Unit 2c: Execution Model

1 Introduction

Some differences between languages lie in the process the programmer has to follow to create and execute statements and/or complete programmes. Some languages allow the programmer to write and execute code interactively at some kind of command prompt while others require complete units of source code to be explicitly compiled before being executed.

There are also differences in the way the program code is translated into something that will execute on the physical processor on the computer: compiled vs. interpreted; native code vs. virtual machine.

These aspects of a programming language are independent of typing system and programming model.

2 Compilation (native)

Compilation is the process of translating source code into something that is executable; the compilation process and tools are specific to a language and may be specific to a platform (OS/processor) e.g. *gcc* compiles C/C++ on the Linux platform.

The process actually involves a number of steps and tools, each of which may create intermediate files other than the original source and final executable:

A *Compiler* translates source code to *assembly language* which expresses machine level instructions for a specific platform; the compiler also *checks syntax* of source code and will not complete the translation if errors found.

An *Assembler* turns this into *Object code* which includes data to allow a unit of code to be linked with external code that it depends on.

A *Linker* combines object files and libraries into a single *native executable* program file (which can then be executed directly by the OS/processor).

Compilation involves several key stages/tools. A *Tokenizer (or lexer)* recognises "words" i.e. divides input (source code) into individual tokens (strings with identified meaning) and passes these to a parser. The *Parser* recognises structure: it has knowledge of language syntax, identifies syntax errors and translates token stream into a data structure that represents the structure of the source code – the data structure is called the parse tree, or abstract syntax tree (AST). The *Code generator* uses the parse tree to generate code in the target language.

3 Static and dynamic libraries

Linking static libraries involves including the object file code from the library into the target executable binary. This results in a larger size on disk and slower launch times because the library's code is added directly to the linked target's binary, it means that to update any code in the library, the linked target would also have to be rebuilt.

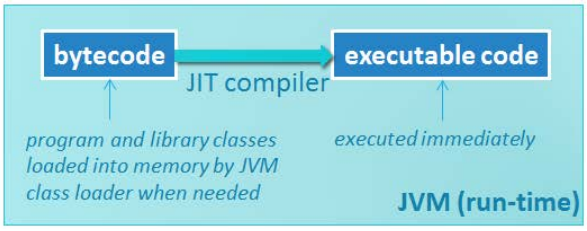
Linking dynamic libraries means none of the library's code is included directly into the linked target; libraries are loaded into memory at runtime prior to having symbols (function call interfaces) getting resolved. This means that libraries can be updated with new features or bug-fixes without having to recompile and relink executable e.g. Dynamic Link Libraries (DLLs) on Windows, shared libraries (.so) on Linux.

4 Virtual machines

A virtual machine (VM) is a software-driven emulation of a given computer system. There are many types of virtual machines, and they are classified by how precisely they are able to emulate or substitute for actual physical machines.

- A System VM emulates a complete platform on which you can run an operating system, e.g. VMWare.
- A Process VM is less fully-functional and can run one program or process, e.g. run a Windows executable on a Linux OS.
- A Language VM is a kind of process VM which provides a layer of abstraction between compiled source code and a native platform: the compiler produces some intermediate language which is similar to assembly language but is not specific to a platform and is translated by the VM to native code at runtime - compile once, run anywhere.

Compilation for Java does not create native executable code; the compiler creates bytecode (in .class files) which is not specific to any platform. A Java Virtual Machine runs on top of the OS; JVMs are available for many platforms, and translate bytecode to native executable code at run time using a Just-in-time (JIT) compiler. The extra work during runtime can affect performance, though modern JVMs can do optimisations.



A number of other languages can run on the JVM, e.g. Scala, Groovy, code is compiled to bytecode. Some other modern "compiled" languages use similar VM-based model, e.g. .NET languages.

5 Interpreters

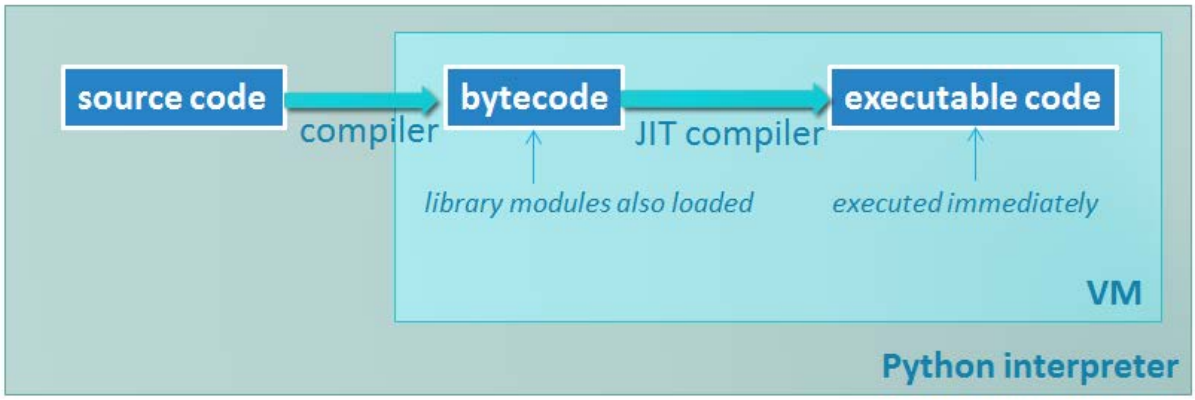
An interpreter directly executes instructions written in a programming or scripting language, without previously compiling them into a machine language program; there is no checking of syntax before run-time, execution stops at point where an error is encountered. This is useful for interactive execution of code and rapid script development.

Strategies include: parse the source code and perform its behaviour directly; or, translate (compile) source code into an intermediate representation and immediately execute this.

Most modern "interpreted" languages (e.g. Python, Ruby, JavaScript, Groovy) use the second strategy – compile to some kind of bytecode which is executed by a VM.

The distinction between compilation/interpretation is blurred.

5.1 Example – Python interpreter



Actually there is no such thing as an "interpreted language" or a "compiled language" – there can be different implementations of the same language that use different strategies for execution e.g. while all current implementations of Python use the VM strategy, there are versions that use different VMs, e.g. JVM (Jython), .NET CLR (IronPython).

6 Interactive language shell

Commonly known as a *read-eval-print loop (REPL)*; this evaluates each expression or statement as it is entered and displays output from any "print" statements - may also display results of expression evaluation.

7 Scripting

Script is created as a source code file containing statements and executed without the need to compile first.

8 Summary

Language implementations for "interpreted" languages typically provide REPL and command line tools to run scripts. Note JavaScript is a special case in that scripts can also be embedded in web

pages and executed by browser. Execution in either mode makes use of one of the interpreter strategies described earlier.

Languages such as Java, C, C# are considered to be “compiled” languages and the standard developer kits don’t include interactive tools such as REPLs. Note that the java command line tool is sometimes called the “java interpreter” but it invokes execution of code that has been previously compiled.

Java doesn’t lend itself well to interactive execution of statements as syntax requires at least one class, with a main method, to be defined, however, there are implementations of REPLs for Java which behind the scenes need to “wrap” statements in a class for compilation and execution - CodePad in the BlueJ IDE is an example of this; similar tools can be found for other compiled languages.

Interaction modes are related to the tools available more than to the language itself.