CSC 8505 Handout : JVM & Jasmin

Note: This handout provides you with the basic information about JVM. Although we tried to be accurate about the description, there may be errors. Feel free to check your compiler's output (JVM code in Jasmin syntax) with that of **javac.** To achieve this, you need to write a Java class equivalent to your C− program, then compile it and then disassemble the generated .class file using **D-Java** with '–o jasmin' flag (see page 10 for an example).

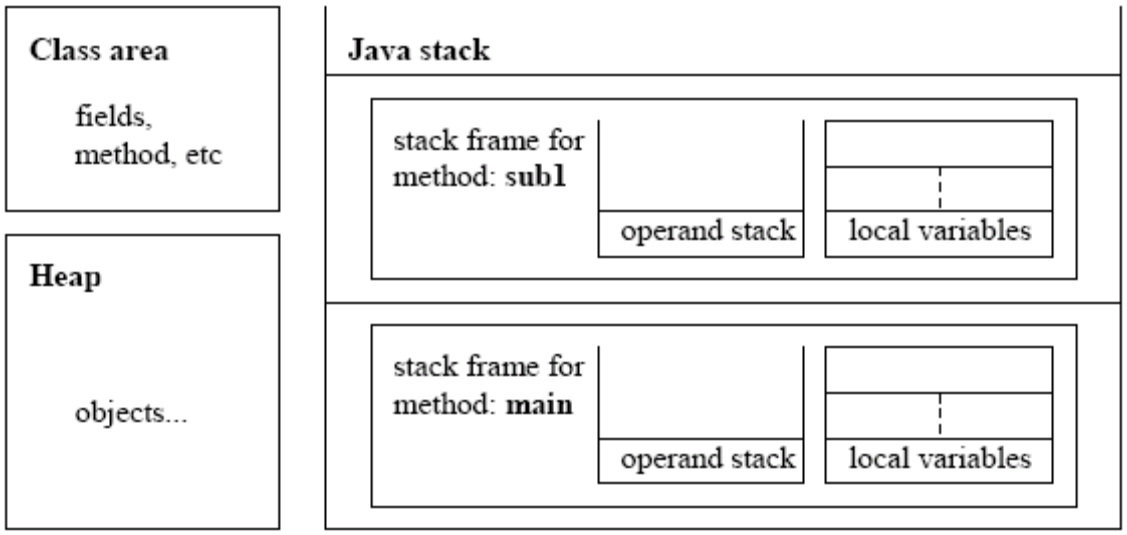
1. JVM Basics

The Java Virtual Machine (JVM) specifies the following:

- A set of instructions and their semantics
- A binary format of **class** files
- An algorithm to *verify* virtual machine code for integrity.

Properties of the JVM

- It is a virtual machine. The specification was designed independent of existing architectures.
- Unlike concrete architectures, JVM does not allow access to specific memory locations. This approach can provide a better program security.
- The JVM has the following organization:

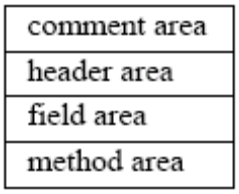


- The class area stores information associated with the class, e.g., fields, methods.
- The heap stores objects associated with the class.
- The Java stack is a higher-level stack, which maintains stack frames corresponding to methods. The top frame is current/active. In addition, each stack frame contains an operand stack and an array of local variables. One or two stack-top elements are popped from the stack, an arithmetic operation is performed on them, and the result is pushed on the stack. Each stack frame may contain up to 256 local variables.
- There are a number of relatively high-level JVM instructions (compared to assembly languages for concrete architectures). Thus, while JVM programming follows the format of assembly language programming, it also retains the flavor of Java programming, esp. with respect to object handling.

2. Jasmin Code Structure

Since Java class files are hard to read, we use Jasmin as the target language of our compiler. Assembling Jasmin code to a class file and disassembling a class file to a Jasmin code can be done by Jasmin and D-Java, respectively (refer to the project handout for the detail).

A Jasmin code consists of the following four sections:



- The comment area provides the developer and source file information. Jasmin comments are of the end-of-line type and begins with a semicolon '**;**'. Comments can appear anywhere. For example:

```
; Output created by D-Java (mailto:umsilve1@cc.umanitoba.ca)
;

;Classfile version:
;    Major: 49
;    Minor: 0
```

- The header area contains the following three lines.

```
.source  Test.java
.class   synchronized Test
.super   java/lang/Object
```

- The field area contains field declarations. For example, the following line corresponds to the field (in Java) **`public static int x;`**.

```
.field public static x I
```

- The method area contains method/constructor declarations. For example, the default constructor of a class is realized as a JVM method as follows:

```
; >> METHOD 1 <<

.method
   <init>()V
   .limit stack
   1
   .limit
   locals 1
```

```
.line 1
   aload_0
   invokenonvirtual
   java/lang/Object/<init>()V
   return
.end method

; >> METHOD 2 <<

.method public static f(I)V
    .limit stack 2
    .limit locals 2
.line 12
    iload_0
    sipush 300
    iadd
    istore_1
.line 13
    return
.end method
```

3. Jasmin Directives

Jasmin directives are not JVM instructions, but controls Jasmin assembler by providing meta-level information. All Jasmin directives begin with a period '**.**'. We need to know the following directives for our project.

- The required lines in header section: **`.source`**, **`.class`**, and **`.super`**
  Pattern:
  `.source` `<source file name>`
  `.class synchronized` `<class name>`
  `.super` `<super class name>` (**`java/lang/Object`** by default)

- To declare fields: **`.field`**
  Pattern: **`.field`** `<modifiers> <variable name> <type descriptor>`
  Note: We'll have **`public static`** as modifiers. See the next section about type descriptors.

- To indicate the beginning of a method: **`.method`**
  Pattern (method):
  `.method` `<modifiers> <class name>/<method name>`**`(`**`<arguments>`**`)`**`<return type>`
  Pattern (constructor): **`.method <init>(`**`<arguments>`**`)V`**
  Note: We'll have **`public static`** as modifiers. <arguments> are sequence of type descriptors and <return type> is a type descriptor. See the next section about type descriptors.

- To indicate the end: **`.end`**
  (e.g., end of a method) **.end method**

- To set the limit on the operand stack depth or the size of the local variable array: **`.limit`**
  `.limit stack <number>`
  `.limit locals <number>`
  Note: You may use a reasonably high, fixed value, say 32, as the limit. Exact limit requires

analysis.

- **`.line`**: To indicate the line number in the source file. **Note:** You do not need to generate this.

4. Types in Jasmin Codes

Data types in Jasmin codes are indicated by the following two means.

- **`Type descriptor`**: Type information used for field declaration, argument/return type specification, etc. E.g.,

```
.field public static x I
.method public static f(I)V
```

- **`Type mnemonic`**: A part of a JVM instruction that corresponds to a Java type. Type mnemonic often appears as the first character of JVM instructions, written in lower case. E.g.,

```
iadd ; add ints
fadd ; add floats
```

The following table shows type descriptor and type mnemonic with the corresponding Java type.

Java type	Type **descriptor** (uppercase)	Type **mnemonic** (lowercase)
int	**I**	**i**
long	J	l
short	S	*s*
float	F	f
double	D	d
byte	B	b
char	C	c
boolean	**Z**	*n/a* (use i)
reference	**L***name*;	**a**
array	**[** (as prefix to the base type)	*n/a*
void	**V**	*n/a*

Note: In our project, only those in bold face will be needed.

5. Data Operations

In this and the following several sections, we overview JVM instructions used in Jasmin codes. Complete listing is available online. See http://jasmin.sourceforge.net/instructions.html as well as http://mrl.nyu.edu/~meyer/jvmref/. This section introduces instructions for data operations. JVM does not have a separate Boolean type. Integer constant 1 is treated as TRUE and integer constant 0 is treated as FALSE.

Type	Array type	Operand range/value	Push a constant	Pop a value and store it in a local	Push the value in a local
Overloaded			`ldc`	n/a	n/a
`int`		4-byte	**`ldc`**	n/a	n/a
		2-byte	`sipush`	**`istore`**	**`iload`**
		1-byte	`bipush`		
		0-5	`iconst_`*i*		
		0-3		`istore_`*i*	`iload_`*i*
		-1	`iconst_m1`	n/a	n/a
`long`		2-byte	n/a	`lstore`	`lload`
		0-1	`lconst_`*i*	`lstore_`*i*	`lload_`*i*
`float`		2-byte	`ldc`	`fstore`	`fload`
		0-2	`fconst_`*i*	`fstore_`*i*	`fload_`*i*
`double`		2-byte	n/a	`dstore`	`dload`
		0-1	`dconst_`*i*	`dstore_`*i*	`dload_`*i*
String		2-byte	`ldc`	`astore`	`aload`

Type	Array type	Operand range/value	Push a constant	Pop a value and store it in a local	Push the value in a local
Array	int	n/a	n/a	**`iastore`**	**`iaload`**
	long			`lastore`	`laload`
	short			`sastore`	`saload`
	byte			`bastore`	`baload`
	char			`castore`	`caload`
	float			`fastore`	`faload`
	double			`dastore`	`daload`
	Reference			`aastore`	`aaload`
Reference (in general)		2-byte	n/a	**`astore`**	**`aload`**
		0-3	n/a	`astore_`*i*	`aload_`*i*
		`null`	`aconst_null`	n/a	n/a

The above table looks rather complicated. But all you need is the ones in bold face. The reason why there are so many similar instructions for some data types is as follows. JVM instructions are all byte length. This means that we can only have 256 instructions. Notice that an instruction such as load 0 is much more frequently used than, say, load 1234567. If we allocate an instruction for frequently used operands, we can reduce the size of the class file and also achieve optimized interpretation of such codes. So, for operands of small numbers, more specialized JVM instructions are prepared.

For example, while you can write **ldc 0** (1+4 bytes), you may also write **sipush 0** (1+2 bytes), **bipush 0** (1+1 byte), or **iconst_0** (1 byte). You will see all of these instructions when you disassemble (by D-Java) a class. You can reduce the code size and increase the performance by choosing a more optimal instruction. Note that efficiency/speed is not the issue in our project and you can use the most general forms (in bold face) for each data type and operation.

See the example at the end of the next section.

6. Arithmetic Operations

Arithmetic operations are always performed on the top one or two elements on the stack and returns the result on the top of the stack.

Type	addition	subtraction	multiplication	division	negation
int	**iadd**	**isub**	**imul**	**idiv**	**ineg**
long	ladd	lsub	lmul	ldiv	lneg
float	fadd	fsub	fmul	fdiv	fneg
double	dadd	dsub	dmul	ddiv	dneg

The following example illustrates the use of data/arithmetic operations:

```
; y = x * 2, assume y is local variable #2 and x is local variable #1

iload 1  ; push value of x
ldc 2    ; push 2. can be replaced by shorter iconst_2
imul     ; pop the top two values, multiply and push the result
istore 2 ; pop and store the top value in y
```

7. Method Operations

There are two return instructions: **`return`** and **`ireturn`**. While **`return`** simply returns from the method, **`ireturn`** returns the stack top.

Method invocation involves three different instructions:

- **`invokenonvirtual/invokespecial`**: To invoke a *constructor* or a *private* method. The second one is a replacement of the first one in newer JDKs. But both of them can still be used.
- **`invokevirtual`**: To invoke a *public non-static* method.
- **`invokestatic`**: To invoke a *static* method.

Static method (function) call pattern (analogous for other method call instructions):

<push arg_1>
<push arg_2>
<push arg_n>
`invokestatic` <class name>/<method name>**(**<arguments>**)**<return type>

As before, <arguments> are sequence of type descriptors and <return type> is a type descriptor. For non-static methods, use **`invokevirtual`** instruction instead. Also, the first parameter to be pushed is the object reference. For example, the following corresponds to calling a static method (function) **`f(x,3)`** in a class **`Class1`** where **x** is the local variable # 0 and the method's return type is **`int`**:

```
iload 0
ldc 3
invokestatic Class1/f(II)I
```

8. Flow Control

The destination of jump can be indicated by a label like in other assembly languages. To unconditionally jump to a label, use "**goto** *label*". For conditional jump, there are branching instructions that branch based on the top of stack being zero (e.g., **ifeq** *label*) and there are branching instructions that compare the top two values to determine the jump. For this latter group of instructions, two values must be pushed on to the stack:

if_icmpeq *label*	**;** Jump to the label if the top two values on the stack are equal
if_icmpne *label*	**;** Jump to the label if the top two values on the stack are not equal
if_icmplt *label*	**;** Jump f the first-pushed value is less than the second-pushed value
if_icmple *label*	**;** Jump if the first value is less than or equal to the second value
if_icmpgt *label*	**;** Jump if the first value is greater than the second value
if_icmpge *label*	**;** Jump if the first value is greater than or equal to the second value

The following example corresponds to **`y = x < 20;`** where **x** is the local variable # 0 and **y** is local vaiable # 1:

```
iload 0
ldc 20
if_icmplt Label2
ldc 0
goto Label1
Label2:
ldc 1
Label1:
istore 1
```

The following example corresponds to **`if (x < 3) y = 10; else y = 20;`** where **x** is the local variable # 0 and **y** is a static variable of class **ifex** (see next section):

```
iload 0
ldc 3
if_icmplt Label2
ldc 0
goto Label1
Label2:
ldc 1
Label1:
ifeq Label3
ldc 10
putstatic ifex/y I
goto Label4
Label3:
ldc 20
putstatic ifex/y I
Label4:
```

If you examine the generated code above, you'll realize that a better code can be generated in this case since we do not actually compute and store the value of the conditional expression. We can simply make use of the two-value conditional jump instruction to jump to the `then' or the `else'

part. This requires carrying a context. You need not worry about this detail in your project. Here is a better code for the same statement:

```
iload_0
iconst_3
if_icmpge Label1
bipush 10
putstatic ifex/y I
goto Label2
Label1:
bipush 20
putstatic ifex/y I
Label2:
```

Note that the interpretation of the conditional in the source language is 'reversed'. That is, the conditional jump is when the condition is false.

9. Field Access/Assignment

To access fields in a class **`Class1`**, the following instructions can be used:

• To access a static field

```
getstatic Class1/y I
```

• To access a non-static field

```
aload_0
getfield Class1/x I
```

• To assign a value to a static field in Class1

```
ldc 123
putstatic Class1/y I
```

• To assign a value to a non-static field

```
aload_0
ldc 456
putfield Class1/x I
```

Note that to access a non-static field, we need to push the (reference to) current object which is assumed stored in the local variable # 0.

10. Array Processing

In order to allocate an array field, we need to specify the array type descriptor, e.g., **`[I.`** To allocate a local array variable, we simply reserve a local variable for it. So, there will be no corresponding Jasmin code for a declaration **`int[] x;`** much like a non-array declaration.

To create an actual array (object), we need to use the instruction **`newarray`**. To access and assign array elements, we need **`iaload`** and **`iastore`**, respectively. The following example illustrates this situation. Here, let's assume that **`x`** and **`y`** are represented as local variables # 1 and # 2,

respectively.

C-: **`int x[5];`** Jasmin:

```
ldc 5
newarray int ; create a new array object
astore 1     ; store the reference into x
```

C-/Java: **`x[2] = 123;`** Jasmin:

```
aload 1   ; array reference
ldc 2     ; index value
ldc 123   ; value to be stored
iastore   ; stores into apecified array location
```

C-/Java: **`y = x[2];`** Jasmin:

```
aload 1
ldc 2
iaload
istore 2
```

Note: For static arrays, use **`getstatic`** instead of **`aload`**. For example, if x were a static array of class Class1, then **`getstatic Class1/x [I`** instead of **`aload 1`**. Also, in this case the creation of array will use **`putstatic Class1/a [I`** instead of **`astore 1`**.

Appendix A. C− to Java to Jasmin Code Example

An Example C− Program

`int i;`	Global Variables i and a[10]
`int a[10];`	
`void f(int x)`	Local Variable #0, x
`{ int y;`	Local Variable #1, y
`}`	
`int g(int b, int c[] )`	Local Variable #0, b
`{ int z[5];`	Local Variable #1, c[]
`}`	Local Variable #2, z
`void main(void)`	
`{ int w;`	Local Variable #1, w
`}`	(Not 0 Since Java's main requires String[] which is variable #0)

It's Java Equivalent

`class someclass`	
`{ public static int i;`	Class/static variable i
`  public static int[] a = new int[10];`	Class/static variable a
`  public static void f( int x )`	Class/static method f
`  { int y;`	x is local variable #0
`  }`	y is local variable #1
`  public static int g(int b, int []c)`	b is local variable #0
`  { int[] z = new int [5];`	c is local variable #1
`  }`	z is local variable #2
`  public static void main (String[] args)`	Java requires args
`  { int w;`	args is local variable #0
`  }`	w is local variable #1
`}`	

The Complete C-/Java Program

```
    int i;
    int a[10];

    void f(int x) {
      int y;
      y = x + 300;
    }

    int g(int b, int[] c) {
      int z[5];
      z[2] = b + c[3];
      return z[2];
    }

    void main(void) {
      int w;
      w = g(i, a);
    }
```

```
class SomeClass {

    public static int i;
    public static int[] a = new int[10];


    public static void f(int x) {
      int y;
      y = x + 300;
    }

    public static int g(int b, int[] c) {
      int[] z = new int[5];
      z[2] = b + c[3];
      return z[2];
    }

    public static void main(String[] args) {
      int w;
      w = g(i, a);
    }
}
```

Jasmin Code

Note: The file was first compiled using *javac* and then disassembled with *D-Java* with *–o jasmin* flag.

```
;
; Output created by D-Java (mailto:umsilve1@cc.umanitoba.ca)
;

;Classfile version:
;    Major: 49
;    Minor: 0

.source SomeClass.java
.class  synchronized SomeClass
.super  java/lang/Object


.field public static i I
.field public static a [I

; >> METHOD 1 <<
.method <init>()V
    .limit stack 1
    .limit locals 1
.line 1
    aload_0
    invokenonvirtual java/lang/Object/<init>()V
    return
.end method

; >> METHOD 2 <<
.method public static f(I)V
    .limit stack 2
    .limit locals 2
.line 9
    iload_0
    sipush 300
    iadd
    istore_1
.line 10
    return
.end method

; >> METHOD 3 <<
.method public static g(I[I)I
    .limit stack 5
    .limit locals 3
.line 13
    iconst_5
    newarray int
    astore_2
.line 14
    aload_2
    iconst_2
    iload_0
    aload_1
    iconst_3
```

```
    iaload
    iadd
    iastore
.line 15
    aload_2
    iconst_2
    iaload
    ireturn
.end method

; >> METHOD 4 <<
.method public static main([Ljava/lang/String;)V
    .limit stack 2
    .limit locals 2
.line 20
    getstatic SomeClass/i I
    getstatic SomeClass/a [I
    invokestatic SomeClass/g(I[I)I
    istore_1
.line 21
    return
.end method

; >> METHOD 5 <<
.method static <clinit>()V
    .limit stack 1
    .limit locals 0
.line 4
    bipush 10
    newarray int
    putstatic SomeClass/a [I
    return
.end method
```