Languages and Compilers (SProg og Oversættere)

Bent Thomsen

Department of Computer Science

Aalborg University

With acknowledgement to Norm Hutchinson whose slides this lecture is based on.

The JVM

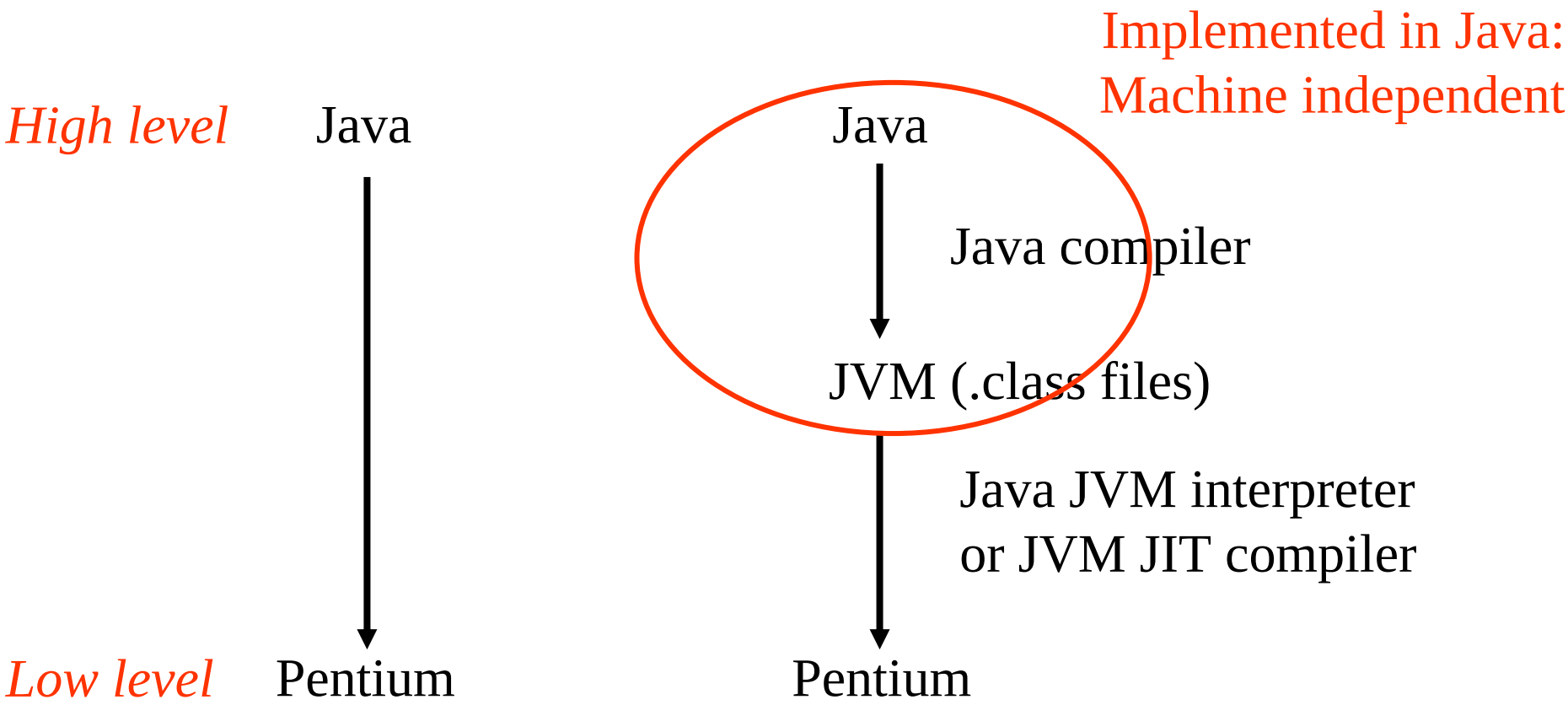
In this lecture we look at the JVM as an example of a real-world runtime system for a modern object-oriented programming language.

The material in this lecture is interesting because:

1) it will help understand some things about the JVM
2) JVM is probably the most common and widely used VM in the world.
3) You'll get a better idea what a real VM looks like.

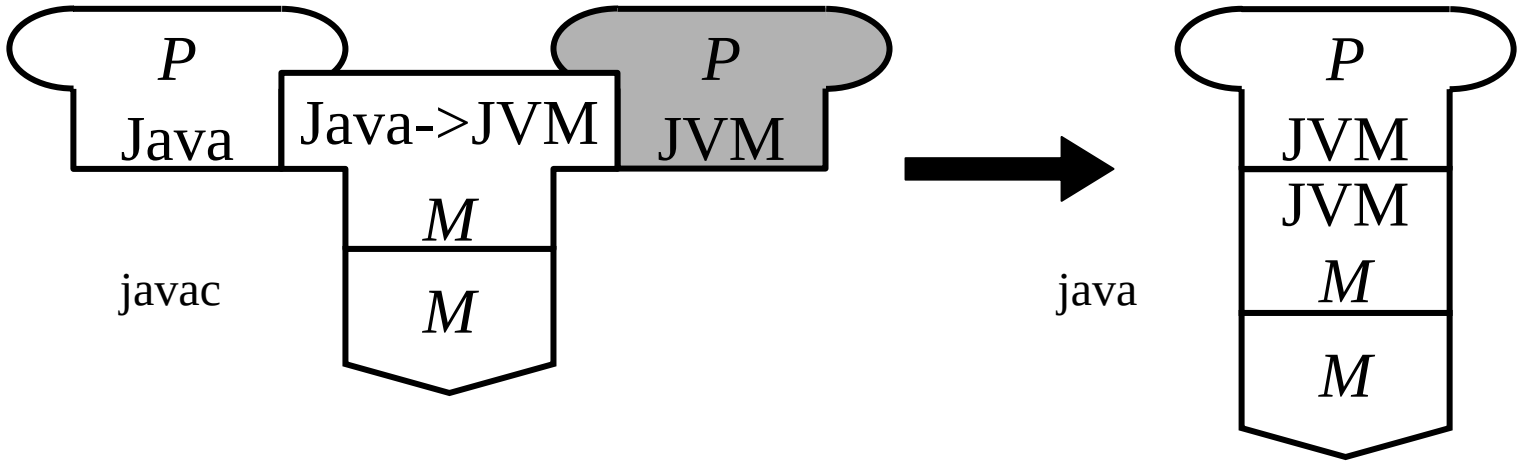
Abstract Machines

Abstract machine implements an intermediate language "in between" the high-level language (e.g. Java) and the low-level hardware (e.g. Pentium)

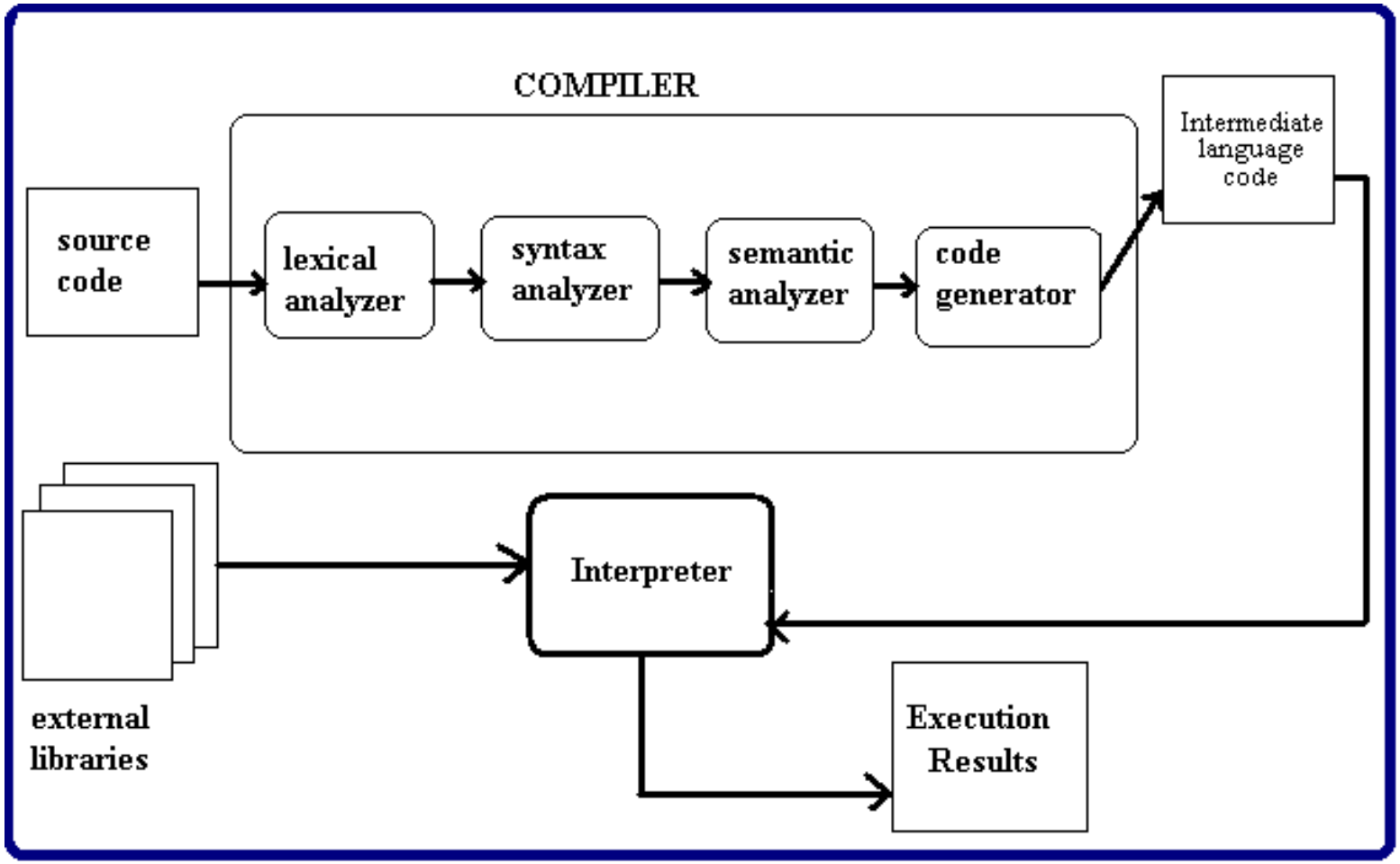


Interpretive Compilers

Remember: our “Java Development Kit” to run a Java program *P*



Hybrid compiler / interpreter



Abstract Machines

An abstract machine is intended specifically as a runtime system for a particular (kind of) programming language.

- JVM is a virtual machine for Java programs:

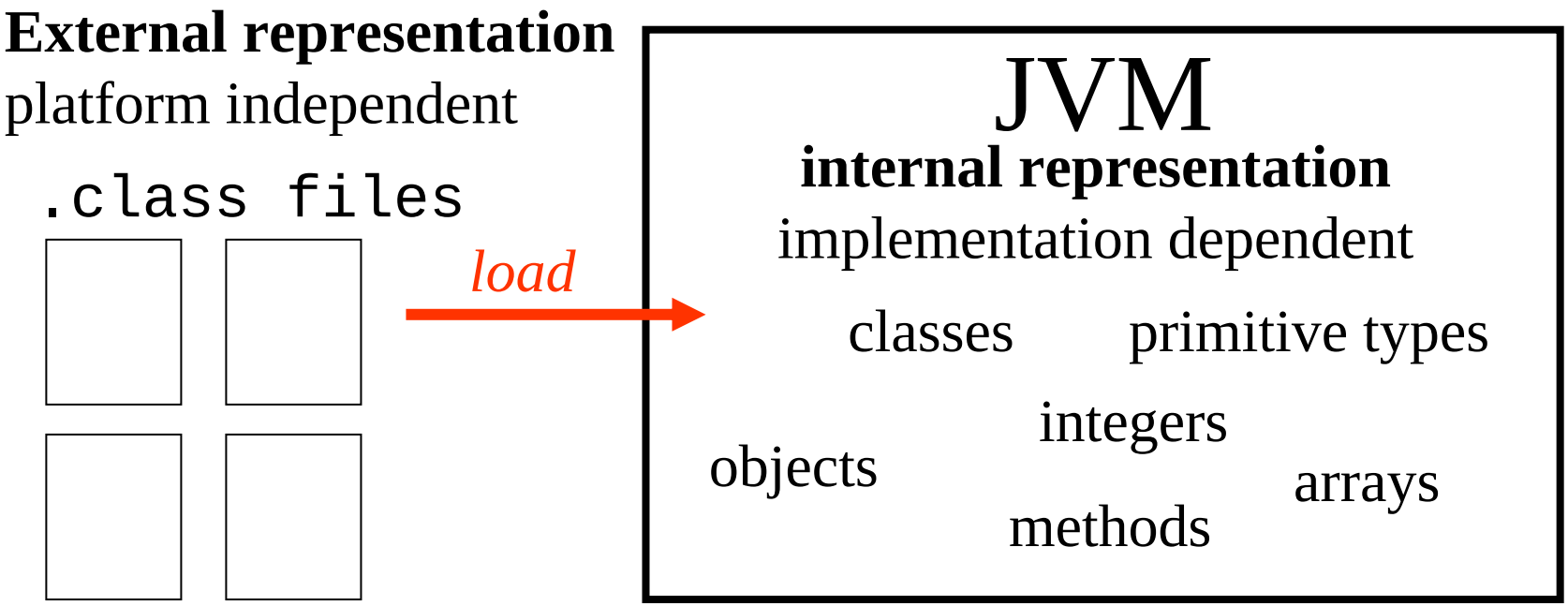
- It directly supports object oriented concepts such as classes, objects, methods, method invocation etc.

- easy to compile Java to JVM

=> 1) easy to implement compiler
2) fast compilation

- another advantage: portability

Class Files and Class File Format



The **JVM is** an **abstract** machine in the true sense of the word.

The JVM spec. does not specify implementation details (can be dependent on target OS/platform, performance requirements etc.)

The JVM spec defines a machine independent "**class file format**" that all JVM implementations must support.

Class File

- Table of constants.
- Tables describing the class
  - name, superclass, interfaces
  - attributes, constructor
- Tables describing fields and methods
  - name, type/signature
  - attributes (private, public, etc)
- The code for methods.

```
ClassFile {
      u4 magic;
      u2 minor_version;
      u2 major_version;
      u2 constant_pool_count;
      cp_info constant_pool[constant_pool_count-1];
      u2 access_flags;
      u2 this_class;
      u2 super_class;
      u2 interfaces_count;
      u2 interfaces[interfaces_count];
      u2 fields_count;
      field_info fields[fields_count];
      u2 methods_count;
      method_info methods[methods_count];
      u2 attributes_count;
      attribute_info attributes[attributes_count];
   }
```

Data Types

JVM (and Java) distinguishes between two kinds of types:

Primitive types:

- boolean: `boolean`
- numeric integral: `byte, short, int, long, char`
- numeric floating point: `float, double`
- internal, for exception handling: `returnAddress`

Reference types:

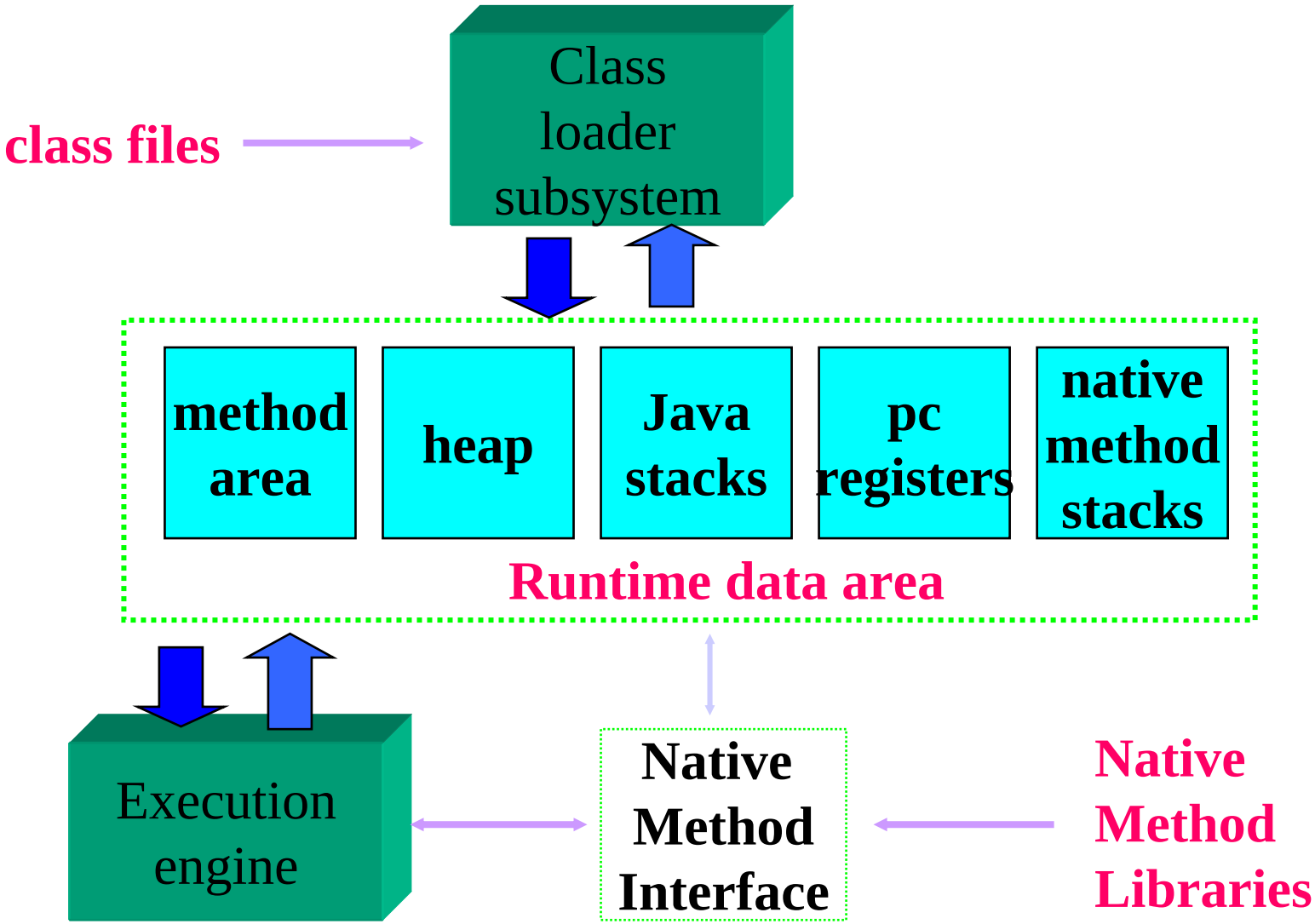
- class types
- array types
- interface types

Note: Primitive types are represented directly, reference types are represented indirectly (as pointers to array or class instances)

Data Types: Some additional remarks

- Return Address Type
  - Used by the JVM instructions
    - jsr (jump to subroutine)
    - jsr_w (wide jump to subroutine)
    - ret (return from subroutine)
- The boolean Type
  - Very limited support for boolean type in JVM
    - Java´s boolean type is compiled to int type
    - Coding: true = 1, false = 0
    - Explicit support for boolean arrays implemented as byte-arrays
- Floating Point Types
  - No Exceptions (signaling conditions acc. to IEEE 754)
  - Positive and negative zero, positive and negative infinity special value NaN (not a number, comparision always yields false)

Internal Architecture of JVM

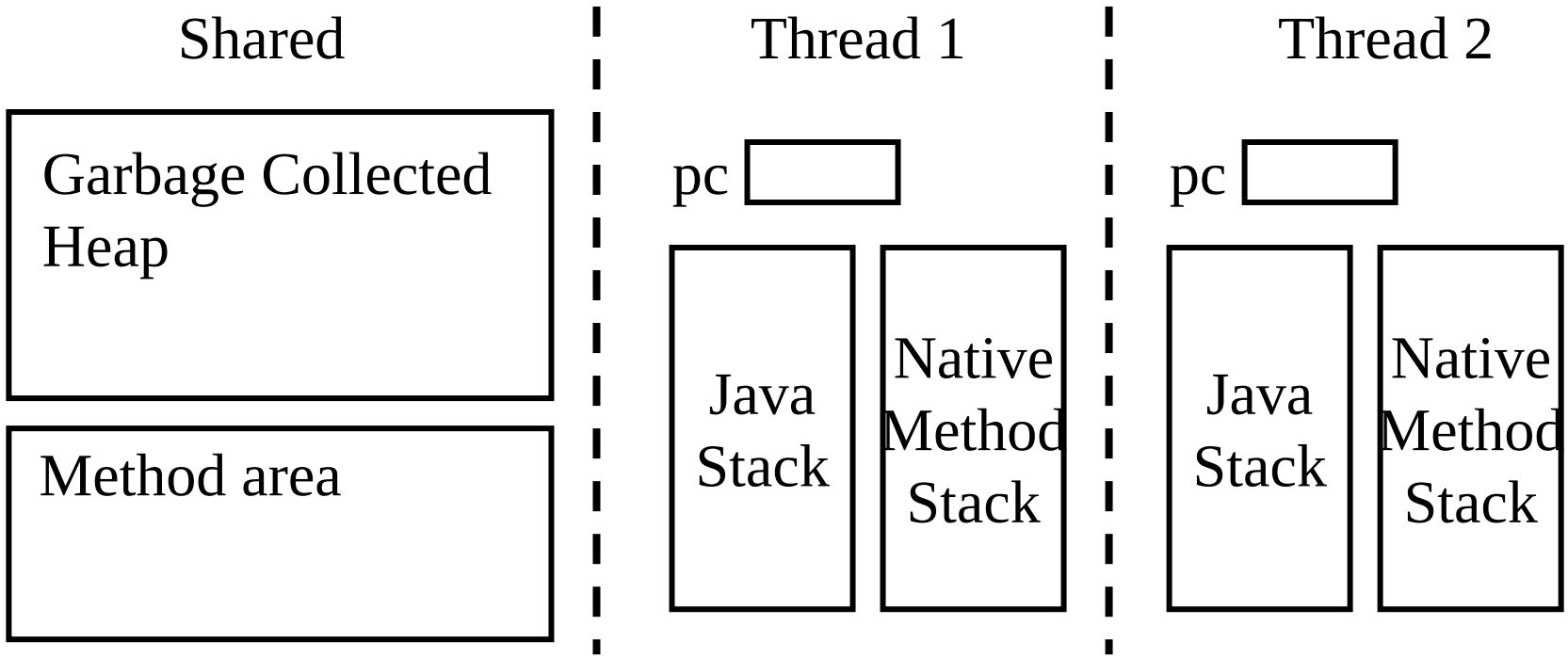


JVM: Runtime Data Areas

Besides OO concepts, JVM also supports multi-threading. Threads are directly supported by the JVM.

=> Two kinds of runtime data areas:

1) shared between all threads
2) private to a single thread



Java Stacks

JVM is a stack based machine, much like TAM.
JVM instructions

- implicitly take arguments from the stack top
- put their result on the top of the stack

The stack is used to

- pass arguments to methods
- return result from a method
- store intermediate results in evaluating expressions
- store local variables

JVM Interpreter

The core of a JVM interpreter is basically this:

```
do {
    byte opcode = fetch an opcode;
    switch (opcode) {
      case opCode1 :
          fetch operands for opCode1;
          execute action for opCode1;
          break;
      case opCode2 :
          fetch operands for opCode2;
          execute action for opCode2;
          break;
      case ...
} while (more to do)
```

Instruction-set: typed instructions!

JVM instructions are explicitly typed: different opCodes for instructions for integers, floats, arrays and reference types.

This is reflected by a naming convention in the first letter of the opCode mnemonics:

Example: different types of “load” instructions

`iload`	integer load
`lload`	long load
`fload`	float load
`dload`	double load
`aload`	reference-type load

Instruction set: kinds of operands

JVM instructions have three kinds of operands:

- from the top of the operand stack
- from the bytes following the opCode
- part of the opCode

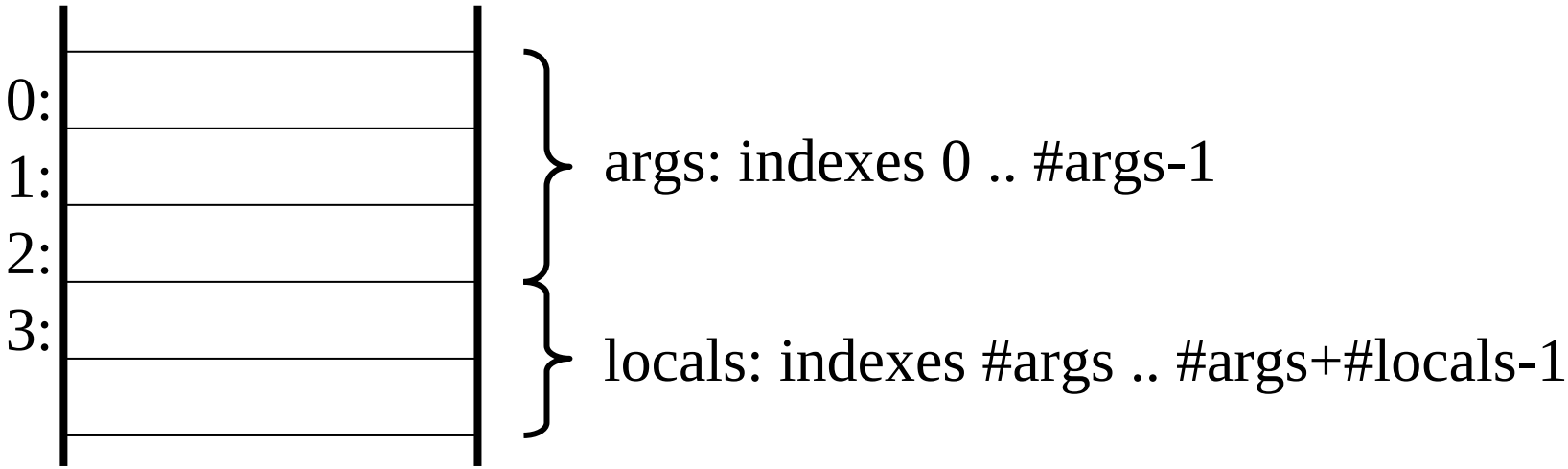
One instructions may have different “forms” supporting different kinds of operands.

Example: different forms of “iload”.

Assembly code	Binary instruction code layout		
`iload_0`	26		
`iload_1`	27		
`iload_2`	28		
`iload_3`	29		
`iload` *n*	21	*n*	
`wide iload` *n*	196	21	*n*

Instruction-set: accessing arguments and locals

arguments and locals area inside a stack frame



Instruction examples:

iload_1	istore_1
iload_3	astore_1
aload 5	fstore_3
aload_0	

- A load instruction: loads something from the args/locals area to the top of the operand stack.
- A store instruction takes something from the top of the operand stack and stores it in the argument/local area

Instruction-set: non-local memory access

In the JVM, the contents of different “kinds” of memory can be accessed by different kinds of instructions.

accessing locals and arguments: `load` and `store` instructions

accessing fields in objects: `getfield, putfield`

accessing static fields: `getstatic, putstatic`

Note: static fields are a lot like global variables. They are allocated in the “method area” where also code for methods and representations for classes are stored.

Q: what memory area are getfield and putfield accessing?

Instruction-set: operations on numbers

Arithmethic

add: `iadd, ladd, fadd, dadd`
subtract: `isub, lsub, fsub, dsub`
multiply: `imul, lmul, fmul, dmul`

etc.

Conversion

```
i2l, i2f, i2d
l2f, l2d, f2s

f2i, d2i, …
```

Instruction-set …

Operand stack manipulation

`pop, pop2, dup, dup2, dup_x1, swap, …`

Control transfer

Unconditional : `goto, goto_w, jsr, ret, …`

Conditional: `ifeq, iflt, ifgt, …`

Instruction-set …

Method invocation:

`invokevirtual`: usual instruction for calling a method on an object.

`invokeinterface`: same as invokevirtual, but used when the called method is declared in an interface. (requires different kind of method lookup)

`invokespecial`: for calling things such as constructors. These are not dynamically dispatched (this instruction is also known as `invokenonvirtual`)

`invokestatic`: for calling methods that have the “static” modifier (these methods “belong” to a class, rather an object)

Returning from methods:

`return, ireturn, lreturn, areturn, freturn, …`

Instruction-set: Heap Memory Allocation

Create new class instance (object):

`new`

Create new array:

`newarray:` for creating arrays of primitive types.

`anewarray, multianewarray:` for arrays of reference types

Example

As an example on the JVM, we will take a look at the compiled code of the following simple Java class declaration.

```
class Factorial {

    int fac(int n) {
        int result = 1;
        for (int i=2; i<n; i++) {
            result = result * i;
        }
        return result;
    }
}
```

Compiling and Disassembling

```
% javac Factorial.java
% javap -c -verbose Factorial
Compiled from Factorial.java
public class Factorial extends java.lang.Object {
    public Factorial();
        /* Stack=1, Locals=1, Args_size=1 */
    public int fac(int);
        /* Stack=2, Locals=4, Args_size=2 */
}

Method Factorial()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return
```

Compiling and Disassembling ...

```
                    // address: 0    1 2      3
Method int fac(int) // stack: this n result i
  0 iconst_1        // stack: this n result i 1
  1 istore_2        // stack: this n result i
  2 iconst_2        // stack: this n result i 2
  3 istore_3        // stack: this n result i
  4 goto 14
  7 iload_2         // stack: this n result i result
  8 iload_3         // stack: this n result i result i
  9 imul            // stack: this n result i result i
  10 istore_2
  11 iinc 3 1
  14 iload_3        // stack: this n result i i
  15 iload_1        // stack: this n result i i n
  16 if_icmple 7    // stack: this n result i
  19 iload_2        // stack: this n result i result
  20 ireturn
```

JASMIN

- JASMIN is an assembler for the JVM
  - Takes an ASCII description of a Java classes
  - Input written in a simple assembler like syntax
    - Using the JVM instruction set
  - Outputs binary class file
  - Suitable for loading by the JVM

- Running JASMIN
  - `jasmin myfile.j`
- Produces a .class file with the name specified by the .class directive in myfile.j

Writing Factorial in “jasmin”

```
.class package Factorial
.super java/lang/Object

.method package <init>()V
.limit stack 50
.limit locals 1
aload_0
invokenonvirtual java/lang/Object/<init>()V
return
.end method
```

Writing Factorial in “jasmin”

```
.method package fac(I)I
.limit stack 50
.limit locals 4
iconst_1
istore 2
iconst_2
istore 3
Label_1:
iload 3
iload 1
if_icmplt Label_4
iconst_0
goto Label_5
Label_4:
iconst_1
Label_5:
ifeq Label_2
iload 2
iload 3
imul
dup
istore 2
pop
Label_3:
iload 3
dup
iconst_1
iadd
istore 3
pop
goto Label_1
Label_2:
iload 2
ireturn
iconst_0
ireturn
.end method
```

Another example: out.j

```
.class public out
.super java/lang/Object

.method public <init>()V
        aload_0
        invokespecial java/lang/Object/<init>()V
        return
.end method

.method public static main([Ljava/lang/String;)V
        .limit stack 2

        getstatic java/lang/System/out Ljava/io/PrintStream;
        ldc “Hello World”
        invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

        return
.end method
```

The result: out.class

```
00000000h: CA FE BA BE 00 03 00 2D 00 1B 0C 00 17 00 1A 01 ; 栲瑣...-........
00000010h: 00 16 28 5B 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 ; ..([Ljava/lang/S
00000020h: 74 72 69 6E 67 3B 29 56 01 00 10 6A 61 76 61 2F ; tring;)V...java/
00000030h: 6C 61 6E 67 2F 4F 62 6A 65 63 74 01 00 06 3C 69 ; lang/Object...<i
00000040h: 6E 69 74 3E 07 00 03 0C 00 04 00 08 07 00 10 01 ; nit>............
00000050h: 00 03 28 29 56 07 00 13 01 00 04 43 6F 64 65 01 ; ..()V......Code.
00000060h: 00 04 6D 61 69 6E 09 00 09 00 01 01 00 0A 53 6F ; ..main........So
00000070h: 75 72 63 65 46 69 6C 65 01 00 05 6F 75 74 2E 6A ; urceFile...out.j
00000080h: 0C 00 11 00 19 01 00 13 6A 61 76 61 2F 69 6F 2F ; ........java/io/
00000090h: 50 72 69 6E 74 53 74 72 65 61 6D 01 00 07 70 72 ; PrintStream...pr
000000a0h: 69 6E 74 6C 6E 0A 00 05 00 06 01 00 10 6A 61 76 ; intln........jav
000000b0h: 61 2F 6C 61 6E 67 2F 53 79 73 74 65 6D 01 00 0B ; a/lang/System...
000000c0h: 48 65 6C 6C 6F 20 57 6F 72 6C 64 0A 00 07 00 0F ; Hello World.....
000000d0h: 08 00 14 01 00 03 6F 75 74 07 00 17 01 00 15 28 ; ......out......(
000000e0h: 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 74 72 69 6E ; Ljava/lang/Strin
000000f0h: 67 3B 29 56 01 00 15 4C 6A 61 76 61 2F 69 6F 2F ; g;)V...Ljava/io/
00000100h: 50 72 69 6E 74 53 74 72 65 61 6D 3B 00 21 00 18 ; PrintStream;.!..
00000110h: 00 05 00 00 00 00 00 02 00 01 00 04 00 08 00 01 ; ................
00000120h: 00 0A 00 00 00 11 00 01 00 01 00 00 00 05 2A B7 ; ..............*?
00000130h: 00 12 B1 00 00 00 00 00 09 00 0B 00 02 00 01 00 ; ..?.............
00000140h: 0A 00 00 00 15 00 02 00 01 00 00 00 09 B2 00 0C ; .............?.
00000150h: 12 16 B6 00 15 B1 00 00 00 00 00 01 00 0D 00 00 ; ..?.?.........
00000160h: 00 02 00 0E                                     ; ....
```

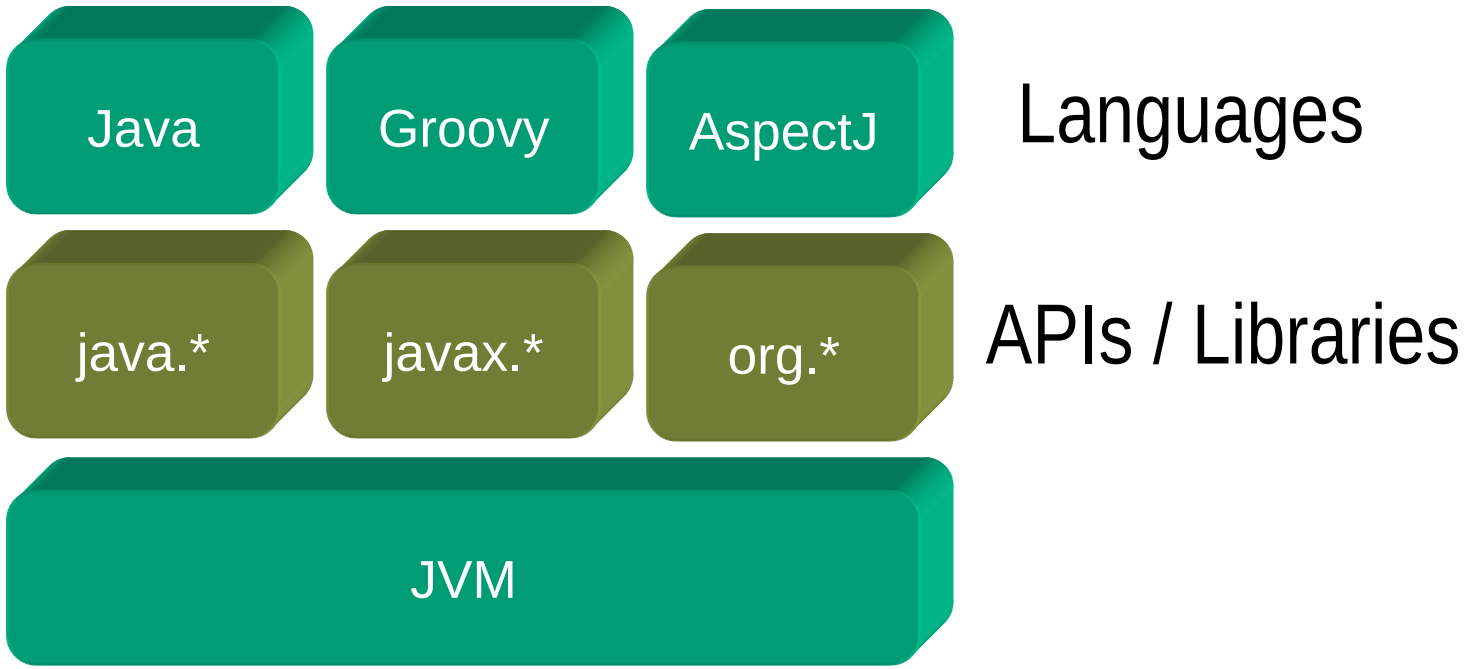
Jasmin file format

- Directives
  - .catch . Class .end .field .implements .interface .limit .line
  - .method .source .super .throws .var
- Instructions
  - JVM instructions: ldc, iinc bipush
- Labels
  - Any name followed by : - e.g. Foo:
  - Cannot start with = : . *
  - Labels can only be used within method definitions

The JVM as a target for different languages

When we talk about Java what do we mean?

- "Java" isn't just a language, *it is a platform*
- *The list of languages targeting the JVM is very long!*
  - *Languages for the Java VM*



Reusability

- Java has a lot of great APIs and libraries
  - Core libraries (java[x].*)
  - Open source libraries
  - Third party commercial libraries

- What is it that we are *reusing* when we use these tools?
  - We are reusing the bytecode
  - We are reusing the fact that the JVM has a nice spec

- This means that we can innovate on top of this binary class file nonsense ☺

Not just one JVM, but a whole family

- JVM (J2EE & J2SE)
  - SUN Classis, SUN HotSpots, IBM, BEA, …
- CVM, KVM (J2ME)
  - Small devices.
  - Reduces some VM features to fit resource-constrained devices.
- JCVM (Java Card)
  - Smart cards.
  - It has least VM features.

- And there are also lots of other JVMs

Java Platform & VM & Devices

Java Technology Targets a Broad Range of Devices

Hardware implementations of the JVM

Figure 6. aJ-100 package (larger than actual size).