CSc 3210
Slides by Michael Weeks
Copyright 2015

Over-view

• Unix basics
• javac
• java
• .j files
• javap

Java programs

compile a java program to a java class file
javac HelloWorld.java
run the java program (class file) after it is compiled
java HelloWorld

Java programs

disassemble a java class file
(-c says to show the code)

djavap -c HelloWorld
djavap -c HelloWorld > myprogram.j
dassemble a java virtual machine program to byte-codes
djava -jar jasmin-2.4/jasmin.jar HelloWorld.j

Note that javap output is not formatted for Jasmin input

Logging on, and logging off

Log on to the system
ssh -l your_account solar.cs.gsu.edu
dthat is a lower-case L, not a one.

Logoff the system
dexit
Unix commands to know

Keep a log of what you do (creates the file mylogWednesday)

script mylogWednesday

The log will end when you type `exit`.

Print the environmental variable `SHELL`

`echo $SHELL`

Look for all occurrences of `Hello` in the files, then send the results to the awk program, which will print only the 5th thing, then a space, then the 1st thing.

`grep Hello * | awk '{ print $5 " $1 }'`

Get a long list of the files

`ls -l`

Get a long list of the files, including ones that start with .

`ls -al`

Get a list of the commands that you’ve used

`history`

vi is an editor

`vi HelloWorld.java`

xxd converts from/to hex

Convert to hexadecimal

`xxd HelloWorld.class`

`xxd HelloWorld.class > Hello.hex`

`xxd -r` converts from hexadecimal back to binary

`xxd -r Hello.hex > Hello.recon`

`xxd -h` is for help

`xxd -h`

Example 6502 microprocessor commands

- `TAX` ; Transfer A to X
- `LDA #$15` ; Load Accumulator, immediate
- `LDA $1234` ; Load Accumulator, absolute

The first does not need a parameter, so one byte encodes it.
The second command says to put a byte value in the Accumulator (A register), so we also need to say what byte value, resulting in 2 bytes.
The third command says to get a byte value from the memory location $1234, and put it in the Accumulator. We need 1 byte for the opcode, and 2 bytes for the address.
(Yes, the 6502 stores 16 bit addresses with low byte, high byte order.)

asciz = ascii, only terminated with a null (zero)

`bipush 42` = pushes byte integer value 42 onto the stack

`icnst_3` = pushes the integer constant 3 onto the stack

`icnst_4` = pushes the integer constant 4 onto the stack


- `b` = byte
- `s` = short
- `i` = int
- `l` = long
- `f` = float
- `d` = double
- `c` = char
- `a` = reference (address?)
How to assign a value to a variable

There are no registers, only the stack and local variables.

```java
; localVariable1 = 3
iconst_3
istore_1
```

We push the value onto the stack, then use istore to assign it to local variable 1.

How to assign a larger value to a variable

iconst works for small values. bipush works for 8-bit values. Use sipush for short (16 bit) values.

```java
; localVariable1 = 300
sipush 300
istore_1
```

We push the value onto the stack, then use istore to assign it to local variable 1. For even larger values, we can use shifts, or the constant pool (later).

How to copy a value from one variable to another

Use the stack to copy local variables.

```java
; localVariable2 = localVariable1
iload_1
istore_2
```

We push the value of local variable 1 onto the stack, then use istore to assign it to local variable 2.

An example java program

```java
public class loop {

    public static void main(String[] args) {

        // Find the sum 1..5
        int sum = 0;
        int i;
        for (i=1; i<=5; i++) {
            sum = sum + i;
        }
        System.out.println(sum);
    }
}
```

Compiling and running the java program

```
$ javac loop.java
$ java loop
15
$ ```
public class loop extends java.lang.Object {
    public loop();
    Code:
    0: aload_0
    1: invokespecial #1 ; //Method java/lang/Object."<init>":( )V
    4: return

public static void main(java.lang.String[]);
    Code:
    0: iconst_0
    1: istore_1
    2: iconst_1
    3: istore_2
    4: iload_2
    5: iinc 2, 1
    8: goto 4
    11: iadd
    13: isub 2, 1
    15: goto 4
    18: getstatic #2; //Field java/lang/System.out:Ljava/io/PrintStream;
    21: iload_1
    22: invokevirtual #3; //Method java/io/PrintStream.println:(I)V
    25: return
}

Looking at the javap output

• We examine the main method

• Comments are only about that line

Looking at the javap output

• Combine JVM instructions into more abstract comments

• The code below does not compile; it is a work-in-progress

• A few lines have two labels; we will eliminate the numeric ones soon
Looking at the javap output

- Identify the original Java commands
- The code below does not compile; it is a work-in-progress
- The commas are for us; the compiler will not understand

```java
iconst_0, istore_1 ; sum = 0
iconst_1, istore_2 ; i = 1

loop:
iload_2, iconst_5, if_icmpgt 19 ; if !(i<=5), goto done
iload_1, iload_2, iadd, istore_1 ; sum = sum + i
iinc 2, 1 ; i++
goto 4 ; goto loop
done:
getstatic #2, iload_1,
invokevirtual #3 ; System.out.println(sum)
```

Converting javap output to Jasmin input

- Labels must be on their own line
- No commas
- Semi-colons start comments
- We need compiler directives: .class, .super, .limit, .method, .end

```
.Compiler directives

- .class provides information about the class
- .super provides information about the super-class
- .method starts a method,
  .end ends a method
- .limit stack 3 means there will be at most 3 things on the stack for this method
- .limit locals 4 means there will be at most 4 local variables for this method
```

Converting from javap to Jasmin

Besides the above, there are other details when converting from javap to Jasmin.

javap produces lines like these

```
0: ldc #2; // String Russel Cuyler
5: getstatic #3; // Field java/lang/System.out;Ljava/io/PrintStream;
8: invokevirtual #4; // Method java/lang/String.charAt:(I)C
14: iinc 2, 1
```

The byte-offsets are not necessary, so this equivalent. Just make sure to leave any ones used with goto or if commands.
Converting from javap to Jasmin

The numbers after the number signs in the javap output correspond to constant pool entries. For Jasmin, we must define what the constant pool entries are. So replace things like #2 with the constant, e.g. "Russel Cuyler".

Javap output

```java
ldc #2; // String Russel Cuyler
```

Jasmin input

```java
28
```

Converting from javap to Jasmin

Javap output

```java
getstatic #3; // Field java/lang/System.out:Ljava/io/PrintStream;
```

Jasmin input

```java
getstatic java/lang/System/out Ljava/io/PrintStream;
```

Get rid of the number, the “Field” word, change the colon (:) to a space.

Converting from javap to Jasmin

Javap output

```java
iinc 2, 1
```

Jasmin input

```java
iinc 2 1
```

Get rid of the comma.

Converting from javap to Jasmin

Javap output

```java
invokevirtual #4; // Method java/lang/String.charAt:(I)C
```

Jasmin input

```java
invokevirtual java/lang/String.charAt(I)C
```

Get rid of the number, the “Method” word, delete the colon (:) and put a forward slash before the less-than sign.
We can decode a program with javap, and look at the JVM commands.

Next, we can compile it, by hand, to byte code.

This means looking up each command, and note the hexadecimal byte-code, e.g.:

```
  iload_0
  iinc 2 1
```

(We won’t need to repeatedly do this, but we should do it at least once to get the idea.)

Then we use xxd to view the hex codes for the class file.

We should see the codes that we produced, above.

We can check to verify that they are correct.

Also, we should see a lot of other stuff, put in by the compiler. What is this? (next slides)
The .class file

- Magic number
- Minor/major numbers
- Constant pool
- Class information

The first four bytes, *cafe babe*, were intentionally chosen. These form the “magic number”, telling the computer that this is a java class file.

Then the minor version number is 0000, and the major version number is 0032. The *javac --version* command reveals that it is *javac 1.6.0_33*. If we use *javap --verbose* *loop*, we get:

```
minor version: 0
major version: 50
```

Another computer responds with *javac 1.6.0_65* for the version. Compiling *loop.java*, then using *javap --verbose* reveals that it has the following version information.

```
minor version: 0
major version: 50
```

Thus, the release is not stored, but the code is made for a version 1.6 JVM.

Class information

- Access settings
- This class
- Super class
- Interfaces, fields, and methods

A JVM can look at the major and minor version and know that if it is generated for a later JVM version.

Next is the number of “constant pool” items (2 bytes). Each starts with a tag, identifying how it should be treated. Then any data will follow it. Some items, like text strings, define their own lengths. In this example, we have 001c constants defined.

The Descriptor

```
.method public static main([Ljava/lang/String;)V
```

C means char element. Lclassname; means class or interface (because we already use C for something else?). [http://docs.oracle.com/javase/6/docs/api/java/lang/Class.html#getName%28%29]
Creating an Integer Array

- make a new array with 10 elements
- use localVariable2 to refer to it

\[
\text{bipush} \ 10 \quad ; \text{number of elements} \\
\text{newarray} \ int \quad ; \text{make the array} \\
\text{astore}_2 \quad ; \text{put the array address} \\
; \text{localVariable2}
\]

Storing a value in an Integer Array

\[
\text{localVariable2[localVariable1]} = 3
\]

- push localVariable2 as address
- push localVariable1 as integer (index)
- push 3
- localVariable2[localVariable1] = stack_top

Accessing a value in an Integer Array

\[
\text{localVariable3} = \text{localVariable2[5]}
\]

- push localVariable2 as address
- push 5 as integer (index)
- localVariable3 = localVariable2[5]
- load_2 ; localVariable2 is the array address
- iload_1 ; use localVariable1’s value as the index
- iconst_3 ; the integer value to store
- iastore ; array[localVariable1] = 3

Creating a new integer array

Create the array, of 100 integers.

\[
\text{bipush} \ 100 \\
\text{newarray} \ int
\]

The array address will now be on the stack.

\[
\text{dup} \quad ; \text{Duplicate the last thing on the stack,} \\
; \text{which is a reference to the array.} \\
\text{iconst}_0 \quad ; \text{push 0 as the index} \\
\text{iconst}_1 \quad ; \text{push 1 as the value to store} \\
\text{iastore} \quad ; \text{pop the last 3 things, and}
\]

; store the value at index in the integer array \\
; array[0] = 1

The array address will still be on the stack, since we did not remove it.

We repeat this with different indices and values. Later, we store the array address into a local variable.

\[
\text{astore}_1 \quad ; \text{put the array address into localVariable1}
\]
Defining a method

; public static void method1(int)
.method public static method1(I)V
    .limit stack 3
    .limit locals 2
    ; the integer is in local variable 0
    ; ...
    return
.end method

Calling a method

Call method1 with a parameter of 10. This goes in the main method.

bipush 10
invokestatic shiftExample2/method1(I)V

shiftExample2 is the class name.

Defining a method

; public static void defineMaze(int[])
.method public static defineMaze([I)V
    ; the integer array's address is in local variable 0
    ; ...
    return
.end method

Calling a method

The integer array's address is in local variable 1. Call defineMaze with the integer array as a parameter.

aload_1
invokestatic myclass/defineMaze([I)V

myclass is the class name.

Returning a value

This method takes an integer as a parameter, adds 1 to it, and returns it.

;public static int retSomething(int);
.method public static retSomething(I)I
    .limit stack 3
    .limit locals 1
    iload_0 ; push the parameter
    iconst_1 ; push 1
    iadd ; add them, result is on stack
    ireturn ; return the integer value
.end method

Returning a value

The following calls the retSomething method with a parameter of 3.

iconst_3
invokestatic ex7/retSomething(I)I

The result will be on the stack.
Using line numbers

- How do we know where to look when we get an exception?
- Use .line
- Jasmin parameter -g may also work

Using line numbers

Here is a subset of lines with an intentional error: println should be println.

```java
.getstatic java/lang/System/out Ljava/io/PrintStream;
iload_1
.invokevirtual java/io/PrintStream.println(I)V
```

Using line numbers

We know exactly where the problem is when we run it.

```
$ java -jar jasmn-2.4/jasmn.jar lineExample.j
Generated: lineExample.class
$ java lineExample
Exception in thread "main" java.lang.NoSuchMethodError:
java.io.PrintStream.println(I)V
at lineExample.main(lineExample.j:3)
```

Two dimensional arrays

We can simulate a 2-D array with a 1-D array. Suppose the array should have 5 rows and 8 columns. Let row be an index from 0..4 and col be an index from 0..7. Map the row, column to the index.

```
index = row * 8 + col
```

The 8 is the maximum column value.

Switch statements

There are two ways to implement a switch statement in JVM bytecode. Suppose that we have the following code.

```
switch (i) {
case 0:
    System.out.println("zero");
    break;
case 1:
    System.out.println("one");
    break;
default:
    System.out.println("not zero or one");
    break;
}
```

Switch statements

The code (without the println calls) looks like this.

```
iload_1
lookupswitch
    0 : print_zero
    1 : print_one
default : print_neither
print_zero:
    ; more after this
```

First, we put the integer value on the stack. Depending on its value, the program branches to the print_zero label, print_one, or print_neither.
Switch statements

Here is another way to implement the switch statement.

```java
iload_1
tableswitch 0 1
    print_zero
    print_one
default : print_neither

print_zero:
; more after this
```

Like before, we first put the integer value on the stack. The minimum and maximum case values are after `tableswitch`.

Why have two switch statements?

- The `lookupswitch` command allows any case values that you want.
- The JVM may need to examine many cases before finding the right one.
- The `tableswitch` command should be more efficient; based on the value, the JVM can load the branch destination from a table.

Using -g with jasmin

Here is an example of using the `-g` option with jasmin.

```bash
$ java -cp . -jar jasmin-2.4/jasmin.jar atomJVM.j
Generated: atomJVM.class
$ java atomJVM
atomJVM
Exception in thread "main" java.lang.NoSuchFieldError: useColor
    at atomJVM.<init>(atomJVM.j)
at atomJVM.main(atomJVM.j)
```

Where do we look for the problem?

*By the way, `-cp .` adds the current directory to the current path.

Using -g with jasmin

Here is line 98:

```java
putfield atomJVM/useColor I
```
To fix it, we have to define `useColor`.

```java
.field private useColor I = 1 ;
```
With the line above in the program (before the first `method`), it works.
Programming Stubs

methods with little to no content

- Called “stubs”

- Useful for testing
e.g. comment out code

- Useful as place-holders
e.g. planning a program’s structure

Example Programming Stub

This method does nothing, but can be called.

```
.method private static updateBoard()V
.limit stack 3
.limit locals 8
.return
.end method
```

Example Programming Stub

This method returns a value.

```
.method public stub2()I
.limit stack 8
.limit locals 8
.iconst_0
.ireturn
.end method
```

We can call it with this:

```
... invokestatic myclass/stub2()I istore_2 ...
```

Java Static Variable

```
public static final int MAX_WIDTH;
```

becomes

```
.field public MAX_WIDTH I
```

then later, access it with

```
yourclassname/MAX_WIDTH
```

Static Variable Examples

columns = 20

```
aload_0 ; or wherever the address is
bipush 20
putfield yourclassname/columns I
```

get the value of rows

```
aload_0
getfield yourclassname/rows I
```

Array address

Instead of storing an array’s address as

```
astore_2
```

we could use

```
putstatic myclass/myIntArray [I
to give it a name
```
Example Array Address

This goes before .method at the top

.field static Board [I

Create a 400-element, integer array

sipush 400
newarray int
putstatic h3/Board [I

Example Array Address

We can access an element with this:

; refer to the array address
getstatic h3/Board [I

; use var0 as index
iload_0

; put the array value on the stack
iaload

Things to check

• Variables must be in the .j file, too, before the .method line

• public static final int MAX_WIDTH; should be .field public MAX_WIDTH I

• private static int[] Board; should be .field Board [I

• Refer by the full name, like
  invokestatic myclass/initBoard()V or
  myclass/Board instead of just
  Board

Private Classes

• javap does not show private classes

• over-ride the defaults to see them

  Show only the public classes
  javap -c myprogram

  Show public and private classes
  javap -c -private myprogram

Fixing errors

Stubborn compilation errors and exceptions come from

• not including the full names of things
  e.g. yourclassname/rows

• not having the syntax just right for calls
  e.g. invokevirtual, invoke...

• not having enough stack space or local variables
  e.g. .limit stack...

• missing things
  e.g. .field public MAX_WIDTH I

• other close-but-not-exactly-right references
  e.g. board where you mean Board
Fixing errors

- getting the syntax just right, e.g.
  invokevirtual java/lang/String.length()I should be:
  invokevirtual java/lang/String/length()I

- having semi-colons at the end of text where it is not needed, e.g.
  invokespecial java/lang/Object<init>()V;
  gives an error
  invokespecial java/lang/Object/<init>()V
  is the correct form

Code won’t compile or run?

- Group lines together

- Add other lines so that the program still works
  e.g. this can generate a stack underflow
    ; iload_1
  so add a line like
    icnst_0 ; remove after testing

It STILL won’t compile or run?

- Start with a new, working template

- Put your code in but comment everything out
  cat badprogram.j | sed 's/^/;/'

- Repeat:
  - Uncomment half, test it
  - Uncomment the other half, test it

- See your TA or professor