Advanced Issues

in Object Oriented Programming

Java: the JVM, Verification, and Loading

We shall discuss

- code development and deployment
- run-time organization
- bytecode
- bytecode verifier
- formalization of the bytecode verifier

You can read more in:

- Tim Lindholm, Frank Yellin: The Java Virtual Machine, Addison Wesley, 1999
- Raymie Stata, Martin Abadi: A Type System for Java Bytecode Subroutines, POPL 1998, and in ToPLaS 1999.
- Vijay Saraswat: Java is not type-safe, http://www.research.att.com/~vj/bug.html (for fun)
- Sheng Liang, Gilad Bracha: Dynamic Class Loading in the JVM, OOPSLA 98
- Zhenyu Qian, Allen Goldberg, Alessandro Coglio: A Formal Specification of Java Class Loading, OOPSLA 2000 (for fun)
- Sophia Drossopoulou: An abstract model of Java dynamic linking, loading and verification, Types in Compilation 2000, LNCS 2071 (for fun)
- Sophia Drossopoulou, Susan Eisenbach: Observing the dynamic linking process in Java, www-dse.doc.ic.ac.uk/projects/slurp/dynamic_link/linking.htm
- Sophia Drossopoulou, Giovanni Lagorio, Susan Eisenbach: Flexible Models of Dynamic Linking, ESOP 2003 (for fun)

Code development and deployment – “conventional”

- Classes compiled separately, in an order consistent with their dependencies (dependency introduced through using, or through subclassing).
- Linker links *all* object code into one object code (“baked brick”), performs some checks.
- The linked code is executed.

Trust comes through trust in the compiler and linker.

Linking code produced by different compilers not obvious.

Code development and deployment – Java/C#

- Classes are compiled separately into bytecode, in any order.
- Bytecode loaded and linked as needed, user-defined loaders provide additional flexibility.
- Bytecode verified before execution.
- Bytecode interpreted by virtual machine.

Trust comes through trust in the verifier, and the design of the virtual machine.

Linking code produced by different compilers default.

Advantages and disadvantages of Java approach:

\+ new versions of code easy to obtain, do not require recompilation of complete application.

\+ initial loading fast.

\+ possible to extend the behaviour of program at run-time.

\- "surprises" during execution.

\- trust (safety) more difficult to establish.

JVM runtime organization

There are the following "data areas":

- the program counters (pc)
- stacks of frames; a frame contains receiver, arguments, local variables, and operand stack.
- heap
- method area
- constant pool
- native method stacks

Note: we have frame stacks *and* operand stacks – more later.

JVM bytecode (is similar to Smalltalk bytecode)

Bytecode instructions are

- load (from some local variable to top of stack)
- store (from top of stack to some local variable)
- arithmetic operations on element(s) on top of stack
- conditional goto depending on value on top of stack
- access field of object top of stack (address of)
- call method – receiver and argument on top of stack
- return value from top of stack (and pop current frame from frame stack)
- …

where “stack” stands for operand stack

In more detail (but with some abstraction)

- aload_<n>, iload_<n> …
- astore_<n>, istore_<n> …
- iadd, imul, isub, ineg, iinc …
- ifnull branch1 branch2 …
- getfield <fSign>, putfield <fSign> …
- invokevirtual <mSign> …
- ireturn, areturn, return, …

The type of the operand is reflected in the prefix of the instruction, ie a for reference (address), i for integer etc.

Also, <n> is some number, <fSign> is a field signature, <mSign> is a method signature.

Each class `A` is compiled into an `A.class` file, containing

- name of class,
- name of superclass, superinterfaces
- types of fields
- signatures of methods called
- signatures of fields accessed
- method bodies

Each method body contains

- name of the method,
- signature of method(type of parameters, type of result)
- size of stack (stack required for intermediate results)
- size for locals (locals are the arguments and local variables)
- size for arguments (including implicitly passed receiver, at the 0-th position)
- bytecode

Types are represented by strings (dangerous).

Notes:

- we use ”stack” for “operand stack”, and shall explicitly say “frame stack” when we mean that .
- One can inspect bytecode, by using `javac` and `javap`

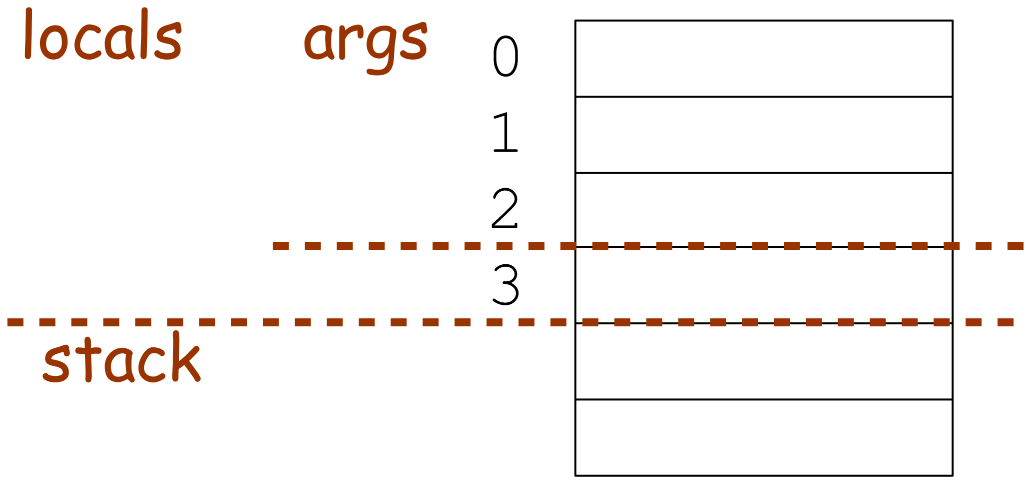
Example 1: for the following Java code

```
class A{
    int m(int x, int y){
        int z;
        z = x + y;
        return z; }
}
```

method `int m(int,int)` is translated (almost) to:

```
class A extends java.langObject{
Method  int m(int, int)
/* Stack = 2, Locals = 4, Args_size = 3 */
      0    iload_1
      1    iload_2
      2    iadd
      3    istore_3
      4    iload_3
      5    ireturn
```

Therefore, the frame for method **int** m(**int**,**int**) from A could look like



(the relative position of stack and locals, and of locals and args is arbitrary)

Some information on jvm instructions:

- `iload m` stores the `m`-th local variable (interpreted as `int`) onto the top of the stack.
- `aload m` stores the `m`-th local variable (interpreted as reference) onto the top of the stack
- `iadd` pops the two top values from the stack (interpreted as `int`), and pushes their sum onto the top of the stack.
- `istore m` pops the value from top of the stack (interpreted as `int`) and stores it onto the `m`-th local variable.
- `ireturn` pops value from top of stack (interpreted as `int`), discharges current frame, pushes value on top of stack of frame of invoking method, and returns control to invoking method.

Questions

1) What is the effect of the instructions `iload`, `aload`, `iadd`, `istore` on the length of the stack?

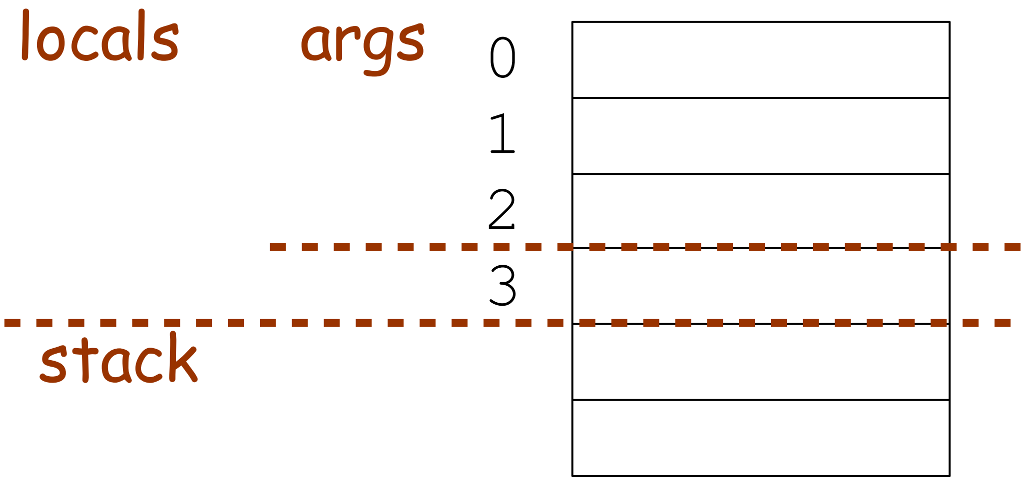
2) Could anything "go wrong" when executing the instructions `iload`, `aload`, `iadd`, `istore`, `ireturn`? If yes, how much should I worry about security in Java?

3) Question

Consider method **int** m(**int**,**int**) translated to:

```
method  int m(int, int)
/* Stack = 2, Locals = 4, Args_size = 3 */
       0   iload_1
       1   iload_2
       2   iadd
       3   istore_3
       4   iload_3
       5   ireturn
```

Execute `A a; a = new A();` // a is at address `6AF7`,
and the call `a.m(25,45)`

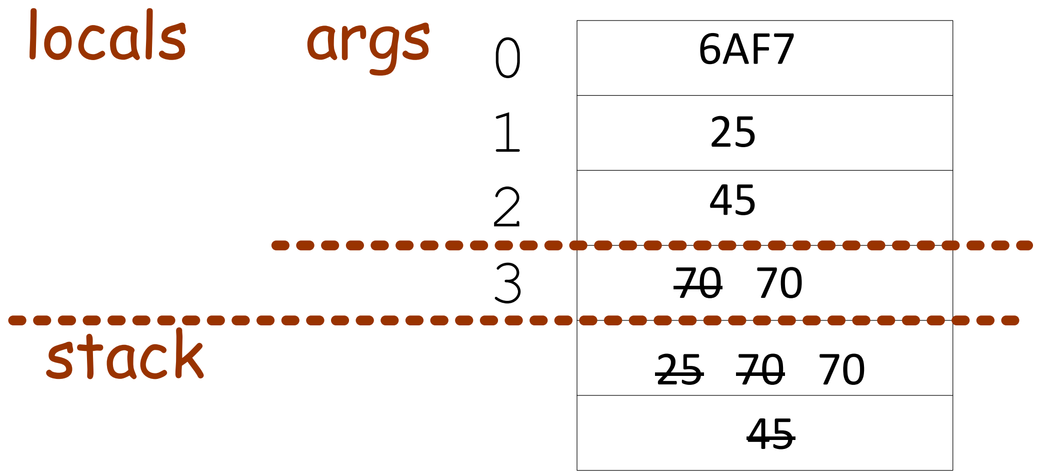


3) Question

Consider method **int** m(**int**,**int**) translated to:

```
method  int m(int, int)
/* Stack = 2, Locals = 4, Args_size = 3 */
        0   iload_1
        1   iload_2
        2   iadd
        3   istore_3
        4   iload_3
        5   ireturn
```

Execute `A a; a = new A();` // a is at address `6AF7`,
and the call `a.m(25,45)`



Some further information on jvm instructions:

- `getfield fSign` pops the top of stack and interprets it as the address of an object in the heap; selects the field with signature `fSign` from that object and pushes it onto stack.
- `putfield fSign` pops a value from the top of stack; then pops the top of the stack and interprets it as the address of an object in the heap; and overwrites its field with signature `fSign` with the value.
- `iconst_m` pushes `int` constant `m` onto stack.
- `return` discharges current frame, invoking method.and returns control to caller

Question: `putfield` expects the top of the stack to be a value, and the top-1 of the stack to be a reference. Why is it not the other way round?

Example 2: for Java classes

```
class A{ int fa; }
class B extends A{
    int fb;
    void m(B x){ fb = x.fa + 1; }
}
```

the method `void m(B x)` is translated (almost) to:

```
Method void m(B)
/* Stack = 3, Locals = 2, Args_size = 2 */
      0    aload_0
      1    aload_1
      2    getfield  <int fa> from A
      5    iconst_1
      6    iadd
      7    putfield  <int fb> from B
      10   return
```

`<int fa> from A` ≡ field `fa`, in class `A`, type `int`

```
Method void m(B)
/* Stack = 3, Locals = 2,
   Args_size = 2 */
  0  aload_0
  1  aload_1
  2  getfield  <int fa> from A
  5  iconst_1
  6  iadd
  7  putfield <int fb> from B
  10 return
```

Execution of the function body, for

```
B b1; B b2;
b1 = new B();
b2 = new B();
b1.fb = 30;
b2.fa = 20;
```

then, assuming that `b1` was allocated at `3068` and `b2` was allocated at `3070`, write out the execution of:

```
b1.m(b2)
```

locals	args		
		0	
		1	
stack			

heap

3068	(B)
	0
	30
3070	(B)
	20
	0

Example 2 – revisited

```
Method void m(B)
/* Stack = 3, Locals = 2,
   Args_size = 2 */
  0  aload_0
  1  aload_1
  2  getfield  <int fa> from A
  5  iconst_1
  6  iadd
  7  putfield <int fb> from B
  10 return
```

Execution of the function body, for

```
B b1; B b2;
b1 = new B();
b2 = new B();
b1.fb = 30;
b2.fa = 20;
```

where `b1` was allocated at `3068` and `b2` was allocated at `3070`, write out the execution of:

```
b1.m(b2)
```

locals args

0	3068
1	3070

stack

~~3068~~
~~3070~~ ~~20~~ ~~21~~
~~1~~

heap

3068	(B)
	20
	~~30~~ 21
3070	(B)
	20
	0

Questions

1) What is the effect of the instructions `putfield`, `getfield`, `iconst` on the length of the stack?

2) How do the instructions `putfield`, `getfield`, distinguish between the different sizes required for fields of type boolean, reference, double etc?

3) Could anything "go wrong" when executing the instructions `putfield`, `getfield`, `iconst`, `return`? If yes, how much should I worry about Java security?

4) The instructions `putfield`, `getfield`, look as expensive as field lookup and update in L2. Is that so?

Some further information on jvm instructions:

- `invokevirtual mSign` creates a new frame and pushes on its arguments part the `nargs` top entries from the old operand stack (`nargs` the number of arguments, incl receiver, from `mSign`), selects the method with signature `mSign` from the object referred to by top of (new) stack
- `if_acmpne m` pops the top two reference values of the stack, if equal, execution continues at next instruction, otherwise execution proceeds at `m` (we simplified offsets)
- `if_acmpeq m ....`
- `return` discharges current frame, and returns control to invoking method

Question: `invokevirtual` expects the top n entries of the stack to be the arguments, and top-n to be the receiver. Why not the other way round?

Example 3: for Java classes

```
class A{ int m(B x, A y){…} }
class B {
int m(A y, A z){ return y.m(this,z); }
}
```

the method `int m(A y, A z)` is translated (almost) to:

```
Method int m(A, A)
/* Stack = 3, Locals = 3, Args_size = 3 */
    0  aload_1
    1  aload_0
    2  aload_2
    3  invokevirtual  <int m(B,A)> from A
```

`<int m(B,A)> from A` ≡

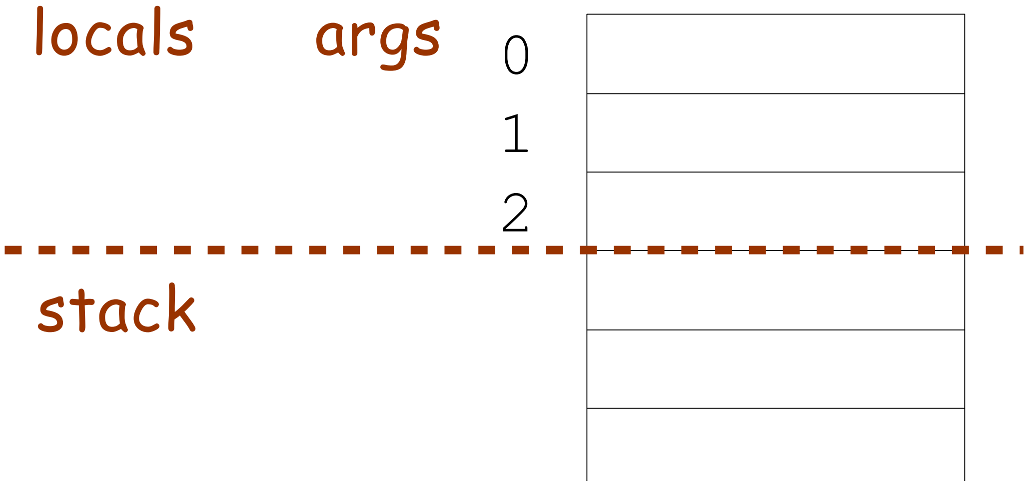
method `m`,
argument types `B` and `A`, return type `int`,
in class `A`.

```
Method int m(A, A)
/* Stack = 3, Locals = 3, Args_size =
3 */
   0   aload_1
   1   aload_0
   2   aload_2
   3   invokevirtual <int m(B,A)> from A
```

Execution of

```
A a1, a2; B b;
a1    = new  A();
a2 =new A();
  b  =  new  B();
  b.m(a1,a2)
```

would proceed as



- -

Example 4: for Java classes

```
class A{    }
class B extends A { }
class C {
      A m(A x, B y ){
          if (x==y){ return x; };
        return y; }   }
```

the method `A m(A x)` is translated (almost) to:

```
Method A m(A,B)
/* Stack = 2, Locals = 3, Args_size = 3 */
      0    aload_1
      1    aload_2
      2    if_acmpne 7
      5    aload_1
      6    areturn
      7    aload_2
      8    areturn
```

So far so good, but …

- the bytecode is a language with limited types, implementing a language with types
- there is no guarantee as to who/how produced bytecode

This might open opportunities for disasters.

Potential disaster #1

Example 5: Consider:

```
class A{ C fa; }
class B{    }
class C{    }
class D{    }
```

and some method:

```
Method void int m(B,D)
/* Stack = 2, Locals = 3, Args_size = 3
  0  aload_1
  1  aload_2
  2  putfield <C fa> from class A
  5  return
```

... destroys integrity of heap.

Aside ------------------------------------

You can create the above situation if

1) you "hack" the class files

or

2) you compile

```
class A{ C fa; }
class B extends  A {     }
class C{    }
class D extends C {     }
class Trouble {
   void m(B b, D d)
          { b.fa=d; }   }
```

and then compile

```
class B {    }

class D {    }
```

---------------------------------- End Aside

Potential disaster #2

Example 6: Consider:

```
class A{ C m(B x){…} }
class B{    }
class C{    }
class D{ private void f(A y){ … } }
```

and some bytecode method:

```
Method void m(C,D)
/* Stack = 2, Locals = 3, Args_size = 3
  0  aload_2
  1  aload_1
  2  invokevirtual  <C m(B)> from class A
  5  return
```

… may call privileged methods.

Potential disaster #3

Example 7: Consider some bytecode method:

```
Method void m(int)
/* Stack = 2, Locals = 2, Args_size = 2
  0  aload_0
  1  aload_1
  2  putfield  C,int,f
  3  putfield  C,int,f
  4  putfield  C,int,f
  5  putfield  C,int,f
  6  putfield  C,int,f
  7  putfield  C,int,f
  8  return
```

… destroys the integrity of frames stack .

The possibilities for such misuses are vast, and through the use of conditionals, subroutines, threads etc their detection becomes more difficult

The answer is …

… *Verifier to the rescue*