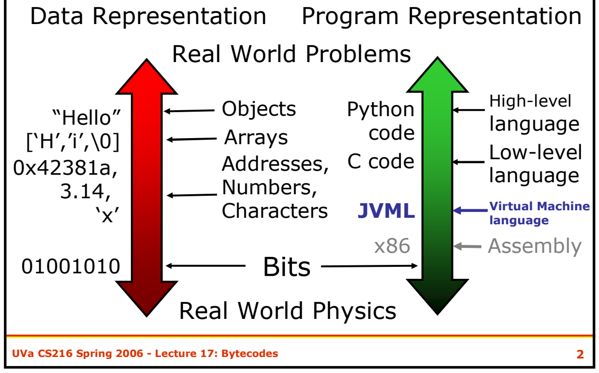
CS216: Program and Data Representation
University of Virginia Computer Science
Spring 2006 David Evans

Lecture 17: 0xCAFEBABE (Virtual Machines)

http://www.cs.virginia.edu/cs216

CS216 Roadmap



Java Virtual Machine

JVML is a detour:
everything else we have seen is part of running the PS1 Python program

Java™: Programming Language

"A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language." [Sun95]

Wednesday's class

Properties of language implementations, not languages

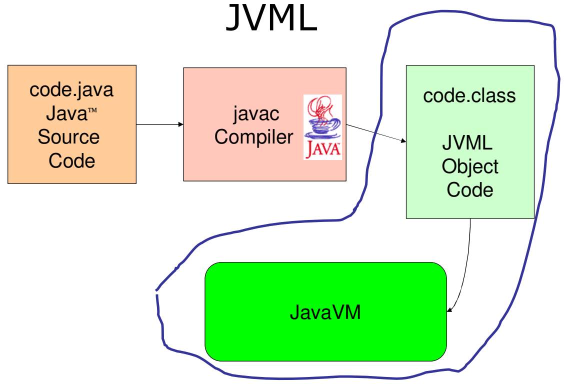
Java™: Programming Language

compared to C++, not to C sort of

"A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language." [Sun95]

Java: int is 32 bits
C: int is >= 16 bits

JVML



Java Virtual Machine

- Small and simple to implement
- All VMs will run all programs the same way
- "Secure"

Java Ring (1998)

Implementing the JavaVM

```
load class into memory
set the instruction pointer to point to the
     beginning of main
while not finished:
   fetch the next instruction
   execute that instruction
```

Some other issues we will talk about Wednesday:
Verification – need to check byte codes satisfy security policy

Java Byte Codes

- Stack-based virtual machine
- Small instruction set: 202 instructions (all are 1 byte opcode + operands)
  - Intel x86: ~280 instructions (1 to 17 bytes long!)
- Memory is typed
- Every Java class file begins with magic number 3405691582
  = **0xCAFEBABE** in hex

Stack-Based Computation

- **push** – put something on the top of the stack
- **pop** – get and remove the top of the stack

Stack

push 2
push 3
add

2 5
3

Does 2 pops, pushes sum

Some JVML Instructions

Opcode	Mnemonic	Description
0	nop	Does nothing
1	aconst_null	Push null on the stack
3	iconst_0	Push int 0 on the stack
4	iconst_1	Push int 1 on the stack
…		

Why do we need both aconst_null and iconst_0?

Load Constant

Opcode	Mnemonic	Description
18	ldc <value>	Push a one-word (4 bytes) constant onto the stack

Constant may be an int, float or String

```
ldc "Hello"
ldc 216
```

The String is really a reference to an entry in the string constant table!

Arithmetic

Opcode	Mnemonic	Description
96	iadd	Pops two integers from the stack and pushes their sum

```
iconst_2
iconst_3
iadd
```

Arithmetic

Opcode	Mnemonic	Description
96	iadd	Pops two integers from the stack and pushes their sum
97	ladd	Pops two long integers from the stack and pushes their sum
…		
106	fmul	Pops two floats from the stack and pushes their product
…		
119	dneg	Pops a double from the stack, and pushes its negation

Java Byte Code Instructions

- 0: nop
- 1-20: putting constants on the stack
- 96-119: arithmetic on ints, longs, floats, doubles

- 1 byte opcode: 146 left
- What other kinds of instructions do we need?

Other Instruction Classes

- Control Flow (~20 instructions)
  - if, goto, return
- Loading and Storing Variables (65 instructions)
- Method Calls (4 instructions)
- Creating objects (1 instruction)
- Using object fields (4 instructions)
- Arrays (3 instructions)

Control Flow

- **ifeq <label>**

  Pop an int off the stack. If it is zero, jump to the label. Otherwise, continue normally.
- **if_icmple <label>**

  Pop two ints off the stack. If the second one is <= the first one, jump to the label. Otherwise, continue normally.

Referencing Memory

- **iload <varnum>**
  - Pushes the int in local variable <varnum> (1 bytes) on the stack
- **istore <varnum>**
  - Pops the int on the top of the stack and stores it in local variable <varnum>

Referencing Example

```
Method void main(java.lang.String[])
  0 iconst_2
  1 istore_1
  2 iconst_3
  3 istore_2
  4 iload_1
  5 iload_2
  6 iadd
  7 istore_3
  8 getstatic #2 <Field java.io.PrintStream err>
 11 new #3 <Class java.lang.StringBuffer>
 14 dup
 15 invokespecial #4 <Method java.lang.StringBuffer()>
 18 ldc #5 <String "c: ">
 20 invokevirtual #6 <Method java.lang.StringBuffer append(java.lang.String)>
 23 iload_3
 24 invokevirtual #7 <Method java.lang.StringBuffer append(int)>
 27 invokevirtual #8 <Method java.lang.String toString()>
 30 invokevirtual #9 <Method void println(java.lang.String)>
 33 return
```

```
public class Locals1 {
   static public void main (String args[]) {
      int a = 2;
      int b = 3;
      int c = a + b;

      System.err.println ("c: " + c); } }
```

Method Calls

- **invokevirtual <method>**
  - Invokes the method <method> on the parameters and object on the top of the stack.
  - Finds the appropriate method at run-time based on the actual type of the this object.

```
invokevirtual <Method void println(java.lang.String)>
```

Method Calls

- **invokestatic <method>**
  - Invokes a static (class) method <method> on the parameters on the top of the stack.
  - Finds the appropriate method at run-time based on the actual type of the this object.

Example

```
public class Sample1 {
   static public void main (String args[]) {
      System.err.println ("Hello!");
      System.exit (1);
   }
}
```

```
public class Sample1 {
   static public void main (String args[]) {
      System.err.println ("Hello!");
      System.exit (1); } }
```

```
> javap -c Sample1
Compiled from Sample1.java
public class Sample1 extends java.lang.Object {
    public Sample1();
    public static void main(java.lang.String[]);
}

Method Sample1()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return
Method void main(java.lang.String[])
   0 getstatic #2 <Field java.io.PrintStream err>
   3 ldc #3 <String "Hello!">
   5 invokevirtual #4 <Method void println(java.lang.String)>
   8 iconst_1
   9 invokestatic #5 <Method void exit(int)>
  12 return
```

Charge

- PS5: Due Wednesday
  - Question 2 is a "tricky" question
  - Focus on correctness: implement something simple for questions 7-9 first
  - You can describe clever designs for question 6, simplicity should be the main factor in deciding what to implement