

# Virtual Machines

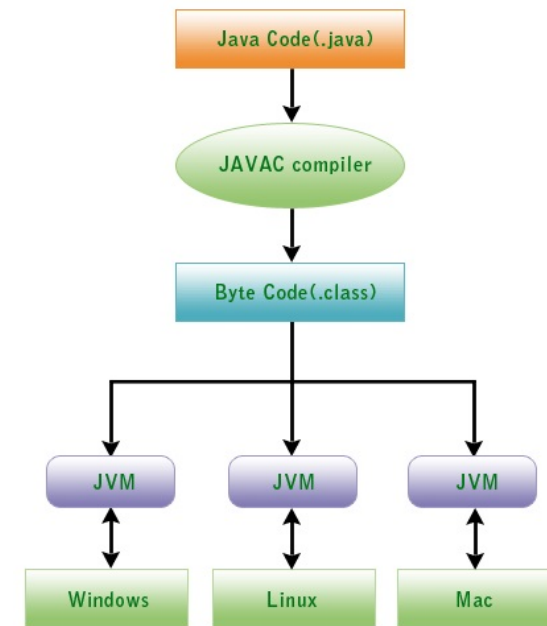
COMP 520: Compiler Design (4 credits)

Alexander Krolik

`alexander.krolik@mail.mcgill.ca`

MWF 9:30-10:30, TR 1080

<http://www.cs.mcgill.ca/~cs520/2018/>



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

# Announcements (Wednesday, February 7th)

## Milestones

- Group signup form <https://goo.gl/forms/L6Dq5CHLvbjNhT8w1>

## Assignment 2

- Any questions?
- **Due:** Sunday, February 11th 11:59 PM

## Midterm

- Friday, March 16th, 1.5 hour “in class” midterm
- **Option #1:** 9:00-10:30 TR 2100
- **Option #2:** 9:30-11:00 ARTS W-20

# Readings

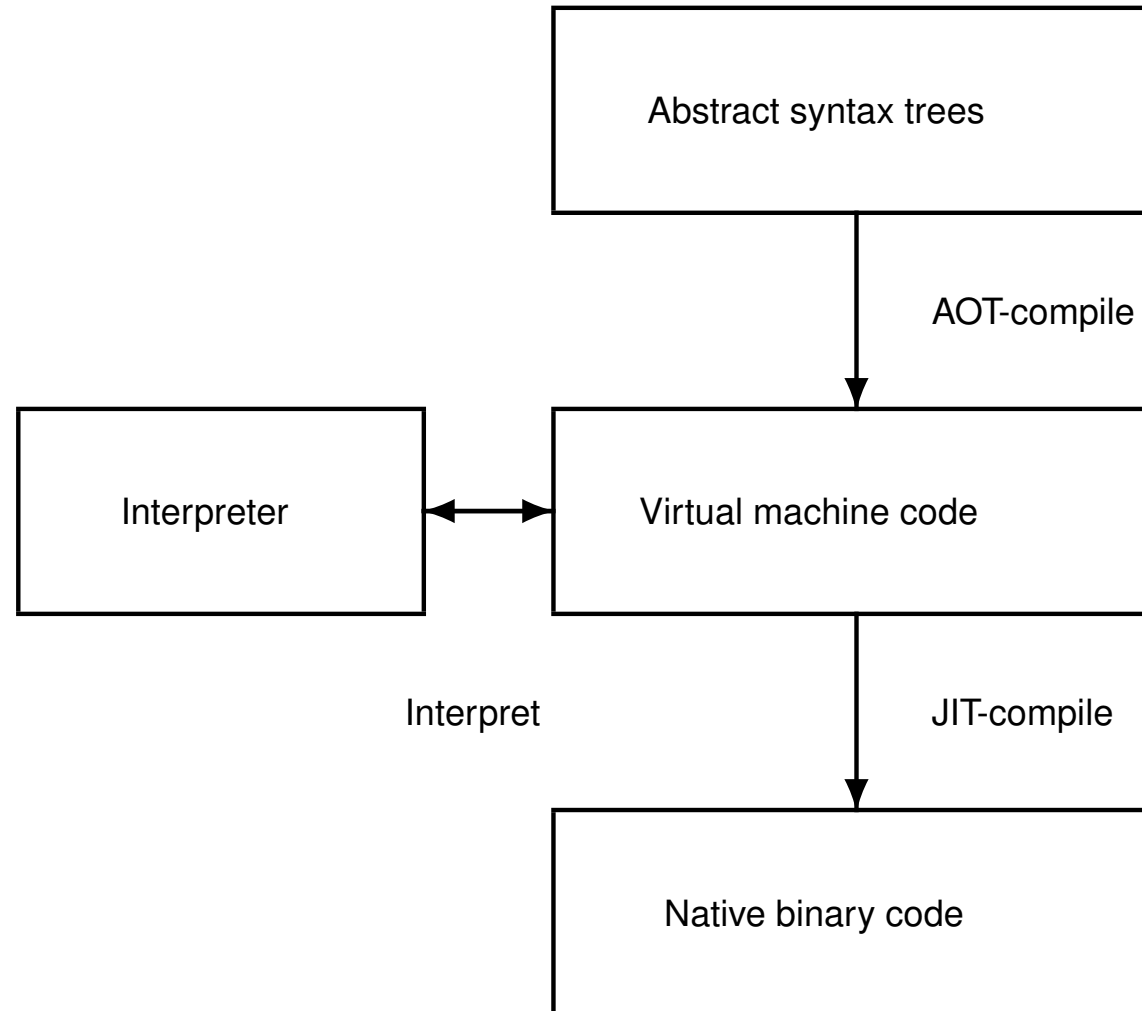
## Crafting a Compiler (recommended)

- Chapter 10.1-10.2
- Chapter 11

## Optional

- JVM specification: <http://docs.oracle.com/javase/specs/jvms/se7/html/>

# Compilation and Execution in Virtual Machines



# Ahead-of-Time (AOT) Compilation

Compilers traditionally transformed source code to machine code ahead-of-time (before execution)

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

## Advantages

- Fast execution, since the code is already ready to be executed;
- The code can exploit many details of the underlying architecture (given a smart compiler); and
- Intermediate languages like RTL facilitate production of code generators for many target architectures.

## Disadvantages

- Runtime information (program or architecture) is ignored;
- A code generator must be built for each target architecture in the compiler.

# Interpreting Virtual Machine Code

Alternatively, code can be interpreted – instructions read one at a time and executed in a “virtual” environment. The code *is not* compiled to the target architecture.

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

## Advantages

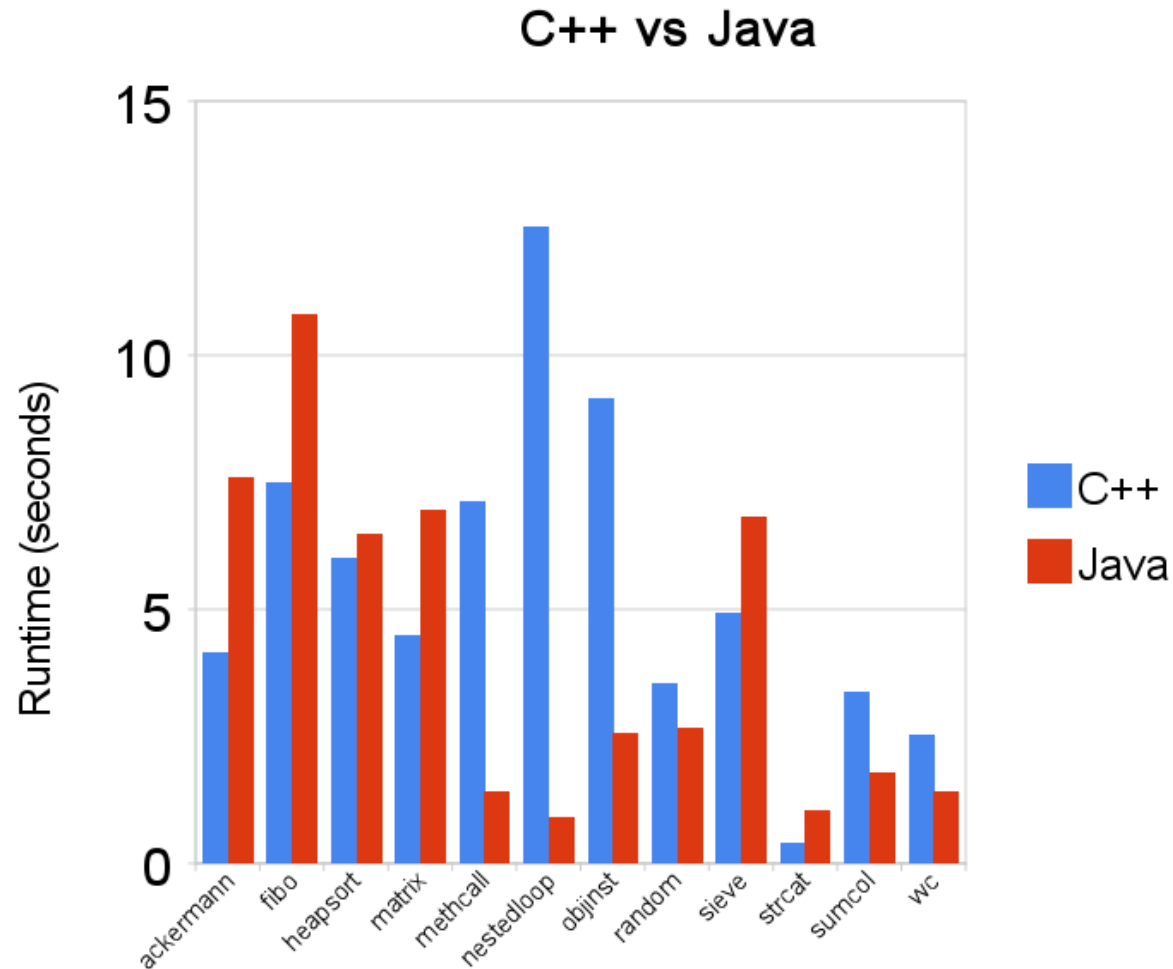
- Easy to generate virtual machine code;
- The code is architecture independent; and
- Bytecode can be more compact (macro operations).

## Disadvantages

- Poor performance due to interpretative overhead (typically 5-20 × slower)
  - Every instruction considered in isolation;
  - Confuses branch prediction; and . . .

# Interpreters vs Compilers

But, modern Java is quite efficient – virtual machine code can also be JIT compiled!



# Virtual Machines

In this class we will look at two different virtual machines

**VirtualRISC:** register-based IR

**Java Virtual Machine:** stack-based IR



# VirtualRISC

VirtualRISC is a simple RISC machine (similar to what you've seen in COMP 273)

- Memory;
- Registers;
- Condition codes; and
- Execution unit.

## **In this model we ignore**

- Caches;
- Pipelines;
- Branch prediction units; and
- Advanced features.

We focus instead on the basic architecture of register-based machines.

# VirtualRISC Memory

VirtualRISC has several types of memory for storing program information

- 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

VirtualRISC has general purpose registers used for computation, and special registers that are managed by the machine

- Unbounded number of general purpose registers  $R_i$ ;
- Stack pointer ( $sp$ ) which points to the top of the stack;
- Frame pointer ( $fp$ ) which points to the current stack frame; and
- Program counter ( $pc$ ) which points to the current instruction.

# VirtualRISC Execution

## Condition codes

- Condition codes are set by instructions which evaluate a predicate (i.e. comparisons); and
- Are used for branching instructions.

## 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  $pc$ .

# VirtualRISC Program

A VirtualRISC program consists of a list of instructions and labels

## Instruction types

- Moves between registers and memory;
- Mathematical operations;
- Comparisons;
- Branches; or
- Other, special instructions.

Operands to instructions can either be memory addresses, registers, or constants.

# Memory Move Instructions

[ . . ] indicates the memory location stored in the register

## Store

Store instructions copy the contents from a register to a memory location: `st <src>, <dst>`

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

## Load

Load instructions copy the contents from a memory location to a register: `ld <src>, <dst>`

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

## Move

The last move instruction `mov <src>, <dst>` copies the contents between registers. The source register may also be replaced by a constant (i.e. `mov 5, R1`)

```
mov Ri, Rj           Rj := Ri
```

# Mathematical Operations

Mathematical operations are performed between two source registers and stored in a destination register

$$\text{op } \langle \text{src1} \rangle, \langle \text{src2} \rangle, \langle \text{dst} \rangle$$

The source registers may be replaced by constants (i.e. `add R1, 5, R2`)

|                             |                    |
|-----------------------------|--------------------|
| <code>add Ri, Rj, Rk</code> | $R_k := R_i + R_j$ |
|-----------------------------|--------------------|

|                             |                    |
|-----------------------------|--------------------|
| <code>sub Ri, Rj, Rk</code> | $R_k := R_i - R_j$ |
|-----------------------------|--------------------|

|                             |                    |
|-----------------------------|--------------------|
| <code>mul Ri, Rj, Rk</code> | $R_k := R_i * R_j$ |
|-----------------------------|--------------------|

|                             |                    |
|-----------------------------|--------------------|
| <code>div Ri, Rj, Rk</code> | $R_k := R_i / R_j$ |
|-----------------------------|--------------------|

# Branching Instructions

The `cmp` instruction sets the condition codes depending on the relation between its operands

```
cmp Ri, Rj
```

Just like the mathematical operators, constants may be used as operands.

## Branching instructions

Depending on the condition codes, the branch operation may/may not be executed

```
b L
```

```
bg L
```

```
bge L
```

```
bl L
```

```
ble L
```

```
bne L
```

To express `if R1 <= 0 goto L1` we write

```
cmp R1, 0
```

```
ble L1
```

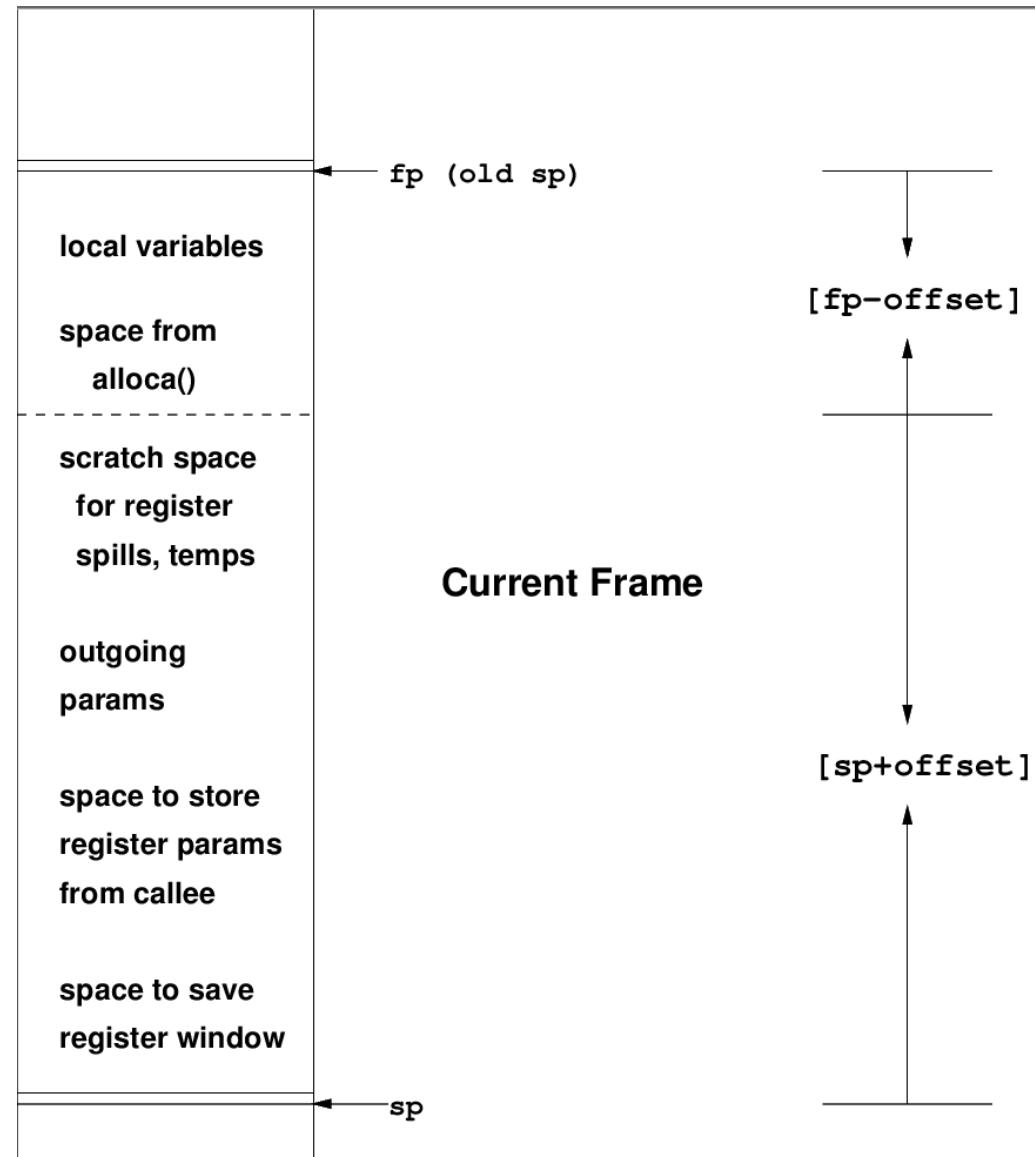


## Other Special Instructions

VirtualRISC also has the following special instructions for managing the stack with function calls

|                              |  |
|------------------------------|--|
| <code>call L</code>          | <code>R15:=pc; pc:=L</code>  |
| <code>save sp, -C, sp</code> | <code>save registers,</code><br><code>allocating C bytes</code><br><code>on the stack</code> |
| <code>restore</code>         | <code>restore registers</code>   |
| <code>ret</code>             | <code>pc:=R15+8</code>   |
| <code>nop</code>             | <code>do nothing</code>  |

# Stack Frame



# Stack Frames

- Store the function call hierarchy and the respective program memory;
- `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.

# Calling semantics

## Calling

- Functions start by allocating the stack frame using `save sp, -C, sp`; and
- Functions end by restoring the previous stack frame and register window (`restore`) and returning (`ret`).

## Parameters

- Passed in registers `R0`, `R1`, etc; and
- May be stored in memory. By convention we use  $f_{p+68+4k}$  where  $k$  is some non-negative integer.  
Note that this means we are storing parameters in the *callers* frame!

## Local variables

- Use any general purpose register; and
- May be stored in memory. By convention we use  $f_{p-4k}$  where  $k$  is some non-zero integer

## Writing VirtualRISC Code

Write the following C code in VirtualRISC. Try using no register allocation scheme - this means that values should be loaded into registers directly before operations and the value stored back to memory immediately.

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

# Writing 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           // R0 := 2
    st R0, [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

```

# Fibonacci

More practice! Write the following C program in VirtualRISC

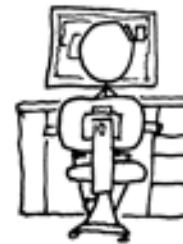
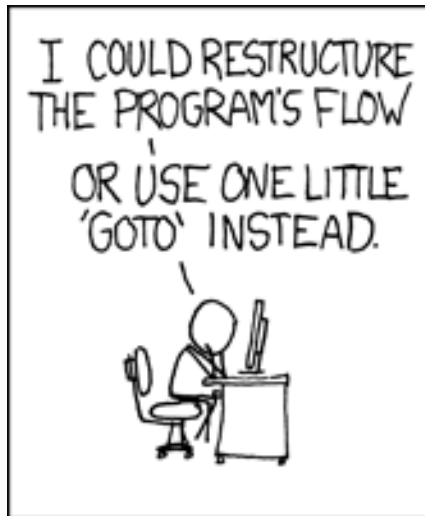
```
int fib(int x) {
    int current, last, sum;

    current = 1;
    last = 1;
    sum = 1;

    x = x - 2;
    while (x > 0) {
        sum = current + last;
        last = current;
        current = sum;
        x = x - 1;
    }

    return sum;
}
```

# Goto!





# What does this go?

Try writing the VirtualRISC code

```
int thing(int a, int b) {
    int temp, iter;

    while (1) {
        temp = a;
        iter = 0;

        while (iter - b) {
            temp = temp + 1;

            if (temp - b) {
                iter = iter + 1;
            } else {
                goto ret;
            }
        }

        a = a - b;
    }
ret:
    return a;
}
```

# Announcements (Friday, February 9th)

## Milestones

- Group signup form <https://goo.gl/forms/L6Dq5CHLvbjNhT8w1>
- Project milestone 1 out on *Monday*

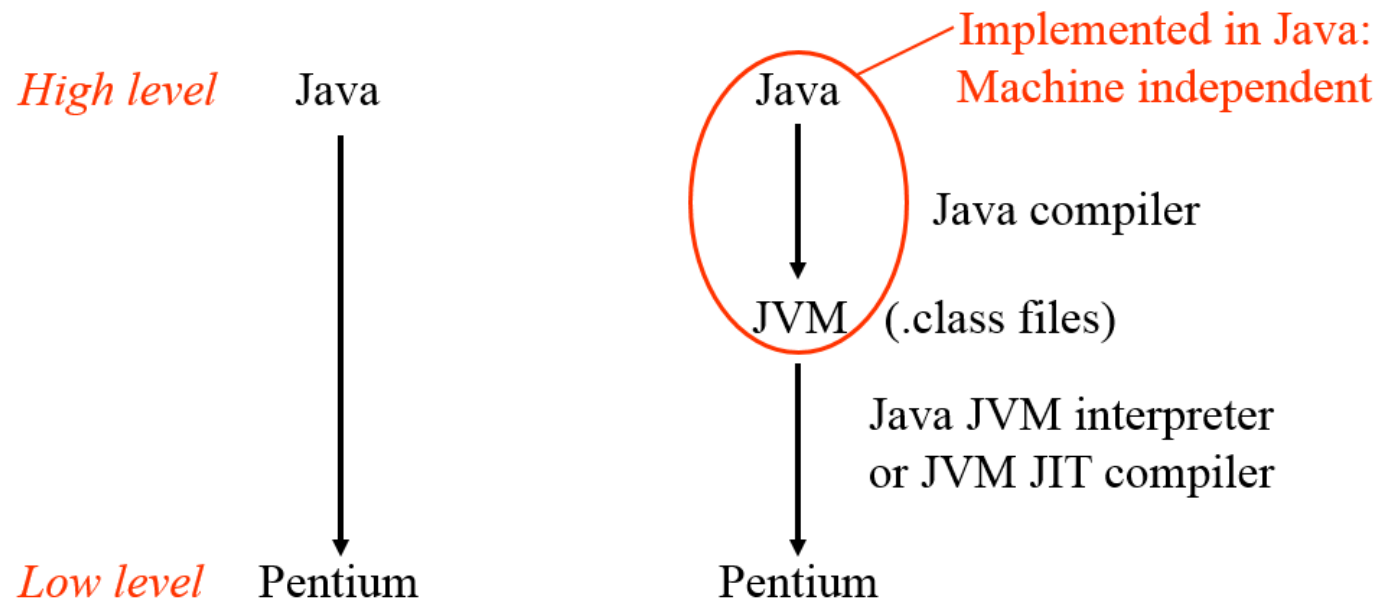
## Assignment 2

- Any questions?
- How's everyone doing? Any extension needed? **Yes**
- **Extended Due Date:** Tuesday, February 13th 11:59 PM

# Java Virtual Machine

## 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)

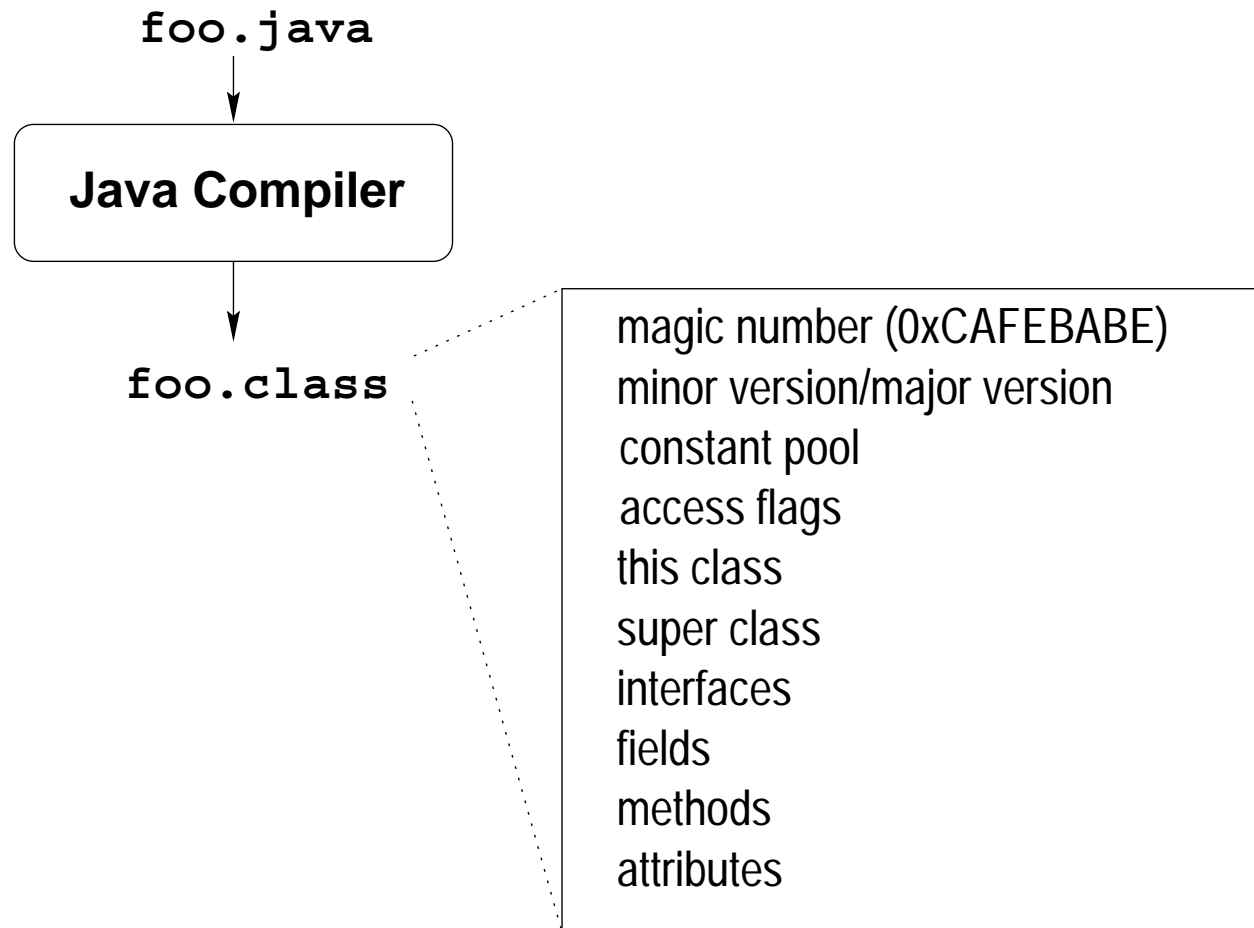


The Java Virtual Machine

Note: slides of this format from [http://cs434.cs.ua.edu/Classes/20\\_JVM.ppt](http://cs434.cs.ua.edu/Classes/20_JVM.ppt)

# Java Compilers

Java compilers like `javac` translate source code to class files. Class files include the bytecode instructions for each method.



# Java Class Loading

To execute a Java program, classes must first be loaded into the virtual machine

1. Classes are loaded lazily when first accessed;
2. Class name must match file name;
3. Super classes are loaded first (transitively);
4. The bytecode is verified;
5. Static fields are allocated and given default values; and
6. Static initializers are executed.

# Java 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; and
  - 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; and
  - Transforming classes during loading.

# Java Virtual Machine

The JVM is a *stack machine* which has the same overall components as VirtualRISC

- Memory;
- Registers;
- Condition codes; and
- Execution unit.

# Java Virtual Machine Memory

The JVM has several types of memory for storing program information

- 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 Stack Frames

The Java Virtual Machine has two types of stacks

- Call stack: function call frames; and
- *Baby/operand/local* stack: operands and results from instructions.

Each function call frame contains

- A reference to the constant pool;
- A reference to the current object (`this`);
- The method arguments;
- The local variables; and
- A local stack used for intermediate results (the baby stack).

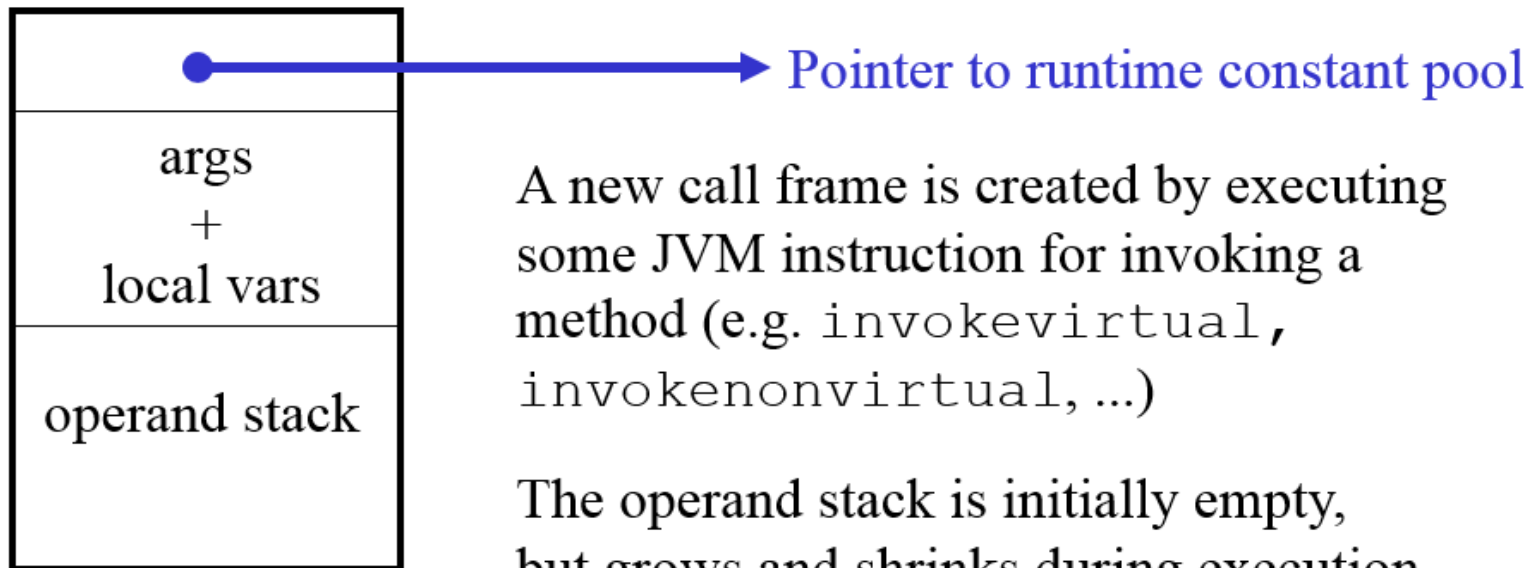
To compute the correct frame size, the number of local slots and the maximum size of the local stack are fixed at compile-time.

# 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:



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 Virtual Machine Execution

## Condition codes

- Condition codes are set by instructions that evaluate predicates; and
- Are used for branching instructions.

Unlike VirtualRISC, the JVM instruction set does not differentiate between these two operations.

## 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  $pc$ .

---

# 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).

# Jasmin Code

Jasmin is the textual representation of Java bytecode that we will study (and write!) in class

## Types in jasmin

- `boolean: Z`
- `float: F`
- `int: I`
- `long: J`
- `void: V`

## Reference types (classes)

- Types are given as their fully qualified names;
- i.e. `String` in the package `java.lang` has fully qualified name `java.lang.String`;
- In Jasmin code, we replace “.” by “/”;
- i.e. `String` is written as `Ljava/lang/String`.

# Writing Jasmin Code - Methods

In Jasmin code, a method consists of

- Signature

```
.method <modifiers> <name> (<parameter types>) <return type>
```

- Height of the “baby” stack

```
.limit stack <limit>
```

- Number of locals (including explicit and implicit arguments)

```
.limit locals <limit>
```

- Method body

- Termination line

```
.end method
```

## Example

```
.method public Abs(I) I  
.limit stack 2  
.limit locals 2  
[...]  
.end method
```

## Example Jasmin

Consider the following Java method for computing the absolute value of an integer

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

Write the corresponding bytecode in Jasmin syntax



## Example Jasmin

### Corresponding Jasmin codes

```

.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 Else // [ o -3 ] [ * * ]
    iload_1 // [ o -3 ] [ -3 * ]
    iconst_m1 // [ o -3 ] [ -3 -1 ]
    imul // [ o -3 ] [ 3 * ]
    ireturn // [ o -3 ] [ * * ]
Else:
    iload_1
    ireturn
.end method

```

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

# 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;
}
```

## 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 |          |          |
|---------------------------|--------------------------------|----------|----------|
| <code>iload_0</code>      | 26                             |          |          |
| <code>iload_1</code>      | 27                             |          |          |
| <code>iload_2</code>      | 28                             |          |          |
| <code>iload_3</code>      | 29                             |          |          |
| <code>iload n</code>      | 21                             | <i>n</i> |          |
| <code>wide iload n</code> | 196                            | 21       | <i>n</i> |

---

# Java Virtual Machine Instructions

The JVM has 256 instructions for

- Arithmetic operations
- Constant loading operations
- Local operations
- Branch operations
- Stack operations
- Class operations
- Method operations

The JVM specification gives the full list

# Arithmetic Operations

Arithmetic operations use operands from the stack, and store the result back on the stack

## 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 (stack not used)

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

# Constant Loading Operations

Constant loading instructions push constant values onto the top of the stack

```
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

Locals operations load and store values on the stack from the local variables

```
iload k          [...] -> [...:local[k]]
```

```
istore k         [...:i] -> [...]
```

```
local[k] = i
```

```
aload k          [...] -> [...:local[k]]
```

```
astore k         [...:o] -> [...]
```

```
local[k] = o
```

## Field operations

Field operations get and put elements on the stack into fields of an object

```
getfield f sig   [...:o] -> [...:o.f]
```

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

```
o.f = v
```

Note that these instructions require the full name of the field (`Class.field`) and its signature (type)

# Branch Operations

## Nullary branch operations

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

## Unary branch operations

Unary branch instructions compare the top of the stack against zero

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

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

There are also other comparators `ifgt`, `ifge`, `iflt`, `ifle` for unary branching

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

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



# Branch Operations

## Binary branch operations

Binary branch instructions compare the top two elements on the stack against each other

```
if_icmpeq L    [...:i1:i2] -> [...]
```

```
branch if i1 == i2
```

```
if_icmpne L    [...:i1:i2] -> [...]
```

```
branch if i1 != i2
```

There are also other comparators `if_icmpgt`, `if_icmpge`, `if_icmplt`, `if_icmple` for binary branching

```
if_acmpeq L    [...:o1:o2] -> [...]
```

```
branch if o1 == o2
```

```
if_acmpne L    [...:o1:o2] -> [...]
```

```
branch if o1 != o2
```

# Stack Operations

Stack instructions are value agnostic operations that change the state of the stack

dup            [ ... : v1 ] -> [ ... : v1 : v1 ]

pop            [ ... : v1 ] -> [ ... ]

swap           [ ... : v1 : v2 ] -> [ ... : v2 : v1 ]

nop            [ ... ] -> [ ... ]

# Announcements (Monday, February 12th)

## Project

- Milestone 1 out Today!

## Assignment 2

- Any last minute questions?
- **Due:** Tuesday, February 13th 11:59 PM

# Announcements (Wednesday, February 14th)

## Milestones

- Assignment 2 will be graded by around Sunday (or so)

## Milestone 1

- You should all have access to the GitHub repos
- Get started early!
- **Due:** Sunday, February 25th 11:59 PM

## Class Operations

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

The `new` instruction by itself only allocates space on the heap. To execute the constructor and initialize the object, you must call `<init>` using `invokespecial` and the appropriate parameters

```
invokespecial C/<init>()V      [...:o] -> [...]
```

### Class properties of an object

```
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

Methods are invoked using an `invokevirtual` instruction

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

### Internally

Invoking methods consists of selecting the appropriate method, setting up the stack frame and locals, and jumping to the body

```
// Overloading is 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 0 points to "this"  
local[1] = a1;  
...  
local[n] = an;
```

```
pc = block.code;
```

## Method Operations

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

### Internally

```
// Overloading is 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 0 points to "this"
local[1] = a1;
...
local[n] = an;
```

```
pc = block.code;
```

For which method calls is `invokespecial` used? `<init>(..)`, `private`, `super` method calls.

`invokevirtual` uses the class of the object itself, whereas `invokespecial` calls a specific class in the hierarchy. There are also bytecode instructions `invokestatic` and `invokeinterface`

## Method Operations

Return operations can either take (a) a single element; or (b) no elements.

```
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.



## Example Java Method

Consider the following Java method from the `Cons` class

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

Write the corresponding Java bytecode in Jasmin syntax

# Example Java Method

## Corresponding bytecode (in Jasmin syntax)

```
.method public member(Ljava/lang/Object;)Z
.limit locals 2           // local[0] = o
                          // local[1] = item
.limit stack 2           // [ * * ]
  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 0xCAFEBAFE;
- 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; and
  - Execution cannot fall off the end of the code.

# Bytecode Verification - Interesting Properties

Given a class 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;
- No local variable can be accessed before it has been assigned a value; and
- 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
```

# 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; and
- 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;
- 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; which
  - Speeds up construction of the “proof tree”  $\Rightarrow$  also called “Proof-Carrying Code”.
- Java 7 (version 51.0 bytecodes) JVMs enforce presence of these attributes.

## 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 jasmin test.j
```

assembles test.j written in Jasmin code. See Jasmin documentation for mo

*Add -classpath tinapoc.jar:bcel-5.1.jar with the appropriate paths*

## javap

The Java provided tool `javap` also provides disassembly support including the constant pool

```
> javap -c Test.class
```

will disassemble Text.class and produce Jasmin output

## Consider the Following 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 Jasmin Code

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

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

.method public static main([Ljava/lang/String;)V
.limit stack 2
.limit locals 2
    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
    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`

# Stack Code for Optimization

Is stack code really suitable for optimizations and transformations?

**No**, tools like Soot are useful for this: <http://sable.github.io/soot/>

Optimizing stack based intermediate representations requires

- Reasoning and maintaining information about the stack (which changes height); and
- Does not correspond to actual execution!



# Power1 Example

```
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;
    }
}
```

# Power1 Example

Using Soot to create Jimple 3 address code (`soot -f jimple p1`)

```
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;
}
}
```

# Decompiling Class Files

Soot can also decompile `.class` files to the equivalent `.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;
    }
}
```

Some program information (such as variable names) is lost from the original source

# This Class

## Java bytecode

- The JOOS compiler produces Java bytecode in Jasmin format; and
- The JOOS peephole optimizer transforms bytecode into more efficient bytecode.

## 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

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

## Assumptions

- $x$  is in R0 and  $n$  is in R1 on input;
- The result should be returned in R0; and
- The variables are mapped to following spots in the stack frame.

|             |                 |         |                    |         |
|-------------|-----------------|---------|--------------------|---------|
| Parameters: | $x \rightarrow$ | [fp+68] | $n \rightarrow$    | [fp+72] |
| Locals:     | $i \rightarrow$ | [fp-12] | $prod \rightarrow$ | [fp-16] |

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

## VirtualRISC Code (Loop Invariant Removal)

```
_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
```

## Let's Practice!

Write the Java bytecode version of the static method for computing the power.

```
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
```

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



# Jasmin Code

```
.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 (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 ; i++
 22: goto Label1

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