**TABLE 3.1:** *Mnemonic type letters*

| Letter | Type |
|---|---|
| a | `reference` |
| b | `byte` or `boolean` |
| c | `char` |
| d | `double` |
| f | `float` |
| i | `int` |
| l | `long` |
| s | `short` |

letter of the mnemonic often tells you what type the mnemonic operates on. For example, `iconst_1` loads an `int` 1 onto the stack, while `lconst_1` loads a `long` 1 onto the stack. The letter `i` indicates an `int`, and the letter `l` indicates a `long`. Table 3.1 summarizes the mnemonic naming conventions.

Do not be misled by the naming convention into believing in mnemonics that do not exist. Each mnemonic corresponds to a number between 0 and 255 in the `class` file. This number is called an *opcode*. Just because there's an opcode for the mnemonic `iand` to compute the bitwise `and` of two `int`s, it doesn't mean that there's an opcode for the mnemonic `dand` to compute the bitwise `and` of two `double`s. The complete list of mnemonics can be found in appendix A.

## 3.4 Testing Code Examples

Code examples appear throughout this chapter. Unfortunately, we haven't really discussed how to do output, so it will be a little tricky to actually try out any of these code examples. Output involves some mucking about with method calls and sometimes with creating new objects and other such complicated things that are better left for a later chapter.

The good news is that you can use Java to assemble a test harness for these code samples. You can wrap the test code up in a small, easy-to-write class, then use Java to perform the input and output. For example, here is some code to answer the question, "What do you get if you multiply 6 by 9?"

```
bipush 6        ; Push 6
bipush 9        ; Push 9
imul            ; Result is 54
```

### 4.4.2 Inheriting Fields

When one class has another class as a superclass, it inherits all of the nonstatic fields of that class. Consider an extension to the `Greeting` class to handle greetings in Russian:

```
class RussianGreeting extends Greeting
{
    String intro = "Zdravstvuite";
}
```

An instance of `RussianGreeting` has two fields, both named `intro`. An instance of `RussianGreeting` looks like Figure 4.4. The full names of the fields are different. In Java, the English-language `intro` is *hidden* behind the Russian-language version. In Oolong, they are both equally accessible:

```
.method static internationalGreetings(LRussianGreeting;)V
aload_0                ; There is a RussianGreeting in register 0
                       ; Push "Zdravstvuite":
getfield RussianGreeting/intro Ljava/lang/String;
aload_0                ; Reload the same object
                       ; Push "Hello":
getfield Greeting/intro Ljava/lang/String;
;; Rest of the method omitted
.end method
```

At the end of the code shown, there will be two objects on the stack. The bottom of the stack will contain a `reference` to the `String` “`Zdravstvuite`”, and above it will be a `reference` to the `String` “`Hello`”.

### 4.4.3 Changing Field Values

To get the value of a field, use `getfield`, which takes an object on the stack and leaves in its place the value of the field. The counterpart of `getfield` for changing

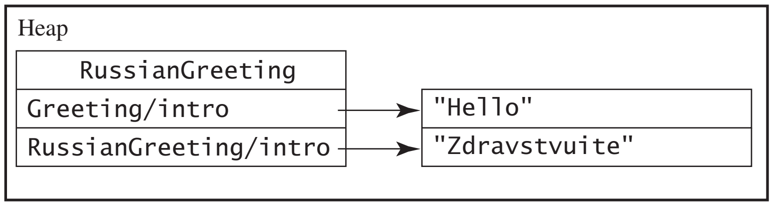


**FIGURE 4.4:** *An instance of* `RussianGreeting`

```
.field private elements [Ljava/lang/Object;
.field current I

.method public getNext()Ljava/lang/Object;
;; Return elements[current], and increment current
.end method

.method public anyMore()Ljava/lang/Object;
;; Return false if current < the length of elements
.end method
```

The implementor of `Business` can change the implementation of `inventory` without altering any of the classes that use it, since the implementation of `Enumerator` was kept separate from the interface definition.

Invoking methods through interfaces is slightly different from invoking methods using `invokevirtual` or `invokespecial`. A third instruction, `invokeinterface`, is required. To call `anyMore`, use code like this:

```
; Assume there's a Business in local variable 0
aload_0                                        ; Get the
                                               ; Business
invokevirtual Business/inventory ()LEnumerator; ; Get its
                                               ; inventory
invokeinterface Enumerator/anyMore ()Z 1       ; Call anyMore
```

The last argument to the `invokeinterface` instruction is the number of stack words used as parameters to the method call, including the receiver itself.[2] In this example, the only parameter is the `Enumerator` object itself, so the value is 1. For the interface:

```
.interface Searchable
.method find(Ljava/lang/String;)Ljava/lang/Object;
.end method
```

A call to `find` looks like this:

```
; Assume there's a Searchable object in register 1
aload_1              ; Get the object
ldc "potato chips"   ; We want an element named "potato chips"
invokeinterface Searchable/find
                (Ljava/lang/String;)Ljava/lang/Object; 2
```

[2] This number should not be necessary, since it can be derived from the method descriptor. However, it is included in the Oolong language because it is part of the underlying JVM bytecodes.

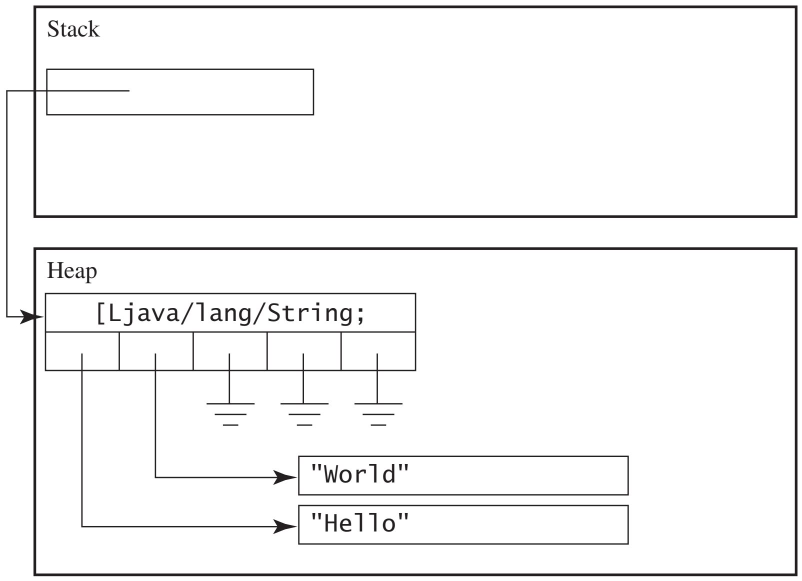


**FIGURE 4.9:** *After setting array slots 0 and 1*

```
dup                         ; Dup the array reference
ldc "Hello"                 ; Store hello
iconst_0                    ; Into slot 0
aastore

; The array reference is still on the stack
dup
ldc "World"                 ; Store hello
iconst_1                    ; Into slot 1
aastore
```

To get elements out of the array, you use `aaload`. To get the `reference` to `"World"` on the stack, use

```
iconst_1            ; Push int 1
aaload              ; Load array slot 1
```

Now the memory picture looks like the diagram in Figure 4.10. The top of the stack has been replaced with a `reference` to the `World` string.

You can think of an array as being a little like an object whose fields have numbers instead of names. Whenever you store into an array, you must meet the

**TABLE 6.1:** *Constant tags*

| Tag | Type | Format | Interpretation |
|---|---|---|---|
| 1 | UTF8 | 2+*n* bytes | The first two bytes are an unsigned integer *n*; the remaining *n* bytes are the text of the constant. |
| 2 | *not defined* | | |
| 3 | Integer | 4 bytes | Signed integer |
| 4 | Float | 4 bytes | IEEE 754 floating-point number |
| 5 | Long | 8 bytes | Long signed integer |
| 6 | Double | 8 bytes | IEEE 754 double-precision number |
| 7 | Class | 2 bytes | Reference to a UTF8 constant that is the name of a class |
| 8 | String | 2 bytes | Reference to a UTF8 constant that is the value of the String |
| 9 | Fieldref | 4 bytes | The first two bytes are a reference to a Class; the second two point to a NameAndType. |
| 10 | Methodref | 4 bytes | Same as Fieldref |
| 11 | InterfaceMethodref | 4 bytes | Same as Fieldref |
| 12 | NameAndType | 4 bytes | The first two bytes point to a UTF8 that is the name of the field or method; the second two point to a UTF8, which is its descriptor. |

Most sections begin with a count, which is a two-byte unsigned integer, followed by that many instances of some pattern of bytes. For example, following the major version number is the count of the number of constants. Each constant begins with a tag describing what sort of constant it is, which in turn tells how many bytes make up the constant. The set of constant tags is defined by the virtual machine specification. If any constant tag is invalid, or if the file ends before the correct number of constants are found, then the file is rejected. The valid constant tags are given in Table 6.1.

Similar rules apply to the other sections. If the file ends before all of the parts are found, or if there are extra bytes at the end, then the file is rejected. For more about the details of the inner workings of the class file, see chapter 9.

## 6.3 Are All Constant References Correct?

After asking whether or not the file looks like a properly formatted class file, the verification algorithm knows where the constant pool is to be found and how

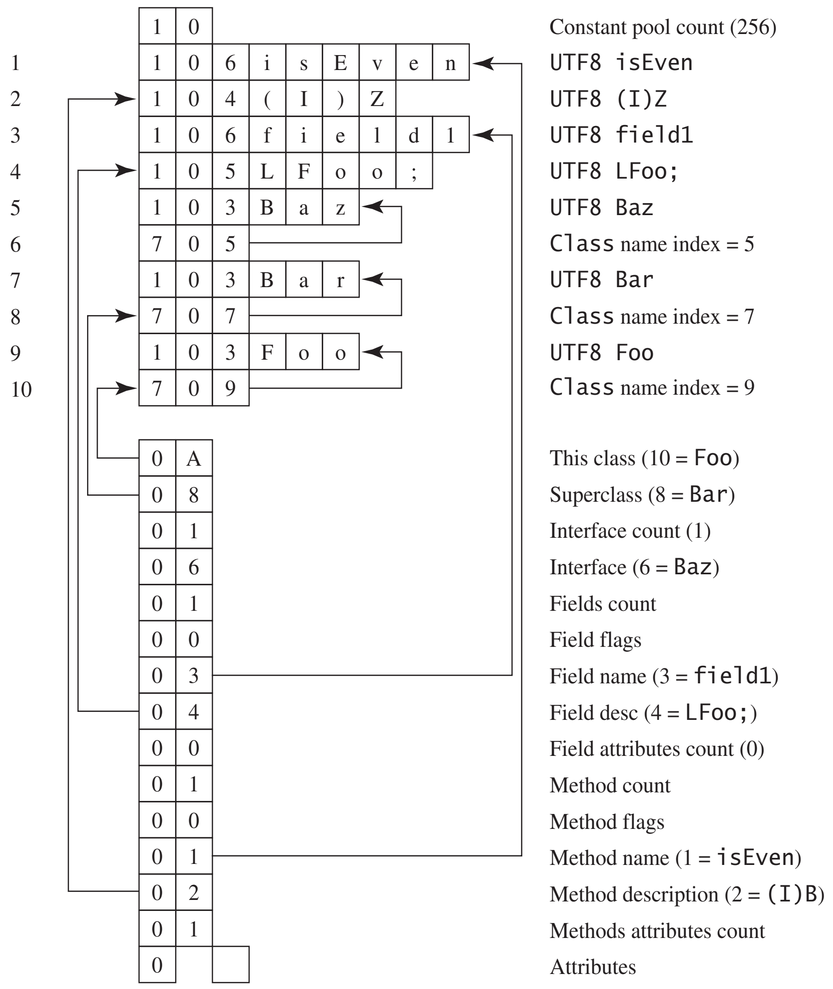


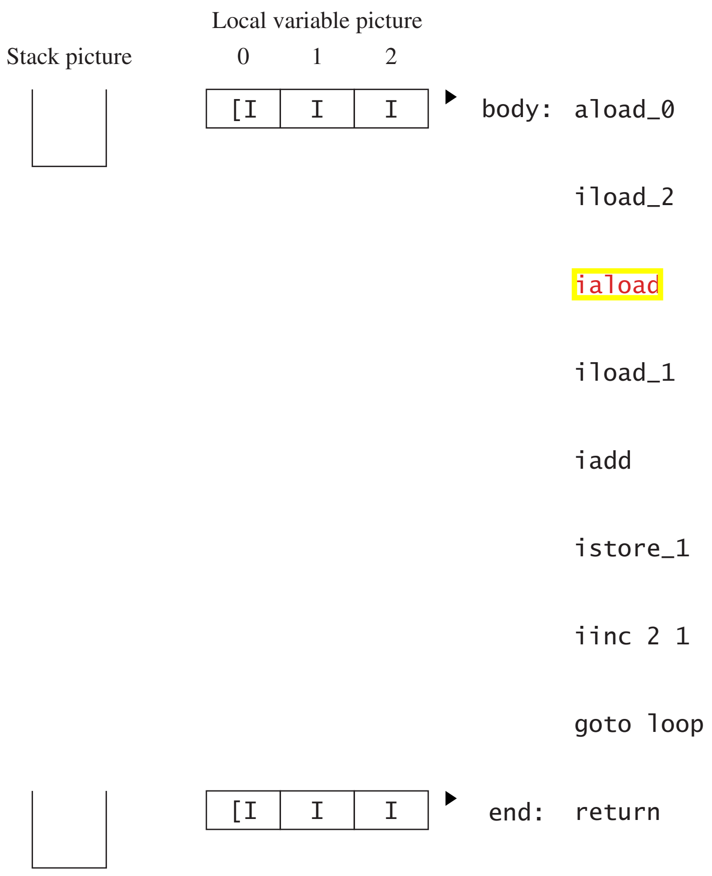
**FIGURE 6.2:** *Are all constant references correct?*

## 6.4 Are All the Instructions Valid?

Now that you know that the overall class structure is valid, you can look at the method bodies to see if the instructions within the method are correctly formed. Following are some of the questions to ask.

- Does each instruction begin with a recognized opcode?
- If the instruction takes a constant pool reference as an argument, does it point to an actual constant pool entry with the correct type?

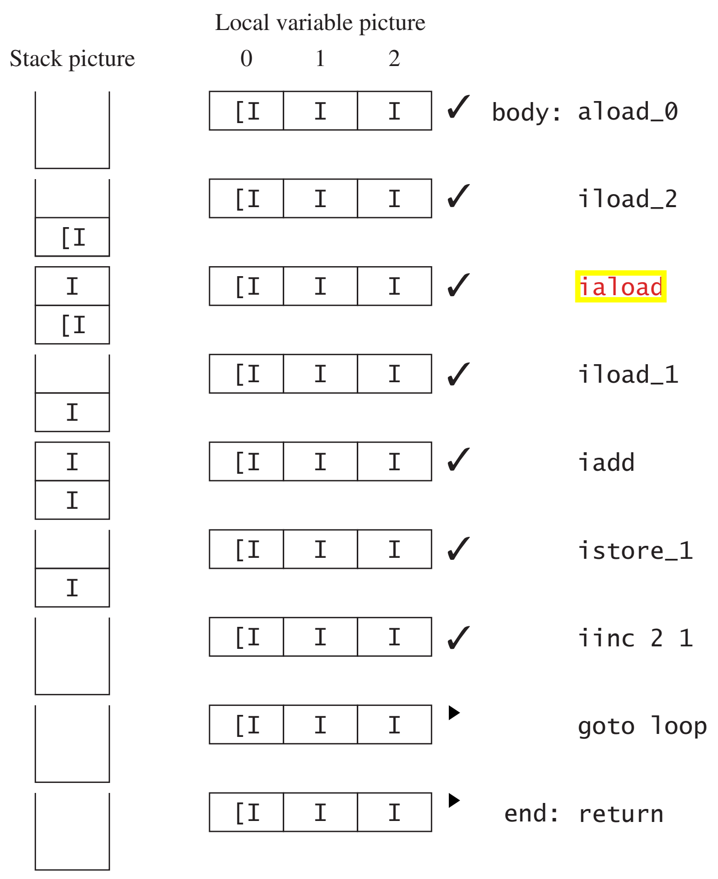
The `if_icmpge` instruction pops two elements off the stack. After executing it, the stack is empty. Control will go to either `body` (if the test fails) or `end` (if it succeeds). You have to annotate both instructions with the new stack picture:



```
▶ body:  aload_0

         iload_2

         iaload

         iload_1

         iadd

         istore_1

         iinc 2 1

         goto loop

▶ end:   return
```

▶ : instruction to check
✓: already checked instructions

You must check both paths, but it doesn't matter which one you choose to do first. In this example, we'll check the path through `body` first. We proceed as before until we hit the `goto` instruction. The picture now looks like this:



```
body: aload_0
      iload_2
      iaload
      iload_1
      iadd
      istore_1
      iinc 2 1
      goto loop
end:  return
```

▶ : instruction to check
✓: already checked instructions

The verification algorithm will reject this method, because it can't be sure of a constant stack height. If it permitted this method to execute, the stack would grow by 1 each time. Eventually, the program might overflow the stack. Because you don't want that to happen, the verification algorithm rejects this code.

### 6.5.4 Example 4: Dealing with Subclasses

One more complication: it isn't necessary for two stack pictures to be identical when two different flows of control come to the same place. Here's a (somewhat contrived) example. The example depends on three classes:

```
abstract class Person {
   abstract void printName();
}

class Programmer extends Person {
   void printName() { /* Implementation goes here */ }
}

class Author extends Person {
   void printName() { /* Implementation goes here */ }
}
```

The code we wish to verify is

```
.method public static print(ZLProgrammer;LAuthor;)V
   iload_0                  ; Is the boolean false?
   ifeq false               ; If not,
   true: aload_1            ; then push the programmer
   goto print
   false: aload_2           ; Otherwise, push the author
   print: invokevirtual Person/printName ()V
                            ; Call printName on the Person
                            ; This works whether it's an
                            ; Author or a Programmer,
                            ; since each is a Person
   done: return
.end method
```

This method takes three arguments: a `boolean` control, a `Programmer`, and an `Author`. If the control is `true` then it prints the name of the `Programmer`. Otherwise, it prints the name of the `Author`. The program arrives at `print` with either

This way of checking external references gives JVM programmers tremendous flexibility to change classes without having to recompile all the classes that use them. You only have to check for the fields and methods that are actually used. You can add or delete fields and methods, without altering the correctness of classes that don't use them.

In addition, because they're checked by name and type, you can move methods around in the source file without affecting other classes that use this class. This helps solve the "fragile base class" problem familiar to C++ programmers, who know that even a seemingly negligible change such as changing the order of methods can require recompilation of every file in the project. It is a cause of many subtle and mysterious bugs in C++ projects.

Many JVM implementations, including Sun's Java Development Kit, don't actually test external references until they are needed. This can save a lot of time. You can start using the class immediately, without having to wait for all the related classes (and all the classes related to those classes, and so on) to load. It's quite possible that many of the references will never need to be checked because the field or method is never used.

## 6.7 Java Language and Verification Algorithm

The most common programming language for writing Java virtual machine programs is Java. One thing the language guarantees is that anything you can say in Java can be translated into virtual machine code that will pass the verification algorithm.

The verification rules state that the arithmetic instructions (`iadd, ddiv`, etc.) always take two values of the same type (two `int`s, two `double`s, etc.). Say you have the following Java program fragment:

```
int i = 70;              // Call this variable 1
float j = 111.1;         // Call this variable 2
double k = i+j;          // Call this variable 3 (and 4)
```

A naïve (and invalid) translation into bytecodes would be

```
bipush 70       ; Initialize i (variable 1) to 70
istore_1
ldc 111.1       ; Initialize j (variable 2) to 111.1
fstore_2
iload_1         ; Push the integer 70
fload_2         ; Push the float 111.1
```

```
.method static fahrenheitToCelsius (F)V
.limit stack 2
.var 0 is fahrenheit from begin to end_of_computation
.var 0 is celsius from end_of_computation to end
begin:
   fload_0               ; Push fahrenheit in variable 0
   ldc 32.0              ; Subtract 32
   fsub
   ldc 5.0               ; Multiply by 5
   fmul
   ldc 9.0               ; Divide by 9
   fdiv
end_of_computation:
   fstore_0              ; Now variable 0 is celsius
   getstatic java/lang/System/out Ljava/io/PrintStream;
   fload_0               ; Print variable 0
   invokevirtual java/io/PrintStream/println (F)V
   return
end:
.end method
```

It's also possible in Java for two different variables to have the same name in different parts of a method:

```
{
   int i;
   /* i is variable 1 */
}
{
   int j;
   int i;
   /* Here, j is variable 1 and i is variable 2 */
}
```

To let the debugger know which variable is named what, use this Oolong code:

```
.var 1 is i I from scope1begin to scope1end
.var 1 is j I from scope2begin to scope2end
.var 2 is i I from scope2begin to scope2end
scope1begin:
   ; Here variable 1 is i, and variable 2 is unnamed
```

Under the Java 1.1 and later platforms, the call to defineClass is replaced with

```
c = defineClass(name, bytes, 0, bytes.length);
```

The first example is acceptable under Java 1.1 but deprecated (that is, it's considered bad style, and one day it may no longer be acceptable). If the name doesn't match the name of the class found in the bytes, then a ClassFormatError is thrown.

If the class is found neither in the cache nor wherever findClass looks for it, the class loader calls findSystemClass to see whether the system can locate a definition for the class. If findSystemClass doesn't find it, it throws a ClassNotFoundException, since that was the last chance to find the class.

### 8.3.1 Caching Classes

It's important that a class loader return the same Class object each time it's given a particular name. If the same class were loaded more than once, it would be confusing to users who might find that two classes with identical names aren't identical. Class static constructors might be invoked multiple times, causing problems for classes that were designed to expect them to be called only once.

Under Java 1.0, it was the responsibility of the class loader to cache classes itself. This is usually done with a Hashtable, as shown in the template. However, this still leaves the possibility of confusion, since two different class loaders might each load a class into the system with the same name. Java 1.1 resolves this problem by handling the caching itself. It makes this cache available to the class loader developer through a method called findLoadedClass:

```
Class findLoadedClass(String name);
```

A call to findLoadedClass replaces the cache lookup. When defineClass is called, it maps the name of the class to the Class that is returned. After that, findLoadedClass always returns that Class whenever it's given the same name, no matter which class loader invokes it.

When implementing your class loader, you will have to decide whether to use the Java 1.0 interface or the 1.1 interface. The 1.0 interface is supported on virtual machines supporting Java 1.1 but not vice versa. However, using the 1.0 interface will have different results on a JVM 1.1 if the class loader tries to define a class more than once. On a JVM 1.0, it would actually load the classes multiple times, and the system would have two different classes with the same name. These classes wouldn't share static fields or use private fields or methods on the other. On 1.1 and later JVMs, however, defineClass throws an exception when it's asked to define the class a second time anywhere in the virtual machine, even if the bytes are identical.

**TABLE 10.6:** *Selecting an arithmetic operator*

| Operator | Type | | | |
|---|---|---|---|---|
| | int | long | float | double |
| + | iadd | ladd | fadd | dadd |
| - | isub | lsub | fsub | dsub |
| / | idiv | ldiv | fdiv | ddiv |
| * | imul | lmul | fmul | dmul |
| % | irem | lrem | frem | drem |
| unary - | ineg | lneg | fneg | dneg |
| & | iand | land | - | - |
| \| | ior | lor | - | - |
| ^ | ixor | lxor | - | - |

that both subexpressions have the same type. If the expressions have different types, then one must be coerced to have the same type as the other.

## 10.8.1 Numeric Coercions

Coercion is the process of converting a value of one type into a value of a different type. Although the two values are completely different to the JVM, they mean something similar to the user. The number `1.0` (the floating-point number 1) is completely different from the number `1` (the integer 1); arithmetic instructions that apply to one of them do not apply to the other. However, it is clear that `1.0` and `1` are corresponding values in the different domains of numbers.

For example, consider this Java expression:

```
1.0 + 1
```

According to *The Java Language Specification,* the result should be the floating-point number `2.0`. The naïve transformation into bytecodes is this:

```
fconst_1                ; Push 1.0
iconst_1                ; Push 1
fadd                    ; ERROR! Can't add a float to an int
```

In order to make these two values have the same type, it is necessary to convert one of them to have the same type as the other. The primary goal is to preserve the magnitude of the number, and the secondary goal is to preserve the precision.

**TABLE 10.10:** *Casting numeric types*

| Expression | Result |
|---|---|
| `(int) f` | Demote `f` to an `int`. |
| `(double) f` | Promote `f` to a `double`. |
| `(long) f` | Demote `f` to a `long`. |
| `(float) l` | Promote `l` to a `float`. |
| `(int) i` | No change |
| `(double) (i+f)` | First `i` is converted to a `float`, then the final result is converted to a `double`. |

truncation. This means that the `char` value is always positive, but the `short` and `byte` equivalents may be negative. Table 10.11 lists some examples of what happens when you cast an `int` to a smaller type.

### 10.8.3 ~ Operator

The ~ operator is not represented in Table 10.6. The ~ operator takes an `int` or a `long` and inverts each bit. There is no instruction for this operator. Java compilers take note of the fact that, for a single bit `x`, computing `~x` is equivalent to computing `x` + –1, where + is the exclusive-or operator. To invert all the bits in the number at once, the Java compiler uses the `lxor` or `ixor` instruction with the value consisting of 64 or 32 1's. In the two's complement notation used in the Java virtual machine, an integer consisting of all 1's is equal to `-1`. For example,

```
~x
```

**TABLE 10.11:** *Converting between `int` types*

| Expression | Resu lt |
|---|---|
| `(short) 65555` | `-1` |
| `(char) 65535` | `65535` |
| `(byte) 65535` | `-1` |
| `(short) 160` | `160` |
| `(char) 160` | `160` |
| `(byte) 160` | `-2` |

```
            public Object nextElement()
            {
                if(count == 0)
                    throw new NoSuchElementException();
                else
                {
                    count--;
                    return elements[count];
                }
            }
        }
    }
```

The inner class `Enumerator` within `Stack` uses the fields `elements` and `top` from `Stack`. These refer to the fields within the object that created the `Enumerator`.

To support these, the translation includes a reference to the enclosing object, called `this$0`. It is initialized in the constructor to the object responsible for the creation of the `Enumerator`. All references to `top` and `elements` come from this reference.

The compilation of `Enumerator` produces these definitions:

```
.class Stack$Enumerator
.implements java/util/Enumeration

.field this$0 LStack;         ; The enclosing object
.field count I                ; The current count

.method <init>(LStack;)V
aload_0                       ; Call super constructor
invokespecial java/lang/Object/<init>()V
aload_0                       ; Store the enclosing object
aload_1                       ; in this$0
putfield data/structure/Stack$Enumerator/this$0 LStack;

aload_0                                ; This is the body of the
aload_1                                ; constructor:
getfield Stack/top I                   ; count = top;
putfield count I

return
.end method
```

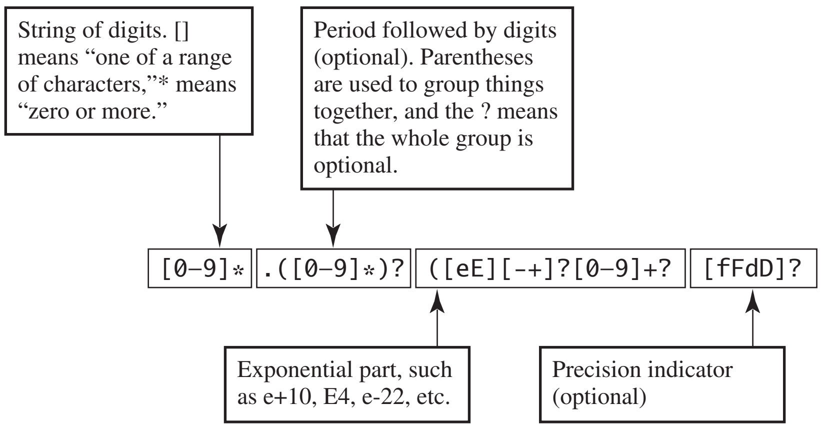

String of digits. [] means "one of a range of characters,"* means "zero or more."
Period followed by digits (optional). Parentheses are used to group things together, and the ? means that the whole group is optional.
[0-9]*
.([0-9]*)?
([eE][-+]?[0-9]+?
[fFdD]?
Exponential part, such as e+10, E4, e-22, etc.
Precision indicator (optional)

FIGURE 11.1: *Reading a regular expression*

next character, using the new state, until you reach the end of the input. If you end up in a terminal state, then the regular expression matches the input string. If the final state isn't a terminal state, or if you find yourself in a state where there is no

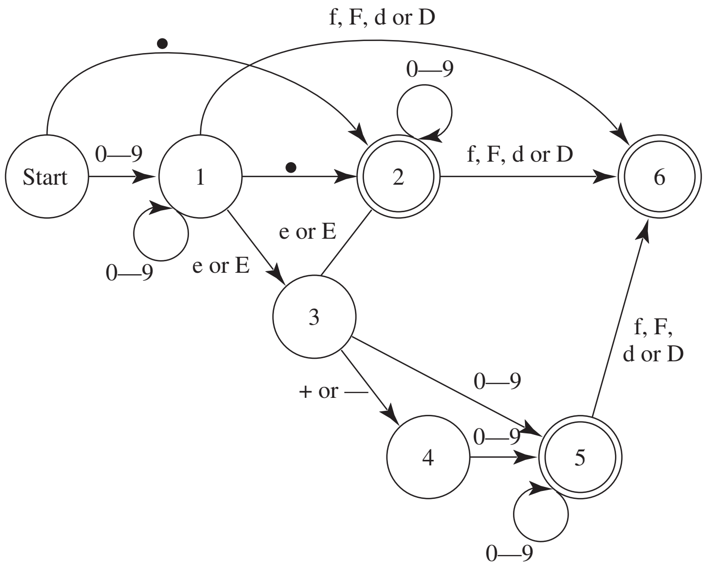


FIGURE 11.2: *Finite state machine recognizing floating-point numbers*

```
    /** Return the value of sym */
    public Object lookup(String sym) throws UnboundSymbolException
    {
        Object o = cache.get(sym);
        if(o != null)
            return o;
        if(ancestor == null)
            throw new UnboundSymbolException(sym);
        return ancestor.lookup(sym);
    }

    /** Bind the symbol to the value */
    public void bind(String symbol, Object value)
    {
        cache.put(symbol, value);
    }
}
```

The hash table `cache` maps symbols (`Strings`) to values (`Objects`). Each environment contains a pointer to the previous environment, called the *ancestor*. The `lookup` method looks up the symbol in the cache. If it is not found, it attempts to look it up in the ancestor. If it's not found in any environment, then an exception is thrown, indicating that the symbol is not bound.

The evaluator compiles each form into bytecodes that have the same semantics as the form. Each form evaluates to a value. The bytecodes leave the value of that form on the stack.

Numbers compile into instances of the class `Integer`. To compile the form

```
5
```

the compiler generates this code:

```
new java/lang/Integer                          ; Create an integer
dup
iconst_5                                       ; Push the value
invokespecial java/lang/Integer/<init>(I)V     ; Construct the
                                               ; object
```

To compile a symbol, the program looks up the symbol in the current binding environment using the `lookup` method. The compiler keeps the current binding environment in local variable 1. To compile the form

```
x
```

Sometimes you can compute the results of an operation even before it is executed and replace the code with a simple push of the results. Consider this Java code:

```
int seconds_in_a_day = 60*60*24;
```

A naïve compiler might compile this to

```
bipush 60
bipush 60
imul
bipush 24
imul
```

A better compiler would generate

```
ldc 86400
```

This single instruction is likely to execute significantly faster than the five instructions. Although the difference may seem trivial, when it is executed a million times the differences add up.

### 14.2.4 Loop Unrolling

In some loops, the time it takes to execute the body of the loop may not be all that much larger than the loop tests. Consider, for example, this code to count the number of 1 bits in the `int m`:

```
for(int i = 0; i < 32; i++)
   n += m >> i & 1;
```

A naïve Java compiler might generate this code:

```
loop: iload_1        ; Compare i to 32
bipush 32
if_icmpge break      ; Break when i >= 32
iload_3              ; Push n
iload_2              ; Push m
iload_1              ; Push i
ishr                 ; Compute m >> i
iconst_1
iand                 ; Compute m >> i & 1
iadd                 ; Add that value to n
istore_3             ; Store n
iinc 1 1             ; Increment i
goto loop            ; Loop again
```

This code will increment i 32 times and test if i is less than 32 the same number of times. It's obvious that the test will fail the first 31 times and succeed the last time. You can eliminate the tests by "unrolling" the loop, like this:

```
iload_2      ; Push m
iconst_0     ; Push 0
ishr         ; Compute m >> 0
iconst_1     ; Push 1
iand         ; Compute m >> 0 & 1 (the first bit)
iload_3      ; Push n
iadd         ; Add the result to n
istore_3     ; store n

iload_2      ; Push m
iconst_1     ; Push 1
ishr         ; Compute m >> 1
iconst_1     ; Push 1
iand         ; Compute m >> 1 & 1 (the second bit)
iload_3      ; Add n
iadd
istore_3     ; Store n

;; Repeat this pattern 29 more times

iload_2      ; Push m
bipush 31    ; Push 31
ishr         ; Compute m >> 31 (the leftmost bit)
iconst_1     ; Push 1
iand         ; Compute m >> 31 & 1
iload_3      ; Push n
iadd         ; Add the last bit to n
istore_3     ; Store n
```

Although this greatly expands the size of the resulting code, the total number of instructions executed is reduced. It also allowed us to completely eliminate local variable 1, which had been used as the loop counter. This results in a speedup of 200% to 400%, depending on the virtual machine implementation used.

### 14.2.5 Peephole Optimization

Compilers often generate code that has redundant instructions. A peephole optimizer looks at the resulting bytecodes to eliminate some of the redundant instructions. This involves looking at the bytecodes a few instructions at a time, rather than doing wholesale reorganization of the code. The name "peephole optimization" comes from the tiny "window" used to view the code.

which can be further reduced to

```
ldc 86400
```

Table 14.1 lists some other rules for a peephole optimizer. The variables `x` and `y` stand for any field or local variable. For the operations `aload` and `astore`, you may perform the same operations on the corresponding `float` and `int` instructions.

**TABLE 14.1:** *Poophole optimization rules*

| Replace | With | Reason |
|---|---|---|
| `dup`<br>`swap` | `dup` | The result of a `dup` is two of the same element; swapping produces an identical result. |
| `swap`<br>`swap` | | A double swap accomplishes nothing. |
| `iinc 1 1`<br>`iinc 1 1` | `iinc 1 2` | Since it takes just as long to increment by 2 as to increment by 1 (on most systems), you might as well do the increment just once. |
| `nop` | | A `nop` instruction has no effect, so you might as well eliminate it. |
| `getfield` *x*<br>`getfield` *x* | `getfield` *x*<br>`dup` | Two `getfield` instructions one immediately after the other leave no opportunity for the field value to change, so you can replace the second one with a `dup`. *Exception:* If the field is marked `volatile`, then its value may be changed by some other thread, and this optimization would be in error. |
| `aload` *x*<br>`aload` *x* | `aload` *x*<br>`dup` | Some systems have special fast operations for duplicating the top element of the stack. |
| `astore` *x*<br>`aload` *x* | `dup`<br>`astore` *x* | Some systems have special fast operations for duplicating the top element of the stack. |
| `aload` *x*<br>`astore` *x* | | These instructions cancel each other out. |
| `astore` *x*<br>`astore` *x* | `astore` *x*<br>`pop` | Storing into the same variable twice is unnecessary; only the second store needs to be performed. |
| `aload` *x*<br>`pop` | | These instructions cancel each other out. |
| `aload` *x*<br>`aload` *y*<br>`areturn` | `aload` *y*<br>`areturn` | Since the value of *x* is left on the top of the stack at the time of the return, there is no reason to push it. This optimization applies to any instruction except a method invocation, branch instruction, or storing into a field. |

```
    public static setSecurityManager(SecurityManager sm)
    {
         if(security != null)
              throw new SecurityException
                 ("SecurityManager already set");
         security = sm;
    }
}
```

The integrity of the `security` field is critical, since an applet that could set that field could control the security policy of the system. This makes `security` a likely target for attack.

### 15.4.3 Bypassing Java Security

The most straightforward attack would be to try to set the `security` field. The attacker's applet might be written in Java:

```
public class NastyApplet extends Applet
{
   void attack()
   {
        System.security = null;
        // The security manager is null, so everything is
        // permitted
        // Put code to wreak havoc here
   }
}
```

When this class is compiled, the Java compiler will notice that the `security` field is private to the `System` class and refuse to compile the file. This would not stop a determined attacker, who would proceed to rewrite the applet in Oolong:

```
.class public NastyApplet
.super java/applet/Applet

.method attack()V
aconst_null              ; Install null as the
                         ; security manager
putstatic java/lang/System/security Ljava/lang/SecurityManager;
;; Wreak havoc
.end method
```

### Exercise 3.4

This is more than just a matter of substituting D's for I's; you also have to change the variable numbers because each value takes two slots. The answer:

```
.method public static Dcalc(DDDD)D
.var 0 is a D
.var 2 is b D
.var 4 is c D
.var 6 is x D
dload_0     ; Push a
dload 6     ; Push x
dload 6     ; Push x again
dmul        ; Calculate x^2
dmul        ; Calculate ax^2

dload_2     ; Push b
dload 6     ; Push x
dmul        ; Push bx

dload 4     ; Push c

dadd        ; Compute bx+c
dadd        ; Compute ax^2+bx+c

dreturn
.end method
```

The maximum stack height written this way is 6.

### Exercise 3.5

```
.method public static icalc2(IIII)I
iload_0     ; Push a
iload_3     ; Push x
imul        ; Calculate ax

iload_1     ; Push b
iadd        ; Calculate ax+b

iload_3     ; Push x
imul        ; Compute ax^2+bx
```