docs.oracle.com

Chapter 3. Compiling for the Java Virtual Machine

63-80 minutes

Chapter 3. Compiling for the Java Virtual Machine

The Java Virtual Machine machine is designed to support the Java programming language. Oracle's JDK software contains a compiler from source code written in the Java programming language to the instruction set of the Java Virtual Machine, and a run-time system that implements the Java Virtual Machine itself. Understanding how one compiler utilizes the Java Virtual Machine is useful to the prospective compiler writer, as well as to one trying to understand the Java Virtual Machine itself. The numbered sections in this chapter are not normative.

Note that the term "compiler" is sometimes used when referring to a translator from the instruction set of a Java Virtual Machine to the instruction set of a specific CPU. One example of such a translator is a just-in-time (JIT) code generator, which generates platform-specific instructions only after Java Virtual Machine code has been loaded. This chapter does not address issues associated with code generation, only those associated with compiling source code written in the Java programming language to Java Virtual Machine instructions.

This chapter consists mainly of examples of source code together with annotated listings of the Java Virtual Machine code that the `javac` compiler in Oracle’s JDK release 1.0.2 generates for the examples. The Java Virtual Machine code is written in the informal “virtual machine assembly language” output by Oracle's `javap` utility, distributed with the JDK release. You can use `javap` to generate additional examples of compiled methods.

The format of the examples should be familiar to anyone who has read assembly code. Each instruction takes the form:

```
<index> <opcode> [ <operand1> [ <operand2>...
]] [<comment>]
```

The <index> is the index of the opcode of the instruction in the array that contains the bytes of Java Virtual Machine code for this method. Alternatively, the <index> may be thought of as a byte offset from the beginning of the method. The <opcode> is the mnemonic for the instruction's opcode, and the zero or more <operandN> are the operands of the instruction. The optional <comment> is given in end-of-line comment syntax:

```
8   bipush 100      // Push int constant 100
```

Some of the material in the comments is emitted by `javap`; the rest is supplied by the authors. The <index> prefacing each instruction may be used as the target of a control transfer instruction. For instance, a *goto 8* instruction transfers control to the instruction at index 8. Note that the actual operands of Java

Virtual Machine control transfer instructions are offsets from the addresses of the opcodes of those instructions; these operands are displayed by `javap` (and are shown in this chapter) as more easily read offsets into their methods.

We preface an operand representing a run-time constant pool index with a hash sign and follow the instruction by a comment identifying the run-time constant pool item referenced, as in:

```
10   ldc #1          // Push float constant 100.0
```

or:

```
9    invokevirtual #4    // Method
Example.addTwo(II)I
```

For the purposes of this chapter, we do not worry about specifying details such as operand sizes.

3.2. Use of Constants, Local Variables, and Control Constructs

Java Virtual Machine code exhibits a set of general characteristics imposed by the Java Virtual Machine's design and use of types. In the first example we encounter many of these, and we consider them in some detail.

The `spin` method simply spins around an empty for loop 100 times:

```
void spin() {
    int i;
    for (i = 0; i < 100; i++) {
        ;    // Loop body is empty
```

```
    }
}
```

A compiler might compile `spin` to:

```
0    iconst_0         // Push int constant 0
1    istore_1         // Store into local variable
1 (i=0)
2    goto 8           // First time through don't
increment
5    iinc 1 1         // Increment local variable
1 by 1 (i++)
8    iload_1          // Push local variable 1 (i)
9    bipush 100       // Push int constant 100
11   if_icmplt 5      // Compare and loop if less
than (i < 100)
14   return           // Return void when done
```

The Java Virtual Machine is stack-oriented, with most operations taking one or more operands from the operand stack of the Java Virtual Machine's current frame or pushing results back onto the operand stack. A new frame is created each time a method is invoked, and with it is created a new operand stack and set of local variables for use by that method (§2.6). At any one point of the computation, there are thus likely to be many frames and equally many operand stacks per thread of control, corresponding to many nested method invocations. Only the operand stack in the current frame is active.

The instruction set of the Java Virtual Machine distinguishes operand types by using distinct bytecodes for operations on its various data types. The method `spin` operates only on values of type `int`. The instructions in its compiled code chosen to operate on typed data (*iconst_0*, *istore_1*, *iinc*, *iload_1*, *if_icmplt*) are all specialized for type `int`.

The two constants in `spin`, `0` and `100`, are pushed onto the operand stack using two different instructions. The `0` is pushed using an *iconst_0* instruction, one of the family of *iconst_<i>* instructions. The `100` is pushed using a *bipush* instruction, which fetches the value it pushes as an immediate operand.

The Java Virtual Machine frequently takes advantage of the likelihood of certain operands (`int` constants *-1*, *0*, *1*, *2*, *3*, *4* and *5* in the case of the *iconst_<i>* instructions) by making those operands implicit in the opcode. Because the *iconst_0* instruction knows it is going to push an `int 0`, *iconst_0* does not need to store an operand to tell it what value to push, nor does it need to fetch or decode an operand. Compiling the push of `0` as *bipush 0* would have been correct, but would have made the compiled code for `spin` one byte longer. A simple virtual machine would have also spent additional time fetching and decoding the explicit operand each time around the loop. Use of implicit operands makes compiled code more compact and efficient.

The `int i` in `spin` is stored as Java Virtual Machine local variable *1*. Because most Java Virtual Machine instructions operate on values popped from the operand stack rather than directly on local variables, instructions that transfer values between local variables and the operand stack are common in code compiled for the Java Virtual Machine. These operations also have special support in the instruction set. In `spin`, values are transferred to and from local variables using the *istore_1* and *iload_1* instructions, each of which implicitly operates on local variable *1*. The *istore_1* instruction pops an `int` from the operand stack and stores it in local variable *1*. The *iload_1* instruction pushes the value in local variable *1* on to the operand stack.

The use (and reuse) of local variables is the responsibility of the compiler writer. The specialized load and store instructions should encourage the compiler writer to reuse local variables as much as is feasible. The resulting code is faster, more compact, and uses less space in the frame.

Certain very frequent operations on local variables are catered to specially by the Java Virtual Machine. The *iinc* instruction increments the contents of a local variable by a one-byte signed value. The *iinc* instruction in `spin` increments the first local variable (its first operand) by *1* (its second operand). The *iinc* instruction is very handy when implementing looping constructs.

The `for` loop of `spin` is accomplished mainly by these instructions:

```
5   iinc 1 1        // Increment local variable
1 by 1 (i++)
8   iload_1         // Push local variable 1 (i)
9   bipush 100      // Push int constant 100
11  if_icmplt 5     // Compare and loop if less
than (i < 100)
```

The *bipush* instruction pushes the value *100* onto the operand stack as an `int`, then the *if_icmplt* instruction pops that value off the operand stack and compares it against *i*. If the comparison succeeds (the variable `i` is less than `100`), control is transferred to index *5* and the next iteration of the `for` loop begins. Otherwise, control passes to the instruction following the *if_icmplt*.

If the `spin` example had used a data type other than `int` for the loop counter, the compiled code would necessarily change to reflect the different data type. For instance, if instead of an `int` the `spin` example uses a `double`, as shown:

```
void dspin() {
    double i;
    for (i = 0.0; i < 100.0; i++) {
        ;    // Loop body is empty
    }
}
```

the compiled code is:

```
Method void dspin()
0    dconst_0        // Push double constant 0.0
1    dstore_1        // Store into local
variables 1 and 2
2    goto 9          // First time through don't
increment
5    dload_1         // Push local variables 1
and 2
```

```
6    dconst_1        // Push double constant 1.0
7    dadd            // Add; there is no dinc instruction
8    dstore_1        // Store result in local variables 1 and 2
9    dload_1         // Push local variables 1 and 2
10   ldc2_w #4       // Push double constant 100.0
13   dcmpg           // There is no if_dcmplt instruction
14   iflt 5          // Compare and loop if less than (i < 100.0)
17   return          // Return void when done
```

The instructions that operate on typed data are now specialized for type `double`. (The *ldc2_w* instruction will be discussed later in this chapter.)

Recall that `double` values occupy two local variables, although they are only accessed using the lesser index of the two local variables. This is also the case for values of type `long`. Again for example,

```
double doubleLocals(double d1, double d2) {
    return d1 + d2;
}
```

becomes

```
Method double doubleLocals(double,double)
```

```
0    dload_1        // First argument in local
variables 1 and 2
1    dload_3        // Second argument in local
variables 3 and 4
2    dadd
3    dreturn
```

Note that local variables of the local variable pairs used to store `double` values in `doubleLocals` must never be manipulated individually.

The Java Virtual Machine's opcode size of 1 byte results in its compiled code being very compact. However, 1-byte opcodes also mean that the Java Virtual Machine instruction set must stay small. As a compromise, the Java Virtual Machine does not provide equal support for all data types: it is not completely orthogonal (Table 2.11.1-A).

For example, the comparison of values of type `int` in the `for` statement of example `spin` can be implemented using a single *if_icmplt* instruction; however, there is no single instruction in the Java Virtual Machine instruction set that performs a conditional branch on values of type `double`. Thus, `dspin` must implement its comparison of values of type `double` using a *dcmpg* instruction followed by an *iflt* instruction.

The Java Virtual Machine provides the most direct support for data of type `int`. This is partly in anticipation of efficient implementations of the Java Virtual Machine's operand stacks and local variable arrays. It is also motivated by the frequency of `int` data in typical programs. Other integral types have less direct support. There are no `byte`, `char`, or `short` versions of the store, load, or add instructions, for instance. Here is the `spin` example written using a `short`:

```
void sspin() {
    short i;
    for (i = 0; i < 100; i++) {
        ;    // Loop body is empty
    }
}
```

It must be compiled for the Java Virtual Machine, as follows, using instructions operating on another type, most likely `int`, converting between `short` and `int` values as necessary to ensure that the results of operations on `short` data stay within the appropriate range:

```
Method void sspin()
0   iconst_0
1   istore_1
2   goto 10
5   iload_1          // The short is treated as
though an int
6   iconst_1
7   iadd
8   i2s              // Truncate int to short
9   istore_1
10  iload_1
11  bipush 100
13  if_icmplt 5
```

```
16   return
```

The lack of direct support for `byte`, `char`, and `short` types in the Java Virtual Machine is not particularly painful, because values of those types are internally promoted to `int` (`byte` and `short` are sign-extended to `int`, `char` is zero-extended). Operations on `byte`, `char`, and `short` data can thus be done using `int` instructions. The only additional cost is that of truncating the values of `int` operations to valid ranges.

The `long` and floating-point types have an intermediate level of support in the Java Virtual Machine, lacking only the full complement of conditional control transfer instructions.

The Java Virtual Machine generally does arithmetic on its operand stack. (The exception is the *iinc* instruction, which directly increments the value of a local variable.) For instance, the `align2grain` method aligns an `int` value to a given power of 2:

```
int align2grain(int i, int grain) {
    return ((i + grain-1) & ~(grain-1));
}
```

Operands for arithmetic operations are popped from the operand stack, and the results of operations are pushed back onto the operand stack. Results of arithmetic subcomputations can thus be made available as operands of their nesting computation. For instance, the calculation of `~(grain-1)` is handled by these instructions:

```
5    iload_2          // Push grain
```

```
6    iconst_1         // Push int constant 1
7    isub             // Subtract; push result
8    iconst_m1        // Push int constant -1
9    ixor             // Do XOR; push result
```

First `grain-1` is calculated using the contents of local variable *2* and an immediate `int` value `1`. These operands are popped from the operand stack and their difference pushed back onto the operand stack. The difference is thus immediately available for use as one operand of the *ixor* instruction. (Recall that `~x ==`

`-1^x`.) Similarly, the result of the *ixor* instruction becomes an operand for the subsequent *iand* instruction.

The code for the entire method follows:

```
Method int align2grain(int,int)
0    iload_1
1    iload_2
2    iadd
3    iconst_1
4    isub
5    iload_2
6    iconst_1
7    isub
8    iconst_m1
9    ixor
10   iand
11   ireturn
```

3.4. Accessing the Run-Time Constant Pool

Many numeric constants, as well as objects, fields, and methods, are accessed via the run-time constant pool of the current class. Object access is considered later (§3.8). Data of types `int`, `long`, `float`, and `double`, as well as references to instances of class `String`, are managed using the *ldc*, *ldc_w*, and *ldc2_w* instructions.

The *ldc* and *ldc_w* instructions are used to access values in the run-time constant pool (including instances of class `String`) of types other than `double` and `long`. The *ldc_w* instruction is used in place of *ldc* only when there is a large number of run-time constant pool items and a larger index is needed to access an item. The *ldc2_w* instruction is used to access all values of types `double` and `long`; there is no non-wide variant.

Integral constants of types `byte`, `char`, or `short`, as well as small `int` values, may be compiled using the *bipush*, *sipush*, or *iconst_<i>* instructions (§3.2). Certain small floating-point constants may be compiled using the *fconst_<f>* and *dconst_<d>* instructions.

In all of these cases, compilation is straightforward. For instance, the constants for:

```
void useManyNumeric() {
    int i = 100;
    int j = 1000000;
    long l1 = 1;
    long l2 = 0xffffffff;
```

```
    double d = 2.2;
    ...do some calculations...
}
```

are set up as follows:

```
Method void useManyNumeric()
0   bipush 100   // Push small int constant
with bipush
2   istore_1
3   ldc #1       // Push large int constant
(1000000) with ldc
5   istore_2
6   lconst_1     // A tiny long value uses
small fast lconst_1
7   lstore_3
8   ldc2_w #6    // Push long 0xffffffff (that
is, an int -1)
        // Any long constant value can be
pushed with ldc2_w
11  lstore 5
13  ldc2_w #8    // Push double constant
2.200000
        // Uncommon double values are also
pushed with ldc2_w
16  dstore 7
...do those calculations...
```

3.5. More Control Examples

Compilation of `for` statements was shown in an earlier section (§3.2). Most of the Java programming language's other control constructs (`if-then-else`, `do`, `while`, `break`, and `continue`) are also compiled in the obvious ways. The compilation of `switch` statements is handled in a separate section (§3.10), as are the compilation of exceptions (§3.12) and the compilation of `finally` clauses (§3.13).

As a further example, a `while` loop is compiled in an obvious way, although the specific control transfer instructions made available by the Java Virtual Machine vary by data type. As usual, there is more support for data of type `int`, for example:

```
void whileInt() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

is compiled to:

```
Method void whileInt()
0   iconst_0
1   istore_1
2   goto 8
5   iinc 1 1
8   iload_1
9   bipush 100
11  if_icmplt 5
```

```
14   return
```

Note that the test of the `while` statement (implemented using the *if_icmplt* instruction) is at the bottom of the Java Virtual Machine code for the loop. (This was also the case in the `spin` examples earlier.) The test being at the bottom of the loop forces the use of a *goto* instruction to get to the test prior to the first iteration of the loop. If that test fails, and the loop body is never entered, this extra instruction is wasted. However, `while` loops are typically used when their body is expected to be run, often for many iterations. For subsequent iterations, putting the test at the bottom of the loop saves a Java Virtual Machine instruction each time around the loop: if the test were at the top of the loop, the loop body would need a trailing *goto* instruction to get back to the top.

Control constructs involving other data types are compiled in similar ways, but must use the instructions available for those data types. This leads to somewhat less efficient code because more Java Virtual Machine instructions are needed, for example:

```
void whileDouble() {
    double i = 0.0;
    while (i < 100.1) {
        i++;
    }
}
```

is compiled to:

```
Method void whileDouble()
0    dconst_0
1    dstore_1
2    goto 9
5    dload_1
6    dconst_1
7    dadd
8    dstore_1
9    dload_1
10   ldc2_w #4       // Push double constant
100.1
13   dcmpg           // To compare and branch we
have to use...
14   iflt 5          // ...two instructions
17   return
```

Each floating-point type has two comparison instructions: *fcmpl* and *fcmpg* for type `float`, and *dcmpl* and *dcmpg* for type `double`. The variants differ only in their treatment of NaN. NaN is unordered (§2.3.2), so all floating-point comparisons fail if either of their operands is NaN. The compiler chooses the variant of the comparison instruction for the appropriate type that produces the same result whether the comparison fails on non-NaN values or encounters a NaN. For instance:

```
int lessThan100(double d) {
    if (d < 100.0) {
        return
1;
    } else {
```

```
        return
-1;
    }
}
```

compiles to:

```
Method int lessThan100(double)
0    dload_1
1    ldc2_w #4       // Push double constant
100.0
4    dcmpg           // Push 1 if d is NaN or d >
100.0;
                     // push 0 if d == 100.0
5    ifge 10         // Branch on 0 or 1
8    iconst_1
9    ireturn
10   iconst_m1
11   ireturn
```

If `d` is not NaN and is less than `100.0`, the *dcmpg* instruction pushes an `int` *-1* onto the operand stack, and the *ifge* instruction does not branch. Whether `d` is greater than `100.0` or is NaN, the *dcmpg* instruction pushes an `int` *1* onto the operand stack, and the *ifge* branches. If `d` is equal to `100.0`, the *dcmpg* instruction pushes an `int` *0* onto the operand stack, and the *ifge* branches.

The *dcmpl* instruction achieves the same effect if the comparison is reversed:

```
int greaterThan100(double d) {
    if (d > 100.0) {
        return 1;
    } else {
        return -1;
    }
}
```

becomes:

```
Method int greaterThan100(double)
0   dload_1
1   ldc2_w #4       // Push double constant
100.0
4   dcmpl           // Push -1 if d is NaN or d
< 100.0;
                    // push 0 if d == 100.0
5   ifle 10         // Branch on 0 or -1
8   iconst_1
9   ireturn
10  iconst_m1
11  ireturn
```

Once again, whether the comparison fails on a non-NaN value or because it is passed a NaN, the *dcmpl* instruction pushes an `int` value onto the operand stack that causes the *ifle* to branch. If both of the *dcmp* instructions did not exist, one of the example methods would have had to do more work to detect NaN.

If *n* arguments are passed to an instance method, they are received, by convention, in the local variables numbered *1* through *n* of the frame created for the new method invocation. The arguments are received in the order they were passed. For example:

```
int addTwo(int i, int j) {
    return i + j;
}
```

compiles to:

```
Method int addTwo(int,int)
0   iload_1         // Push value of local
variable 1 (i)
1   iload_2         // Push value of local
variable 2 (j)
2   iadd            // Add; leave int result on
operand stack
3   ireturn         // Return int result
```

By convention, an instance method is passed a `reference` to its instance in local variable *0*. In the Java programming language the instance is accessible via the `this` keyword.

Class (`static`) methods do not have an instance, so for them this use of local variable *0* is unnecessary. A class method starts using local variables at index *0*. If the `addTwo` method were a class method, its arguments would be passed in a similar way to the first version:

```
static int addTwoStatic(int i, int j) {
    return i + j;
```

```
}
```

compiles to:

```
Method int addTwoStatic(int,int)
0    iload_0
1    iload_1
2    iadd
3    ireturn
```

The only difference is that the method arguments appear starting in local variable *0* rather than *1*.

The normal method invocation for a instance method dispatches on the run-time type of the object. (They are virtual, in C++ terms.) Such an invocation is implemented using the *invokevirtual* instruction, which takes as its argument an index to a run-time constant pool entry giving the internal form of the binary name of the class type of the object, the name of the method to invoke, and that method's descriptor (§4.3.3). To invoke the `addTwo` method, defined earlier as an instance method, we might write:

```
int add12and13() {
    return addTwo(12, 13);
}
```

This compiles to:

```
Method int add12and13()
0    aload_0                 // Push local variable
```

```
0 (this)
1    bipush 12              // Push int constant 12
3    bipush 13              // Push int constant 13
5    invokevirtual #4       // Method
Example.addtwo(II)I
8    ireturn                // Return int on top of
operand stack;
                            // it is the int result
of addTwo()
```

The invocation is set up by first pushing a `reference` to the current instance, `this`, on to the operand stack. The method invocation's arguments, `int` values `12` and `13`, are then pushed. When the frame for the `addTwo` method is created, the arguments passed to the method become the initial values of the new frame's local variables. That is, the `reference` for `this` and the two arguments, pushed onto the operand stack by the invoker, will become the initial values of local variables *0*, *1*, and *2* of the invoked method.

Finally, `addTwo` is invoked. When it returns, its `int` return value is pushed onto the operand stack of the frame of the invoker, the `add12and13` method. The return value is thus put in place to be immediately returned to the invoker of `add12and13`.

The return from `add12and13` is handled by the *ireturn* instruction of `add12and13`. The *ireturn* instruction takes the `int` value returned by `addTwo`, on the operand stack of the current frame, and pushes it onto the operand stack of the frame of the invoker. It then returns control to the invoker, making the invoker's frame current. The Java Virtual Machine provides distinct return instructions for many of its numeric and `reference` data types, as well as a *return* instruction for methods with no return value. The same set of return instructions is used for all varieties of method invocations.

The operand of the *invokevirtual* instruction (in the example, the run-time constant pool index *#4*) is not the offset of the method in the class instance. The compiler does not know the internal layout of a class instance. Instead, it generates symbolic references to the methods of an instance, which are stored in the run-time constant pool. Those run-time constant pool items are resolved at run-time to determine the actual method location. The same is true for all other Java Virtual Machine instructions that access class instances.

Invoking `addTwoStatic`, a class (`static`) variant of `addTwo`, is similar, as shown:

```
int add12and13() {
    return addTwoStatic(12, 13);
}
```

although a different Java Virtual Machine method invocation instruction is used:

```
Method int add12and13()
0   bipush 12
2   bipush 13
4   invokestatic #3     // Method
Example.addTwoStatic(II)I
7   ireturn
```

Compiling an invocation of a class (`static`) method is very much like compiling an invocation of an instance method, except this is not passed by the invoker. The method arguments will thus be received beginning with local variable *0* (§3.6). The *invokestatic* instruction is always used to invoke class methods.

The *invokespecial* instruction must be used to invoke instance initialization methods (§3.8). It is also used when invoking methods in the superclass (`super`) and when invoking `private` methods. For instance, given classes `Near` and `Far` declared as:

```
class Near {
    int it;
    public int getItNear() {
        return getIt();
    }
    private int getIt() {
        return it;
    }
}

class Far extends Near {
    int getItFar() {
        return super.getItNear();
    }
}
```

the method `Near.getItNear` (which invokes a `private` method) becomes:

```
Method int getItNear()
0    aload_0
1    invokespecial #5     // Method Near.getIt()I
4    ireturn
```

The method `Far.getItFar` (which invokes a superclass method) becomes:

```
Method int getItFar()
0    aload_0
1    invokespecial #4     // Method
Near.getItNear()I
4    ireturn
```

Note that methods called using the *invokespecial* instruction always pass `this` to the invoked method as its first argument. As usual, it is received in local variable *0*.

To invoke the target of a method handle, a compiler must form a method descriptor that records the actual argument and return types. A compiler may not perform method invocation conversions on the arguments; instead, it must push them on the stack according to their own unconverted types. The compiler arranges for a `reference` to the method handle object to be pushed on the stack before the arguments, as usual. The compiler emits an *invokevirtual* instruction that references a descriptor which describes the argument and return types. By special arrangement with method resolution (§5.4.3.3), an *invokevirtual* instruction which invokes the `invokeExact` or `invoke` methods of `java.lang.invoke.MethodHandle` will always link, provided the method descriptor is syntactically well-formed and the types named in the descriptor can be resolved.

3.8. Working with Class Instances

Java Virtual Machine class instances are created using the Java Virtual Machine's *new* instruction. Recall that at the level of the Java Virtual Machine, a constructor appears as a method with the compiler-supplied name `<init>`. This specially named method is known as the instance initialization method (§2.9). Multiple instance initialization methods, corresponding to multiple constructors, may exist for a given class. Once the class instance has been created and its instance variables, including those of the class and all of its superclasses, have been initialized to their default values, an instance initialization method of the new class instance is invoked. For example:

```
Object create() {
    return new Object();
}
```

compiles to:

```
Method java.lang.Object create()
0   new #1              // Class
java.lang.Object
3   dup
4   invokespecial #4    // Method
java.lang.Object.<init>()V
7   areturn
```

Class instances are passed and returned (as `reference` types) very much like numeric values, although type `reference` has its own complement of instructions, for example:

```
int i;                                    // An
instance variable
MyObj example() {
    MyObj o = new MyObj();
    return silly(o);
}
MyObj silly(MyObj o) {
    if (o != null) {
        return o;
    } else {
        return o;
    }
}
```

becomes:

```
Method MyObj example()
0   new #2              // Class MyObj
3   dup
4   invokespecial #5    // Method MyObj.
<init>()V
7   astore_1
8   aload_0
9   aload_1
10  invokevirtual #4    // Method
```

```
Example.silly(LMyObj;)LMyObj;
13   areturn

Method MyObj silly(MyObj)
0    aload_1
1    ifnull 6
4    aload_1
5    areturn
6    aload_1
7    areturn
```

The fields of a class instance (instance variables) are accessed using the *getfield* and *putfield* instructions. If `i` is an instance variable of type `int`, the methods `setIt` and `getIt`, defined as:

```
void setIt(int value) {
    i = value;
}
int getIt() {
    return i;
}
```

become:

```
Method void setIt(int)
0    aload_0
1    iload_1
2    putfield #4    // Field Example.i I
5    return
```

```
Method int getIt()
0    aload_0
1    getfield #4    // Field Example.i I
4    ireturn
```

As with the operands of method invocation instructions, the operands of the *putfield* and *getfield* instructions (the run-time constant pool index *#4*) are not the offsets of the fields in the class instance. The compiler generates symbolic references to the fields of an instance, which are stored in the run-time constant pool. Those run-time constant pool items are resolved at run-time to determine the location of the field within the referenced object.

Java Virtual Machine arrays are also objects. Arrays are created and manipulated using a distinct set of instructions. The *newarray* instruction is used to create an array of a numeric type. The code:

```
void createBuffer() {
    int buffer[];
    int bufsz = 100;
    int value = 12;
    buffer = new int[bufsz];
    buffer[10] = value;
    value = buffer[11];
}
```

might be compiled to:

```
Method void createBuffer()
0   bipush 100     // Push int constant 100
(bufsz)
2   istore_2       // Store bufsz in local
variable 2
3   bipush 12      // Push int constant 12
(value)
5   istore_3       // Store value in local
variable 3
6   iload_2        // Push bufsz...
7   newarray int   // ...and create new int
array of that length
9   astore_1       // Store new array in buffer
10  aload_1        // Push buffer
11  bipush 10      // Push int constant 10
13  iload_3        // Push value
14  iastore        // Store value at buffer[10]
15  aload_1        // Push buffer
16  bipush 11      // Push int constant 11
18  iaload         // Push value at
buffer[11]...
19  istore_3       // ...and store it in value
20  return
```

The *anewarray* instruction is used to create a one-dimensional array of object references, for example:

```
void createThreadArray() {
    Thread threads[];
    int count = 10;
```

```
    threads = new Thread[count];
    threads[0] = new Thread();
}
```

becomes:

```
Method void createThreadArray()
0   bipush 10             // Push int constant 10
2   istore_2              // Initialize count to
that
3   iload_2               // Push count, used by
anewarray
4   anewarray class #1    // Create new array of
class Thread
7   astore_1              // Store new array in
threads
8   aload_1               // Push value of
threads
9   iconst_0              // Push int constant 0
10  new #1                // Create instance of
class Thread
13  dup                   // Make duplicate
reference...
14  invokespecial #5      // ...for Thread's
constructor
                          // Method
java.lang.Thread.<init>()V
17  aastore               // Store new Thread in
array at 0
```

```
18  return
```

The *anewarray* instruction can also be used to create the first dimension of a multidimensional array. Alternatively, the *multianewarray* instruction can be used to create several dimensions at once. For example, the three-dimensional array:

```
int[][][] create3DArray() {
    int grid[][][];
    grid = new int[10][5][];
    return grid;
}
```

is created by:

```
Method int create3DArray()[][][]
0   bipush 10                   // Push int 10
(dimension one)
2   iconst_5                    // Push int 5
(dimension two)
3   multianewarray #1 dim #2 // Class [[[I, a
three-dimensional
                                // int array; only
create the
                                // first two
dimensions
7   astore_1                    // Store new
array...
8   aload_1                     // ...then prepare
to return it
```

```
9    areturn
```

The first operand of the *multianewarray* instruction is the run-time constant pool index to the array class type to be created. The second is the number of dimensions of that array type to actually create. The *multianewarray* instruction can be used to create all the dimensions of the type, as the code for `create3DArray` shows. Note that the multidimensional array is just an object and so is loaded and returned by an *aload_1* and *areturn* instruction, respectively. For information about array class names, see §4.4.1.

All arrays have associated lengths, which are accessed via the *arraylength* instruction.

Compilation of `switch` statements uses the *tableswitch* and *lookupswitch* instructions. The *tableswitch* instruction is used when the cases of the `switch` can be efficiently represented as indices into a table of target offsets. The `default` target of the `switch` is used if the value of the expression of the `switch` falls outside the range of valid indices. For instance:

```
int chooseNear(int i) {
    switch (i) {
        case 0:  return  0;
        case 1:  return  1;
        case 2:  return  2;
        default: return -1;
    }
}
```

compiles to:

```
Method int chooseNear(int)
0   iload_1              // Push local variable
1 (argument i)
1   tableswitch 0 to 2: // Valid indices are 0
through 2
      0: 28              // If i is 0, continue
at 28
      1: 30              // If i is 1, continue
at 30
      2: 32              // If i is 2, continue
at 32
      default:34         // Otherwise, continue
at 34
28  iconst_0             // i was 0; push int
constant 0...
29  ireturn              // ...and return it
30  iconst_1             // i was 1; push int
constant 1...
31  ireturn              // ...and return it
32  iconst_2             // i was 2; push int
constant 2...
33  ireturn              // ...and return it
34  iconst_m1            // otherwise push int
constant -1...
35  ireturn              // ...and return it
```

The Java Virtual Machine's *tableswitch* and *lookupswitch* instructions operate only on `int` data. Because operations on `byte`, `char`, or `short` values are internally promoted to `int`, a `switch` whose expression evaluates to one of those types is compiled as though it evaluated to type `int`. If the `chooseNear` method had been written using type `short`, the same Java Virtual Machine instructions would have been generated as when using type `int`. Other numeric types must be narrowed to type `int` for use in a `switch`.

Where the cases of the `switch` are sparse, the table representation of the *tableswitch* instruction becomes inefficient in terms of space. The *lookupswitch* instruction may be used instead. The *lookupswitch* instruction pairs `int` keys (the values of the `case` labels) with target offsets in a table. When a *lookupswitch* instruction is executed, the value of the expression of the `switch` is compared against the keys in the table. If one of the keys matches the value of the expression, execution continues at the associated target offset. If no key matches, execution continues at the `default` target. For instance, the compiled code for:

```
int chooseFar(int i) {
    switch (i) {
        case -100: return -1;
        case 0:    return  0;
        case 100:  return  1;
        default:   return -1;
    }
}
```

looks just like the code for `chooseNear`, except for the *lookupswitch* instruction:

```
Method int chooseFar(int)
0   iload_1
1   lookupswitch 3:
         -100: 36
            0: 38
          100: 40
      default: 42
36  iconst_m1
37  ireturn
38  iconst_0
39  ireturn
40  iconst_1
41  ireturn
42  iconst_m1
43  ireturn
```

The Java Virtual Machine specifies that the table of the *lookupswitch* instruction must be sorted by key so that implementations may use searches more efficient than a linear scan. Even so, the *lookupswitch* instruction must search its keys for a match rather than simply perform a bounds check and index into a table like *tableswitch*. Thus, a *tableswitch* instruction is probably more efficient than a *lookupswitch* where space considerations permit a choice.

3.11. Operations on the Operand Stack

The Java Virtual Machine has a large complement of instructions that manipulate the contents of the operand stack as untyped values. These are useful because of the Java Virtual Machine's reliance on deft manipulation of its operand stack. For instance:

```
public long nextIndex() {
    return index++;
}

private long index = 0;
```

is compiled to:

```
Method long nextIndex()
0    aload_0          // Push this
1    dup              // Make a copy of it
2    getfield #4      // One of the copies of this
is consumed
                      // pushing long field index,
                      // above the original this
5    dup2_x1          // The long on top of the
operand stack is
                      // inserted into the operand
stack below the
                      // original this
6    lconst_1         // Push long constant 1
7    ladd             // The index value is
incremented...
8    putfield #4      // ...and the result stored
in the field
```

```
11  lreturn        // The original value of
index is on top of
                   // the operand stack, ready
to be returned
```

Note that the Java Virtual Machine never allows its operand stack manipulation instructions to modify or break up individual values on the operand stack.

3.12. Throwing and Handling Exceptions

Exceptions are thrown from programs using the `throw` keyword. Its compilation is simple:

```
void cantBeZero(int i) throws TestExc {
    if (i == 0) {
        throw new TestExc();
    }
}
```

becomes:

```
Method void cantBeZero(int)
0   iload_1              // Push argument 1 (i)
1   ifne 12              // If i==0, allocate
instance and throw
4   new #1               // Create instance of
TestExc
7   dup                  // One reference goes
to its constructor
8   invokespecial #7     // Method TestExc.
```

```
<init>()V
11   athrow                  // Second reference is
thrown
12   return                  // Never get here if we
threw TestExc
```

Compilation of `try-catch` constructs is straightforward. For example:

```
void catchOne() {
    try {
        tryItOut();
    } catch (TestExc e) {
        handleExc(e);
    }
}
```

is compiled as:

```
Method void catchOne()
0    aload_0                 // Beginning of try
block
1    invokevirtual #6        // Method
Example.tryItOut()V
4    return                  // End of try block;
normal return
5    astore_1                // Store thrown value
in local var 1
6    aload_0                 // Push this
7    aload_1                 // Push thrown value
```

```
8   invokevirtual #5    // Invoke handler
method:
                        //
Example.handleExc(LTestExc;)V
11  return              // Return after
handling TestExc
Exception table:
From    To      Target      Type
0       4       5           Class TestExc
```

Looking more closely, the `try` block is compiled just as it would be if the `try` were not present:

```
Method void catchOne()
0   aload_0             // Beginning of try
block
1   invokevirtual #6    // Method
Example.tryItOut()V
4   return              // End of try block;
normal return
```

If no exception is thrown during the execution of the `try` block, it behaves as though the `try` were not there: `tryItOut` is invoked and `catchOne` returns.

Following the `try` block is the Java Virtual Machine code that implements the single `catch` clause:

```
5   astore_1            // Store thrown value
in local var 1
6   aload_0             // Push this
7   aload_1             // Push thrown value
```

```
8   invokevirtual #5    // Invoke handler
method:
                        //
Example.handleExc(LTestExc;)V
11  return              // Return after
handling TestExc
Exception table:
From    To      Target      Type
0       4       5           Class TestExc
```

The invocation of `handleExc`, the contents of the `catch` clause, is also compiled like a normal method invocation. However, the presence of a `catch` clause causes the compiler to generate an exception table entry (§2.10, §4.7.3). The exception table for the `catchOne` method has one entry corresponding to the one argument (an instance of class `TestExc`) that the `catch` clause of `catchOne` can handle. If some value that is an instance of `TestExc` is thrown during execution of the instructions between indices *0* and *4* in `catchOne`, control is transferred to the Java Virtual Machine code at index *5*, which implements the block of the `catch` clause. If the value that is thrown is not an instance of `TestExc`, the `catch` clause of `catchOne` cannot handle it. Instead, the value is rethrown to the invoker of `catchOne`.

A `try` may have multiple `catch` clauses:

```
void catchTwo() {
    try {
        tryItOut();
    } catch (TestExc1 e) {
```

```
        handleExc(e);
    } catch (TestExc2 e) {
        handleExc(e);
    }
}
```

Multiple `catch` clauses of a given `try` statement are compiled by simply appending the Java Virtual Machine code for each `catch` clause one after the other and adding entries to the exception table, as shown:

```
Method void catchTwo()
0    aload_0                // Begin try block
1    invokevirtual #5       // Method
Example.tryItOut()V
4    return                 // End of try block;
normal return
5    astore_1               // Beginning of handler
for TestExc1;
                            // Store thrown value
in local var 1
6    aload_0                // Push this
7    aload_1                // Push thrown value
8    invokevirtual #7       // Invoke handler
method:
                            //
Example.handleExc(LTestExc1;)V
11   return                 // Return after
handling TestExc1
```

```
12   astore_1              // Beginning of handler
for TestExc2;
                           // Store thrown value
in local var 1
13   aload_0               // Push this
14   aload_1               // Push thrown value
15   invokevirtual #7      // Invoke handler
method:
                           //
Example.handleExc(LTestExc2;)V
18   return                // Return after
handling TestExc2
Exception table:
From    To      Target      Type
0       4       5           Class TestExc1
0       4       12          Class TestExc2
```

If during the execution of the `try` clause (between indices *0* and *4*) a value is thrown that matches the parameter of one or more of the `catch` clauses (the value is an instance of one or more of the parameters), the first (innermost) such `catch` clause is selected. Control is transferred to the Java Virtual Machine code for the block of that `catch` clause. If the value thrown does not match the parameter of any of the `catch` clauses of `catchTwo`, the Java Virtual Machine rethrows the value without invoking code in any `catch` clause of `catchTwo`.

Nested `try-catch` statements are compiled very much like a `try` statement with multiple `catch` clauses:

```
void nestedCatch() {
```

```
    try {
        try {
            tryItOut();
        } catch (TestExc1 e) {
            handleExc1(e);
        }
    } catch (TestExc2 e) {
        handleExc2(e);
    }
}
```

becomes:

```
Method void nestedCatch()
0   aload_0                 // Begin try block
1   invokevirtual #8        // Method
Example.tryItOut()V
4   return                  // End of try block;
normal return
5   astore_1                // Beginning of handler
for TestExc1;
                            // Store thrown value
in local var 1
6   aload_0                 // Push this
7   aload_1                 // Push thrown value
8   invokevirtual #7        // Invoke handler
method:
                            //
Example.handleExc1(LTestExc1;)V
```

```
11   return                   // Return after
handling TestExc1
12   astore_1                 // Beginning of handler
for TestExc2;
                              // Store thrown value
in local var 1
13   aload_0                  // Push this
14   aload_1                  // Push thrown value
15   invokevirtual #6         // Invoke handler
method:
                              //
Example.handleExc2(LTestExc2;)V
18   return                   // Return after
handling TestExc2
Exception table:
From     To       Target       Type
0        4        5            Class TestExc1
0        12       12           Class TestExc2
```

The nesting of `catch` clauses is represented only in the exception table. The Java Virtual Machine does not enforce nesting of or any ordering of the exception table entries (§2.10). However, because `try-catch` constructs are structured, a compiler can always order the entries of the exception handler table such that, for any thrown exception and any program counter value in that method, the first exception handler that matches the thrown exception corresponds to the innermost matching `catch` clause.

For instance, if the invocation of `tryItOut` (at index *1*) threw an instance of `TestExc1`, it would be handled by the `catch` clause that invokes `handleExc1`. This is so even though the exception occurs within the bounds of the outer `catch` clause (catching `TestExc2`) and even though that outer `catch` clause might otherwise have been able to handle the thrown value.

As a subtle point, note that the range of a `catch` clause is inclusive on the "from" end and exclusive on the "to" end (§4.7.3). Thus, the exception table entry for the `catch` clause catching `TestExc1` does not cover the *return* instruction at offset *4*. However, the exception table entry for the `catch` clause catching `TestExc2` does cover the *return* instruction at offset *11*. Return instructions within nested `catch` clauses are included in the range of instructions covered by nesting `catch` clauses.

(This section assumes a compiler generates `class` files with version number 50.0 or below, so that the *jsr* instruction may be used. See also §4.10.2.5.)

Compilation of a `try-finally` statement is similar to that of `try-catch`. Prior to transferring control outside the `try` statement, whether that transfer is normal or abrupt, because an exception has been thrown, the `finally` clause must first be executed. For this simple example:

```
void tryFinally() {
    try {
        tryItOut();
    } finally {
        wrapItUp();
```

```
    }
}
```

the compiled code is:

```
Method void tryFinally()
0   aload_0                 // Beginning of try
block
1   invokevirtual #6        // Method
Example.tryItOut()V
4   jsr 14                  // Call finally block
7   return                  // End of try block
8   astore_1                // Beginning of handler
for any throw
9   jsr 14                  // Call finally block
12  aload_1                 // Push thrown value
13  athrow                  // ...and rethrow value
to the invoker
14  astore_2                // Beginning of finally
block
15  aload_0                 // Push this
16  invokevirtual #5        // Method
Example.wrapItUp()V
19  ret 2                   // Return from finally
block
Exception table:
From    To      Target      Type
0       4       8           any
```

There are four ways for control to pass outside of the `try` statement: by falling through the bottom of that block, by returning, by executing a `break` or `continue` statement, or by raising an exception. If `tryItOut` returns without raising an exception, control is transferred to the `finally` block using a *jsr* instruction. The *jsr 14* instruction at index *4* makes a "subroutine call" to the code for the `finally` block at index *14* (the `finally` block is compiled as an embedded subroutine). When the `finally` block completes, the *ret 2* instruction returns control to the instruction following the *jsr* instruction at index *4*.

In more detail, the subroutine call works as follows: The *jsr* instruction pushes the address of the following instruction (*return* at index *7*) onto the operand stack before jumping. The *astore_2* instruction that is the jump target stores the address on the operand stack into local variable *2*. The code for the `finally` block (in this case the *aload_0* and *invokevirtual* instructions) is run. Assuming execution of that code completes normally, the *ret* instruction retrieves the address from local variable *2* and resumes execution at that address. The *return* instruction is executed, and `tryFinally` returns normally.

A `try` statement with a `finally` clause is compiled to have a special exception handler, one that can handle any exception thrown within the `try` statement. If `tryItOut` throws an exception, the exception table for `tryFinally` is searched for an appropriate exception handler. The special handler is found, causing execution to continue at index *8*. The *astore_1* instruction at index *8* stores the thrown value into local variable *1*. The following *jsr* instruction does a subroutine call to the code for the `finally` block. Assuming that code returns normally, the *aload_1* instruction at index *12* pushes the thrown value back onto the operand stack, and the following *athrow* instruction rethrows the value.

Compiling a `try` statement with both a `catch` clause and a `finally` clause is more complex:

```
void tryCatchFinally() {
    try {
        tryItOut();
    } catch (TestExc e) {
        handleExc(e);
    } finally {
        wrapItUp();
    }
}
```

becomes:

```
Method void tryCatchFinally()
0   aload_0               // Beginning of try
block
1   invokevirtual #4      // Method
Example.tryItOut()V
4   goto 16               // Jump to finally
block
7   astore_3              // Beginning of handler
for TestExc;
                          // Store thrown value
```

```
in local var 3
8   aload_0              // Push this
9   aload_3              // Push thrown value
10  invokevirtual #6     // Invoke handler
method:
                         //
Example.handleExc(LTestExc;)V
13  goto 16              // This goto is
unnecessary, but was
                         // generated by javac
in JDK 1.0.2
16  jsr 26               // Call finally block
19  return               // Return after
handling TestExc
20  astore_1             // Beginning of handler
for exceptions
                         // other than TestExc,
or exceptions
                         // thrown while
handling TestExc
21  jsr 26               // Call finally block
24  aload_1              // Push thrown value...
25  athrow               // ...and rethrow value
to the invoker
26  astore_2             // Beginning of finally
block
27  aload_0              // Push this
28  invokevirtual #5     // Method
Example.wrapItUp()V
```

```
31  ret 2                 // Return from finally
block
Exception table:
From    To      Target      Type
0       4       7           Class TestExc
0       16      20          any
```

If the `try` statement completes normally, the *goto* instruction at index *4* jumps to the subroutine call for the `finally` block at index *16*. The `finally` block at index *26* is executed, control returns to the *return* instruction at index *19*, and `tryCatchFinally` returns normally.

If `tryItOut` throws an instance of `TestExc`, the first (innermost) applicable exception handler in the exception table is chosen to handle the exception. The code for that exception handler, beginning at index *7*, passes the thrown value to `handleExc` and on its return makes the same subroutine call to the `finally` block at index *26* as in the normal case. If an exception is not thrown by `handleExc`, `tryCatchFinally` returns normally.

If `tryItOut` throws a value that is not an instance of `TestExc` or if `handleExc` itself throws an exception, the condition is handled by the second entry in the exception table, which handles any value thrown between indices *0* and *16*. That exception handler transfers control to index *20*, where the thrown value is first stored in local variable *1*. The code for the `finally` block at index *26* is called as a subroutine. If it returns, the thrown value is retrieved from local variable *1* and rethrown using the *athrow* instruction. If a new value is thrown during execution of the `finally` clause, the `finally` clause aborts, and `tryCatchFinally` returns abruptly, throwing the new value to its invoker.

Synchronization in the Java Virtual Machine is implemented by monitor entry and exit, either explicitly (by use of the *monitorenter* and *monitorexit* instructions) or implicitly (by the method invocation and return instructions).

For code written in the Java programming language, perhaps the most common form of synchronization is the `synchronized` method. A `synchronized` method is not normally implemented using *monitorenter* and *monitorexit*. Rather, it is simply distinguished in the run-time constant pool by the `ACC_SYNCHRONIZED` flag, which is checked by the method invocation instructions (§2.11.10).

The *monitorenter* and *monitorexit* instructions enable the compilation of `synchronized` statements. For example:

```
void onlyMe(Foo f) {
    synchronized(f) {
        doSomething();
    }
}
```

is compiled to:

```
Method void onlyMe(Foo)
0   aload_1             // Push f
1   dup                 // Duplicate it on the
stack
```

```
2   astore_2            // Store duplicate in
local variable 2
3   monitorenter        // Enter the monitor
associated with f
4   aload_0             // Holding the monitor,
pass this and...
5   invokevirtual #5    // ...call
Example.doSomething()V
8   aload_2             // Push local variable
2 (f)
9   monitorexit         // Exit the monitor
associated with f
10  goto 18             // Complete the method
normally
13  astore_3            // In case of any
throw, end up here
14  aload_2             // Push local variable
2 (f)
15  monitorexit         // Be sure to exit the
monitor!
16  aload_3             // Push thrown value...
17  athrow              // ...and rethrow value
to the invoker
18  return              // Return in the normal
case
Exception table:
From    To      Target      Type
4       10      13          any
13      16      13          any
```

The compiler ensures that at any method invocation completion, a *monitorexit* instruction will have been executed for each *monitorenter* instruction executed since the method invocation. This is the case whether the method invocation completes normally (§2.6.4) or abruptly (§2.6.5). To enforce proper pairing of *monitorenter* and *monitorexit* instructions on abrupt method invocation completion, the compiler generates exception handlers (§2.10) that will match any exception and whose associated code executes the necessary *monitorexit* instructions.

The representation of annotations in `class` files is described in §4.7.16-§4.7.22. These sections make it clear how to represent annotations on declarations of classes, interfaces, fields, methods, method parameters, and type parameters, as well as annotations on types used in those declarations. Annotations on package declarations require additional rules, given here.

When the compiler encounters an annotated package declaration that must be made available at run time, it emits a `class` file with the following properties:

- The `class` file represents an interface, that is, the `ACC_INTERFACE` and `ACC_ABSTRACT` flags of the `ClassFile` structure are set (§4.1).
- If the `class` file version number is less than 50.0, then the `ACC_SYNTHETIC` flag is unset; if the `class` file version number is 50.0 or above, then the `ACC_SYNTHETIC` flag is set.
- The interface has package access (JLS §6.6.1).
- The interface's name is the internal form (§4.2.1) of *`package-`*

`name``.package-info`.

- The interface has no superinterfaces.
- The interface's only members are those implied by *The Java Language Specification, Java SE 8 Edition* (JLS §9.2).
- The annotations on the package declaration are stored as `RuntimeVisibleAnnotations` and `RuntimeInvisibleAnnotations` attributes in the `attributes` table of the `ClassFile` structure.