Compilation and Interpretation; Overview of Java Virtual Machine

Ata Kaban
University of Birmingham

Levels of programming

- High level
  - e.g. Java, C, Prolog, Haskell, etc
  - Easier for humans
- Lowest level
  - Machine code – instructions stored in memory (...opcodes)
  - hard to read or write by humans
- Next level up: Assembly code
  - Can be written or read by humans (...mnemonics)

Converting high level to low level

- To execute on a computer we must have machine code
- Assembly code is translated to machine code to run
  - Assembler does this (e.g. works out the *relative* addresses for jumps etc.). Relocatable.
  - Linker: combines different assembled parts into a whole
  - Loader: loads into memory ay a given location

Executing high level programs

- A program P written in a high level language can be run in 2 ways:
  - Compiled into a program in the native machine language and then run on the target machine
  - Directly interpreted and the execution is simulated within an interpreter

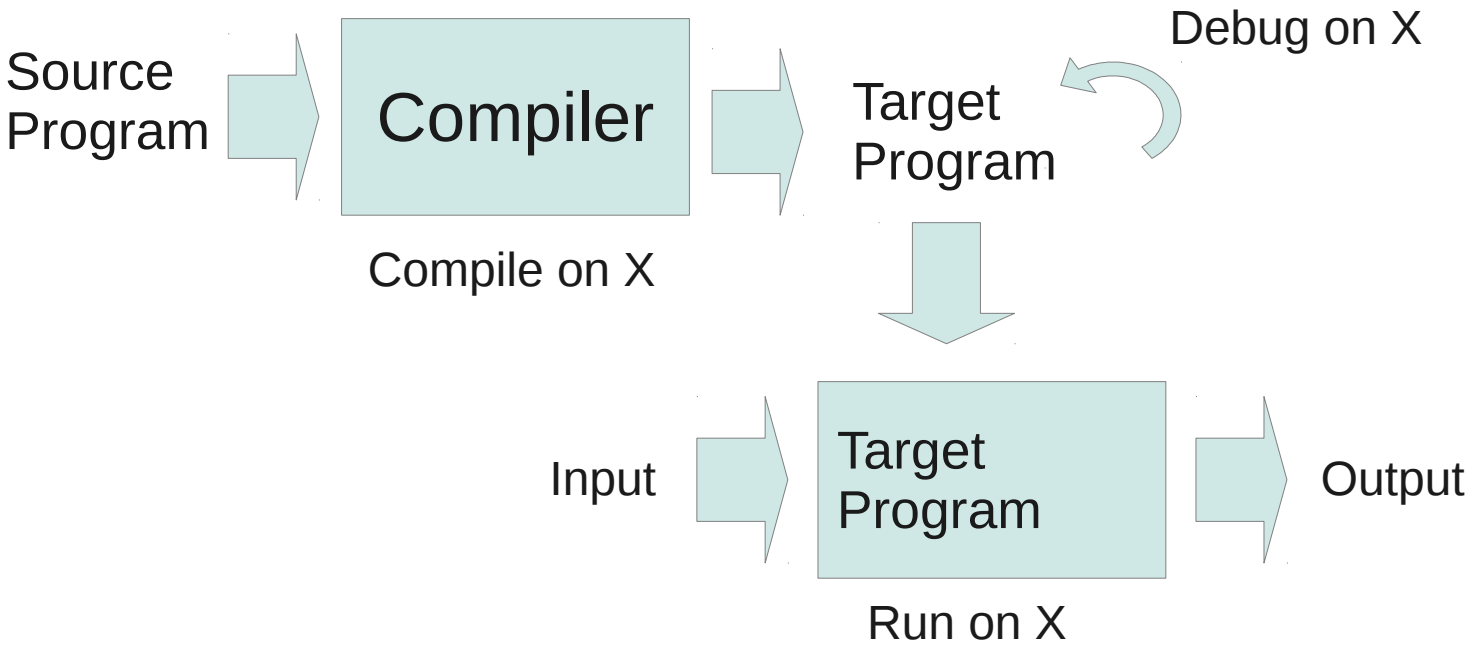
- Q: Which approach is more efficient?
  - Think of C++ vs. Python

Compilation

- Compiler: converts source code (text of P) into object code – e.g. machine code – that does the same thing as P
- Usually object code is relocatable, so can be later linked and loaded
- Advantages:
  - Done once for each P
  - With clever tricks to optimise object code (by exploiting hardware features) so it will run fast
- Disadvantages:
  - Harder than interpreting
  - Hardware dependent

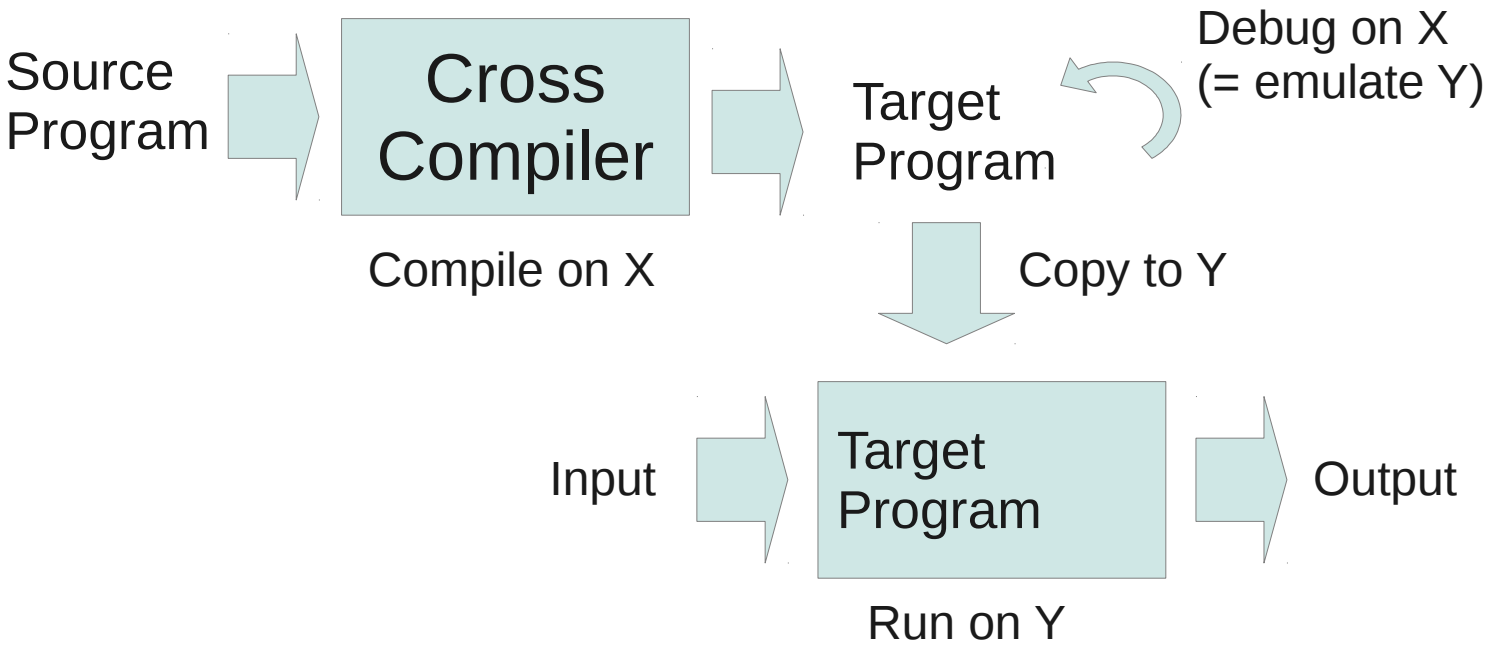
Compilation

- Compiler runs on the same platform X as the target code



Cross Compilation

- Compiler runs on platform X, target code runs on platform Y

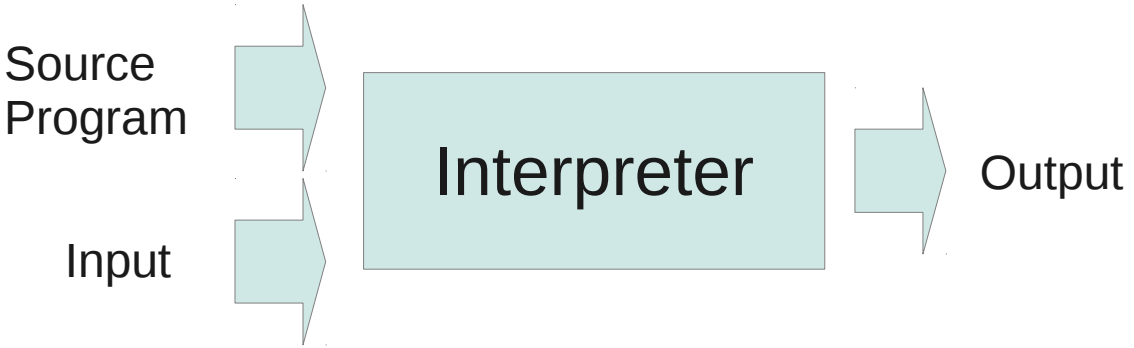


Interpretation

- Interpreter = another program that follows the source code (text of P) and does appropriate *actions*
- Same principle as:
  - Humans running through instructions of P
  - A processor (CPU) can be viewed as a hardware implementation of an interpreter for machine code
- Advantages:
  - Facilitates interactive debugging & testing
    - User can modify the values of variables; can invoke procedures from the command line
- Disadvantages:
  - slow

Interpretation

- Running high-level code by an interpreter



Compiling combined with interpreting

Executing high level programs P

- Compile to an intermediate level (between high and low) language that can be efficiently interpreted
  - Slower than pure compiling
  - Faster than pure interpreting
  - A single compiler, independent of CPU
  - Separate task for each CPU is to interpret the intermediate language

E.g. Java

Executing high level programs P

Source code
.java files

- Compile to an intermediate level (between high and low) language that can be efficiently interpreted

Java bytecode
.class files

  - Slower than pure compiling
  - Faster than pure interpreting
  - A single compiler, independent of CPU

javac

  - Separate task for each CPU is to interpret the intermediate language

The command 'java' calls the JRE

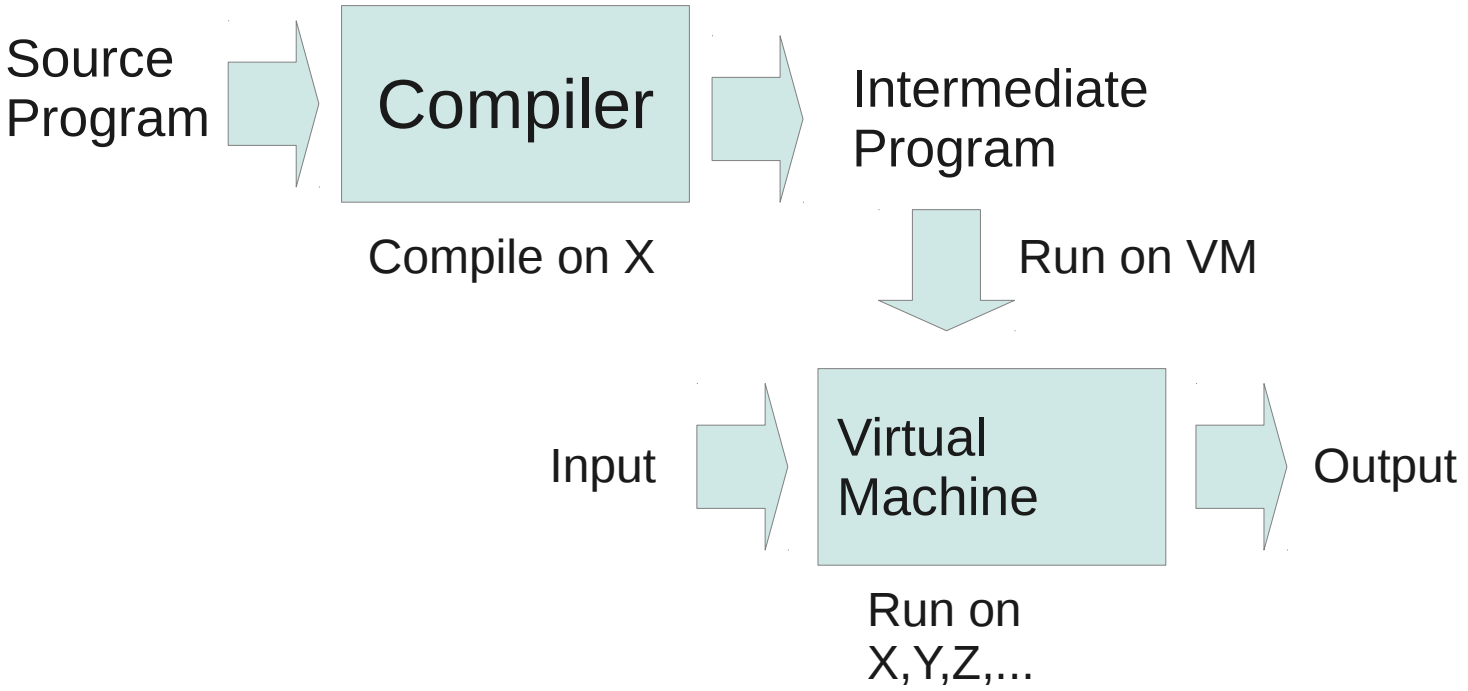
Java Runtime Environment (JRE) using Java Virtual Machine (JVM)

Virtual Machines

- A virtual machine executes an instruction stream in software
- Adopted by Pascal, Java, Smalltalk-80, C#, functional and logic languages, and some scripting languages
- Pascal compilers generate P-code that can be interpreted or compiled into object code
- Java compilers generate bytecode that is interpreted by the Java virtual machine (JVM)
- The JVM may translate bytecode into machine code by just-in-time (JIT) compilation

Compilation and execution on virtual machines

- Compiler generates intermediate program
- Virtual machine interprets the intermediate program
  - Have virtual machine on each platform



The Java Virtual Machine (JVM) - overview -

- The concept and design
- Stacks and their role
- Instructions and their format
- Compiling to JVM

The Java concept

- Before Java…[Bell Labs]
  - C and C++ (object-oriented C) for systems programming
  - WWW evolving fast
- How to load and run a program over WWW?
  - different target machines, word length, instruction sets
  - security an issue
- Java [mid-1990s, Sun Microsystems]
  - language based on C++
  - has a virtual machine, hence portable
  - can be downloaded over WWW and executed (applet)

Portability of Java

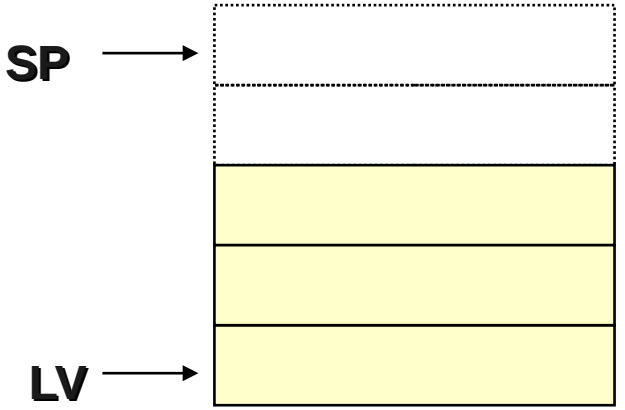
- Why not compile Java to machine code?
  - need to generate code for each target machine
  - cannot exchange executable code
- The Sun Java solution
  - design machine architecture (JVM) specifically for Java
  - translate Java source code into JVM code (bytecode)
  - write software interpreter for JVM in C (widely available)
- Thus
  - bytecode can be exchanged
  - remote execution possible

The JVM architecture

- The architecture
  - Stack machine! Closer to modern high-level languages than the von Neumann machine.
  - Memory: 32 bit words (=4 bytes)
  - Instructions: 226 in total, variable length, 1-5 bytes
  - Program: byte stream
  - Data: stored in words
  - Program Counter (PC) contains byte addresses
- Here simplified, Integer JVM (IJVM)
  - no floating point arithmetic

Stack Machines

- Stack
  - area of memory, extends upwards or shrinks down
  - LV, base of stack
  - SP, top of stack
- Operations
  - push on top (increment SP)
  - pop (decrement SP)
  - add top two arguments on the stack, replace with result

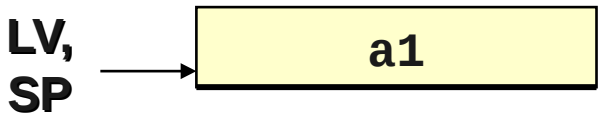


Evaluating expressions on stack

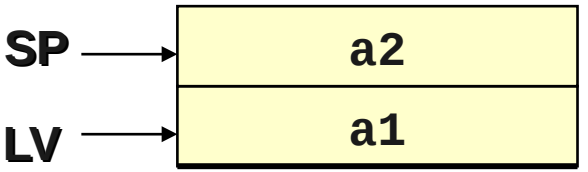
Evaluate

`a1+a2`

PUSH a1



PUSH a2



ADD

What are stacks good for?

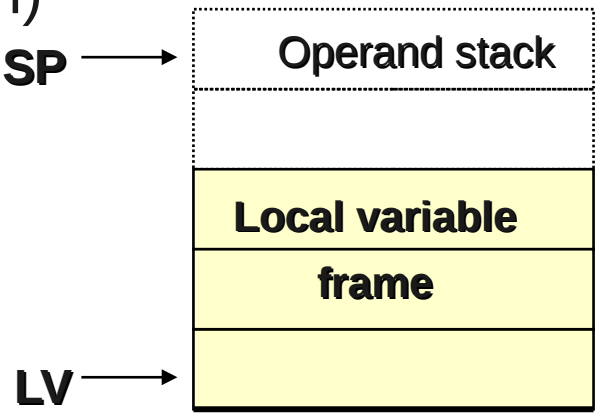
- Expression evaluation
  - can handle bracketed expressions

    `(a1+a2)*a3`

    without temporary variables:

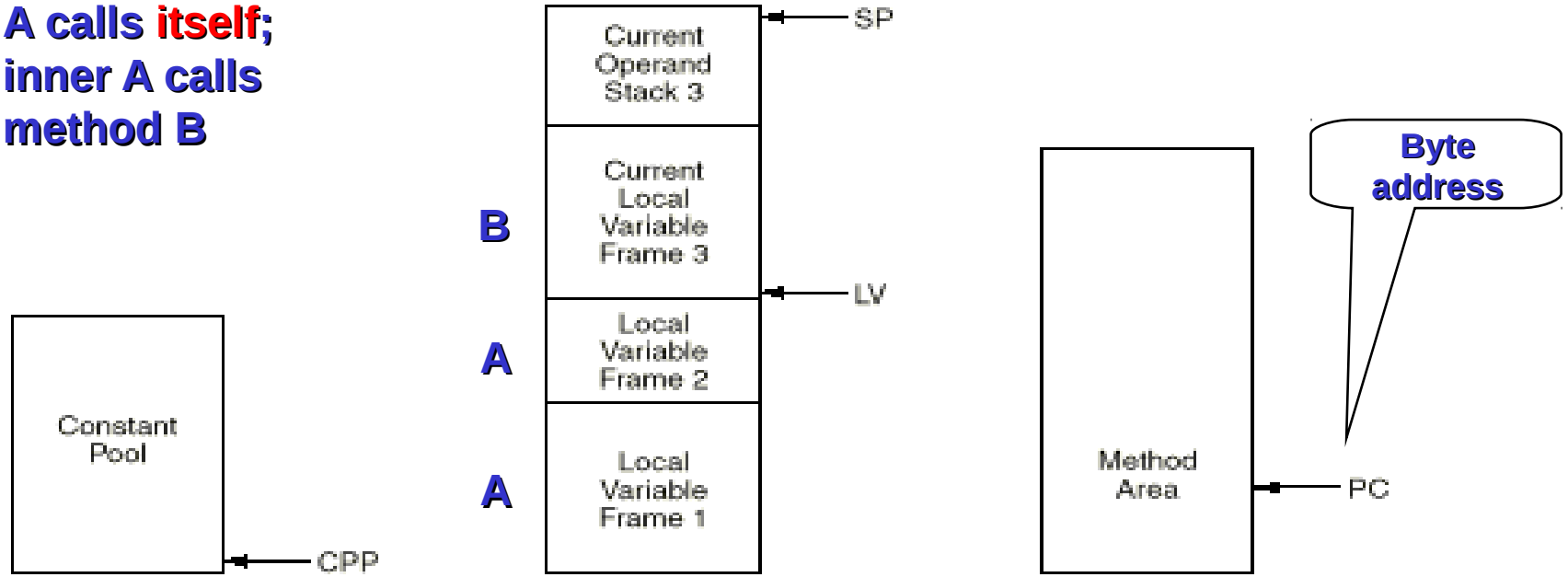
    PUSH a1, **PUSH a2**, **ADD**, **PUSH a3**, **MULT**

    (see also reverse Polish notation)
- Direct support for
  - local variables for methods (stored at the base of stack, deleted when method exited)
  - (recursive) method calls: to store return address



IJVM Memory

A calls itself; inner A calls method B



Protected area
(contains constants, strings, pointers, etc)

Stack
(local variables, expression eval.)

Method area
(contains the program – byte array)

Main IJVM Instruction Groups

- Stack operations
  - **PUSH/POP** - push/pop word on a stack
  - **BIPUSH** - push byte on stack
  - **ILOAD/ISTORE** - load/store local variable onto/from stack
- Integer Arithmetic
  - **IADD/ISUB** - add/subtract two top words on stack
- Branching
  - **IFEQ** - pop top word from stack, branch if zero
- Invoke a method/return from a method
  - **INVOKEVIRTUAL**, **RETURN**

All act on stack, not accumulator and memory

IJVM Instruction Set

One byte:
byte, const, varnum

Two bytes:
disp, index, offset

Hex	Mnemonic	Meaning
0x10	BIPUSH *byte*	Push byte onto stack
0x59	DUP	Copy top word on stack and push onto stack
0xA7	GOTO *offset*	Unconditional branch
0x60	IADD	Pop two words from stack; push their sum
0x7E	IAND	Pop two words from stack; push Boolean AND
0x99	IFEQ *offset*	Pop word from stack and branch if it is zero
0x9B	IFLT *offset*	Pop word from stack and branch if it is less than zero
0x9F	IF_JCMPEQ *offset*	Pop two words from stack; branch if equal
0x84	IINC *varnum const*	Add a constant to a local variable
0x15	ILOAD *varnum*	Push local variable onto stack
0xB6	INVOKEVIRTUAL *disp*	Invoke a method
0x80	IOR	Pop two words from stack; push Boolean OR
0xAC	IRETURN	Return from method with integer value
0x36	ISTORE *varnum*	Pop word from stack and store in local variable
0x64	ISUB	Pop two words from stack; push their difference
0x13	LDC_W *index*	Push constant from constant pool onto stack
0x00	NOP	Do nothing
0x57	POP	Delete word on top of stack
0x5F	SWAP	Swap the two top words on the stack
0xC4	WIDE	Prefix instruction; next instruction has a 16-bit index

Compiling Java to IJVM

Java	Intermediate	Hex	Stack
i = j+k	**ILOAD** j	0x15 0x02	j
	ILOAD k	0x15 0x03	k j
	IADD	0x60	j+k
	ISTORE i	0x36 0x01	

JVM Instruction Summary

- Different from most CPUs
- Closer to high-level programming languages, rather than von Neumann computer
- No accumulator/registers - just the stack!
- Small, straightforward instruction set
- Variable length of instruction
- Typed instructions, i.e. different instruction for LOADing integer and for LOADing pointer (this is to help verify security constraints)

Interpreting JVM

- Software interpreter for JVM in C (the original Sun Microsystems solution),
  - memory for the constant pool, method area and stack
  - procedure for each instruction
    - program which fetches, decodes and executes instructions
- Produce micro-programmed interpreter
- Manufacture hardware chip (picoJava II) for embedded Java applications
  - see e.g. Tanenbaum

Just In Time (JIT) Compiling

- Why not compile directly to target architecture?
  - more expensive - many varying architectures
  - more time needed to compile each instruction
- but
  - execution is slower with an interpreter!!!
  - instructions may have to be parsed repeatedly
- Just In Time Compiling…
  - include Java compiler to target machine within a browser
  - compile instructions, and reuse them
  - longer wait till arrival of executable code

Summary

- Compilation vs. Interpreting
- Interpreted languages
  - execute with the help of a layer of software, not directly on a CPU
  - usually translated into intermediate code
- Java
  - conceived as an interpreted language, to enhance portability and downloading to foreign architectures (applets)
  - has **JVM**, a virtual stack machine
  - interpreted via a C language interpreter, or a hardware chip (picoJava II for embedded Java applications)

Next time: Stacks and stack frames, in more detail.