The Java Virtual Machine

3.1 Introduction

It is arguable that Java and its “compile once, run anywhere” slogan started the current interest in virtual machines; indeed, it would appear to have popularised the term “virtual machine”.

This chapter is organised as follows. First, the Java language is briefly introduced. Next, the gross organisation of the Java Virtual Machine—the JVM for short—will be described. In that section, the storage organisation used by the JVM and the organisation of the stack is presented and major concepts such as the Runtime Constant Pool and its contents, including the method areas and class file representation are introduced. The instruction pointer (or program counter—pc in JVM terms) is also briefly discussed. This is followed by a relatively short description of the “class file”, a runtime representation of each class; this description is followed by a brief outline of so-called “class resolution,” the process of locating and loading classes at runtime.

Section 4 is concerned with the JVM’s instruction set. The instruction set can be described in a number of ways. A subset of the instruction set is clearly typed, while another is not. Some instructions are at a relatively high level (e.g., those dealing with locks and exception), while others (e.g., jumps and arithmetic) are not. Finally, there are special-purpose instructions directly tailored to Java’s needs: those dealing with locks, monitors, and method and data location, for example.

The final section acts as a summary of the main points. It also discusses the relationship between the main components of the JVM and the source structure of Java programs.

It is not possible, given the space available here, to describe the JVM exhaustively. Instead, all that can be done is to give the reader a general impression that is detailed in some places and superficial in others. Readers interested in the exact details should consult [33]. For information about Java itself, the language definition [22] should be consulted. It is, of course, not possible to understand the details of the JVM in complete detail unless the
language is completely understood. It is strongly recommended that interested readers should consult both of these texts.

3.2 JVM Organisation: An Overview

This section contains an overview of the JVM's organisation. This description is based upon the published specification [33].

The JVM is a stack-based virtual machine. It contains these basic structures:

- The heap store (main store);
- The stack;
- The method area;
- Runtime Constant Pools;
- The PC register;

The stack and the "class file" objects are stored in the heap, as are the Constant Pools and the method area. In addition, there should be structures to support threads and monitors—they will be considered only (Section 3.9).

The JVM specification is silent on issues pertaining to the heap's management. A JVM can use a mark and scan, stop-and-copy or a generational garbage collector, therefore. A pure reference-counting storage management regime cannot be used, however, unless it is supported by some other mechanism. The reason for this is that circular links can exist between entities stored in the heap (Java has an imperative semantics and, therefore, supports assignment).

There are, in fact, two stacks in the JVM specification: the "native code" stack (or "C stack") and the "Java stack". The first can be disposed of fairly readily. It is the stack used to implement the JVM itself; it is also the stack used for intermediate storage by all the primitive routines in a JVM and by all the code implementing JVM instructions. Additional primitives, as well as native methods, are implemented using the "native code" stack. In most implementations, this stack will be the one used by the C runtime system. This stack will not be of further interest because it is beyond the control of the JVM specification.

The other stack is the JVM stack proper. It is a framed stack. A stack frame is allocated when control enters a method. It is deallocated when control returns from the method that caused its allocation. There are two cases of return from a method:

- Normal return. This is performed by the execution of a return instruction.
- Abnormal return. This is performed when an exception of some kind is caught by a handler that is not inside the method invocation associated with the stack frame.
The JVM specification uses the term *completion* to refer to return from a method; it also uses the term *abrupt completion* for what is termed "abnormal return" in the above list. A method can return zero or more values; the value can be of a simple (scalar) type, an array type or a class type (i.e., a reference to an instance of a class).

### 3.2.1 The stack

The frames of the stack are allocated in the heap. Each stack frame consists of:

- A purely local stack called the *operand stack*;
- A set of local variables;
- A reference to the code associated with the method being executed. This is in the Runtime Constant Pool of the class with which the method is associated;
- A slot for the PC register.

The operand stack is for the storage of temporary and intermediate results in the usual fashion. It is used, *inter alia*, for the evaluation of expressions and parameters.

The set of local variables is used to store:

- The variables local to the method whose activation (invocation) caused the stack frame to be allocated. This is called the *local variable array* in [33].
- Parameters passed from the invoking context.
- The *this* pseudo variable that always points to the instance upon which the method operates.

Each local variable is 32 bits in length; this corresponds to the internal JVM integer length. It generally also corresponds to the length of a pointer on the host machine. When an entity is longer than 32 bits, two consecutive locals are allocated. The index of the entity in such a case is considered to be the lowest of the indices required to represent it. For floating point numbers (which are implemented according to most of the IEEE 444 standard) or long integer values, which occupy 64 bits on a 32-bit machine, two consecutive local variables are allocated. The standard defines a big-endian representation for all values.

Because of these considerations, there will, in general, be more local variables in a stack frame's local variable array than there are local variables in the corresponding method's source code. In addition, the JVM specification permits implementations of the Java compiler to allocate more local variables when they are needed to optimise method code.

The JVM contains instructions to access and update local variables in the current stack frame's variable array. There are also instructions to manipulate the local stack.
When a method is invoked, a stack frame is created for it. The parameters to be passed to it are evaluated and the results stored in the new stack frame. The code of the method is then executed using the newly created stack frame as its context.

If the method is a class method (i.e., a method declared as static in the associated class definition), the parameters passed to it begin at the first element of the local variable array. If the method is an instance method, the first element of the local variable array is reserved for the self (or this) pointer; the parameters are then allocated to the second and subsequent elements. The allocation of parameters is, in both cases, contiguous.

In order to return control to its caller in a normal fashion (i.e., a non-error return or normal completion as the JVM specification terms it), the method executes one of the return instructions. These instructions are used to return zero or more values.

When normal completion of a method occurs and the result is passed to the calling method, the called method’s stack frame is removed from the stack and is garbage collected. The PC in the exiting stack frame is stored in the JVM’s PC register as a return point.

If a method makes an abnormal return (or abnormal completion) by throwing an exception that is not handled within its body, a search is performed along the call chain. The stack frame for each method between the point at which the exception is thrown and that at which it is handled is removed from the stack and left to be garbage collected (in other words, the stack is collapsed). The exception is then handled in the context of the method defining the handler for exception. It must be noted that the handler is always the nearest one along the dynamic (call) chain, not along the static chain.

If an exception is thrown by a method called inside a thread and is handled by a handler that is outside that thread, the stack associated with the thread is terminated. Thread termination causes the store allocated for its stack to be reclaimed; other structures associated with the thread also become garbage.

In most implementations of the JVM, there is always at least one thread running at any time. In the case in which there is just one active thread, should an exception be thrown and not handled by that thread, an exception must be raised by the JVM itself; the thread’s data structures are also consigned to the garbage collector.

There are other cases in which the JVM has to handle exceptions. They will not be documented here. The interested reader should consult [33] for details.

### 3.2.2 Method areas

Method code is stored in method areas at runtime. The actual location of each method area is within the class file object representing the class to which the method directly belongs (the one in which it was defined in the source code). Method areas contain byte codes and symbolic references.
The method area also contains the Runtime Constant Pool for each class or interface loaded in the JVM. The Runtime Constant Pool is the runtime representation of the `constant_pool` table in the class file associated with the class or interface. Each Runtime Constant Pool contains a number of different types of constants. The constants it contains range from numeric literals to method and field references; the former are determined at compile time, while the latter are determined (or “resolved”) at runtime.

Class files are used to derive a runtime representation of classes. They contain the code for methods (in the method area), as well as the variables associated with the class and other information (initialisation values, for example). Methods and variables can refer to entities within the same class or to entities within other classes. These references are represented in the class file as symbolic references that have to be converted at runtime into addresses in the JVM’s store. The stack frame of the currently executing method contains a reference to the (runtime representation of the) class associated with that method. This reference permits the dynamic linkage of methods.

A class file object is allocated when a class is loaded. When the class is no longer of use (there are no more references to it either on a stack or in the heap), the storage it occupies become garbage (and is reclaimed by the garbage collector).

When a class (or interface—this is discussed in a little more detail below in Section 3.3) is loaded, the data in the class file is processed in various ways. Some of it is used for verification purposes; the rest is stored in a newly allocated space in the method area. Class files will be discussed in more detail below (Section 3.3).

Runtime Constant Pools act in a way that is reminiscent of symbol tables in other languages.

### 3.2.3 The PC register

This is the program counter or instruction pointer register. It points to the instruction currently being executed by the JVM, provided that that instruction is a Java bytecode (otherwise, the native code stack—the C stack—and the host machine’s registers hold the context and code and the PC register’s value is undefined). When the JVM is executing more than one thread, each thread has its own stack and its own PC register. At any point during the execution of the JVM, one thread is being executed.

As noted above, the current method’s stack frame contains a slot for the PC register. This is used in the normal way to store the return point at which execution continues after the method returns. In order to cope with native code routines calling JVM coded methods, the PC register (and associated slot in stack frames) must be large enough to hold a native code address.
3.2.4 Other structures

The stack structure is permitted by the JVM specification [33] to contain data other than that described above. Additional data can be used for debugging, for example.

A JVM implementation has also to support threads. The actual implementation is relatively free. There are two basic forms of thread implementation: so-called “green” threads and operating system specific ones. The latter is introduced so that the thread mechanisms of the host operating system can be used to implement Java threads (e.g., Linux threads). The former is an independent thread mechanism that is implemented by the JVM itself.

If the JVM implements the thread mechanism directly, storage structures must be provided to support it. For example, monitors and queues must be implemented, as well as a way to store state information for threads and thread groups. These structures will, probably, reside in the JVM’s heap.

If the native thread mechanism is used, the structures to implement it are provided by the native operating system and fall outside of the JVM (using the C stack escape mechanism mentioned above).

In addition, structures required to interface to such things as the host’s graphics system (say, X Windows), sockets and remote procedure calls are also required. This interface might require the use of the JVM heap to hold the necessary state information. The C stack escape is also used as part of this interface.

3.3 Class Files

The class file structure is fundamental. It contains the compiled code for classes in a standard format. The JVM specification states that the output of a Java compiler need not be in class-file format; however, the runtime system depends upon the information contained in class files. Indeed, the first step taken by the JVM in loading a class file is to verify that the entity being loaded is in class file format. The JVM specification defines what a class file should contain and the representation to be used; it defines the order in which class file components appear at runtime. It also defines the verification processes that should take place before a class can be considered to have been loaded into the JVM.

There is one class file for each class currently loaded in the JVM. Class files can be directly input by a Java compiler, loaded from file or from a local database. They can also be loaded from remote sources such as remote files, remote databases, code servers or just from a socket. The class loading mechanism is relatively straightforward but complicated by the fact that it can be replaced by an application-specific mechanism; it is associated with the class resolution mechanism. The loading mechanism is also the place where most of the security mechanisms of a JVM is located (bytecode verification,
among other things). The details of the class loader are omitted from this
description for reasons of its complexity. Instead, the class-resolution process
will be described because it is this, at runtime, that resolves references stored
in class files to its own methods (the methods defined inside that class), to its
own variables (the variables declared inside that class) and to other classes.

It should be observed that there is also one class file for each interface cur-
tently loaded in the JVM. This is reasonable, for interfaces define entities in
their own right. It makes even more sense for Java 2 because it has augmented
interfaces so that they are, in effect, abstract classes supporting multiple in-
heritance rather than simply a device for providing multiple inheritance in a
single-inheritance language. For this reason, a class file can define a class or
an interface.

The organisation of a class file is rather complex. The JVM specification
is highly detailed and includes specifications of data formats. Here, the organ­
isation will only be summarised. As usual, the interested reader is directed to
the relevant sections of [33] for the details.

The top-level structure of the class file are indicative of the information it
contains. The JVM specification ([33], p. 94) defines the structure as contain­
ing:

- **magic**: The magic number identifying this as a class file. The value should be
  0xCafeBabe.
- **minor version**: This, together with the next field, define the version number
  of the class file. The version number is used by the JVM to determine
  whether it can execute the class in a class file.
- **major version**: The number of entries in the constant pool
- **constant pool**: The constant pool. This is a vector of bytes at the top level.
  The bytes are, however, structured internally as a table.
- **access flags**: This is a bit mask whose bits represent flags. The flags denote
  access permissions to and properties of this class (or interface). The flags
  defined in [33] are as follows (their values are omitted):
  - **public**: If this flag is set, the class was declared public and can be accessed
    from outside the package in which it was defined.
  - **abstract**: If this flag is set, the class has been declared abstract. This means
    that the class cannot be instantiated and any attempts to instantiate
    it must cause an error.
  - **final**: If this flag is set, the class was declared as begin final. This means
    that it is not permitted to define subclasses of this class.
  - **super**: If this flag is set, superclass methods should be treated specially
    when invoked by the **invokespecial** instruction.
  - **interface**: If this flag is set, the entity defined by this class file is an inter­
    face, not a class.
this class This is an index into the constant pool table. The entry in the constant pool at the offset in this field must be a structure of type CONSTANT_Class_info that represents the class or interface defined by this class file. This must be a valid index.

super class If this class file defines a class, the value of this field must either be zero or a valid index into the constant pool. If the value is non-zero, the entry at the offset it specifies must be a CONSTANT_Class_info structure that represents the direct superclass of the class defined by this class file. It is not permitted that any of the superclasses of this class be declared as final.

interfaces count This is the number of direct super interfaces of the class or interface defined by the contents of this class file. It must be a valid index.

interfaces This is an array of offsets into the constant pool table. Each entry indexed by this array must be a CONSTANT_Class_info structure that represents an interface that is a direct super interface of the class or interface defined by the contents of this class file.

fields count This is another numeric field whose value represents the number of elements in the fields table.

fields Each element of this table is a field_info structure representing a complete description of a field in this class or interface. The table only includes those fields that are declared by this class or interface (so excludes inherited ones). A field can be a class or an instance variable.

methods count This numeric field contains the number of methods declared in this class or interface. It is used as the size of the methods table that immediately follows.

methods Each element of this table must be a method_info structure giving a complete description of a method that this class or interface defines. The method can be native or abstract (in which case, no JVM instructions are included). If the method is neither native nor abstract, the JVM instructions that implement the method are included in the structure. The table includes all methods declared (defined) by the class or interface defined by this class file; therefore, it includes static (class) methods and any class or interface initialisation methods that are inherited from superclasses or superinstances.

attributes count This is another numeric field. It contains the size of the attributes table that ends the class file.

attributes This is a table of attributes for the class. In [33] (p. 98), it is stated that the only attributes causing the JVM to take any action are the Source-File and Deprecated attributes; all other values are ignored by the JVM.

In the JVM definition, when field names in the above list are composed of more than one word, connecting underscores ("_") are used to create a valid C (Java) identifier. The underscores have been omitted above in order to render them more legible. It should be noted that, below, what should be spelled as "constant_pool" will always be spelled without the underscore.
The class file description refers to the fact that constant pool entries are structured. In addition, it refers to structures of type CONSTANT._Class_info, field_info and method_info. These will be described in the next few paragraphs. The description is necessarily limited, so the interested reader is referred to [33], Chapter 4 (the relevant sections and page numbers are included below).

The constant pool entries ([33], Section 4.4, p. 103 et seq.) are structures containing a tag and some information. The tag is of fixed size and indicates the type of the constant. The possible tags and their interpretations are:

Class A reference to a class or interface. The information associated with this tag must be a valid constant pool index. The entry at that index must be a Utf8 structure (essentially a string) that represents a fully qualified class or interface name in the internal form (this is a form containing slashes instead of periods separating the path name of the package containing the named class).

Fieldref, Methodref, InterfaceMethodref A reference to a field, method or interface method, as appropriate. The information associated with these three tags have the same format (this justifies their common treatment here):

- class_index This field must contain a valid constant pool index. The corresponding entry must be a CONSTANT.Class_info structure that represents the class or interface type containing the field or method declaration.
- name_and_type_index This should be another valid constant pool index. Its entry must be a CONSTANT_NameAndType_info structure indicating the name and descriptor of the field or method as appropriate. If this is a Methodref and the entry is of the appropriate type, the entry can begin with the character <, the name must be the special <init> name (thus representing an instance initialisation method—these methods should return no value, note).

String This represents a constant string. The information associated with it is an index into the constant pool. The element at that index must be a string (encoded as a Utf8 object).

Integer This represents a constant integer. The associated information represents the value of that integer constant. The bytes are stored in big-endian format (high-order byte first).

Float This represents a floating point numeric constant in IEEE 754 floating point single format. The bytes are stored in big-endian format (high-order byte first).

Long This represents a long integer constant. The associated information represents the value of the long in the form four highest-order bytes followed by four lowest-order bytes.

1 The JVM structure type Utf8 will just be regarded as a string. Its name will always be written in this font for ease of reading. It is actually defined as a length and a sequence of suitably encoded bytes representing the characters of the string.
Double This represents a double constant in IEEE 754 double format. The high-order four bytes appear first and are followed by the four low-order bytes.

NameAndType This is used to represent a field or method. It does not indicate the class or interface to which it belongs. Its associated information is in the form:

name_index This must be a valid constant_pool index. The entry at that index must be a Utf8 structure (string) representing the name of the field or method; it must be a valid identifier as defined by the Java language specification [22]; it can be the special identifier <init>, denoting an initialiser (see below, Section 3.5).

descriptor_index The contents of this field must again be a valid constant pool index. The entry corresponding to this value must be a Utf8 structure representing a valid field or method descriptor.

Utf8 An object of this type is, basically, a text string. It contains a field containing the length of the string and another containing the sequence of characters.

(The actual tag name has the string CONSTANT_ as a prefix.) It should be remembered that the values stored in these entries are constants.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUBLIC</td>
<td>can be accessed outside its package</td>
</tr>
<tr>
<td>PRIVATE</td>
<td>accessible only within its defining class</td>
</tr>
<tr>
<td>PROTECTED</td>
<td>accessible within subclasses</td>
</tr>
<tr>
<td>STATIC</td>
<td>static (class method)</td>
</tr>
<tr>
<td>FINAL</td>
<td>cannot be overridden</td>
</tr>
<tr>
<td>SYNCHRONIZED</td>
<td>must be wrapped in a monitor lock</td>
</tr>
<tr>
<td>NATIVE</td>
<td>not implemented in Java</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>no implementation provided (permitted in defining class)</td>
</tr>
<tr>
<td>STRICT</td>
<td>FP-strict floating point mode</td>
</tr>
</tbody>
</table>

The structure type method_info ([33], Section 4.6, p. 114 et seq) has five fields as follows:

access_flags These are defined in Table 3.1. A class method can only specify one of PUBLIC, PRIVATE or PROTECTED. If such a method has ABSTRACT set, it may not have FINAL, NATIVE, PRIVATE, NATIVE, STRICT or SYNCHRONIZED also set. All interface methods must have PUBLIC and ABSTRACT set. A named initialisation method can have at most one of PRIVATE, PROTECTED and PUBLIC and, possibly, STRICT but no other flags set. Only the value of the STRICT flag is used by the JVM if the
method is a class or initialisation method; the other values are ignored. (The actual flags are all spelled with an **ACC_** prefix; it has been omitted in order to improve legibility.)

**name_index** This must be a valid constant pool index. The corresponding entry must be a Utf8 string that represents one of the special method names, `<init>` for instances or `<clinit>` for classes, or a valid method identifier as defined by the Java language definition [22]. The special initialisation methods are described in Section 3.5 below.

**descriptor_index** This is another constant pool index; it must be valid. The corresponding entry is a Utf8 string representing a valid method descriptor.

**attributes_count** The number of attributes appearing in the following field of this structure type.

**attributes** The only attributes of interest are the **code** and **exception** attributes. All others are silently ignored by the JVM.

A **method descriptor** is a (Utf8) string that is specified as:

```
(ParameterDescriptor* ) ReturnDescriptor
```

where ( and ) represent open and close parenthesis; these parentheses appear literally in the method descriptor. The **ParameterDescriptor** part allows for zero or more parameters, each of which is a field descriptor. The **ReturnDescriptor** has the form:

```
ReturnDescriptor: FieldType | V
```

where **V** denotes the void type and where **FieldDescriptor** has the following syntax:

```
FieldDescriptor: FieldType
ComponentType: FieldType
FieldType: BaseType | ObjectType | ArrayType
```

The **BaseType** denotes the specification of one of Java’s primitive types, while **ObjectType** and **ArrayType** specify object or array types. In a descriptor, a standard encoding for types is employed (it is defined in [33], p. 101). Typically, the encoding is a single character for primitive types and more complex strings for object and array types. The basic encoding is shown in Table 3.2.

The codes shown in Table 3.2 are sufficient to describe any valid Java type. For example, the array type:

```
double d[][]
```

is encoded as:

```
[[[D
```

while an ordinary integer is denoted by **I** (just that: the letter "I" on its own). Finally, the type **Object** is denoted by the descriptor **Ljava/lang/Object**.

As noted above, there are two attributes of particular importance to methods: the **code** and **exception** attributes. They are considered in turn.
Table 3.2. JVM basic descriptor type codes.

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Designated Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>byte</td>
<td>signed byte</td>
</tr>
<tr>
<td>C</td>
<td>char</td>
<td>unicode character</td>
</tr>
<tr>
<td>D</td>
<td>double</td>
<td>double-precision floating point</td>
</tr>
<tr>
<td>F</td>
<td>float</td>
<td>single-precision floating point</td>
</tr>
<tr>
<td>I</td>
<td>int</td>
<td>integer</td>
</tr>
<tr>
<td>J</td>
<td>long</td>
<td>long integer</td>
</tr>
<tr>
<td>L&lt;classname&gt;;</td>
<td>reference</td>
<td>instance of class classname</td>
</tr>
<tr>
<td>S</td>
<td>short</td>
<td>signed short</td>
</tr>
<tr>
<td>Z</td>
<td>boolean</td>
<td>boolean value</td>
</tr>
<tr>
<td>[</td>
<td>reference</td>
<td>one array dimension</td>
</tr>
</tbody>
</table>

The code attribute is another variable-length attribute. As will be seen, this attribute contains the JVM instructions implementing the method, as well as other information, as will now be outlined.

attribute.name.index This must be a valid index into the constant pool whose corresponding entry is a Utf8 structure representing the string “Code”.

attribute.length This numeric field denotes the length of the entire attribute minus the first six bytes (i.e., the first two fields of this structure).

max_stack This numeric field denotes the maximum size to which the operand stack can grow during execution of the method represented by this attribute.

max_locals This numeric field denotes the size of the local variables array in the stack frame for each invocation of this method. The value includes the number of elements required for passing parameters into the method represented by this structure.

code.length This field contains the length of the code field (which follows). The length is expressed in bytes.

code This is a vector containing the actual JVM instructions that implement the method represented by this structure.

exception_table.length This field contains the number of entries in the exception_table attribute (which follows).

exception_table This is a table whose entries are structures of the form:

start.pc, end.pc These two values indicate the range in the code vector in which the exception handler is active. The value of start.pc must be a valid index of the opcode of an instruction in the code vector. The value of end.pc must either be a valid index of the opcode of an instruction in the code vector or must be equal to code.length. The value of start.pc must be less than that of end.pc. The value of start.pc is inclusive and that of end.pc is exclusive.
3.3 Class Files

**handler_pc** The value of this field indicates the start of the exception handler. This must be a valid index into the code vector and must be the index of an instruction’s opcode.

**catch_type** If this has a non-zero value, it must be a valid constant pool index whose corresponding entry must be a CONSTANT_CLASS_INFO structure that represents an exception class (class Throwable or one of its subclasses) that this handler is intended to handle. The handler will be called only if the thrown exception is an instance of the stated class or of one of its subclasses. If the value is zero, this handler is called for all exceptions; this is used to implement finally clauses.

**attributes_count** This field contains the number of entries in the last attribute.

**attributes** The attributes for the method. There are many such attributes. They are ignored here.

When the code in a code attribute is read into store on a byte-addressable machine, if the first byte of the vector is aligned on a 4-byte boundary, the 32-bit offsets in the tableswitch and lookupswitch instructions will be correctly aligned (on 4-byte boundaries).

The exception attribute is another variable-length structure. It records the exception types that a method can throw. It contains four fields as follows:

**attribute_name_index** This must be a valid constant pool index which points to a Utf8 structure containing the string “Exception”.

**attribute_length** This numerical field records the length of the exception attribute structure minus the first six bytes (this attribute and the previous one).

**number_of_exceptions** This numerical field records the number of entries in the following field.

**exception_index_table** Each element of this vector must be a valid constant pool index which refers to a Utf8 structure representing a class type that the method is declared to throw.

A method can throw an exception if at least one of the following conditions is met at runtime:

- The exception is an instance of class RuntimeException or one of its subclasses.
- The exception is an instance of Error or one of its subclasses.
- The exception is an instance of one of the exception classes (or one of its subclasses) specified in the exception_index_table described in the last list.

Finally, the structure type field_info ([33], Section 4.5, p. 112 et seq) has five fields as follows:

**access_flags** The permitted values are as shown in Table 3.3. (Note that the actual flags have the prefix ACC, which is omitted in the table.) Only one of the flags PUBLIC, PRIVATE and PROTECTED can be set for any field. Only one of FINAL and VOLATILE can be set for any field. All interface
fields must be PUBLIC, STATIC and FINAL; no other flags may be set in this case.

name_index This is a constant pool index; it must be valid. The entry specified by this field must be a Utf8 string representing a field name that is a valid identifier according to the Java language standard [22] and the syntax is as shown above.

descriptor_index This is another constant pool index; it must be valid. It must be another Utf8 string representing a valid field descriptor. The coding scheme for the field descriptor is as shown in Table 3.2.

attributes_count This is the number of attributes in the following descriptor field.

attributes The valid attribute values are Synthetic, Deprecated and Constant-Value. The last is the only one that a JVM is mandated to acknowledge.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUBLIC</td>
<td>can be accessed from outside the package</td>
</tr>
<tr>
<td>PRIVATE</td>
<td>can only be accessed inside the defining class</td>
</tr>
<tr>
<td>PROTECTED</td>
<td>can be accessed inside subclasses</td>
</tr>
<tr>
<td>STATIC</td>
<td>static (class variable, etc.)</td>
</tr>
<tr>
<td>FINAL</td>
<td>no assignment after initialisation permitted</td>
</tr>
<tr>
<td>VOLATILE</td>
<td>cannot be cached (thread annotation)</td>
</tr>
<tr>
<td>TRANSIENT</td>
<td>not to be written or read by a persistent object manager</td>
</tr>
</tbody>
</table>

The JVM specification [33] also defines formats for such things as inner classes. As stated above, there is insufficient space to cover all details of the class file format in this chapter and the interested reader is directed to the JVM specification where all details can be found.

3.4 Object Representation at Runtime

The JVM specification does not impose any special structure on the representation of objects at runtime. JVM implementations are, therefore, free to adopt a representation that is the most convenient for their use.

Some Sun implementations, for example, implement a reference to a class instance as a pointer to a further pair of pointers. One of these pointers is to a table and a pointer to the class object representing the type of the instance. The other pointer is a reference to a data area in heap that is used to store the
instance’s local variables. The table contains the object’s methods (or pointers to them).

The information stored in the constant pool for each class or interface is defined by the specification. When a class or interface is loaded into the JVM, a Runtime Constant Pool is allocated in the method area of the heap for it. The JVM uses the constant pool table in the class file to construct a binary representation when the class or interface is created. The constant pool table in the class file contains symbolic references, as was seen in the last section. The runtime symbolic references in the runtime constant pool are derived from structures in the binary representation of the class or interface in the following ways:

- A symbolic reference to a class or interface is derived from a CONSTANT_Class_info structure. A reference of this type provides the name of the class or interface in the form that is returned by the Java Class.getName method.

- A symbolic reference to a field of a class or interface is derived from a CONSTANT_Fieldref_info structure in the binary class representation. This structure provides the name of the field as well as a descriptor of its contents; in addition it provides a symbolic reference to the class or interface in which it is located.

- A symbolic reference to a method of a class is derived from a CONSTANT_Methodref_info structure. This provides the name and descriptor for the method. It also provides a symbolic reference to the method’s class.

- A symbolic reference to an interface is derived from a CONSTANT_InterfaceMethodref_info structure. This provides the name and descriptor pertaining to the interface method; a symbolic reference to the interface to which the method belongs is also derived from the structure.

Non-reference types are also derived from the information held in a class file. In particular, the following are performed:

- Runtime constants are derived from the constant pool structures: CONSTANT_Integer_info, CONSTANT_Float_info, CONSTANT_Long_info, as well as CONSTANT_Double_info.

- Runtime constants whose values are literal strings are derived from CONSTANT_String_info structures that specify their component Unicode characters; the result is an instance of type String. There are some complications, however, to string derivation; they are:
  - The Java language definition [22] specifies that strings composed of the same sequence of characters should be implemented as references to a single instance of String. This single instance actually contains the characters. If the String.intern method is called on any string, the result is a reference to the same instance of String that would be returned if the string appeared as a source-code literal.
  - In order to derive a string literal, the JVM examines the characters that are provided by the CONSTANT_String_info structure. If String.intern
has previously been called on an instance of String containing an identical sequence of Unicode characters, the result is a reference to that very instance of String. If, on the hand, a new instance of String is created to contain these characters, that instance is the result of string derivation. The String.intern method is then called upon this newly created string.

The final types that can be obtained from the constant pool are CONSTANT_NameAndType_info and CONSTANT_Utf8_info (the class file representation of names, in this case). These structures are used only in an indirect fashion to derive symbolic references to classes, interfaces, methods and fields, as well as during string literal derivation.

It should be remembered that these symbolic references are eventually resolved into actual addresses in the heap. Thus, references become computationally more tractable than the manipulation of indices and complex string-based structures.

Although the process of derivation is specified in [33], the organisation of the method area and the runtime constant pool are not. The precise organisation of the area and the structures that reside in it are implementation-dependent details not specified by the specification.

### 3.5 Initialisation

The creation of a class or interface is performed by creating an implementation-specific internal representation of that class or interface in the method area in the JVM's heap. The process is initiated by some other class or interface referencing the class using its runtime constant pool (which, necessarily, contains a reference to the new class or interface, as the case may be). The invocation of methods can also cause classes and interfaces to be created.

The details of class and interface creation are somewhat complex. They are documented in Chapter 5 of [33] (p. 155 et seq). Suffice it to say that the loading of a class or interface can cause a cascade of class and/or interface loads (those of its supers), if those entities are not already loaded. In addition to the derivation of heap structures, a number of verification procedures can be undertaken, and linkage is performed. Linkage can involve runtime searches up super chains and can fail for a variety of reasons documented in Chapter 5 of [33].

The creation and finalisation (destruction) of class instances is a somewhat simpler process. It is outlined in the remainder of this section.

There are two ways in which an instance of a class might be created: an explicit and an implicit way.

The explicit way relies upon one of the following to occur. Either the evaluation of a class instance creation expression creates a new instance of the class referenced by name in that expression or the invocation of the newInstance method of the Class class creates a new instance of the class represented by the Class object for which that method was invoked.
3.5 Initialisation

The *implicit* way also consists of two cases:

1. The loading of a class or interface containing a literal of type `String` can create a new `String` instance to represent that literal (this has been already encountered above in the last section).

2. The execution of a string concatenation operation not part of a constant expression creates a new `String` instance to represent the result of the concatenation. Temporary wrapper objects can also be created for primitive type values during string concatenation.

When a new class instance is created, the JVM allocates sufficient heap store to contain all the instance variables that are declared in the corresponding class type, as well as for all the instance variables declared by the superclasses of the instance’s class type (this includes *all* of those instance variables that might not be visible). Should there be insufficient space available, the JVM raises an exception (`OutOfMemoryError`); otherwise, the instance variables thus created are initialised to their default values (generally, zero, null or `false`, depending upon type).

Once the allocation and initialisation have been performed and just prior to returning a reference to the newly created instance, the constructor that is indicated by the creation operation is invoked to perform specific initialisations. There are five main steps to be performed:

1. The actual parameters of the call to the constructor are bound to the formal parameters.

2. If this constructor begins with an explicit call to another constructor in the same class (using `this` to denote its location), the arguments are evaluated and these five steps are recursively applied. Should this nested constructor terminate abnormally ("abruptly" in the terminology of [33]), the entire constructor-application process also terminates abnormally. Otherwise, the next step is applied.

3. If this constructor does not begin with a call to another constructor in the same class and the class of this instance is other than `Object`, this constructor executes either an implicit or explicit call to its superclass’ constructor (using `super`). This is another recursive application of these five steps. Otherwise, the next step is applied.

4. The initialisers for the instance variables of the class of this instance are executed in the *left-to-right order* in which they occur in the source text of the class. Should any of these initialisers raise an exception, no further initialisers are executed and this procedure terminates abnormally. Otherwise, the next step is applied.

5. Finally, the rest of the constructor’s body is executed. If this execution terminates abnormally, the entire creation process terminates abnormally (the instance becomes garbage). Otherwise, the procedure terminates normally and a reference to the newly created, initialised instance is returned to the caller.
If, during the execution of a constructor, methods are invoked that are overridden in the instance that is being initialised, the *overriding* methods are called, even if this requires their invocation *before* the new instance is completely created.

### 3.6 Object Deletion

Unlike C++, Java does not provide mechanisms for the explicit cleanup and deletion of objects. However, class *Object* provides a *protected* method called *finalize*—this method can be overridden by other classes (all classes are subclasses of *Object*). The definition of the *finalize* method in any particular class is called the *finalizer* of that class's instances.

Before the storage for an object is reclaimed by the garbage collector, the JVM executes the finalizer on that object so that resources are freed that cannot otherwise be released (e.g., file descriptors and other host system structures that do not reside in the JVM heap). If there were no finalizer, there would be no guarantee that such resources would be released.

The Java language makes no prescriptions as to the time at which a finalizer should be run; it merely states that it will occur before the storage for the object is re-used. The language does not specify the thread in which any finalizer will execute. However, if an uncaught exception is thrown during finalization, the exception is ignored and the object’s finalization terminates.

The actual finalizer defined in *Object* is of little use: it takes no action. The existence of a finalizer in *Object* does guarantee that the finalizer method for any class is always able to invoke the *finalize* method in its superclass. This is usually desirable. The invocation of the finalizer in the superclass must, however, be performed explicitly via a *super* call; the JVM does not automatically invoke the finalizers along the superchain (constructors, however, are always invoked along the superchain).

It should be noted that the *finalize* method can be explicitly called. In this respect, it is just like any other method. Such a call, however, has no effect whatsoever on the object’s eventual automatic finalization.

Finally, the JVM imposes no order on the execution of *finalizer* methods. They can be invoked in any order or even concurrently.

Classes and interfaces can be *unloaded* if and only if its class loader is unreachable. A class loader can be defined and loaded for any class or collection of classes (it is a class) to alter the way in which or the location from which classes are loaded. (The JVM specification [33] contains a detailed description of class loaders and the associated classes.) There is always a default class loader class present in the JVM, so system classes can never be unloaded; most classes use the default class loader and will never be unloaded.
3.7 JVM Termination

The JVM terminates when one of two events occur:

1. All non-daemon threads terminate.
2. A particular thread invokes the `exit` method of the class `Runtime` or the class `System`. This is permitted only when the security manager permits it to occur.

(Daemon threads are threads that are created for internal reasons by the JVM.)

In older versions of the JVM, it was possible to specify that finalizers should be called just prior to system exit. The operation, the `runFinalizersOnExit` in class `System`, is deprecated from Java 2 platform 1.2.

3.8 Exception Handling

This is an interesting feature of the JVM, one whose implementation can be puzzling.

In the JVM, each `catch` or `finally` clause of a method is represented by a range of offsets into the code implementing that method. Each exception handler specifies the range of offsets for which it is active, as well as the type of exception it handles. It also specifies the location of the code that handles that exception. A thrown exception matches an exception handler if the offset of the instruction that caused the exception is in the exception handler’s range of offsets and the type of exception is the same or a subclass of that handled by the handler.

When an exception is thrown, the JVM searches for a matching handler in the current method. If a match is found, the code for that handler is executed. If there is no match, the JVM begins a search. First, the current method invocation terminates abruptly (abnormally), causing the current stack frame to be discarded. The exception is then `rethrown` in the caller’s context. This continues until either the exception is handled or there are no more contexts in which to search. In the latter case, the thread in which the exception was thrown is terminated.

The order in which exception handlers are searched matters. In the class file, the exception handlers for each method are stored in a table (see Section 3.3). At runtime, the JVM searches through the exception handlers in the same order as that in the class file. Java’s `try` commands are structured, so a Java compiler can always order the the exception handler table’s entries in such a way that, for any thrown exception and any value of the `pc` (instruction pointer or program counter) in the method in which the exception is thrown, the first exception handler matching the thrown exception will always correspond to the innermost matching `catch` or `finally` clause.
3.9 Instructions

Instructions are represented by bytecodes. A bytecode consists of an operation field and zero or more operands. The bytecode field is eight bits in length (hence the name); the operands, if present, can be any even multiple of eight bits. A maximum of eight bytes is usually imposed, the maximum represents the number of bits required to hold a floating point number. However, some architectures might require sixty-four bits for an address, as well.

The instruction set can be divided into familiar groups:

- Control instructions;
- Data-manipulation instructions;
- Stack-manipulation instructions.

Control instructions perform transfers of control within a program. They include jump, call and return instructions. Java contains an exception mechanism (the throw and try/finally constructs) and operations to throw and handle exceptions are included within this group (there is a code convention that applies to exceptions to make the location of handlers easier). In the data-manipulation group are included instructions that perform arithmetic and logical operations, bit manipulation and so on. The stack-manipulation instructions access and store variables in the local variable array, as well as operating on the stack (swap and dup operations, for example).

The data and stack manipulating instructions must operate on values whose size ranges from eight to sixty-four bits. The runtime storage requirements for each primitive (source-language) type is as shown in Table 3.4.

It can be seen that types requiring fewer than the standard 32 bits are stored in a full 32-bit word at runtime. This is reasonable when it is considered that the local stack elements and the elements of the local variable array are all 32 bits wide. The only types longer than 32 bits are, as noted above (Section 3.2.1), are long (integer) and floating point (double). The float type is only 32 bits wide.

<table>
<thead>
<tr>
<th>Type</th>
<th>Size (in 32-bit units)</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>1</td>
</tr>
<tr>
<td>character</td>
<td>1</td>
</tr>
<tr>
<td>short integer</td>
<td>1</td>
</tr>
<tr>
<td>integer</td>
<td>1</td>
</tr>
<tr>
<td>long integer</td>
<td>2</td>
</tr>
<tr>
<td>float</td>
<td>1</td>
</tr>
<tr>
<td>return address</td>
<td>1</td>
</tr>
<tr>
<td>double</td>
<td>2</td>
</tr>
</tbody>
</table>
The instruction set also contains instructions related to object-oriented programming languages. These instructions perform such operations as:

- Creation of class instances;
- Invocation of methods;
- Access and update of instance and class variables.

The second and third of these groups include instructions whose semantics includes search along the superclass chain to locate the methods and variables addressed by the instructions.

Finally, the instruction set has instructions that support threads and monitors. Under this heading are monitor entry and exit operations, as well as operations for handling thread groups.

The JVM specification [33] does not define how thread scheduling should be performed. It does, however, specify the behaviour of the locking operations required to ensure correct access to shared data inside monitors.

When describing instruction sets, there is always the risk that the result will just be a long list, something that is to be avoided. Therefore, rather than describe all of the JVM’s instructions, a few interesting ones will be described in detail, while the remainder are summarised and the interesting points addressed.

Before moving on, it is important to observe that a great many JVM instructions encode type information. Thus, what would be a single data movement and arithmetic instruction on another virtual machine is implemented as a set of type-specific instructions in the JVM. For example, the addition instruction comes in the following forms:

- iadd – integer addition;
- ladd – long addition;
- fadd – (single-precision) floating-point addition;
- dadd – (double-precision) floating-point addition.

There is similar replication for other operations. Types are often encoded in an instruction by a single letter:

- i for integer;
- l for long;
- s for short;
- b for byte;
- c for character;
- f for (single-precision) floating-point (float);
- d for (double-precision) floating-point (double);
- a for reference.

Note that the JVM supports a reference type, written reference, below. This type represents references to objects (including arrays). The JVM also supports an address type for code, written returnAddress.
In the following summaries, where an instruction has many typed forms, its general name will have the letter ‘T’ as a prefix. For example, the addition instruction would be written ‘Tadd’.

### 3.9.1 Data-manipulation instructions

The arithmetic instructions are: addition (Tadd), subtraction (Tsub), multiplication (Tmul), division (Tdiv), remainder (Trem) and negation (change of sign—Tneg). Each of these instructions expects its arguments to be on the top of the local operand stack. They pop them, perform the operation and push the result back onto the stack.

The following are restricted to integer and long operands:

- `ishl` `ish` Left shift;
- `ishr` `ishr` Arithmetic Right shift;
- `iushr` `lushr` Logical Right shift.

These instructions expect the stack to be:

```
... value1, value2
```

where `value1` is the mask to be shifted and `value2` is the amount to shift. For integer shifts, both operands should be integers. For long shifts, the shift should be an integer and the mask should be long.

The logical operations are the following:

- `iand` `land` Logical “and”;
- `ior` `lor` Logical “or”;
- `ixor` `lxor` Logical “xor” (exclusive or).

They all expect their operands to be on the top of the operand stack. The result is pushed back onto the stack.

The logical operations provide an opportunity for explaining how the JVM represents values that require fewer than 32 bits (byte, characters, short integers and logical values). It must be noted that there is a bit string class in the Java library, so arbitrary bit strings can be handled in ways other than “twiddling bits”.

Quite simply, the JVM represents all smaller types as 32-bit quantities, so the operands to the logical operations just listed should be integers.

In some cases, this requires a proper conversion, while, in others, it just requires a truncation. Some conversions lose information. Sometimes, the sign of the result (Java integers are always signed—there is no unsigned int at present) might not have the same sign as the input.

Since there are type-specific operations for manipulating data, it is clearly necessary to have type-changing instructions. The instructions are listed. Only some of the conversions are explained. The reason for this is that conversions between floating point (both single- and double-length) are somewhat complex.

The type-conversion instructions are as follows:
3.9 Instructions

Integer to $T$: The instruction expects a single operand to be on the top of the stack. The operand should be an integer. The result is pushed onto the stack.

- **i2b**: Integer to byte. The operand is truncated, then sign-extended to integer. (The result might not have the same sign as the operand. Information can be lost.)
- **i2c**: Integer to character. The operand is truncated, then zero-extended to integer. (The result might not have the same sign as the operand. Information can be lost.)
- **i2s**: Integer to short. The operand is truncated, then sign-extended to an integer. Information might be lost. The sign of the result is not always the same as the operand.
- **i2l**: Integer to long. The operand is sign-extended to a long. This is an exact operation.
- **i2f**: Integer to (single-length) floating point. There might be a loss of precision because single-length floats only occupy 24 bits.
- **i2d**: Integer to (double-length) floating point. No information is lost.

Long to $T$: The instruction expects a single operand to be on the top of the stack. The operand should be a long. The result is pushed onto the stack.

- **l2i**: Long to integer.
- **l2f**: Long to (single-length) floating point.
- **l2d**: Long to (double-length) floating point.

(Single) Float to $T$: The instruction expects a single operand to be on the top of the stack. The operand should be a (single-length) float (float). The result is pushed onto the stack.

- **f2i**: Float to integer.
- **f2l**: Float to long.
- **f2d**: Float to double.

Double to $T$: The instruction expects a single operand to be on the top of the stack. The operand should be a (double-length) float (double). The result is pushed onto the stack.

- **d2i**: Double to integer.
- **d2l**: Double to long.
- **d2f**: Double to (single-length) floating point.

**lcmp** Compare long. Both operands should be long. It expects the stack to be:

... $value_1$, $value_2$

The two values are compared. The result is pushed onto the stack. The result is computed as follows:

- $value_1 > value_2$ Result 1
- $value_1 = value_2$ Result 0
- $value_1 < value_2$ Result 1
Tcmp1 (T can be f or d.) Compare operands. Both operands are expected on
the stack and should be of the same type.

Tcmpg (T can be f or d.) Compare operands. Both operands are expected on
the stack and should be of the same type.

The Tcmp1 and Tcmpg instructions differ only in the way in which they handle
the NaN value defined by the IEEE 754 floating point standard. The way in
which these instructions, often referred to as fcmp<op>, work is as follows.
First, the stack should be of the form:

...value1, value2

A floating point comparison is performed after they have been popped from
the stack. A value-set conversion is performed immediately prior to the com­
parison. The results of the value-set conversion are denoted \( v'_1 \) and \( v'_2 \). The
result (an integer) is computed as follows:

- \( v'_1 > v'_2 \), the result is int 1;
- \( v'_1 = v'_2 \), the result is int 0;
- \( v'_1 < v'_2 \), the result is int -1;
- If at least one of \( v'_1 \) and \( v'_2 \) is NaN, the result depends upon which instruction
  is being executed:
  - Tcmpg The result is int 1;
  - Tcmp1 The result is int -1.

The instructions consider +0 = −0.

<table>
<thead>
<tr>
<th>iinc opcode</th>
<th>index</th>
<th>const</th>
</tr>
</thead>
</table>

Fig. 3.1. The JVM iinc instruction format.

Finally, it is extremely useful to be able to increment (and sometimes
decrement) a register or memory location in one instruction. The JVM has
exactly one instruction for this: iinc. The instruction has the format shown in
Figure 3.1. The format is similar to that employed in other cases: first, there
is the symbolic opcode, followed by the operands. In this case, the opcode is
iinc, the first operand is index and the second operand is const. The index
is an unsigned byte; this must be a valid index into the local variable array
in the current stack frame; that local variable must contain an integer value.
The const is a signed byte, which is first sign-extended and then added to the
local variable at index.
3.9.2 Control instructions

The following instructions implement unconditional transfer of control:

goto This is a transfer of control whose operand is a 16-bit offset which is added to the address of the goto instruction’s opcode to produce the destination address in the current method’s code. Control is transferred to the destination address.

goto.w This instruction has an opcode, followed by four bytes. The four bytes are shifted and or’ed to produce an offset into the current method’s code. The destination address of the jump is computed by adding the offset to the address of the instruction’s opcode. There is an a priori limit to the size of a method’s code of 65535 bytes (this is for “historical reasons”).

jsr This instruction is the subroutine call instruction. It consists of an opcode and two bytes, the latter comprising a signed 16-bit offset formed by shifting and or-ing the two bytes. The address is pushed onto the stack as a value of type returnAddress. Control is transferred to that address from the address of (the opcode of) the jsr instruction. The target of the transfer must be the opcode of an instruction.

jsr.w This is a second subroutine call instruction. It has a four-byte offset. The destination is constructed by shifting and or-ing to form a 32-bit signed offset. The offset must respect the maximum method code length. The jsr.w and ret instructions are used to implement finally clauses.

ret This instruction has a single (unsigned) byte as its operand. The operand is used to index the local variable array in the current stack frame. The element of the local variable array thus referenced must contain a value of type returnAddress. This value is copied to the JVM’s pc register to perform a transfer of control.

return The return instruction is used when the current method’s return type is void. It has no operands. When executed, the instruction returns control to the method that called the one in whose code this instruction occurs. It reinstates the caller’s stack frame, discarding the current one. If the method in whose code this instruction occurs is declared synchronized, monitor locks must be released. Exceptions can be thrown, causing failure of this instruction.

Treturn Returns a value of type $T$ from the current method ($T$ can be $i$, $l$, $f$, $d$ or $a$—recall that types requiring fewer than 32-bits are converted to a 32-bit representation). The value to be returned must be on top of the local operand stack; it should be of the appropriate type. If the current method is declared synchronized, monitor locks are released. The instruction pushes the value to be returned onto the local stack of the method that called the current one. Control is returned to the caller by the JVM; the caller’s stack frame is reinstated. Exceptions can be thrown, causing failure of this instruction.

The JVM supports a set of conditional branch instructions. They have the general name if<cond> and a common format. The form that each of these
instructions takes is shown in Figure 3.2. The first element is the one-byte opcode. There follow the two bytes that constitute the destination offset.

The instruction pops the top element from the local stack and performs a test on it. If the test succeeds, the destination offset is constructed in the usual way and added to the address of the if instruction’s opcode to form a new offset into the current method’s code.

If the value popped from the stack is written as \( v \), the forms for if can be summarised as:

- ifeq — if \( v = 0 \), control transfers to the destination (\( v \) must be an integer);
- iflt — if \( v < 0 \), control transfers to the destination (\( v \) must be an integer);
- ifle — if \( v \leq 0 \), control transfers to the destination (\( v \) must be an integer);
- ifne — if \( v \neq 0 \), control transfers to the destination (\( v \) must be an integer);
- ifgt — if \( v > 0 \), control transfers to the destination (\( v \) must be an integer);
- ifge — if \( v \geq 0 \), control transfers to the destination (\( v \) must be an integer);
- ifnull — if \( v \) is equal to null, control transfers to the destination (\( v \) must be of type reference);
- ifnonnull — if \( v \) is not equal to null, control transfers to the destination (\( v \) must be of type reference).

If the test fails, the instruction immediately following the if<cond> is executed.

The following generic instructions have a three-byte format: opcode followed by two address bytes. If the test succeeds, the two bytes are used to construct a signed 16-bit offset into the current method’s code. In both cases, the stack should have the form:

\[ \ldots \ v_1, \ v_2 \]

These values are popped from the stack. For if.icmpOP, these two values should be integers, while if.acmpOP expects them both to be of type reference.

if.icmpOP Branch if an int comparison succeeds. The values taken by OP are:

- eq If \( v_1 = v_2 \), the branch is executed.
- ne If \( v_1 \neq v_2 \), the branch is executed.
- lt If \( v_1 < v_2 \), the branch is executed.
- le If \( v_1 \leq v_2 \), the branch is executed.
- gt If \( v_1 > v_2 \), the branch is executed.
- ge If \( v_1 \geq v_2 \), the branch is executed.
if.acmpOP Branch if a reference comparison succeeds. The values taken by OP are:

- **eq** If $v_1 = v_2$, the branch is executed.
- **ne** If $v_1 \neq v_2$, the branch is executed.

<table>
<thead>
<tr>
<th>tableswitch (opcode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-3 byte padding</td>
</tr>
<tr>
<td>defaultbyte1</td>
</tr>
<tr>
<td>defaultbyte2</td>
</tr>
<tr>
<td>defaultbyte3</td>
</tr>
<tr>
<td>defaultbyte4</td>
</tr>
<tr>
<td>lowbyte1</td>
</tr>
<tr>
<td>lowbyte2</td>
</tr>
<tr>
<td>lowbyte3</td>
</tr>
<tr>
<td>lowbyte4</td>
</tr>
<tr>
<td>highbyte1</td>
</tr>
<tr>
<td>highbyte2</td>
</tr>
<tr>
<td>highbyte3</td>
</tr>
<tr>
<td>highbyte4</td>
</tr>
<tr>
<td>jump offsets ...</td>
</tr>
</tbody>
</table>

Fig. 3.3. The **tableswitch instruction format**.

<table>
<thead>
<tr>
<th>lookupswitch (opcode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-3 byte padding</td>
</tr>
<tr>
<td>defaultbyte1</td>
</tr>
<tr>
<td>defaultbyte2</td>
</tr>
<tr>
<td>defaultbyte3</td>
</tr>
<tr>
<td>defaultbyte4</td>
</tr>
<tr>
<td>npairs1</td>
</tr>
<tr>
<td>npairs2</td>
</tr>
<tr>
<td>npairs3</td>
</tr>
<tr>
<td>npairs4</td>
</tr>
<tr>
<td>match-offset pairs ...</td>
</tr>
</tbody>
</table>

Fig. 3.4. The **lookupswitch instruction format**.

The next pair of instructions are used to implement switch commands in Java source. Both instructions are of variable length. In both cases, the instruction is padded by up to three bytes; these bytes should be filled with zero. The padding is required to ensure that 4-byte alignments are maintained. The *defaultbytes* are used to construct a signed 32-bit value.
tables\text{\textbf{w\textit{h}}\textit{\textit{i}}\textit{t\textit{c}}} The format of this instruction is shown in Figure 3.3. The \textit{low\textbf{\textit{b\textit{ytes}}} and \textit{high\textbf{\textit{b\textit{ytes}}} are used to form two 32-bit values, referred to as \textit{low} and \textit{high}, respectively. The bytes indicated by \textit{jump offsets} in Figure 3.3 represent 32-bit values representing \textit{high} – \textit{low} + 1 offsets, the offsets into the jump table. Note that \textit{low} \leq \textit{high}.

The instruction expects an integer value to be on top of the stack. This value is popped; it is the index into the table. If the index is less than \textit{low} or greater than \textit{high}, a destination address is computed by adding the \textit{default} to the address of the opcode of this instruction. Otherwise, the destination is computed by subtracting \textit{low} from index and, using this as the offset of a \textit{jump offset}. This second offset is added to the address of the \textit{tables\text{\textbf{w\textit{h\textit{t\textit{c}}}}} instruction to produce the address to which to jump, thus causing transfer to one of the non-default cases.

\textit{lookup\text{\textbf{w\textit{h\textit{t\textit{c}}}}} The format of this instruction is shown in Figure 3.4. The \textit{match\textbf{-offset pairs} must be sorted in increasing numerical order by \textit{match}. A value, called \textit{key}, is expected to be on top of the local stack; it must be of type integer. The value of \textit{key} is compared with the \textit{match} values and, if equal to one of them, a destination address is computed by adding the corresponding \textit{offset} to the address of the opcode of this instruction. If there is no match, the destination is computed by adding the \textit{default} to the address of the opcode of this instruction. Execution continues at the address thus computed.

\textit{ath\text{\textbf{h\textit{\textit{\textit{\textit{r}}}}} This instruction expects its operand to be on top of the local stack. The operand must be of type \textit{reference} and, therefore, should refer to an object. This object should be of type \textit{Throwable} (or a subclass thereof). The object reference is then thrown. This is done by searching in the local code for a matching exception handler. If a handler is found, its code is executed; otherwise, a search for a suitable handler is begun.

### 3.9.3 Stack-manipulating instructions

The JVM requires instructions to represent constant values. The interpretation of these instructions is that they have a literal operand which is pushed onto the operand stack. There are constant instructions for integer (\textit{i\text{\textbf{\textit{c\textit{onst})}}, long (\textit{l\text{\textbf{\textit{c\textit{onst})}}, float (\textit{f\text{\textbf{\textit{c\textit{onst})}}, double (\textit{d\text{\textbf{\textit{c\textit{onst}) and reference (\textit{a\text{\textbf{\textit{c\textit{onst}}.\n
In addition, there are the following constant operations:

- \textit{b\text{\textbf{ipush}} Push the operand (a byte) onto the stack.
- \textit{s\text{\textbf{ipush}} Push the operand (a \textit{short} formed by or-ing the two operands) onto the stack.
- \textit{\text{\textbf{l\textit{d\text{\textbf{c}}} This instruction consists of the opcode followed by an operand, \textit{index}. This is an unsigned byte representing an offset into the runtime constant pool of the current class. The entity at that location in the constant pool must be of type \textit{int}, \textit{float} or a symbolic reference to a string literal. If the value is an \textit{int} or a \textit{float}, the value is pushed onto the local stack.
Otherwise, the constant pool entry must be a reference to an instance of \texttt{String}; that reference is pushed onto the local stack.

\texttt{1dc.w} This is the same as \texttt{1dc} but has a 16-bit index (represented by operand bytes following the opcode in store).

\texttt{1dc2.w} This is similar to \texttt{1dc} but loads \texttt{long} or \texttt{double} onto the stack.

\texttt{aconstnull} Push \texttt{null} onto the stack.

\texttt{iconst.<i>} This is a family of instructions: \texttt{iconst.-1} to \texttt{iconst.5}. They push the (integer) value indicated after the underscore onto the stack.

\texttt{1const.<1>} Similar to \texttt{iconst} but pushes a \texttt{long}.

\texttt{fconst.<f>} Similar to \texttt{iconst} but pushes a \texttt{float}.

\texttt{dconst.<d>} Similar to \texttt{iconst} but pushes a \texttt{double}.

\texttt{Tload} A family of instructions, each composed of a single byte opcode followed by a single byte that must be a valid index into the local variable array. The value located at that index is pushed onto the stack. The type of the value pushed is indicated by \texttt{T}.

\texttt{Tload.<n>} A family of instructions, each composed only of an opcode. The operand is encoded in the instruction; it is used as an index into the local variable array. The value stored at that index is pushed onto the stack. The type of the value pushed is indicated by \texttt{T}.

\texttt{Tstore} This is a family of two-byte instructions: the first byte is the opcode, the second an index into the local variable array. The stack is popped to yield a value that is stored in the local variable array at the indicated index. The type of the value popped is indicated by \texttt{T}.

\texttt{Tstore.<n>} A family of instructions that encode their operand in the opcode. The operand is an offset into the local variable array. The value on the stack is popped and stored into the local variable array at the indicated index. The type of the value popped is indicated by \texttt{T}.

The \texttt{wide} instruction is a complicated instruction, so the reader should consult the description in [33], pp. 360–1. This instruction modifies the behaviour of other instructions. It takes one of two formats, the actual one depending upon the particular instruction being modified. Its first format is used for: \texttt{Tload}, \texttt{Tstore} and \texttt{ret}. Its second format is used for \texttt{iinc} only. The instruction constructs a 16-bit signed offset into the local variable array. The effect of this instruction is to widen the index of its target opcode when the first format is employed. The second format widens the range of the \texttt{iinc} instruction it targets. The instruction that is thus modified must not be the target of a control transfer.

The following instructions support direct manipulation of the stack.

\texttt{pop} Pop the local stack.

\texttt{pop2} Pop the top two values from the local stack.

\texttt{dup} Push a copy of the top stack element onto the stack. This instruction is used only when the top element is 32 bits wide.
**dup2** Push a copy of the top stack element onto the stack. This instruction is used only when the top element is 64 bits wide.

**dup.x1** Transform the stack as follows. Given:

\[ \ldots \ v_2, \ v_1 \]

change it to:

\[ \ldots \ v_1, \ v_2, \ v_1 \]

This instruction operates when the \( v_i \) are 32-bit quantities.

**dup.x2** Transform the stack as follows. Given:

\[ \ldots \ v_3, \ v_2, \ v_1 \]

change it to:

\[ \ldots \ v_1, \ v_3, \ v_2, \ v_1 \]

if all the \( v_i \) are 32 bits wide. If \( v_1 \) is 32 bits wide but \( v_2 \) is 64 bits, then the stack should be transformed into:

\[ \ldots \ v_1, \ v_2, \ v_1 \]

**swap** This swaps the top two stack elements.

The **dup2.x1** and **dup2.x2** instructions are variations on **dup.x1** and **dup.x2** (see [33], pp. 222–4, for details).

### 3.9.4 Support for object orientation

The **new** instruction creates a reference to a new object on the stack. The instruction is three bytes in length, the second and third forming an index into the constant pool for the current class. The entry thus indexed should be a symbolic reference to a class, interface or array type. The new instance is created in the heap and the reference to it returned to the instruction to be pushed onto the local stack.

**newarray** Create a new array. The length of the array is expected to be on the top of the local operand stack; it must be an integer (it is popped by the instruction). The one-byte operand following this instruction’s opcode denotes the type of the array to be created. A reference to the newly created array is pushed onto the local stack.

**anewarray** Create a new array of reference (i.e., whose elements are of type reference). See [33], p. 181, for details.

**multianewarray** This instruction creates a multi-dimensional array. (See [33], pp. 339–340 for details.)

The four field access instructions have a common format, shown in Figure 3.5. The opcode is followed by two bytes that, together, form an index into the constant pool of the current class. The element at that index should be a symbolic reference to a field. This reference yields the name and type of the field, as well as a symbolic reference to the class in which it is located.
getfield This operation obtains the value stored in an object’s field and pushes it onto the stack. It expects a reference to an object to be on top of the local stack. It pops that reference and uses the index bytes to access the field.

getstatic This operation is similar to getfield but requires the field it accesses to be a static field of the object. The stack only holds the value to be obtained from the static field (there is no object reference).

putfield This operation expects the stack to have, as its top element, the value to be stored. Immediately beneath, it expects to find a reference to the object in which the value is to be stored. The index is used to resolve the field; checks are made to ensure that the assignment is legal (access and type checks are made).

putstatic This is similar to putfield but operates on static fields. The stack is expected only to hold the value to be stored. Checks are again made to ensure that the assignment is permitted and of the correct type.

There is a collection of instructions dedicated to arrays, the most general of which are (again, using the convention described above).

Taload (T can be b, s, i, l, f, d, c or a). These instructions expect a valid array index to be on top of the local stack; immediately beneath this should be a reference to an array. These operands are popped from the stack. The array is indexed using the popped value and the element at that index is pushed onto the stack.

Tastore (T can be b, s, i, l, f, d, c or a). This stores a value in an array. The instruction expects three operands to be on the stack: the value to be stored, an index into the array and a reference to the array. All three values are popped from the stack.

arraylength Returns the length of an array.

There are some general object-oriented instructions:

instanceof This instruction determines whether an object is of a given type. It expects the object to be referenced by the top element of the local stack (which it pops). The two bytes following the opcode form an index into the constant pool for the current class, where it should index a symbolic reference to a class, array or interface type.

checkcast Another complex instruction ([33], pp. 193-4). It verifies that the object referenced by the top stack element is of the given type. The two bytes following the opcode are a constant pool index that should refer to
a symbolic reference to a class, array or interface type. The instruction pops the object reference from the stack.

![opcode]
![indexbyte1]
![indexbyte2]

**Fig. 3.6.** Format of the JVM invoke instructions (except invokeinterface).

![invokeinterface]
![indexbyte1]
![indexbyte2]
![count]
![0]

**Fig. 3.7.** Format of the invokeinterface instruction.

The following instructions invoke methods in various places. They are all complex operations, so the reader is strongly urged to read the descriptions in [33]; what follows is just an indicative account.

The first three have a common format that is depicted in Figure 3.6. The last of the group has the format shown in Figure 3.7. The indexbytes are used to construct an index into the runtime constant pool of the current class. The entity thus addressed must be a symbolic reference to a method of the appropriate kind.

**invokespecial** Call an instance method on an instance; special handling for superclass, private and instance initialisation method calls. The operation expects the arguments to the method to be on the local operand stack as well as a reference to the object to which the method belongs.

**invokevirtual** Invoke a method using dynamic dispatch. That is, invoke an instance method. The actual method being dispatched depends upon the class referred to under the arguments on the stack.

**invokestatic** Similar to invokevirtual but the method must be declared static.

**invokeinterface** Similar to invokestatic but the method belongs to an interface.
3.9.5 Synchronisation

There are two synchronisation instructions:

- `monitorenter` Enter a monitor;
- `monitorexit` Leave a monitor.

The functioning of these instructions is complex and the interested reader should consult all the relevant sections of [33].

3.10 Concluding Remarks

In this chapter, the Java Virtual Machine (JVM) has been reviewed. The JVM’s overall structure consists of:

- A heap region;
- A stack.

The heap region contains the code to be executed by the JVM, as well as a special Constant Pool data structure. The constant pool holds, at runtime, information about the classes that have been loaded into the JVM. The stack is of the framed variety and is also allocated in the heap.

One important aspect of the heap is its storage of class file structures in the constant pools. It is the class file that contains information about each class that has been loaded. The overall organisation of this structure and some of its uses have been described in this chapter.

The JVM executes instructions in the usual way. The instructions in the JVM are aligned on byte boundaries. Instructions, called bytecodes, are stored in the method code vectors located in class files. The instructions executed by the JVM can be divided into two main classes:

1. Simple instructions. In this class are instructions such as jumps (the JVM supports conditional as well as unconditional jumps) and arithmetic instructions. The arithmetic instructions are typed: there are arithmetic instructions for the main numeric types.
2. Complex (or High-Level) instructions. In this class are instructions such as those for the allocation of class instances and arrays, accessing arrays, accessing and updating class and instance variables, throwing and executing exceptions, and instructions for invoking methods of all kinds.

In addition to this Java functionality, the JVM also supports so-called “C” stacks. These stacks allow methods implemented in native code (code that executes directly on the host machine) to be integrated with code implemented as JVM bytecodes.