JVM and Code Generation For Java

Wuu Yang

wuuyang@cs.nctu.edu.tw

National Chiao-Tung University, Taiwan, R.O.C.

first draft: May 7, 2004

current version: December 28, 2011

This set of notes is adapted from JVM Spec, Chap 7, Compiling for the Java Virtual Machine.

Contents

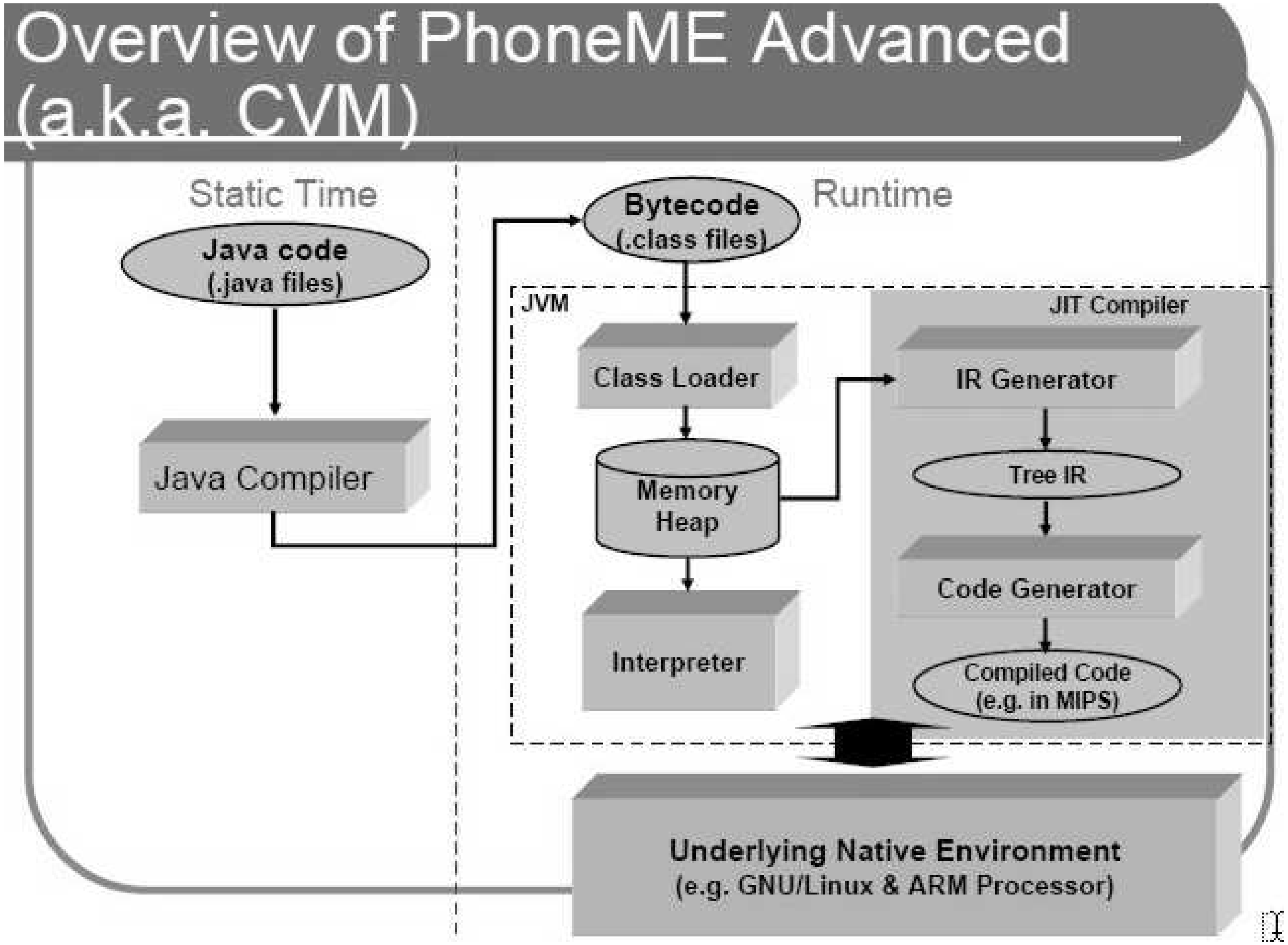

Figure 1: Java architecture

1 Java Virtual Machines

JVM is an abstract specification of a virtual machine intended for Java programs. See The Java Virtual Machine Specification, 2nd ed. by Tim Lindholm and Frank Yellin. There are many implementations of JVM on various hardware and software platforms, some of which are free.

When a Java application starts, a run-time instance of JVM is created. The instance is destroyed when the application completes. Each Java application runs inside its own JVM instance.

JVM is a stack-based machine. All arithmetic operations are done through the *operand stack*. Byte code is essentially a postfix form of the program.

JVM also does many security checks before running a procedure.

JVM makes use of dynamic loading and linking. It loads classes during run time only when necessary.

JVM provides a *native method interface* for invoking native method libraries.

Run-time data structures of a JVM instance:

- a method area: shared by all threads in the JVM
- a heap: shared by all threads in the JVM
- a Java stack per thread
- a program counter per thread
- native method stacks

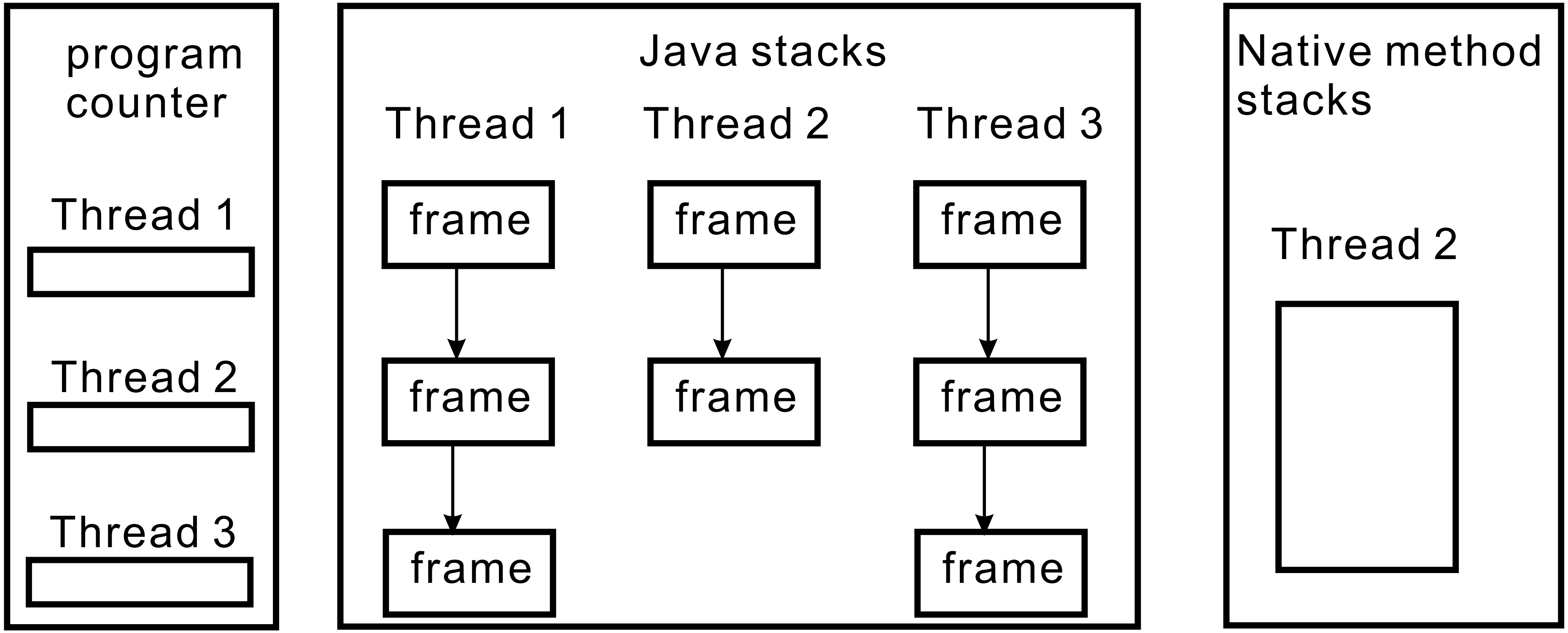

Figure 2: JVM run-time data structure

Data types in JVM

- primitive type
  1. numeric type: float, double, byte, short, int, long, char
  2. boolean type
  3. return address type
- reference type
  1. class type
  2. interface type
  3. array type

class loader

The class loader locates a class when reference, possibly from a remote machine, loads that class, and initializes that class.

- loading: find and import the class
- linking
  1. verification: ensure the imported class file is correct, not tampered.
  2. preparation: allocate memory for class variables and set up their default values.
  3. resolution: translate symbolic reference to direct references.
- initialization: initialize class variables to their starting values.

Every JVM includes a *bootstrap class loader*. The bootstrap class loader in Sun Java 2 SDK looks for system classes only in the installation directory. It will not consult the `classpath` environment variable.

On the other hand, the `system class loader` will consult `classpath`. The system class loader is a user-defined class loader that is created when the JVM starts up.

Method area

Methods from the loaded classes are stored in the method area.

Class (static) variables are also stored in the method area.

All threads share the same method area.

Constant pool

The constant pool is the symbol table for a class file. It contains literals (strings, integers, floating-point constants) and symbolic references to types, methods, and fields in the class.

Constant pool is used during dynamic linking.

All constant pools are allocated in the method area.

Heap

All instances of a class or an array are allocated from the heap, which is shared among all the threads for the same application.

There is a separate heap for each individual running application.

JVM provides an instruction that allocates memory for new objects on the heap but does not provide instructions to free the memory. Instead JVM provides a garbage collector for reclaiming useless memory.

Garbage collection

Pointers exist in the Java stacks, the heap, the method area, and the native method stacks.

Arrays

In Java, arrays are objects with a class. They are stored in the heap.

Multi-dimensional arrays are represented as arrays of arrays. (See L6-jvm.pdf, p. 15.)

The program counter occupies one word. It can be a native pointer or an offset from the beginning of a method's bytecode.

Java stacks

A Java stack consists of many frames. A frame includes local variables, operand stack, and frame data.

The size of a frame (in particular, the size of the operand stack) for a method is calculated by the compiler. This creates a constraint on methods.

The local variables in a frame are addressed by their indices. A method's parameters are stored in the local variable section in the frame. In the frame for an instance method, the first word is a hidden pointer to the object itself. There is not such a pointer in the frame for a class method.

JVM is essentially a stack machine. Many instructions make use of the operand stack, which is allocated in the stack frame for a method.

The frame data section includes data for constant pool resolution, normal method returns, and exception handling.

Many instructions reference entries in the constant pool, such as pushing a constant value from the constant pool to the operand stack, referencing classes and arrays when creating new instances, accessing fields of a class, or invoking methods in the class, and determining the class or interface of a particular objects, etc.

Native methods also need a stack. This native stack is separated from the Java stack, which is intended for the Java methods.

Note that Java methods may invoke native methods. Similarly, native methods may also invoke Java methods as well as other native methods.

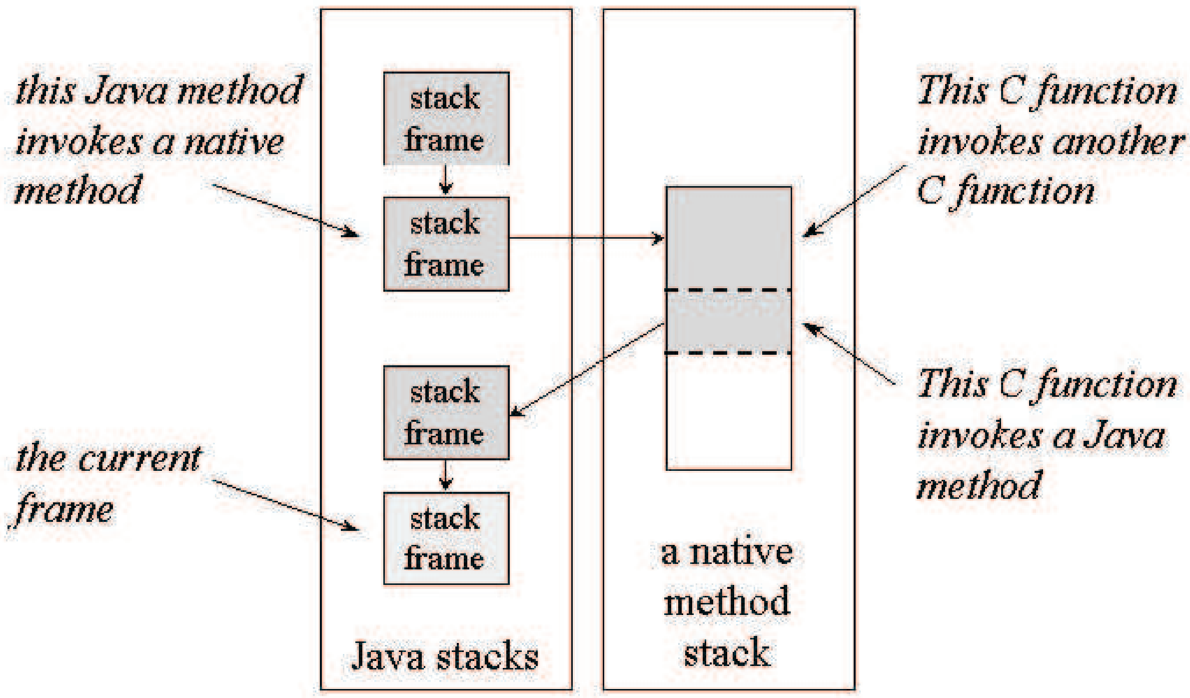

Figure 3: Java stacks vs. native stack.

A JVM may use invisible threads for other tasks, such as garbage collection.

Java instruction format:

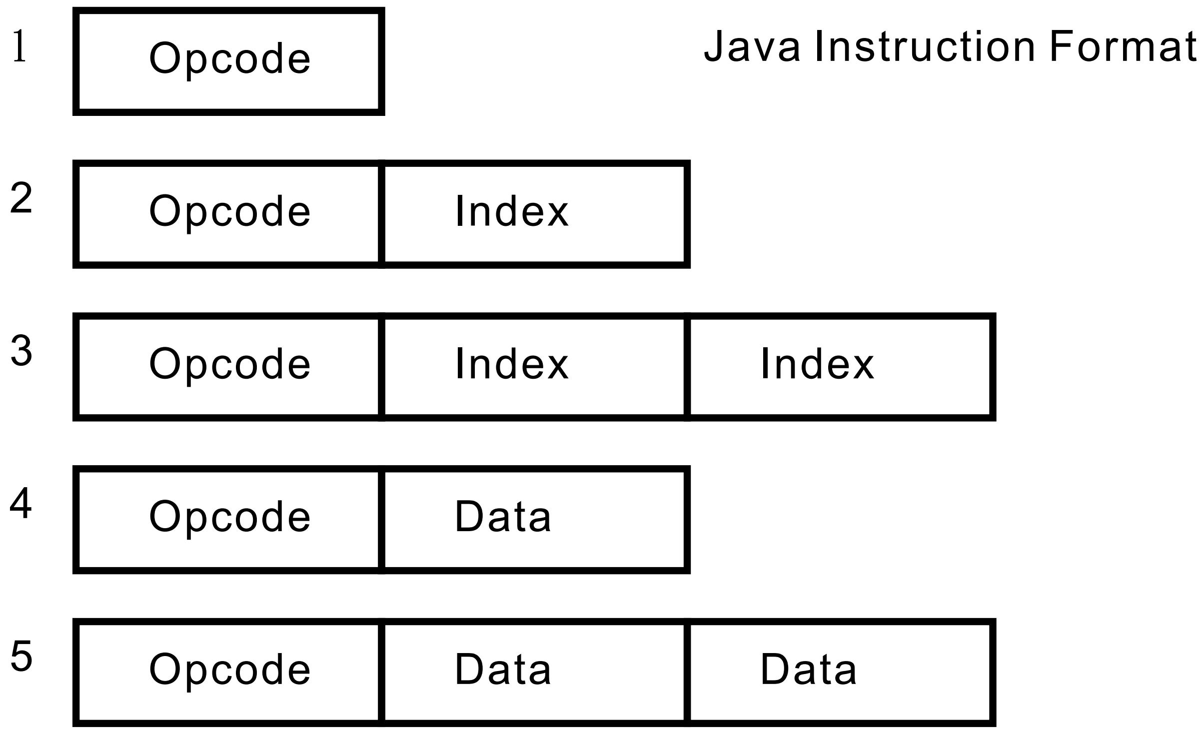

Figure 4: 6 Java instruction formats.

Java instruction set: see the manual.

- load and store instructions
- arithmetic instructions
- type conversion instructions
- object creation and manipulation instructions
- operand stack management instructions
- control transfer instructions
- method invocation instructions
- method return instructions

2 Bytecode

Headers: Magic number, version
Constant Pool: all external references; numerical and string constants; type signatures
Class Data: access flags, name, superclass, interfaces
Fields: indexes into constant pool for name and type; access flags; (optional) constant value
Methods: indexes into constant pool for name and type; access flags; code; exception handlers and thrown exceptions, debug information
Attributes: source file

Figure 1. Structure of a Java classfile

(slide 3) Operands for a bytecode instruction may be taken from

- the operand stack
- immediate operand (part of the instruction)
- implicit operand (in the instruction, e.g., `iload1`)
- local variable (from the run-time stack)
- the constant pool

(slide 18) Need to do bytecode verification to check the depth of the operand stack. Different paths from `<start>` to `<stop>` use the same amount of the operand stack.

Constants, Local Variables, and Control Structures

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

```
Method void spin()
 0  iconst_0    // Push constant 0
 1  istore_1    // local var 1 is i
 2  goto 8      // loop starts here
 5  iinc 1 1    // i++
 8  iload_1     // Push local var 1
 9  bipush 100 // Push constant 100
11  if_icmplt 5 // Compare and loop
14  return      // Return void
```

A new frame is created for every method invocation. Local variables are located in the current frame.
A frame also has an operand stack.

We use `iconst_0` (with an implicit operand, 1 byte) instead of `bipush 0` (2 bytes).

The local variable `i` is stored in the current frame. We also use `iload_1` and `istore_1` to load from local variable 1 and store the top of the stack to local variable 1, respectively.

Note `iinc` increments a local variable, rather than the value on top of the operand stack. This saves a `push` and a `pop` operations. `iinc` is intended for loops.

For `int` type, we may use `if_icmplt` for comparison and jumping. For `double` type, we have to use two instructions: `dcmpg` and `iflt`.

Java bytecode provides more support for `int` type than other types.

Spins on Doubles

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

```
Method void dspin()
 0  dconst_0 // Push constant 0.0
 1  dstore_1
 // store in local var 1 and 2
 2  goto 9
 5  dload_1 //Push var 1 and 2
 6  dconst_1 // Push constant 1.0
 7  dadd     // double-add
 8  dstore_1 // in var 1 and 2
 9  dload_1 //Push var 1 and 2
10  ldc2_w #4  // Push 100.0
13  dcmpg  // no if_dcmplt inst
14  iflt 5 // Compare and loop
17  return // Return void
```

A `double` value occupies two local variables.

```
double doubleLocals(double d1, double d2) {
                      Method double doubleLocals(double,double)
   return d1 + d2;     0  dload_1   // 1st arg in local var 1 and 2
}                      1  dload_3   // 2nd arg in local var 3 and 4
                       2  dadd
                       3  dreturn
                       // The return value is on stack top.
```

There are no `byte`, `short`, and `char` versions of `load`, `store` or `add` instructions.

```
void sspin() {
    short j;
    for (j = 0; j < 100; j ++) {
        ;   // empty loop
    }
}
```

```
Method void sspin()
 0   iconst_0
 1   istore_1 // j = 0
 2   goto 10
 5   iload_1  // Treat short as int
 6   iconst_1
 7   iadd
 8   i2s // Truncate int to short
 9   istore_1
10   iload_1   // check if j < 100
11   bipush 100
13   if_icmplt 5
16   return
```

`byte`, `short`, and `char` are transformed to the `int` type inside JVM. `byte` and `short` are sign-extended to `int`; `char` is zero-extended. Operations on `byte`, `short`, and `char` are done by `int` instructions.

`long` and floating-point types are supported by JVM, lacking only conditional control-transfer instructions.

JVM generally does arithmetic operations on the operand stack (`iinc` is an exception).

```
int align2grain(int j, int grain) {
   return ((j + grain-1) & ~(grain-1));
}
Method int align2grain(int,int)
  0    iload_1          // push j
  1    iload_2          // push grain
  2    iadd             // j + grain
  3    iconst_1         // push int constant 1
  4    isub             // j + grain - 1
  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
```

```
10    iand
11    ireturn          // return value is on stack top.
```

Run-Time Constant Pool

(constant pool in obfuscation)

Many numeric constants as well as objects, fields, and methods are accessed via the run-time constant pool of the current class.
Data of types `int`, `float` as well as references to `String` instances are managed with `ldc` and `ldc_w` instructions. `ldc_w` has a larger index range.
`ldc2_w` is for `long` and `double`.
`byte`, `short`, `int`, and `char` are managed with `bipush`, `sipush` and `iconst_<i>`. Certain small floating-point constants can be done with `fconst_<f>` and `dconst_<d>`.

```
void useManyNumeric() {     Method void useManyNumeric()
    int k = 100;            0 bipush 100 //Push small int with bipush
    int j = 1000000;        2 istore_1
    long l1 = 1;            3 ldc #1  // Push int constant 1000000;
                            // a larger int value uses ldc
```

```
    long l2 = 0xffffffff; 5 istore_2
    double d = 2.2;       6 lconst_1
                // A tiny long value uses short, 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 using ldc2_w
                          11 lstore 5
                          13 ldc2_w #8
                // Push double constant 2.200000; uncommon
                // double values are also pushed using ldc2_w
                          16 dstore 7
```

A Simple `while` Loop

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

```
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 is at the bottom of the loop. This incurs an extra `jump` instruction if the loop body is not executed. However, if the loop is executed at least once, this code saves one `jump` per iteration.

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

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

Each floating-point type has two comparison instructions: `fcompl` and `fcmpg` for type `float` and `dcmpl` and `dcmpg` for type `double`. The variants differ only in their treatment of `NaN`. `NaN` is unordered so all floating-point comparisons fail if any operand is `NaN`.

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


        return -1;
    }
}
```

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

Receive Arguments

If n arguments are passed to an instance method, they are received in the local variables numbered 1 through n. By convention, an instance method is pssed a `reference` to its instance in local variable 0 (that is, `this` in Java).

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

```
Method int addTwo(int,int)
   0    iload_1 // Push local var 1, i
   1    iload_2 // Push local var 2, j
   2    iadd    // Add
   3    ireturn // Return int result
```

However, for `class` methods, there is not such a `reference` parameter. The usual parameters are at local variables 0, 1, etc.

```
static int add2Static(int i, int j)  Method int add2Static(int,int)
{   return i + j;                                0    iload_0  // i
}                                                1    iload_1  // j
                                                 2    iadd
                                                 3    ireturn
```

There are 4 types of invoke instructions:

1. `invokevirtual index1 index2`: for invoking instance methods. *index1 index2* together forms an index into the constant pool where there is a descriptor of the called method (including address, number and types of arguments, max size of the operand stack).
2. `invokeinterface`: for interface methods
3. `invokespecial`: for special functions, such as initialization methods. The storage of an object is allocated with the `new` instruction and one of the constructor is called with the `invokespecial` instruction.
4. `invokestatic`: for class methods

There are several `return` instructions:

1. `ireturn`: returns an integer result

2. `dreturn`: returns a double result
3. `areturn`: returns an address result
4. `return`: returns a void value

Call an Instance Method with `invokevirtual`

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 (of `add12and13`).
The argument to `invokevirtual`, i.e., `#4`, is a symbolic reference.

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

```
Method int add12and13()
 0  aload_0  // Push local var 0, this
 1  bipush 12  // Push int constant 12
 3  bipush 13  // Push int constant 13
 5  invokevirtual #4
 // Method Example.addTwo(II)I
 // signature
 // Return int on top of operand stack;
 // it is the int result of addTwo()
 8  ireturn
```

Call a Class Method with `invokestatic`

Note that there is no `this` reference in local variable 0.

```
int add12and13() {
    return add2Static(12, 13);
}
```

```
Method int add12and13()
 0  bipush 12
 2  bipush 13
 4  invokestatic #3
 // Example.add2Static(II)I
 7  ireturn
```

Call Instance Initialization Methods with `invokespecial`

`invokespecial` is also used to invoke methods in the superclass (`super`) and `private` methods. Note that methods called using `invokespecial` always pass `this` to the invoked method as its first argument. As usual, it is received in local variable 0.

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

    private int getIt() {
        return it;
    }
}
```

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

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

}
```

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

Create Instances with `new`

Once the class instance is 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. Note that all constructors of a class are named `init` inside JVM.

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

```
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, though with its own complement of instructions.

```
int i; // An instance var
MyObj example() {
    MyObj o = new MyObj();
    return silly(o);
}
```

```
Method MyObj example()
 0    new #2  // Class MyObj
 3    dup
 4    invokespecial #5
 // Method MyObj.<init>()V
 7    astore_1  // o
 8    aload_0   // this
 9    aload_1   // o
10    invokevirtual #4
// Method Example.silly(LMyObj;)LMyObj;
13    areturn
```

```java
MyObj silly(MyObj o) {
   if (o != null) {
       return o;
   } else {
       return o;
   }
}
```

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

The fields of an instance (i.e., instance variables) are acessed with `getfield` and `putfield`.

```
int i;  // instance var
...
void setIt(int value) {        Method void setIt(int)
   i = value;                   0    aload_0       // this
}                               1    iload_1       // value
                                2    putfield #4   // Field Example.i
                                5    return        // void

int getIt() {                  Method int getIt()
   return i;                    0    aload_0       // this
}                               1    getfield #4   // Field Example.i
                                4    ireturn       // int
```

Some data movement instructions

`iconst_1`: pushes the integer constant 1 onto the stack.

`bipush data`: pushes a one-byte signed integer

`sipush data1 data2`: pushes a two-byte signed integer

`ldc index`: push single-word constant onto stack

`ldc_w index1 index2`: push single-word constant onto stack (wide index)

Conditional branches make use of PC relative addresses.

`ifeq data1 data1`: check if the integer on stack top is zero. If so, branch to the PC relative address with an offset calculated by concatenating the two data bytes.

`if_icmpeq data1 data2`: check if the two integers on stack top are equal. If so, branch to the PC relative address with an offset calculated by concatenating the two data bytes.

`ifnull data1 data2`: check the reference on stack top is null.

`newarray` is used to create arrays of numeric types.

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

```
Method void createBuffer()
 0     bipush 100
 2     istore_2      // bufsz
 3     bipush 12
 5     istore_3      // value
 6     iload_2       // Push bufsz
 7     newarray int  // create array
 9     astore_1      // buffer
10     aload_1       // Push buffer
11     bipush 10
13     iload_3       // Push value
14     iastore       // buffer[10]
15     aload_1       // Push buffer
16     bipush 11
18     iaload        // buffer[11]
```

```
19    istore_3     // value
20    return
```

`anewarray` is used to create 1-dim arrays of object references.

```
void createThreadArray() {
    Thread threads[];
    int count = 10;
    threads = new Thread[count];
    threads[0] = new Thread();

}
```

```
Method void createThreadArray()
 0    bipush 10  // Push 10
 2    istore_2   // count
 3    iload_2
 4    anewarray class #1
 // Create array of class Thread
 7    astore_1   // threads
 8    aload_1
 9    iconst_0
10    new #1
// Create instance of class Thread
13    dup
14    invokespecial #5
// pass to instance initialization method
// Method java.lang.Thread.<init>()V
```

```
17    aastore
// Store new Thread at threads[0]
18    return
```

`multianewarray` creates multi-dimesional arrays.

```
int[][][] create3DArray() {      Method int create3DArray()[][][]
   int grid[][][];                0    bipush 10
   grid = new int[10][5][];       2    iconst_5
   return grid;                   3    multianewarray #1 dim #2
                // Class [[[I, a three dimensional int array;
                // only create first two dimensions
}                                 7    astore_1  // grid
                                  8    aload_1
                                  9    areturn
```

All arrays have associated lengths, by the `arraylength` instruction.

`tableswitch` **and** `lookupswitch` **instructions**:

```
int chooseNear(int i) {              Method int chooseNear(int)
   switch (i) {                       0    iload_1      // local var i
       case 0:  return 0;             1    tableswitch 0 to 2:
                                                    0: 28
                                                    1: 30
                                                    2: 32
                                              default: 34
       case 1:  return 1;            28    iconst_0
       case 2:  return 2;            29    ireturn
       default: return -1;           30    iconst_1
   }                                 31    ireturn
}                                    32    iconst_2
                                     33    ireturn
                                     34    iconst_m1
                                     35    ireturn
```

`tableswitch` and `lookupswitch` instructions operate on the `int` type.

When the cases of `switch` are sparse, the table representation of `tableswitch` becomes inefficient in terms of space. `lookupswitch` may be used instead.

```
int chooseFar(int i) {                 Method int chooseFar(int)
   switch (i) {                         0    iload_1  // local var i
      case -100: return -1;             1    lookupswitch 3:
                                                      -100: 36
                                                         0: 38
                                                       100: 40
                                                   default: 42
      case 0:    return 0;             36    iconst_m1
      case 100:  return 1;             37    ireturn
      default:   return -1;            38    iconst_0
   }                                   39    ireturn
```

```
}                                  40    iconst_1
                                   41    ireturn
                                   42    iconst_m1
                                   43    ireturn
```

Operations on Operand Stack

```
public long nextIndex() {              Method long nextIndex()
   return index++;                      0    aload_0  // Push this
}                                       1    dup      // two copies
private long index = 0;                 2    getfield #4   // index
               // One copy of this is consumed
               // index above the original this
                                        5    dup2_x1
                                        6    lconst_1  // long constant 1
                                        7    ladd
                                        8    putfield #4  // index
              // use the original copy of this
                                       11    lreturn
```

Throw Exceptions With `athrow`

```
void cantBeZero(int i) throws TestExc {

    if (i == 0) {
        throw new TestExc();
    }
}
```

```
Method void cantBeZero(int)
 0    iload_1  // i
 1    ifne 12  // if i != 0
 4    new #1   // instance of TestExc
 7    dup
 // One reference is for constructor
 8    invokespecial #7
 // Method TestExc.<init>()V
11    athrow
// Second reference is thrown
12    return
```

Compiling `try-catch` Blocks

```
void catchOne() {
    try {
        tryItOut();

    } catch (TestExc e) {
        handleExc(e);

    }
}
```

```
Method void catchOne()
 0  aload_0  // this
 1  invokevirtual #6
 // Method Example.tryItOut()V
 4  return   // normal return
 5  astore_1 // 1st handler
 // Store thrown value in local var 1
 6  aload_0  // Push this
 7  aload_1  // Push thrown value
 8  invokevirtual #5 // handler method:
 // Example.handleExc(LTestExc;)V
 11  return  // Return after TestExc
 Exception table:
   From  To  Target  Type
     0     4     5    Class TestExc
```

The `catch` block generates an entry in the exception table. The exception shows that exceptions occurring between line 0 and line 3 (not including line 4) will be handled by the handler.

There could be multiple `catch` clauses.

There could also be nested `try-catch` blocks.

Here is an example with two exception handlers.

```
void catchTwo() {
   try {
       tryItOut();
   } catch (TestExc1 e) {
       handleExc(e);
   } catch (TestExc2 e) {
       handleExc(e);
   }
}
```

```
Method void catchTwo()
 0  aload_0  // this
 1  invokevirtual #5
 // Method Example.tryItOut()V
 4  return   // normal return
 5  astore_1 // 1st handler
 6  aload_0  // push this
 7  aload_1  // push e
 8  invokevirtual #7
 // Example.handleExc(LTestExc1;)V
11  return // after handling TestExc1
12  astore_1 // 2nd handler
13  aload_0  // Push this
14  aload_1  // Push e
15  invokevirtual #7
```

```
18  return // after handling TestExc2
Exception table:
 From  To  Target  Type
  0     4     5    Class TestExc1
  0     4    12    Class TestExc2
```

Nested try-catch Blocks

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

```
Method void nestedCatch()
 0  aload_0   // push this
 1  invokevirtual #8
 // Method Example.tryItOut()V
 4  return    // normal return
 5  astore_1 // 1st handler
 6  aload_0   // Push this
 7  aload_1   // Push e
 8  invokevirtual #7
 // Example.handleExc1(LTestExc1;)V
11  return//after handling TestExc1
12  astore_1 // 2nd handler
13  aload_0   // Push this
14  aload_1   // Push thrown value
15  invokevirtual #6
```

```
  // Example.handleExc2(LTestExc2;)V
  18  return//after handling TestExc2
Exception table:
 From  To  Target  Type
  0    4     5     Class TestExc1
  0   12    12     Class TestExc2
```

Note the order of the two exception handlers in the exception table.

Compiling `finally`

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

```
Method void tryFinally()
 0  aload_0 // push this
 1  invokevirtual #6
 // Method Example.tryItOut()V
 4  jsr 14  // call finally block
 7  return
 8  astore_1  // handler for any throw
 9  jsr 14  // Call finally block
12  aload_1 // Push thrown value
13  athrow  // rethrow the value to invoker
14  astore_2 // Beginning of finally,
             // return address
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
```

The `finally` block is compiled as an embedded subroutine and invoked with `jsr`. When `finally` block completes, `ret 2` returns control to the instruction folllowing `jsr`. Specifically, `jsr` pushes the return address onto operand stack. `astore_2` saves that address in local variable 2. Finally, `ret 2` jumps to the address saved in local variable 2.

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

```
Method void tryCatchFinally()
 0  aload_0
 1  invokevirtual #4 // tryItOut()V
 4  goto 16
 7  astore_3 // Beginning of 1st handler
 8  aload_0  // Push this
 9  aload_3  // Push thrown value
10  invokevirtual #6 // handleExc
13  goto 16
16  jsr 26   // Call finally block
19  return   // after handling TestExc
20  astore_1 // Beginning of 2nd handler
21  jsr 26   // Call finally block
24  aload_1  // Push thrown value...
25  athrow   //...and rethrow to invoker
26  astore_2 // Beginning of finally
```

```
27  aload_0  // Push this
28  invokevirtual #5 // wrapItUp()V
31  ret 2    // Return from finally
Exception table:
 From  To  Target  Type
  0     4    7     Class TestExc
  0    16   20     any
```

Synchronization with `monitorenter` and `monitorexit`

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

```
Method void onlyMe(Foo)
 0  aload_1  // Push f
 1  astore_2 // Store it in local var 2
 2  aload_2  // Push local variable 2 (f)
 3  monitorenter // Enter the monitor of f
 4  aload_0  // this
 5  invokevirtual #5 // doSomething()V
 8  aload_2  // Push local variable 2 (f)
 9  monitorexit  // Exit the monitor of f
10  return   // Return normally
11  aload_2  // In case of throw, stop here
12  monitorexit  // Be sure to exit monitor
13  athrow   // ...then rethrow to invoker
Exception table:
 From  To  Target  Type
```

`4 8 11 any`

However, the most common use of synchronization is a `synchronized` method. A `synchronized` method is *not* implemented with `monitorenter` and `monitorexit`. Rather, an `ACC_SYNCHRONIZED` flag of the method is set, which is checked by the method invocation instructions. When invoking a method with the `ACC_SYNCHRONIZED` flag set, the current thread acquires a monitor, invokes the method, and releases the monitor after the method completes. If an exception occurs during a method execution and the method does not handle the exception, the monitor of the method is automatically released before the exception is thrown out of the `synchronized` method.

Nested (i.e. Inner) Classes and Interfaces

See inner classes specification.

3 Sample Java Code

```
public static void main(String[] args){
  int i;
  for(i = 0; i < 10; i++) {
    if (i == 5) System.out.println("i = 5");
    else System.out.println("i != 5");
  }
}
```

```
   0:   iconst_0
   1:   istore_1
   2:   iload_1
   3:   bipush  10
   5:   if_icmpge 38
   8:   iload_1
```

```
9:   iconst_5
10:  if_icmpne 24
13:  getstatic #2;
       //Field java/lang/System.out:Ljava/io/PrintStream;
16:  ldc     #3; //String i = 5
18:  invokevirtual #4;
       //Method java/io/PrintStream.println:(Ljava/lang/String;)V
21:  goto    32
24:  getstatic #2;
       //Field java/lang/System.out:Ljava/io/PrintStream;
27:  ldc     #5;    //String i != 5
29:  invokevirtual   #4;
       //Method java/io/PrintStream.println:(Ljava/lang/String;)V
32:  iinc    1, 1
35:  goto    2
38:  return
```