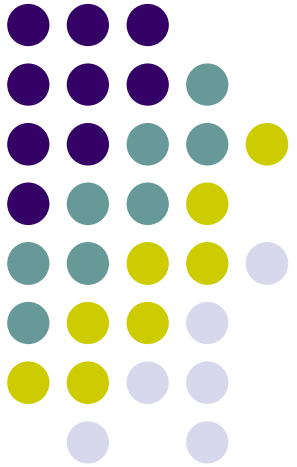
Looking into Java .class files

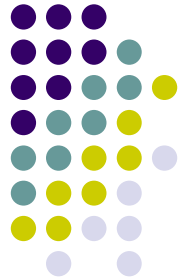
Ando Saabas

Theory days, Voore

30.09.2006

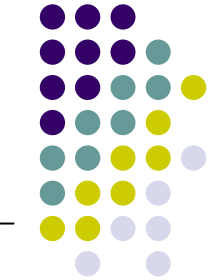


Why understand the JVM and bytecode?



- Needed when writing a...
  - Compiler
  - Bytecode verifier
  - Optimizer
  - Decompiler
  - Obfuscator
  - ....
- Important when debugging, memory usage and performance tuning
- Lots for theoretical computer scientists to do research about

What is Java Virtual Machine?

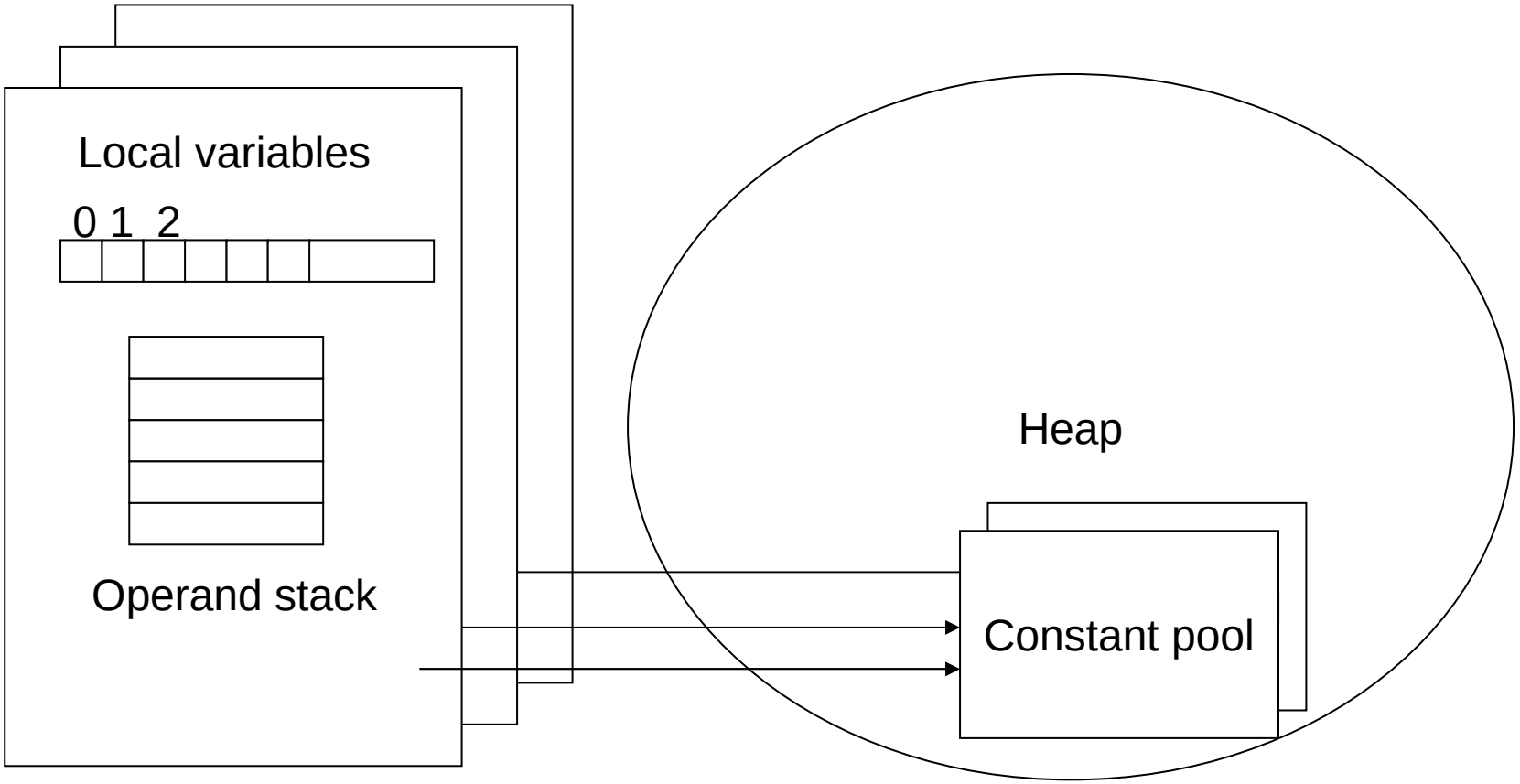


- A virtual machine that executes Java bytecode
- It is a stack based machine, which runs programs that are in a standardized portable binary format (class files)
- Contains a
  - Interpreter/just-in-time compiler which executes the bytecode
  - Bytecode verifier, which verifies all class files before they are executed

How does a JVM work?

- Each thread has a JVM stack which stores *frames*
- A frame is created each time a method is invoked. It consists of
  - an operand stack
  - an array of local variables,
  - reference to the runtime constant pool of the class of the method
- When a method completes, a frame is destroyed

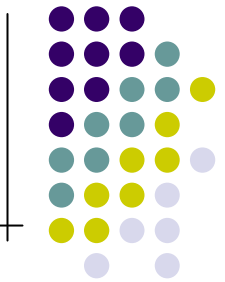
Frame stack



Stack and local variable array

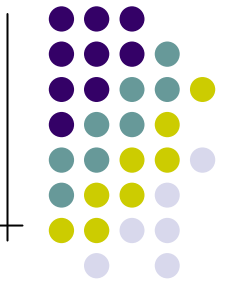
- The array of local variables contains the parameters of the method and the local variables
  - The first variable is a reference to *this*
  - After that, the n parameters of the method are stored (variables 1..n)
  - The size of the array is determined at compiletime
- The stack is a LIFO stack used to push and pop values.
  - Also used to receive return values from methods
  - The size of the stack is determined at compiletime

Heap and method area



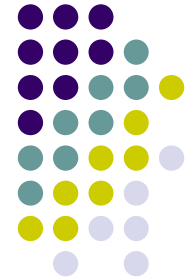
- The heap is shared among all virtual machine threads
  - Its the data area from which memory for all class instances and arrays are located
  - Objects are never explicitly deallocated, but storage is reclaimed by an automatic storage management system.
- Method area is logically part of the heap. In it, per-class structures like runtime constant pool, field and method data resides.

Types in JVM



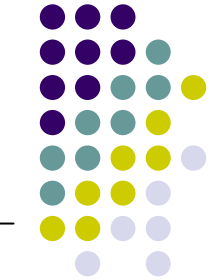
- JVM operates on primitive and reference types
  - byte, 8-bit signed integers
  - short, 6-bit signed integers
  - int, 32-bit signed integers
  - long, 64-bit signed integers
  - char, 16-bit unsigned integers representing Unicode characters
  - float, 32-bit IEEE 754 floating-point numbers
  - double, 64-bit IEEE 754 floating-point numbers
  - returnAddress types
  - (no boolean types)
- class types, interface types, and array types

Instruction set summary



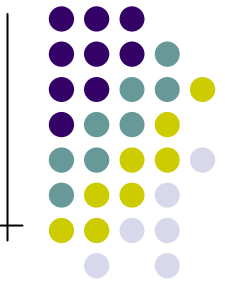
- Instruction consists of a 1 byte opcode, followed by 0 or more operands
- Most instructions encode type information about the operations they perform
  - *Example: iload* loads a value which must be an integer from the local variable on top of the stack

Kinds of instructions



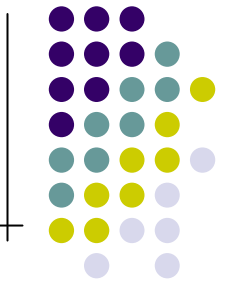
- Load and store instructions (iload, astore, ...)
  - Transfer values between local variables and the operand stack
- Arithmetic instructions (iadd, fdiv,…)
  - Apply a function on values in the operand stack, pop the values, push back the result
- Type conversion instructions (between numeric types: i2d, f2l, ...)
- Object creation and manipulation (new, newarray, getfield, putstatic,...)

Kinds of instructions



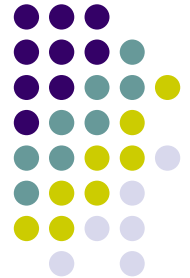
- Operand stack management instructions (pop, dup, swap,...).
- Control transfer instructions (goto, ifeq, ifgt, *if_acmpne*, ...)
- Method invocation and return instructions (invokevirtual, invokestatic, areturn, return,...)
- Throwing exceptions (athrow)
- Implementing finally (jsr, ret, ...)
- Synchronization instructions

Class file

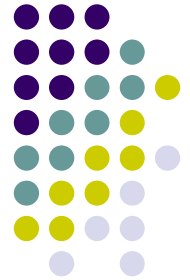


- Includes general information about the class (access flags, class and superclass name etc.) and 5 main components
  - Constant pool
  - Interfaces
  - Fields
  - Methods
  - Attributes

Constant pool

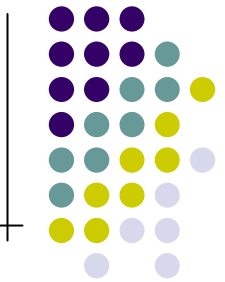


- Contains all constants needed in the class, from numeric and string literals known at compiletime to method and field references that must be resolved at runtime.
- Instructions refer to symbolic information in the constant pool



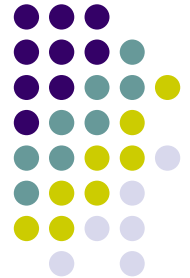
- CONSTANT_Class
- CONSTANT_Fieldref
- CONSTANT_Methodref
- CONSTANT_InterfaceMethodref
- CONSTANT_String
- CONSTANT_Integer
- CONSTANT_Float
- CONSTANT_Long
- CONSTANT_Double
- CONSTANT_NameAndType
- CONSTANT_Utf8

Interfaces and fields



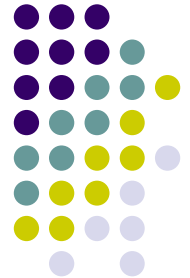
- Interfaces
  - An array of indexes into the constant pool table containing a class_info constant
- Fields
  - Defined by its access flags, name and descriptor index, and a number of attributes (e.g. the ConstantValue attribute).

Methods



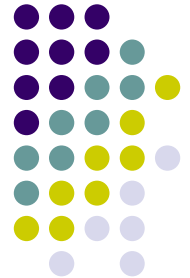
- A method is described by its access flags, name and descriptor index and an array of attributes
- Some method attributes:
  - Code – the bytecode of the method
  - Exceptions
  - Line number table (for debugging)
  - Local variable table (for debugging)

Important method attributes



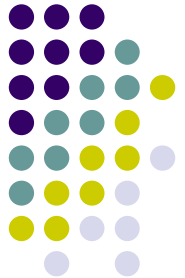
- Code attribute
  - Max stack size
  - Max number of locals
  - Code length
  - Code
- Exception table
  - Start pc
  - End pc
  - Handler pc
  - Catch type

Sample bytecode



```
class Circle {
   int radius;
   double getArea() {
        return radius * radius * Math.PI;
   }
}
 0 aload_0
 1 getfield #2 <Test4/radius I>
 4 aload_0
 5 getfield #2 <Test4/radius I>
 8 imul
 9 i2d
10 ldc2_w #3 <3.141592653589793>
13 dmul
14 dreturn
```

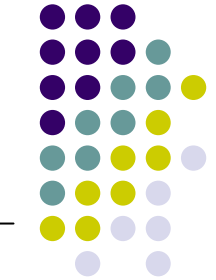
Sample bytecode



```
int divide(int x, int y) {
   try {
     return x/y;
   } catch (ArithmeticException ae){
     ae.printStackTrace();
     return 0;
   }
}
 0 iload_1
 1 iload_2
 2 idiv
 3 ireturn
 4 astore_3
 5 aload_3
 6 invokevirtual #14
   <java/lang/ArithmeticException/printStackTrace()V>
 9 iconst_0
10 ireturn
```

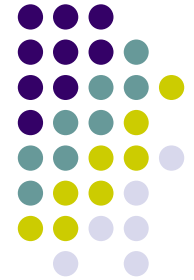
0	3	4	#13

Why view/edit bytecode?



- To understand how a JVM or compiler works
- To test a
  - Verifier
  - Decompiler
  - Optimizer
  - ....
- To hand-optimize code

Java Bytecode Editor



- Built on top of the open source jclasslib bytecode viewer by ej-technologies
- Makes use of the Apache's bytecode engineering library
- Allows adding/removing constants, fields, interfaces, methods, exceptions and editing method code
- Integrates the JustIce bytecode verifier
- Can be downloaded from http://cs.ioc.ee/~ando/jbe/