

7. Extensions to *j--*

7.1 The Purpose of this Chapter

Our intention here is to identify Java language constructs that are not yet in *j--* but might be added. We will discuss some of the issues that must be addressed in implementing each of these constructs. The actual tasks of implementing these constructs in the *j--* compiler are left as exercises.

We assume that you have already modified Scanner (or the JavaCC file) for recognizing any necessary tokens, and that you have already modified Parser (or the JavaCC file) for parsing these constructs and creating new J-nodes (classes) for representing these in the abstract syntax tree (AST).

7.2 Implementing Additional Java Constructs in *j--*

7.2.1 Introduction

We see the following extensions to *j--*.

- Adding Java interfaces; an interface is nearly a purely abstract class.
- Adding Java static blocks and instance blocks.
- Adding both the classic Java for-statement and the enhanced Java for statement.

7.2.2 Interfaces

An interface defines a type that obeys a contract. Declare a variable to be of an interface type and you can be sure that it will refer to objects that support certain behaviors. Any class that implements the interface must support those same behaviors. Think of an interface as a purely abstract class.

From the point of view of analysis, interfaces have all sorts of rules, which must be enforced. Method `analyze()`, in both `JInterfaceDeclaration` and `JClassDeclaration` is where most of these rules may be enforced. These rules include:

1. Although an interface may have `public`, `protected` or `private` modifiers, only `public` is meaningful for interfaces not nested within other classes.
2. The compiler must also allow the modifiers `abstract`, `static` and `strictfp`. The `abstract` modifier is redundant (as all interfaces are abstract) but the compiler must allow it. The `static` modifier has meaning only for member (nested) interfaces. The `strictfp` modifier has meaning if and only if one is supporting (`float` or `double`) floating-point arithmetic.
3. An interface may extend any number of (super-) interfaces.

7-1

4. A class may implement as many interfaces as it likes. Notice that this will require our changing the parser's `classDeclaration()` method so that it parses an (optional) `implements` clause and stores a (possibly empty) list of interfaces.
5. Every field declaration in an interface is implicitly `public`, `static` and `final`, but any of these three modifiers may be present even if unnecessary. Every field must be initialized. The initializing expression need not be constant and may reference other fields appearing textually before the reference.
6. The initializing expression in an interface's field declaration
7. There are rules about the inheritance of fields; see the language specification.
8. Every method declaration in an interface is implicitly `public` and `abstract` but either of these modifiers may be present in the declaration. Because the methods are abstract, its body is always a semicolon.
9. An interface's method can never be declared `static` or `final`, but class methods implementing them can be `final`.
10. There are additional rules governing method inheritance, overriding, and overloading abstract methods in interfaces; see the language specification.
11. Any class implementing an interface (or a list of interfaces) must implement all of the methods of the interface(s).

In code generation, there are several facts to consider:

1. In the JVM, an interface is a kind of class. So, just as for a class, one uses `CLEmitter`'s `addClass()` method for introducing an interface.
2. The list of modifiers for the `addClass()`'s formal parameter `accessFlags` must include both `interface` and `abstract`. The `superClass` argument should be the JVM name for `java.lang.Object`. Of course, the argument passed for `superInterfaces` should be the (possibly empty) list of the super-interfaces (from the `extends` clause).
3. The list of modifiers of any field defined by `addField()` must include `public`, `static`, and `final`.
4. The list of modifiers of any method defined by `addMethod()` must include `public` and `abstract`.

Otherwise, use `JClassDeclaration` as a guide to, and so a starting point for implementing `JInterfaceDeclaration`.

7.2.3 Static Initializers and Instance Initializers

A *static initializer* is a class (or interface) member. It is a block of initializing code that is executed before the class enclosing it is referenced. It is identified by the `static` keyword as in,

```
static {
```

7-2

```
    ...  
}
```

The code in the static initializer is executed before the enclosing class is referenced. As such, the code goes into the `<clinit>` method – the same place that class field initializations go. To learn more about static initializers, see section 8.7 of the *Java Language Specification* (Gosling, 2005). The order that fields and static initializers are executed (and so the order they appear in `<clinit>`) is the order that the members appear in the class; the only exception to this is “that `final` class variables and fields of interfaces whose values are compile-time constants are initialized first.” See section 12.4.2 of (Gosling, 2005).

Rules, which static initializers must follow include:

1. It is a compile-time error to **throw** an exception or to **return** in the initializing code.
2. Static initializers may not refer to **this** or **super**.
3. There are restrictions in referring to class variables that are defined textually further in the code.

An instance initializer is a block of initializing code that is executed each time an instance of the enclosing class is instantiated. Therefore, its code goes into the constructors, that is `<init>`. Make sure this code isn’t executed twice – that is, in a constructor whose code starts off with `this ()`.

Rules, which instance initializers must follow include:

1. It is a compile-time error to **throw** an exception or to **return** in the initializing code.
2. Instance initializers may refer to **this**.

7.2.4 Control Constructs

Control constructs generally control the sequencing of execution in a program. In Chapter 6, we saw how the if-statement and while-statements compile to JVM code.

7.2.4.1 The do-while statement

The do-while statement takes the form,

```
do  
    Statement  
while ( Test )
```

The Statement is always executed at least once and the Test is at the bottom of the loop.

7-3

As an example, consider

```
do {
    sum = sum + i;
    i = i - 1;
} while (i > 0);
```

The Sun Java compiler, javac would produce for this:

```
top:  iload sum'
      iload i'
      iadd
      istore sum'
      iload i'
      iconst_1
      isub
      istore i'
      iload i'
      ifgt top
```

Compiling the do-while statement is very much like compiling the while statement.

7.2.4.2 The Classic `for` statement

What's interesting about the classic for-statement is that it may be reformulated as a while-statement, even in Java. For example, the template

```
for (Initialization; Test; Increment)
    Statement
```

may be expressed as

```
{
    Initialization
    while ( Test ) {
        Statement;
        Increment;
    }
}
```

Of course, we must take account of the fact that either `<initialization>`, `<test>` or `<increment>` may be empty, and translate appropriately.

This means that the for-statement may be translated in either of two ways:

1. We can generate JVM code directly, following the pattern illustrated in Chapter 6.

2. We can first rewrite the AST, replacing the JForStatement node with the block above, and then apply codegen() to that subtree. Notice how the enclosing block captures the limited scope of any variables declared in <initialization>.

Which way is better?

7.2.4.3 The Enhanced `for` statement

The Java enhanced `for` statement is used to iterate over collections. Syntactically, it looks something like

```
for ( Type Identifier : Expression )  
    Statement
```

How this can be interpreted depends on the type of Expression. If Expression's type is a subtype of `Iterable`, let I be the type of Expression. `iterator()`. Then our enhanced `for` statement may be expressed as the following classic `for` statement.

```
for ( I i' = Expression.iterator(); i'.hasNext() ; ) {  
    Type Identifier = i'.next();  
    Statement  
}
```

The variable `i'` is compiler generated in such a way as not to conflict with any other variables.

Otherwise, Expression must have an array type `T[]`. In this case our enhanced `for` statement may be expressed as

```
T[] a' = Expression;  
for ( int i' = 0; i' < a'.length; i'++ ) {  
    Type Identifier = a'[i'];  
    Statement  
}
```

The variables `a'` and `i'` are compiler generated in such a way as not to conflict with any other variables. This can be compiled similarly to the classic `for` statement.

7.2.4.5 The `switch` statement

The `switch` statement is more involved than other control constructs. Fortunately, the JVM provides special support for it. Code generation for the `switch` statement is discussed in section 7.10 of the JVM specification (Lindholm and Yellin, 1999).

As an example, let's consider a method that makes use of a simple switch statement.

```
int digitTight( char c ) {
    switch ( c ) {
        case '1': return 1;
        case '0': return 0;
        case '2': return 2;
        default: return -1;
    }
}
```

This method converts digit characters '0', '1', and '2' to the integers they denote; any other character is converted to -1. This is not the best way to do this conversion but it makes for a simple example of the switch statement. Let's take a look at the code that `javac` produces, using the `javap` tool.

```
int digitTight(char);
Code:
  Stack=1, Locals=2, Args_size=2
  0:   iload_1
  1:   tableswitch{ //48 to 50
           48: 30;
           49: 28;
           50: 32;
           default: 34 }
 28:   iconst_1
 29:   ireturn
 30:   iconst_0
 31:   ireturn
 32:   iconst_2
 33:   ireturn
 34:   iconst_m1
 35:   ireturn
```

Notice a few things here. Firstly, the JVM uses a table to map the Unicode character representations to the code locations for each of the corresponding cases. For example, 48 (the Unicode representation for the character '0') maps to location 30, the location of the JVM code for the `return 0`. Notice also that the characters have been sorted on their Unicode values: 48, 49 and 50. Finally, since the characters are consecutive, the `tableswitch` instruction can simply calculate an offset in the table to obtain the location to which it wants to branch.

Now consider a second example, where the range of the characters we are switching on is a little sparser; that is, the characters are not consecutive, even when sorted.

```
int digitSparse( char c ) {
    switch ( c ) {
        case '4': return 4;
        case '0': return 0;
    }
}
```

7-6

```

        case '8': return 8;
        default: return -1;
    }
}

```

The code that javac produces for this takes account of this sparseness.

```

int digitSparse(char);
Code:
Stack=1, Locals=2, Args_size=2
0:   iload_1
1:   lookupswitch{ //3
           48: 38;
           52: 36;
           56: 40;
           default: 43 }
36:  iconst_4
37:  ireturn
38:  iconst_0
39:  ireturn
40:  bipush  8
42:  ireturn
43:  iconst_m1
44:  ireturn

```

Again, a table is used, but this time a `lookupswitch` instruction is used in place of the previous `tableswitch`. The table here represents a list of pairs, mapping a Unicode representation for each character to a location. Again the table is sorted on the Unicode representations but the sequence of characters is too sparse for using the offset technique for finding the appropriate code location. Rather, `lookupswitch` searches the table for the character it is switching on and then branches to the corresponding location. Because the table is sorted on the characters, a binary search strategy may be used for large tables.

Consider one more example, where the sorted characters are almost but not quite sequential.

```

int digitClose( char c) {
    switch ( c ) {
        case '1': return 1;
        case '0': return 0;
        case '3': return 3;
        default: return -1;
    }
}

```

That the range of characters we are switching on is not *too* sparse suggests using the `tableswitch` instruction, which uses an offset to choose the proper location to branch

to. The `javac` compiler does just this, as illustrated by the following output provided by the `javap` tool

```
.
int digitClose(char);
Code:
  Stack=1, Locals=2, Args_size=2
  0:   iload_1
  1:   tableswitch{ //48 to 51
                48: 34;
                49: 32;
                50: 38;
                51: 36;
                default: 38 }
 32:   iconst_1
 33:   ireturn
 34:   iconst_0
 35:   ireturn
 36:   iconst_3
 37:   ireturn
 38:   iconst_m1
 39:   ireturn
```

Notice a sequential table is produced. Of course, the entry for character '2' (Unicode 50) maps to location 38 (the default case) because it is not one of the explicit cases.

So our compiler must construct a list of value-label pairs, mapping a case value to a label we will emit to mark the location of code to branch to. We then sort that list, on the case values and decide, based on sparseness, whether to use a `tableswitch` instruction (not sparse) or a `lookupswitch` (sparse) instruction. `CLEmitter` provides a method for each choice.

`CLEmitter`'s `addTABLESWITCHInstruction()` method provides for several arguments: a `default` label, a lower bound on the case values, an upper bound on the case values, and a sequential list of labels.

`CLEmitter`'s `addLOOKUPSWITCHInstruction()` method provides for a different set of arguments: a `default` label, a count of the number of value-label pairs in the table, and a `TreeMap` that maps case values to labels.

Of course, our compiler must decide which of the two instructions to use.

7.2.5 Conditional Expressions

Conditional expressions are compiled in a manner identical to if-else statements. The only difference is that in this case, both the consequent and the alternative are expressions.

7-8

For example, consider the assignment,

```
z = x > y ? x - y : y - x;
```

The javac compiler produces the following code for this:

```
43:    iload_1
44:    iload_2
45:    if_icmple 54
48:    iload_1
49:    iload_2
50:    isub
51:    goto 57
54:    iload_2
55:    iload_1
56:    isub
57:    istore_3
```

As for the if-then, we compile the boolean test expression using the 3-argument version of `codegen()`.

7.2.6 Conditional || operation.

The conditional ||, like the conditional &&, is short-circuited. That is, in

$e_1 || e_2$

if e_1 evaluates to true, then e_2 is not evaluated and the whole expression is true. For example, javac compiles

```
if (x < 0 || y > 0 )
    x = y;
```

to produce

```
58:    iload_1
59:    iflt 66
62:    iload_2
63:    ifle 68
66:    iload_2
67:    istore_1
68:
```

On the other hand,

```
while (x < 0 || y > 0 )
    x = y;
```

compiles to produce

```
68:    iload_1
69:    iflt 76
72:    iload_2
73:    ifle 81
76:    iload_2
77:    istore_1
78:    goto 68
81:
```

7.2.7 Exception Handling

Exception handling in Java is captured by the **try-catch-finally** and **throw** statement. Additionally, there is the **throws** clause, which serves to help the compiler ensure that there is a catch for every exception thrown.

7.2.7.1 Introduction to Exception Handling

7.2.7.2 The try-catch-finally statement

7.2.7.3 The throw statement

7.2.7.2 The throws declaration

7.2.8 Implementing Additional Primitive Types

7.2.8.1 What it Means to Add a Type

When we add a type, we are usually adding a primitive type. (There is no reason why we can't add built-in reference types such as matrices for which we can overload the arithmetic operators; but doing so would go against the spirit of Java.) And, since the primitive types in Java that are not already in *j--* are numeric types, the new types must work in concert with the current types.

7.2.8.2 Adding double

Adding double precision floating point numbers to `j`—introduces several wrinkles:

1. Arithmetic requires new JVM instructions, e.g., `dadd`.
2. Each double precision floating-point number occupies *two* words. This requires that the offset be incremented by 2 each time a double variable is declared.
3. Mixed arithmetic introduces the need for implicit conversion. For example, if `d` and `e` are double variables, and `i` is an integer variable,

For example, consider the following contrived method.

```
void foo() {
    int i = 4;
    double d = 3.0;
    double e = 5.0;

    e = d * i;
}
```

The `javac` compiler produces the following JVM code:

```
void foo();
Code:
  Stack=4, Locals=6, Args_size=1
  0:  iconst_4
  1:  istore_1
  2:  ldc2_w    #2; //double 3.0d
  5:  dstore_2
  6:  ldc2_w    #4; //double 5.0d
  9:  dstore   4
 11:  dload_2
 12:  iload_1
 13:  i2d
 14:  dmul
 15:  dstore   4
 17:  return
```

Notice several things here. Firstly, notice that the stack offsets take account of `i` being a one-word `int`, and `d` and `e` being two-word doubles. Secondly, notice the mix of types – an `iload` instruction is used to load the integer and a `dload` instruction is used to load the double precision value. The arithmetic is done in double precision, using the `dmul` instruction. Therefore, the `int` `i` must be promoted (converted to a more precise representation) to a `double` value using the `i2d` instruction.

In the assignment and operations, `analyze()` must be modified for dealing with this mixed arithmetic. In the case of a mix of `int` and `double`, the rule is pretty simple: convert the `int` to a `double`. Of course, the reader will want to consult the Java

7-11

Language Specification (Gosling, 2005), especially Chapters 5 and 15. The rules are pretty easy to ferret out for just `ints` and `doubles`.

7.2.8.3 Adding additional primitive types

When adding additional (to `int` and `double`) primitive types, the rules become more complicated. One must worry about when to do conversions and how to do them. One must follow the Java Language Specification (Gosling, 2005).

Once one has more than two numeric types to deal with, it is probably best to re-think the conversion scheme and build a simple framework for deciding which type to promote and to promote one type to another. There are fifteen primitive type conversion instructions available to us in the JVM and we will want to make effective use of them.

7.3 More Challenging (and Involved) Extensions

7.3.1 Nested Classes

7.3.2 Enum Types

7.3.3 Generic Types

Further Reading

Much more information about both the JVM and class files may be found in (Lindholm and Yellin, 1999). See Chapter 3 for detailed information about the JVM's structure. See 7-12

Chapter 4 for a detailed description of the class file format. See Chapter 7 for hints on compiling various Java language constructs to JVM byte code.

Exercises

- 7.1 Add interfaces (as a type declaration) to *j--*. Make any changes that are necessary to classes and class declarations for accommodating interfaces.
- 7.2 Add conditional expressions to *j--*.
- 7.3 Add the conditional or operator `||` to *j--*. Be sure to implement short-circuiting correctly.
- 7.4 Add the **switch** statement to *j--*.
- 7.5 Add the classic for statement to *j--*.
- 7.6 Add the enhanced for statement to *j--*.
- 7.7 Add the break statement to *j--*.
- 7.8 Add the continue statement to *j--*.
- 7.9 Add exception handling to *j--*: the try-catch-finally statement, the throw statement and the throws clause.
- 7.10 Add the double precision type and double precision arithmetic to *j--*. Although you need not implement implicit conversions, you should support all the necessary casts.
- 7.11 Add the float type to *j--*.
- 7.12 Add the short type to *j--*.
- 7.13 Add the byte type to *j--*.

