

Understanding Programming Languages: Compiled, Bytecode, and Interpreted Languages



Prayag Sangode

Follow

6 min read · Oct 29, 2024



1



1



HOW PROGRAMMING LANGUAGES WORK: COMPILED, BYTECODE, AND INTERPRETED BASICS



BY PRAYAG SANGODE
PROGRAMING BLOG SERIES

What Is Source Code?

Source code is the human-readable code that developers write. It contains instructions in a specific programming language that tells a computer what

tasks to perform. Since computers only understand machine code (binary), source code needs to be converted or processed into this machine-readable format.

Compiled Languages

In compiled languages, the source code is converted directly into **machine code** by a **compiler**. This machine code can then be executed directly by the CPU, making compiled languages generally faster than interpreted or bytecode languages.

Compiled Languages workflow:

```
[Source Code] - (Compiler) → [Machine Code] → [Executed Directly by CPU]
```

How It Works

Write: Developers write source code in a compiled language (e.g., C, C++, Go).

Compile: The code is processed by a compiler, which converts it into machine code.

Execute: This machine code is run directly by the CPU, resulting in fast execution.

Example Use Cases

C: For high-performance tasks like operating systems.

C++: Used in software requiring speed, like video games and financial applications.

Go: Known for building cloud applications and microservices efficiently.

Bytecode Languages

In bytecode languages, the source code is first compiled into **bytecode** (intermediate code) that is then run by a **virtual machine (VM)** like the **Java Virtual Machine (JVM)**. The VM interprets the bytecode or uses **Just-In-Time (JIT)** compilation to translate it into machine code just before execution. This method **balances speed and portability**.

Bytecode Languages work flow:

```
[Source Code] - (Compiler) → [Bytecode] → [Virtual Machine] → [Machine Code] →
```

How It Works

Write: Code is written in a bytecode language (e.g., Java, C#).

Compile to Bytecode: A compiler converts the code to bytecode.

Execute with a VM: A VM interprets or compiles this bytecode into machine code just before execution.

Example Use Cases

Java: Known for its “write once, run anywhere” feature, popular in web and Android development.

C#: Often used in enterprise applications, game development, and the Microsoft ecosystem.

Interpreted Languages

In interpreted languages, code is **not compiled** beforehand. Instead, an **interpreter** reads and executes the source code line-by-line at runtime. This method makes interpreted languages more flexible and **easier for beginners** but often **slower than compiled languages**.

Interpreted Languages workflow:

[Source Code] - (Interpreter) → [Executed Line-by-Line by CPU]

How It Works

Write: Code is written in an interpreted language (e.g., Python, JavaScript).

Interpret: The interpreter reads and executes the code line-by-line in real-time.

Execute: Each line of code is translated into machine code at runtime, often slower than compiled code.

Example Use Cases

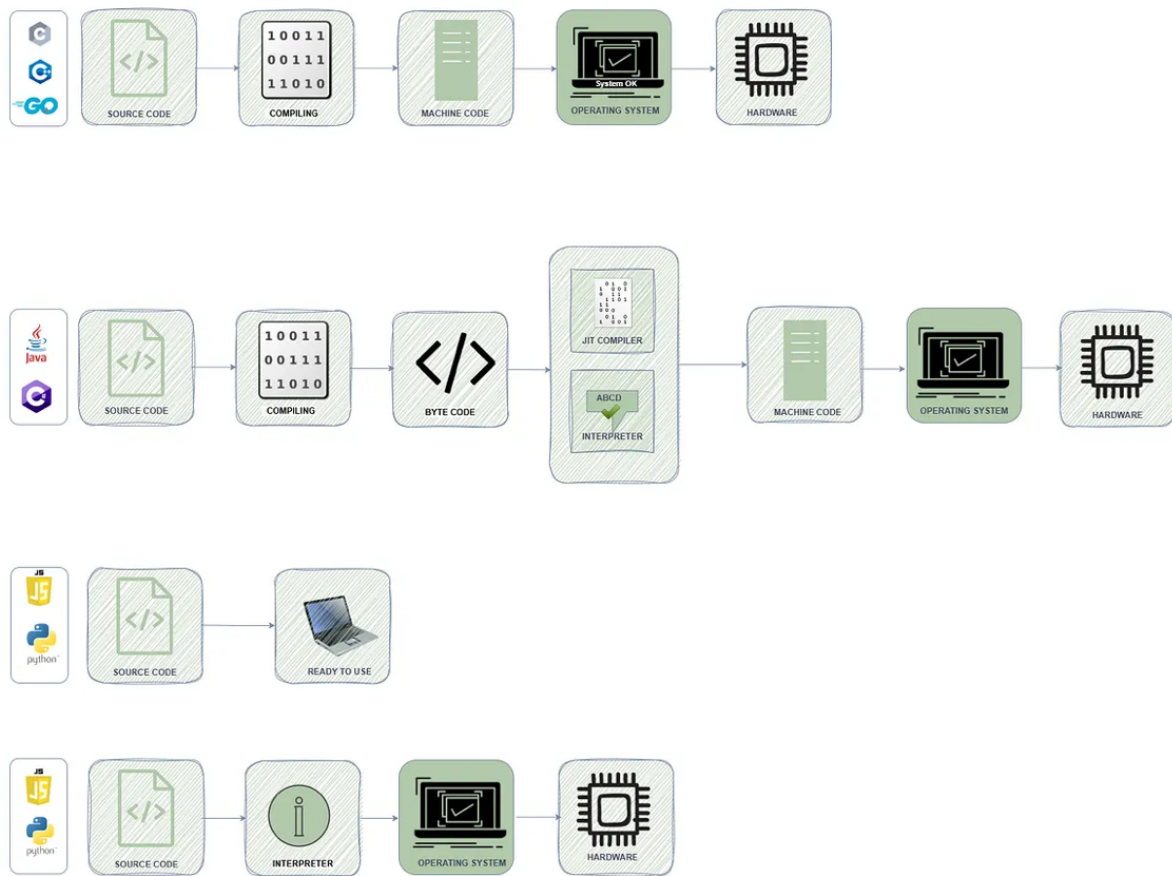
Python: Widely used in data science, machine learning, and web development.

JavaScript: Essential for creating interactive web applications.

Comparison Summary

Language Type	Conversion Method	Execution Method	Examples
Compiled	Source → Machine Code	Executed directly by CPU	C, C++, Go
Bytecode	Source → Bytecode → Machine Code	Bytecode executed by VM or JIT-compiled	Java, C#
Interpreted	Source → Line-by-Line Execution	Translated to machine code at runtime by interpreter	Python, JavaScript, Ruby

Each type offers distinct advantages, from the speed of compiled languages to the portability of bytecode languages and the flexibility of interpreted languages.



Understanding Source Code, Machine Code, Bytecode, & Interpreted Code

C Language (Compiled Language)

Code Example (source code):

```
#include <stdio.h>
int main() {
    printf("Hello, World!\n");
    return 0;
}
```

```
}
```

Compilation to Machine Code

Step: Use **gcc hello.c -o hello** to compile this code.

```
gcc hello.c -o hello
```

Machine Code Output: The compiler produces an executable file (e.g., **hello**) with machine code that the CPU can directly execute.

C++ Language (Compiled Language)

Code Example (source code):

```
#include <iostream>
int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

Compilation to Machine Code

Step: Use **g++ hello.cpp -o hello** to compile.


```
g++ hello.cpp -o hello
```

Machine Code Output: Like C, this generates an executable (e.g., **hello**) that is in machine code format.

Java Language (Bytecode Language)

Code Example (source code):

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

Compilation to Bytecode

Step: Use **javac HelloWorld.java** to compile.

```
javac HelloWorld.java
```

Bytecode Output: The compiler produces a **.class** file (e.g., **HelloWorld.class**) containing bytecode. Here's an example of what Java bytecode might look like (in hexadecimal format):

```
cafebabe 00000034 00210007 07000201 00104865 6c6c6f57
...
```

Execution by JVM

The bytecode in **HelloWorld.class** is loaded by the Java Virtual Machine (JVM), which interprets or JIT-compiles it into machine code for execution.

Get Prayag Sangode's stories in your inbox

Join Medium for free to get updates from this writer.

Enter your email

Subscribe

C# Language(Bytecode Language)

Code Example (source code):

```
using System;
class HelloWorld {
    static void Main() {
        Console.WriteLine("Hello, World!");
    }
}
```

Compilation to Bytecode (Intermediate Language — IL)

Step: Use **csc HelloWorld.cs** to compile.

```
csc HelloWorld.cs
```

Bytecode Output: The compiler generates an executable (**HelloWorld.exe**), containing **Intermediate Language (IL) code**, which the **.NET runtime executes**. Here's an example of what IL code might look like:

```
IL_0001: ldstr "Hello, World!"  
IL_0006: call void [mscorlib]System.Console::WriteLine(string)  
...
```

Execution by .NET Runtime

The IL code is JIT-compiled by the .NET runtime to machine code when it runs.

Python Language(Interpreted Language)

Code Example (source code):

```
print("Hello, World!")
```

Interpretation by Python Interpreter

Execution: Run `python hello.py`, and the Python interpreter reads and

executes each line in real-time.

```
python hello.py
```

Python also generates **bytecode** when the script runs (e.g., **hello.pyc** file), which looks like:

```
e3 00 00 00 63 00 00 00 64 00 00 83 01 01 64 01 00 53 29 02 e9  
...
```

This bytecode is interpreted or JIT-compiled as needed by the Python interpreter.

JavaScript Language (Interpreted Language)

Code Example (source code):

```
console.log("Hello, World!");
```

Execution by JavaScript Engine

Execution: JavaScript engines (e.g., Chrome's V8) interpret this code and

execute it line-by-line in the browser or Node.js environment.

JavaScript engines often compile code **just-in-time (JIT)** into machine code for faster execution, but this is handled at runtime, so developers don't typically see bytecode or machine code directly.

Summary Table

Language	Type	Source Code	Intermediate/Bytecode Example	Execution Method
C	Compiled	.c file	Machine Code (hello)	Executed directly by CPU
C++	Compiled	.cpp file	Machine Code (hello)	Executed directly by CPU
Java	Bytecode	.java file	Bytecode (HelloWorld.class)	JVM interprets or JIT-compiles
C#	Bytecode	.cs file	IL code (HelloWorld.exe)	.NET runtime JIT-compiles
Python	Interpreted	.py file	Bytecode (hello.pyc, optional)	Interpreted by Python
JavaScript	Interpreted	.js file	Not saved as bytecode	Interpreted/JIT-compiled

In the context of programming languages and compilation, the linker plays a crucial role in the process of converting source code into an executable program. Here's a breakdown of what a linker does and its significance:

Role of the Linker

Combining Object Files:

When you compile a program, the compiler converts the source code (e.g., C or C++) into object files, which contain machine code but are not yet executable.

If your program is divided into multiple source files, each will be compiled into its own object file.

The linker combines these object files into a single executable file.

Resolving References:

During compilation, function and variable references may be declared but not defined in a single source file.

The linker resolves these references by linking together the object files.

For example, if one source file calls a function defined in another source file, the linker ensures that the call points to the correct memory address

where the function resides.

Linking Libraries:

The linker can also link external libraries that your program depends on, such as standard libraries (like the C Standard Library).

It can include static libraries (integrated into the final executable) or dynamic libraries (linked at runtime).

Creating an Executable:

Once all the object files and libraries are combined, the linker generates the final executable file that can be run on the target system.

This file contains all the necessary machine code, along with any required metadata for execution.

Handling Symbols:

The linker maintains a symbol table that keeps track of all the variable and function names (symbols) in the program.

It uses this information to resolve which addresses correspond to which symbols during linking.

Example Workflow

Compilation:

Source code written in a high-level language is compiled into object files.

For example, in C:

```
// file1.c
int add(int a, int b) {
```

Medium

Search

Write

Sign up

Sign in



```
// file2.c
#include <stdio.h>
extern int add(int, int);
int main() {
    int result = add(5, 3);
    printf("Result: %d\n", result);
    return 0;
}
```

Object generated using gcc -c

```
gcc -c file1.c generates
file1.o
```

and

```
gcc -c file2.c generates
file2.o
```


Linking:

The linker combines **file1.o** and **file2.o** into a single executable:

```
gcc file1.o file2.o -o myprogram
```

This creates **myprogram**, which can be executed.

The linker is an essential component in the build process of programs written in compiled languages.

It takes care of combining multiple object files, resolving references, and creating a final executable that the operating system can run.

Understanding the linker's role helps clarify how different components of a program interact and are assembled into a cohesive application.

I hope you found this article to be useful in some way. I will be back with more such articles soon.



Written by Prayag Sangode

146 followers · 53 following

Follow

Cloud | Kubernetes | DevSecOps | LLMOPs | MLOPs | DataOps Enthusiast

Responses (1)



Write a response

What are your thoughts?



Sourav Sarangi

Jan 26




Nice explanation.



Reply


More from Prayag Sangode

 Prayag Sangode

OpenShift Installation On VMware

3 Node OpenShift Cluster on VMWare

Nov 6, 2023 🖱 9

 Prayag Sangode

Installing Kubeflow

In this article we will learn about installing Kubeflow on minikube cluster.


May 29, 2024 🖱 14 💬 2

 Prayag Sangode

Thanos vs VictoriaMetrics: Solving Prometheus' Limitations with...

When working with Prometheus for monitoring, many teams quickly realize that...

Sep 25, 2024 🖱 3

 Prayag Sangode

GCP Redis MemoryStore—An In-Memory Database

In-Memory database: A database that keeps all of its information in volatile memory, suc...

Jul 27, 2023 🖱 20 💬 2



[See all from Prayag Sangode](#)


Recommended from Medium

 Sohail Saifi

Why Japanese Developers Write Code Completely Differently (An...

For the past three years, I have delved into the software development landscape in...

★ Jul 17 🖱 11.1K 💬 288


 Yash Batra

The 47-Line Code That Made One Developer \$2 Million from AI

In late 2024, a solo indie developer pushed 47 lines of Python code to GitHub.

★ Jul 8 🖱 1.8K 💬 57




 Abhinav

Docker Is Dead—And It's About Time

Docker changed the game when it launched in 2013, making containers accessible and...


★ Jun 8 🖱️ 3.9K 💬 92

 In Predict by iswarya writes

GPT-5 Is Coming in July 2025—And Everything Will Change

“It’s wild watching people use ChatGPT... knowing what’s coming.” —OpenAI insider


★ Jul 7 🖱️ 10.6K 💬 388

 In Python in Plain English by Maria Ali

Why Every Serious Python Developer Is Quietly Switching to...

Selenium is the past. Puppeteer is Chrome-only. Playwright is quietly taking over

★ 5d ago 🖱️ 551 💬 8

 Artem Khrienov

Modern Terminal Emulators: Ghostty vs iTerm2

Terminal emulators are a critical tool for developers, sysadmins, and power users....

Feb 1 🖱️ 45



See more recommendations

[Help](#) [Status](#) [About](#) [Careers](#) [Press](#) [Blog](#) [Privacy](#) [Rules](#) [Terms](#) [Text to speech](#)