CSc 620

Debugging, Profiling, Tracing, and Visualizing Programs

1 : JVM

Christian Collberg

collberg+620@gmail.com

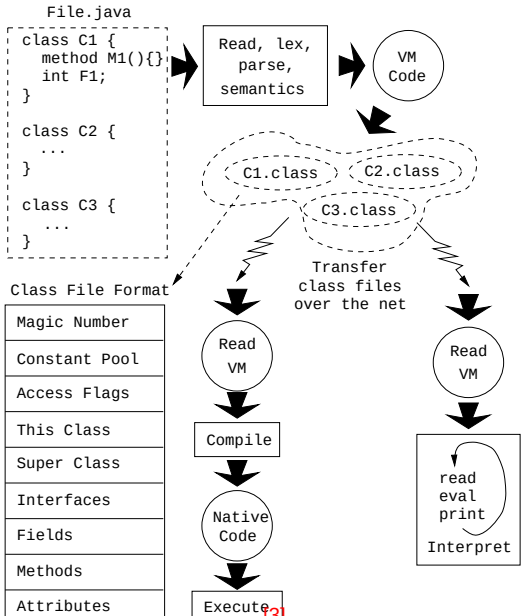
Department of Computer Science

University of Arizona

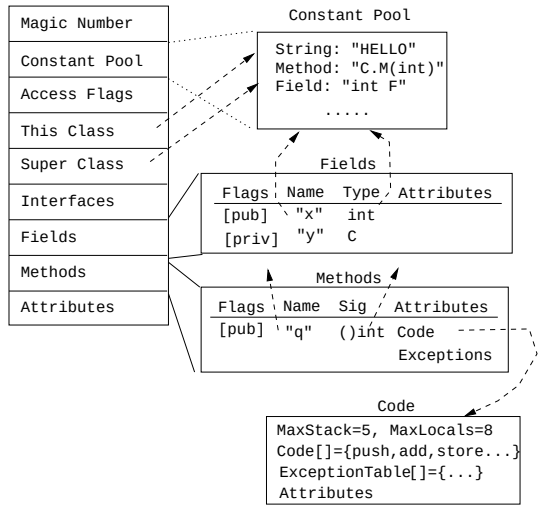
The Java Virtual Machine

- The Java VM has gone the “many complex instructions/large VM engine” way.
- Each Java source file may contain several Java classes. The Java compiler compiles each of these classes to a single Java *class file*.
- The Java class file stores all necessary data regarding the class. There is a symbol table (called the *Constant Pool*) which stores strings, large literal integers and floats, names and of all fields and methods.
- Each method is compiled to Java bytecode, a stack VM format.
- The class file is (almost) isomorphic to the source.

Compiling Java



The Java Class File Format



Java Byte Codes I

- The Java bytecodes can manipulate data in these formats: integers (32-bits), longs (64-bits), floats (32-bits), doubles (64-bits), shorts (16-bits), bytes (8-bits), object references (32/64-bit pointers), and arrays.
- The bytecodes are 1 byte wide.
- Each method can have up to 256 local variables and formal parameters. The bytecode reference these by number.
- Actually, we can have up to 65536 local vars. There is a special `wide` instruction that modifies load and store instructions to reference the high-numbered locals. Hack.
- The Java stack is 32-bits wide. Longs and doubles hence take two stack entries.

Java Byte Codes II

- The bytecodes reference data from the class' constant pool. These references are 8 or 16 bits long. To push a reference to a literal string with constant pool # 4567, use `'ldc2 4567'`. If the # is 123, use `'ldc2 123'`.

$\texttt{int}_8$	An 8-bit integer value.
$\texttt{int}_{16}$	A 16-bit integer value.
$\texttt{int}_{32}$	A 32-bit integer value.
$\texttt{CP}_8$	An 8-bit constant pool index.
$\texttt{CP}_{16}$	A 16-bit constant pool index.
`FIdx`	An 8-bit local variable index.
$\texttt{FIdx}_{16}$	A 16-bit local variable index.
`CP[`i`]`	The i:th constant pool entry.
`Var[`i`]`	The i:th variable/formal parameter in the current method.

Opcode	Mnemonic	Args	Stack	Description
0	`nop`		[] ⇒ []	
1	`aconst_null`		[] ⇒ [null]	Push null object
2	`iconst_m1`		[] ⇒ [-1]	Push -1
$3 \ldots 8$	`iconst_`n		[] ⇒ [n]	Push integer constant $n, 0 \leq n \leq 5$
$9 \ldots 10$	`lconst_`n		[] ⇒ [n]	Push long constant $n, 0 \leq n \leq 1$
$11 \ldots 13$	`fconst_`n		[] ⇒ [n]	Push float constant $n, 0 \leq n \leq 2$
$14 \ldots 15$	`dconst_`n		[] ⇒ [n]	Push double constant $n, 0 \leq n \leq 1$

Opcode	Mnemonic	Args	Stack	Description
16	`bipush`	n:$\texttt{int}_8$	[] ⇒ [n]	Push 1-byte signed integer
17	`sipush`	n:$\texttt{int}_{16}$	[] ⇒ [n]	Push 2-byte signed integer
18	`ldc1`	n:$\texttt{CP}_8$	[] ⇒ [CP[n]]	Push item from constant pool
19	`ldc2`	n:$\texttt{CP}_{16}$	[] ⇒ [CP[n]]	Push item from constant pool
20	`ldc2w`	n:$\texttt{CP}_{16}$	[] ⇒ [CP[n]]	Push long/double from constant pool

Opcode	Mnemonic	Args	Stack
$21 \ldots 25$	X`load`	n:`FIdx`	[] $\Rightarrow$ [Var[n]]
	$X \in$ {i,l,f,d,a}, Load int, long, float, double, object from local var.		
$26 \ldots 29$	`iload_`n		[] $\Rightarrow$ [Var[n]]
	Load local integer var $n, 0 \leq n \leq 3$		
$30 \ldots 33$	`lload_`n		[] $\Rightarrow$ [Var[n]]
	Load local long var $n, 0 \leq n \leq 4$		
$34 \ldots 37$	`fload_`n		[] $\Rightarrow$ [Var[n]]
	Load local float var $n, 0 \leq n \leq 4$		
$38 \ldots 41$	`dload_`n		[] $\Rightarrow$ [Var[n]]
	Load local double var $n, 0 \leq n \leq 4$		

Opcode	Mnemonic	Args	Stack
$42 \ldots 45$	`aload_`n		[] $\Rightarrow$ [Var[n]]
	Load local object var $n, 0 \leq n \leq 4$		
$46 \ldots 53$	X`load`		$[A, I] \Rightarrow [V]$
	$X \in$ {ia,la,fa,da,aa,ba, ca,sa}. Push the value V (an int, long, etc.) stored at index I of array A.		
$54 \ldots 58$	X`store`	n:`FIdx`	[Var[n]] $\Rightarrow$ []
	$X \in$ {i,l,f,d,a}, Store int, long, float, double, object to local var.		
$59 \ldots 62$	`istore_`n		[Var[n]] $\Rightarrow$ []
	Store to local integer var $n, 0 \leq n \leq 3$		
$63 \ldots 66$	`lstore_`n		[Var[n]] $\Rightarrow$ []
	Store to local long var $n, 0 \leq n \leq 4$		

Opcode	Mnemonic	Args	Stack
$67 \ldots 70$	`fstore_`n		[Var[n]] $\Rightarrow$ []
	Store to local float var $n, 0 \leq n \leq 4$		
$71 \ldots 74$	`dstore_`n		[Var[n]] $\Rightarrow$ []
	Store to local double var $n, 0 \leq n \leq 4$		
$75 \ldots 78$	`astore_`n		[Var[n]] $\Rightarrow$ []
	Store to local object var $n, 0 \leq n \leq 4$		
$79 \ldots 86$	X`store`		$[A, I, V] \Rightarrow$ []
	$X \in$ {ia,la,fa,da,aa,ba,ca,sa}. Store the value V (an int, long, etc.) at index I of array A.		
87	`pop`		$[A] \Rightarrow$ []
	Pop top of stack.		

Opcode	Mnemonic	Stack	Description
88	`pop2`	$[A, B] \Rightarrow$ []	Pop 2 elements.
89	`dup`	$[V] \Rightarrow [V, V]$	
	Duplicate top of stack.		
90	`dup_x1`	$[B, V] \Rightarrow [V, B, V]$	Duplicate.
91	`dup_x2`	$[B, C, V] \Rightarrow [V, B, C, V]$	Duplicate.
92	`dup2`	$[V, W] \Rightarrow [V, W, V, W]$	Duplicate.
93	`dup2_x1`	$[A, V, W] \Rightarrow [V, W, A, V, W]$	Duplicate.
94	`dup2_x2`	$[A, B, V, W] \Rightarrow$ $[V, W, A, B, V, W]$	Duplicate.
95	`swap`	$[A, B] \Rightarrow [B, A]$	
	Swap top stack elements.		

Opcode	Mnemonic	Stack	Description
96 . . . 99	X`add`	$[A, B] \Rightarrow [R]$	$X \in${i,l,d,f}. $R = A + B$
100 . . . 103	X`sub`	$[A, B] \Rightarrow [R]$	$X \in${i,l,d,f}. $R = A - B$
104 . . . 107	X`mul`	$[A, B] \Rightarrow [R]$	$X \in${i,l,d,f}. $R = A * B$
108 . . . 111	X`div`	$[A, B] \Rightarrow [R]$	$X \in${i,l,d,f}. $R = A/B$
112 . . . 115	X`mod`	$[A, B] \Rightarrow [R]$	$X \in${i,l,d,f}. $R = A\%B$
116 . . . 119	X`neg`	$[A] \Rightarrow [R]$	$X \in${i,l,d,f}. $R = -A$
120 . . . 121	X`shl`	$[A, B] \Rightarrow [R]$	$X \in${i,l}. $R = A << B$
122 . . . 123	X`shl`	$[A, B] \Rightarrow [R]$	$X \in${i,l}. $R = A >> B$
124 . . . 125	X`ushr`	$[A, B] \Rightarrow [R]$	$X \in${i,l}. $R = A >>> B$
126 . . . 127	X`and`	$[A, B] \Rightarrow [R]$	$X \in${i,l}. $R = A\&\&B$
128 . . . 129	X`or`	$[A, B] \Rightarrow [R]$	$X \in${i,l}. $R = A\|\|B$
130 . . . 131	X`xor`	$[A, B] \Rightarrow [R]$	$X \in${i,l}. $R = A$`xor`B

Opcode	Mnemonic	Args	Stack
133 . . . 144	$X2Y$`cnv`		$[F] \Rightarrow [T]$
	Convert F from type X to T of type Y. $X \in${i,l,f,d}, $Y \in${i,l,f,d}.		
145 . . . 147	`i2`X		$[F] \Rightarrow [T]$
	$X \in${b,c,s}. Convert integer F to byte, char, or short.		
148,149,151	X`cmp`		$[A, B] \Rightarrow [V]$
	$X \in${l,f,d}. $A > B \Rightarrow V = 1$, $A < B \Rightarrow V = -1$, $A = B \Rightarrow V = 0$. $A =$ `NaN` $\vee B =$ `NaN` $\Rightarrow V = -1$		
150,152	X`cmp`		$[A, B] \Rightarrow [V]$
	$X \in${f,d}. $A > B \Rightarrow V = 1$, $A < B \Rightarrow V = -1$, $A = B \Rightarrow V = 0$. $A =$ `NaN` $\vee B =$ `NaN` $\Rightarrow V = 1$		
153 . . . 154	`if`$\diamond$	L:`int`$_{16}$	$[A] \Rightarrow []$
	$\diamond$={eq,ne,lt,ge,gt,le}. If $A \diamond 0$ goto $L +$ `pc`.		

Opcode	Mnemonic	Args	Stack
159 . . . 164	`if_icmp`$\diamond$	L:`int`$_{16}$	$[A, B] \Rightarrow []$
	$\diamond$={eq,ne,lt,ge,gt,le}. If $A \diamond B$ goto $L +$ `pc`.		
165 . . . 166	`if_acmp`$\diamond$	L:`int`$_{16}$	$[A, B] \Rightarrow []$
	$\diamond$={eq,ne}. A, B are object refs. If $A \diamond B$ goto $L +$ `pc`.		
167	`goto`	I:`int`$_{16}$	$[] \Rightarrow []$
	Goto instruction I.		
168	`jsr`	I:`int`$_{16}$	$[] \Rightarrow []$
	Jump subroutine to instruction $I +$ `pc`.		
172 . . . 177	X`return`		$[V] \Rightarrow []$
	$X \in${i,f,l,d,a,v}. Return V.		
169	`ret`	L:`FIdx`	$[] \Rightarrow []$
	Return from subroutine. Address in local var L.		

Opcode	Mnemonic	Args	Stack
170	`tableswitch`	D:`int`$_{32}$,l,h:`int`$_{32}$,o^{h-l+1}	$[K] \Rightarrow []$
	Jump through the K:th offset. Else goto D.		
171	`lookupswitch`	D:`int`$_{32}$,n:`int`$_{32}$,$(m, o)^n$	$[K] \Rightarrow []$
	If, for one of the (m, o) pairs, $K = m$, then goto o. Else goto D.		
178	`getstatic`	F:`CP`$_{16}$	$[] \Rightarrow [V]$
	Push value V of static field F.		
180	`getfield`	F:`CP`$_{16}$	$[R] \Rightarrow [V]$
	Push value V of field F in object R.		
179	`putstatic`	F:`CP`$_{16}$	$[] \Rightarrow [V]$
	Store value V into static field F.		
181	`putfield`	F:`CP`$_{16}$	$[R, V] \Rightarrow []$
	Store value V into field F of object R.		

Opcode	Mnemonic	Args	Stack
182	`invokevirtual`	P:$\texttt{CP}_{16}$	$[R, A_1, A_2, \ldots] \Rightarrow []$
	Call virtual menthod P, with arguments $A_1 \cdots A_n$, through object reference R.		
183	`invokespecial`	P:$\texttt{CP}_{16}$	$[R, A_1, A_2, \ldots] \Rightarrow []$
	Call private/init/superclass menthod P, with arguments $A_1 \cdots A_n$, through object reference R.		
184	`invokestatic`	P:$\texttt{CP}_{16}$	$[A_1, A_2, \ldots] \Rightarrow []$
	Call static menthod P with arguments $A_1 \cdots A_n$.		
185	`invokeinterface`	P:$\texttt{CP}_{16}$,n:$\texttt{int}_{16}$	$[R, A_1, A_2, \ldots] \Rightarrow []$
	Call interface menthod P, with n arguments $A_1 \cdots A_n$, through object reference R.		
187	`new`	T:$\texttt{CP}_{16}$	$[] \Rightarrow [R]$
	Create a new object R of type T.		

Opcode	Mnemonic	Args	Stack
188	`newarray`	T:$\texttt{int}_8$	$[C] \Rightarrow [R]$
	Allocate new array R, element type T, C elements long.		
191	`athrow`		$[R] \Rightarrow [?]$
	Throw exception.		
193	`instanceof`	C:$\texttt{CP}_{16}$	$[R] \Rightarrow [V]$
	Push 1 if object R is an instance of class C. Else push 0.		
194	`monitorenter`		$[R] \Rightarrow []$
	Get lock for object R.		
195	`monitorexit`		$[R] \Rightarrow []$
	Release lock for object R.		
196	`wide`	C:$\texttt{int}_8$,I:$\texttt{FIdx}_{16}$	$[] \Rightarrow []$
	Perform opcode C on variable `Var[`I`]`. C is one of the load/store instructions.		

Opcode	Mnemonic	Args	Stack
197	`multianewarray`	T:$\texttt{CP}_{16}$,D:$\texttt{CP}_8$	$[d_1, d_2, \ldots] \Rightarrow [R]$
	Create new D-dimensional multidimensional array R. $d_1, d_2, \ldots$ are the dimension sizes.		
198	`ifnull`	L:$\texttt{int}_{16}$	$[V] \Rightarrow []$
	If $V =$ `null` goto L.		
199	`ifnonnull`	L:$\texttt{int}_{16}$	$[V] \Rightarrow []$
	If $V \neq$ `null` goto L.		
200	`goto_w`	I:$\texttt{int}_{32}$	$[] \Rightarrow []$
	Goto instruction I.		
201	`jsr_w`	I:$\texttt{int}_{32}$	$[] \Rightarrow []$
	Jump subroutine to instruction I.		

Examples

JVM Example I

```
    void spin() {
        int i;
        for (i = 0; i < 100; i++) {
                ;    // Loop body is empty
    }}
```

```
0   iconst_0    // Push int constant 0
1   istore_1    // Store into local 1 (i=0)
2   goto 8      // First time through don't
                // increment
5   iinc 1 1    // Increment local 1 by 1
                // (i++)
8   iload_1     // Push local 1 (i)
9   bipush 100  // Push int constant (100)
11  if_icmplt 5 // Compare, loop
                // if < (i < 100)
14  return      // Return void when done
```


JVM Example II

```
    void dspin() {
        double i;
        for (i = 0.0; i < 100.0; i++) {; /* Loop body is empty */ }}
```

```
0   dconst_0  // Push double constant 0.0
1   dstore_1  // Store into locals 1 and 2 (i = 0.0)
2   goto 9    // First time no incr
5   dload_1   // Push double
6   dconst_1  // Push double 1.o onto stack
7   dadd      // Add;
8   dstore_1  // Store result in locals 1 and 2
9   dload_1   // Push local
10  ldc2_w #4 // Double 100.000000
13  dcmpg
14  iflt 5    // Compare, loop if <  (i < 100.000000)
17  return    // Return void when done
```


JVM Example III

```
    double doubleLocals(double d1, double d2) {
        return d1 + d2;
    }
```

```
0  dload_1 // First argument in locals 1 and 2
1  dload_3 // Second argument in locals 3 and 4
2  dadd    // Each also uses two words on stack
3  dreturn
```


JVM Example IV

```
    int align2grain(int i, int grain) {

        return ((i + grain-1) & ~(grain-1));
    }
```

```
0   iload_1
1   iload_2
2   iadd
3   iconst_1
4   isub
5   iload_2
6   iconst_1
7   isub
8   iconst_m1
9   ixor
10  iand
11  ireturn
```

JVM Example V

```
    void useManyNumeric() {
        int i = 100; int j = 1000000;
        long l1 = 1; long l2 = 0xffffffff;
        double d = 2.2;
    }
```

```
0  bipush 100 // Push a small int
2  istore_1
3  ldc #1     // Integer 1000000; a larger
              // int value uses ldc
5  istore_2
6  lconst_1   // A tiny long value
7  lstore_3
8  ldc2_w #6  // A long 0xffffffff. A long
              // constant value.
11 lstore 5
13 ldc2_w #8  // Double 2.200000
16 dstore 7
```


JVM Example VI

```
    void whileInt() {
        int i = 0;
        while (i < 100) {
            i++;
        }
    }
```

```
0   iconst_0
1   istore_1
2   goto 8
5   iinc 1 1
8   iload_1
9   bipush 100
11  if_icmplt 5
14  return
```


JVM Example VII

```
    int lessThan100(double d) {
        if (d < 100.0) {
            return 1;
        } else {
            return -1;
        }
    }
```

```
0   dload_1
1   ldc2_w #4 // Double 100.000000
4   dcmpg     // Push 1 if d is NaN or
              //     d < 100.000000;
              // push 0 if d == 100.000000
5   ifge 10   // Branch on 0 or 1
8   iconst_1
9   ireturn
10  iconst_m1
11  ireturn
```


JVM Example VIII

```
int add12and13() {return addTwo(12, 13);}
```

```
0  aload_0          // Push this local 0 (this) onto stack
1  bipush 12        // Push int constant 12 onto stack
3  bipush 13        // Push int constant 13 onto stack
5  invokevirtual #4 // Method Example.addtwo(II)I
8  ireturn          // Return int on top of stack; it is
                    // the int result of addTwo()
```

JVM Example IX

```
Object create() {return new Object();}
```

⇓

```
0  new #1           // Class java.lang.Object
3  dup
4  invokespecial #4 // Method java.lang.Object.<init>()V
7  areturn
```


JVM Example X

```
void createBuffer() {
   int buf[]; int bsz = 100; int val=12; buf = new int[bsz];
   buf[10]=val; value = buf[11]; }
```

⇓

```
0   bipush 100   // Push bsz
2   istore_2     // Store bsz in local 2
3   bipush 12    // Push val
5   istore_3     // Store val in local 3
6   iload_2      // Push bsz...
7   newarray int // and create new int array
9   astore_1     // Store new array in buf
10  aload_1      // Push buf
11  bipush 10    // Push constant 10
13  iload_3      // Push val
14  iastore      // Store val at buf[10]
15  aload_1      // Push buf
16  bipush 11    // Push constant 11
18  iaload       // Push value at buf[11]
19  istore_3     // ...and store it in value
```


JVM Example XI

```
int chooseNear(int i) {
    switch (i) {
        case 0:  return 0;  case 2:  return 2; default: return -1;
}}
```

⇓

```
0   iload_1     // Load local 1 (argument i)
1   tableswitch 0 to 2:
    0: 28       // If i is 0, continue at 28
    1: 32       // If i is 1, continue at 34
    2: 30       // If i is 2, continue at 32
    default:34  // Otherwise, continue at 34
28  iconst_0    // i was 0; push int 0...
29  ireturn     // ...and return it
30  iconst_2    // i was 2; push int 2...
31  ireturn     // ...and return it
32  iconst_m1   // otherwise push int -1...
33  ireturn     // ...and return it
```


The Jasmin Assembler

Jasmin

```
.class public HelloWorld
.super java/lang/Object
.method public <init>()V
  aload_0
  invokenonvirtual java/lang/Object/<init>()V
  return
.end method
.method public static main([Ljava/lang/String;)V
  .limit stack 2   ; up to two items can be pushed
  ; push System.out onto the stack
  getstatic java/lang/System/out Ljava/io/PrintStream;
  ; push a string onto the stack
  ldc "Hello World!"
  ; call the PrintStream.println() method.
  invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
  return
.end method
```


Running Jasmin

The jasmin command runs Jasmin on a file. For example:

```
> jasmin myfile.j
```

assembles the file `"myfile.j"`. Jasmin looks at the .class directive contained in the file to decide where to place the output class file. So if myfile.j starts with:

```
> jasmin myfile.j
 .class mypackage/MyClass
```

then Jasmin will place the output class file `"MyClass.java"` in the subdirectory `"mypackage"` of the current directory. It will create the mypackage directory if it doesn't exist.

Directives

Directive statements are used to give Jasmin meta-level information. Directive statements consist of a directive name, and then zero or more parameters separated by spaces, then a newline. All directive names start with a "." character. The directives in Jasmin are:

```
.catch .class .end .field .implements .interface
.limit .line .method .source .super .throws .var
```

Some example directive statements are:

```
.limit stack 10
.method public myMethod()V
.class Foo\begin{itemize}
```


Instructions

An instruction statement consists of an instruction name, zero or more parameters separated by spaces, and a newline.

Jasmin uses the standard mnemonics for JVM opcodes as instruction names. For example, aload_1, bipush and iinc are all Jasmin instruction names.

Here are some examples of instruction statements:

```
ldc    "Hello World"
iinc   1 -1
bipush 10
```

Names and Types

- Class names in Jasmin should be written using the Java class file format conventions, so `java.lang.String` becomes `java/lang/String`.
- Type information is also written as they appear in class files (e.g. the descriptor I speficies an integer, `[Ljava/lang/Thread;` is an array of Threads, etc.).

Methods

- Method names are specified using a single token, e.g.

  `java/io/PrintStream/println(Ljava/lang/String;)V`

  is the method called `"println"` in the class `java.io.PrintStream`, which has the type descriptor `"(Ljava/lang/String;)V"`.
- In general, a method specification is formed of three parts: the characters before the last '/' form the class name. The characters between the last '/' and '(' are the method name. The rest of the string is the type descriptor for the method.

foo/baz/Myclass/	myMethod	(Ljava/lang/String;)V

Methods

- As another example, you would call the Java method:

```
class mypackage.MyClass {
    int foo(Object a, int b[]) { ... }
}
```

  using:

```
invokevirtual mypackage/MyClass/foo(Ljava/lang/Object;[I)I
```


Fields

- Field names are specified in Jasmin using two tokens, one giving the name and class of the field, the other giving its descriptor. For example:

  `getstatic mypackage/MyClass/my_font  Ljava/lang/Font;`

  gets the value of the field called `"my_font"` in the class `mypackage.MyClass`. The type of the field is `"Ljava/lang/Font;"` (i.e. a Font object).

File Structure

- Jasmin files start by giving information on the class being defined in the file - such as the name of the class, the name of the source file that the class originated from, the name of the superclass, etc.
- Typically, a Jasmin file starts with the three directives:

```
.source <source-file>
.class  <access-spec> <class-name>
.super  <class-name>
```


File Structure

- For example, the file defining MyClass might start with the directives:

```
.source MyClass.j
.class  public MyClass
.super  java/lang/Object
```

- `access-spec` is a list of zero or more of the following keywords:

```
public, final, super, interface, abstract
```


Field Definitions

- After the header information, the next section of the Jasmin file is a list of field definitions.
- A field is defined using the .field directive:

```
.field <access-spec> <field-name> <descriptor> [=<value>]
```

- Examples:

```
public int foo;
public static final float PI = 3.14;
```

compiles to

```
.field public foo
.field public static final PI F = 3.14
i
```


Method Definitions

- After listing the fields of the class, the rest of the Jasmin file lists methods defined by the class.
- A method is defined using the basic form:

```
.method <access-spec> <method-spec>
    <statements>
.end method
```

- Always add an explicit return at the end of the method.

```
.method foo()V
    return     ; must give a return statement
 .end method
```

Method Directives

- `.limit stack <integer>` Sets the maximum size of the operand stack required by the method.
- `.limit locals <integer>` Sets the number of local variables required by the method.
- `.var <var-number> is <name> <descriptor>` `from <label1> to <label2>` The `.var` directive is used to define the name, type descriptor and scope of a local variable number.

Method Directives – Example

```
.method foo()V
   .limit locals 1
    ; declare variable 0 as an "int Count;"
    ; whose scope is the code between Label1 and Label2

    .var 0 is Count I from Label1 to Label2
Label1:
    bipush 10
    istore_0
Label2:
    return
.end method
```


Exceptions

- `.throws <classname>` Indicates that this method can throw exceptions of the type indicated by `<classname>`. e.g.

  ```
  .throws java/io/IOException
  ```

- `.catch <classname>` `from <label1> to <label2> using <label3>`
  Appends an entry to the end of the exceptions table for the method. The entry indicates that when an exception which is an instance of `<classname>` or one of its subclasses is thrown while executing the code between `<label1>` and `<label2>`, then the runtime system should jump to `<label3>`. e.g.

```
.catch java/io/IOException from L1 to L2 using IO_Handler
```


JVM Instructions

- JVM instructions are placed between the `.method` and `.end method` directives.
- VM instructions can take zero or more parameters. Examples:

```
iinc 1 -3     ; decrement local variable 1 by 3
bipush 10     ; push the integer 10 onto the stack
pop           ; remove the top item from the stack.
```

Local variable instructions

```
ret <var-num>
aload <var-num>
astore <var-num>
dload <var-num>
dstore <var-num>
fload <var-num>
fstore <var-num>
iload <var-num>
istore <var-num>
lload <var-num>
lstore <var-num>
```

for example:

```
aload 1 ; push local variable 1 onto the stack
ret 2   ; return to the address held in local variable 2
```


The bipush, sipush and iinc instructions

```
bipush <int>
sipush <int>
```

for example:

```
bipush 100    ; push 100 onto the stack
```

The iinc instruction takes two integer parameters:

```
iinc  <var-num> <amount>
```

for example:

```
iinc 3 -10    ; subtract 10 from local variable 3
```


Branch instructions

```
goto  <label>
goto_w  <label>
if_acmpeq  <label>
if_acmpne  <label>
if_icmpeq  <label>
if_icmpge  <label>
if_icmpgt  <label>
if_icmple  <label>
if_icmplt  <label>
if_icmpne  <label>
ifeq  <label>
ifge  <label>
ifgt  <label>
ifle  <label>
iflt  <label>
ifne  <label>
```


Branch instructions. . .

```
ifnonnull  <label>
ifnull  <label>
jsr  <label>
jsr_w  <label>
```

For example:

```
Label1:
    goto Label1    ; jump to the code at Label1
                   ; (an infinite loop!)
```

Class and object operations

```
anewarray  <class>
checkcast  <class>
instanceof <class>
new        <class>
```

For example:

```
new java/lang/String   ; create a new String object
```


Method invokation

```
invokenonvirtual  <method-spec>
invokestatic      <method-spec>
invokevirtual     <method-spec>
```

For example:

```
; invokes java.io.PrintStream.println(String);
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
```


Field manipulation instructions

```
getfield  <field-spec> <descriptor>
getstatic  <field-spec> <descriptor>
putfield  <field-spec> <descriptor>
putstatic  <field-spec> <descriptor>
```

For example:

```
; get java.lang.System.out, which is a PrintStream
getstatic java/lang/System/out Ljava/io/PrintStream;
```


Array instructions

```
newarray  <array-type>
multianewarray  <array-descriptor> <num-dimensions>
```

For example:

```
newarray int
newarray short
newarray float
multianewarray [[[I 2
```

The ldc and ldc_w instructions

```
ldc  <constant>
ldc_w  <constant>
```

<constant> is either an integer, a floating point number, or a quoted string. For example:

```
ldc 1.2              ; push a float
ldc 10               ; push an int
ldc "Hello World"    ; push a String
ldc_w 3.141592654    ; push PI as a double
```


Switch instructions

```
<lookupswitch> ::=
    lookupswitch
       <int1> : <label1>
       <int2> : <label2>
        ...
        default : <default-label>

<tableswitch> ::=
    tableswitch <low>
        <label1>
        <label2>
         ...
         default : <default-label>
```


lookupswitch example

```
; If the int on the stack is 3,
; jump to L1.
; If it is 5, jump to Label2.
; Otherwise jump to DefaultLabel.
lookupswitch
        3 : L1
        5 : Ll2
  default : Default
L1:
   ... got 3
L2:
   ... got 5
Default:
   ... got something else
```


tableswitch example

```
; If the int on the stack is 0,
; jump to Label1.
; If it is 1, jump to Label2.
; Otherwise jump to DefaultLabel.
tableswitch 0
      Label1
      Label2
   default : DefaultLabel
Label1:
   ... got 0
Label2:
   ... got 1
DefaultLabel:
   ... got something else
```

Example I - Count.j

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

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

.method public static main([Ljava/lang/String;)V
   ; set limits used by this method
   .limit locals 4
   .limit stack 3
```


Example I - Count.j

```
   ;    1 - the PrintStream object held in java.lang.System.out
   getstatic java/lang/System/out Ljava/io/PrintStream;
   astore_1

   ;    2 - the integer 10 - the counter used in the loop
   bipush 10
   istore_2

   ; now loop 10 times printing out a number
```


Example I - Count.j

```
Loop:
  bipush 10     ; compute 10 - <local variable 2> ...
  iload_2
  isub
  invokestatic java/lang/String/valueOf(I)Ljava/lang/String;
  astore_3
  ; ... and print it
  aload_1    ; push the PrintStream object
  aload_3    ; push the string we just created - then ...
  invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

  iinc 2 -1    ; decrement the counter and loop
  iload_2
  ifne Loop

  return
.end method
```


Example II - Arrays.j

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

.method public <init>()V
   ...
.end method

.method public static main([Ljava/lang/String;)V
  .limit locals 2
  .limit stack 4

  ;  String[] myarray = new String[2];
  iconst_2
  anewarray java/lang/String
  astore_1  ; stores this in local variable 1
```

Example II - Arrays.j

```
  ;      myarray[0] = args[0];
  aload_1     ; push my array on the stack
  iconst_0
  aload_0     ; push the array argument to main() on the stack
  iconst_0
  aaload      ; get its zero'th entry
  aastore     ; and store it in my zero'th entry

  ; now print out myarray[0]
  getstatic java/lang/System/out Ljava/io/PrintStream;
  aload_1
  iconst_0
  aaload
  invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

  return
.end method
```


Example III - Newarray.j

```
.class public NewArray
.method public static main([Ljava/lang/String;)V
   .limit stack 4
   .limit locals 2

   iconst_2
   newarray boolean        ; boolean b[] = new boolean[2]
   astore_1                ; stores it in local var 1

   aload_1                 ; b[0] = true;
   iconst_0
   iconst_1
   bastore

   return
.end method
```


Example IV - Switch.j

```
.method public static main([Ljava/lang/String;)V
   .limit stack 3
   iconst_1
   lookupswitch
        1 : Hello
        2 : Goodbye
     default : Foo
Hello:
Goodbye:
Foo:
   return
.end method
```


Example IV - Switch.j

```
.method public static main([Ljava/lang/String;)V
   .limit stack 3
   iconst_1
   tableswitch 0
        Hello
        Goodbye
     default : Foo
Hello:
Goodbye:
Foo:
   return
.end method
```

Example V - CheckCast.j

```
.class examples/Checkcast
.super java/lang/Object

....

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

   ; push System.out onto the stack
   getstatic java/lang/System/out Ljava/io/PrintStream;

   ; check that it is a PrintStream
   checkcast java/io/PrintStream

   return
.end method
```


Example VI - Abs.j

```
;   public static int abs(int x);
;     if (x < 0) x = -x
;     return x
.method public static abs(I)I
       .limit locals 1
       .limit stack  2

       iload_0    ; x
       dup        ; x x
       ifge  done ; x    is x < 0?
       ineg       ; -x   yes. x = -x
done:
       ireturn
.end method
```


Example VII - Dist.j

```
;    public static int dist(int x, int y)
;       returns abs(x - y)
.method public static dist(II)I
       .limit locals 2
       .limit stack 2

       iload_0   ; x
       iload_1   ; y x
       isub      ; y - x

       invokestatic java/lang/Math/abs(I)I
       ireturn
.end method
```


Readings and References

- Jasmin, the Java assembler: http://jasmin.sourceforge.net.
- The information on Jasminhas been shamelessly stolen from Jonathan Meyer's Jasmin pages, http://mrl.nyu.edu/~meyer/jasmin/. Additional examples are from http://www.csam.montclair.edu/~bredlau/jasmin/JVM.html.
- http://bcel.sourceforge.net/JasminVisitor.java is a program that uses the BCEL bytecode editor to generate Jasmin assembly code from a Java class file.