Microsoft. The Microsoft *shared-source CLI,* despite the name, is another example of a CLI implementation (*shared-source CLR* would be more appropriate). Sometimes, it is useful to discuss only the instruction set portion of a VM architecture. In the case of Java, it is generally referred to as *Java bytecodes* because, as we shall see, the instructions can be viewed as a stream of bytes. The instruction part of the CLI is known variously as the *Microsoft intermediate language* (MSIL), the *common intermediate language* (CIL), or simply the IL. The combination of a JVM implementation and a set of standard Java libraries (APIs) is called a *Java platform.* Finally, a CLI implementation, along with a set of standard libraries, implements the Microsoft .NET framework.

5.2.1 Security and Protection

A given application running on a local platform should have access to certain specified files (program and data) on both its local system and on remote systems. Meanwhile, there must be the ability to prevent the guest application from accessing other files on both the remote and local systems. In addition, the VM implementation software that is running the application should be protected from the application itself, even though both are running on the host platform as part of the same process. Therefore, the type of protection that must be provided is different from that supported by a conventional operating system, where users are protected from one another, but a given user's programs have access to all of that user's resources. In the HLL environment, untrusted programs are invited into the user's domain but must have strictly limited access within that domain. That is, a guest program is treated as managed code that can only operate within its protection sandbox. In modern HLL VMs such as Java, the VM implementation provides much of this additional protection "above" the operating system. As we shall see, a fundamental aspect of the popular HLL VMs is that the high-level languages (e.g., Java or C#) are strongly typed and legal programs must strictly define the scope of both data accesses (to objects) and

must strictly define the scope of both data accesses (to objects) and control flow. This information is then conveyed to the VM through the V-ISA.

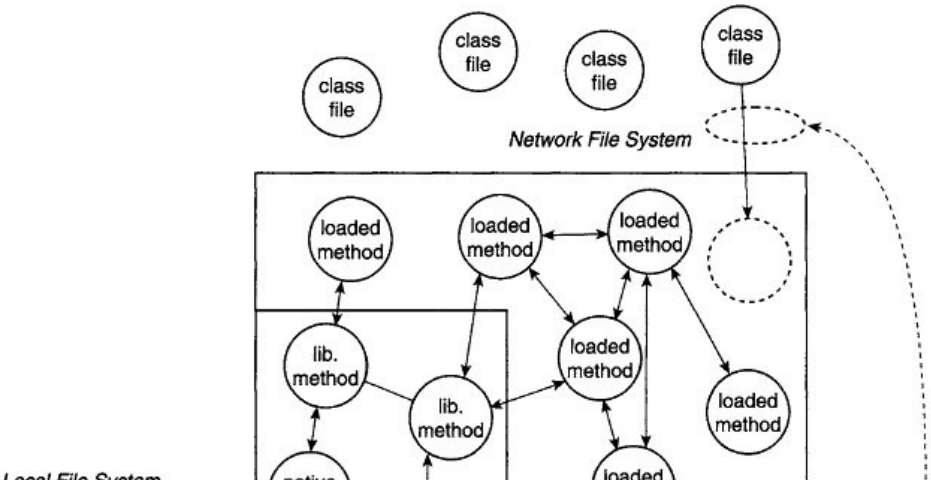
There are three main aspects to the overall security/protection sandbox. The first is gaining access to public data, e.g., stored in files that are on remote systems, while being prevented from accessing any other data on the remote system. Here, we include locating public files as part of the overall process. The second aspect is that similar properties must hold for local files. And the third aspect is that the managed application must be prevented from accessing memory or executing code that is outside the sandbox, even though it shares the process memory space with the VM runtime software. To simplify explanation of the basic principles, the following discussion is based on the original Java sandbox model. The original sandbox is relatively inflexible and was primarily targeted at *applets*—small application programs accessed via the Internet; it remains the default security model. The more recent security model (used in Java 2) provides greater flexibility, including finer-grained access control and significantly less reliance on trusted code. This model will receive more discussion in **Section 6.2**.

Remote files are found via agreed-upon naming conventions. A specific convention is not part of a standard *per se*, but most current operating systems, including Linux and Windows, use hierarchical file systems based on directories. Consequently, naming conventions typically rely on this hierarchical structure. Furthermore, the Internet connects these systems and also uses a hierarchical naming system. Hence, as part of an overall framework—whether a Java platform or the .NET framework—there is normally a naming convention based on a combination of Internet addresses and local file structure addresses. For example, in the Java platform a particular method might be denoted as

```
edu.wisc.ece.jes.testpackage.shape.area
```

which is formed as a concatenation of an Internet address (in reverse order)—**ece.wisc.edu**—and a pathname—~jes/testpackage/shape/area. The critical thing is that a local system, given a name that follows the convention, must be able to find the stored representation of the requested item.

Protection of resources within the remote system's domain is not really a responsibility of the HLL VM running on the local system, even though data on the remote system may be part of an overall network computing framework. It is up to the user(s) of the remote system to determine which data files and/or directories should be made accessible over the network and which permissions (read or write) should be granted. Hence, even if the local system gives a correct path, if the file is not publicly accessible, then an attempt to access it will be denied permission. As noted earlier, this simple protection method is only the Java default. By using security, authentication, and cryptography APIs, a much more flexible security infrastructure can be built and supported by HLL VM platforms; for example, the Java 2 Standard Edition platform contains a number of APIs to support network-based security (**Gong, Ellison, and Dageforde 2003**). However, such VM platform security involves a number of higher-level issues that are well beyond our scope, although there is some additional discussion in the next chapter.



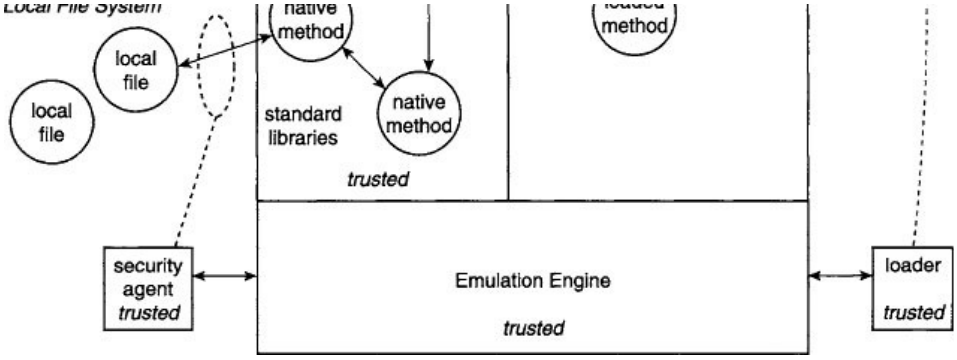


Figure 5.4 Components of the Java Protection Sandbox.

As shown in **Figure 5.4**, a Java program consists of a number of binary classes (or *modules* in .NET terminology), in a standard format, consisting of both platform-independent code and metadata. These binary classes may be held in a local file system, or they may be accessible via a network. The VM implementation includes a loader (or set of loaders) able to take a binary class, verify that it is correct, and then translate it into an implementation-dependent format that is loaded into the VM's memory region. The verification process will be described in more detail later. A loader is *trusted* software; that is, the local user trusts it to work correctly and to be secure. The trusted loader may be supplied by the user or, more likely, by a software developer trusted by the user. The binary classes define the loaded methods and object types, which are operated on by the methods.

There are both user methods and system methods, some of which may be native methods. The application program contains user methods. System methods belong to standard libraries (APIs) that are part of the local VM implementation and are often handled as trusted software.[3]

Native methods are maintained as native binary code, as the name suggests. Native methods can be compiled from any of a number of high-level languages, including C, or assembly language. They are useful for interfacing with legacy code and features of the underlying host platform, such as OS calls. They can also be used by pro-

grams written in other languages to draw on the rich set of APIs implemented in Java. The HLL methods and native methods can interact by following established conventions at their respective interfaces; the Java/native interface (JNI) defines a convention for Java. Native methods are often incorporated as parts of trusted libraries, but a user may also write and install native methods (at his or her own risk because they are not subject to the same security checks as conventional methods).

The emulation engine, another trusted component, emulates the V-ISA code embodied in methods using techniques similar to those described in **Chapters 2** through **4**. That is, it may use interpretation or may execute code that has been translated (compiled) to native instructions. If a managed application program needs to access a local file or any other local resource, it can only do so through a security manager within which the user has either directly or indirectly provided permissions (**Section 6.2**). Whenever the managed application requests access to a local file, for example, the request is made through a method belonging to a standard library. The standard library method consults the security manager to see if permissions should be granted. If so, the file access proceeds.

The managed application code includes software in the form of actual binary classes and methods they contain. This untrusted application software and the VM software share the same address space and run at the same privilege level within the host system; consequently, the VM software must be protected from the application as it executes. For example, arbitrary access to the VM tables via application program load and store instructions must be prohibited. Similarly, the application code should not be allowed to jump to arbitrary locations in the VM code.

A combination of static and run-time checks implement address space protection. As the emulation engine executes a managed guest application, it can perform a memory bounds check on every load

application, it can perform a memory bounds check on every load, store, or jump instruction that static (load-time) code analysis cannot guarantee to be safe. These checks ensure that the guest only accesses its portion of the address space; i.e., it stays within the sandbox. Today's more popular HLL VMs, including both the JVM and CLI, are targeted at strongly typed HLLs and rely to a high degree on static (load-time) checking for protection, with minimal run-time checking. Because the strongly typed HLL approach is widely used, it is the one we focus on. In contrast, with conventional binaries, as would be generated by C or C++, static binary code analysis is rather limited, and many loads, stores, and jumps would require dynamic run-time checking. An especially efficient implementation of this full dynamic-checking approach simply forces all accesses to be within predefined, easily checked guest memory segments (**Wahbe et al. 1993**). This technique allows virtually any HLL to be used for generating guest programs. Such an approach is taken in the Omniware system (**Lucco, Sharp, and Wahbe 1995**).

We provide more detail in the next chapter when discussing Java virtual machine implementation, but the basic approach to checking is first to restrict the virtual ISA so that all the data structures are defined as part of the metadata and all memory accesses are to fixed fields within typed structures. The metadata, of course, reflects the object structure defined in the high-level language program. Similarly, all branch instructions are to fixed offsets within the code region. The only indirection allowed for control transfers is through explicit call and return instructions. As part of the loading process, the load and store instructions in the machine-independent code are checked for consistency with respect to the metadata specifications. The loader also checks all control flow (branch and jump) instructions for consistency. These load-time static checks provide most of the protection against out-of-bounds loads, stores, and control transfers. Exceptions are for those accesses that use a computed index value into a data structure, e.g., arrays. These accesses are checked dynamically at run time, along with checks for accesses us-

ing null pointers (actually *references,* in Java terminology).

In summary, we see that the sandbox around a managed execution environment consists of a number of interacting components. The files and other resources on a remote system are protected by the remote system itself, based on its conventional protection mechanisms. The files and other resources on the local machine are protected by trusted libraries and a security manager. Finally, the VM runtime software is protected from untrusted application software by a combination of static checking performed by a trusted loader and dynamic checks performed by a trusted emulation engine.

5.2.2 Robustness—Object-Oriented Programming

As noted earlier, the two most widely used HLL VMs are both based on an object-oriented model. We can't possibly cover all the aspects of object-oriented programming here. Rather, we provide a quick overview to introduce terminology and to make the following discussion of HLL VM implementations easier to understand.

In an object-oriented programming environment, the data-carrying entities are *objects.* The data contained in each object can only be accessed and manipulated via *methods,* or procedures, that are defined for that type of object. A *class* defines a type of object and its associated methods. At run time, objects may be created as *instances* of the class. In essence, the objects are a complex programmer-defined data type, and the methods are specific to the particular type for which they are defined. Objects can only be accessed via managed pointers, or *references,* that are generated at the time the object is created. In Java and CLI, *arrays* are defined as part of the language and are essentially an intrinsic form of object.

The data portion of an object consists of a number of fields. These fields may be individual data elements, or they may be references to

fields may be individual data elements, or they may be references to other objects and arrays. The data portion of an object can be either *static* or *dynamic.* If it is static, then only one copy of the associated data is created, and it is shared by all objects of the given type. If data is dynamic, then a new copy of the data is generated for each object that is created.

Through *inheritance,* new classes can be defined by extending an existing class. A derived class has the same members (e.g., fields) as the base class from which it is inheriting. When a class is *extended,* the function of an existing method for the base class can be overridden, by defining a new method having the same name and *signature.* A method's signature is an ordered list of its argument and return types. The new, overriding method is the one applied to the subclass. Furthermore, a subclass can have other members and methods that do not exist in the base class.

Inheritance, as just described, is a way of achieving *polymorphism* among subclasses. With inheritance and polymorphism, the exact code executed when a method is called depends on the specific type of the object given as an argument. Finally, an *interface* is similar to a class but has no associated object; it only contains method signatures. However, if a class (which does have an associated object) implements an interface, then the methods identified by the interface's signatures operate on the class's object. A subclass can only extend one class, but it can implement multiple interfaces.

Invoking a method that operates on an object is often likened to "sending a message" to the object, where the message indicates what should be done and may return a response value. The actual details of the object's implementation, for example, the way it is laid out in memory, are hidden and can therefore be implementation dependent. What is important is that methods always produce the same results regardless of the implementation.

There are also ways of restricting the scope over which an object

There are also ways of restricting the scope over which an object can be accessed by other parts of a program. In Java, for example, a class may be declared to be *public, private,* or *protected.* These designations indicate the visibility (and ability to use) an object outside of a given library of objects. The public and private designations are more or less self-explanatory: If a class is public, it can be used by code outside the library; if it is private, it can only be used by other classes inside the same library. Protected classes are similar to private ones, except any inheriting class is given access.

Example

Figure 5.5 is a Java program that defines a Rectangle class via two methods, one that finds the area and one that finds the perimeter of the Rectangle. The program also extends the Rectangle class with a Square class. The Square class has a somewhat simplified perimeter method that will override the Rectangle's perimeter method. The main program reads pairs of integers from the command line. These integers represent a rectangle's length and width. If the sides are the same, a Square object is instantiated; otherwise, a Rectangle is instantiated. Then the area and perimeter methods are called, with a reference to the Square or Rectangle as the argument. The particular perimeter method used depends on the type of object to which the reference points. That is, the specific perimeter method is *dynamically dispatched* at run time. This program is admittedly contrived—the sides array is somewhat superfluous—but it allows us to illustrate a number of JVM features in a small example.

Garbage Collection

One of the nice features of the object-oriented model is that, from the programmer's perspective, memory is a large, virtually unbounded space populated with objects. These objects essentially "float" in the memory space, with references tethering them to executing programs. Objects can be created dynamically, used for a

while, and then abandoned by overwriting or deleting all references to the object. For example, in the code given in **Figure 5.5**, when the reference a is reassigned to a new Rectangle or Square, the previous Rectangle/Square object is no longer referenced and becomes *garbage*.

```
public class Shapes {
   public static void main (String args []) {
       Rectangle a;
       int length, width;

       for ( int i = 0; i < 4 ; i++) {
         length = Integer.parseInt (args [2*i]);
         width = Integer.parseInt (args [2*i+1]);
         if (length==width)
          a = new Square (length);
         else
          a = new Rectangle (length, width);
         System.out.println(a.area());
         System.out.println(a.perimeter());
       }
   }
}

class Rectangle {
   protected int sides [];
   public Rectangle (int length, int width) {
       sides = new int [2] ;
       sides [0]  = length;
       sides [1] = width;
   }

   public int perimeter ( ) {
       return 2*(sides[0] +sides [1]);
   }

   public int area ( ) {
       return (sides [0]*sides[1]);
   }
}

class Square extends Rectangle {
   public Square (int length) {
       super (length, length);
   }
   public int perimeter ( ) {
       return 4*sides [0];
   }
}
```

Figure 5.5 A Java Program That Computes the Area and Perimeter of a Sequence of Squares and Rectangles.

While this model of an unbounded memory space is nice from the programmer's perspective, in a real implementation the memory cannot be unbounded, of course. Consequently, there must be some

way of recycling the memory resources used by objects that are no longer needed. In some object-oriented languages, C++, for example, it is expected that the programmer will explicitly release an unneeded object so that memory management software can reuse its memory. This places a burden on the programmer, however, and if object memory is not properly released, a "memory leak" can develop where unused objects grow in number until the processor's memory resources are exhausted. On the other hand, if a program bug allows the same memory to be freed twice (a *double-free*), then a security hole may be created.

To avoid memory leaks and double-free bugs as well as to enhance program robustness, the modern HLL VMs relieve the programmer of the burden of keeping track of unneeded, garbage objects. In particular, the VMs automatically collect garbage in a manner that is hidden from the application program. The VM implementation is able to find objects for which references no longer exist and to collect these objects on behalf of the user. The programmer is not burdened with explicitly returning unused objects to memory management software, and the possibility of programmer memory management bugs is eliminated.

5.2.3 Networking

In the network computing environment, network bandwidth is sometimes a limited resource. In an HLL VM there are at least two ways that network bandwidth usage can be reduced. The first is by reducing the size of a program (or dynamically linked library routines) that must be moved over the network. This leads to program encodings (i.e., instruction sets) that can specify a program in a dense manner. A key point is that with a VM performing emulation, the information transported over the network is a *specification* of the program to be executed, not necessarily the actual instructions that will eventually be executed. Translation converts the specification into real native instructions—interpretation does the same, for

tion into real native instructions—interpretation does the same, for that matter, albeit much less efficiently.

Using a stack-oriented instruction set with variable-length instructions leads to a fairly compact program specification, especially when compared with the relatively sparse encoding found in a typical RISC ISA, which uses lots of registers and fixed-length instructions. With an operand stack, it is not uncommon for an instruction to require only one byte, i.e., to specify the opcode; the operands are implied by their positions on the stack.

Although the stack instruction sets used in most modern HLL VMs may save some network bandwidth, the addition of metadata adds to network bandwidth requirements. Overall, it is probably a wash, with modest savings of network bandwidth (at best). However, as described earlier, the presence of the metadata brings some additional advantages that a conventional program binary does not offer. The bigger win with respect to network bandwidth savings comes from dynamic loading of program (class) files. That is, a VM implementation generally loads only binary classes that it anticipates it will need, thereby not wasting network bandwidth on binary classes that are never used.

5.2.4 Performance

An object-oriented language provides a number of significant advantages when compared with a language such as C, which is really only a step above assembly language. However, the object-oriented model also makes it significantly more difficult to achieve high performance on conventional hardware. Hence, for many (but not all) applications there is heavy emphasis on improving HLL VM performance. Most of the emulation techniques described in **Chapters 2**–**4** can be used for enhancing HLL VM performance. The execution engine of an HLL VM implementation can use interpretation, binary translation to native code, or some combination.

The Java instruction set is designed for straightforward interpretation. On the other hand, the Microsoft intermediate language is not intended to be interpreted (however, it could be, albeit very slowly). Both the Java instruction set and MSIL use rather constrained, well-specified control flow instructions (branches and method calls), so the code-discovery problem is very simple. As soon as a method is entered, all the code belonging to the method can be immediately discovered. This leads to just-in-time (JIT) compilation, where a method, when first entered, is compiled[4] in its entirety. Often, JIT compilation generates nonoptimized or lightly optimized code. Beyond the two basic methods of interpretation and JIT compilation lies a range of more advanced staged emulation strategies. For example, an implementation can begin interpreting, with profiling added. Then hot methods can be compiled (or sections of hot methods). Finally, the compiled methods that are very frequently used can be much more highly optimized. **Section 6.6** contains further discussion of higher-performance HLL VM implementation techniques.

5.3 The Java Virtual Machine Architecture

To look at HLL VMs in more detail, we focus on the Java virtual machine (JVM) (**Lindholm and Yellin 1999**). The JVM came before the Microsoft CLI and is in some sense a purer form of HLL VM. The CLI, on the other hand, is part of the more general .NET framework, where both managed and unmanaged applications can easily interoperate, if the user wishes. Following the discussion of the Java VM in this section, we overview the CLI in the next section.

5.3.1 Data Types

Java consists of primitive data types and references, along with ob

Java consists of primitive data types and references, along with objects, which are composed of the primitive types and references.

Primitive Data Types

The Java HLL VM supports a number of primitive data types from which objects are composed. The types are int (integer), char, byte, short, float, double, and returnAddress. A boolean value is implemented in the JVM as a primitive type int or byte. It is important to note that the primitive types are defined according to the values they can take, not the number of bits of storage they will consume in the implementation; for example, an integer data type is defined to represent any integer in the range —2^{31} to +2^{31}—1. Nor are the actual bit patterns explicitly specified. This allows the primitive types to be held in an implementation-dependent fashion on any given host platform. Of course, many implementations will represent integers with 32-bit words and a two's complement representation, but other implementations could use more storage bits and/or other representations. The returnAddress type, by the way, is a rather obscure type not present in the Java HLL but present in the Java ISA. It is used by "miniature subroutines" implemented with jsr and ret instructions (these are typically used in implementing the Java HLL finally clauses, and we don't discuss them any further).

References

In addition to the foregoing primitive types, the JVM includes a reference type, which can hold reference values. A reference value points to an object stored in memory (discussed next). A reference can also have a null (undefined) value if it has not been assigned. As is the case with primitive types, a reference has an implementation-dependent internal representation; the number of bits necessary for holding a reference is not defined as part of the Java ISA.

Objects and Arrays

As described earlier, objects carry data held in logical structures declared by the programmer. As such, objects are composed of primitive data types and references that may point to other objects. An array object is a special, intrinsic type of object, with explicit instruction set support. Each array is defined, at the time it is declared, to have a fixed number of elements that do not change during the program's execution. The elements of an array must all be of the same primitive type or must all be references. If they are references, then they must all point to objects of the same type. In their basic form, arrays have a single dimension, but multidimensional arrays can be built using arrays of references.

5.3.2 Data Storage

In the JVM, there are three general types of data storage—global, local, and operand. *Global* storage is main memory, where globally declared variables reside. *Local* storage is temporary storage for variables that are local to a method. *Operand* storage holds variables while they are being operated on by the functional instructions (arithmetic, logical, shifts). All storage is divided into *cells* or *slots*, where a cell or slot can usually hold a single data item (the exceptions are double-precision floating-point numbers and long integers, which consume two slots). The actual amount of space required by a cell or slot is implementation dependent, but all addressing is in terms of the logical memory cells.

The Stack

Local and operand storage are allocated on the stack, and procedure arguments are passed on the stack as well. Instructions can never place arrays and objects on the stack; only references and individual array elements can be placed on the stack. As each method is called, a stack frame is allocated (**Figure 5.6**), with arguments, local storage, and operand storage being allocated in that order. Local

storage for a given method is of fixed size; the proper amount of stack space required for local storage can be determined at compile time. As noted earlier, the stack holds data in slots. In many conventional ISAs, operand storage is a register file; but just as in P-code, both Java bytecodes and MSIL use the stack for operands.
Depending on the actual instruction sequence, a stack often has advantages with the respect to instruction set encoding density (no instruction fields are needed for specifying registers), and a stack is amenable to platform independence (the host platform can have any number of registers in its ISA).

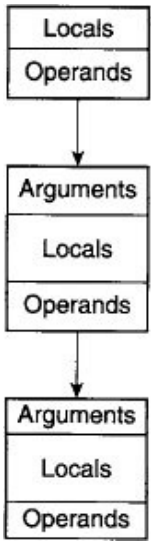


Figure 5.6 Java Stack Structure.

Global Memory

The logical main memory architecture in Java contains a method area for holding code and a global storage area for holding arrays and objects. The global memory area is managed as a heap of unspecified size with respect to the JVM architecture; i.e., its total size is an implementation-dependent feature. The heap can hold both static and dynamic objects, including arrays; these objects are created on the heap at run time. When an object is dynamically created on the heap, a reference is generated to point to it at that time. Objects in the heap can only be accessed via a reference having a

type that matches the type of the object. For example, in **Figure 5.5** the reference a is declared to be of type Rectangle, and it can only be used for pointing to objects of type Rectangle (or a subclass of Rectangle).

Constant Pool

Instructions often use constant values, as integer operands or as addresses in local memory, for example. The ISA allows some constant values to be placed in the instruction stream as immediate operands. But in general the constants have a range of lengths, and some of them are used by a number of different instructions. So to make the Java ISA a little more compact and uniform, constant data associated with a program is placed in a block known as the *constant pool.* Any instructions that need constant values can then index into the constant pool to retrieve them. Note that although the exact representations of integers and references when held in global memory or the stack are not defined part of the ISA, their representations when held in the constant pool are defined as part of the ISA. For example, an integer in the constant pool is represented as a 32-bit two's complement number. It is important to keep in mind that the constant pool is a part of a program's specification, just as much as the actual instructions are, and it does not change as a program executes.

Memory Hierarchy

Figure 5.7 illustrates the memory hierarchy in Java. In the figure, an array has been allocated on the heap, and a reference to the array is included as part of another object. Also note that in the lower left corner is an object with no references pointing to it. This is an object ready to be garbage collected. A specific field within a referenced object is accessed via an offset contained in the constant pool. That is, there is no indirection when identifying a field (other than the indirection provided by the reference itself).

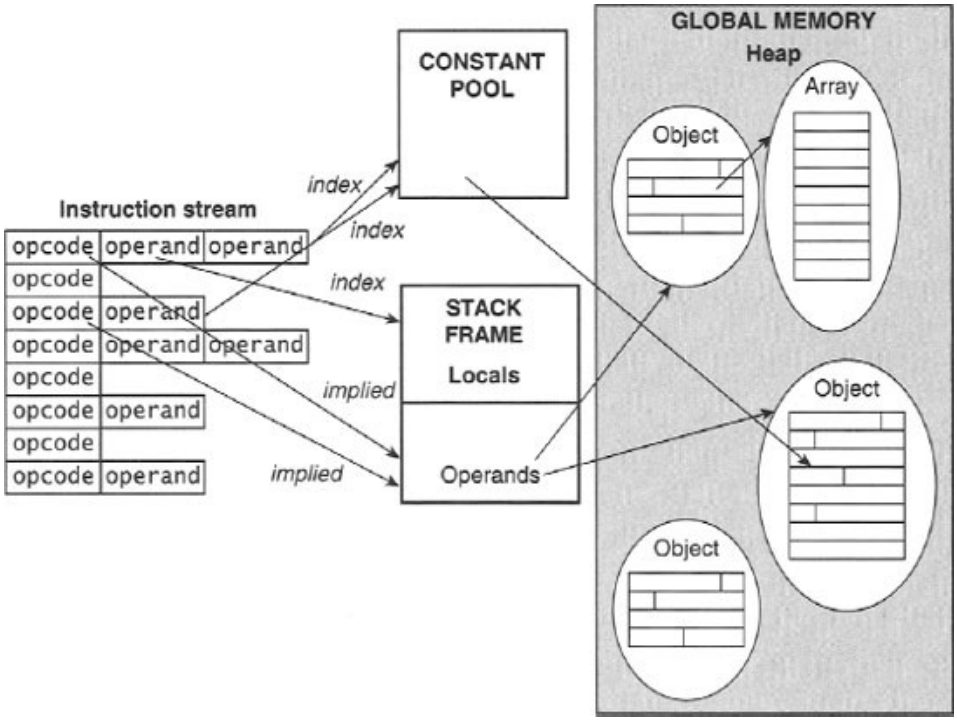


Figure 5.7 Memory Hierarchy Used by a Java Program.

5.3.3 Java Instruction Set

The Java ISA is stack based, and, at least superficially, it is similar to the one used in P-code.

Instruction Formats

All instructions contain an opcode byte and zero or more subsequent bytes, depending on the opcode. **Figure 5.8** shows the common instruction formats. Each instruction field consists of exactly one byte, although two or more such byte fields may be concatenated to form a single operand. Many instructions are a single byte, i.e., just the opcode, as shown in **Figure 5.8**a. Some instructions consist of an opcode and an index byte (or bytes). Two examples are in **Figure 5.8**b and c. The index bytes are used as indices into the constant pool or into local storage locations. Another class of instruction formats consists of an opcode and one or more data bytes (**Figure 5.8**d and e). The data bytes may be immediate data or off-

sets for PC-relative branch instructions. Because the instructions appear as a stream of bytes—either opcodes, indices, or data—an instruction set of this type is commonly referred to as a *bytecode* instruction set or simply as *bytecodes*. Using a single opcode byte implies there are 256 or fewer total opcodes; but a special wide opcode, when appended to a regular opcode, essentially creates a new instruction that performs the same operation as the regular opcode but with additional index bytes. And a bytecode instruction set can always be expanded by adding "escape code" bytes similar to the ones used for extending conventional instruction sets when all opcodes have been used up.

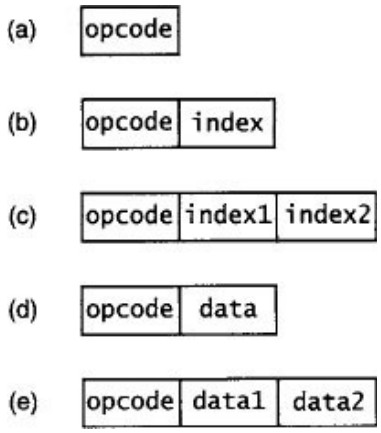


Figure 5.8 Typical Bytecode Instruction Formats.

A basic property of the Java instruction set is that each of the primitive types has specific bytecode instructions that can operate on them. For example, the iadd opcode (integer add) is defined to operate only on integer operands on the stack. In a legal Java bytecode program, the operand types must match those required by the opcode. In the following instruction descriptions, we will typically use the integer forms of instructions as examples; the ISA defines similar instructions for the other primitive types.

The following subsections will overview the various types of instructions. A complete list of instructions can be found in a book focused on the Java virtual machine (e.g., **Lindholm and Yellin 1999**;

Venners 1998). Instruction descriptions include the instruction format (opcode first), followed by operand byte(s), if any. All operand bytes are shown separately, even if they only have meaning when concatenated. For example, a 16-bit data item is represented as two bytes: data1 and data2.

Data-Movement Instructions

Figure 5.9 illustrates the basic flow of data movement. All loads and stores from global or local memory must be to the operand stack, and all functional instructions operate on operands held on the stack.

One set of data-movement instructions push constant values onto the stack. There is some redundancy in these stack instructions; i.e., some constants can be pushed onto the stack via more than one type of instruction. For example, any of five different instructions can push the constant 1 onto the stack. The advantage of having this redundancy is that it can improve code density and/or interpretation speed. The most common constants (e.g., small integers) are each given their own opcode, so a one-byte instruction is sufficient. For example, iconst1 is a single-byte instruction that pushes the integer constant 1 onto the stack. Other short constants can be pushed via the bipush data or sipush data1 data2 instructions—these require two or three instruction bytes, respectively. Any arbitrary constant can be pushed via the ldc index instruction (two bytes total), provided it is in one of the first 256 slots in the constant pool. (Of course, the constant pool entry itself contains additional data bytes.) Finally, the ldc_w index1 index2 instruction uses three bytes and can load a constant from any of 16K slots in the constant pool. The main difference between the ldc instructions and the push instructions is that the former requires a level of indirection when being interpreted. If translated (compiled) code is used, there is no practical difference between the two types of instructions.

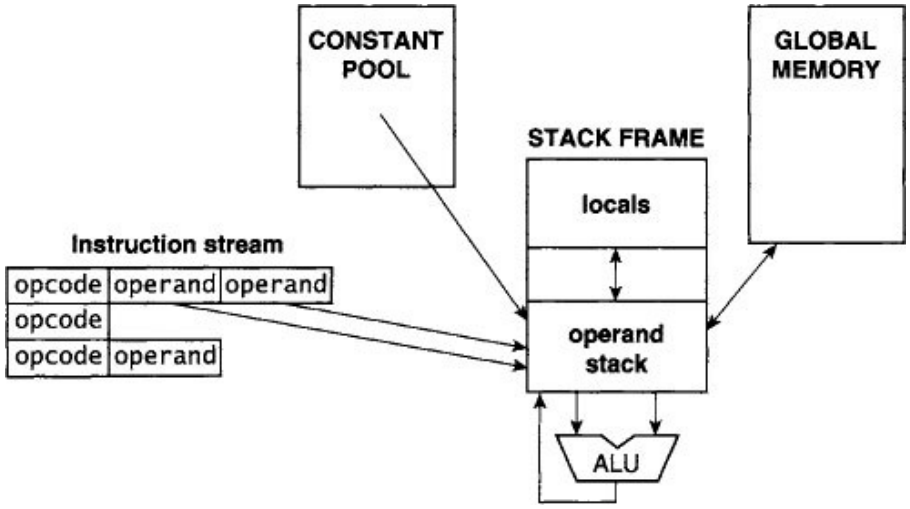


Figure 5.9 Data Movement Supported by the Java ISA.

A second set of data-movement instructions manipulate the stack entries. For example, the pop instruction pops the top element from the stack and discards it. The swap instruction swaps the positions of the top two stack elements, and dup duplicates the top stack element to form the top two stack elements.

The third set of data-movement instructions moves values between the local storage area of the current stack frame and the operand stack. These instructions specify the local storage address via a constant—either directly in the instruction or via an index to the constant pool. Also, the type of the data being moved is explicitly given in the opcode. This information is used for type checking when the program is initially loaded (to be discussed later). For example, the iload_1 instruction takes the integer from local storage slot 1 and pushes it onto the stack. The iload index instruction specifies the local storage slot number via a constant pool entry pointed to by the index. Similarly, istore_1 and istore index instructions move data from the stack to local storage in the current stack frame.

The final set of data-movement instructions deal with global memory data, either objects or arrays. An object is created via the new index1 index2 instruction that concatenates two bytes to form an in-

dex into the constant pool. The constant pool entry essentially specifies the object, and a new instance of the object is created on the heap and initialized. A reference to the object is pushed onto the stack. Similarly, the newarray type instruction creates an array containing elements of a specified primitive type.

For accessing data held in objects, the primary data-movement instructions are the getfield index1 index2 and putfield index1 index2 instructions, which point to a constant field entry containing information regarding the field, e.g., its type, size, and offset into the object that contains it. As with all data-movement instructions, these move data between the addressed data item and the operand stack. The getstatic and putstatic instructions are similar, except they deal with static rather than dynamic objects. There are similar instructions that move data to and from arrays.

In addition to instructions that create and access objects and arrays, there are instructions that can perform run-time checks to see what type of object is being pointed to by a reference. For example, the checkcast index1 index2 instruction indexes into the constant pool to find the specification for a specific class or interface. Then the object pointed to by a reference on the top of the stack is checked to see if it is an instance of the type specified by the constant pool entry. If not, a CheckCastException is thrown. This instruction is used for checking whether a run-time cast (conversion of an object reference) is safe (e.g., the reference being cast is to a member of the same class or a subclass of the object to which it currently points). This instruction allows a program to test for a potentially unsafe cast and to handle it in a graceful way via the exception mechanism (described later).

Type Conversion

Some instructions convert one type of data item on the stack to another. An example conversion instruction is 12f, which pops an inte-

ger from the stack, converts it to a float, and pushes the float back onto the stack. Not every possible pair of primitive types is supported directly by a conversion instruction, however. In some cases a combined pair of instructions performs a conversion.

Functional Instructions: Arithmetic, Logicals, Shifts

There are a number of instructions that take input operands, perform operations on them, and produce a result. For the most part, these instructions consist of a single byte. Operands are always taken from the stack and results are placed on the stack. Three examples follow. The iadd instruction pops two integers from the stack, adds them, and pushes the sum onto the stack. The iand instruction pops two integers from the stack, performs a logical AND on them, and pushes the result onto the stack. The ishfl instruction pops two integers from the stack, shifts the top element left by an amount specified by the second element, and pushes the result onto the stack.

Because the internal representations are implementation dependent, what the shift and logical instruction do, strictly speaking, is to convert the integer value to a standard two's complement form before performing the operation and then convert it back to the internal representation before pushing it on the stack; i.e., the logicals and shifts operate *as if* the values are stored as 32-bit two's complement numbers, regardless of whether they are actually stored that way or not.

Control Flow Instructions

The control flow instructions (e.g., branches and jumps) are designed to expose all control flow paths within a method. This feature enables both complete discovery of all code within a method at the time it is first entered and load-time tracking of all variable types held in local storage (as explained shortly).

One set of conditional branches compares the top stack element with zero. For example, the ifeq data1 data2 instruction pops an integer from the stack and compares it with zero. If it is equal to zero, then there is a PC relative branch to an offset found by concatenating the two data bytes. Other conditional branch instructions compare two data items with each other. For example, the if_icmpeq data1 data2 instruction pops two integer values from the stack and compares the first with the second. If they are equal, then there is a PC relative branch to an offset found by concatenating the two data bytes. Finally, there are conditional branches to check for null/non-null references. For example, ifnull data1 data2 pops a reference from the stack and checks it for null.

There are also more complex control flow instructions. The lookupswitch instruction, given here, is typically used to implement a switch statement:

```
lookupswitch default1 default2 default3 default4
npairs1 npairs2 npairs3 npairs4
match1_1 match1_2 match1_3 match1_4
     offset1_1 offset1_2 offset1_3 offset1_4
match2_1 match2_2 match2_3 match2_4
     offset2_1 offset2_2 offset2_3 offset2_4
additional n-2 match/offset pairs
```

This is all one instruction, laid out in a way to make it more readable. In this instruction, the four default bytes are concatenated to yield a PC relative branch offset if none of the cases match. The four npairs bytes indicate the number of match/offset pairs included in the switch statement. The integer at the top of the stack is popped and is compared with each of the match values (four bytes each). If a match is found, then there is a PC relative branch to the corresponding offset value (four bytes). If there is no match, the branch is to the default location.

Methods are called via one of the invoke instructions, which take a statically defined set of arguments. The return instructions are defined to return either no value or a single value. There are four types of invoke instructions. One of the more commonly used is invokevirtual index1 index2, which begins by indexing into a constant pool location that contains a description of the called method. This description includes the address of the method, the number and types of arguments it takes, the number of locals it uses, and its maximum operand stack depth. Arguments on the stack are checked to make sure they match the specified argument types. If they do, a stack frame of the appropriate size is allocated, and the arguments are pushed as locals onto the stack. Then there is a jump to the called method. The return PC is saved on a stack, but the return PC value is not accessible, other than indirectly through a return instruction.

The other types of invoke instructions are: invokeinterface, which is used for invoking an interface method; invokespecial, which provides special operations for certain types of methods, initialization methods, for example; and invokestatic, which is used for static methods.

A typical return instruction is ireturn, which begins by popping an integer from the current stack frame before removing the current stack frame. The integer is then pushed back onto the stack (for use by the calling method). Finally, there is a return jump to the calling method. The simple return instruction is used when there is a void return value.

As pointed out earlier, we see that an important property of control flow instructions is that all control paths can be easily tracked. All branches (including the switches) are to fixed, compile-time PC offsets, so they are known at load time. Furthermore, the method calls and returns can be tracked as well, because they jump directly to a method (via a fixed index into the constant pool), not indirectly

method (via a fixed index into the constant pool), not indirectly through a pointer.

Operand Stack Tracking

Besides the individual instruction specifications, the Java bytecode ISA also defines some overall properties that must hold in order for a sequence of instructions to be part of a valid program. These overall properties represent an important difference between HL V-ISAs and conventional ISAs. One such property is that at any given point in the program, the operand stack must have the same number and types of operands, and in the same order, regardless of the path that was followed to get to that point. This allows the loader to analyze the program prior to execution in order to check the types that are being moved to or from memory (either global or local). That is, the types of the values on the operand stack can be tracked through *static* program analysis; it is not necessary to look at the actual types dynamically as the program is executed in order to make sure the proper types are being used. This property also means that the maximum depth of the operand stack can be determined for each method at the time it is compiled.

Figure 5.10 presents three examples to illustrate the static stack-tracking property. All three examples have control flow paths that diverge (due to conditional branches) and then reconverge. Hence, there are multiple paths that will lead to the reconvergence point. **Figure 5.10**a is a legal code sequence. It first pushes an integer A onto the stack. Then it tests B; if B is equal to zero, it pushes a second integer, C, onto the stack. Otherwise, it pushes the integer F onto the stack. At that point the two control paths reconverge and the top two stack elements are added and then stored to D. The key point is that when the two control paths reconverge, the operand stack has two integers regardless of the path taken.

```
iload   A       //push int. A from local mem.
iload   B       //push int. B from local mem.
```

```
          If_cmpne 0 else1 //branch if B ne 0
          iload    C       //push int. C from local mem.
          goto     endelse1
else1     iload    F       //push F
endelse1  add              //add from stack; result to stack
          istore   D       //pop sum to D
```

(a)

```
          iload    B       //push int. B from local mem.
          If_cmpne 0 skip1 //branch if B ne 0
          iload    C       //push int. C from local mem.
skip1     iload    D       //push D
          iload    E       //push E
          if_cmpne 0 skip2 //branch if E ne 0
          add              //add stack; result to stack
skip2     istore   F       //pop to F
```

(b)

```
          iload    A       //push int. A from local mem.
          If_cmpne 0 else1 //branch if A ne 0
          aload    B       //push reference B from local mem.
          goto     endelse1
else1     iload    C       //push integer C from local mem.
endelse1  iload    D       //push int. D from local mem.
          If_cmpne 0 else2
          astore   E       //pop reference to local mem.
          goto     endelse2
else2     istore   F       //pop integer to local mem.
endelse2  …
```

(c)

Figure 5.10 Examples of Code Sequences with Reconverging Control Flow. *(a) A valid sequence; (b) an invalid sequence where converging paths have different numbers of operands depending on the path followed; (c) an invalid sequence where reconverging paths have different types of operands depending on the path followed.*

The second code sequence (**Figure 5.10**b) is not a legal one. If B is equal to zero, the integer C is pushed onto the operand stack; then integer D is pushed onto the stack regardless of the value of B. Next, the code sequence tests E. If E is zero, the top two stack elements are added; otherwise nothing is done. Finally, the top of the stack is stored to local memory location F. This code has the property that if B is equal to zero, then the stack has two elements when the branch on the value of E is performed; otherwise the stack has only one element at the same point in the code. Now, it may be the case that E is zero whenever B is zero, and vice versa. However, it may not be possible to determine this fact by static analysis of the code, so the exact stack contents at the end of this code sequence are unknown

exact stack contents at the end of this code sequence are unknown.

The third code example (**Figure 5.10**c) is also illegal. It pushes a reference onto the stack if A is zero; otherwise it pushes an integer. Then it pops a reference if D is zero; otherwise it pops an integer. Here, the stack contents are different, depending on the path taken to reach the compare with D. If it happens that D is zero whenever A is zero, and vice versa, then the data types will "work out" at run time. However, as in the earlier case, it may not be possible to determine this through static analysis.

Because of the way the V-ISA specifies control flow (branches and procedure call/returns), a relatively simple analysis can determine if the proper stack discipline is followed. Because all branches are to constant locations within the same method, all paths in the method can be determined statically. At a method invocation, it is known that anything on the operand stack will not be touched by the invoked method, so the operand stack will have the same values when the method returns (with the possible addition of a return value whose type is given by the invoke opcode). Hence, the analysis algorithm simply traces all the static control flow paths and simulates a stack symbolically. If the symbolic stack contents ever differ at a control flow convergence point, then the program is not valid.

Example Program

Figure 5.11 is a bytecode compilation of the Java Rectangle perimeter method given in **Figure 5.5**. The numbers in the leftmost column are the PC values (as byte addresses) corresponding to the given instructions. The first instruction pushes a constant 2 onto the operands stack, and the second instruction pushes local variable 0 onto the stack (by convention, local variable 0 is the argument being passed to perimeter(); in this case it is a reference to the rectangle object). The getfield instruction then gets the reference to the sides array from the Rectangle object. The object reference is on the stack,

and constant pool entry 2 contains a description of the field that is being accessed, i.e., that it is a reference. The getfield instruction pushes the reference to the sides array on the stack. Then the next two instructions (iconst_0 and iaload) push index number 0 onto the stack (on top of the reference to array sides) and load element 0 from the array sides (via the reference and index number taken from the stack). The next four instructions are similar to the preceding four, and they load the value of sides[1] onto the stack. At that point, the bottom of the stack is a constant 2, the next stack value is sides[0], and the top of the stack is sides[1]. Then the iadd, imul sequence forms 2*(sides[0] + sides[1]). Finally, the last instruction returns, with the integer result on top of the stack.

```
public int perimeter () ;

0:  iconst_2
1:  aload_0
2:  getfield#2; //Field:  sides reference
5:  iconst_0
6:  iaload
7:  aload_0
8:  getfield#2; //Field:  sides reference
11: iconst_1
12: iaload
13: iadd
14: imul
15: ireturn
```

Figure 5.11 Java Bytecodes for the perimeter Method Defined for the Rectangle Class (in Figure 5.5).

5.3.4 Exceptions and Errors

In the Java architecture, some exceptions (or errors) are defined as part of the ISA and may be thrown as a result of program execution. Other exceptions can be user defined and are explicitly thrown by executing a throw instruction (described later).

A key property of the JVM is that all exceptions must be handled *somewhere*. That is, there is no global way to turn them off, as is the case with most conventional ISAs, where there is usually a mask bit corresponding to each trap condition. The rationale for dealing with

exceptions this way is not really specific to high-level VMs; rather, it is an overall program robustness consideration. If an exception is not handled by the method that throws the exception, then the current stack frame is popped and the calling method takes over responsibility; if the calling method doesn't have a handler, then another stack frame is popped, etc., until eventually the main program is reached. At that point if there is no handler, a standard handler takes over (and likely terminates the program). In a way, this approach forces the programmer to think about all exceptions in order to deal with them actively rather than just turning them off and forgetting about them.

In Java there is a distinction made between errors and exceptions. *Errors* are not necessarily caused by behavior that is inherent to the application program; rather, they may be caused by limitations of the VM implementation or VM bugs. *Exceptions,* on the other hand, are caused by program behavior that occurs dynamically—as the program executes. Static checking catches many programming mistakes and oversights, but some types of behavior cannot be caught until run time. An example of an error is StackOverflowError, which indicates that the available stack space is exhausted. There is no architected stack size in the V-ISA, and some VM implementations may run out of stack space before others. Hence, when this error occurs, it is not necessarily due to a program bug; it may just be an indication that the host platform does not have enough memory for the application's needs. On the other hand, it could also result from a program bug, such as runaway recursion. As another example, the InternalError indicates that the VM has encountered some type of internal bug. Just as with any well-designed program, the VM implementation software itself should contain internal error checking. If the VM should catch such an internal bug, it throws this error.

The Java ISA defines a number of exceptions. These are the program exceptions that must be checked for dynamically. Two common exceptions are the NullPointerException and the

ArrayIndexOutOfBoundsException, which are clearly described by their names. All references ("pointers") must be checked to make sure they are not null at the time they are used (although some explicit checks may be removed by optimizations). Also, all array indices must be checked. Array indices are the only way of indirectly accessing a Java data structure. There are a number of exceptions related to type checking and object accesses. For example, an IncompatibleClassChangeError occurs when a getfield instruction is applied to a static field; even though the name indicates an "error," it is in fact an exception.

In the Java ISA, it is possible to specify an exception handler, depending on where an exception occurs. To accomplish this, the JVM associates a table with each method, the *exception table*. Here is an example entry in an exception table:

From	To	Target	Type
8	12	96	Arithmetic Exception

This table entry indicates that if an arithmetic exception occurs anywhere between instructions at locations 8 and 12, then there should be a jump to a handler at location 96. The table is built from information encoded in the high-level language program, for example, with Java try and catch regions. If a From/To region encloses a method call, then unless the method has a handler of its own, the handler in the calling method is used.

At the time an exception is thrown, the operand stack is immediately flushed. The JVM then looks up the exception type in the exception table. If the exception type and the PC at the time of an exception match a table entry From/To range, then there is a jump to the target PC specified in the table. Otherwise, the current stack frame is removed, the PC at the time of the call is reinstated, and the

frame is removed, the PC at the time of the call is reinstated, and the table is checked again. This continues until either there is a table match or the outermost procedure (main) is reached.

Besides the exception table, JVM support for exceptions includes the instruction athrow index1 index2. This instruction explicitly throws an exception. The values index1 and index2 form an index into the constant pool. The constant pool location contains a description of the exception being thrown.

5.3.5 Binary Classes

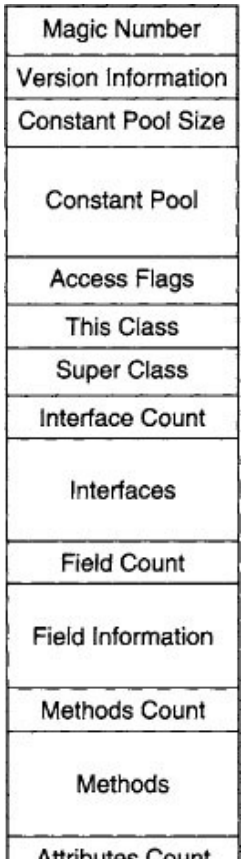
When an HLL VM program is distributed, not only is the code included, but so is a detailed specification of the data structures and their relationships, i.e., the *metadata.* In Java terminology, the combination of code plus metadata is a *binary class* that is typically included in a *class file* (in the Microsoft CLI it is called a *module*). The format of the binary class is, in effect, the "official" interface supported by the underlying virtual machine—the binary class format plays the same specification role as in a conventional ISA. Thus far, we have described the components of a binary class in a general way. In the remainder of this section we describe the overall structure of a Java binary class.

All the Java binary classes that form a complete program do not have to be loaded when a program is started (although they could be). Rather, the binary classes can be loaded on demand, at the time they are needed by the program. Among other things, this saves bandwidth for loading binary classes that are never used, and it allows a Java program to start up quickly, using only some initial binary classes. For efficiency, the first time a binary class is used, it is parsed and placed into *method memory,* a part of the VM implementation. Then on subsequent calls, it is much more efficient to consult the preprocessed method information in the method memory.

Either each component of a binary class is of a fixed size or the size

Either each component of a binary class is of a fixed size or the size is explicitly given immediately before the component contents. In this manner, the loader can parse the entire binary class from beginning to end, with each of the components being easily recognized and delineated. Within each component, the same principle is used.

Figure 5.12 gives the layout of a binary class. It consists first of some header information, beginning with a *magic number* that identifies this block of data as a binary class. The magic number is simply a character sequence that is the same for all Java binary classes. When presented with a file purported to contain a binary class the loader can use the magic number as a quick, initial check to gain some assurance that it does in fact contain a binary class. The header also contains version information for the binary class. Over time, it is expected that there will be extensions to the Java VM, and the version number can be used by the loader to restrict its checks and operations to the correct version.



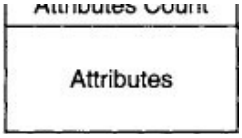


Figure 5.12 Format of a Java Binary Class.

Following the header information is a sequence of large structures, each preceded by a size indication or the number of contained elements. The major structures are the Constant Pool, a table describing method Interfaces, a table describing object Fields, a table containing the Methods, and a table of Attributes that contains much of the detailed information for the other tables. The following subsections describe each of these major structures as well as some of the minor ones.

Constant Pool

The constant pool essentially holds all the constant values and references used by the methods that are to follow. Each of the constants is appended with type information so that type checking can be performed when the constant is used. Many of the constants are in symbolic form, i.e., the names of classes, interfaces, methods, and fields.

Access Flags

As the name suggests, these flags provide access information, for example, whether this particular type (class) is public, whether it is an interface rather than a class, whether it is final (i.e., it can never be dynamically overridden).

This_Class and Super_Class

This region contains the names of this class and the superclass of this class. Both are given as indexes into the constant pool. The names themselves are in the constant pool. All classes except for the Object class (see **Section 5.4.2**) have a super_class. For the Object

class, the super_class is zero.

Interfaces

This region contains a number of references to the superinterfaces of this class, i.e., the interfaces through which this class can be directly accessed. These are given as indices into the constant pool. The constant pool entries are references to the interfaces.

Fields

This component contains the specification of the fields declared for this class. This information is included in a small table for each field; the table includes access information (public, private, protected), a name index (an offset into a constant pool entry that contains the name of this field), a descriptor index (which contains an index into the constant pool where the descriptor for this field can be found), and attributes information.

Methods

This component contains information regarding each method, e.g., the name and descriptor, as well as the method itself, encoded as a bytecode instruction stream. Each method can also have attribute tables, for example, giving the maximum operand stack depth for the method and the number of locals. The code itself appears as part of an attribute table. Also included is the exception table, described earlier, which give types of exceptions to be checked and the PC ranges over which they should be checked.

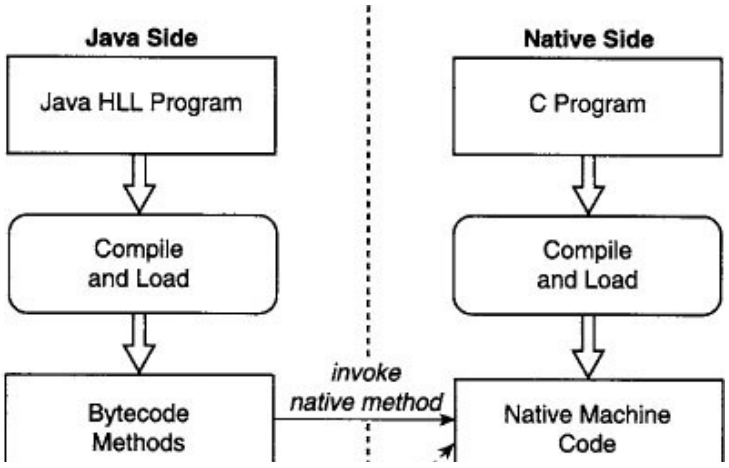
Attributes

The attributes region contains the detailed information regarding the other components listed earlier. In general, an HLL VM would have a fairly large number of attribute types.

5.3.6 The Java Native Interface

The Java native interface (JNI) allows java code and native compiled code to interoperate. For example, it allows Java code to call a routine compiled from C, and vice versa, although we will emphasize the former situation in this discussion. By using the JNI, a C program can even invoke a Java virtual machine.

An overview of the JNI and its operation is given in **Figure 5.13**. On the left-hand side of the figure, the "Java side," we see the Java architecture as described in the preceding sections. Code (and data) compiled to the native platform ISA resides on the right-hand side of the figure, i.e., the "native side." It is important to note, however, that the Java side can be compiled from any language where there is a compiler that can produce standard binary classes; for example, C# could be used. The native side can be compiled from C or from any other language for which JNI support exists, including assembly language. Each side of **Figure 5.13** compiles to its own binary format. On the Java side, these are standard binary classes; on the native side, they are program binaries in the native platform's machine code. Data on the Java side exists as objects and arrays on the heap and variables on the stack. On the native side, data is organized in whatever way the compiler happens to lay it out; i.e., it is compiler dependent.



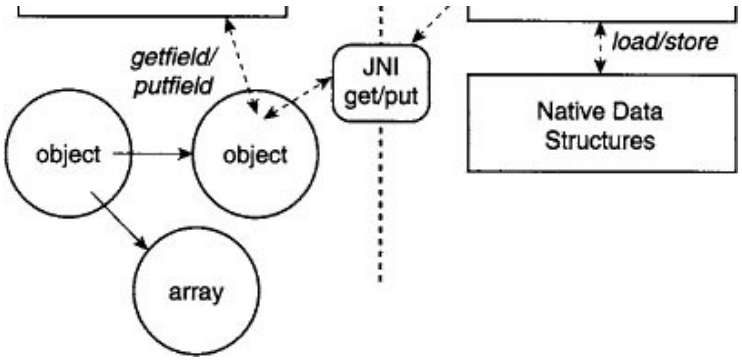


Figure 5.13 Java Native Interface. *The Java native interface allows Java software and native compiled software to interoperate.*

The JNI provides an interface for a Java method to call a native method (as shown in the figure). To do this, the native method must be declared as native by the calling Java class. After compiling the Java class that declares the native method call, it can be given to a program javah, which will produce a header file for the native method. Then the header and native method code can be compiled to form the callable native method.

The JNI specification allows control to be transferred back and forth between Java code and native methods; arguments can be passed and values can be returned. Furthermore, exceptions can be caught and thrown in the native code for handling in the Java application. In order for code on the native side to access data from the Java side (or for that matter to create objects on the Java side), the JNI provides a number of native methods—for example, GetArrayLength will obtain the length of a Java array, and GetIntArrayElements allows the native code to obtain a pointer to a Java array's elements. Similarly, code on the native side can get and put object field data through JNI methods.

5.4 Completing the Platform: APIs

In **Section 5.2** we stated that a Java platform is a combination of a JVM and a set of standard libraries, or APIs. The JVM is at the core of the platform, but it is the APIs that provide most of the features that

the platform, but it is the APIs that provide most of the features that are visible to users and software developers. These include support for secure network computing and component-based software, as well as many traditional functions, such as support for graphical user interfaces (GUIs). Similarly, the .NET framework is built around the CLI, but it is also includes a very large number of APIs that implement its rich set of features. Although the focus of this book is on the virtual machine, not the APIs, we would be remiss in not discussing some of the capabilities they provide. Furthermore, some of the APIs directly interact with the virtual machine and objects at a low level; these are of particular interest to us and will be discussed further.

5.4.1 Java Platforms

A number of computer companies provide Java platforms. To support interoperable application software, these platforms must conform to a standard specification and contain the same APIs. The current Java specification is for the Java 2 platform (**Shannon et al. 2000**). Also specified are *editions,* which are targeted more at general categories of applications and users. Each of the editions is distinguished from the others by the packages of APIs it incorporates.

J2SE—Standard Edition: As the name suggests, this edition is targeted at probably the largest number of Java users and developers. The standard edition platform supports typical PC users and client-side applications within an enterprise computing environment. It also incorporates APIs to support reusable component-based software development based on JavaBeans (discussed briefly later).

J2EE—Enterprise Edition: This edition is directed at development of large enterprise software infrastructure, including server-side applications. It includes the Enterprise JavaBeans API, which supports component-based development of distributed server applications.

J2ME—Micro Edition: This edition defines a lightweight platform

J2ME—Micro Edition: This edition defines a lighter-weight platform for consumer-based embedded systems, which are often resource limited. Consequently, it contains a smaller set of targeted APIs. Examples of embedded systems that may be based on the Micro Edition range from set-top boxes to pagers and smart cards.

5.4.2 Java APIs

There is a long list of Java APIs, too many to present and discuss here. Rather, we give some of the more important "core" API packages that exemplify the variety available, and then we single out some specific, particularly important low-level functions for additional discussion in subsequent subsections.

Java.lang

This is the core Java programming language API package; it contains classes that are at the heart of the Java programming language. These begin with the Object class—the superclass of all the Java classes. It also contains "wrapper" classes that contain primitive data types, such as Character and Integer when stored in the heap as individual items. In addition, java.lang contains low-level runtime and system static methods as well as floating-point math support. Other selected members of java.lang that are of special interest to us follow.

The Class class maintains descriptions of Java classes. Each class that is loaded has a Class object associated with it. Among other things, a class's Class object allows a program to extract information regarding the classes it is using, via a process known as *reflection,* described in the next subsection.

The Process object serves as a platform-independent interface to native processes external to the JVM, and the Thread class supports multithreading within the JVM. Use of the Thread class is discussed

further in **Section 5.4.4**.

The SecurityManager is the class that defines the methods that implement a security policy. These methods are called in order to determine if a requested operation (e.g., to read or write a file) should be permitted. The security manager's overall role is described briefly in **Section 5.2.1**, and it is discussed further in **Section 6.2.2**.

java.util

This package contains a number of classes that perform fundamental data structure operations as well as maintaining date and time facilities. For example, the Vector class supports arrays of objects that may grow as objects are added; the Enumeration interface is very commonly used for looping through the elements of a data structure. The Hashtable class supports associative arrays (typically implemented as hash tables). java.util.zip contains classes with methods for compressing and decompressing zip files, and java.util.jar contains classes for file archiving.

java.awt

The java.awt (abstract windowing toolkit) contains classes for constructing GUIs from components and graphics shapes. Layout manager classes control graphics components held in container objects.

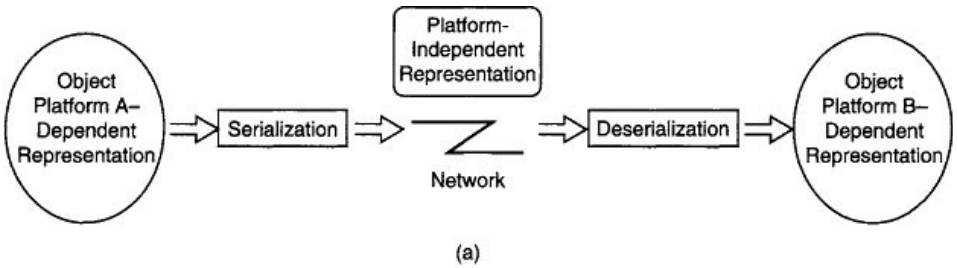
java.io and java.net

The java.io API package contains classes for performing input/output operations. These classes manage data streams and file system I/O, and they are the counterparts of the I/O libraries supported by conventional high-level languages and operating systems. The java.net API package contains classes for networking support. For example, the socket class allows a program to connect to a port and read and write data through the port.

We now consider two general functions where the core API part of the Java platform interacts very closely with the lower-level operation of the JVM. These are by no means the only situations where such close interaction is required, however; they are singled out because they are fundamental to many applications that run on Java platforms.

5.4.3 Serializability and Reflection

One of the basic concepts of object-oriented programming is concealing internal features of an object. However, there are situations where it is important to discover features of objects. Serializability and reflection involve exposing the features of an object to programs operating on the object and to the world outside a given VM. For example, one may want to communicate an object from one Java program to another; perhaps the two programs are running on different platforms connected by a network. This is commonly done as part of remote method invocation (RMI), where, for example, a client may want to invoke a method on a remote server and an object may be passed as an argument or return value. Each of the platforms may have a different way of storing objects in memory, so simply copying the internal data structures that implement an object will not work. Instead, an object to be communicated must be converted into an implementation-independent form. Then the receiving program can convert the implementation-independent form into its own internal implementation-dependent form (see **Figure 5.14**a).



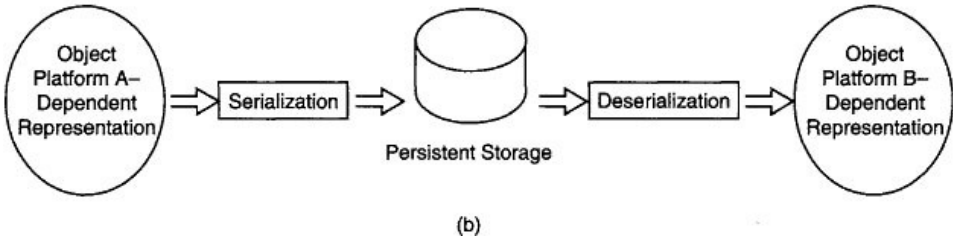


Figure 5.14 Serialization. *Serialization converts an object into an implementation-independent form that can be (a) passed over to a network to a different platform or (b) stored persistently for later use.*

As another example, objects created by a program only live as long as the program exists; when the program terminates, the objects go away. However, there are many situations where it is desirable for objects to *persist* between program executions, stored on a disk, for example. Here, again, the solution is to convert objects into an implementation-independent form for storage and later retrieval (see **Figure 5.14**b). These two situations are similar: In the network example an object is passed from one program to another through space, and in the other it is passed through time.

The process of converting an object into an implementation-independent form is called *serialization.* In order to serialize an object it must be declared to be serializable, which means that it implements the Serializable interface. This interface then allows an object to be written as a byte stream in a canonical form. Serialization serializes not only a given object but all the objects for which the original object contains references, all the objects for which those objects contain references, etc. The serialization process involves *reflection,* that is, the ability to look inside an object to find all of its members. Then once found, they can be organized as a standardized byte stream.

Besides serialization, a number of other interesting applications require reflection in order to determine class information at run time. In a typical program development environment, the compiler can access information regarding classes used by a program at compile

access information regarding classes directly at compile time, so the compiler can incorporate this class information as it generates new binary classes. However, there are situations where class information is not known at compile time, that is, where a running program may be given a reference to an object whose type is unknown. In these situations, information concerning the object must be retrieved at run time. This is done through reflection. In Java, the java.lang.reflect API is an interface to a class's Class object, which exists for every class used by a program and contains a description of the class. The Class objects are constructed by the JVM when an object type is loaded.

Reflection is a key enabler for component-based programming. In Java, basic program components are called *JavaBeans*. To facilitate component-based program design, JavaBeans must conform to a set of specified *design patterns* or coding conventions. For example, all readable properties of a JavaBean must have an accessory method of the form getproperty_name. Then if a program invokes an accessory method on an object, with, for example, the getFields() method, it will be given an array containing Field objects reflecting all the accessible public fields of the class or interface represented by this Class object.

A visual component-based program design tool allows a programmer to work at a high level by manipulating JavaBeans. Under programmer direction, the tool can load a JavaBean and analyze the bean's interface using the Reflection API and the JavaBean design patterns. With the information collected via the Reflection API, the developer's tool presents a graphical representation of the JavaBean that the developer can then use to modify the properties of the bean and enable communication between components by connecting an event source in one bean to an event listener in another.

5.4.4 Java Threads

A typical JVM implementation supports the multithreading

In a typical JVM implementation, much of the multithreading support is provided by Java libraries that are part of the java.lang. This makes sense, because support for multiple threads must ultimately be provided through the OS on the underlying platform, and the way to communicate with the underlying OS is through libraries.

As noted earlier, the java.lang API includes a Thread class. Threads can be defined by extending this class (and inheriting its methods). Methods defined for the Threads class, include: run(), which starts a thread, stop(), which stops a thread, and suspend() and resume(), which, as their names imply, allow the suspension and resumption of a thread. Calling run() for an object of type Class will initiate a new thread. The JVM gives each Java thread its own stack, and the threads synchronize through mechanisms called *monitors*.

Monitors are supported at the lowest level by architected locks and two Java bytecode instructions. There is a lock associated with each object and each class (the Class object for a given class contains its lock). Only one thread at a time can own a particular lock. A lock actually operates as a counter rather than as a single bit flag (as one might expect). The bytecode instruction monitorenter acquires the lock for the object pointed to by the reference on the top of the operand stack. If the lock is already held by another thread, then the requesting thread simply blocks and waits. If the lock is not held by another thread, then the lock is incremented and the requesting thread continues execution. Note that the acquiring thread may already hold the lock, in which case it does the increment and goes on. The monitorexit instruction decrements the lock for the object pointed to by the reference on top of the operand stack. If the lock becomes zero as a result of the decrement, then it is released and can be acquired by a waiting thread (if there is one).

The basic lock mechanism can be used for implementing monitors that guard critical sections or for locking individual objects. A *critical section* is a code region that only one thread can execute at a

given time. Critical sections typically manipulate shared data structures that require exclusive access, usually when they are updated. The synchronize keyword in the high-level Java language compiles down to monitorenter and monitorexit instructions.

Besides the bytecode instructions, the class Object declares a number of methods that support additional synchronization.

wait()—releases the lock and enters the *wait set* for the given lock. It remains there until it is *notified* (see later). The wait set is essentially a VM-supported pool of threads waiting for a notification regarding the lock.

wait(long timeout, int nanos)—releases the lock and enters the "wait set" for the given lock. It remains there either until it is notified or until timeout milliseconds plus nanos nanoseconds have elapsed.

notify()—wakes up one thread in the lock's wait set.

notifyAll()—wakes up all threads in the lock's wait set.

The wait and notify mechanisms are typically used for supporting monitors that manage cooperation among threads typified by producer–consumer interaction, where, for example, one thread might produce data items and place them in a shared buffer to be consumed by a different thread. The producer thread notifies the consuming thread when data is in the buffer. The consuming thread, after emptying the buffer, waits until it is again notified.

5.5 The Microsoft Common Language Infrastructure: A Flexible High-Level Language Virtual Machine

The common language infrastructure (CLI) is a virtual machine architecture that is part of Microsoft's .NET framework (**Box 2002**). The common language runtime (CLR) is Microsoft's implementation of the CLI. Although it supports object-oriented network-based computation, just as Java does, the CLI objectives are ostensibly broader than those for Java.

5.5.1 The Common Language Interface

The aspect of the CLI that is most similar to the Java environment is that it can support object-oriented programs within a managed runtime environment that includes features such as built-in protection checking and garbage collection. Although a number of high-level languages (some with extensions) can be compiled to Java binary classes, the JVM was designed specifically to support the Java HLL. In contrast, the CLI is designed to support multiple, interoperating high-level languages, and it also can support programs that do not have to be (or cannot be) verified by a loader. Hence, an essential feature that separates the CLI from the Java virtual machine is that the CLI strives for HLL independence as well as platform independence. This is illustrated in **Figure 5.15**.

The common language runtime (CLR) is an implementation of the CLI, and is part of the overall .NET framework. The modules are of a standard format and contain both metadata and code in the Microsoft intermediate language (MSIL). In the figure, modules (the analog of Java binary classes) can be generated from a number of languages, including, but not restricted to, C# (an object-oriented version of C that provides a garbage-collected heap), Java, Visual Basic .NET, and Managed C++. Managed C++ programs can run in the managed runtime environment provided by the CLR, but not all Managed C++ modules must be verifiable.

In the CLI, a code module can be verified for type safety in a man-