Formal Methods for Java

Lecture 24: Excursion: JVM

Jochen Hoenicke

Software Engineering
Albert-Ludwigs-University Freiburg

Jan 27, 2012

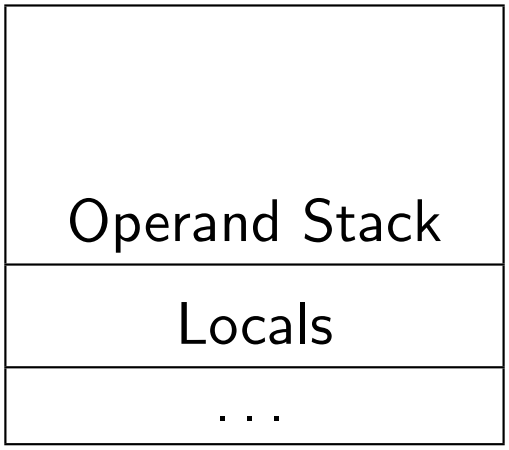
JVM Basics

- JVM interprets .class files
- .class files contain
  - a description of classes (name, fields, methods, inheritance relationships, referenced classes, . . .)
  - a description of fields (name, type, attributes (visibility, `volatile`, `transient`, . . .))
  - bytecode for the methods
- Stack machine
- Integer stack
- Typed instructions
- Bytecode verifier to ensure type safety

Calling Methods

Activation Frame contains:

- Variables local to the called method
- Stack space for instruction execution (Operand Stack)



One activation frame per method call: $x.foo()$

1. pushes new activation frame
2. calls the method foo
3. pops the activation frame

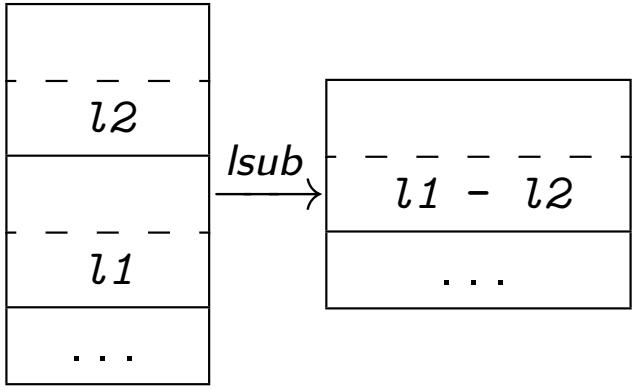
Executing Instructions

- Arguments are on the operand stack
  ➥ Some instructions move local variables or constants to the stack
- Most instructions pop topmost arguments from the stack and push result onto the stack

Example: `lsub`

Subtract two `long` values *l1* and *l2*.

```
long l2 = popLong();
long l1 = popLong();
push(l1 - l2);
```



JVM Instructions

- Most instructions are typed,
- but internally, only `int`, `long`, `float`, and `double` matter.
- Other types only used by bytecode verifier
- Instructions can be grouped

Instruction Group "Load Instructions"

- `tload` where $t \in \{a, i, l, f, d\}$
  Stores local variable on the operand stack
- `taload` where $t \in \{a, b, s, i, l, f, d\}$
  Stores element of an array on the operand stack
- `aconst_null`
  Stores `null` on the operand stack
- `tconst_<n>` where $t \in \{i, l, d\}$
  Stores constant on the operand stack (only limited values possible)
- `bipush`, `sipush`
  Push `byte` resp. `short` constant on the operand stack
- `ldc`
  Load constant from the constant pool

Instruction Group "Store Instructions"

- `tstore` where $t \in \{a, i, l, f, d\}$
  Store top of operand stack into local variable
- `tastore` where $t \in \{a, b, s, i, l, f, d\}$
  Store top of operand stack into array

Instruction Group "Stack Manipulation"

- `pop` and `pop2`
  Remove the topmost (2) elements from the operand stack
- `dup`, ...
  Duplicate the top element(s) of the stack
- `swap`
  Exchange the topmost two elements on the operand stack

Instruction Group "Conversion Instructions"

- `i2t` where $t \in \{b, c, d, f, l, s\}$
  Convert `int`
- `l2t` where $t \in \{d, f, i\}$
  Convert `long`
- `f2t` where $t \in \{d, i, l\}$
  Convert `float`
- `d2t` where $t \in \{f, i, l\}$
  Convert `double`

Instruction Group "Branching Instructions"

- `if_acomp`
  Compare two references and jump on success
- `if_icomp`
  Compare two `int`s and jump on success
- `if`
  Compare against 0 and jump on success
- `tcmp` where $t \in \{f, d\}$
  Compare two floating point numbers (don't jump)
- `ifnonnull`
  Jump if reference is not `null`
- `ifnull`
  Jump if reference is `null`
- `goto`
  Unconditional jump
- `jsr`
  Jump to subroutine

Instruction Group "Switch Instructions"

- `lookupswitch`
  Switch based upon a search in an ordered offset table
- `tableswitch`
  Switch based on index into an offset table

Instruction Group "Return Instructions"

- `treturn` where $t \in \{a, i, l, f, d\}$
  Return a value from a method
- `return`
  Return from a `void` method
- `ret`
  Return from subroutine

Instruction Group "Arithmetic Instructions"

- `tneg` with $t \in \{i, l, f, d\}$
  Negate a number
- `tadd` with $t \in \{i, l, f, d\}$
  Add two numbers
- `tsub` with $t \in \{i, l, f, d\}$
  Subtract two numbers
- `tmul` with $t \in \{i, l, f, d\}$
  Multiply two numbers
- `tdiv` with $t \in \{i, l, f, d\}$
  Divide two numbers
- `trem` with $t \in \{i, l, f, d\}$
  Compute the remainder of a division ($result = value_1 - (value_2 * q)$)
- `iinc`
  Increment integer by constant

Instruction Group "Logic Instructions"

- `tand` where $t \in \{i, l\}$
  Bitwise and
- `tor` where $t \in \{i, l\}$
  Bitwise or
- `txor` where $t \in \{i, l\}$
  Bitwise xor
- `tshr` where $t \in \{i, l\}$
  Logical shift right with sign extension
- `tushr` where $t \in \{i, l\}$
  Logical shift right with zero extension
- `tshl` where $t \in \{i, l\}$
  Logical shift left

Instruction Group "Object Creation Instructions"

- `new`
  Create a new object on the heap
- `newarray`
  Create a new array containing only elements of a primitive type on the heap
- `anewarray`
  Create a new array containing only elements of a reference type on the heap
- `multianewarray`
  Create a new multi-dimensional array on the heap

Instruction Group "Field Access Instructions"

- `getfield`
  Get the value of an instance field
- `getstatic`
  Get the value of a static field
- `putfield`
  Write the value of an instance field
- `putstatic`
  Write the value of a static field

Instruction Group "Method Invocation"

- `invokeinterface`
  Invoke method with polymorphic resolution
- `invokespecial`
  Invoke method without polymorphic resolution
- `invokestatic`
  Invoke a static method
- `invokevirtual`
  Invoke method with polymorphic resolution.

Instruction Group "Monitor Instructions"

- `monitorenter`
  Enter a critical section
- `monitorexit`
  Leave a critical section

Instruction Group "Miscellaneous"

- `arraylength`
  Get the length of an array
- `checkcast`
  Check a cast and throw a *`ClassCastException`* if cast fails
- `instanceof`
  Check if reference points to an instance of the specified class
- `athrow`
  Throw an exception or an error
- `nop`
  Do nothing
- `wide`
  Enable bigger operands