming Environments

Programming Paradigms

A programming paradigm is a fundamental style of structuring a program. Three prominent paradigms are procedural programming, object-oriented programming (OOP), and functional programming.

- **Procedural programming** is a paradigm based on the concept of the **procedure call**. In this approach, a program is structured as a series of instructions that tell the computer what to do step-by-step. Data is passed between functions or procedures, and the program's control flow is top-down. The main program is responsible for calling these procedures to perform tasks.
  - Languages popular in this paradigm include **C, Fortran, and Pascal**. Scripting languages such as **Python, JavaScript, and PowerShell** can be written procedurally.
- **Object-Oriented Programming** (OOP) is a paradigm that structures a program around **objects**. An object is a self-contained unit that encapsulates both data (its **state**) and the procedures (its **behavior** or **methods**) that operate on that data. This is a significant departure from procedural programming, where data and procedures are separate. The core idea of OOP is to model real-world entities. For example, a program for a car dealership might have a `Car` object with data like `color`, `make`, and `model`, and methods like `startEngine()` or `accelerate()`.
  - **Java** is an example of an object-oriented language. It is often used to build complex, scalable, and maintainable applications because of its organized, modular nature. Other popular OOP languages include **Python, C++, and C#**.
- **Functional programming** is a paradigm where programs are constructed by applying and composing **functions**. Its key characteristics include treating computation as the evaluation of mathematical functions and avoiding state and mutable data. This means that a function's output depends only on its inputs, not on any external or hidden state, which makes it easier to reason about and test code.
  - Functional programming is used in many modern languages, and **Java** has incorporated some functional features, such as lambda expressions, since Java 8. Popular languages in this paradigm include **Haskell, Lisp, and Scala**.

Integrated Development Environments (IDEs)

While programs can be written in simple text editors like Notepad, they are often developed using **Integrated Development Environments (IDEs)** , which offer numerous advantages beyond basic text editing.

An IDE is a software application that provides comprehensive facilities to computer programmers for software development. It typically consists of a source code editor, build automation tools, and a debugger.

IDEs provide features like **autocomplete**, **syntax highlighting**, and the ability to **run, test, and debug programs** from a single interface. They also integrate with **version control systems** like Git, making it easier to manage code changes, and include **refactoring tools** to safely restructure code.

IDEs can be either general-purpose, supporting many languages, or specialized, optimized for a specific language or a small set of related languages.

Here are some of the most popular IDEs:

- **Visual Studio Code (VS Code)**: A highly popular, free, and open-source code editor that has become a de facto standard for many developers due to its lightweight nature and extensive marketplace of extensions. It supports a huge number of languages, including JavaScript, Python, C++, and Java.
- **IntelliJ IDEA**: Considered one of the best IDEs for **Java** development, IntelliJ IDEA is known for its smart code completion, on-the-fly error highlighting, and powerful tools for refactoring and debugging.
- **PyCharm**: An IDE from the same company as IntelliJ IDEA, designed specifically for **Python**. It's widely used for web development, data science, and machine learning with Python.
- **Xcode**: **Apple's** proprietary IDE, used exclusively for developing applications for iOS and macOS using languages like Swift and Objective-C. It includes a built-in GUI builder and a device simulator.
- **Android Studio**: The official IDE for **Android** app development. It is tailored for building high-quality Android apps with tools like an emulator and mobile-specific libraries.
- **WebStorm**: A specialized IDE for **web development** with JavaScript, TypeScript, and related technologies.
- **Visual Studio**: A more robust, full-featured IDE from Microsoft, primarily used for large-scale, enterprise-level application development. It provides comprehensive tools for debugging, testing, and project management and supports over 30 languages.

Command-Line Interface (CLI)

You may think of software as having a graphical user interface, or **GUI**, which uses windows, icons, and menus. However, a significant portion of software, particularly in development, relies on **command-line interfaces** (CLIs) and **text-based user interfaces** (TUIs).

In this course, we will initially focus on CLI apps for several reasons:

- **Simplicity and Focus:** TUIs are straightforward to develop because they eliminate the complexities of graphical elements. By focusing solely on

data input and output through text, we can concentrate on core programming concepts like **variables**, **decision structures**, and **loops** without the distraction of building a complex visual interface.
- **Foundation for Servers:** TUI programs often serve as the foundation for **server-side applications**. A server program's primary role is to process data and interact with other programs (like a website's GUI or a mobile app) rather than with a human user directly through a graphical interface. For example, a web server processes requests and sends back data without ever needing a window to display it.
- **Debugging:** The command line is an essential tool for developers. By learning to interact with our programs through a terminal, we become comfortable with a powerful environment for **testing and debugging** code. This skill is transferable to almost any programming context.

Version Control and Collaboration

Git and GitHub are two essential tools in modern software development. **Git** is a **version control system** (VCS), while **GitHub** is a web-based service that hosts Git repositories. Think of Git as an engine that tracks changes, and GitHub as a social platform and cloud storage service built around that engine.

Git

Git is a free, open-source, distributed version control system that developers install locally on their computers. Its primary purpose is to track and manage changes to files over time, especially source code. Git allows developers to:

- **Create a complete history** of their code, with "snapshots" called **commits**.
- **Revert to previous versions** if something goes wrong.
- **Work offline** because each developer has a full, local copy of the entire project's history.
- **Create branches** to work on new features or bug fixes in isolation without affecting the main codebase.
- **Merge changes** from different branches back into the main project.

The core of Git operates through a command-line interface, where developers use commands like `git clone`, `git commit`, `git push`, and `git pull`.

GitHub

GitHub is a company that provides a cloud-based hosting service for Git repositories. It was founded in 2008 and acquired by Microsoft in 2018. GitHub takes the core functionality of Git and adds a rich ecosystem of tools for collaboration and project management.

Key features GitHub provides include:

- A **centralized, shared repository** in the cloud that serves as the "source of truth" for a project.
- A **Graphical User Interface (GUI)** via a web-based interface that makes Git easier to use.
- **Collaboration tools** like **pull requests**, **code reviews**, **issue tracking**, and **project management boards**.
- **User management and security features**, such as two-factor authentication.

While you can use Git without GitHub, using a service like GitHub makes it much easier to collaborate on and share code with others, manage large projects, and back up your work. It is the world's largest host for source code and is widely used for both open-source and commercial projects. # 1.4 Why Java? After learning the foundational concepts of computer systems, programming languages, and software development, we move on to discuss the topic we're here to learn: Java.

The last section mentioned that Java is a compiled, high-level programming language that tries to balance the power and expressiveness of low-level languages with the portability and developer experience improvements of high-level languages.

This section aims to describe Java's history, philosophy, and role in software development so that you gain a sense of why Java is and continues to be a solid choice for a programming language.

Java Then

The story of Java begins in late 1990 with a small, isolated group of Sun Microsystems engineers led by **James Gosling**. Their secretive mission, initially called the **Green Project**, was chartered by Sun to anticipate the next wave in computing: combining digitally-controlled consumer devices (like VCRs, TVs, cable boxes, and other appliances) with computers.

The team, comprising only 13 people, deliberately cut off regular communications with Sun and worked around the clock for 18 months in an anonymous office on Sand Hill Road in Silicon Valley. Their goal was to build a platform-independent, distributed system for embedded electronics, like VCRs, elevators, locomotives, and interactive TV set-top boxes.

Gosling initially tried to adapt C and C++ to build the systems, but found fundamental issues—like manual memory management and platform-specific compilation—too deeply embedded in the languages. He concluded that a new language, designed from the ground up to be portable, secure, and reliable, was the only way to achieve the Green Project's goals. This new language, initially called **Oak**, was designed with features like automatic memory management (garbage collection) and a virtual machine that allowed code to run on any device without recompilation.

A key insight for this new software came to Gosling while attending a Doobie Brothers concert. He described the experience to a reporter in 1995, who wrote:

> As he sat slouched in front-row seats letting the music wash over him, Gosling looked up at wiring and speakers and semi-robotic lights that seemed to dance to the music. "I kept seeing imaginary packets flowing down the wires making everything happen,'' he recalls."I'd been thinking a lot about making behavior flow through networks in a fairly narrow way. During the concert, I broke through on a pile of technical issues. **I got a deep feeling about how far this could all go: weaving networks and computers into even fine details of everyday life.**''

In the summer of 1992, the Green Team unveiled their working demo: a hand-held home-entertainment device controller called **Star7**, featuring an animated touchscreen interface where the Java's now-famous mascot, **Duke**, made his debut. This device ran on the processor-independent Oak, embodying Gosling's principle of **"write once, run anywhere."**

> **not-so-ancient history:** Watch the Star7 Demo by James Gosling on Internet Archive (9 minutes). Java was invented for this device in 1992.

Despite this innovation, the team struggled to find a market for their technology in the nascent interactive TV industry, losing bids to companies like Time-Warner. Facing a dead end, an epiphany occurred to the team in 1994: **"Why not the Internet?"**.

The emerging World Wide Web was transforming into the interactive network they had initially envisioned for cable TV. The team quickly developed **WebRunner** (later renamed **HotJava**), a Java-based browser that brought "animated, moving objects and dynamic executable content inside a Web browser" for the first time.

James Gosling and (Sun director) James Gage captivated an influential audience with a legendary demonstration of WebRunner at the **TED Conference in early 1995** by dragging a rotating 3D molecule through the middle of text on a web page. The impact of the event was described in an article about the early years of Java:

> Gosling moved the mouse over an illustration of a 3D molecule in the middle of the text. The 3D molecule rotated with the mouse movement. Back and forth, up and around. "The entire audience went 'Aaaaaaah!'" says Gosling. "Their view of reality had completely changed because it MOVED." Now everyone was paying close attention. Suddenly, everyone in the room was rethinking the potential of the Internet. [2]

This public debut led to an overwhelming demand for Java's code, with the team jumping for joy with each download. An unexpected front-page story in the San Jose Mercury News and **Netscape's announcement on May 23, 1995, to incorporate Java into Navigator** solidly positioned Java as The Next Big Thing for the Internet.

While applets initially aimed for client-side Web dominance, Java's unexpected and massive success came on the server side due to the JVM's strengths in building reliable, concurrent, and dynamically linked software.

Java Now

Java and its **Java Virtual Machine (JVM)** have proven themselves as foundational technologies in enterprise software development. Java's robustness, powerful features, and comprehensive standard library make it a reliable choice for building large, scalable, and secure applications. The extensive ecosystem of frameworks (like Spring) and tools further cements its popularity for mission-critical systems and microservices.

Java's core principle is **write once, run anywhere**. This is achieved by compiling Java source code into an intermediate format called **bytecode**. This bytecode is not tied to a specific machine architecture or operating system. Instead, it's the job of the JVM to interpret and execute this bytecode. Since a JVM implementation exists for virtually every platform—from servers and desktops to mobile devices—the same compiled bytecode can run on any system that has a compatible JVM, ensuring a consistent execution environment regardless of the underlying hardware.

The JVM's design makes it a highly flexible and language-agnostic platform. It doesn't exclusively run Java; it runs **bytecode**. This means that any programming language can run on the JVM as long as it has a compiler that can translate its source code into the correct bytecode format. This flexibility has given rise to a rich ecosystem of languages like **Kotlin**, **Scala**, and **Groovy**, which all leverage the JVM's performance optimizations, robust garbage collection, and massive library ecosystem while offering different programming paradigms and syntax.

Here are a few organizations that use Java:

Company	Industry	How They Use Java
Google	Technology	Android app development, internal tools, backend services
Amazon	E-commerce & Cloud	Backend systems, microservices on AWS
Netflix	Streaming Services	Scalable backend infrastructure using Java-based microservices
LinkedIn	Social Media/Professional Networking	Backend systems, data processing

Company	Industry	How They Use Java
Uber	Mobility/Transport	Real-time data handling and backend APIs
Airbnb	Hospitality	Backend architecture and data services
eBay	E-commerce	Enterprise backend systems
Spotify	Music Streaming	Backend services, data analytics pipelines
Salesforce	SaaS / Cloud CRM	Cloud platform development and integrations
Instagram	Social Media	Java used in backend services and API handling
NASA	Aerospace / Research	Scientific computing tools and simulations
TCS	IT Services	Client software, web services, and enterprise apps
Infosys	IT & Consulting	Java for full-stack and enterprise-level solutions
Oracle	Software & Databases	Core development of Java (owns the Java platform)

In particular, the Android operating system by **Google** is built on a modified version of the Java Virtual Machine called the Android Runtime (ART), which was created to be more efficient for mobile devices. There are over three billion active Android devices worldwide.

Additionally, the backend services at **Netflix** are **predominantly all Java**, even though front-end UIs are developed in languages best suited for specific devices.

> **not-so-ancient history:** If you're interested to learn more about real-world uses of Java, watch this video: How Netflix Uses Java in 2025.

Sources:

1. David Bank, "The Java Saga", *WIRED* (*Dec. 1, 1995*)
2. Jon Byous, "Java Technology: The Early Years", Sun Microsystems (April 2003)

Java Philosophy and Architecture

I won't say that Java is the perfect programming language, but I do appreciate how it combines good ideas from both the low-level and the high-level languages to give developers a powerful and intuitive experience.

But Java isn't just a programming language, it's also the environment where Java applications run. The Java platform includes the Java Virtual Machine (JVM), the Java Application Programming Interface (API), and the Java Development Kit (JDK).

Learning the philosophy behind the Java programming language and the architecture of the Java platform will empower you to effectively choose, use, and even create the tools and libraries you need to build Java applications.

What makes the Java language different?

Here are five factors that differentiate Java from other programming languages:

- **Object-Oriented Programming (OOP)**: Java is fundamentally an OOP language. This means it structures programs around objects rather than just functions and logic. This approach promotes modularity, reusability of code, and easier management of complex applications. Languages that are not primarily object-based are called **procedural** or **scripting languages**. Examples of procedural programming languages include Python, JavaScript, and PowerShell.
- **Platform Independence ("Write Once, Run Anywhere")**: This is arguably Java's most significant advantage. Java achieves platform independence at both the source and binary levels. You write Java code once, compile it into bytecode, and this bytecode can then run on any system that has a Java Virtual Machine (JVM). This contrasts with languages that compile directly to machine code such as C or C++, which must be recompiled for each different operating system or hardware architecture.
- **Memory Management**: Java has strong memory management capabilities, including automatic garbage collection, which helps prevent memory leaks and related errors. Developers don't typically need to manually allocate and deallocate memory like with C and C++, which simplifies development and reduces common programming errors.
- **Robustness, Security, and Ease of Use**: Java was designed with robustness in mind, aiming to prevent user programs from crashing the host machine or interfering with other operations like in C or C++. It includes features for secure execution, particularly for remote code. Java was also designed to be relatively easy to learn, write, compile, and debug compared to some other programming languages, thanks to its simpler syntax.
- **Strongly Typed**: Java is a strongly typed language, meaning variables must have a declared type, and type compatibility is checked at compile time. This helps catch errors early in the development process. Languages that do not require variables to have declared types are called **dynamically typed** or **weakly typed languages**. Examples include JavaScript and Python.

What is the Java platform?

The Java platform is called the **Java Runtime Environment (JRE)**. The JRE is the bundle that provides the **Java Virtual Machine (JVM)** and the **Java API** libraries. It's everything an end-user needs to run a Java application.

In order to write, compile, and run Java code, developers install the **Java Development Kit (JDK)**. The JDK is a comprehensive software development kit that includes everything in the JRE, plus additional tools needed for developing, compiling, and debugging Java applications.

Some of the most important JDK tools include:

- `javac`: The **Java compiler**, which translates your human-readable Java code (`.java` files) into Java bytecode (`.class` files).
- `java`: The program launcher, which starts the JVM to run your compiled bytecode.
- `jdb`: The **Java debugger**, a command-line tool for finding and fixing errors in your code.
- `jar`: The **archiver**, which packages multiple `.class` files and other resources into a single, distributable `.jar` file.
- `javadoc`: A tool that generates HTML documentation from comments in your source code.
- `jshell`: An interactive tool for quickly experimenting with Java code.

Who makes the JDK?

Historically, Oracle (and previously Sun Microsystems) was the primary provider of the JDK. However, due to licensing changes and the open-sourcing of Java, multiple vendors now provide their own builds of the JDK.

Open source means that the source code is publicly accessible, allowing anyone to view, modify, and distribute the code as they see fit. This fosters decentralized and collaborative development, peer review, and often leads to lower costs, more flexibility, and greater longevity compared to proprietary software.

Sun Microsystems open-sourced the JDK in 2006 to increase Java adoption and community involvement, and ultimately drive more sales to their hardware. After acquiring Sun Microsystems in 2010, **Oracle** continued the open-source effort, but also released a commercial version of the JDK.

The source code for the JDK is here: https://github.com/openjdk/jdk. You are free to download it, alter it in any way you see fit, and redistribute it.

However, because this could lead to incompatibilities, it's best to stick to a reputable JDK provider. Some prominent JDK providers include:

- **OpenJDK (Oracle's build)**: Oracle itself provides free production-ready OpenJDK binaries.

- **Eclipse Temurin (formerly AdoptOpenJDK)**: A widely used, free, and verified OpenJDK distribution developed under the Eclipse Foundation.
- **Amazon Corretto**: A no-cost, long-term supported distribution from Amazon.
- **Red Hat OpenJDK**: Red Hat is an active contributor to the OpenJDK project and provides free distributions.
- **IBM Semeru Runtime (with OpenJ9 JVM)**: IBM provides builds based on the OpenJDK class libraries and their own OpenJ9 JVM, which is known for low memory usage.
- **Microsoft Build of OpenJDK**: Microsoft also released its own free and long-term supported OpenJDK distribution.
- **SapMachine**: An SAP-supported version of OpenJDK.
- **BellSoft Liberica JDK**: A 100% open-source Java implementation built from OpenJDK.

What is the Java Virtual Machine (JVM)?

The Java Virtual Machine (JVM) provides a runtime environment for Java programs and other languages that compile to Java bytecode. Think of it as an small computer within your computer whose only job is to run Java programs.

Here's how the JVM works:

1. **Loads Bytecode**: The JVM loads Java bytecode (which is the compiled form of your Java source code) into its memory.
2. **Verifies Bytecode**: It performs bytecode verification to ensure the code is safe and adheres to Java language specifications.
3. **Manages Memory**: The JVM manages memory allocation for objects and uses automatic garbage collection to reclaim memory from objects that are no longer needed.
4. **Executes Bytecode**: It executes bytecode instructions. It may use an interpreter to execute instructions one by one, and often employs a Just-In-Time (JIT) compiler to dynamically translate frequently executed bytecode into optimized native machine code for better performance.
5. **Platform Independence**: By abstracting the underlying hardware and operating system, the JVM provides a consistent execution environment, enabling Java's "write once, run anywhere" capability.

The default JVM included with most Java distributions is called **HotSpot**. There are other types of JVM, such as GraalVM and the Android Runtime, that focus on supporting different types of systems. HotSpot is a **Just-In-Time (JIT) compiler**, and its primary philosophy is **adaptive optimization**.

Adaptive optimization in HotSpot works like this:

- When you run your Java program, the JVM starts by running code slowly using an interpreter.

- It then monitors the running program to find "hot spots"—the parts of the code that are executed most frequently.
- It compiles only these hot spots into highly optimized native machine code.
- This compilation happens *while the program is running.* This allows the JIT compiler to make powerful, dynamic optimizations that are impossible for a traditional, static compiler to make.

This warm-up period is a trade-off: the program starts a bit slower, but once the JIT compiler has done its work, the long-running application can achieve exceptional peak performance.

What is the Java API?

The **Java API (Application Programming Interface)** is a vast collection of pre-written packages, classes, and interfaces that provide a standardized way for developers to interact with various functionalities.

It's essentially a rich library of pre-built code that saves developers significant time and effort, allowing them to focus on the unique logic of their applications rather than building everything from scratch.

The Java API offers functionalities for a wide range of tasks, from basic operations like mathematical functions to complex features for database connectivity, web development, networking, security, and more.

The source code for the base Java API classes is here: https://github.com/openjdk/jdk/tree/master/src/java.base/share/classes/java

> **Coming attractions**: When you use `System.out.println()` or `Scanner` in your code, you are utilizing classes and methods provided by the Java API. We'll see these in the next chapter.