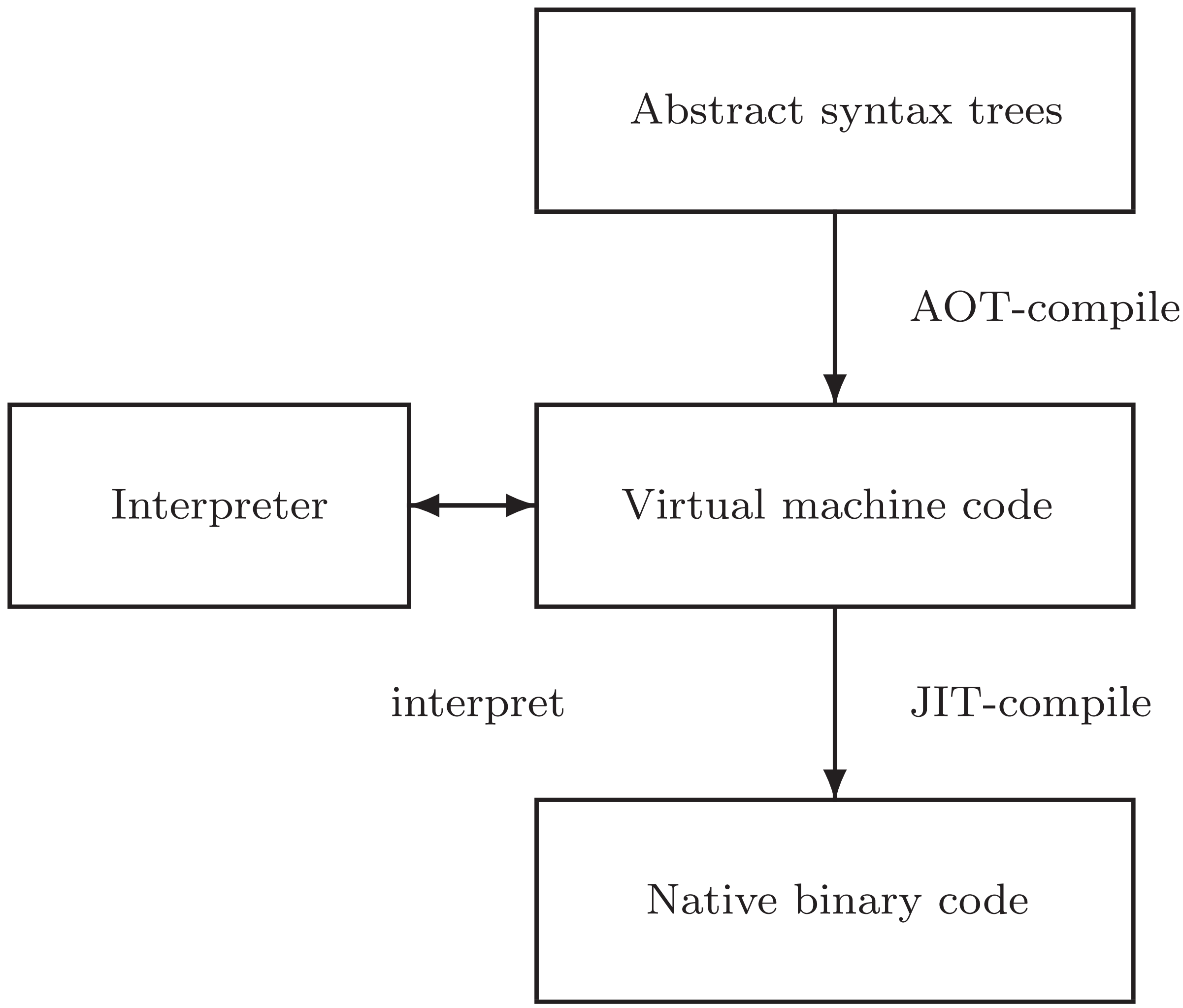
Virtual machines

Compilation and execution modes of Virtual machines:



Compilers traditionally compiled to machine code ahead-of-time (AOT).

Example:

- `gcc` translates into RTL (Register Transfer Language), optimizes RTL, and then compiles RTL into native code.

Advantages:

- can exploit many details of the underlying architecture; and
- intermediate languages like RTL facilitate production of code generators for many target architectures.

Disadvantage:

- a code generator must be built for each target architecture.

Interpreting virtual machine code.

Examples:

- P-code for early Pascal interpreters;
- Postscript for display devices; and
- Java bytecode for the Java Virtual Machine.

Advantages:

- easy to generate the code;
- the code is architecture independent; and
- bytecode can be more compact.

Disadvantage:

- poor performance due to interpretative overhead (typically 5-20 $\times$ slower).
  Reasons:
  - Every instruction considered in isolation,
  - confuses branch prediction,
  - ...and many more.

VirtualRISC is a simple RISC machine with:

- memory;
- registers;
- condition codes; and
- execution unit.

In this model we ignore:

- caches;
- pipelines;
- branch prediction units; and
- advanced features.

VirtualRISC memory:

- a stack
  (used for function call frames);
- a heap
  (used for dynamically allocated memory);
- a global pool
  (used to store global variables); and
- a code segment
  (used to store VirtualRISC instructions).

VirtualRISC registers:

- unbounded number of general purpose registers;
- the stack pointer (`sp`) which points to the top of the stack;
- the frame pointer (`fp`) which points to the current stack frame; and
- the program counter (`pc`) which points to the current instruction.

VirtualRISC condition codes:

- stores the result of last instruction that can set condition codes (used for branching).

VirtualRISC execution unit:

- reads the VirtualRISC instruction at the current `pc`, decodes the instruction and executes it;
- this may change the state of the machine (memory, registers, condition codes);
- the `pc` is automatically incremented after executing an instruction; but
- function calls and branches explicitly change the `pc`.

Memory/register instructions:

```
st Ri,[Rj]            [Rj] := Ri
st Ri,[Rj+C]          [Rj+C] := Ri

ld [Ri],Rj            Rj := [Ri]
ld [Ri+C],Rj          Rj := [Ri+C]
```

Register/register instructions:

```
mov Ri,Rj             Rj := Ri
add Ri,Rj,Rk          Rk := Ri + Rj
sub Ri,Rj,Rk          Rk := Ri - Rj
mul Ri,Rj,Rk          Rk := Ri * Rj
div Ri,Rj,Rk          Rk := Ri / Rj
...
```

Constants may be used in place of register values: `mov 5,R1`.

Instructions that set the condition codes:

```
cmp Ri,Rj
```

Instructions to branch:

```
b L
bg L
bge L
bl L
ble L
bne L
```

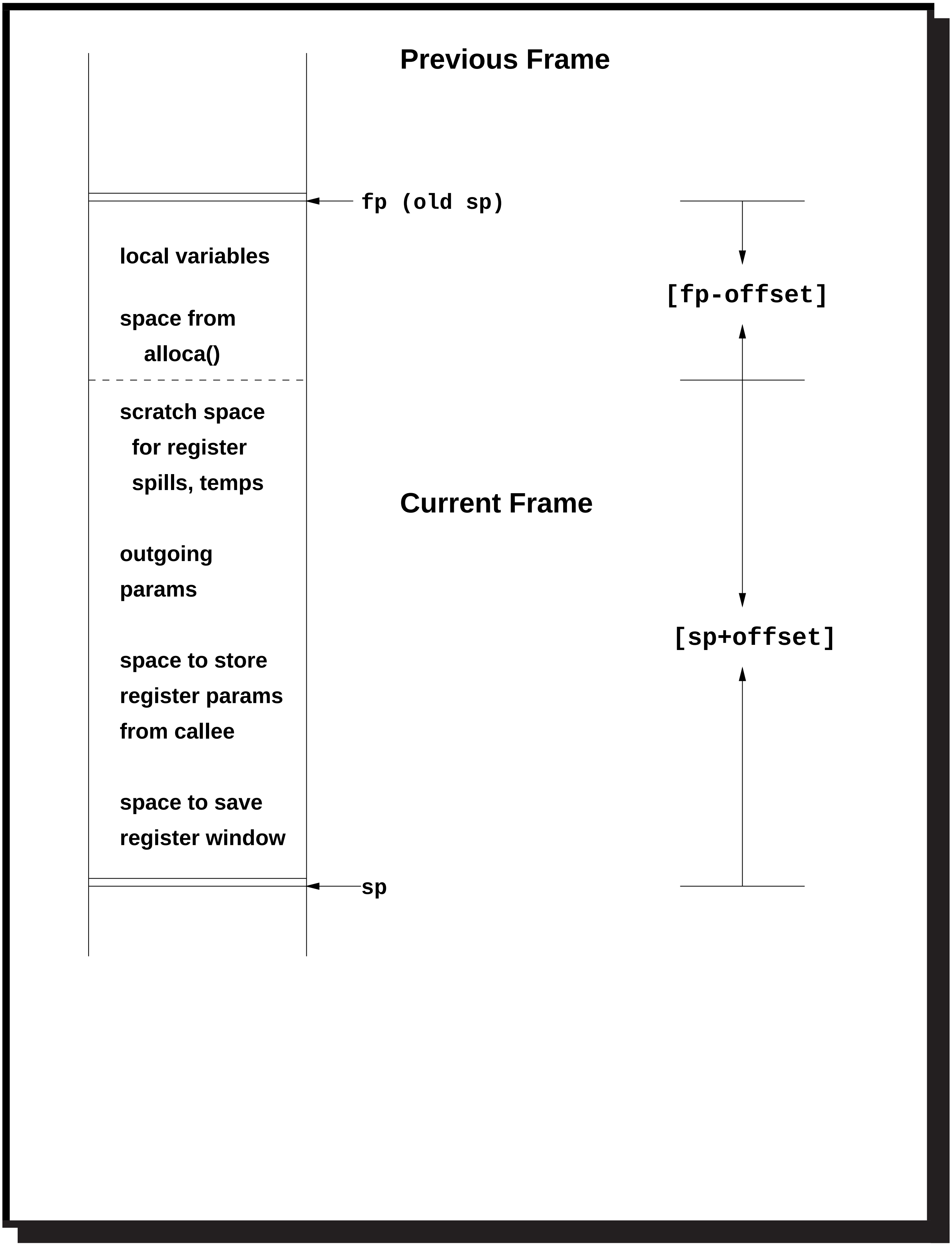
To express: `if R1 <= 9 goto L1`

we code:

```
cmp R1,9
ble L1
```

Other instructions:

```
save sp,-C,sp        save registers,
                     allocating C bytes
                     on the stack
call L               R15:=pc; pc:=L
restore              restore registers
ret                  pc:=R15+8
nop                  do nothing
```


Previous Frame
fp (old sp)
local variables
space from alloca()
scratch space for register spills, temps
outgoing params
space to store register params from callee
space to save register window
sp
Current Frame
[fp-offset]
[sp+offset]

Stack frames:

- stores function activations;
- `sp` and `fp` point to stack frames;
- when a function is called a new stack frame is created:
  `push fp; fp := sp; sp := sp + C`;
- when a function returns, the top stack frame is popped:
  `sp := fp; fp = pop`;
- local variables are stored relative to `fp`;
- the figure shows additional features of the SPARC architecture.

A simple C function:

```
int fact(int n)
{ int i, sum;
  sum = 1;
  i = 2;
  while (i <= n)
    { sum = sum * i;
      i = i + 1;
    }
  return sum;
}
```

Corresponding VirtualRISC code:

```
_fact:
  save sp,-112,sp   // save stack frame
  st R0,[fp+68]     // save input arg n in frame of CALLER
  mov 1,R0          // R0 := 1
  st R0,[fp-16]     // [fp-16] is location for sum
  mov 2,R0          // RO := 2
  st RO,[fp-12]    // [fp-12] is location for i
L3:
  ld [fp-12],R0     // load i into R0
  ld [fp+68],R1     // load n into R1
  cmp R0,R1         // compare R0 to R1
  ble L5            // if R0 <= R1 goto L5
  b L4              // goto L4
L5:
  ld [fp-16],R0     // load sum into R0
  ld [fp-12],R1     // load i into R1
  mul R0,R1,R0      // R0 := R0 * R1
  st R0,[fp-16]     // store R0 into sum
  ld [fp-12],R0     // load i into R0
  add R0,1,R1       // R1 := R0 + 1
  st R1,[fp-12]     // store R1 into i
  b L3              // goto L3
L4:
  ld [fp-16],R0     // put return value of sum into R0
  restore           // restore register window
  ret               // return from function
```

Java Virtual Machine has:

- memory;
- registers;
- condition codes; and
- execution unit.

Java Virtual Machine memory:

- a stack
  (used for function call frames);
- a heap
  (used for dynamically allocated memory);
- a constant pool
  (used for constant data that can be shared);
  and
- a code segment
  (used to store JVM instructions of currently loaded class files).

Java Virtual Machine registers:

- no general purpose registers;
- the stack pointer (`sp`) which points to the top of the stack;
- the local stack pointer (`lsp`) which points to a location in the current stack frame; and
- the program counter (`pc`) which points to the current instruction.

Java Virtual Machine condition codes:

- stores the result of last instruction that can set condition codes (used for branching).

Java Virtual Machine execution unit:

- reads the Java Virtual Machine instruction at the current `pc`, decodes the instruction and executes it;
- this may change the state of the machine (memory, registers, condition codes);
- the `pc` is automatically incremented after executing an instruction; but
- method calls and branches explicitly change the `pc`.

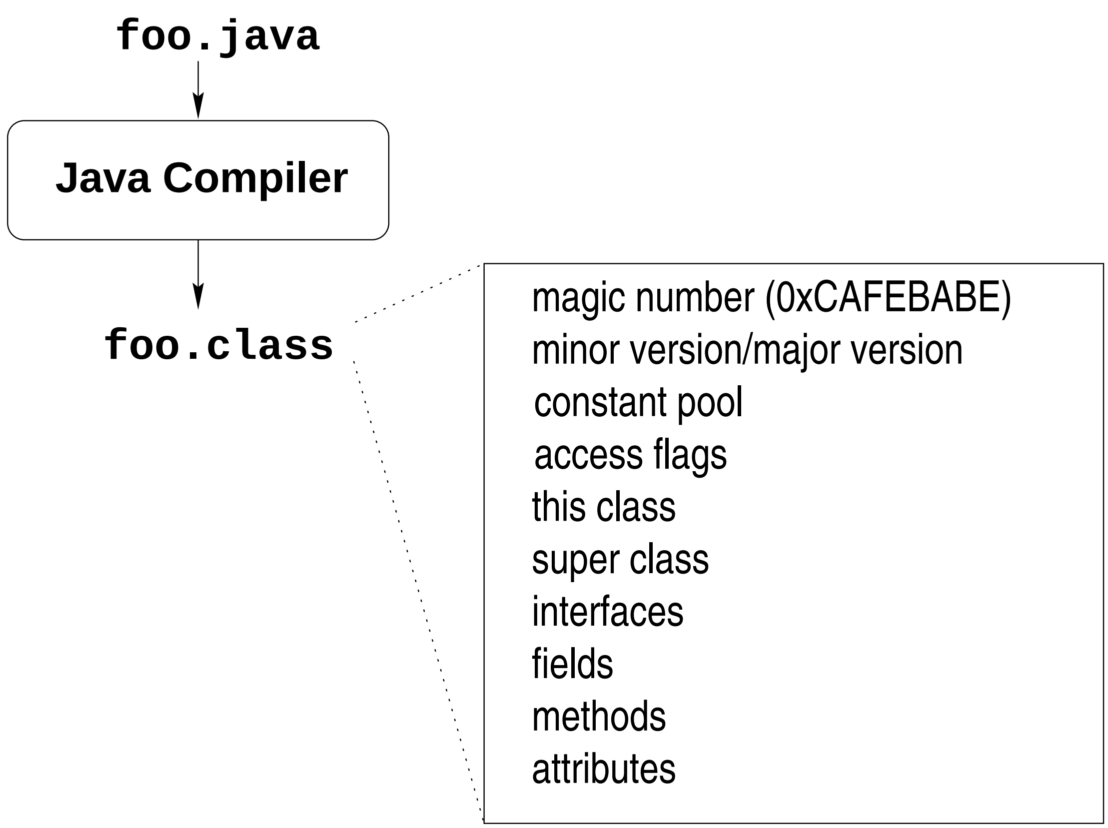
Java Virtual Machine stack frames have space for:

- a reference to the current object (`this`);
- the method arguments;
- the local variables; and
- a local stack used for intermediate results.

The number of local slots and the maximum size of the local stack are fixed at compile-time.

Java compilers translate source code to class files.

Class files include the bytecode instructions for each method.



A simple Java method:

```
public int Abs(int x)
{ if (x < 0)
    return(x * -1);
  else
    return(x);
}
```

Corresponding bytecode (in Jasmin syntax):

```
.method public Abs(I)I // one int argument, returns an int
.limit stack 2         // has stack with 2 locations
.limit locals 2        // has space for 2 locals

                       // --locals--  --stack---
                       // [ o -3 ]     [  *  * ]
  iload_1              // [ o -3 ]     [ -3  * ]
  ifge Label1          // [ o -3 ]     [  *  * ]
  iload_1              // [ o -3 ]     [ -3  * ]
  iconst_m1            // [ o -3 ]     [ -3 -1 ]
  imul                 // [ o -3 ]     [  3  * ]
  ireturn              // [ o -3 ]     [  *  * ]
Label1:
  iload_1
  ireturn
.end method
```

Comments show trace of `o.Abs(-3)`.

A sketch of a bytecode interpreter:

```
pc = code.start;
while(true)
  {  npc = pc + instruction_length(code[pc]);
     switch (opcode(code[pc]))
       {  case ILOAD_1: push(local[1]);
                        break;
          case ILOAD:   push(local[code[pc+1]]);
                        break;
          case ISTORE:  t = pop();
                        local[code[pc+1]] = t;
                        break;
          case IADD:    t1 = pop();  t2 = pop();
                        push(t1 + t2);
                        break;
          case IFEQ:    t = pop();
                        if (t == 0) npc = code[pc+1];
                        break;
          ...
       }
     pc = npc;
  }
```

Unary arithmetic operations:

```
ineg        [...:i] -> [...:-i]
i2c         [...:i] -> [...:i%65536]
```

Binary arithmetic operations:

```
iadd        [...:i1:i2] -> [...:i1+i2]
isub        [...:i1:i2] -> [...:i1-i2]
imul        [...:i1:i2] -> [...:i1*i2]
idiv        [...:i1:i2] -> [...:i1/i2]
irem        [...:i1:t2] -> [...:i1%i2]
```

Direct operations:

```
iinc k a    [...]  -> [...]
            local[k]=local[k]+a
```

Nullary branch operations:

```
goto L          [...]  -> [...]
                branch always
```

Unary branch operations:

```
ifeq L          [...:i] -> [...]
                branch if i == 0
ifne L          [...:i] -> [...]
                branch if i != 0

ifnull L        [...:o] -> [...]
                branch if o == null
ifnonnull L     [...:o] -> [...]
                branch if o != null
```

Binary branch operations:

```
if_icmpeq L   [...:i1:i2] -> [...]
              branch if i1 == i2
if_icmpne L   [...:i1:i2] -> [...]
              branch if i1 != i2
if_icmpgt L   [...:i1:i2] -> [...]
              branch if i1 > i2
if_icmplt L   [...:i1:i2] -> [...]
              branch if i1 < i2
if_icmple L   [...:i1:i2] -> [...]
              branch if i1 <= i2
if_icmpge L   [...:i1:i2] -> [...]
              branch if i1 >= i2

if_acmpeq L   [...:o1:o2] -> [...]
              branch if o1 == o2
if_acmpne L   [...:o1:o2] -> [...]
              branch if o1 != o2
```

Constant loading operations:

```
iconst_0       [...]  -> [...:0]
iconst_1       [...]  -> [...:1]
iconst_2       [...]  -> [...:2]
iconst_3       [...]  -> [...:3]
iconst_4       [...]  -> [...:4]
iconst_5       [...]  -> [...:5]

aconst_null    [...]  -> [...:null]

ldc_int i      [...]  -> [...:i]
ldc_string s   [...]  -> [...:String(s)]
```

Locals operations:

```
iload k          [...]  -> [...:local[k]]
istore k         [...:i] -> [...]
                 local[k]=i

aload k          [...]  -> [...:local[k]]
astore k         [...:o] -> [...]
                 local[k]=o
```

Field operations:

```
getfield f sig   [...:o] -> [...:o.f]
putfield f sig   [...:o:v] -> [...]
                 o.f=v
```

Stack operations:

```
dup            [...:v1] -> [...:v1:v1]
pop            [...:v1] -> [...]
swap           [...:v1:v2] -> [...:v2:v1]
nop            [...]  -> [...]
```

Class operations:

```
new C           [...]  -> [...:o]

instance_of C   [...:o] -> [...:i]
                if (o==null) i=0
                else i=(C<=type(o))

checkcast C     [...:o] -> [...:o]
                if (o!=null && !C<=type(o))
                throw ClassCastException
```

Method operations:

```
invokevirtual m sig
     [...:o:a1:...:an] -> [...]

//overloading already resolved:
// signature of m is known!
entry=lookupHierarchy(m,sig,class(o));
block=block(entry);
push stack frame of size
     block.locals+block.stacksize;
local[0]=o; //local points to
local[1]=a1; //beginning of frame
...
local[n]=an;
pc=block.code;
```

Method operations:

```
invokespecial m sig
     [...:o:a1:...:an] -> [...]

//overloading already resolved:
// signature of m is known!
entry=lookupClassOnly(m,sig,class(o));
block=block(entry);
push stack frame of size
     block.locals+block.stacksize;
local[0]=o; //local points to
local[1]=a1; //beginning of frame
...
local[n]=an;
pc=block.code;
```

For which method calls is `invokespecial` used?

Method operations:

```
ireturn     [...:<frame>:i] -> [...:i]
            pop stack frame,
            push i onto frame of caller

areturn     [...:<frame>:o] -> [...:o]
            pop stack frame,
            push o onto frame of caller

return      [...:<frame>] -> [...]
            pop stack frame
```

Those operations also release locks in `synchronized` methods.

A Java method:

```
public boolean member(Object item)
{ if (first.equals(item))
    return true;
  else if (rest == null)
    return false;
  else
    return rest.member(item);
}
```

Corresponding bytecode (in Jasmin syntax):

```
.method public member(Ljava/lang/Object;)Z
.limit locals 2           // local[0] = o
                          // local[1] = item
.limit stack 2            // initial stack [ * * ]
aload_0                   // [ o * ]
getfield Cons/first Ljava/lang/Object;
                          // [ o.first *]
aload_1                   // [ o.first item]
invokevirtual java/lang/Object/equals(Ljava/lang/Object;)Z
                          // [ b * ] for some boolean b
ifeq else_1               // [ * * ]
iconst_1                  // [ 1 * ]
ireturn                   // [ * * ]
else_1:
aload_0                   // [ o * ]
getfield Cons/rest LCons; // [ o.rest * ]
aconst_null               // [ o.rest null]
if_acmpne else_2          // [ * * ]
iconst_0                  // [ 0 * ]
ireturn                   // [ * * ]
else_2:
aload_0                   // [ o * ]
getfield Cons/rest LCons; // [ o.rest * ]
aload_1                   // [ o.rest item ]
invokevirtual Cons/member(Ljava/lang/Object;)Z
                          // [ b * ] for some boolean b
ireturn                   // [ * * ]
.end method
```

Bytecode verification:

- bytecode cannot be trusted to be well-formed and well-behaved;
- before executing any bytecode, it should be verified, especially if that bytecode is received over the network;
- verification is performed partly at class loading time, and partly at run-time; and
- at load time, dataflow analysis is used to approximate the number and type of values in locals and on the stack.

Interesting properties of verified bytecode:

- each instruction must be executed with the correct number and types of arguments on the stack, and in locals (on all execution paths);
- at any program point, the stack is the same size along all execution paths;
- every method must have enough locals to hold the receiver object (except static methods) and the method's arguments; and
- no local variable can be accessed before it has been assigned a value.

Java class loading and execution model:

- when a method is invoked, a `ClassLoader` finds the correct class and checks that it contains an appropriate method;
- if the method has not yet been loaded, then it is verified (remote classes);
- after loading and verification, the method body is interpreted.
- If the method becomes executed multiple times, the bytecode for that method is translated to native code.
- If the method becomes hot, the native code is optimized.

The last two steps are very involved and companies like Sun and IBM have a thousand people working on optimizing these steps.

$\Rightarrow$ good for you! (why not 1001 people?)

Split-verification in Java 6+:

- Bytecode verification is easy but still polynomial, i.e. sometimes slow, and
- this can be exploited in denial-of-service attacks: `http://www.bodden.de/research/javados/`
- Java 6 (version 50.0 bytecodes) introduced `StackMapTable` attributes to make verification linear.
  - Java compilers know the type of locals at compile time.
  - Java 6 compilers store these types in the bytecode using `StackMapTable` attributes.
  - Speeds up construction of the "proof tree" $\Rightarrow$ also called "Proof-Carrying Code"
- Java 7 (version 51.0 bytecodes) JVMs will enforce presence of these attributes.

Future use of Java bytecode:

- the JOOS compiler will produce Java bytecode in Jasmin format; and
- the JOOS peephole optimizer transforms bytecode into more efficient bytecode.

Future use of VirtualRISC:

- Java bytecode can be converted into machine code at run-time using a JIT (Just-In-Time) compiler;
- we will study some examples of converting Java bytecode into a language similar to VirtualRISC;
- we will study some simple, standard optimizations on VirtualRISC.