

D. CLEmitter and the JVM Instruction Set¹

D.1 Introduction

Chapter 2 presented a flavor of **CLEmitter**, a high-level interface for programmatically creating Java class files; through examples, we introduced our readers to some of the methods supported by the interface. In this appendix, we will describe the interface in its entirety.

We will first discuss some aspects of ClassFile structure that one must know in order to be able to use the **CLEmitter** interface. We will then describe the methods from **CLEmitter** that support creating a Java class representation in memory and converting the representation into a `java.lang.Class` representation, both in-memory and on the file system. We will finally provide a group-wise summary of the JVM instruction set, along with a description of **CLEmitter** methods to invoke to add instructions from that group to methods.

D.2 Names and Descriptors

Class and interface names that appear in the ClassFile structure are always represented in a fully qualified form, with identifiers making up the fully qualified name separated by forward slashes (`'/'`)². This is the so-called internal form of class or interface names. For example, the name of class **Thread** in internal form is `java/lang/Thread`.

The JVM expects the types of fields and methods to be specified in a certain format called descriptors.

A field descriptor is a string representing the type of a class, instance, or local variable. It is a series of characters generated by the grammar³:

FieldDescriptor → FieldType

ComponentType → FieldType

¹ Supported in J2SE 1.5.

² This is different from the familiar syntax of fully qualified names, where the identifiers are separated by periods (`'.'`).

³ This is the so-called EBNF (Extended Backus Naur Form) notation for describing the syntax of languages.

FieldType → BaseType | ObjectType | ArrayType

BaseType → **B** | **C** | **D** | **F** | **I** | **J** | **S** | **Z**

ObjectType → **L** <class name> ; // class name is in internal form

ArrayType → [ComponentType

The interpretation of the base types is shown in the table below:

BaseType Character	Type
B	byte
C	char
D	double
F	float
I	int
J	long
S	short
Z	boolean

For example, the table below indicates the field descriptors for various field declarations:

Field	Descriptor
<code>int i;</code>	<code>I</code>
<code>Object o;</code>	<code>Ljava/lang/Object;</code>
<code>double[][][] d;</code>	<code>[[[D</code>
<code>Long[][] l;</code>	<code>[[Ljava/lang/Long;</code>

A method descriptor is a string that represents the types of the parameters that the method takes and the type of the value that it returns. It is a series of characters generated by the grammar:

MethodDescriptor → ({ParameterDescriptor}) ReturnDescriptor

ParameterDescriptor → FieldType

ReturnDescriptor → FieldType | **V**

For example, the table below indicates the method descriptors for various constructor and method declarations:

Constructor/Method	Descriptor
--------------------	------------

Appendix D - 2

<code>public Queue()</code>	<code>()V</code>
<code>public File[] listFiles()</code>	<code>() [Ljava/io/File;</code>
<code>public Boolean isPrime(int n)</code>	<code>(I)Ljava/lang/Boolean;</code>
<code>public static void main(String[] args)</code>	<code>([Ljava/lang/String;)V</code>

D.3 CLEmitter Interface

In this section, we will describe the methods from the `CLEmitter` interface that support: creating a Java class representation in memory; adding inner classes, fields, methods, exception handlers, and attributes to the class; and converting the in-memory representation of the class to a `java.lang.Class` representation, both in-memory and on the file system.

D.3.1 Instantiate CLEmitter

In order to create a class, one must create an instance of `CLEmitter` as the first step in the process. All subsequent steps involve sending an appropriate message to that instance. Each instance corresponds to a single class. An instance of `CLEmitter` can be created using the constructor:

```
public CLEmitter()
```

D.3.2 Set Destination Directory for the Class

The destination directory for the class file can be set by sending to the `CLEmitter` instance the following message:

```
public void destinationDir(String destDir)
```

where `destDir` is the directory where the class file will be written. If the class that is being created specifies a package, then the class file will be written to the directory obtained by appending the package name to the destination directory. If a destination directory is not set explicitly, the default is ".", the current directory.

D.3.3 Add Class

A class can be added by sending to the `CLEmitter` instance the following message:

```
public void addClass(ArrayList<String> accessFlags,
```

Appendix D - 3

```
String thisClass,  
String superClass,  
ArrayList<String> superInterfaces,  
boolean isSynthetic)
```

where **accessFlags**⁴ is a list of class access and property flags, **thisClass** is the name of the class in internal form, **superClass** is the name of the super class in internal form, **superInterfaces** is a list of direct super interfaces of the class in internal form, and **isSynthetic** specifies whether the class appears in source code.

If the class being added is an interface, **accessFlags** must contain appropriate modifiers, **superInterfaces** must contain the names of the interface's super interfaces (if any) in internal form, and **superClass** must always be `java/lang/Object`.

D.3.4 Add Inner Classes

While an inner class *C* is just another class and can be created using the **CLEmitter** interface, the parent class *P* that contains the class *C* has to be informed about *C*, which can be done by sending the **CLEmitter** instance for *P* the following message:

```
public void addInnerClass(ArrayList<String> accessFlags,  
                          String innerClass,  
                          String outerClass,  
                          String innerName)
```

where **accessFlags** is a list of inner class access and property flags, **innerClass** is the name of the inner class in internal form, **outerClass** is the name of the outer class in internal form, and **innerName** is the simple name of the inner class.

D.3.5 Add Fields

After a class is added, fields can be added to the class by sending to the **CLEmitter** instance the following message:

```
public void addField(ArrayList<String> accessFlags,  
                    String name,  
                    String type,  
                    boolean isSynthetic)
```

⁴ Note that the **CLEmitter** expects the access and property flags as **Strings** and internally translates them to a mask of flags. For example, “public” is translated to **ACC_PUBLIC (0x0001)**.

where **accessFlags** is a list of field access and property flags, **name** is the name of the field, **type** is the type descriptor for the field, and **isSynthetic** specifies whether the field appears in source code.

A **final** field of type **int**, **short**, **byte**, **char**, **long**, **float**, **double**, or **String** with an initialization must be added to the class by sending to the **CLEmitter** instance the respective message from the list of messages below:⁵

```
public void addField(ArrayList<String> accessFlags,  
                    String name,  
                    String type,  
                    boolean isSynthetic,  
                    int i)
```

```
public void addField(ArrayList<String> accessFlags,  
                    String name,  
                    String type,  
                    boolean isSynthetic,  
                    float f)
```

```
public void addField(ArrayList<String> accessFlags,  
                    String name,  
                    String type,  
                    boolean isSynthetic,  
                    long l)
```

```
public void addField(ArrayList<String> accessFlags,  
                    String name,  
                    String type,  
                    boolean isSynthetic,  
                    double d)
```

```
public void addField(ArrayList<String> accessFlags,  
                    String name,  
                    String type,  
                    boolean isSynthetic,  
                    String s)
```

The last parameter in each of the above messages is the value of the field. Note that the JVM treats **short**, **byte**, and **char** types as **int**.

⁵ The **field_info** structure for such fields must specify a **ConstantValueAttribute** reflecting the value of the constant, and these **addField()** variants take care of that.
Appendix D - 5

D.3.6 Add Method

A method can be added to the class by sending to the **CLEmitter** instance the following message:

```
public void addMethod(ArrayList<String> accessFlags,  
                    String name,  
                    String descriptor,  
                    ArrayList<String> exceptions,  
                    boolean isSynthetic)
```

where **accessFlags** is a list of method access and property flags, **name** is the name⁶ of the method, **descriptor** is the method descriptor, **exceptions** is a list of exceptions in internal form that this method throws, and **isSynthetic** specifies whether the method appears in source code.

D.3.7 Add Exception Handlers to Method

An exception handler to code a **try-catch** block, can be added to a method by sending to the **CLEmitter** instance the following message:

```
public void addExceptionHandler(String startLabel,  
                               String endLabel,  
                               String handlerLabel,  
                               String catchType)
```

where **startLabel** marks the beginning of the **try** block, **endLabel** marks the end of the **try** block, **handlerLabel** marks the beginning of a **catch** block, and **catchType** specifies the exception that is to be caught in internal form. If **catchType** is null, this exception handler is called for all exceptions; this is used to implement **finally**.

createLabel() and **addLabel(String label)** can be invoked on the **CLEmitter** instance to create unique labels and for adding them to mark instructions in the code indicating where to jump to.

A method can specify as many exception handlers as there are exceptions that are being caught in the method.

D.3.8 Add Optional Method, Field, Class, Code Attributes

⁶ Instance initializers must have the name **<init>** and static initializers must have the name **<clinit>**.

Attributes are used in the `ClassFile` (**CLFile**), `field_info` (**CLFieldInfo**), `method_info` (**CLMethodInfo**), and `Code_attribute` (**CLCodeAttribute**) structures of the class file format. While there are many kinds of attributes, only some are mandatory; these include: `InnerClasses_attribute` (class attribute), `Synthetic_attribute` (class, field, and method attribute), `Code_attribute` (method attribute), and `Exceptions_attribute` (method attribute).

CLEmitter implicitly adds the required attributes to the appropriate structures. The optional attributes can be added by sending to the **CLEmitter** instance one of the following messages:

```
public void addMethodAttribute(CLAttributeInfo attribute)
public void addFieldAttribute(CLAttributeInfo attribute)
public void addClassAttribute(CLAttributeInfo attribute)
public void addCodeAttribute(CLAttributeInfo attribute)
```

Note that for adding optional attributes, you need access to the constant pool table, which the **CLEmitter** exposes through its `constantPool()` method, and also the program counter `pc`, which it exposes through its `pc()` method. The abstractions for all the attributes (code, method, field, and class) are defined in the **CLAttributeInfo** class.

D.3.9 Checking for Errors

The caller, at any point during the creation of the class, can check if there was an error, by sending to the **CLEmitter** the following message:

```
public boolean errorHasOccurred()
```

D.3.10 Write Class File

The in-memory representation of the class can be written to the file system by sending to the **CLEmitter** instance the following message:

```
public void write()
```

The destination directory for the class is either the default (current) directory or the one specified by invoking `destinationDir(String destDir)` method. If the class specifies a package, then the class will be written to the directory obtained by appending the package information to the destination directory.

Alternatively, the representation can be converted to `java.lang.Class` representation in memory by sending to the **CLEmitter** instance the following message:

Appendix D - 7

```
public Class toClass()
```

D.4 JVM Instruction Set

The instructions supported by the JVM can be categorized into various groups: object, field, method, array, arithmetic, bit, comparison, conversion, flow control, load and store, and stack instructions. In this section, we provide a brief summary of the instructions belonging to each group. The summary includes the mnemonic⁷ for the instruction, a one-line description of what the instruction does, and how the instruction affects the operand stack. For each set of instructions, we also specify the `CLEmitter` method to invoke while generating class files, to add instructions from that set to the code section of methods.

For each instruction, we represent⁸ the operand stack as follows:

$$\dots, value1, value2 \Rightarrow \dots, result$$

which means that the instruction begins by having *value2* on top of the operand stack with *value1* just beneath it. As a result of the execution of the instruction, *value1* and *value2* are popped from the operand stack and replaced by *result* value, which has been calculated by the instruction. The remainder of the operand stack, represented by an ellipsis (...), is unaffected by the instruction's execution.

Values of types `long` and `double` are represented by a single entry on the operand stack.

D.4.1 Object Instructions

Mnemonic	Operation	Operand Stack
<code>new</code>	Create new object	$\dots \Rightarrow \dots, objectref$
<code>instanceof</code>	Determine if object is of given type	$\dots, objectref \Rightarrow \dots, result$
<code>checkcast</code>	Check whether object is of given type	$\dots, objectref \Rightarrow \dots, objectref$

The above instructions can be added to the code section of a method by sending the `CLEmitter` instance the following message:

```
public void addReferenceInstruction(int opcode,  
                                   String type)
```

⁷ These mnemonics (also called opcodes) are defined in `jminusminus.CIConstants`.

⁸ This is the representation used in The Java Virtual Machine specification, 2nd edition. Appendix D - 8

where **opcode** is the mnemonic of the instruction to be added, and **type** is the reference type in internal form or a type descriptor if it is an array type.

D.4.2 Field Instructions

Mnemonic	Operation	Operand Stack
getfield	Get field from object	..., <i>objectref</i> ⇒ ..., <i>value</i>
putfield	Set field in object	..., <i>objectref</i> , <i>value</i> ⇒ ...
getstatic	Get static field from class	... ⇒ ..., <i>value</i>
putstatic	Set static field in class	..., <i>value</i> ⇒ ...

The above instructions can be added to the code section of a method by sending the **CLEmitter** instance the following message:

```
public void addMemberAccessInstruction(int opcode,
                                     String target,
                                     String name,
                                     String type)
```

where **opcode** is the mnemonic of the instruction to be added, **target** is the name (in internal form) of the class to which the field belongs, **name** is the name of the field, and **type** is the type descriptor of the field.

D.4.3 Method Instructions

Mnemonic	Operation	Operand Stack
invokevirtual	Invoke instance method; dispatch based on class	..., <i>objectref</i> , [<i>arg1</i> , [<i>arg2</i> ...]] ⇒ ...
invokeinterface	Invoke interface method	..., <i>objectref</i> , [<i>arg1</i> , [<i>arg2</i> ...]] ⇒ ...
invokespecial	Invoke instance method; special handling for superclass, private, and instance initialization method invocations	..., <i>objectref</i> , [<i>arg1</i> , [<i>arg2</i> ...]] ⇒ ...
invokestatic	Invoke a class (static) method	..., [<i>arg1</i> , [<i>arg2</i> ...]] ⇒ ...

invokedynamic	Invoke instance method; dispatch based on class	<i>..., objectref, [arg1, [arg2 ...]] ⇒ ...</i>
----------------------	---	---

The above instructions can be added to the code section of a method by sending the **CLEmitter** instance the following message:

```
public void addMemberAccessInstruction(int opcode,
                                     String target,
                                     String name,
                                     String type)
```

where **opcode** is the mnemonic of the instruction to be added, **target** is the name (in internal form) of the class to which the method belongs, **name** is the name of the method, and **type** is the type descriptor of the method.

Mnemonic	Operation	Operand Stack
ireturn	Return int from method	<i>..., value ⇒ [empty]</i>
lreturn	Return long from method	<i>..., value ⇒ [empty]</i>
freturn	Return float from method	<i>..., value ⇒ [empty]</i>
dreturn	Return double from method	<i>..., value ⇒ [empty]</i>
areturn	Return reference from method	<i>..., objectref ⇒ [empty]</i>
return	Return void from method	<i>... ⇒ [empty]</i>

The above instructions can be added to the code section of a method by sending the **CLEmitter** instance the following message:

```
public void addNoArgInstruction(int opcode)
```

where **opcode** is the mnemonic of the instruction to be added.

D.4.4 Array Instructions

Mnemonic	Operation	Operand Stack
newarray	Create new array	<i>..., count ⇒ ..., arrayref</i>
anewarray	Create new array of reference type	<i>..., count ⇒ ..., arrayref</i>

The above instructions can be added to the code section of a method by sending the **CLEmitter** instance the following message:

```
public void addArrayInstruction(int opcode,
```

Appendix D - 10

String type)

where **opcode** is the mnemonic of the instruction to be added, and **type** is the type descriptor of the array.

Mnemonic	Operation	Operand Stack
multianewarray	Create new multidimensional array	<i>..., count1, [count2, ...] ⇒ ..., arrayref</i>

The above instruction can be added to the code section of a method by sending the **CLEmitter** instance the following message:

```
public void addMULTIANEWARRAYInstruction(String type,
                                         int dim)
```

where **type** is the type descriptor of the array, and **dim** is the number of dimensions.

Mnemonic	Operation	Operand Stack
baload	Load byte or boolean from array	<i>..., arrayref, index ⇒ ..., value</i>
caload	Load char from array	<i>..., arrayref, index ⇒ ..., value</i>
saload	Load short from array	<i>..., arrayref, index ⇒ ..., value</i>
iaload	Load int from array	<i>..., arrayref, index ⇒ ..., value</i>
laload	Load long from array	<i>..., arrayref, index ⇒ ..., value</i>
faload	Load float from array	<i>..., arrayref, index ⇒ ..., value</i>
daload	Load double from array	<i>..., arrayref, index ⇒ ..., value</i>
aaload	Load from reference array	<i>..., arrayref, index ⇒ ..., value</i>
bastore	Store into byte or boolean array	<i>..., arrayref, index, value ⇒ ...</i>
castore	Store into char array	<i>..., arrayref, index, value ⇒ ...</i>
sastore	Store into short array	<i>..., arrayref, index, value ⇒ ...</i>
iastore	Store into int array	<i>..., arrayref, index, value ⇒ ...</i>
lastore	Store into long array	<i>..., arrayref, index, value ⇒ ...</i>
fastore	Store into float array	<i>..., arrayref, index, value ⇒ ...</i>
dastore	Store into double array	<i>..., arrayref, index, value ⇒ ...</i>
aastore	Store into reference array	<i>..., arrayref, index, value ⇒ ...</i>
arraylength	Get length of array	<i>..., arrayref ⇒ ..., length</i>

The above instructions can be added to the code section of a method by sending the **CLEmitter** instance the following message:

```
public void addNoArgInstruction(int opcode)
```

Appendix D - 11

where **opcode** is the mnemonic of the instruction to be added.

D.4.5 Arithmetic Instructions

Mnemonic	Operation	Operand Stack
iadd	Add int	<i>..., value1, value2 ⇒ ..., result</i>
ladd	Add long	<i>..., value1, value2 ⇒ ..., result</i>
fadd	Add float	<i>..., value1, value2 ⇒ ..., result</i>
dadd	Add double	<i>..., value1, value2 ⇒ ..., result</i>
isub	Subtract int	<i>..., value1, value2 ⇒ ..., result</i>
lsub	Subtract long	<i>..., value1, value2 ⇒ ..., result</i>
fsub	Subtract float	<i>..., value1, value2 ⇒ ..., result</i>
dsub	Subtract double	<i>..., value1, value2 ⇒ ..., result</i>
imul	Multiply int	<i>..., value1, value2 ⇒ ..., result</i>
lmul	Multiply long	<i>..., value1, value2 ⇒ ..., result</i>
fmul	Multiply float	<i>..., value1, value2 ⇒ ..., result</i>
dmul	Multiply double	<i>..., value1, value2 ⇒ ..., result</i>
idiv	Divide int	<i>..., value1, value2 ⇒ ..., result</i>
ldiv	Divide long	<i>..., value1, value2 ⇒ ..., result</i>
fdiv	Divide float	<i>..., value1, value2 ⇒ ..., result</i>
ddiv	Divide double	<i>..., value1, value2 ⇒ ..., result</i>
irem	Remainder int	<i>..., value1, value2 ⇒ ..., result</i>
lrem	Remainder long	<i>..., value1, value2 ⇒ ..., result</i>
frem	Remainder float	<i>..., value1, value2 ⇒ ..., result</i>
drem	Remainder double	<i>..., value1, value2 ⇒ ..., result</i>
ineg	Negate int	<i>..., value ⇒ ..., result</i>
lneg	Negate long	<i>..., value ⇒ ..., result</i>
fneg	Negate float	<i>..., value ⇒ ..., result</i>
dneg	Negate double	<i>..., value ⇒ ..., result</i>

The above instructions can be added to the code section of a method by sending the **CLEmitter** instance the following message:

```
public void addNoArgInstruction(int opcode)
```

where **opcode** is the mnemonic of the instruction to be added.

D.4.6 Bit Instructions

Mnemonic	Operation	Operand Stack
ishl	Shift left int	..., <i>value1</i> , <i>value2</i> ⇒ ..., <i>result</i>
ishr	Arithmetic shift right int	..., <i>value1</i> , <i>value2</i> ⇒ ..., <i>result</i>
iushr	Logical shift right int	..., <i>value1</i> , <i>value2</i> ⇒ ..., <i>result</i>
lshl	Shift left long	..., <i>value1</i> , <i>value2</i> ⇒ ..., <i>result</i>
lshr	Arithmetic shift right long	..., <i>value1</i> , <i>value2</i> ⇒ ..., <i>result</i>
lushr	Logical shift right long	..., <i>value1</i> , <i>value2</i> ⇒ ..., <i>result</i>
ior	Boolean OR int	..., <i>value1</i> , <i>value2</i> ⇒ ..., <i>result</i>
lor	Boolean OR long	..., <i>value1</i> , <i>value2</i> ⇒ ..., <i>result</i>
iand	Boolean AND int	..., <i>value1</i> , <i>value2</i> ⇒ ..., <i>result</i>
land	Boolean AND long	..., <i>value1</i> , <i>value2</i> ⇒ ..., <i>result</i>
ixor	Boolean XOR int	..., <i>value1</i> , <i>value2</i> ⇒ ..., <i>result</i>
lxor	Boolean XOR long	..., <i>value1</i> , <i>value2</i> ⇒ ..., <i>result</i>

The above instructions can be added to the code section of a method by sending the **CLEmitter** instance the following message:

```
public void addNoArgInstruction(int opcode)
```

where **opcode** is the mnemonic of the instruction to be added.

D.4.7 Comparison Instructions

Mnemonic	Operation	Operand Stack
dcmpg	Compare double ; result is 1 if at least one value is NaN	..., <i>value1</i> , <i>value2</i> ⇒ ..., <i>result</i>
dcmpl	Compare double ; result is -1 if at least one value is NaN	..., <i>value1</i> , <i>value2</i> ⇒ ..., <i>result</i>
fcmpg	Compare float ; result is 1 if at least one value is NaN	..., <i>value1</i> , <i>value2</i> ⇒ ..., <i>result</i>
fcmpl	Compare float ; result is -1 if at least one value is NaN	..., <i>value1</i> , <i>value2</i> ⇒ ..., <i>result</i>
lcmp	Compare long	..., <i>value1</i> , <i>value2</i> ⇒ ..., <i>result</i>

The above instructions can be added to the code section of a method by sending the **CLEmitter** instance the following message:

```
public void addNoArgInstruction(int opcode)
```

where `opcode` is the mnemonic of the instruction to be added.

D.4.8 Conversion Instructions

Mnemonic	Operation	Operand Stack
<code>i2b</code>	Convert <code>int</code> to <code>byte</code>	<i>..., value ⇒ ..., result</i>
<code>i2c</code>	Convert <code>int</code> to <code>char</code>	<i>..., value ⇒ ..., result</i>
<code>i2s</code>	Convert <code>int</code> to <code>short</code>	<i>..., value ⇒ ..., result</i>
<code>i2l</code>	Convert <code>int</code> to <code>long</code>	<i>..., value ⇒ ..., result</i>
<code>i2f</code>	Convert <code>int</code> to <code>float</code>	<i>..., value ⇒ ..., result</i>
<code>i2d</code>	Convert <code>int</code> to <code>double</code>	<i>..., value ⇒ ..., result</i>
<code>l2f</code>	Convert <code>long</code> to <code>float</code>	<i>..., value ⇒ ..., result</i>
<code>l2d</code>	Convert <code>long</code> to <code>double</code>	<i>..., value ⇒ ..., result</i>
<code>l2i</code>	Convert <code>long</code> to <code>int</code>	<i>..., value ⇒ ..., result</i>
<code>f2d</code>	Convert <code>float</code> to <code>double</code>	<i>..., value ⇒ ..., result</i>
<code>f2i</code>	Convert <code>float</code> to <code>int</code>	<i>..., value ⇒ ..., result</i>
<code>f2l</code>	Convert <code>float</code> to <code>long</code>	<i>..., value ⇒ ..., result</i>
<code>d2i</code>	Convert <code>double</code> to <code>int</code>	<i>..., value ⇒ ..., result</i>
<code>d2l</code>	Convert <code>double</code> to <code>long</code>	<i>..., value ⇒ ..., result</i>
<code>d2f</code>	Convert <code>double</code> to <code>float</code>	<i>..., value ⇒ ..., result</i>

The above instructions can be added to the code section of a method by sending the `CLEmitter` instance the following message:

```
public void addNoArgInstruction(int opcode)
```

where `opcode` is the mnemonic of the instruction to be added.

D.4.9 Flow Control Instructions

Mnemonic	Operation	Operand Stack
<code>ifeq</code>	Branch if <code>int</code> comparison value == 0 true	<i>..., value ⇒ ...</i>
<code>ifne</code>	Branch if <code>int</code> comparison value != 0 true	<i>..., value ⇒ ...</i>
<code>iflt</code>	Branch if <code>int</code> comparison value < 0 true	<i>..., value ⇒ ...</i>
<code>ifgt</code>	Branch if <code>int</code> comparison value > 0 true	<i>..., value ⇒ ...</i>
<code>ifle</code>	Branch if <code>int</code> comparison value <= 0 true	<i>..., value ⇒ ...</i>
<code>ifge</code>	Branch if <code>int</code> comparison value >= 0 true	<i>..., value ⇒ ...</i>
<code>ifnull</code>	Branch if <code>reference</code> is null	<i>..., value ⇒ ...</i>
<code>ifnonnull</code>	Branch if <code>reference</code> is not null	<i>..., value ⇒ ...</i>

if_icmpeq	Branch if int comparison value1 == value2 true	..., value1, value2 ⇒ ...
if_icmpne	Branch if int comparison value1 != value2 true	..., value1, value2 ⇒ ...
if_icmplt	Branch if int comparison value1 < value2 true	..., value1, value2 ⇒ ...
if_icmpgt	Branch if int comparison value1 > value2 true	..., value1, value2 ⇒ ...
if_icmple	Branch if int comparison value1 <= value2 true	..., value1, value2 ⇒ ...
if_icmpge	Branch if int comparison value1 >= value2 true	..., value1, value2 ⇒ ...
if_acmpeq	Branch if reference comparison value1 == value2 true	..., value1, value2 ⇒ ...
if_acmpne	Branch if reference comparison value1 != value2 true	..., value1, value2 ⇒ ...
goto	Branch always	<i>No change</i>
goto_w	Branch always (wide index)	<i>No change</i>
jsr	Jump subroutine	... ⇒ ..., address
jsr_w	Jump subroutine (wide index)	... ⇒ ..., address

The above instructions can be added to the code section of a method by sending the **CLEmitter** instance the following message:

```
public void addBranchInstruction(int opcode,
                               String label)
```

where **opcode** is the mnemonic of the instruction to be added, and **label** is the target instruction label.

Mnemonic	Operation	Operand Stack
ret	Return from subroutine	<i>No change</i>

The above instruction can be added to the code section of a method by sending the **CLEmitter** instance the following message:

```
public void addOneArgInstruction(int opcode,
                                int arg)
```

where **opcode** is the mnemonic of the instruction to be added, and **arg** is the index of the local variable containing the return address.

Mnemonic	Operation	Operand Stack
-----------------	------------------	----------------------

lookupswitch	Access jump table by key match and jump	..., <i>key</i> ⇒ ...
---------------------	---	-----------------------

The above instruction can be added to the code section of a method by sending the **CLEmitter** instance the following message:

```
public void
    addLOOKUPSWITCHInstruction(String defaultLabel,
                              int numPairs,
                              TreeMap<Integer, String>
                              matchLabelPairs)
```

where **defaultLabel** is the jump label for the default value, **numPairs** is the number of pairs in the match table, and the **matchLabelPairs** is the key match table.

Mnemonic	Operation	Operand Stack
tableswitch	Access jump table by index match and jump	..., <i>index</i> ⇒ ...

The above instruction can be added to the code section of a method by sending the **CLEmitter** instance the following message:

```
public void
    addTABLESWITCHInstruction(String defaultLabel,
                              int low,
                              int high,
                              ArrayList<String> labels)
```

where **defaultLabel** is the jump label for the default value, **low** is smallest value of index, **high** is the highest value of index, and **labels** is a list of jump labels for each index value from **low** to **high**, end values included.

D.4.10 Load Store Instructions

Mnemonic	Operation	Operand Stack
iload_n ; $n \in [0, 3]$	Load int from local variable at index n	... ⇒ ..., <i>value</i>
lload_n ; $n \in [0, 3]$	Load long from local variable at index n	... ⇒ ..., <i>value</i>
float_n ; $n \in [0, 3]$	Load float from local variable at index n	... ⇒ ..., <i>value</i>
dload_n ; $n \in [0, 3]$	Load double from local variable at index n	... ⇒ ..., <i>value</i>
aload_n ; $n \in [0, 3]$	Load reference from local variable at index n	... ⇒ ..., <i>objectref</i>
istore_n ; $n \in [0, 3]$	Store int into local variable at index n	..., <i>value</i> ⇒ ...
lstore_n ; $n \in [0, 3]$	Store long into local variable at index n	..., <i>value</i> ⇒ ...
fstore_n ; $n \in [0, 3]$	Store float into local variable at index n	..., <i>value</i> ⇒ ...

dstore <i>n</i> ; $n \in [0, 3]$	Store double into local variable at index <i>n</i>	..., <i>value</i> ⇒ ...
astore <i>n</i> ; $n \in [0, 3]$	Store reference into local variable at index <i>n</i>	..., <i>objectref</i> ⇒ ...
iconst <i>n</i> ; $n \in [0, 5]$	Push int constant <i>n</i>	... ⇒ ..., <i>value</i>
iconst <i>m1</i>	Push int constant -1	... ⇒ ..., <i>value</i>
lconst <i>n</i> ; $n \in [0, 1]$	Push long constant <i>n</i>	... ⇒ ..., <i>value</i>
fconst <i>n</i> ; $n \in [0, 2]$	Push float constant <i>n</i>	... ⇒ ..., <i>value</i>
dconst <i>n</i> ; $n \in [0, 1]$	Push double constant <i>n</i>	... ⇒ ..., <i>value</i>
aconst <i>null</i>	Push null	... ⇒ ..., <i>null</i>
wide ⁹	Modifies the behavior another instruction ¹⁰ by extending local variable index by additional bytes	<i>Same as modified instruction</i>

The above instructions can be added to the code section of a method by sending the **CLEmitter** instance the following message:

```
public void addNoArgInstruction(int opcode)
```

where **opcode** is the mnemonic of the instruction to be added.

Mnemonic	Operation	Operand Stack
iload	Load int from local variable at an index	... ⇒ ..., <i>value</i>
lload	Load long from local variable at an index	... ⇒ ..., <i>value</i>
fload	Load float from local variable at an index	... ⇒ ..., <i>value</i>
dload	Load double from local variable at an index	... ⇒ ..., <i>value</i>
aload	Load reference from local variable at an index	... ⇒ ..., <i>objectref</i>
istore	Store int into local variable at an index	..., <i>value</i> ⇒ ...
lstore	Store long into local variable at an index	..., <i>value</i> ⇒ ...
fstore	Store float into local variable at an index	..., <i>value</i> ⇒ ...
dstore	Store double into local variable at an index	..., <i>value</i> ⇒ ...
astore	Store reference into local variable at an index	..., <i>objectref</i> ⇒ ...

⁹ The **CLEmitter** interface implicitly adds this instruction where necessary.

¹⁰ Instructions that can be widened are **iload**, **fload**, **aload**, **lload**, **dload**, **istore**, **fstore**, **astore**, **lstore**, **dstore**, and **ret**.

The above instructions can be added to the code section of a method by sending the **CLEmitter** instance the following message:

```
public void addOneArgInstruction(int opcode,
                               int arg)
```

where **opcode** is the mnemonic of the instruction to be added, and **arg** is the index of the local variable.

Mnemonic	Operation	Operand Stack
iinc	Increment local variable by constant	<i>No change</i>

The above instruction can be added to the code section of a method by sending the **CLEmitter** instance the following message:

```
public void addIINCInstruction(int index,
                              int constVal)
```

where **index** is the local variable index, and **constVal** is the constant by which to increment.

Mnemonic	Operation	Operand Stack
bipush	Push byte	<i>... ⇒ ..., value</i>
sipush	Push short	<i>... ⇒ ..., value</i>

The above instructions can be added to the code section of a method by sending the **CLEmitter** instance the following message:

```
public void addOneArgInstruction(int opcode,
                               int arg)
```

where **opcode** is the mnemonic of the instruction to be added, and **arg** is **byte** or **short** value to push.

ldc¹¹ instruction can be added to the code section of a method using one of the following **CLEmitter** functions:

```
public void addLDCInstruction(int i)
public void addLDCInstruction(long l)
public void addLDCInstruction(float f)
public void addLDCInstruction(double d)
```

¹¹ The **CLEmitter** interface implicitly adds **ldc_w** and **ldc2_w** instructions where necessary.

```
public void addLDCInstruction(String s)
```

where the argument is the type of the item.

D.4.11 Stack Instructions

Mnemonic	Operation	Operand Stack
pop	Pop the top operand stack value	<i>..., value ⇒ ...</i>
pop2	Pop the top one or two operand stack values	<i>..., value2, value1 ⇒ ...</i> <i>..., value ⇒ ...</i>
dup	Duplicate the top operand stack value	<i>..., value ⇒ ..., value, value</i>
dup_x1	Duplicate the top operand stack value and insert two values down	<i>..., value2, value1 ⇒ ..., value1, value2, value1</i>
dup_x2	Duplicate the top operand stack value and insert two or three values down	<i>..., value3, value2, value1 ⇒ ..., value1, value3, value2, value1</i> <i>..., value2, value1 ⇒ ..., value1, value2, value1</i>
dup2	Duplicate the top one or two operand stack values	<i>..., value2, value1 ⇒ ..., value2, value1, value2, value1</i> <i>..., value ⇒ ..., value, value</i>
dup2_x1	Duplicate the top one or two operand stack values and insert two or three values down	<i>..., value3, value2, value1 ⇒ ..., value2, value1, value3, value2, value1</i> <i>..., value2, value1 ⇒ ..., value1, value2, value1</i>
dup2_x2	Duplicate the top one or two operand stack values and insert two, three, or four values down	<i>..., value4, value3, value2, value1 ⇒ ..., value2, value1, value4, value3, value2, value1</i> <i>..., value3, value2, value1 ⇒ ..., value1, value3, value2, value1</i> <i>..., value3, value2, value1 ⇒ ..., value2, value1, value3, value2, value1</i> <i>..., value2, value1 ⇒ ..., value1,</i>

		<i>value2, value1</i>
swap	Swap the top two operand stack values	<i>..., value2, value1 ⇒ ..., value1, value2</i>

The above instructions can be added to the code section of a method by sending the **CLEmitter** instance the following message:

```
public void addNoArgInstruction(int opcode)
```

where **opcode** is the mnemonic of the instruction to be added.

D.4.12 Other Instructions

Mnemonic	Operation	Operand Stack
nop	Do nothing	<i>No change</i>
athrow	Throw exception or error	<i>..., objectref ⇒ ..., objectref</i>
monitorenter	Enter monitor for object	<i>..., objectref ⇒ ...</i>
monitorexit	Exit monitor for object	<i>..., objectref ⇒ ...</i>

The above instructions can be added to the code section of a method by sending the **CLEmitter** instance the following message:

```
public void addNoArgInstruction(int opcode)
```

where **opcode** is the mnemonic of the instruction to be added.

Further Reading

JVM Specification (Lindholm and Yellin, 1999). See Chapter 4 of the specification for a detailed description of the class file format. See chapter 6 (updated) for detailed information about the format of each JVM instruction and the operation it performs. The chapter is available online at <http://java.sun.com/docs/books/vmspec/2nd-edition/Java5-Instructions2.pdf>. See Chapter 7 for hints on compiling various Java language constructs to JVM byte code.

