D. The JVM, Class Files and the CLEmitter

D.1 The Java Virtual Machine (JVM)

In the first instance, our compiler's target is the Java Virtual Machine (JVM). The JVM is a *virtual* byte-code machine. We say it is virtual because there is no real JVM computer chip, per say, but the JVM is an abstract architecture which can have any number of implementations. For example, Oracle[1] has implemented a Java Runtime Environment (JRE) which interprets JVM programs, but uses Hotspot technology for further compiling code that is executed repeatedly to native machine code. We say it is a *byte-code* machine, because the programs it executes are sequences of bytes that represent the instructions and the operands.

Although virtual, the JVM has a definite architecture. It has an instruction set and it has an internal organization.

The JVM starts up by creating an initial class, which is specified in an implementation-dependent manner, using the bootstrap class loader. The JVM then links the initial class, initializes it, and invokes its **`public`** class method **`void main(String[] args)`**. The invocation of this method drives all further execution. Execution of JVM instructions constituting the **`main()`** method may cause linking (and consequently creation) of additional classes and interfaces, as well as invocation of additional methods.

A JVM instruction consists of a one-byte opcode (operation code) specifying the operation to be performed, followed by zero or more operands supplying the arguments or data that are used by the operation.

The inner loop of the JVM interpreter is effectively:

```
do {
    fetch an opcode;
    if (operands) fetch operands;
    execute the action for the opcode;
} while (there is more to do);
```

The JVM defines various run-time data areas that are used during execution of a program. Some of these data areas are created on the JVM start-up and are destroyed only when the JVM exits. Other data areas are per thread[2]. Per-thread data areas are created when a thread is created and destroyed when the thread exits.

---

[1] Oracle, Inc. http://www.oracle.com/technetwork/java/javase/downloads/index.html.
[2] *j--* doesn't support implementation of multi-threaded programs.

D.1.1 The pc Register

The JVM can support many threads of execution at once, and each thread has its own **`pc`** (program counter) register. At any point, each JVM thread is executing the code of a single method, the current method for that thread. If the method is not **`native`**, the **`pc`** register contains the address of the JVM instruction currently being executed.

D.1.2 JVM Stacks and Stack Frames

The JVM is not a register machine, but a *stack machine*. Each JVM thread has a run-time data stack, created at the same time as the thread. The JVM stack is analogous to the stack of a conventional language such as C; it holds local variables and partial results, and plays a role in method invocation and return. There are instructions for loading data values onto the stack, for performing operations on the value(s) that are on top of the stack, and there are instructions for storing the results of computations back in variables.

For example, consider the following simple expression.

```
34 + 6 * 11
```

If the compiler does no constant folding, it might produce the following JVM code for performing the calculation.

```
ldc 34
ldc 6
ldc 11
imul
iadd
```

Executing this sequence of instructions takes a run-time stack through the sequence of states illustrated in Figure 2.2.

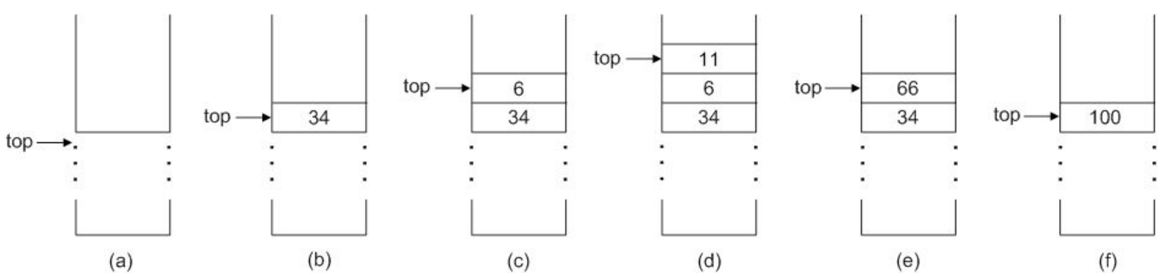

Figure D.1 The Stack States for Computing `34 + 6 * 11`

In (a) the run-time stack is in some initial state, where top points to its top value. Executing the first **`ldc`** (load constant) instruction causes the JVM to push the value 34 onto the stack, leaving it in state (b). The second **`ldc`**, pushes 6 onto the stack, leaving it in state (c). The third **`ldc`** pushes 11 onto the stack, leaving it in state (d). Then, executing the **imul** (integer multiplication) instruction causes the JVM to pop the top two values (11 and 6) off from the stack, multiply them together, and to push the resultant 66 back onto the stack, leaving it in state (e). Finally, the **iadd** (integer addition) instruction pops the top two values (66 and 34) off the stack, adds them together, and pushes the resultant 100 back onto the stack, leaving it in the state (f).

As we saw in Chapter 1, the stack is organized into *stack frames*. Each time a method is invoked, a new stack frame for the method invocation is pushed onto the stack. All actual arguments that correspond to the method's formal parameters, and all local variables in the method's body are allocated space in this stack frame. Also, all of the method's computations, like that illustrated in Figure D.1, are carried out in the same stack frame. Computations take place in the space above the arguments and local variables.

For example, consider the following instance method, **add()**.

```
int add(int x, int y)
{
    int z;
    z = x + y;
    return z;
}
```

Now, say **add()** is a method defined in a class named **Foo**, and further assume that **f** is a variable of type **Foo**. Consider the message expression,

```
f.add(2, 3);
```

When **add()** is invoked, a stack frame like that illustrated in Figure D.2 is pushed onto the run-time stack.

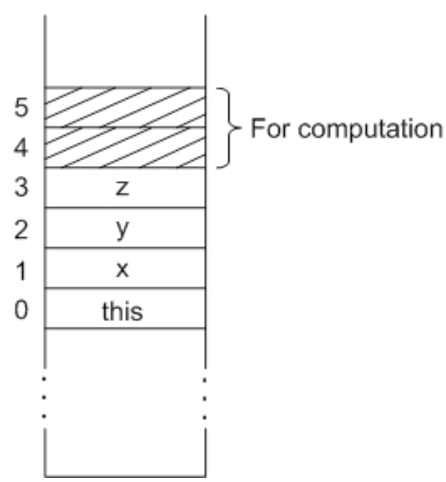

Figure D.2 The Stack Frame for an Invocation of `add()`

Because **`add()`** is an instance method, the object itself, i.e. **`this`**, must be passed as an implicit argument in the method invocation; so **`this`** occupies the first location in the stack frame at offset 0. Then the actual parameter values 2 and 3, for formal parameters **`x`** and **`y`**, occupy the next two locations at offsets 1 and 2 respectively. The local variable **`z`** is allocated space for its value at offset 3. Finally, two locations are allocated above the parameters and local variable for the computation.

Here is a symbolic version[3] of the code produced for **`add()`** by our *j*-- compiler.

```
int add(int, int);
  Code:
   Stack=2, Locals=4, Args_size=3
   0:   iload_1
   1:   iload_2
   2:   iadd
   3:   istore_3
   4:   iload_3
   5:   ireturn
```

Here's how the JVM executes this code.

- The **`iload_1`** instruction loads the integer value at offset 1 (for **`x`**) onto the stack at frame offset 4.
- The next **`iload_2`** instruction loads the integer value at offset 2 (for **`y`**) onto the stack at frame offset 5.
- The **`iadd`** instruction pops the two integers off the top of the stack (from frame offsets 4 and 5), adds them using integer addition, and then pushes the result (**`x + y`**) back onto the stack at frame offset 4.
- The **`istore_3`** instruction pops the top value (at frame offset 4) off the stack and stores it at offset 3 (for **`z`**).
- The **`iload_3`** instruction loads the value at frame offset 3 (for **`z`**) onto the stack at frame offset 4.
- Finally, the **`ireturn`** instruction pops the top integer value from the stack (at frame location 4), pops the stack frame from the stack, and returns control to the invoking method, pushing the returned value onto the invoking method's frame.

Notice that the instruction set takes advantage of common offsets within the stack frame. For example, the **`iload_1`** instruction is really shorthand for

```
iload  1
```

[3] Produced using **`javap -v Foo`**.

The `iload_1` instruction occupies just one byte for the opcode; the opcode for `iload_1` is 27. But the other version requires two bytes: one for the opcode (21) and one byte for the operand's offset (1). The JVM is trading opcode space -- a byte may only represent up to 256 different operations -- to save code space.

A frame may be extended to contain additional implementation-specific data such as the information required to support a run-time debugger.

D.1.3. The Heap

Objects are represented on the stack as pointers into the *heap*, which is shared among all JVM threads. The heap is the run-time data area from which memory for all class instances and arrays is allocated. It is created during the JVM start-up. Heap storage for objects is reclaimed by an automatic storage management system called the *garbage collector*.

D.1.4. The Method Area

The JVM has a method area that is shared among all JVM threads. It is created during the JVM start-up. It stores per-class structures such as the run-time constant pool, field and method data, and the code for methods and constructors, including the special methods used in class and instance initialization and interface type initialization.

D.1.5. The Runtime Constant Pool

The *run-time constant pool* is a per-class or per-interface run-time representation of the `constant_pool` table in a class file. It contains several kinds of constants, ranging from numeric literals known at compile time to method and field references that must be resolved at runtime. It is constructed when the class or interface is created by the JVM.

D.1.6. Abrupt Method Invocation Completion

A method invocation completes abruptly if execution of a JVM instruction within the method causes the JVM to throw an exception, and that exception is not handled within the method. Execution of an `ATHROW` instruction also causes an exception to be explicitly thrown and, if the exception is not caught by the current method, results in abrupt method invocation completion. A method invocation that completes abruptly never returns a value to its invoker.

D.2 The Class File

D.2.1 Structure of a Class File

The byte-code that *j--* generates from a source program is stored in a binary file with a .class extension. We refer to such files as *class files*. Each class file contains the definition of a single class or interface. A class file consists of a stream of 8-bit bytes. All 16-bit, 32-bit, and 64-bit quantities are constructed by reading in two, four, and eight consecutive 8-bit bytes, respectively. Multi-byte data items are always stored in big-endian order, where the high bytes come first.

A class file consists of a single ClassFile structure; in the *C* language it would be:

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

The types `u1`, `u2`, and `u4` represent an unsigned one-, two-, or four-byte quantity, respectively. The items of the ClassFile structure are described below.

`magic`	A magic number (`0xCAFEBABE`) identifying the class file format.
`minor_version, major_version`	Together, a major and minor version number determine the version of the class file format. A JVM implementation can support a class file format of version *v* if and only if *v* lies in some contiguous range of versions. Only Oracle specifies the range of versions a JVM implementation may support.
`constant_pool_count`	Number of entries in the `constant_pool` table plus one.
`constant_pool[]`	A table of structures representing various

	string constants, class and interface names, field names, and other constants that are referred to within the **ClassFile** structure and its substructures.
access_flags	Mask of flags used to denote access permissions to and properties of this class or interface.
this_class	Must be a valid index into the **constant_pool** table. The entry at that index must be a structure representing the class or interface defined by this class file.
super_class	Must be a valid index into the **constant_pool** table. The entry at that index must be the structure representing the direct superclass of the class or interface defined by this class file.
interfaces_count	The number of direct super interfaces of the class or interface defined by this class file.
interfaces[]	Each value in the table must be a valid index into the **constant_pool** table. The entry at each index must be a structure representing an interface that is a direct superinterface of the class or interface defined by this class file.
fields_count	Number of entries in the **fields** table.
fields[]	Each value in the table must be a **field_info** structure giving complete description of a field in the class or interface defined by this class file.
methods_count	Number of entries in the **methods** table.
methods[]	Each value in the table must be a **method_info** structure giving complete description of a method in the class or interface defined by this class file.
attributes_count	Number of entries in the attributes table.
attributes[]	Must be a table of class attributes.

The internals for all of these are fully described in (Lindholm and Yellin, 1999).

One may certainly create class files by directly working with a binary output stream. However, this approach is rather arcane, and involves a tremendous amount of housekeeping; one has to maintain a representation for the **constant_pool** table, the program counter `pc`, compute branch offsets, compute stack depths, perform various bitwise operations, and do much more.

It would be much easier if there were a high level interface that would abstract out the gory details of the class file structure. The `CLEmitter` does exactly this.

D.2.2 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 ('/')[4]. 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 grammar[5]:

FieldDescriptor ::= FieldType

ComponentType ::= FieldType

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`

[4] This is different from the familiar syntax of fully qualified names, where the identifiers are separated by periods ('.').
[5] This is the so-called EBNF (Extended Backus Naur Form) notation for describing the syntax of languages.

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
`int i;`	`I`
`Object o;`	`Ljava/lang/Object;`
`double[][][] d;`	`[[[D`
`Long[][] l;`	`[[Ljava/lang/Long;`

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

D.3 The CLEmitter

D.3.1 CLEmitter Operation

The *j--* compiler's purpose is to produce a class file. Given the complexity of class files we supply a tool called the **`CLEmitter`** to ease the generation of code and the creation of class files.

The `CLEmitter`[6] has a relatively small set of methods that support

- the creation of a class or an interface;
- the addition of fields and methods to the class;
- the addition of instructions, exception handlers, and code attributes to methods;
- the addition of inner classes;
- optional field, method, and class attributes;
- checking for errors; and
- the writing of the class file to the file system.

While it is much simpler to work with an interface like `CLEmitter`, one still has to be aware of certain aspects of the target machine, such as the instruction set.

Figure D.3 outlines the necessary steps for creating an in-memory representation of a class file using the `CLEmitter` interface, and then writing that class file to the file system.

[6] This is a class in the `jminusminus` package under `$j/j--/src` folder. The classes that `CLEmitter` depends on are also in that package and have a `CL` prefix.

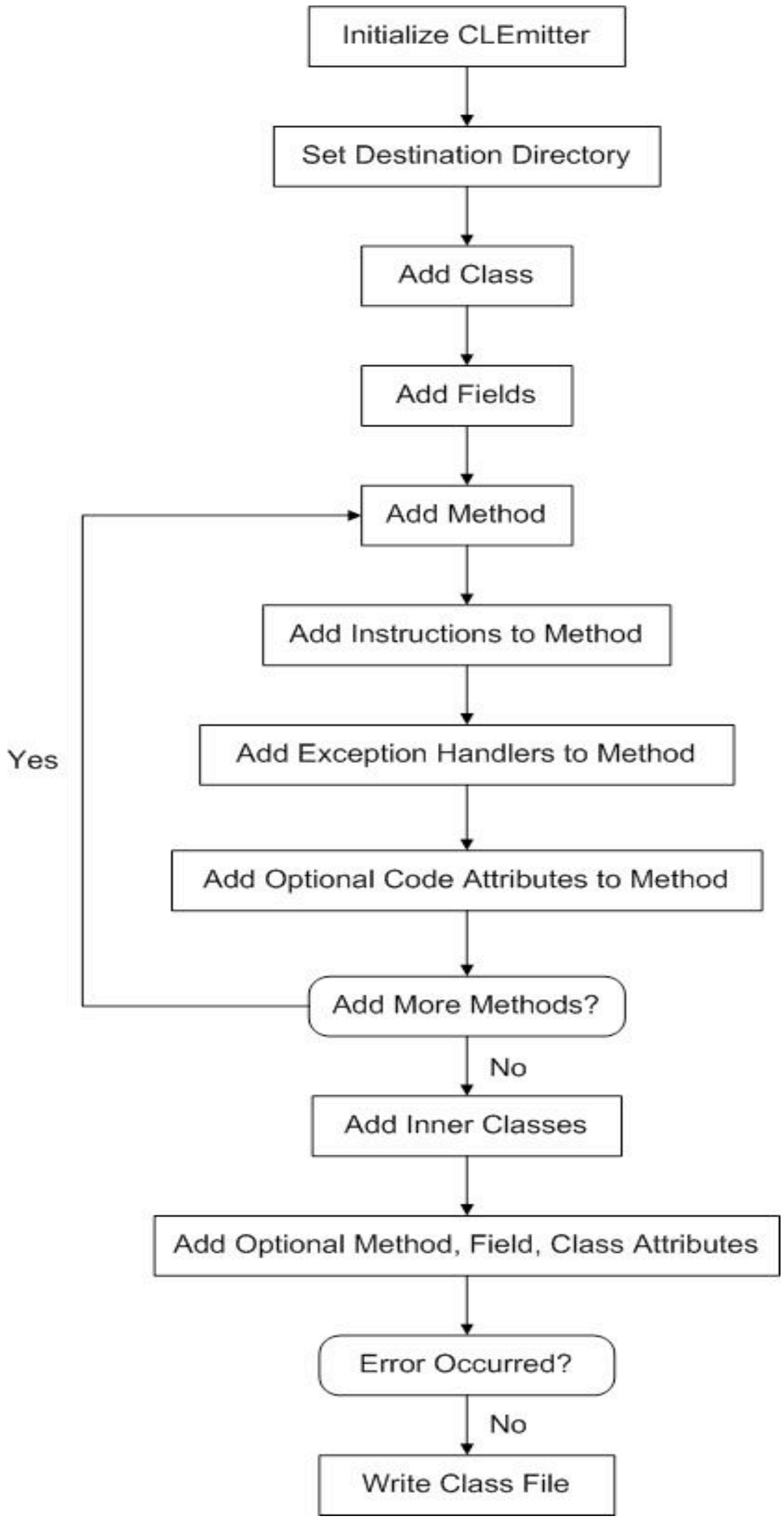

Figure D.3: A Recipe for Creating a Class File

D.3.2 CLEmitter Interface

The CLEmitter interface that supports: 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.3.2.1 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.

To instantiate a **`CLEmitter`**, one simply invokes its constructor.

```
CLEmitter output = new CLEmitter(true);
```

Change the argument to the constructor to `false` if only an in-memory representation of the class is needed, in which case the class will not be written to the file system.

One then goes about adding classes, method, fields and instructions. There are methods for adding all of these.

D.3.2.2 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.3.2.3 Add Class

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

```
public void addClass(ArrayList<String> accessFlags,
                     String thisClass,
                     String superClass,
                     ArrayList<String> superInterfaces,
                     boolean isSynthetic)
```

where `accessFlags`[7] 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 ("interactive" and "abstract") 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.3.2.4 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.3.2.5 Add Field

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)
```

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 `isSythetic` specifies whether the field appears in source code.

[7] Note that the `CLEmitter` expects the access and property flags as `String`s and internally translates them to a mask of flags. For example, "public" is translated to `ACC_PUBLIC (0x0001)`.

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: [8]

```
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**.

D.3.2.6 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)
```

[8] 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.

where **`accessFlags`** is a list of method access and property flags, **`name`** is the name[9] of the method, **`descriptor`** is the method descriptor, **`exceptions`** is a list of exceptions in internal form that this method throws, and **`isSythetic`** specifies whether the method appears in source code.

For example, one may add a method using,

```
accessFlags.add("public");
output.addMethod(accessFlags, "factorial", "(I)I", exceptions,
                 isSynthetic);
```

where **`accessFlags`** is a list of method access and property flags, **`"factorial"`** is the name[10] of the method, **`"(I)I"`** 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 or was synthesized by the compiler.

A comment on the method descriptor is warranted. The *method descriptor* describes the method's signature in a format internal to the JVM. The **`I`** is the internal type descriptor for the primitive type **`int`**, so the

```
(I)I
```

specifies that the method takes one integer argument, and returns a value having integer type.

D.3.2.7 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`**.

---

[9] Instance initializers must have the name **`<init>`** and static initializers must have the name **`<clinit>`**.

[10] Instance constructors must have the name **`<init>`** and static constructors must have the name **`<clinit>`**.

`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.3.2.8 Add Optional Method, Field, Class, Code Attributes

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.3.2.9 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.3.2.10 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:

```
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 mnemonic[11] 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 represent[12] the operand stack as follows:

$$..., value1, value2 \Rightarrow ..., result$$

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
`new`	Create new object	... ⇒ ..., *objectref*
`instanceof`	Determine if object is of given type	..., *objectref* ⇒ ..., *result*
`checkcast`	Check whether object is of given type	..., *objectref* ⇒ ..., *objectref*

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

[11] These mnemonics (also called opcodes) are defined in **`jminusminus.CLConstants.`**
[12] This is the representation used in The Java Virtual Machine specification, 2nd edition.

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

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	..., *objectref* ⇒ ..., *value*
`putfield`	Set field in object	..., *objectref, value* ⇒ ...
`getstatic`	Get static field from class	... ⇒ ..., *value*
`putstatic`	Set static field in class	..., *value* ⇒ ...

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	..., *objectref, [arg1, [arg2 ...]]* ⇒ ...
`invokeinterface`	Invoke interface method	..., *objectref, [arg1, [arg2 ...]]* ⇒ ...
`invokespecial`	Invoke instance method; special handling for superclass, private, and instance initialization method invocations	..., *objectref, [arg1, [arg2 ...]]* ⇒ ...
`invokestatic`	Invoke a class (static)	..., *[arg1, [arg2 ...]]* ⇒ ...

	method	
invokedynamic	Invoke instance method; dispatch based on class	*..., objectref, [arg1, [arg2 ...]] ⇒ ...*

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	*..., value ⇒ [empty]*
lreturn	Return **long** from method	*..., value ⇒ [empty]*
freturn	Return **float** from method	*..., value ⇒ [empty]*
dreturn	Return **double** from method	*..., value ⇒ [empty]*
areturn	Return **reference** from method	*..., objectref ⇒ [empty]*
return	Return **void** from method	*... ⇒ [empty]*

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	*..., count ⇒ ..., arrayref*
anewarray	Create new array of **reference** type	*..., count ⇒ ..., arrayref*

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,
                                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	*..., count1, [count2, ...] ⇒ ..., arrayref*

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	*..., arrayref, index ⇒ ..., value*
`caload`	Load **`char`** from array	*..., arrayref, index ⇒ ..., value*
`saload`	Load **`short`** from array	*..., arrayref, index ⇒ ..., value*
`iaload`	Load **`int`** from array	*..., arrayref, index ⇒ ..., value*
`laload`	Load **`long`** from array	*..., arrayref, index ⇒ ..., value*
`faload`	Load **`float`** from array	*..., arrayref, index ⇒ ..., value*
`daload`	Load **`double`** from array	*..., arrayref, index ⇒ ..., value*
`aaload`	Load from **`reference`** array	*..., arrayref, index ⇒ ..., value*
`bastore`	Store into **`byte`** or **`boolean`** array	*..., arrayref, index, value ⇒ ...*
`castore`	Store into **`char`** array	*..., arrayref, index, value ⇒ ...*
`sastore`	Store into **`short`** array	*..., arrayref, index, value ⇒ ...*
`iastore`	Store into **`int`** array	*..., arrayref, index, value ⇒ ...*
`lastore`	Store into **`long`** array	*..., arrayref, index, value ⇒ ...*
`fastore`	Store into **`float`** array	*..., arrayref, index, value ⇒ ...*
`dastore`	Store into **`double`** array	*..., arrayref, index, value ⇒ ...*
`aastore`	Store into **`reference`** array	*..., arrayref, index, value ⇒ ...*
`arraylength`	Get length of array	*..., arrayref ⇒ ..., length*

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.4.5 Arithmetic Instructions

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

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`	..., *value1*, *value2* ⇒ ..., *result*
`ishr`	Arithmetic shift right `int`	..., *value1*, *value2* ⇒ ..., *result*
`iushr`	Logical shift right `int`	..., *value1*, *value2* ⇒ ..., *result*
`lshl`	Shift left `long`	..., *value1*, *value2* ⇒ ..., *result*
`lshr`	Arithmetic shift right `long`	..., *value1*, *value2* ⇒ ..., *result*
`lushr`	Logical shift right `long`	..., *value1*, *value2* ⇒ ..., *result*
`ior`	Boolean OR `int`	..., *value1*, *value2* ⇒ ..., *result*
`lor`	Boolean OR `long`	..., *value1*, *value2* ⇒ ..., *result*
`iand`	Boolean AND `int`	..., *value1*, *value2* ⇒ ..., *result*
`land`	Boolean AND `long`	..., *value1*, *value2* ⇒ ..., *result*
`ixor`	Boolean XOR `int`	..., *value1*, *value2* ⇒ ..., *result*
`lxor`	Boolean XOR `long`	..., *value1*, *value2* ⇒ ..., *result*

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	..., *value1*, *value2* ⇒ ..., *result*
`dcmpl`	Compare `double`; result is -1 if at least one value is NaN	..., *value1*, *value2* ⇒ ..., *result*
`fcmpg`	Compare `float`; result is 1 if at least one value is NaN	..., *value1*, *value2* ⇒ ..., *result*
`fcmpl`	Compare `float`; result is -1 if at least one value is NaN	..., *value1*, *value2* ⇒ ..., *result*
`lcmp`	Compare `long`	..., *value1*, *value2* ⇒ ..., *result*

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
`i2b`	Convert `int` to `byte`	..., *value* ⇒ ..., *result*
`i2c`	Convert `int` to `char`	..., *value* ⇒ ..., *result*
`i2s`	Convert `int` to `short`	..., *value* ⇒ ..., *result*
`i2l`	Convert `int` to `long`	..., *value* ⇒ ..., *result*
`i2f`	Convert `int` to `float`	..., *value* ⇒ ..., *result*
`i2d`	Convert `int` to `double`	..., *value* ⇒ ..., *result*
`l2f`	Convert `long` to `float`	..., *value* ⇒ ..., *result*
`l2d`	Convert `long` to `double`	..., *value* ⇒ ..., *result*
`l2i`	Convert `long` to `int`	..., *value* ⇒ ..., *result*
`f2d`	Convert `float` to `double`	..., *value* ⇒ ..., *result*
`f2i`	Convert `float` to `int`	..., *value* ⇒ ..., *result*
`f2l`	Convert `float` to `long`	..., *value* ⇒ ..., *result*
`d2i`	Convert `double` to `int`	..., *value* ⇒ ..., *result*
`d2l`	Convert `double` to `long`	..., *value* ⇒ ..., *result*
`d2f`	Convert `double` to `float`	..., *value* ⇒ ..., *result*

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
`ifeq`	Branch if `int` comparison value == 0 true	..., *value* ⇒ ...
`ifne`	Branch if `int` comparison value != 0 true	..., *value* ⇒ ...
`iflt`	Branch if `int` comparison value < 0 true	..., *value* ⇒ ...
`ifgt`	Branch if `int` comparison value > 0 true	..., *value* ⇒ ...
`ifle`	Branch if `int` comparison value <= 0 true	..., *value* ⇒ ...
`ifge`	Branch if `int` comparison value >= 0 true	..., *value* ⇒ ...
`ifnull`	Branch if `reference` is null	..., *value* ⇒ ...
`ifnonnull`	Branch if `reference` is not null	..., *value* ⇒ ...
`if_icmpeq`	Branch if `int` comparison value1 == value2 true	..., *value1, value2* ⇒ ...
`if_icmpne`	Branch if `int` comparison value1 != value2	..., *value1, value2* ⇒ ...

	true	
`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	*No change*
`goto_w`	Branch always (wide index)	*No change*
`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	*No change*

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	..., *key* ⇒ ...

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	*..., index ⇒ ...*

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*	*... ⇒ ..., value*
`lload_n`; $n \in [0, 3]$	Load `long` from local variable at index *n*	*... ⇒ ..., value*
`fload_n`; $n \in [0, 3]$	Load `float` from local variable at index *n*	*... ⇒ ..., value*
`dload_n`; $n \in [0, 3]$	Load `double` from local variable at index *n*	*... ⇒ ..., value*
`aload_n`; $n \in [0, 3]$	Load reference from local variable at index *n*	*... ⇒ ..., objectref*
`istore_n`; $n \in [0, 3]$	Store `int` into local variable at index *n*	*..., value ⇒ ...*
`lstore_n`; $n \in [0, 3]$	Store `long` into local variable at index *n*	*..., value ⇒ ...*
`fstore_n`; $n \in [0, 3]$	Store `float` into local variable at index *n*	*..., value ⇒ ...*
`dstore_n`; $n \in [0, 3]$	Store `double` into local variable at index *n*	*..., value ⇒ ...*
`astore_n`; $n \in [0, 3]$	Store reference into local variable at index *n*	*..., objectref ⇒ ...*
`iconst_n`; $n \in [0, 5]$	Push `int` constant *n*	*... ⇒ ..., value*
`iconst_m1`	Push `int` constant -1	*... ⇒ ..., value*
`lconst_n`; $n \in [0, 1]$	Push `long` constant *n*	*... ⇒ ..., value*

`fconst_n`; $n \in [0, 2]$	Push `float` constant *n*	... ⇒ ..., *value*
`dconst_n`; $n \in [0, 1]$	Push `double` constant *n*	... ⇒ ..., *value*
`aconst_null`	Push null	... ⇒ ..., *null*
`wide`[13]	Modifies the behavior another instruction[14] by extending local variable index by additional bytes	*Same as modified instruction*

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	... ⇒ ..., *value*
`lload`	Load `long` from local variable at an index	... ⇒ ..., *value*
`fload`	Load `float` from local variable at an index	... ⇒ ..., *value*
`dload`	Load `double` from local variable at an index	... ⇒ ..., *value*
`aload`	Load `reference` from local variable at an index	... ⇒ ..., *objectref*
`istore`	Store `int` into local variable at an index	..., *value* ⇒ ...
`lstore`	Store `long` into local variable at an index	..., *value* ⇒ ...
`fstore`	Store `float` into local variable at an index	..., *value* ⇒ ...
`dstore`	Store `double` into local variable at an index	..., *value* ⇒ ...
`astore`	Store `reference` into local variable at an index	..., *objectref* ⇒ ...

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.

[13] The `CLEmitter` interface implicitly adds this instruction where necessary.
[14] Instructions that can be widened are `iinc, iload, fload, aload, lload, dload, istore, fstore, astore, lstore, dstore`, and `ret`.

Mnemonic	Operation	Operand Stack
`iinc`	Increment local variable by constant	*No change*

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`**	... ⇒ ..., *value*
`sipush`	Push **`short`**	... ⇒ ..., *value*

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.

`ldc`[15] 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)
public void addLDCInstruction(String s)
```

where the argument is the type of the item.

D.4.11 Stack Instructions

Mnemonic	Operation	Operand Stack

[15] The **`CLEmitter`** interface implicitly adds **`ldc_w`** and **`ldc2_w`** instructions where necessary.

pop	Pop the top operand stack value	*..., value ⇒ ...*
pop2	Pop the top one or two operand stack values	*..., value2, value1 ⇒ ...* *..., value ⇒ ...*
dup	Duplicate the top operand stack value	*..., value ⇒ ..., value, value*
dup_x1	Duplicate the top operand stack value and insert two values down	*..., value2, value1 ⇒ ..., value1, value2, value1*
dup_x2	Duplicate the top operand stack value and insert two or three values down	*..., value3, value2, value1 ⇒ ..., value1, value3, value2, value1* *..., value2, value1 ⇒ ..., value1, value2, value1*
dup2	Duplicate the top one or two operand stack values	*..., value2, value1 ⇒ ..., value2, value1, value2, value1* *..., value ⇒ ..., value, value*
dup2_x1	Duplicate the top one or two operand stack values and insert two or three values down	*..., value3, value2, value1 ⇒ ..., value2, value1, value3, value2, value1* *..., value2, value1 ⇒ ..., value1, value2, value1*
dup2_x2	Duplicate the top one or two operand stack values and insert two, three, or four values down	*..., value4, value3, value2, value1 => ..., value2, value1, value4, value3, value2, value1* *..., value3, value2, value1 ⇒ ..., value1, value3, value2, value1* *..., value3, value2, value1 ⇒ ..., value2, value1, value3, value2, value1* *..., value2, value1 ⇒ ..., value1, value2, value1*
swap	Swap the top two operand stack values	*..., value2, value1 ⇒ ..., value1, value2*

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	*No change*
`athrow`	Throw exception or error	*..., objectref ⇒ ..., objectref*
`monitorenter`	Enter monitor for object	*..., objectref ⇒ ...*
`monitorexit`	Exit monitor for object	*..., objectref ⇒ ...*

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. See Chapter 7 in that text for hints on compiling various Java language constructs to JVM byte code.