A Little History of Programming

8 Versions of Hello to {Name}

Brendan Shea, PhD

Programming and Problem Solving • Rochester Community and Technical College

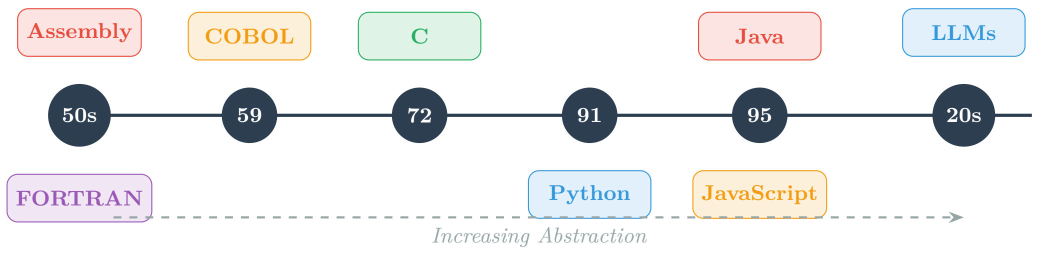
Introduction

Every programmer remembers their first program. For millions of us, that program displayed a simple greeting: "Hello, World!" But have you ever wondered how that simple act of saying hello has changed over nearly 80 years of computing history? In this case study, we'll explore the evolution of programming languages by examining how eight different languages—spanning from the 1950s to today—accomplish the seemingly simple task of greeting a user by name.

This journey will take us from the raw metal of **machine code** through the revolutionary abstraction of **high-level languages**, and finally to the frontier of **natural language programming** with large language models. Along the way, you'll discover how each generation of programming tools reflects the technology, priorities, and philosophy of its era.

The Challenge

Write a program that asks the user for their name and then displays: "Hello, [Name]!" This simple task requires: getting input from a user, storing data in memory, combining text (string concatenation), and displaying output. How we accomplish these tasks reveals everything about a programming language's design philosophy.



1 The Pioneers: Assembly Language (1950s)

In the earliest days of computing, programmers communicated with machines using **assembly language**—a thin layer of human-readable mnemonics sitting directly atop the CPU's binary instructions. Each line of assembly corresponds almost one-to-one with a machine instruction. There is no hiding the computer's architecture here; you must understand registers, memory addresses, and the specific instruction set of your processor.

x86-64 Assembly (Linux)

```
section .data                       ; Data we define ahead of time
    prompt db "Enter your name: ", 0
    hello db "Hello, ", 0
section .bss                        ; Reserve space for user input
    name resb 64                    ; 64 bytes for the name
section .text
    global _start
_start:
    ; STEP 1: Print the prompt to screen
    mov rax, 1             ; System call number for "write"
    mov rdi, 1             ; File descriptor 1 = standard output
    mov rsi, prompt        ; Address of our prompt string
    mov rdx, 17            ; Number of bytes to write
    syscall                ; Ask the operating system to do it

    ; STEP 2: Read the user's input
    mov rax, 0             ; System call number for "read"
    mov rdi, 0             ; File descriptor 0 = standard input
    mov rsi, name          ; Where to store the input
    mov rdx, 64            ; Maximum bytes to read
    syscall

    ; STEP 3: Print "Hello, " then the name...
    ; (Additional code continues similarly)
```

Notice how every operation requires explicit instructions: we must tell the CPU exactly which **register** to use, specify memory addresses, count bytes manually, and invoke operating system services through numbered "syscalls." The command `mov rax, 1` doesn't mean "print"—it loads the number 1 into a register, which *happens* to be the code for the write operation. This granular control yields maximum performance but minimal productivity.

2 Business Computing: COBOL (1959)

While scientists worked with numbers, the business world needed to process text, records, and reports. **COBOL** (Common Business-Oriented Language) was designed by a committee including the legendary Grace Hopper, who believed programs should be readable like English prose.

COBOL

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO-NAME.

DATA DIVISION.
```

```
WORKING-STORAGE SECTION.
01 USER-NAME PIC X(30).          *> Reserve 30 characters for name

PROCEDURE DIVISION.
    DISPLAY "Enter your name: ".  *> Show prompt on screen
    ACCEPT USER-NAME.             *> Wait for and store input
    DISPLAY "Hello, " USER-NAME "!".  *> Combine and display
    STOP RUN.
```

The contrast with assembly is striking. COBOL reads almost like a business memo: `DISPLAY` shows text, `ACCEPT` gets input, and `PIC X(30)` declares a "picture" of 30 characters. The verbose DIVISIONS and SECTIONS may seem excessive, but they made programs easier to read and audit—crucial for financial applications. Remarkably, COBOL remains in use today; billions of lines still process banking transactions and government records.

3 Scientific Roots: FORTRAN (1957)

FORTRAN (Formula Translation) holds the distinction of being the first widely adopted high-level language. Created at IBM by John Backus and his team, FORTRAN allowed scientists and engineers to write mathematical formulas in near-natural notation, with the **compiler** translating these into efficient machine code.

Fortran 90

```
program hello_name
    character(len=50) :: name     ! Declare a 50-character variable

    print *, "Enter your name: " ! Print prompt (* means default format)
    read *, name                 ! Read input into 'name' variable
    print *, "Hello, ", trim(name), "!"   ! trim() removes extra spaces
end program hello_name
```

FORTRAN demonstrated a revolutionary concept: you could write code at a higher level of **abstraction** and trust the compiler to produce efficient results. The `print` and `read` statements handle all the complex details of I/O that assembly required us to manage manually. The `trim()` function shows another advantage: built-in utilities that would require dozens of assembly instructions.

4 Systems Programming: C (1972)

Dennis Ritchie created **C** at Bell Labs to rewrite the Unix operating system. C occupies a unique middle ground: it provides high-level conveniences like functions and structured data types while still allowing direct memory manipulation through **pointers**.

C

```
#include <stdio.h>     // Include standard input/output library

int main() {
    char name[50];     // Create array of 50 characters for the name
    printf("Enter your name: ");  // Print prompt (no newline)
    scanf("%49s", name);    // Read up to 49 chars into 'name'
                            // %s = string format, 49 = safety limit
    printf("Hello, %s!\n", name);  // %s gets replaced by 'name'
    return 0;               // Return 0 = program succeeded
}
```

The `char name[50]` declaration reveals C's philosophy: you explicitly manage memory by declaring arrays of specific sizes. The `%s` is a "format specifier" that tells `printf` where to insert the string. C's influence is immense—your operating system, web browser, and smartphone are largely written in C or its descendants. The syntax with curly braces, semicolons, and `main()` became the template for Java, JavaScript, and many others.

5 Object-Oriented Revolution: Java (1995)

James Gosling and his team at Sun Microsystems designed **Java** with a bold promise: "Write Once, Run Anywhere." Java programs run on a **virtual machine** (JVM) rather than directly on hardware, enabling the same code to execute on any platform.

Java

```java
import java.util.Scanner;  // Import the Scanner tool for input

public class HelloName {
    public static void main(String[] args) {
        // Create a Scanner object to read from keyboard
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter your name: ");  // print = no newline
        String name = scanner.nextLine();  // Read entire line of input

        // Concatenate strings with + operator
        System.out.println("Hello, " + name + "!");  // println = newline
        scanner.close();  // Clean up the Scanner resource
    }
}
```

Java introduced millions of developers to **object-oriented programming** (OOP). Notice how we create a `Scanner` *object* and call its `nextLine()` method. Everything in Java lives inside a *class* (here, `HelloName`). The verbose declaration `public static void main(String[] args)` is Java's required entry point—every application starts here. This explicitness enables large teams to collaborate on massive codebases with confidence.

6 The Web's Language: JavaScript (1995)

Brendan Eich famously created **JavaScript** in just ten days at Netscape. Despite its name (a marketing decision), JavaScript is completely different from Java. It brought programming directly into web browsers.

JavaScript (Browser)

```javascript
// In a browser, these built-in functions handle everything!
const name = prompt("Enter your name:");  // Pop-up input box
alert(`Hello, ${name}!`);  // Pop-up message box
// The backticks and ${} let us embed variables in strings
```

JavaScript (Node.js - Server)

```javascript
const readline = require('readline');  // Import I/O module
const rl = readline.createInterface({
    input: process.stdin,     // Read from keyboard
    output: process.stdout    // Write to screen
});

// question() displays prompt, then runs callback with answer
rl.question("Enter your name: ", (name) => {
    console.log(`Hello, ${name}!`);  // Template literal syntax
    rl.close();
});
```

JavaScript exemplifies **dynamic typing**—variables like `name` don't have declared types; they can hold any kind of data. The browser version accomplishes our task in just two lines! The template literal syntax (`` `Hello, ${name}!` ``) with backticks is modern JavaScript's elegant way to embed variables in strings.

7 Readability First: Python (1991)

Guido van Rossum created **Python** with an explicit philosophy: code readability counts. Python's clean syntax eliminates much of the ceremony found in other languages.

Python

```python
# Get input - input() displays prompt and returns what user types
name = input("Enter your name: ")

# Print with f-string: f"..." lets us embed {variables} directly
print(f"Hello, {name}!")
```

Two lines. No imports for basic I/O, no type declarations, no semicolons, no braces. The `f` before the string creates a "formatted string literal" (f-string)—Python's elegant solution for combining text and variables. Python's **interpreted** nature means you can run code immediately without a separate compilation step. This immediacy has made Python the dominant language in data science, AI, and education.

8 The New Frontier: Natural Language and LLMs (2020s)

We now stand at another inflection point. **Large Language Models** (LLMs) like GPT and Claude can generate working code from plain English descriptions:

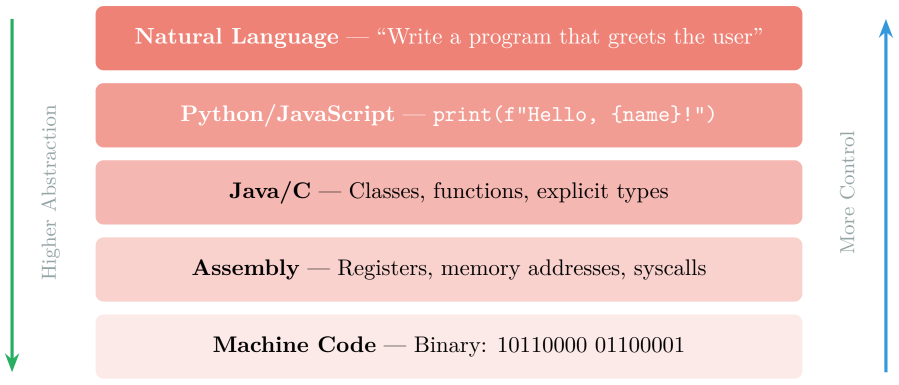
Prompting an LLM

```
Write a Python program that asks for the user's name
and then greets them by saying "Hello" followed by
their name and an exclamation point.
```

This represents **natural language programming**—describing what you want rather than how to do it. The LLM translates your intent into working code. While this technology is new and imperfect (it can produce buggy or insecure code), it suggests a future where the barrier between human intention and computer execution continues to shrink.

9 Patterns Across Time

The Abstraction Pyramid


Natural Language — "Write a program that greets the user"
Python/JavaScript — print(f"Hello, {name}!")
Java/C — Classes, functions, explicit types
Assembly — Registers, memory addresses, syscalls
Machine Code — Binary: 10110000 01100001
Higher Abstraction
More Control

Looking at these eight approaches, several patterns emerge. First, there is a clear trend toward **higher abstraction**—each generation hides more complexity, allowing programmers to think about *what* they want rather than *how* the machine accomplishes it. Second, we see increasing emphasis on **readability**. From COBOL's English-like syntax to Python's clean design, language creators recognize that code is read far more often than it is written. Third, the **target audience** has expanded dramatically. Early languages served specialists with deep hardware knowledge; modern languages welcome beginners. Finally, the very definition of "programming" is shifting—from manipulating registers to describing intentions in natural language.

Yet older languages persist. COBOL processes trillions in transactions. FORTRAN dominates scientific supercomputing. C underlies virtually every operating system. Each language found its niche and, once embedded in critical systems, proved remarkably durable. The best language is always the one that fits the problem, the team, and the context.

Discussion Questions

Discussion Questions

1. **Trade-offs in Abstraction:** Assembly language gives programmers complete control but requires extensive expertise. Python hides many details but limits control over performance. When might the trade-off favor lower-level languages? Consider embedded systems, game engines, operating systems, web applications, and data analysis scripts.

2. **The COBOL Paradox:** COBOL is often mocked as outdated, yet it processes trillions of dollars in transactions daily and there's a shortage of COBOL programmers. Why do organizations continue using 60-year-old technology? What does this tell us about the relationship between programming languages and the systems that depend on them?

3. **Natural Language Programming:** If LLMs can generate code from English descriptions, what skills will programmers need in the future? Will "prompt engineering" replace traditional coding, or will understanding programming fundamentals remain essential for debugging, security, and system design?

4. **Design Philosophy:** Java emphasizes explicit type declarations and verbose syntax; Python emphasizes brevity and readability. Both have passionate advocates. How might team size, project lifespan, or application domain (finance vs. startups vs. scientific research) affect which philosophy is preferable?

5. **Historical Contingency:** JavaScript was created in ten days and now runs on billions of devices. Some languages with elegant designs (like Lisp or Haskell) remain niche. How much of a programming language's success comes from technical merit versus historical accident, corporate backing, or simply being available at the right moment?

Key Terms

Glossary of Key Terms

Abstraction	Hiding complex implementation details behind simpler interfaces, allowing programmers to work at higher conceptual levels without managing every detail.
Assembly Language	A low-level language using mnemonic codes (like MOV, ADD) that correspond directly to a processor's machine instructions.
COBOL	Common Business-Oriented Language (1959); designed for business data processing with English-like, self-documenting syntax.
Compiler	A program that translates human-readable source code into machine code before the program runs, enabling optimization and error checking.
Dynamic Typing	A language feature where variable types are determined at runtime rather than declared in advance, offering flexibility but fewer compile-time checks.
FORTRAN	Formula Translation (1957); the first widely-used high-level language, designed for scientific and mathematical computing.
High-Level Language	A programming language that abstracts away hardware details, allowing portable, human-readable code.
Interpreted	Executing code line-by-line through an interpreter rather than compiling it to machine code first, enabling immediate feedback.

Large Language Model	An AI system trained on vast text data that can generate human-like text and code from natural language prompts.
Machine Code	Binary instructions (1s and 0s) that a CPU executes directly—the lowest level of software.
Natural Language Programming	Using human language to instruct computers, often via AI systems that translate intent into executable code.
Object-Oriented Programming	A paradigm organizing code into "objects" containing both data and the methods that operate on that data.
Pointer	A variable storing a memory address, enabling direct memory manipulation—powerful but error-prone.
Register	A small, ultra-fast storage location inside a CPU, used for data being actively processed.
Virtual Machine	A software layer executing code independently of underlying hardware, enabling platform-independent programs (e.g., Java's JVM).