Formal Methods for Java

Lecture 6: The Java Virtual Machine

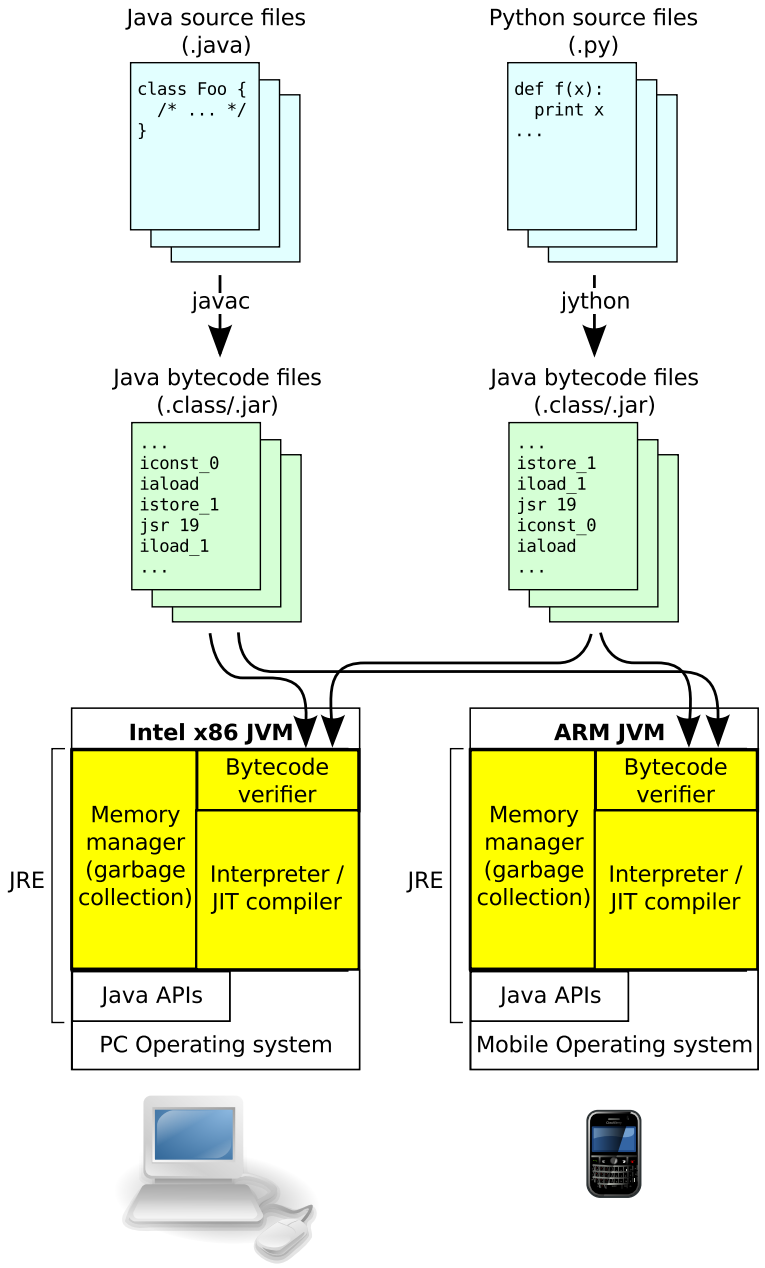
Jochen Hoenicke

Software Engineering
Albert-Ludwigs-University Freiburg

November 9, 2012

Java and the Virtual Machine

- Programs are written in Java or some other language
- Compiler translates this to Java Bytecode.
- Platform-specific Java Virtual Machine executes the code.



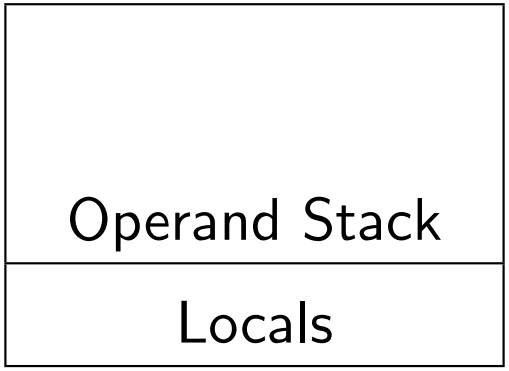
Java Virtual Machine (JVM)

- 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

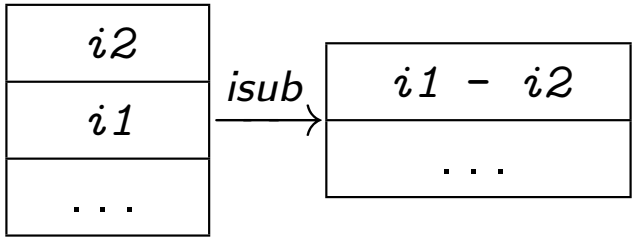
Executing Instructions

- Arguments are on the operand stack
- Most instructions pop the topmost arguments from the stack and push result onto the stack
- Some instructions read/write local variables or object fields.

Example: `isub`

Subtract two `int` values *i1* and *i2*.

```
int i2 = popInt();
int i1 = popInt();
push(i1 - i2);
```



JVM Instructions

- Most instructions are typed,
- JVM bytecode distinguishes between
  `int`, `long`, `float`, `double`, and *`Object`* type
  Indicated by i,l,f,d, and a, respectively.
- Instructions can be grouped

Instruction Group "Local Variable Instructions"

- `aload, iload, lload, fload, dload`
  Stores local variable on the operand stack
- `astore, istore, lstore, fstore, dstore`
  Stores top of operand stack into a local variable
- `iinc`
  Increments a local variable (does not touch the operand stack).

Example

Let x, y be the first and second integer variables.
Then x=y is compiled to the bytecode

```
iload_2
istore_1
```

Instruction Group "Conversion Instructions"

These operations take a value from the operand stack and put the converted value back onto the operand stack.

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

Example

Let x be double and y a byte variable.
Then `y = (byte) x;` is compiled to the bytecode

```
dload_1
d2i
i2b
istore_3
```

On the other hand `x = y;` is compiled to the bytecode

```
iload_3
i2d
dstore_1
```

Instruction Group "Branching Instructions"

- `if_acmpeq, if_acmpne`
  Compare two references and jump on success
- `if_icmpeq, if_icmpgt, if_icmpge, ...`
  Compare two `int`s and jump on success
- `ifeq, ifne, iflt, ...`
  Compare against 0 and jump on success
- t`cmp` where $t \in \{l, f, d\}$
  Compare two long or floating point numbers (don't jump)
- `ifnull, ifnonnull`
  Jump if reference is (not) `null`
- `goto`
  Unconditional jump

Example

The code

```
if (x > y && obj != null)
  x = y;
```

is translated as

```
    iload_1
    iload_2
    if_icmple l1
    aload_3
    if_null   l1
    iload_2
    istore_1
l1:
```

Instruction Group "Switch Instructions"

- `lookupswitch`,`tableswitch`
  Takes an integer operand from the stack.
  Based on its value it jumps to another instruction.

  The instructions only differ in the way they store the value to jump address map.

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

Instruction Group "Arithmetic Instructions"

These operations take one or two values from the operand stack and put the result of the operation onto the operand stack.

- `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 $(result = value_1 - (value_2 * q))$

Instruction Group "Logic Instructions"

These operations take one or two values from the operand stack and put the result of the operation onto the operand stack.

- `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, anewarray, multianewarray`
  Takes a number from the stack and creates a new array containing that many elements.

Instruction Group "Array Instructions"

- t`aload` where $t \in \{a, b, s, i, l, f, d\}$
  Takes the array a and an index i from the operand stack and puts the element $a[i]$ on the operand stack
- t`astore` where $t \in \{a, b, s, i, l, f, d\}$
  Takes the array a, an index i and a value e from the operand stack and stores e into the array $a[i]$.
- `arraylength`
  Takes the array a from the operand stack and puts its length on the operand stack

Example

The code

```
a[j] = a[i];
```

is translated as

```
aload_1  // load a
iload_3  // load j
aload_1  // load a
iload_2  // load i
iaload   // read a[i]
iastore  // store into a[j]
```

Instruction Group "Stack Manipulation"

- `pop` and `pop2`
  Remove the topmost (2) elements from the operand stack
- `dup`, `dup2`, `dup_x1` ...
  Duplicate the top element(s) of the stack
- `swap`
  Exchange the topmost two elements on the operand stack

Example

The code

```
return a[i] += 1;
```

is translated as

```
aload_1   // load a
iload_2   // load i
dup2      // duplicate, stack contains a,i,a,i
iaload    // read a[i], stack now contains a,i,a[i]
iconst_1
iadd      // add one
dup_x2    // duplicate, stack contains a[i]+1,a,i,a[i]+1
iastore   // store a[i]+1 into a[i].
ireturn   // return duplicated result.
```