Virtual Machines

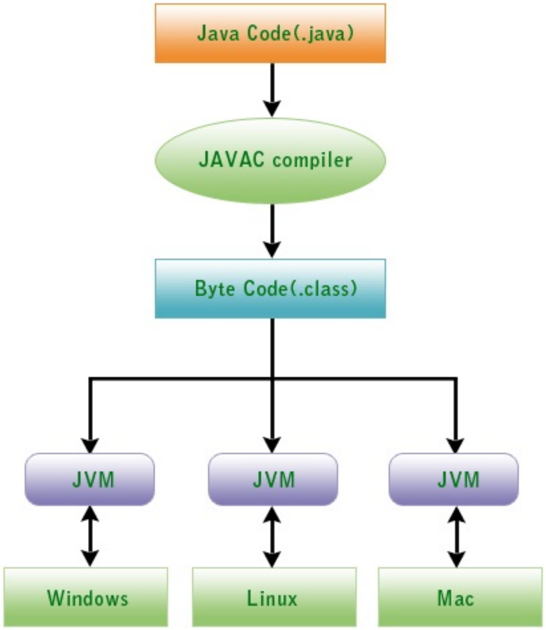
COMP 520: Compiler Design (4 credits)

Alexander Krolik

alexander.krolik@mail.mcgill.ca

MWF 10:30-11:30, TR 1100

http://www.cs.mcgill.ca/~cs520/2020/



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

Readings

Crafting a Compiler (recommended)

- Chapter 10.1-10.2
- Chapter 11

Optional

- JVM specification: `http://docs.oracle.com/javase/specs/jvms/se7/html/`
- The Jalapeño dynamic optimizing compiler for Java:
  `https://dl.acm.org/citation.cfm?id=304113`

Announcements (Wednesday/Friday, February 5th/7th)

Milestones

- Next Monday we will introduce the GoLite project!
- You should have received an invite to the “comp520” organization on GitHub.

Assignment 2

- Any questions? Else if, codegen file
- **Due**: Monday, February 10th 11:59 PM

Virtual Machines

Virtual machines

JVM

Bytecode

Verification

Stack code

Summary

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

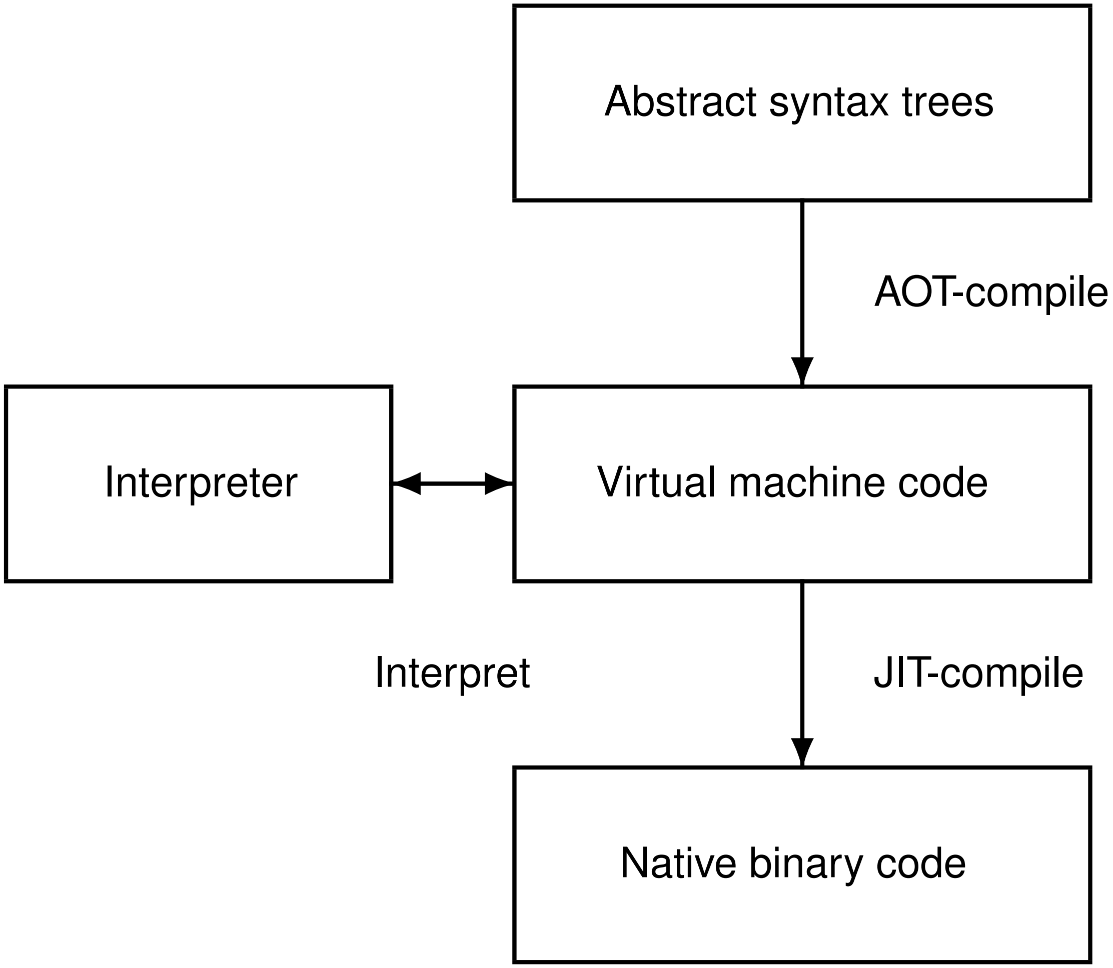
- Fast execution, since the code is already ready to be executed;
- The code can exploit many details of the underlying architecture (given a smart compiler); and
- Intermediate languages like RTL facilitate production of code generators for many target architectures.

Disadvantages

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

Virtual Machines

Programming languages supported by virtual machines delay generating native code (if at all) until execution time. Instead, the AOT compiler produces *virtual machine code*.



Interpreting Virtual Machine Code

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 (in some instances).

Advantages

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

Disadvantages

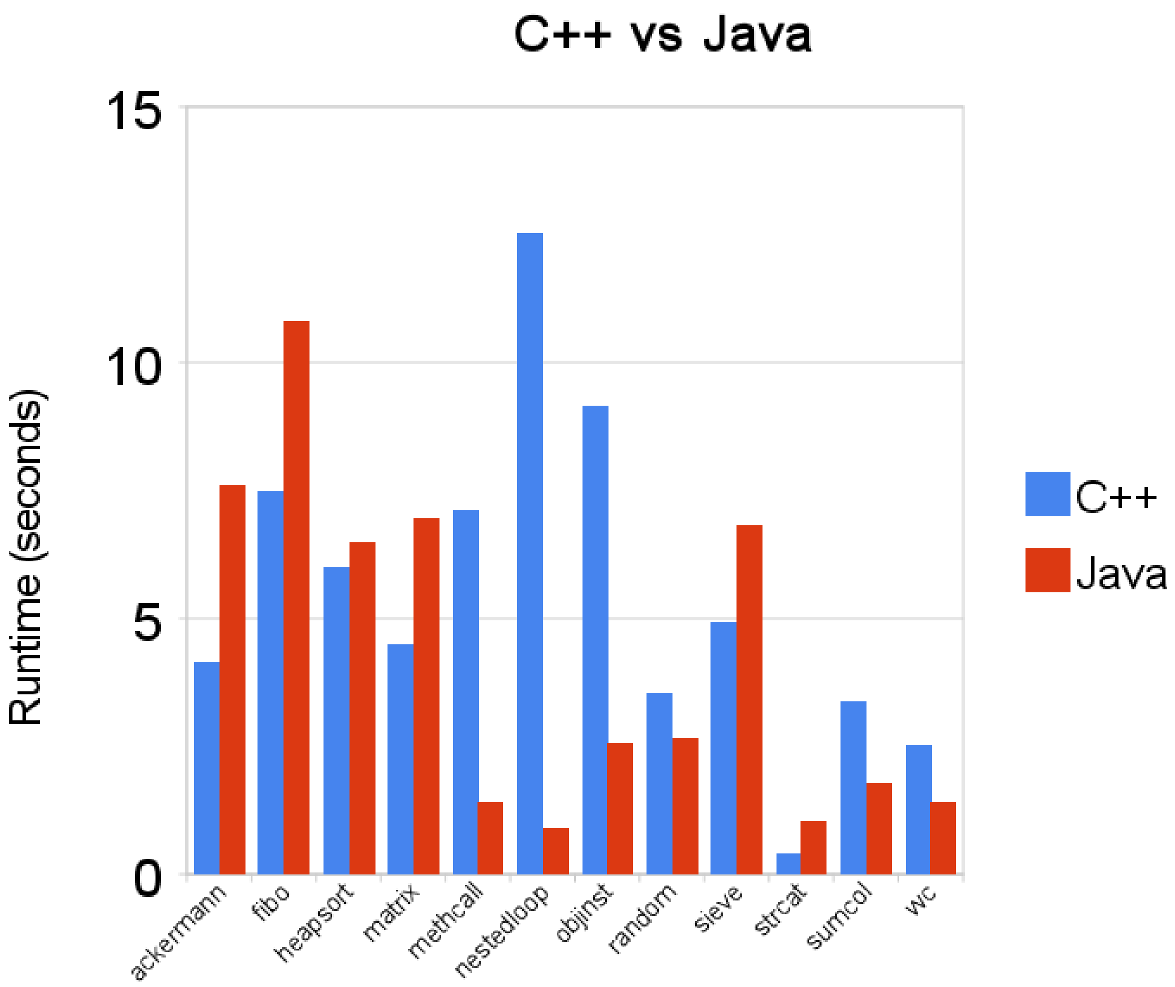
- Poor performance due to interpretative overhead (typically 5-20 × slower)
  - Every instruction considered in isolation;
  - Confuses branch prediction; and . . .

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;
}
```

Virtual Machines vs Compilers

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



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

JIT Compilers

A just-in-time (JIT) compiler generates native code *during* program execution.

Advantages

- Target specific architectural details;
- Observe program properties only possible at runtime;
- Efficiently allocate optimization time towards important methods.

Disadvantages

Now that the program performance depends on compile time, there are competing concerns.

- Compilation time and memory use;
- Code quality.

Effective JIT compilers offset the compilation cost with improved code performance.

Dynamo: A transparent dynamic optimization system. (Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia). PLDI 2000.

Virtual Machines

In this class we will look at two different virtual machines

Java Virtual Machine: stack-based IR

VirtualRISC: register-based IR (after the break)

Virtual Machines

Virtual machines

JVM

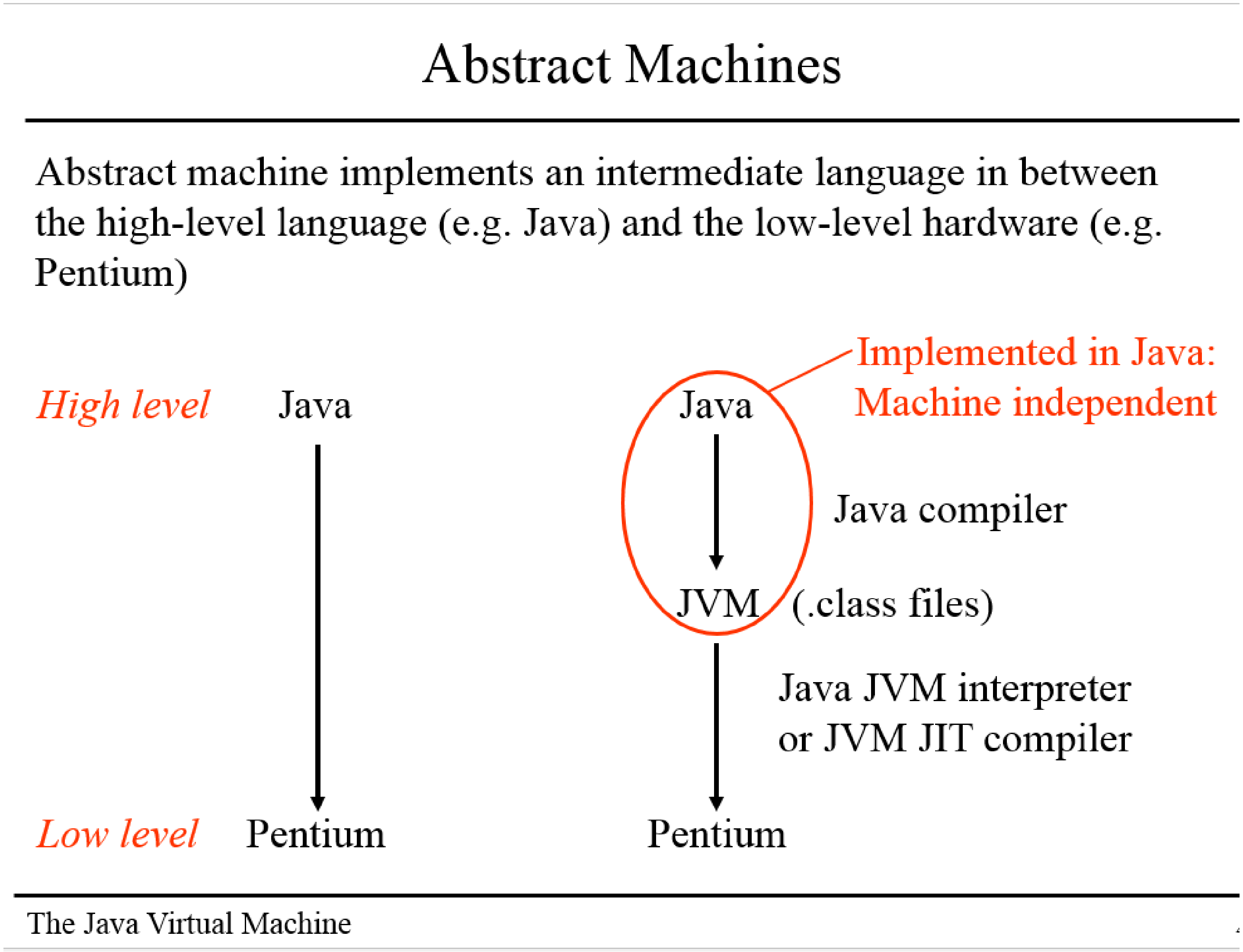
Bytecode

Verification

Stack code

Summary

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)

High level Java

Low level Pentium

Java

Java compiler

JVM (.class files)

Java JVM interpreter
or JVM JIT compiler

Pentium

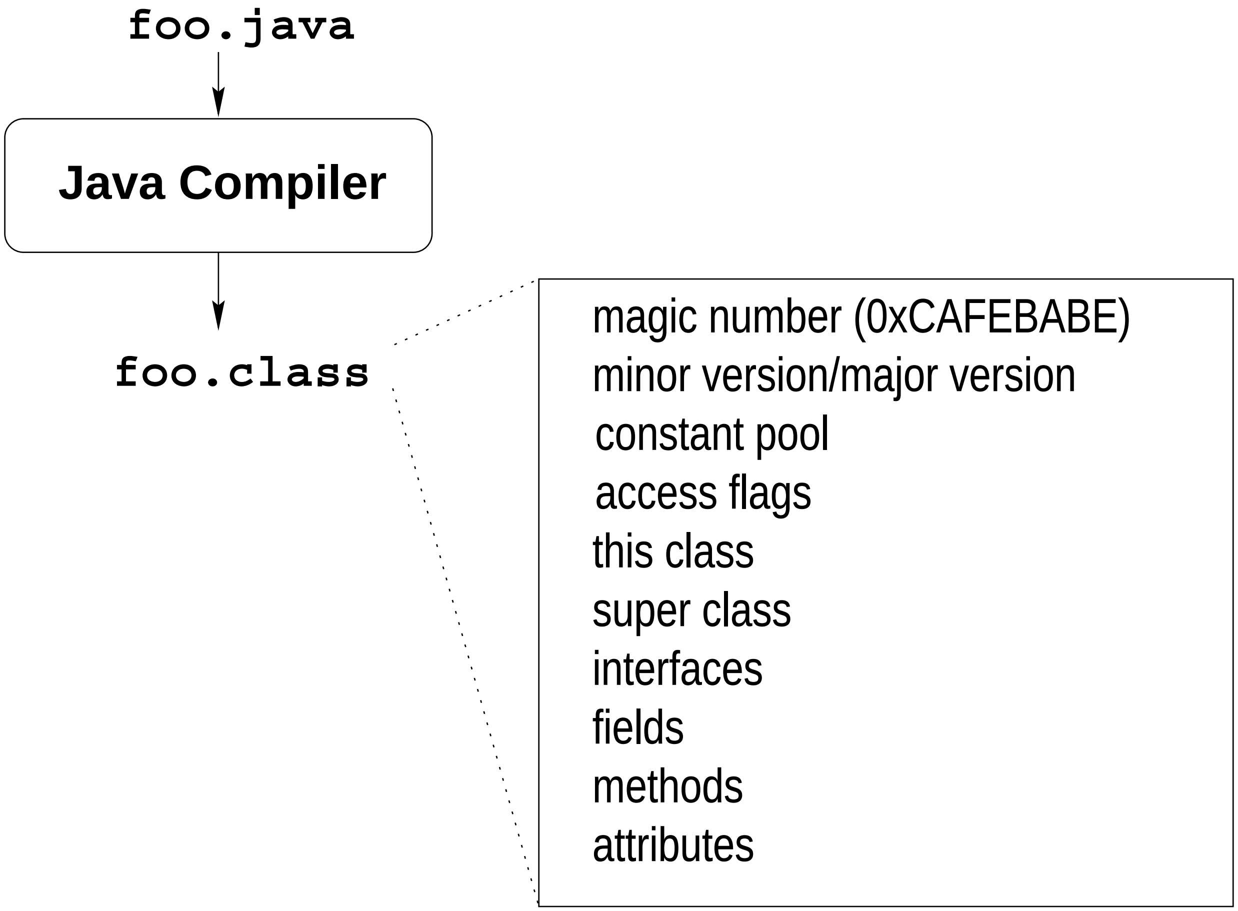
Implemented in Java:
Machine independent

The Java Virtual Machine

Note: slides of this format from `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 along with class properties.



foo.java

Java Compiler

foo.class

magic number (0xCAFEBABE)
minor version/major version
constant pool
access flags
this class
super class
interfaces
fields
methods
attributes

Why `0xCAFEBABE`? `http://radio-weblogs.com/0100490/2003/01/28.html`

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 following components

- 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 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

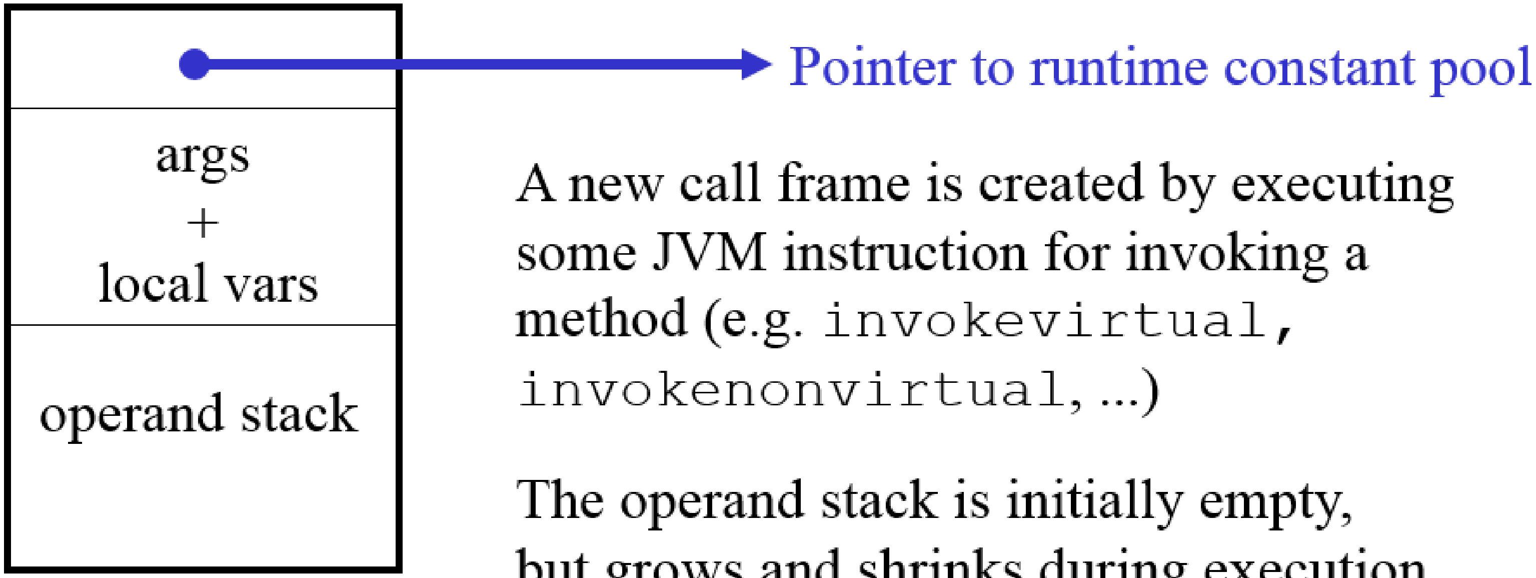
- A reference to the constant pool;
- Locals:
  - A reference to the current object (`this`) if any;
  - The method arguments;
  - The local variables; and
- A local stack used for intermediate results (the baby stack).

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 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 Execution

Condition codes

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

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`.

Virtual Machines

Virtual machines

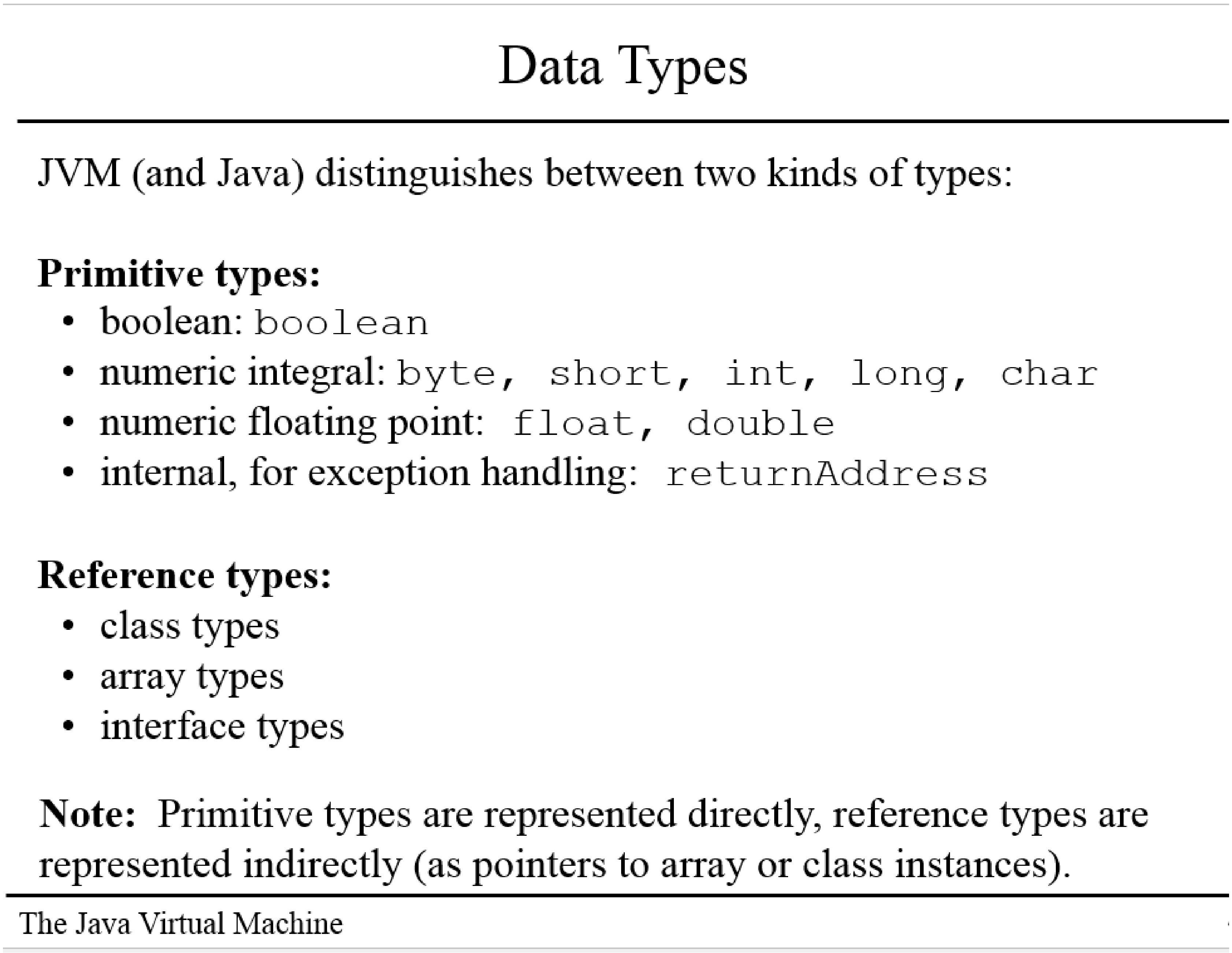
JVM

Bytecode

Verification

Stack code

Summary



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

Primitive 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 prepend L, replace "." by "/" and add ";";
  - 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>
  ```

- Max 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)`

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

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*

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_m1     [...]  -> [...:-1]
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 comparison instructions `if_icmpgt`, `if_icmpge`, `if_icmplt`, `if_icmple`

```
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         [...]  -> [...]
```

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>(T_1,...,T_n)V    [...:o:a_1:...:a_n] -> [...]
```

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

Most methods are invoked using an `invokevirtual` instruction

```
invokevirtual m sig     [...:o:a_1:...:a_n] -> [...:ret]
```

Internally

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

```
entry = lookupHierarchy(m, sig, dynamic_class(o));
block = block(entry);

push stack frame of size: block.locals + block.stacksize;

local[0] = o;    // local 0 points to "this"
local[1] = a_1;
...
local[n] = a_n;

pc = block.code;
```

Method Operations

Other methods are invoked using an `invokespecial` instruction

invokespecial m sig   [...:o:a_1:...:a_n] -> [...:ret]

Internally

```
entry = lookupClassOnly(m, sig, static_class(o));
block = block(entry);

push stack frame of size: block.locals + block.stacksize;

local[0] = o;       // local 0 points to "this"
local[1] = a_1;
...
local[n] = a_n;

pc = block.code;
```

Comparison to `invokevirtual`

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, where `first` is a field of type `Object` and `rest` is a field of type `Cons`.

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
```

Announcements (Wednesday, February 12th)

Assignment 2

- Will be graded by next week. If you have not submitted yet, do so soon!

Midterm

- Tuesday, February 25th 6:00 PM - 7:30 PM
- Conflicts – please contact me

Milestone 1

- Get started early!
- Any questions?
- **Due:** Friday, February 21st 11:59 PM

Reference Compiler (GoLite)

Accessing

- `ssh <socs_username>@teaching.cs.mcgill.ca`
- `~cs520/golitec {keyword} < {file}`
- If you find errors in the reference compiler, bonus points!

Keywords for the first milestone

- `scan`: run scanner only, OK/Error
- `tokens`: produce the list of tokens for the program
- `parse`: run scanner+parser, OK/Error
- `pretty`: produce a pretty output for the program

Virtual Machines

Virtual machines

JVM

Bytecode

Verification

Stack code

Summary

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.

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
  - All instructions are complete;
  - Branch targets are within the code segment;
  - Only legal offsets are referenced;
  - Constants have appropriate types; and
  - Execution cannot fall of the end of the code.

Bytecode Verification - Interesting Properties

Stack properties

- At any program point, the stack is the same size along all execution paths; and
- At any program point, the stack contains the same types along all execution paths.

Why? Conservatively guess instructions executed after the program point are path independent.

Type properties

- Each instruction must be executed with the correct number and types of arguments on the stack, and in locals (on all execution paths); and
- Fields are assigned appropriate values.

Other properties

- 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.

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
```

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” ⇒ also called “Proof-Carrying Code”.
- Java 7 (version 51.0 bytecodes) JVMs enforce presence of these attributes.

Consider the Following Mystery Program

```
public class u1 {
   public static void main(String [] args) {
      int r = mystery(4);
      System.out.println(r);
   }

   static int mystery(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/mystery(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 mystery(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`

Virtual Machines

Virtual machines

JVM

Bytecode

Verification

Stack code

Summary

Converting Class Files

A useful tool for dealing with class files, `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 more details.
```

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
```

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

Virtual Machines

Virtual machines

JVM

Bytecode

Verification

Stack code

Summary

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 (coming soon)

- 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 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
```