JASMIN USER GUIDE

Jonathan Meyer, July 1996

About This Document

This guide describes the rules and syntax used in Jasmin, and how to run Jasmin. Note that this document doesn't explain the Java Virtual Machine itself, or give syntax notes for every instruction known to Jasmin. See the Java Virtual Machine specification for more information on the JVM.

You should also see:

About Jasmin
: describes what Jasmin is, who might find it interesting, etc. Includes an example piece of code.

Jasmin Instructions
: Describes the syntax of JVM instructions in Jasmin.

What is Jasmin?

Jasmin is an assembler for the Java Virtual Machine. It takes ASCII descriptions of Java classes, written in a simple assembler-like syntax using the Java Virtual Machine instruction set. It converts them into binary Java class files, suitable for loading by a Java runtime system.

Jasmin was originally created as a companion to the book "Java Virtual Machine", written by Jon Meyer and Troy Downing and published by O'Reilly Associates. The book is now out of print. Jasmin survives as a SourceForge Open Source project.

Jasmin Design

Jasmin is designed as a simple assembler. It has a clean easy-to-learn syntax with few bells and whistles. Where possible, Jasmin adopts a one-to-one mapping between its syntax and the conventions followed by Java class files. For example, package names in Jasmin are delimited with the '/' character (e.g. "java/lang/String") used by the class file format, instead of the '.' character (java.lang.String) used in the Java language.

The Jasmin assembler does little compile-time processing or checking of the input code. For example, it doesn't check that classes you reference actually exist, or that your type descriptors are well formed. Jasmin also lacks many of the feautures found in full macro assemblers. For example, it doesn't inline mathematical expressions, perform variable substitutions, or support macros.

On the other hand, using Jasmin you can quickly try out nearly all of the features of the Java Virtual Machine, including methods, fields, subroutines, exception handlers, and so on. The Jasmin syntax is also readable and compact.

Running Jasmin

The `jasmin.jar` file is an executable JAR file that runs Jasmin. For example:

```
java -jar jasmin.jar myfile.j
```

or

```
java Jasmin myfile.j
```

(if jasmin.jar is already in your classpath)

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

```
.class mypackage/MyClass
```

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

You can use the "-d" option to tell jasmin to place the output in an alternative directory. For example,

```
java -jar jasmin.jar -d /tmp myfile.j
```

will place the output in /tmp/mypackage/MyClass.class.

Finally, you can use the "-g" option to tell Jasmin to include line number information (used by debuggers) in the resulting .class file. Jasmin will number the lines in the Jasmin source file that JVM instructions appear on. Then, if an error occurs, you can see what instruction in the Jasmin source caused the error. Note that specifying "-g" causes any .line directives within the Jasmin file to be ignored.

Statements

Jasmin source files consists of a sequence of newline-separated statements. There are three types of statement:

- directives
- instructions
- labels

Directives and instructions can take *parameters*. These parameters are placed on the same line as the directive or instruction, separated by spaces.

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
```

The parameters used by each directive are described in more detail later in the document.

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
```

See Jasmin Instructions for more details on the syntax of instructions in Jasmin.

Labels

A Jasmin label statement consists of a name followed by a ':', and a newline. For example:

```
Foo:

Label:
```

Label names cannot start with a numeric digit, and cannot contain any of the special characters:

```
= : . " -
```

You cannot use directive names or instruction names as labels. Other than that, there are few restrictions on label names. For example, you could use the label:

```
#_1:
```

Labels can only be used within method definitions. The names are local to that method.

The Jasmin Tokenizer

Jasmin tokenizes its input stream, splitting the stream into tokens by looking for whitespace characters (spaces, tabs and newlines). The tokenizer looks for:

- directive names
- instruction names
- labels
- comments
- type descriptor names
- class names
- numbers and quoted strings
- etc.

The rules used by the tokenizer are described below:

Comments

A comment is a token that starts with a ';' character, and terminates with the newline character at the end of the line.

Note that the semicolon must be preceded by a whitespace character (a space, tab, newline), i.e. embedded semicolons are ignored. For example,

```
abc;def
```

is treated as a single token "abc;def", and

```
Ljava/lang/String;
```

is the token "Ljava/lang/String;", whereas

```
foo ; baz ding
```

is the token "foo" followed by a comment "baz ding".

Numbers and Strings

In Jasmin, only simple decimal and integer numeric formats are recognized. Floats in scientific or exponent format are not yet supported. Character codes and octal aren't currently supported either. This means you can have:

```
1, 123, .25, 0.03, 0xA
```

but not

```
1e-10, 'a', '\u123'
```

Quoted strings are also very basic. The full range of backslash escape sequences are not supported yet, although "\n" and "\t" are.

Class Names

Class names in Jasmin should be written using the Java class file format conventions, so java.lang.String becomes java/lang/String.

Type Descriptors

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" (i.e. it takes a String and returns no result). 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
---------------         ---------------------
      |         --------         |
      |            |             |
    class        method       descriptor
```

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).

FILES

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>
```

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

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

.source directive

The .source directive is optional. It specifies the value of the "SourceFile" attribute for the class file. (This is used by Java to print out debugging info if something goes wrong in one of the methods in the class). If you generated the Jasmin file automatically (e.g. as the result of compiling a file written in another syntax) you should use the .source directive to tell Java the name of the originating file. Note that the source file name should not include any pathname. Use "foo.src" but not "/home/user/foo.src".

If no .source directive is given, the name of the Jasmin file you are compiling is used instead as the SourceFile attribute instead.

.class and .super directives

The .class and .super directive tell the JVM the name of this class and its superclass. These directives take parameters as follows:

<class-name>
: is the full name of the class, including any package names. For example foo/baz/MyClass.

<access-spec>
: defines access permissions and other attributes for the class. This is a list of zero or more of the following keywords:

    public, final, super, interface, abstract

.interface directive

Note that, instead of using the directive .class, you can alternatively use the directive .interface. This has the same syntax as the .class directive, but indicates that the Jasmin file is defining a Java interface. e.g.

```
.interface public foo
```

.implements directive

After .source, .class and .super, you can list the interfaces that are implemented by the class you are defining, using the .implements directive. The syntax of .implements is:

```
.implements <class-name>
```

where <class-name> has the same format as was used by .class and .super. For example:

```
.class foo
.super java/lang/Object
.implements Edible
.implements java/lang/Throwable
```

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> ]
```

where:

<access-spec>
: is one of the keywords:

    public, private, protected, static, final, volatile, transient

<field-name>
: is the name of the field.

<descriptor>
: is its type descriptor.

<value>
: is an integer, a quoted string or a decimal number, giving the initial value of the field (for final fields).

For example, the Java field definition:

```
public int foo;
```

becomes

```
.field public foo I
```

whereas the constant:

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

becomes

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

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
```

where:

<access-spec>
: is one of the keywords: public, private, protected, static, final, synchronized, native, abstract

<method-spec>
: is the name and type descriptor of the method.

<statements>
: is the code defining the body of the method.

Method definitions cannot be nested. Also, Jasmin does not insert an implicit 'return' instruction at the end of a method. It is up to you to ensure that your methods return cleanly. So the most basic Jasmin method is something like:

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

Method Directives

The following directives can be used only within method definitions:

`.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.

`.line <integer>`

This is used to tag the subsequent instruction(s) with a line number. Debuggers use this information, together with the name of the source file (see .source above) to show at what line in a method things went wrong. If you are generating Jasmin files by compiling a source file, this directive lets you indicate what line numbers in the source file produced corrosponding Jasmin instructions. For example:

```
.method foo()V
.line 5
    bipush 10    // these instructions generated from line 5
    istore_2     // of the source file.
.line 6
    ...
```

`.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. This information is used by debuggers so that they can be more helpful when printing out the values of local variables (rather than printing just a local variable number, the debugger can actually print out the name of the variable). For 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
```

`.throws <classname>`

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

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

This information isn't required by Java runtime systems, but it is used by the Java compiler to check that methods either catch exceptions they can cause, or declare that they throw them.

`.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
```

If classname is the keyword "all", then exceptions of any class are caught by the handler.

Abstract Methods

To declare an abstract method, write a method with no body. e.g.

```
.method abstract myAbstract()V
.end method
```

note that abstract methods can have .throws directives, e.g.

```
.method abstract anotherAbstract()V
    .throws java/io/IOException
.end method
```

Instructions

JVM instructions are placed between the `.method` and `.end method` directives. VM instructions can take zero or more parameters, depending on the type of instruction used. Some example instructions are shown below:

```
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.
```

See Jasmin Instructions for more details on the syntax of instructions in Jasmin.
