UNIVERSITY OF BIRMINGHAM

# Compilation, Interpretation & Overview of Java Virtual Machine

# Lecture Objective

To introduce the basic concepts of compilation, interpretation and Java Virtual Machine.

# Lecture Outline

- Levels of Programming Languages
- High Level to Low Level Translation
- High Level Program Execution
- Compilation vs. Interpretation
- Combined Compilation & Interpretation
- Compilation and Execution on Virtual Machines

# Levels of Programming Languages

- **High** level languages
  - e.g. Java, C/C++/C#, Fortran, Cobol, Pascal, etc
  - Easier for humans
- **Low** level languages
  - Machine code – instructions stored in memory (opcodes)
  - Hard to read and write by humans
- Next level up: **Assembly** code
  - Can be written or read by humans (using mnemonics)

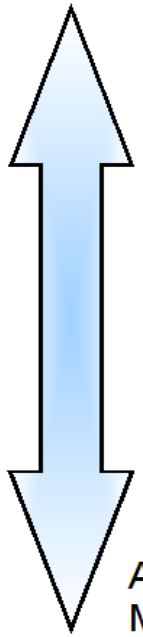
**Watch on Youtube:**

Most Popular Programming Languages 1965 – 2019

# Levels of Programming Languages

- High-level
  - Close to problem
  - System independent

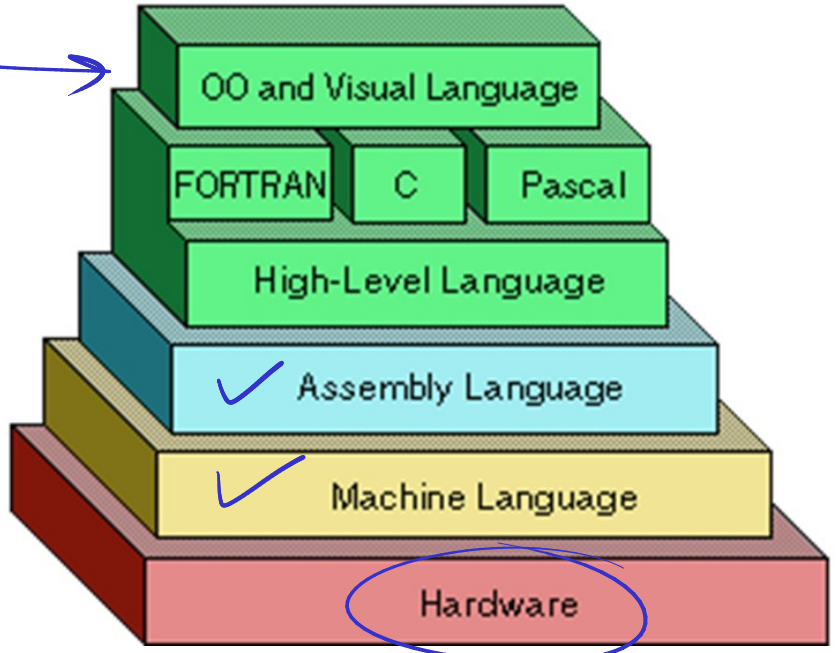
- Low-level
  - Close to system
  - Doesn't reflect problem



SQL
Java, C#
FORTRAN, COBOL, C++

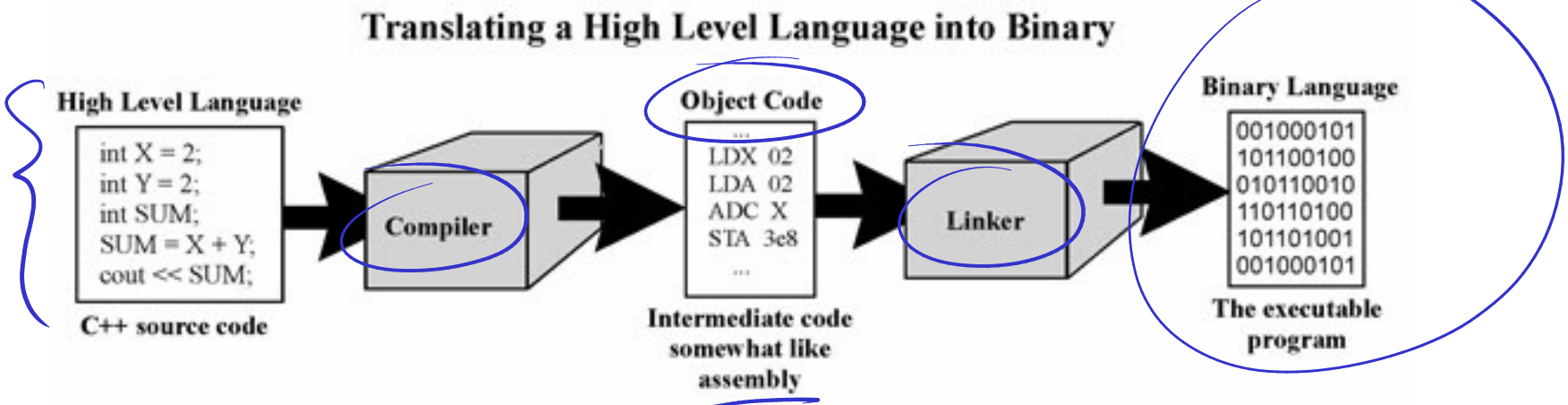
C/C++

Assembler
Machine



# 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 Code.
  - **Linker**: combines different assembled parts into a Whole
  - **Loader**: loads into memory at a given location



# Executing High Level Programs

- A **program** written in a high level language can be run in two different 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

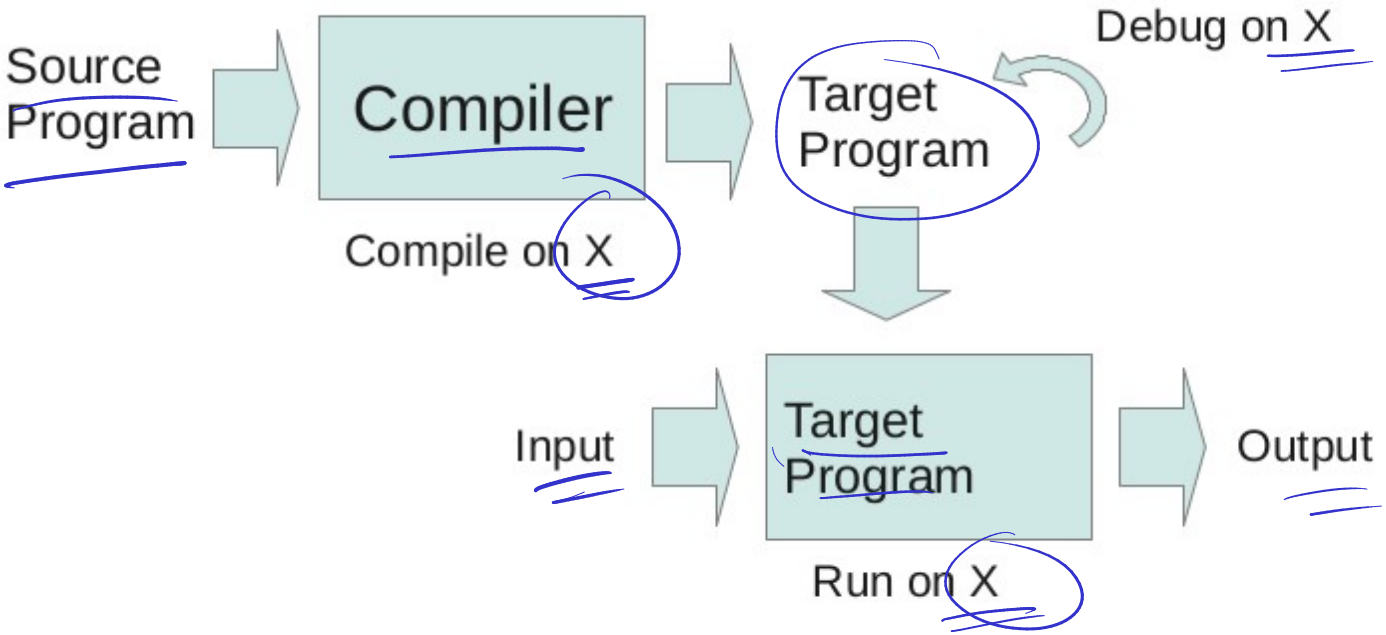
- Which approach is more efficient?
  - Think of C++ vs. JavaScript

# Compilation

- **Compiler:** converts source code (text of a program) into object code – e.g. machine code – that does the same thing as the original program
- Usually object code is **relocatable**, so can be later **linked** and **loaded** into memory.
- Advantages:
  - Done once for each program
  - With clever tricks to optimize object code (by exploiting hardware features) so that it will run fast
- Disadvantages:
  - Harder than interpreting
  - Hardware dependent i.e. cannot run of different platforms

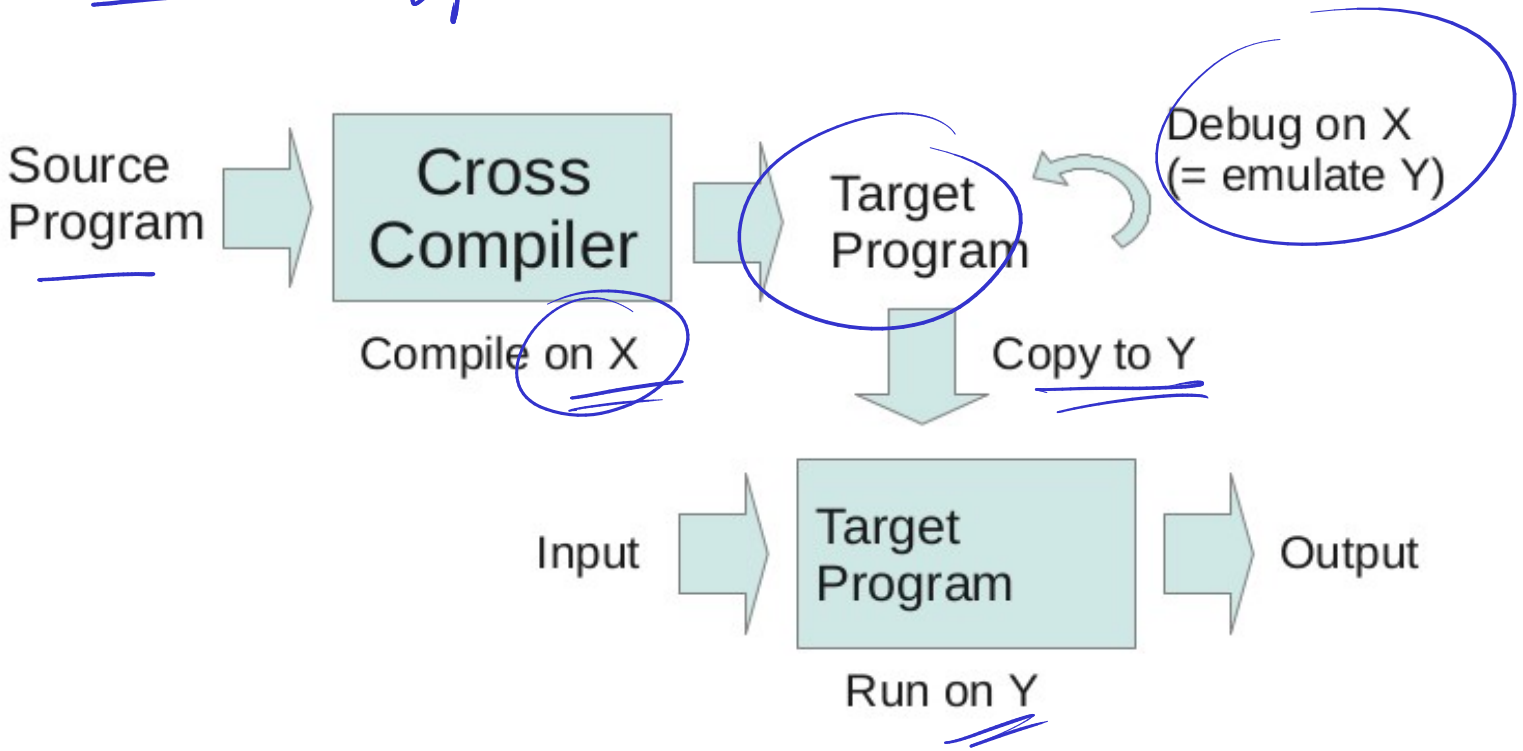
# Compilation

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

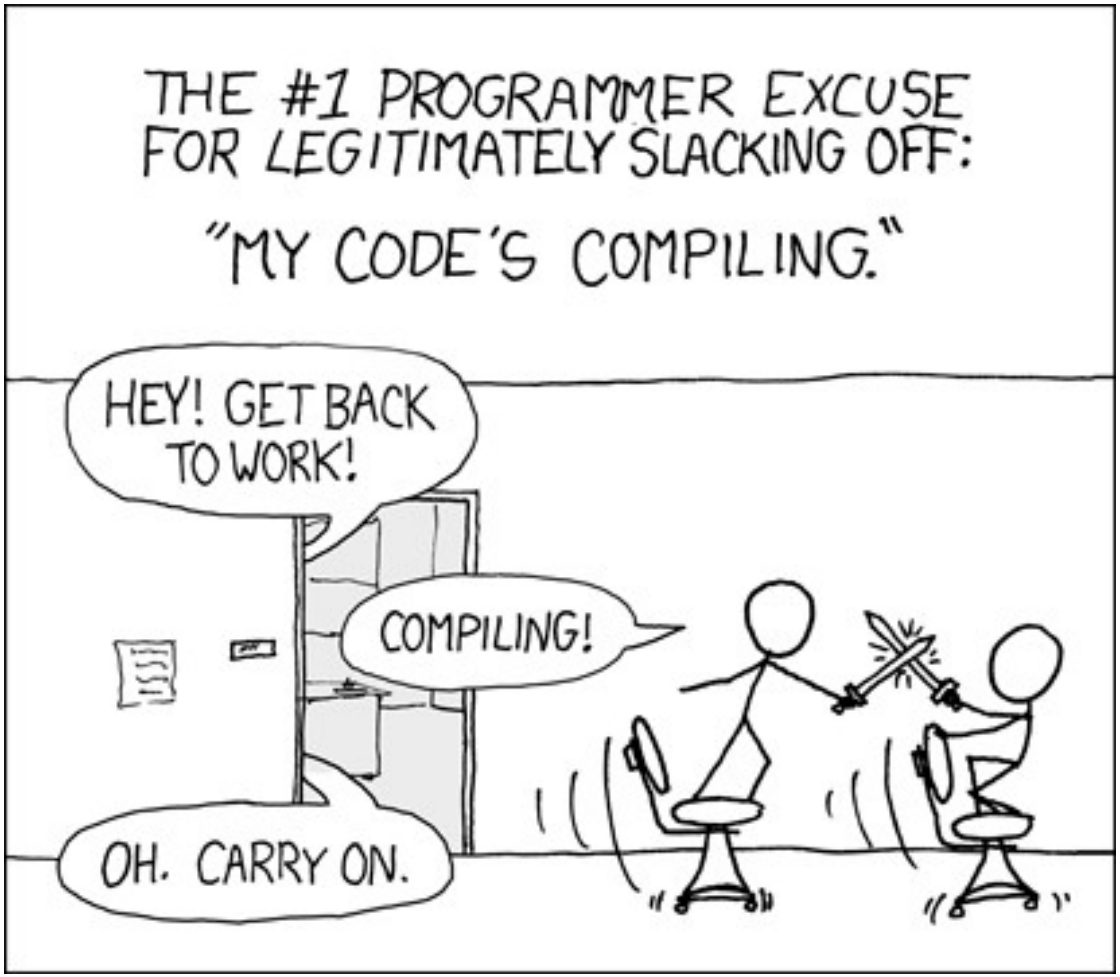


# Cross Compilation

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



# Compilation is a Compute Intensive process!



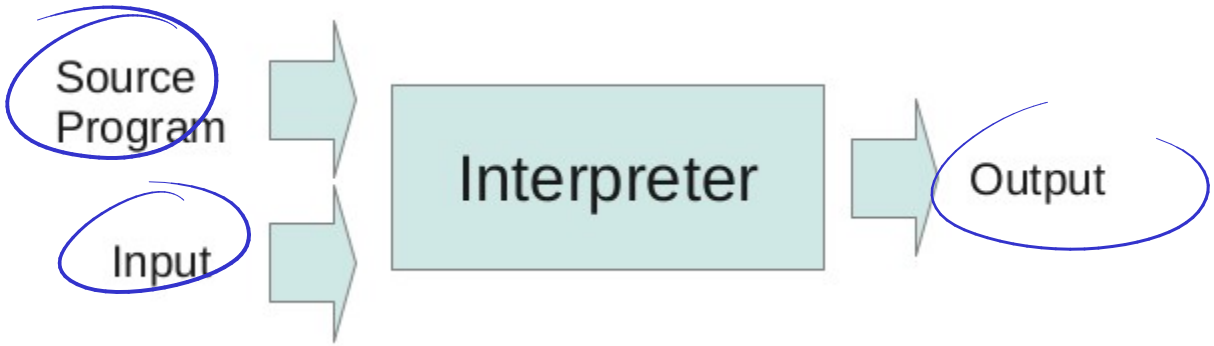

https://xkcd.com/303/

# Interpretation

- Interpreter = another program that follows the source code (text of program) and does appropriate actions
- Same principle as:
  - Humans running through instructions of a program
  - 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 Execution (as compared to compilation)

# Interpretation

- Running high-level code by an interpreter




**Watch on Youtube:**

Compiled vs. Interpreted Languages

# Research Example – Simulation Techniques

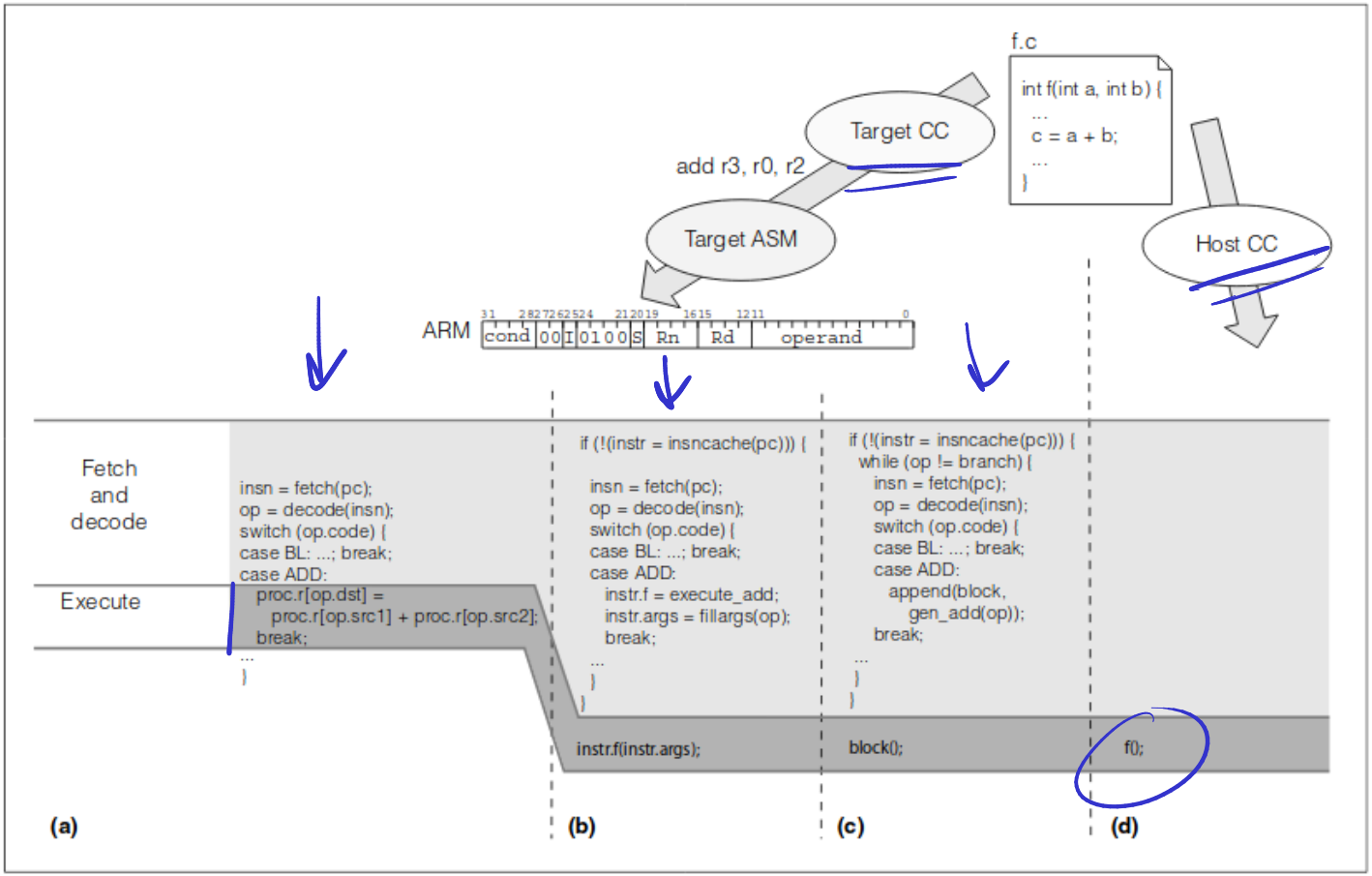



**Figure 2. Software simulation techniques applied to the ARM instruction-set architecture (ISA): instruction-accurate interpretation (a), interpretive predecoding (b), dynamic binary translation (c), and native code execution (d). (ASM: assembly; CC: C language compiler.)**

Full article: https://ieeexplore.ieee.org/document/5620924

# Combined Compilation & Interpretation

Executing high level programs

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

# Example: Java

Source Code
.java files

Java bytecode
.class files

Executing high level programs

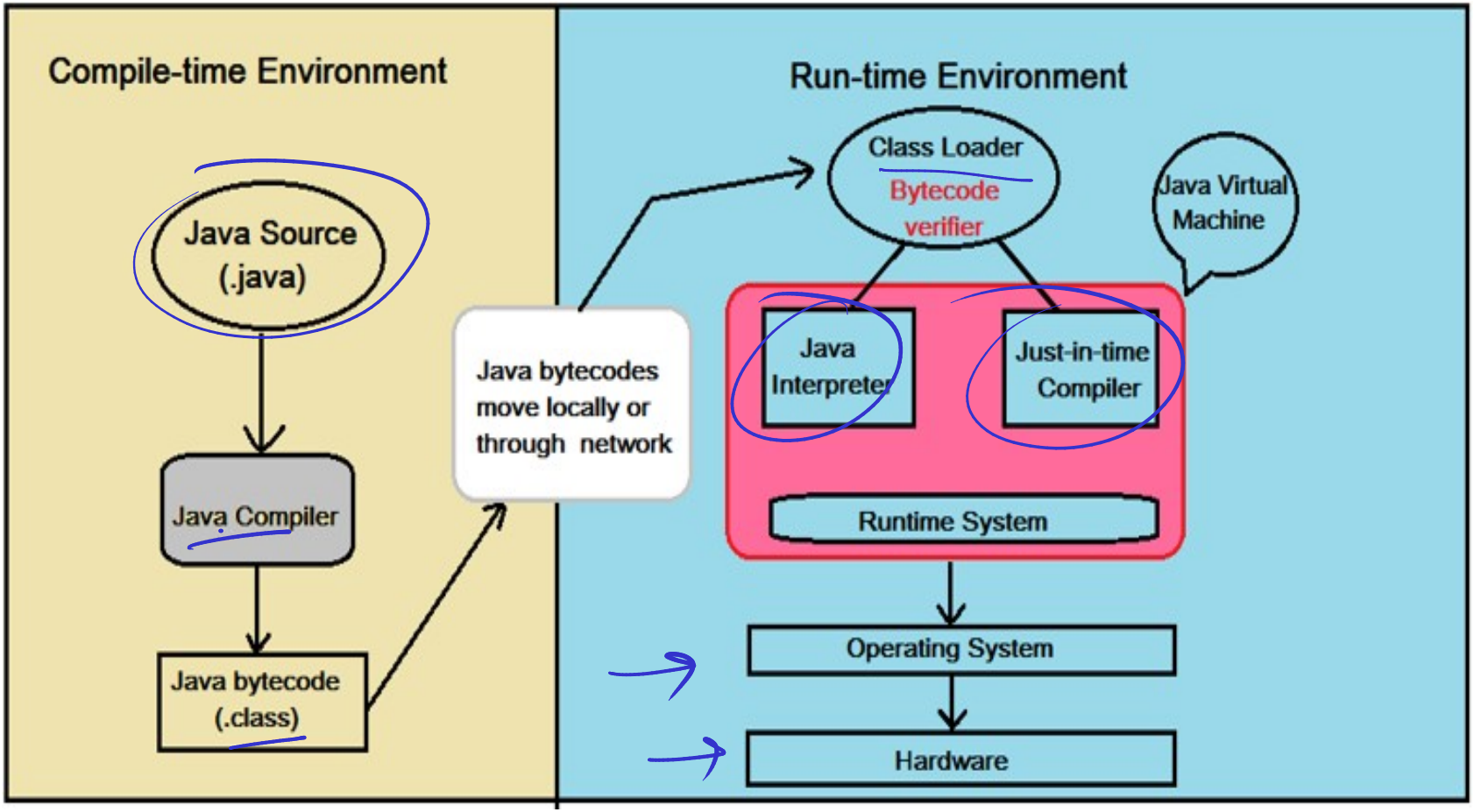
- Compile to an intermediate level (between high and low) language that can be efficiently interpreted
  - Slower than pure compilation
  - Faster than pure interpretation
  - A single compiler, independent of CPU
  - Separate task for each CPU is to interpret the intermediate language

javac

The command “java” calls the JRE

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

# Combined Compilation & Interpretation

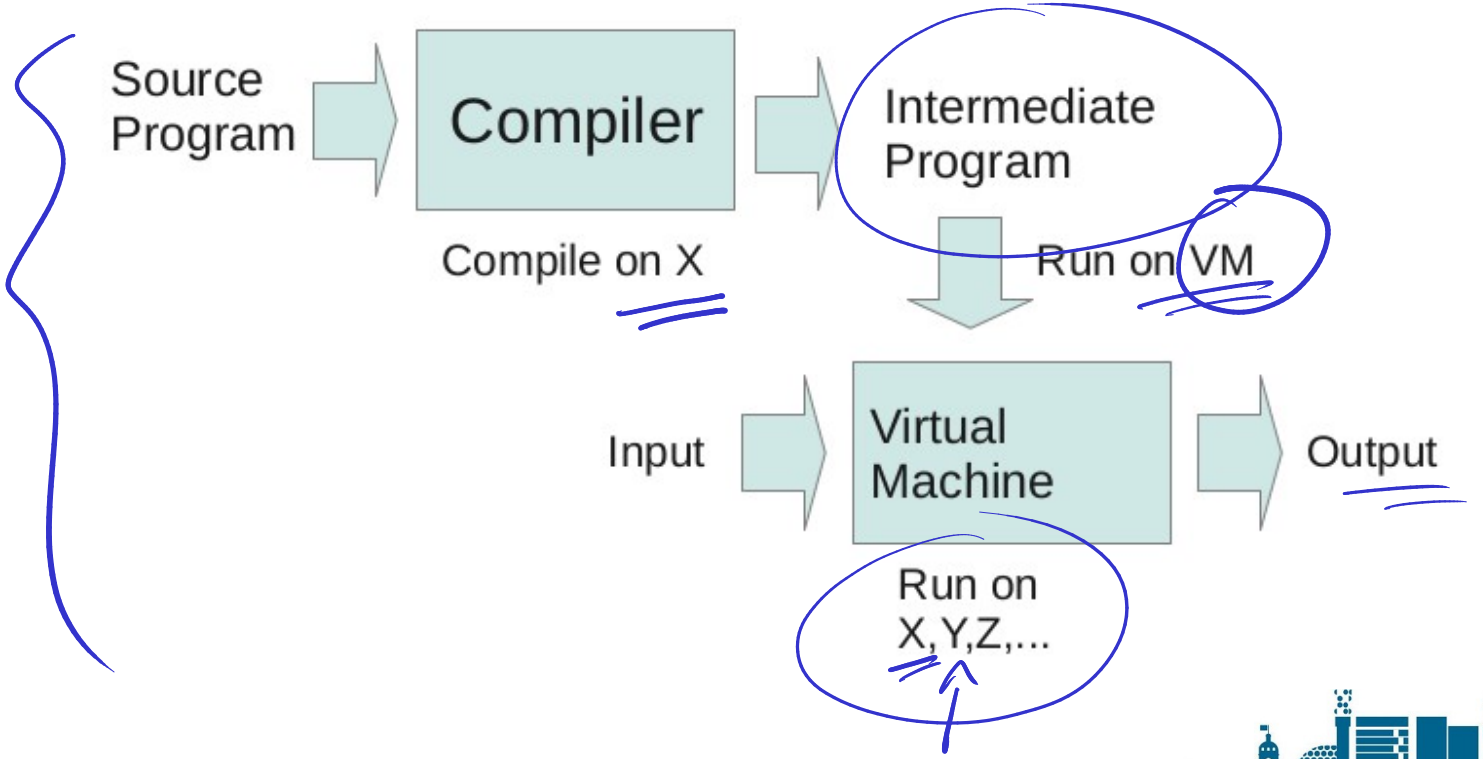


# Virtual Machines

- A virtual machine executes an instruction stream in software (instead of hardware)
- 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 (https://en.wikipedia.org/wiki/P-code_machine)
- 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 (language)
- Virtual machine interprets the intermediate program
  - We need to have virtual machine on each platform



# Java Virtual Machine (JVM) Introduction

**Watch on Youtube:**
What is Java Virtual Machine?

# Lecture Outline

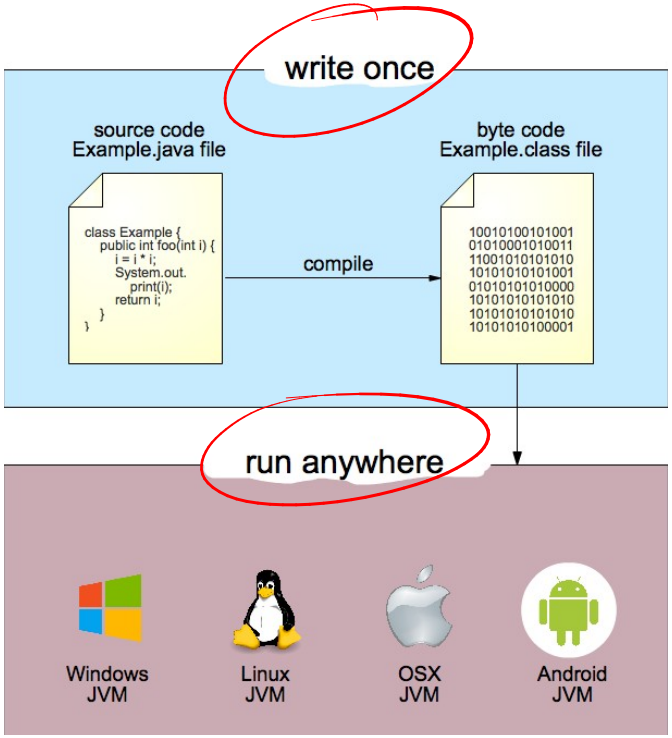
- Java Concept and Portability
- The JVM Architecture
- Stack Machines & Expression Evaluation
- IJVM & IJVM Instruction Set / Groups
- Compiling Java to IJVM
- JVM Instruction Summary
- Interpreting JVM & Just In Time (JIT) Compilation

# The Java Concept

- Before Java … [Bell Labs]
  - C and C++ (object-oriented C) were used for systems programming
  - WWW has evolved very fast (Animated History)
- How to load and run a program over WWW?
  - different target machines, word length, instruction sets
  - Security is another issue!
- **Java** [mid-1990s, Sun Microsystems]
  - language based on C++
  - has a virtual machine, hence portable
  - can be downloaded over WWW and executed remotely (using the applets)

# 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 the Java language
  - translate Java source code into JVM code (bytecode)
  - write software interpreter for JVM in C (widely available)
- Thus bytecode can be exchanged
  - remote execution is possible



# The JVM Architecture

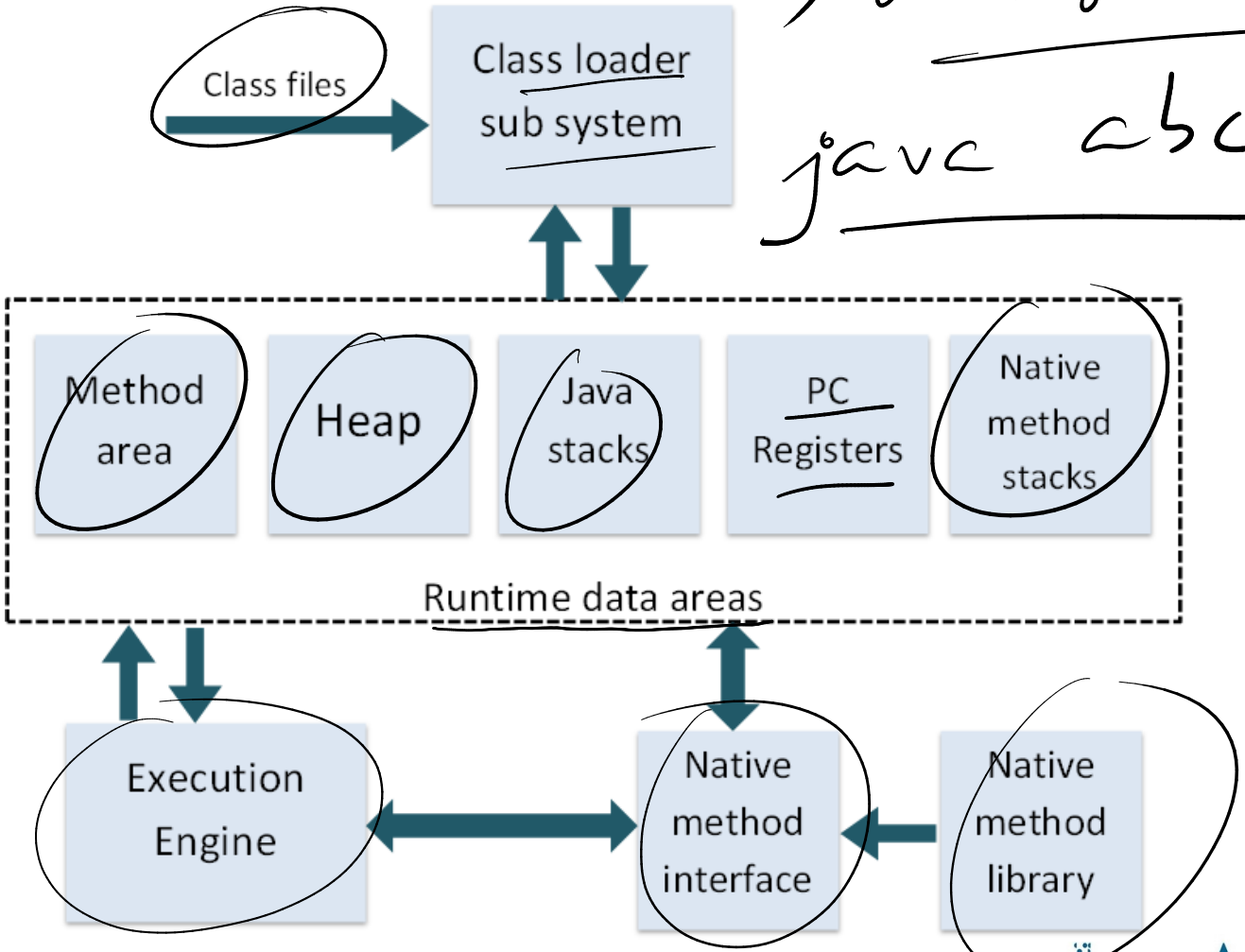
- The architecture
  - Stack machine! Closer to modern high-level languages than the von Neumann machine (Register machines).
  - Memory: 32-bit words (=4 bytes)
  - Instructions: 226 in total, variable length, 1-5 bytes
  - Program: byte stream
  - Data: stored in words
  - Program Counter (PC) contains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

# The JVM Architecture

$> javac abc.java

→ abc.class

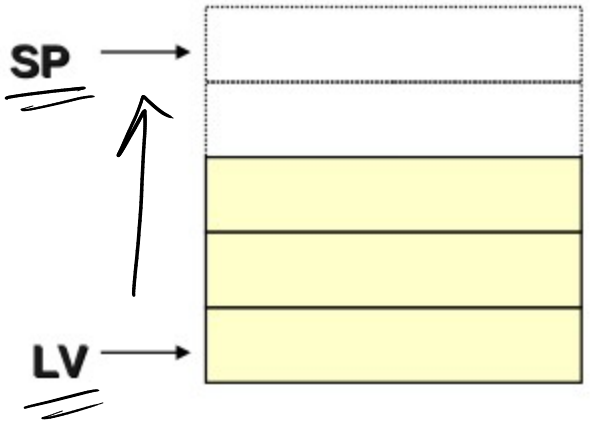
java abc



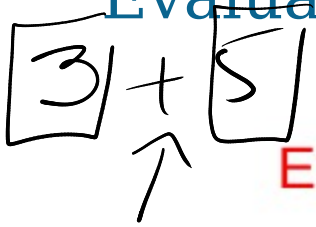

http://www.santhoshreddymandadi.com/java/java-virtual-machine-jvm-architecture.html

# Stack Machines

- **Stack**
  - Area of memory, extends upwards or shrinks downwards
  - LV (Local Variable), base of stack
  - SP (Stack Pointer), top of stack
- **Operations**
  - push on top (increment SP)
  - pop (decrement SP)
  - add top two arguments on the stack, replace with result
  - More details: https://en.wikipedia.org/wiki/Stack_machine



# Evaluating Expressions on Stack



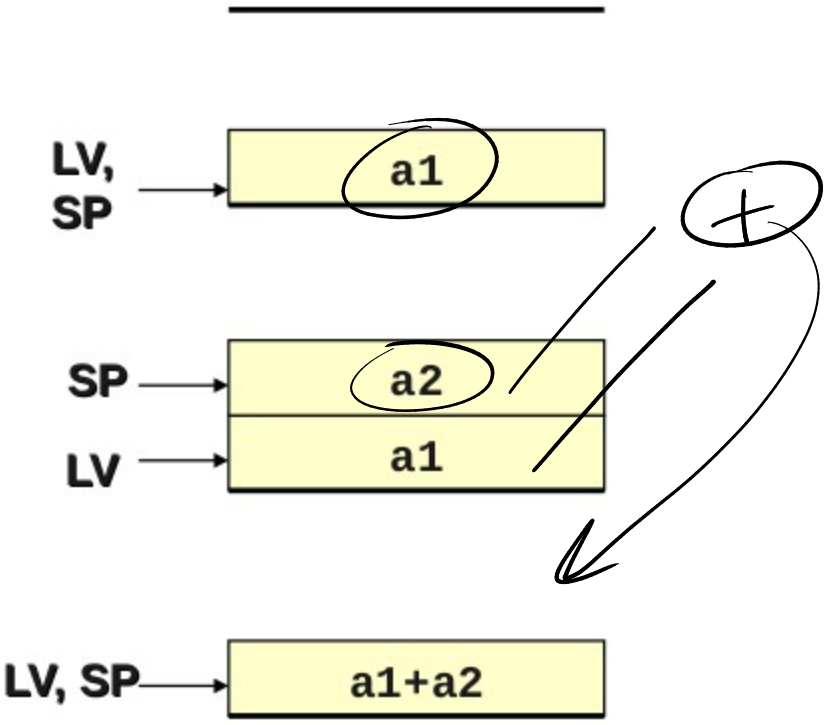

Evaluate

**a1+a2**

**PUSH a1**

**PUSH a2**

**ADD**



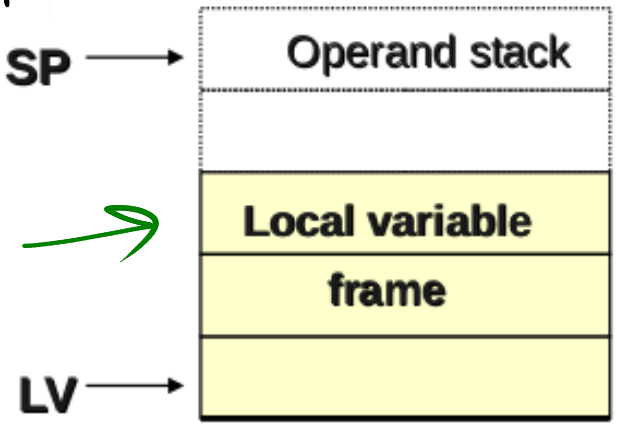
3 5 + 4 *

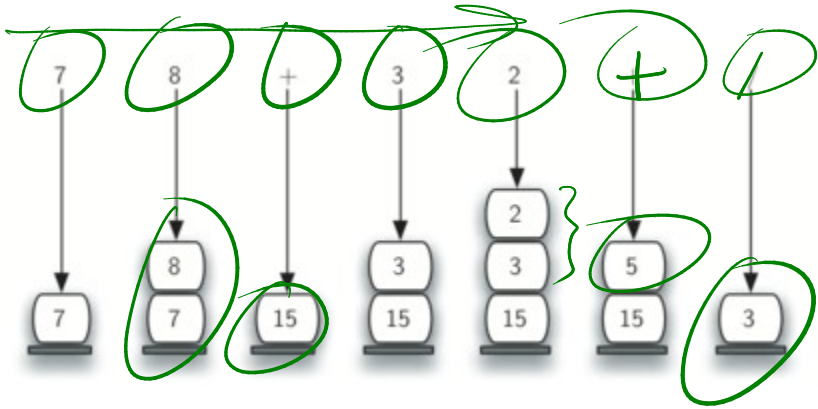
(3+5)*4 = 32

3+5*4 = 23

# 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 RPN & Infix, Prefix & Postfix Expressions)
- Direct Support for
  - Local variables for methods (stored at the base of stack, deleted when the method exits)
  - (recursive) method calls: to store return address

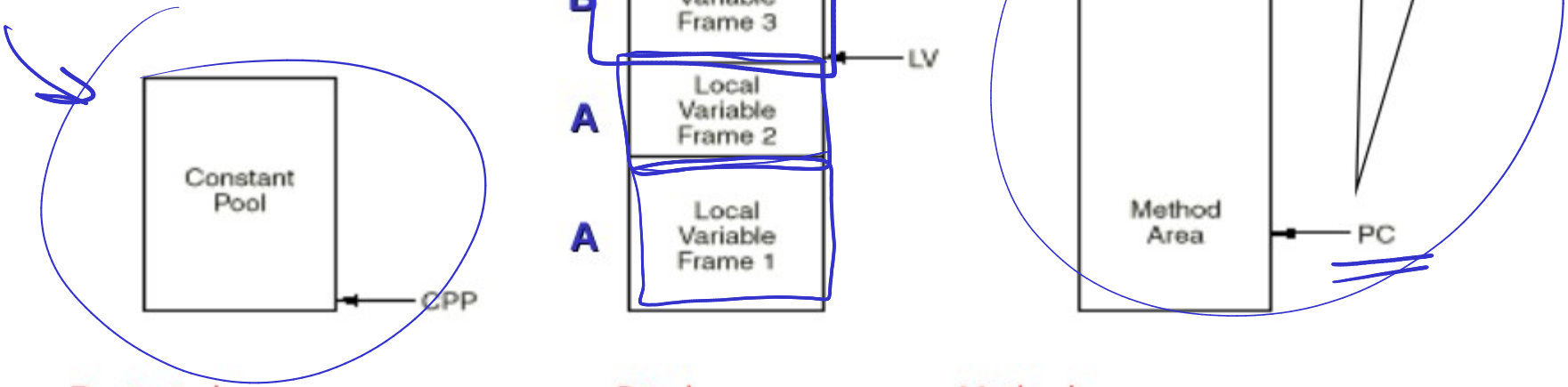



**RPN Example:** 7 8 + 3 2 + /

# 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
- Invoking a method / return from a method
  - **INVOKEVIRTUAL, RETURN**

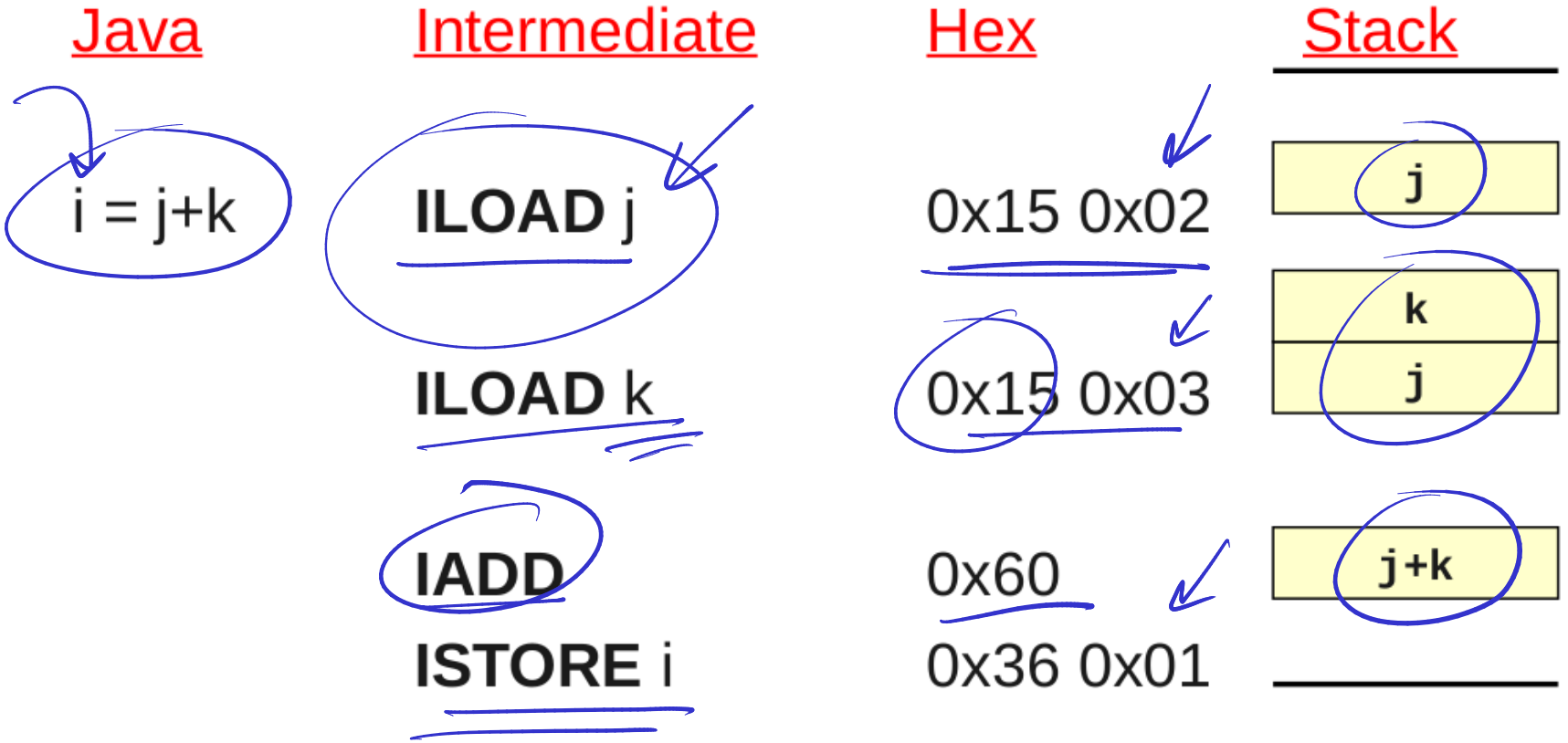
# IJVM Instruction Set

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_ICMPEQ *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

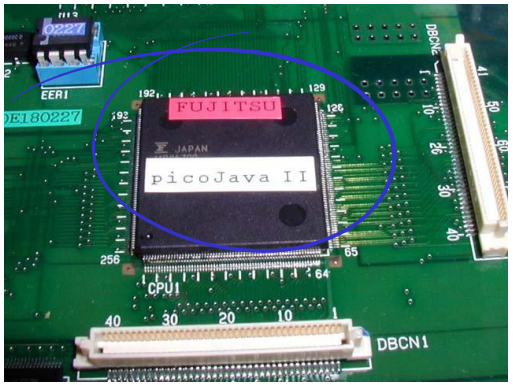


# JVM Instruction Summary

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

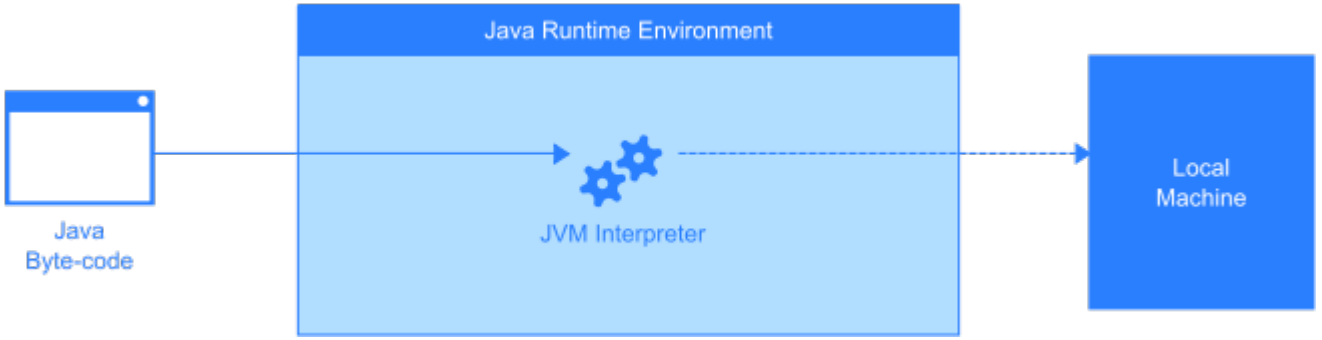
# Interpreting JVM

- 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
  - More details: https://en.wikipedia.org/wiki/PicoJava



# Just In Time (JIT) Compilation

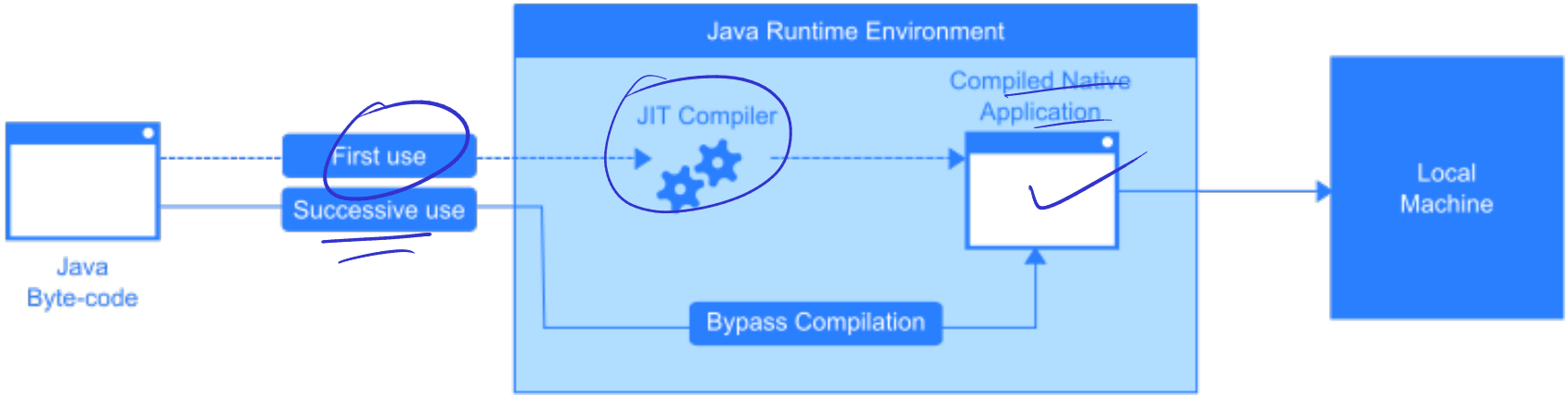
- Why not compile directly to target architecture?
  - more expensive - many varying architectures
  - more time needed to compile each instruction
- But
  - execution is slower with an interpreter!!!
  - instructions may have to be parsed repeatedly




Source:
https://en.wikibooks.org/wiki/Java_Programming/The_Java_Platform

# Just In Time (JIT) Compilation

- Just In Time (JIT) Compilation
  - include Java compiler to target machine within a browser
  - compile instructions, and reuse them
  - longer wait till arrival of executable code




Source:
https://en.wikibooks.org/wiki/Java_Programming/The_Java_Platform

# Summary

- Compilation vs. Interpretation
- 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/remote architectures (applets)
  - has **JVM**, a virtual stack machine
  - interpreted via a C language interpreter, or a hardware chip (picoJava II for embedded Java applications)