Topic 1.1: Compiled versus Interpreted Languages

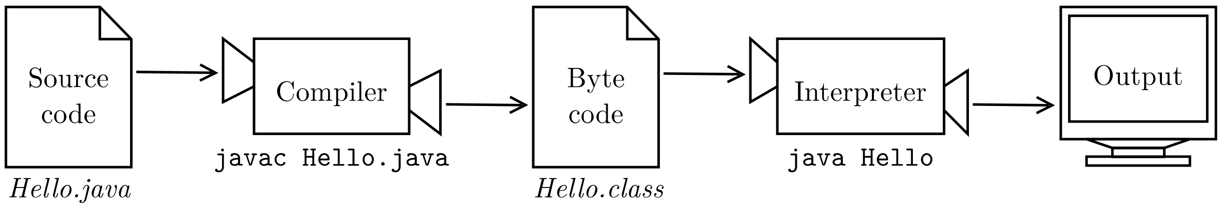
Compiled Programming Languages

Compiled languages, including C, C++, Rust, Golang, and many others, operate by translating human-readable source code into *machine code* – the binary instructions understood by a specific computer processor. This translation is performed in advance by a compiler, resulting in an executable file that can be run independently by the operating system. Machine code is tailored to a particular instruction set architecture (e.g., x86 or ARM), so an executable compiled for one processor will not run on another without a complex translation or emulation layer.

Because compilation is done before execution and the resulting code is ready for the processor to execute, well-written, compiled code will be the fastest code possible.

Compiled Java

Although working with an Integrated Development Environment (IDE), such as Eclipse or VS-Code makes it less obvious, Java code is compiled. However, unlike the compiled languages mentioned above, Java is *not* compiled into processor-specific machine code. Java is compiled into a set low-level of instructions, called bytecode, designed to be executed by the Java Virtual Machine (JVM).



When a Java program is run, the JVM interprets the bytecode (optimizing with Just-In-Time, or JIT, compilation, which I won't discuss) to convert it into native machine code for the underlying hardware. Because bytecode contains no high-level structures like identifiers or loop constructs, only low-level, stack-based operations similar to actual machine code, this translation step is highly efficient, allowing Java to achieve performance very close to (70% to 95%) that of traditionally compiled languages, such as those mentioned above.

Interpreted Languages

Traditional interpreted languages, such as early versions of BASIC or LISP, operate by reading source code line-by-line during execution. For each line of code, the interpreter translates that single line into machine code, executes it, then proceeds to the next line. This approach means that even if a certain line is run multiple times, for example inside a loop, that line will be translated anew on every iteration, contributing to the performance overhead commonly associated with interpreted languages.

Interpreted languages are known for being very slow – when actually interpreted line-by-line, it would not be surprising to be one hundred times slower than a compiled language.

Optimizations to the Modern Interpreters (e.g. Python)

Python improves upon this simple line-by-line approach. When a Python program is executed, the source code is first compiled into intermediate a bytecode, similar in concept to Java bytecode. Unlike Java, which compiles all bytecode ahead of time, Python compiles modules incrementally as they are imported and executed. This compiled bytecode is then fed into the Python Virtual Machine (PVM), that interprets and runs the bytecode on the underlying hardware.

To improve performance across multiple runs, Python may cache compiled bytecode to disk in .pyc files within a `__pycache__` directory. Subsequent executions can bypass the compilation step entirely if the source code has not changed, loading the pre-compiled bytecode directly.

With these optimization, at a speed of 10% to 50% that of C++.

Python for Artificial Intelligence and Numerical Analysis

A significant performance consideration in Python programming is that many core libraries are *not* implemented in pure Python at all. Numerical computing libraries such as NumPy, SciPy, and Pandas are predominantly written in C and C++, with Python acting as a high-level interface.

For example, with Artificial Intelligence GPU inference or training, a well-written PyTorch training loop might spend over 99% of its time in C++/CUDA kernels, and the Python overhead becomes just a rounding error.

Thus, the simplicity and elegance of Python code makes Python a winner for many applications.