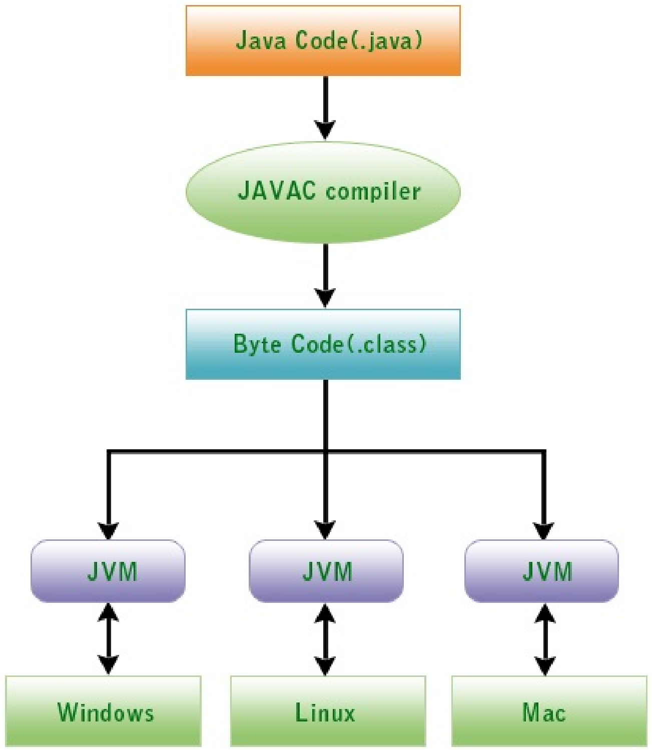
Virtual Machines

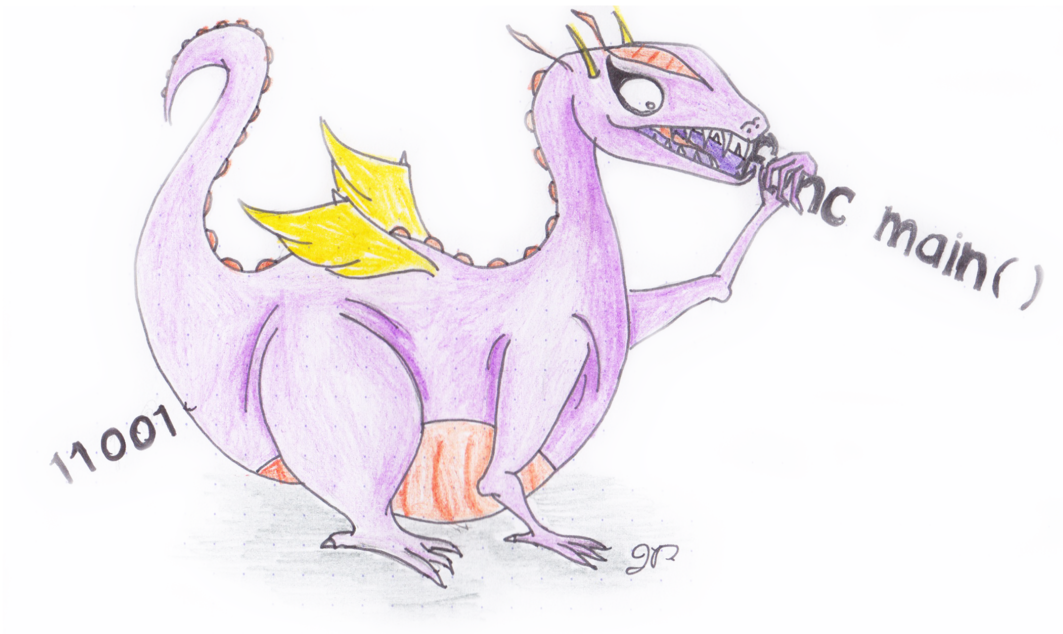
COMP 520: Compiler Design (4 credits)

Professor Laurie Hendren

hendren@cs.mcgill.ca



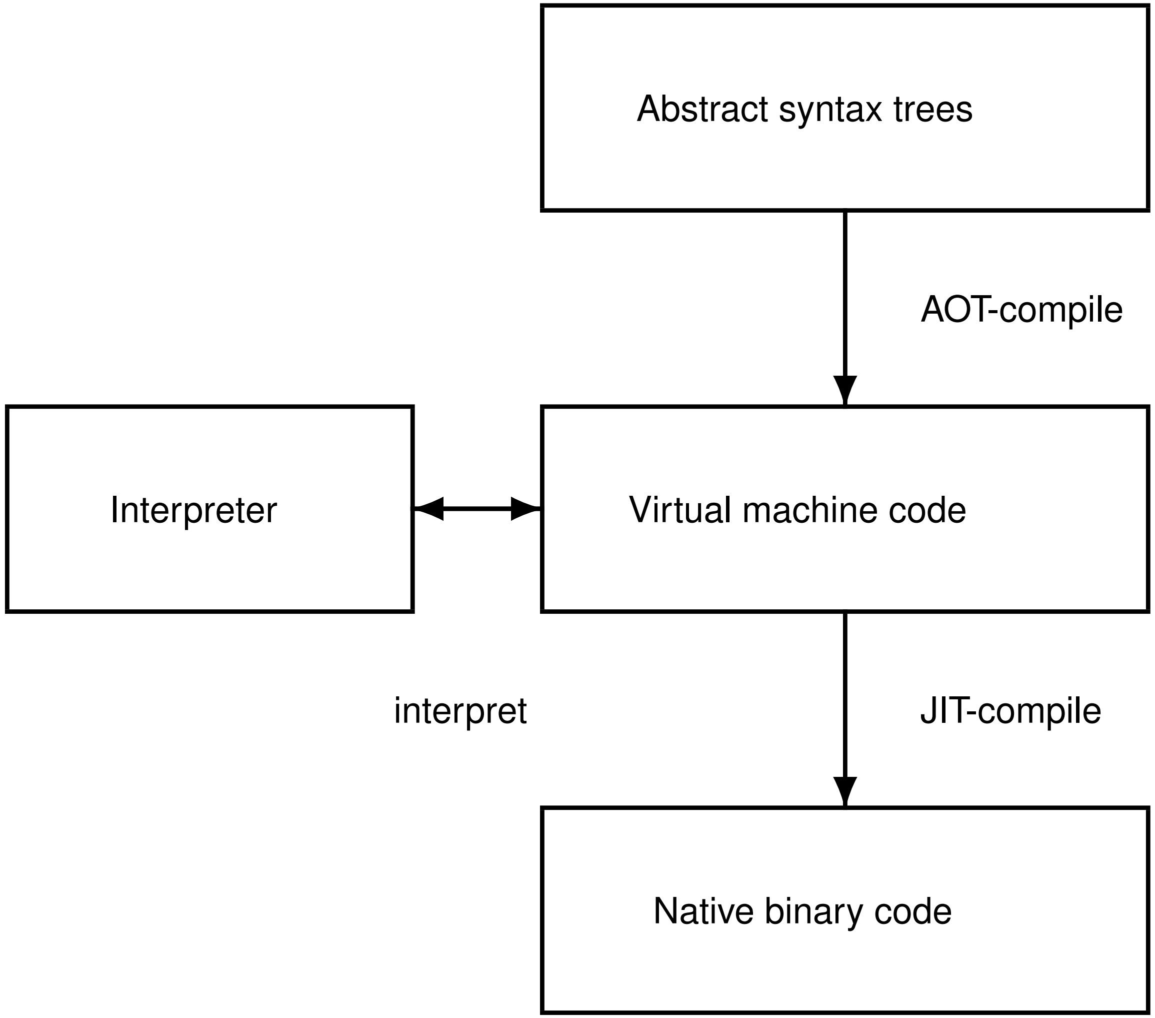
From http://www.devmanuals.com/tutorials/java/corejava/JavaVirtualMachine.html



WendyTheWhitespace-IntolerantDragon

WendyTheWhitespacenogarDtnarelotnI

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.

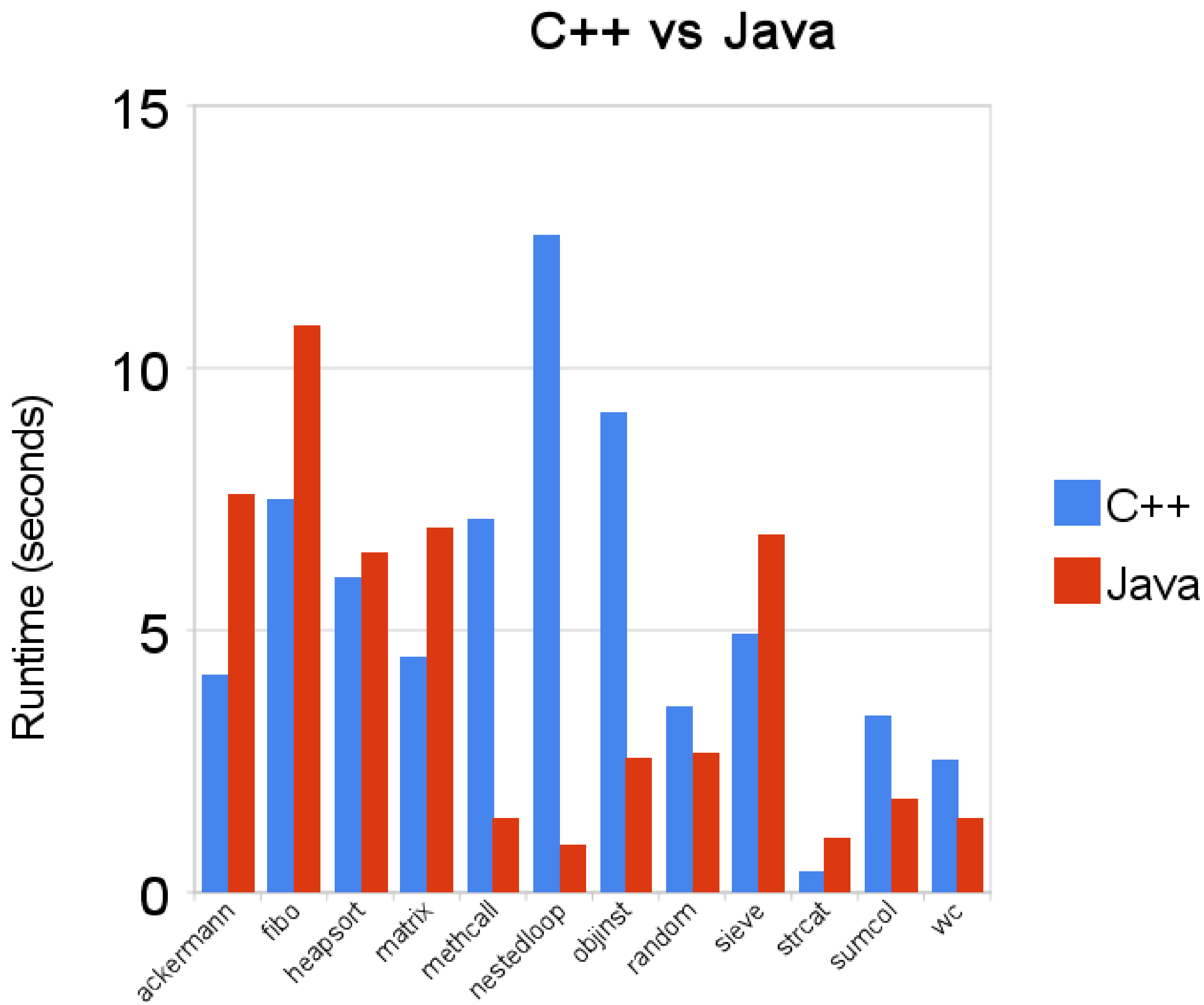
Advantages:

- 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.

But, modern Java is quite efficient



http://blog.cfelde.com/2010/06/c-vs-java-performance/

Let's look at two virtual machines ....

VirtualRISC: register-based IR

Java Virtual Machine: stack-based IR

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:

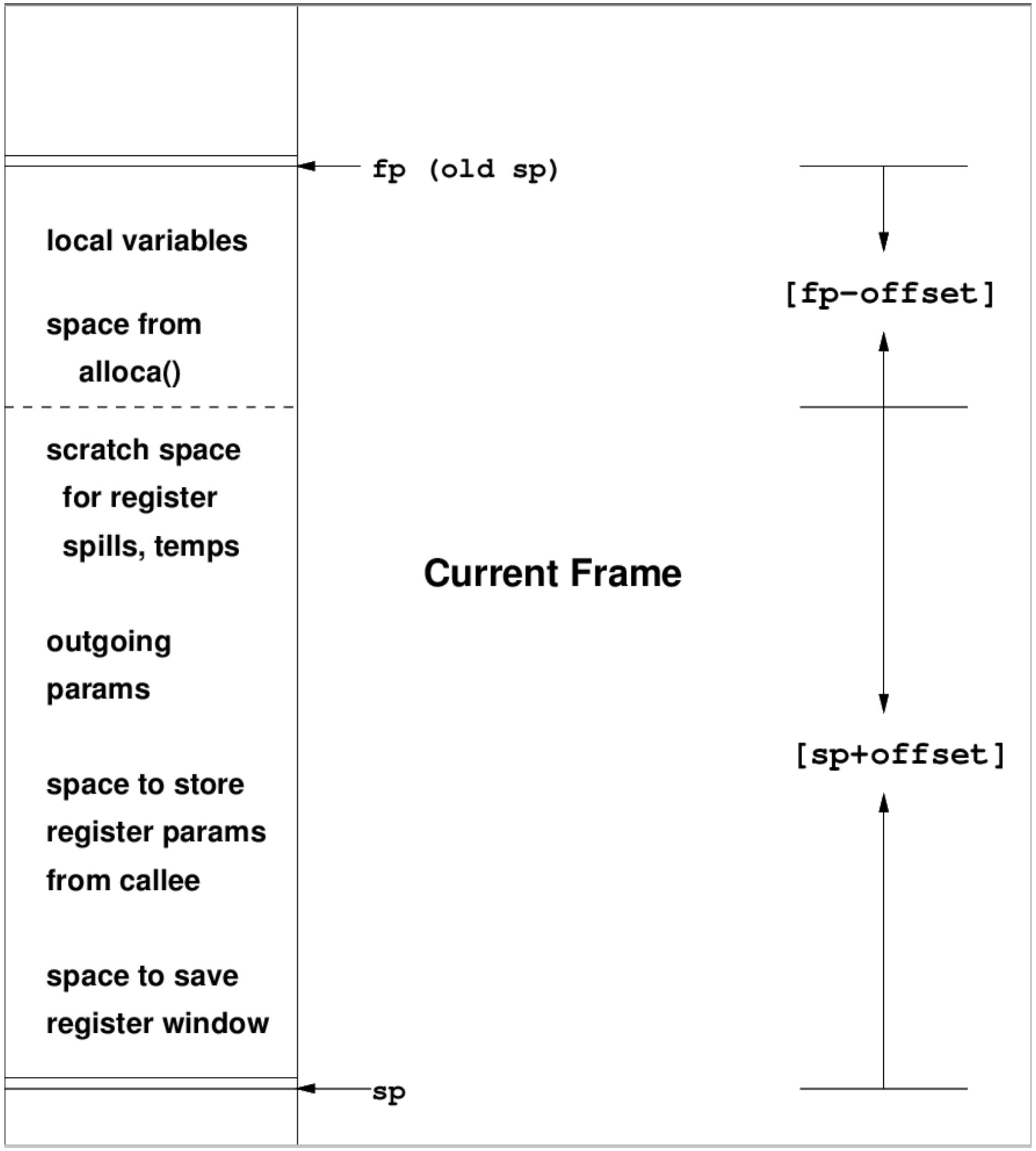
```
b L
bg L
bge L
bl L
ble L
bne L
```

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
```


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;
}
```

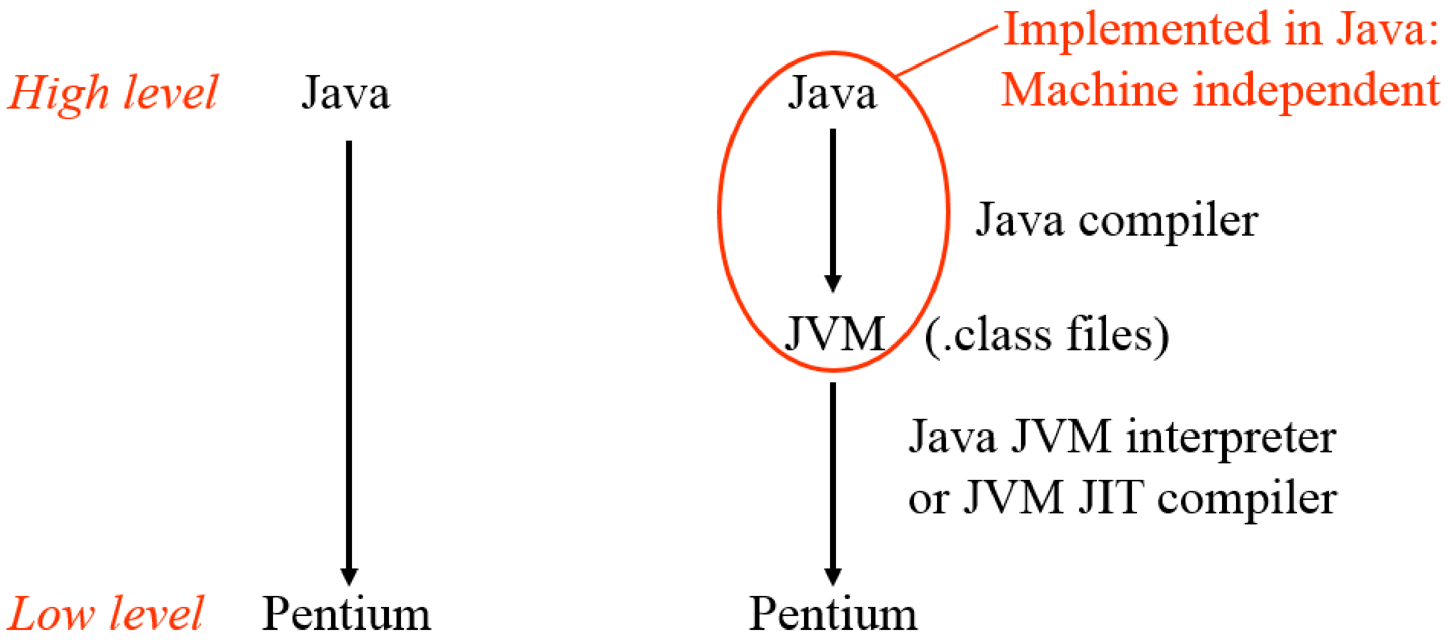
Corresponding VirtualRISC code:

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

```
_fact:
   save sp,-112,sp     // save stack frame
   st R0,[fp+68]       // save 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
```

Abstract Machines

Abstract machine implements an intermediate language in between the high-level language (e.g. Java) and the low-level hardware (e.g. Pentium)



Note: slides of this format from: `http://cs434.cs.ua.edu/Classes/20_JVM.ppt`

Java Virtual Machine has:

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

Java Virtual Machine memory:

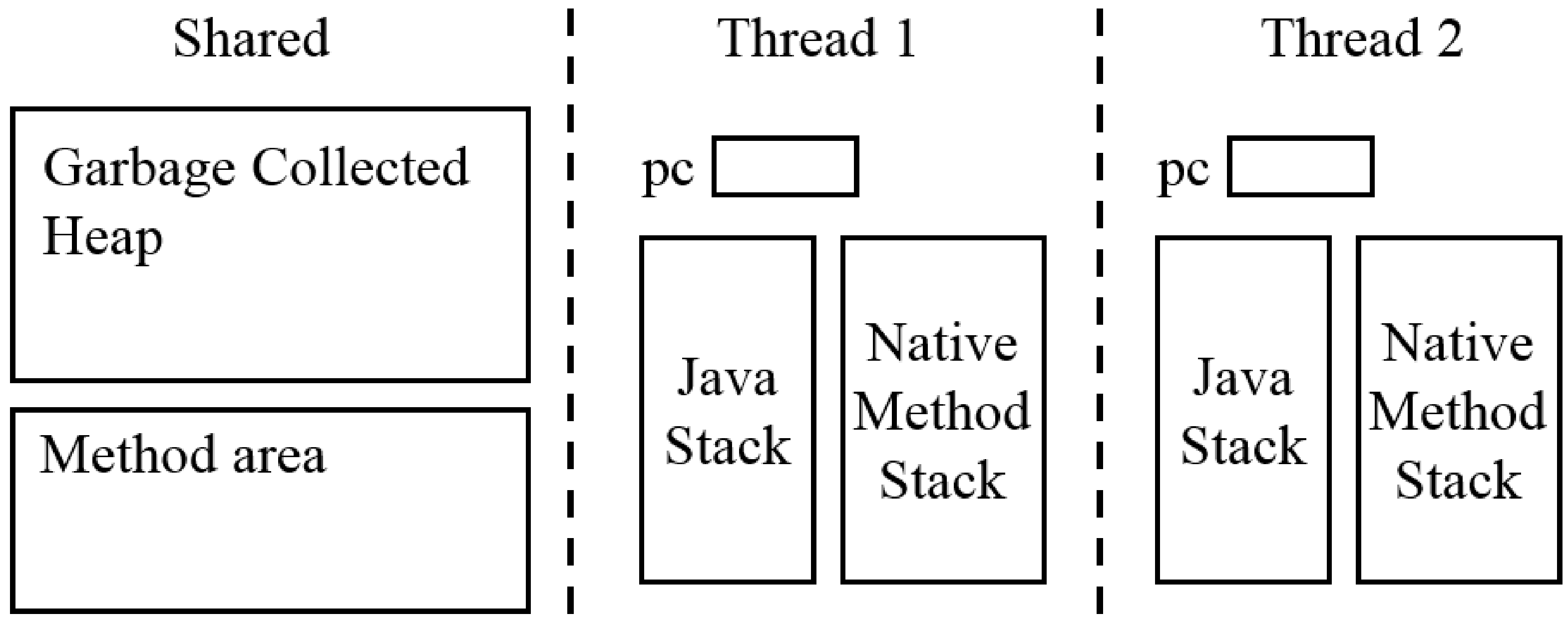
- a stack
  (used for function call frames);
- a heap
  (used for dynamically allocated memory);
- a constant pool
  (used for constant data that can be shared); and
- a code segment
  (used to store JVM instructions of currently loaded class files).

JVM: Runtime Data Areas

Besides OO concepts, JVM also supports multi-threading. Threads are directly supported by the JVM.

=> Two kinds of runtime data areas:

1. shared between all threads
2. private to a single thread



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:

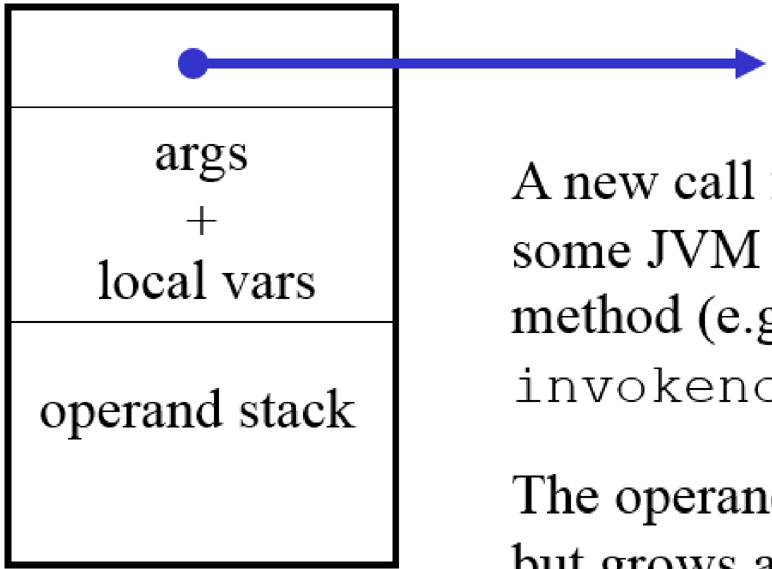
- a reference to the current object (`this`);
- the method arguments;
- the local variables; and
- a local stack used for intermediate results.

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

Stack Frames

The Java stack consists of frames. The JVM specification does not say exactly how the stack and frames should be implemented.

The JVM specification specifies that a stack frame has areas for:



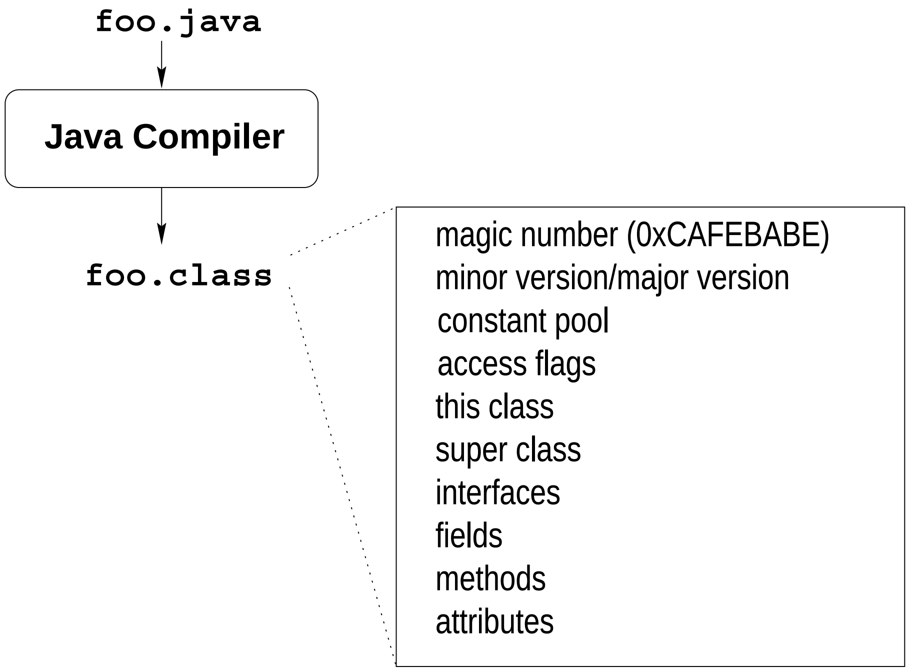
Pointer to runtime constant pool

A new call frame is created by executing some JVM instruction for invoking a method (e.g. `invokevirtual`, `invokenonvirtual`, ...)

The operand stack is initially empty, but grows and shrinks during execution.

Java compilers translate source code to class files.

Class files include the bytecode instructions for each method.



Class Loading

- Classes are loaded lazily when first accessed
- Class name must match file name
- Super classes are loaded first (transitively)
- The bytecode is verified
- Static fields are allocated and given default values
- Static initializers are executed.

Class Loaders

- A *class loader* is an object responsible for loading classes.
- Each class loader is an instance of the abstract class `java.lang.ClassLoader`.
- Every class object contains a reference to the ClassLoader that defined it.
- Each class loader has a parent class loader
  - First try parent class loader if class is requested
  - There is a bootstrap class loader which is the root of the classloader hierarchy.
- Class loaders provide a powerful extension mechanism in Java
  - Loading classes from other sources
  - Transforming classes during loading

Data Types

JVM (and Java) distinguishes between two kinds of types:

Primitive types:

- boolean: `boolean`
- numeric integral: `byte, short, int, long, char`
- numeric floating point: `float, double`
- internal, for exception handling: `returnAddress`

Reference types:

- class types
- array types
- interface types

Note: Primitive types are represented directly, reference types are represented indirectly (as pointers to array or class instances).

Instruction set: kinds of operands

JVM instructions have three kinds of operands:

- from the top of the operand stack
- from the bytes following the opCode
- part of the opCode itself

Each instruction may have different "forms" supporting different kinds of operands.

Example: different forms of "iload"

Assembly code	Binary instruction code layout		
`iload_0`	26		
`iload_1`	27		
`iload_2`	28		
`iload_3`	29		
`iload` *n*	21	*n*	
`wide iload` *n*	196	21	*n*

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;
  }
```

The JVM has 256 instructions for:

- arithmetic operations
- branch operations
- constant loading operations
- local operations
- stack operations
- class operations
- method operations

The JVM specification gives the full list

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]

(note that new only allocates space, must call <init> to execute bo

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?
ANSWER: <init>(..), private, super method calls
Also, `invokestatic` and `invokeinterface`.

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.

Bytecode Verification: Syntax

- The first 4 bytes of a class file must contain the magic number 0xCAFEBABE.
- The bytecodes must be syntactically correct.
  - Branch targets are within the code segment
  - Only legal offsets are referenced
  - Constants have appropriate types
  - All instructions are complete
  - Execution cannot fall of the end of the code

Interesting properties of verified bytecode:

- at any program point, the stack is the same size along all execution paths;
- each instruction must be executed with the correct number and types of arguments on the stack, and in locals (on 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.
- fields are assigned appropriate values

Verification: Gotcha

```
.method public static main([Ljava/lang/String;)V
.throws java/lang/Exception
.limit stack 2
.limit locals 1
ldc -21248564
invokevirtual java/io/InputStream/read()I
return
```

```
java Fake

Exception in thread "main" java.lang.VerifyError:
(class: Fake, method: main signature: ([Ljava/lang/String;)V)
Expecting to find object/array on stack
```

Slides of this format from: `http://cs.au.dk/~mis/dOvs/slides/39a-javavirtualmachine.pdf`

Verification: Gotcha Again

```
.method public static main([Ljava/lang/String;)V
.throws java/lang/Exception
.limit stack 2
.limit locals 2
iload_1
return
```

```
java Fake

Exception in thread "main" java.lang.VerifyError:
(class: Fake, method: main signature: ([Ljava/lang/String;)V)
Accessing value from uninitialized register 1
```

Verification: Gotcha Once More

```
ifeq A
ldc 42
goto B
A:
ldc "fortytwo"
B:
```

```
java Fake

Exception in thread "main" java.lang.VerifyError:
(class: Fake, method: main signature: ([Ljava/lang/String;)V
Mismatched stack types
```

Verification: Gonna Getcha Every Time

```
A:
iconst_5
goto A
```

```
java Fake

Exception in thread "main" java.lang.VerifyError:
(class: Fake, method: main signature: ([Ljava/lang/String;)V
Inconsistent stack height 1 != 0
```

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 a lot of research and industrial effort has been put into good adaptive JIT compilers.

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 produces 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.

Let’s Practice!"

Write virtualRISC code for the following function. Assume that `x` is in `R0` and `n` is in `R1` on input, and that the result should be returned in `R0`.

```
int power1(int x, int n)
{ int i;
  int prod = 1;
  for (i=0; i<n; i++)
   prod = prod * (x+1);
  return(prod);
}
```

You can also assume that the variables are mapped to following spots in the stack frame. You can use registers only if you like.

```
Parameters:  x ->  [fp+68]        n ->  [fp+72]
Locals:      i ->  [fp-12]        prod -> [fp-16]
```

Try, `gcc -S power1.c` and `gcc -O -S power1.c`, and compare the difference.

Possible VirtualRISC code with loop invariant expression removed

```
_power1:
  save sp,-112,sp     // save stack frame
  st R0,[fp+68]       // save input args x, n in frame of CALLER
  st R1,[fp+72]       // R0 holds x,  R1 holds n

  mov 1,R2            // R2 :=1, R2 holds prod
  add R0,1,R4         // R4 := x + 1, loop invariant
  mov 0,R3            // R3 := 0,  R3 holds i
begin_loop:
  cmp R3,R1           // if (i < n)
  bge end_loop
begin_body:
  mul R2,R4,R2        // prod = prod * (x+1)
  add R3,1,R3         // i = i + 1
  goto begin_loop
end_loop:
  mov R2, R0          // put return value of prod into R0
  restore             // restore register window
  ret                 // return from function
```

Now for some bytecode Write the Java bytecode version of the static method.

```
public class p1{
  static int power1(int x, int n)
  { int i;
    int prod = 1;
    for (i=0; i<n; i++)
      prod = prod * (x+1);
    return(prod);
  }
}
```

You can assume the following mapping of variables to bytecode locals:

```
Parameters:  x ->  local 0      n -> local 1
Locals:      i ->  local 2      prod -> local 3
```

What is the “baby" stack limit?

Try: `javac p1.java`, `javap -verbose p1.class`

Jasmin code for power1

```
.method static power1(II)I
  .limit stack 3
  .limit locals 4

    Label2:
       0: iconst_1
       1: istore_3 ; prod = 1

       2: iconst_0
       3: istore_2 ; i = 0;

    Label1:
       4: iload_2
       5: iload_1
       6: if_icmpge Label0 ; (i >= n)?

       9: iload_3
      10: iload_0
      11: iconst_1 ; high water mark for baby stack, 3
      12: iadd
      13: imul
      14: istore_3  ; prod = prod * (x + 1)

      15: iinc 2 1  ; i++
      18: goto Label1

    Label0:
      21: iload_3
      22: ireturn ; return(prod)
.end method
```

Jasmin code for power1, with loop invariant removal

```
.method static power1(II)I
    .limit stack 2
    .limit locals 5

    Label2:
       0: iconst_1
       1: istore_3 ; prod = 1

       2: iload_0
       3: iconst_1
       4: iadd
       5: istore 4 ; t = x + 1

       7: iconst_0
       8: istore_2 ; i = 0

    Label1:
       9: iload_2
      10: iload_1
      11: if_icmpge Label0 (i >= n)?

      14: iload_3
      15: iload 4
      17: imul
      18: istore_3 ; prod = prod * t;

      19: iinc 2 1
      22: goto Label1

    Label0:
      25: iload_3
      26: ireturn ; return (prod)
.end method
```

A useful tool for dealing with class files, tinapoc

`http://sourceforge.net/projects/tinapoc/` supports several tools including:

```
> java dejasmin Test.class
will disassemble Test.class and produce Jasmin output

> java dejasmin --warmode Test.class > Test.dump
will dump the WHOLE content of the class in Test.dump
See dejasmin documentation for more details.

> java jasmin test.j
assembles test.j. See Jasmin documentation for more details.
```

Some useful scripts (Windows cygwin versions)

```
#! /bin/tcsh
# jas - java jasmin
setenv TOOLDIR "/cygdrive/c/Users/Laurie/Documents/Courses/520/Java"
java -classpath `cygpath -wp $TOOLDIR/tinapoc.jar:$TOOLDIR/bcel-5.1.jar`
  jasmin $*
```

```
#!/bin/csh
# djas - java dejasmin
setenv TOOLDIR "/cygdrive/c/Users/Laurie/Documents/Courses/520/Java"
java -classpath `cygpath -wp $TOOLDIR/tinapoc.jar:$TOOLDIR/bcel-5.1.jar`
  dejasmin $*
```

Try: `djas p1.class > p1.j` and `jas p1.j`
`djas -warmode p1.class > p1.dump`

Let's consider this mystery program

```
public class u1 {

  public static void main(String [] args)
    { int r = prod(4);
      System.out.println(r);
    }

  static int prod(int n)
    { ... written only in bytecode ...
    }
}
```

Now in bytecode (jasmin source)

```
.class public ul
.super java/lang/Object

.method public <init>()V
   .limit stack 1
   .limit locals 1
   Label0:
      aload_0
      invokespecial java/lang/Object/<init>()V
   Label1:
      return
.end method

.method public static main([Ljava/lang/String;)V
   .limit stack 2
   .limit locals 2
   Label0:
   ldc 4
   invokestatic ul/prod(I)I
   istore_1
   getstatic java.lang.System.out Ljava/io/PrintStream;
   iload_1
   invokevirtual java/io/PrintStream/println(I)V
   Label1:
   return
.end method
```

What does this method do?

```
.method static prod(I)I
   .limit stack 5
   .limit locals 2
   Begin:
     iconst_1
     istore_1
   PushLoop:
     iload_1
     iinc 1 1
     iload_1
     iload_0
     if_icmple PushLoop
     iconst_1
     istore_1
    PopLoop:
     imul
     iinc 1 1
     iload_1
     iload_0
     if_icmplt PopLoop
     ireturn
.end method
```

Try `java -noverify ul` and `java ul`.

BUT .... is stack code really suitable for optimizations and transformations?

No, tools like Soot are useful for this.

`http://sable.github.io/soot/`, use the nightly build.

A useful script,

```
#! /bin/tcsh
# soot with the soot classpath set
setenv TOOLDIR "/cygdrive/c/Users/Laurie/Documents/Courses/520/Java"
java -jar `cygpath -wp $TOOLDIR/soot-trunk.jar` -cp `cygpath -wp
$TOOLDIR/rt.jar:$TOOLDIR/jce.jar:.` $*
```

Let's look at the power1 example again...

```
public class p1 {

  public static void main(String [] args)
  { int r = power1(10,2);
    System.out.println(r);
  }

  static int power1(int x, int n)
  { int i;
    int prod = 1;
    for (i=0; i<n; i++)
     prod = prod * (x+1);
    return(prod);
  }
}
```

Using Soot to create Jimple: `soot -f jimple p1`

Creates file `sootOutput/p1.jimple`.

```
public class p1 extends java.lang.Object
{ public void <init>()
    { p1 r0;
      r0 := @this: p1;
      specialinvoke r0.<java.lang.Object: void <init>()>();
      return;
    }

  public static void main(java.lang.String[])
    { java.lang.String[] r0;
      int i0;
      java.io.PrintStream $r1;
      r0 := @parameter0: java.lang.String[];
      i0 = staticinvoke <p1: int power1(int,int)>(10, 2);
      $r1 = <java.lang.System: java.io.PrintStream out>;
      virtualinvoke $r1.<java.io.PrintStream: void println(int)>(i0);
      return;
    }
...
```

```
  ...
  static int power1(int, int)
  { int i0, i1, i2, i3, $i4;
    i0 := @parameter0: int;
    i1 := @parameter1: int;
    i3 = 1;
    i2 = 0;
  label1:
    if i2 >= i1 goto label2;
    $i4 = i0 + 1;
    i3 = i3 * $i4;
    i2 = i2 + 1;
    goto label1;
  label2:
     return i3;
    }
}
```

You can also decompile .class files to .java

Try `soot -f dava -p db.renamer enabled:true p1`

```
import java.io.PrintStream;

public class p1 {
 public static void main(String[] args)
   { int i0;
     i0 = p1.power1(10, 2);
     System.out.println(i0);
   }

   static int power1(int i0, int i1)
   { int i, i3;
     i3 = 1;
     for (i = 0; i < i1; i++)
       { i3 = i3 * (i0 + 1);
       }
     return i3;
   }
}
```

EOE