

Chapter 5 of Inside the Java Virtual Machine

The Java Virtual Machine

by Bill Venners

The previous four chapters of this book gave a broad overview of Java's architecture. They showed how the Java virtual machine fits into the overall architecture relative to other components such as the language and API. The remainder of this book will focus more narrowly on the Java virtual machine. This chapter gives an overview of the Java virtual machine's internal architecture.

The Java virtual machine is called "virtual" because it is an abstract computer defined by a specification. To run a Java program, you need a concrete implementation of the abstract specification. This chapter describes primarily the abstract specification of the Java virtual machine. To illustrate the abstract definition of certain features, however, this chapter also discusses various ways in which those features could be implemented.

What is a Java Virtual Machine?

To understand the Java virtual machine you must first be aware that you may be talking about any of three different things when you say "Java virtual machine." You may be speaking of:

- the abstract specification,
- a concrete implementation, or
- a runtime instance.

The abstract specification is a concept, described in detail in the book: *The Java Virtual Machine Specification*, by Tim Lindholm and Frank Yellin. Concrete implementations, which exist on many platforms and come from many vendors, are either all software or a combination of hardware and software. A runtime instance hosts a single running Java application.

Each Java application runs inside a runtime instance of some concrete implementation of the abstract specification of the Java virtual machine. In this book, the term "Java virtual machine" is used in all three of these senses. Where the intended sense is not clear from the context, one of the terms "specification," "implementation," or "instance" is added to the term "Java virtual machine".

The Lifetime of a Java Virtual Machine

A runtime instance of the Java virtual machine has a clear mission in life: to run one Java application. When a Java application starts, a runtime instance is born. When the application completes, the instance dies. If you start three Java applications at the same time, on the same computer, using the same concrete implementation, you'll get three Java virtual machine instances. Each Java application runs inside its own Java virtual machine.

A Java virtual machine instance starts running its solitary application by invoking the `main()` method of some initial class. The `main()` method must be public, static, return `void`, and accept one parameter: a `String` array. Any class with such a `main()` method can be used as the starting point for a Java application.

For example, consider an application that prints out its command line arguments:

```
// On CD-ROM in file jvm/ex1/Echo.java
class Echo {

    public static void main(String[] args) {
        int len = args.length;
        for (int i = 0; i < len; ++i) {
            System.out.print(args[i] + " ");
        }
        System.out.println();
    }
}
```

You must in some implementation-dependent way give a Java virtual machine the name of the initial class that has the `main()` method that will start the entire application. One real world example of a Java virtual machine implementation is the `java` program from Sun's Java 2 SDK. If you wanted to run the `Echo` application using Sun's `java` on Window98, for example, you would type in a command such as:

```
java Echo Greetings, Planet.
```

The first word in the command, "`java`," indicates that the Java virtual machine from Sun's Java 2 SDK should be run by the operating system. The second word, "`Echo`," is the name of the initial class. `Echo` must have a public static method named `main()` that returns

`void` and takes a `String` array as its only parameter. The subsequent words, "Greetings, Planet.," are the command line arguments for the application. These are passed to the `main()` method in the `String` array in the order in which they appear on the command line. So, for the previous example, the contents of the `String` array passed to `main` in `Echo` are: `arg[0]` is "Greetings," `arg[1]` is "Planet."

The `main()` method of an application's initial class serves as the starting point for that application's initial thread. The initial thread can in turn fire off other threads.

Inside the Java virtual machine, threads come in two flavors: *daemon* and *non-daemon*. A daemon thread is ordinarily a thread used by the virtual machine itself, such as a thread that performs garbage collection. The application, however, can mark any threads it creates as daemon threads. The initial thread of an application--the one that begins at `main()`--is a non-daemon thread.

A Java application continues to execute (the virtual machine instance continues to live) as long as any non-daemon threads are still running. When all non-daemon threads of a Java application terminate, the virtual machine instance will exit. If permitted by the security manager, the application can also cause its own demise by invoking the `exit()` method of class `Runtime` or `System`.

In the `Echo` application previous, the `main()` method doesn't invoke any other threads. After it prints out the command line arguments, `main()` returns. This terminates the application's only non-daemon thread, which causes the virtual machine instance to exit.

The Architecture of the Java Virtual Machine

In the Java virtual machine specification, the behavior of a virtual machine instance is described in terms of subsystems, memory areas, data types, and instructions. These components describe an abstract inner architecture for the abstract Java virtual machine. The purpose of these components is not so much to dictate an inner architecture for implementations. It is more to provide a way to strictly define the external behavior of implementations. The specification defines the required behavior of any Java virtual machine implementation in terms of these abstract components and their interactions.

Figure 5-1 shows a block diagram of the Java virtual machine that includes the major subsystems and memory areas described in the specification. As mentioned in previous chapters, each Java virtual machine has a *class loader subsystem*: a mechanism for loading types (classes and interfaces) given fully qualified names. Each Java virtual machine also has an *execution engine*: a mechanism responsible for executing the instructions contained in the methods of loaded classes.

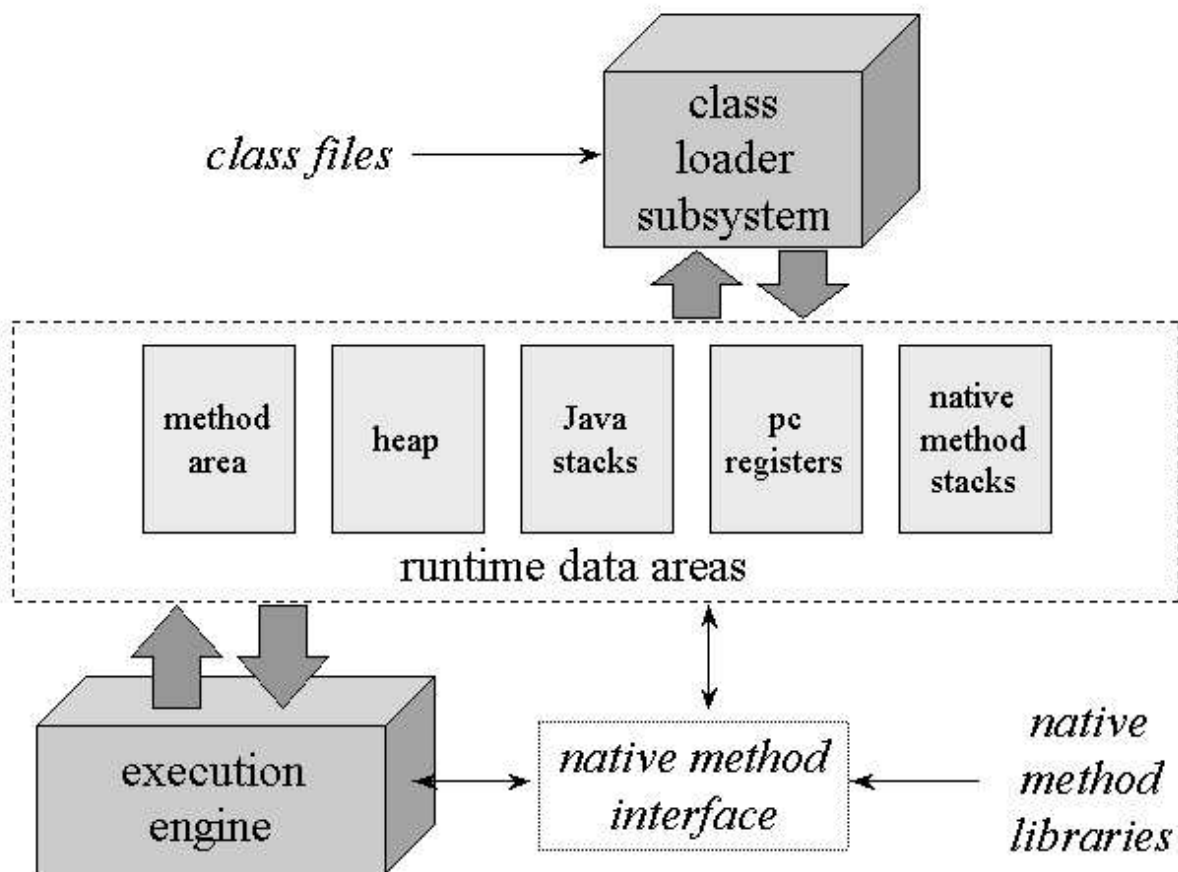


Figure 5-1. The internal architecture of the Java virtual machine.

When a Java virtual machine runs a program, it needs memory to store many things, including bytecodes and other information it extracts from loaded class files, objects the program instantiates, parameters to methods, return values, local variables, and intermediate results of computations. The Java virtual machine organizes the memory it needs to execute a program into several *runtime data areas*.

Although the same runtime data areas exist in some form in every Java virtual machine implementation, their specification is quite abstract. Many decisions about the structural details of the runtime data areas are left to the designers of individual implementations.

Different implementations of the virtual machine can have very different memory constraints. Some implementations may have a lot of memory in which to work, others may have very little. Some implementations may be able to take advantage of virtual memory, others may not. The abstract nature of the specification of the runtime data areas helps make it easier to implement the Java virtual machine on a wide variety of computers and devices.

Some runtime data areas are shared among all of an application's threads and others are unique to individual threads. Each instance of the Java virtual machine has one *method area* and one *heap*. These areas are shared by all threads running inside the virtual machine. When the virtual machine loads a class file, it parses information about a type from the binary data contained in the class file. It places this type information into the method area. As the program runs, the virtual machine places all objects the program instantiates onto the heap. See Figure 5-2 for a graphical depiction of these memory areas.

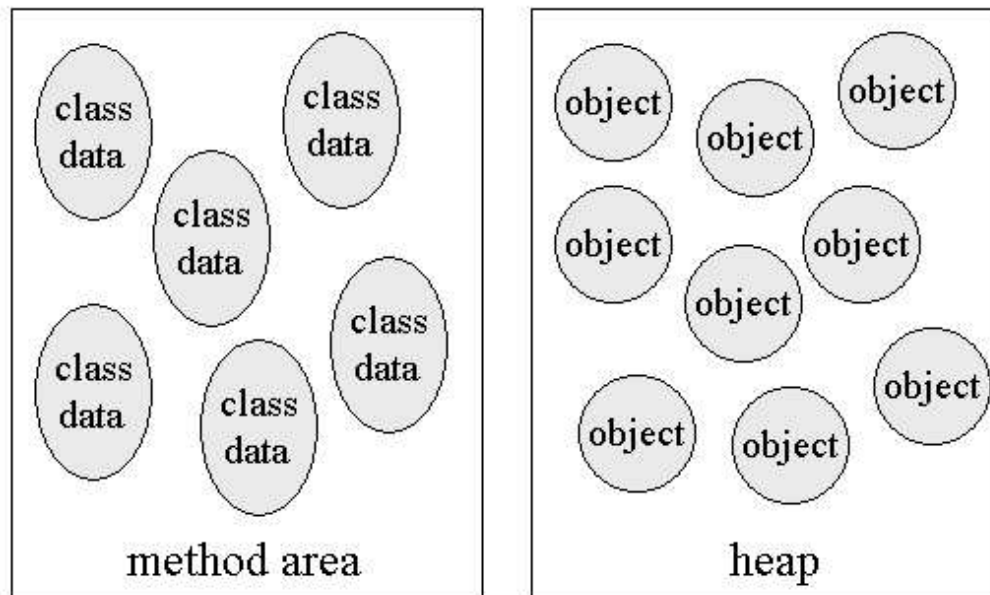


Figure 5-2. Runtime data areas shared among all threads.

As each new thread comes into existence, it gets its own *pc register* (program counter) and *Java stack*. If the thread is executing a Java method (not a native method), the value of the pc register indicates the next instruction to execute. A thread's Java stack stores the state of Java (not native) method invocations for the thread. The state of a Java method invocation includes its local variables, the parameters with which it was invoked, its return value (if any), and intermediate calculations. The state of native method invocations is stored in an implementation-dependent way in *native method stacks*, as well as possibly in registers or other implementation-dependent memory areas.

The Java stack is composed of *stack frames* (or *frames*). A stack frame contains the state of one Java method invocation. When a thread invokes a method, the Java virtual machine pushes a new frame onto that thread's Java stack. When the method completes, the virtual

machine pops and discards the frame for that method.

The Java virtual machine has no registers to hold intermediate data values. The instruction set uses the Java stack for storage of intermediate data values. This approach was taken by Java’s designers to keep the Java virtual machine’s instruction set compact and to facilitate implementation on architectures with few or irregular general purpose registers. In addition, the stack-based architecture of the Java virtual machine’s instruction set facilitates the code optimization work done by just-in-time and dynamic compilers that operate at run-time in some virtual machine implementations.

See Figure 5-3 for a graphical depiction of the memory areas the Java virtual machine creates for each thread. These areas are private to the owning thread. No thread can access the pc register or Java stack of another thread.

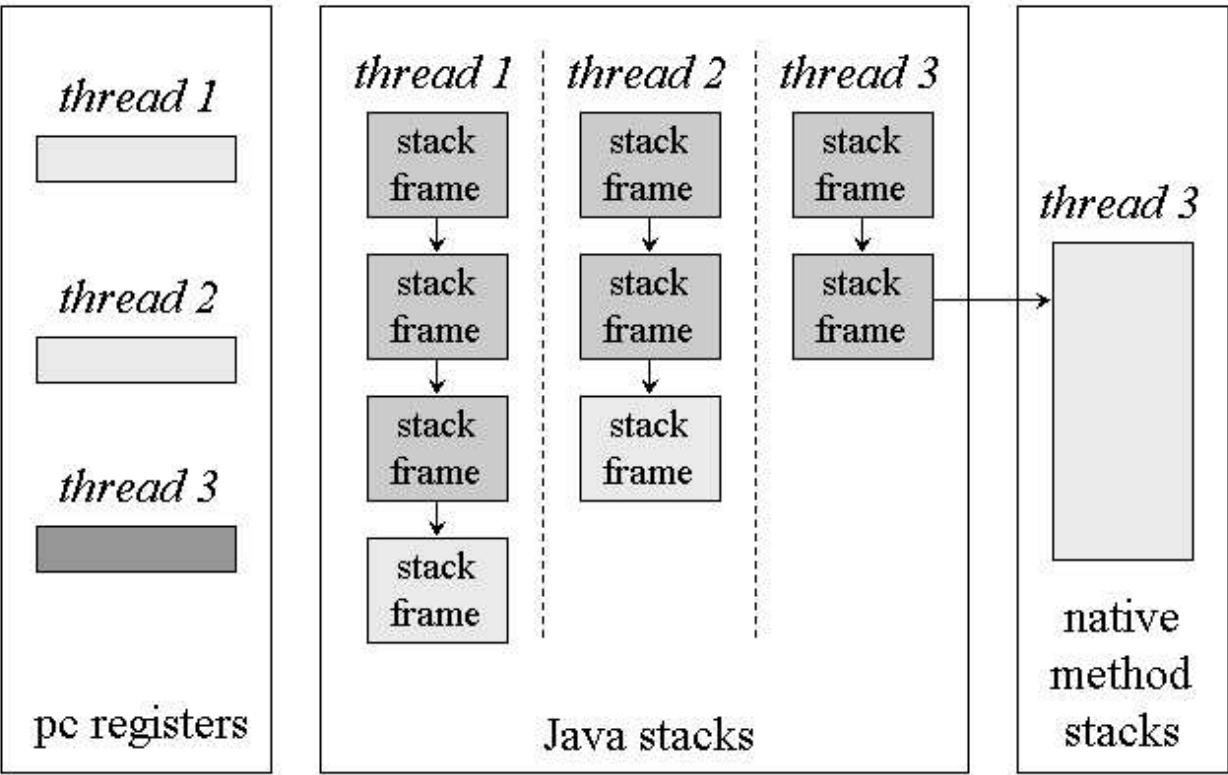


Figure 5-3. Runtime data areas exclusive to each thread.

Figure 5-3 shows a snapshot of a virtual machine instance in which three threads are executing. At the instant of the snapshot, threads one and two are executing Java methods. Thread three is executing a native method.

In Figure 5-3, as in all graphical depictions of the Java stack in this book, the stacks are shown growing downwards. The "top" of each stack is shown at the bottom of the figure. Stack frames for currently executing methods are shown in a lighter shade. For threads that are currently executing a Java method, the pc register indicates the next instruction to execute. In Figure 5-3, such pc registers (the ones for threads one and two) are shown in a lighter shade. Because thread three is currently executing a native method, the contents of its pc register--the one shown in dark gray--is undefined.

Data Types

The Java virtual machine computes by performing operations on certain types of data. Both the data types and operations are strictly defined by the Java virtual machine specification. The data types can be divided into a set of *primitive types* and a *reference type*. Variables of the primitive types hold *primitive values*, and variables of the reference type hold *reference values*. Reference values refer to objects, but are not objects themselves. Primitive values, by contrast, do not refer to anything. They are the actual data themselves. You can see a graphical depiction of the Java virtual machine’s families of data types in Figure 5-4.

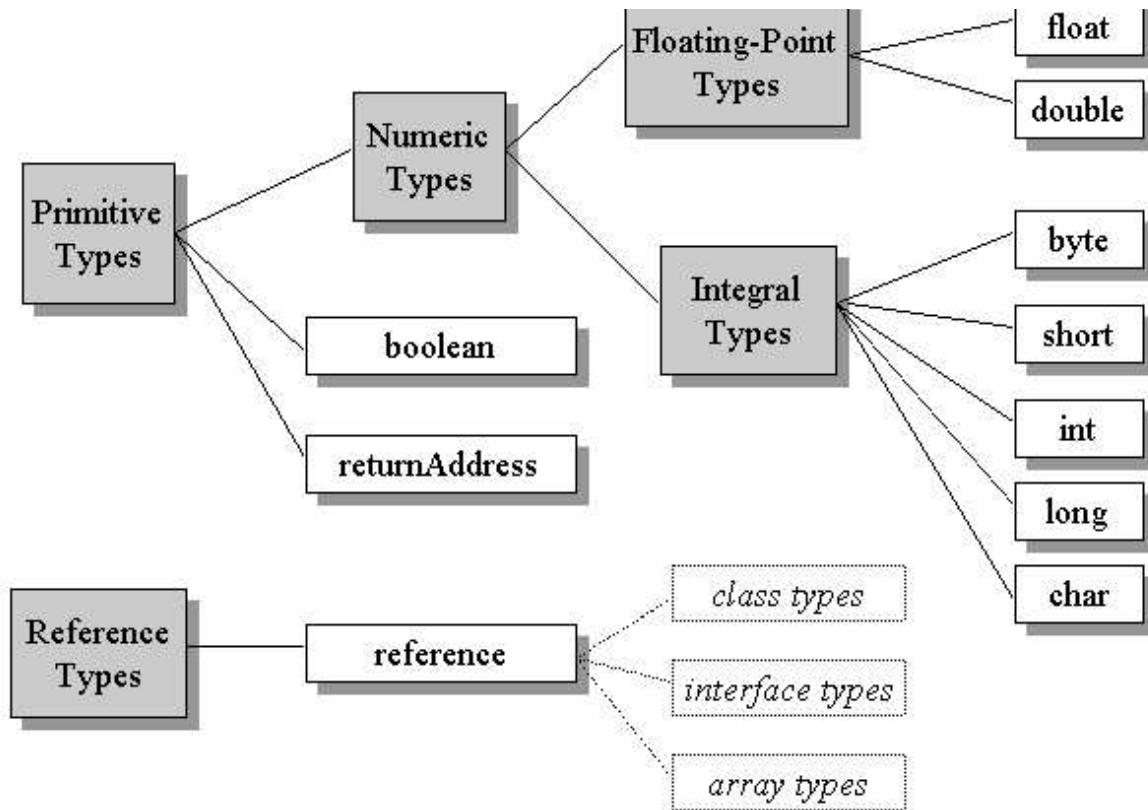


Figure 5-4. Data types of the Java virtual machine.

All the primitive types of the Java programming language are primitive types of the Java virtual machine. Although `boolean` qualifies as a primitive type of the Java virtual machine, the instruction set has very limited support for it. When a compiler translates Java source code into bytecodes, it uses `ints` or `bytes` to represent `booleans`. In the Java virtual machine, `false` is represented by integer zero and `true` by any non-zero integer. Operations involving `boolean` values use `ints`. Arrays of `boolean` are accessed as arrays of `byte`, though they may be represented on the heap as arrays of `byte` or as bit fields.

The primitive types of the Java programming language other than `boolean` form the *numeric types* of the Java virtual machine. The numeric types are divided between the *integral types*: `byte`, `short`, `int`, `long`, and `char`, and the *floating-point types*: `float` and `double`. As with the Java programming language, the primitive types of the Java virtual machine have the same range everywhere. A `long` in the Java virtual machine always acts like a 64-bit signed two's complement number, independent of the underlying host platform.

The Java virtual machine works with one other primitive type that is unavailable to the Java programmer: the `returnAddress` type. This primitive type is used to implement `finally` clauses of Java programs. The use of the `returnAddress` type is described in detail in Chapter 18, "Finally Clauses."

The reference type of the Java virtual machine is cleverly named `reference`. Values of type `reference` come in three flavors: the *class type*, the *interface type*, and the *array type*. All three types have values that are references to dynamically created objects. The class type's values are references to class instances. The array type's values are references to arrays, which are full-fledged objects in the Java virtual machine. The interface type's values are references to class instances that implement an interface. One other reference value is the `null` value, which indicates the `reference` variable doesn't refer to any object.

The Java virtual machine specification defines the range of values for each of the data types, but does not define their sizes. The number of bits used to store each data type value is a decision of the designers of individual implementations. The ranges of the Java virtual machines data type's are shown in Table 5-1. More information on the floating point ranges is given in Chapter 14, "Floating Point Arithmetic."

Type	Range
<code>byte</code>	8-bit signed two's complement integer (-2^7 to $2^7 - 1$, inclusive)

short	16-bit signed two's complement integer (-2^{15} to $2^{15} - 1$, inclusive)
int	32-bit signed two's complement integer (-2^{31} to $2^{31} - 1$, inclusive)
long	64-bit signed two's complement integer (-2^{63} to $2^{63} - 1$, inclusive)
char	16-bit unsigned Unicode character (0 to $2^{16} - 1$, inclusive)
float	32-bit IEEE 754 single-precision float
double	64-bit IEEE 754 double-precision float
returnAddress	address of an opcode within the same method
reference	reference to an object on the heap, or null

Table 5-1. Ranges of the Java virtual machine's data types

Word Size

The basic unit of size for data values in the Java virtual machine is the *word*--a fixed size chosen by the designer of each Java virtual machine implementation. The word size must be large enough to hold a value of type `byte`, `short`, `int`, `char`, `float`, `returnAddress`, or `reference`. Two words must be large enough to hold a value of type `long` or `double`. An implementation designer must therefore choose a word size that is at least 32 bits, but otherwise can pick whatever word size will yield the most efficient implementation. The word size is often chosen to be the size of a native pointer on the host platform.

The specification of many of the Java virtual machine's runtime data areas are based upon this abstract concept of a word. For example, two sections of a Java stack frame--the local variables and operand stack-- are defined in terms of words. These areas can contain values of any of the virtual machine's data types. When placed into the local variables or operand stack, a value occupies either one or two words.

As they run, Java programs cannot determine the word size of their host virtual machine implementation. The word size does not affect the behavior of a program. It is only an internal attribute of a virtual machine implementation.

The Class Loader Subsystem

The part of a Java virtual machine implementation that takes care of finding and loading types is the *class loader subsystem*. Chapter 1, "Introduction to Java's Architecture," gives an overview of this subsystem. Chapter 3, "Security," shows how the subsystem fits into Java's security model. This chapter describes the class loader subsystem in more detail and show how it relates to the other components of the virtual machine's internal architecture.

As mentioned in Chapter 1, the Java virtual machine contains two kinds of class loaders: a *bootstrap class loader* and *user-defined class loaders*. The bootstrap class loader is a part of the virtual machine implementation, and user-defined class loaders are part of the running Java application. Classes loaded by different class loaders are placed into separate *name spaces* inside the Java virtual machine.

The class loader subsystem involves many other parts of the Java virtual machine and several classes from the `java.lang` library. For example, user-defined class loaders are regular Java objects whose class descends from `java.lang.ClassLoader`. The methods of class `ClassLoader` allow Java applications to access the virtual machine's class loading machinery. Also, for every type a Java virtual machine loads, it creates an instance of class `java.lang.Class` to represent that type. Like all objects, user-defined class loaders and instances of class `Class` reside on the heap. Data for loaded types resides in the method area.

Loading, Linking and Initialization

The class loader subsystem is responsible for more than just locating and importing the binary data for classes. It must also verify the correctness of imported classes, allocate and initialize memory for class variables, and assist in the resolution of symbolic references. These activities are performed in a strict order:

1. Loading: finding and importing the binary data for a type
2. Linking: performing verification, preparation, and (optionally) resolution
 - a. Verification: ensuring the correctness of the imported type
 - b. Preparation: allocating memory for class variables and initializing the memory to default values
 - c. Resolution: transforming symbolic references from the type into direct references.
3. Initialization: invoking Java code that initializes class variables to their proper starting values.

The details of these processes are given Chapter 7, "The Lifetime of a Type."

The Bootstrap Class Loader

Java virtual machine implementations must be able to recognize and load classes and interfaces stored in binary files that conform to the Java class file format. An implementation is free to recognize other binary forms besides class files, but it must recognize class files.

Every Java virtual machine implementation has a bootstrap class loader, which knows how to load trusted classes, including the classes of the Java API. The Java virtual machine specification doesn't define how the bootstrap loader should locate classes. That is another

decision the specification leaves to implementation designers.

Given a fully qualified type name, the bootstrap class loader must *in some way* attempt to produce the data that defines the type. One common approach is demonstrated by the Java virtual machine implementation in Sun's 1.1 JDK on Windows98. This implementation searches a user-defined directory path stored in an environment variable named CLASSPATH. The bootstrap loader looks in each directory, in the order the directories appear in the CLASSPATH, until it finds a file with the appropriate name: the type's simple name plus ".class". Unless the type is part of the unnamed package, the bootstrap loader expects the file to be in a subdirectory of one the directories in the CLASSPATH. The path name of the subdirectory is built from the package name of the type. For example, if the bootstrap class loader is searching for class `java.lang.Object`, it will look for `Object.class` in the `java\lang` subdirectory of each CLASSPATH directory.

In 1.2, the bootstrap class loader of Sun's Java 2 SDK only looks in the directory in which the system classes (the class files of the Java API) were installed. The bootstrap class loader of the implementation of the Java virtual machine from Sun's Java 2 SDK does not look on the CLASSPATH. In Sun's Java 2 SDK virtual machine, searching the class path is the job of the *system class loader*, a user-defined class loader that is created automatically when the virtual machine starts up. More information on the class loading scheme of Sun's Java 2 SDK is given in Chapter 8, "The Linking Model."

User-Defined Class Loaders

Although user-defined class loaders themselves are part of the Java application, four of the methods in class `ClassLoader` are gateways into the Java virtual machine:

```
// Four of the methods declared in class java.lang.ClassLoader:
protected final Class defineClass(String name, byte data[],
    int offset, int length);
protected final Class defineClass(String name, byte data[],
    int offset, int length, ProtectionDomain protectionDomain);
protected final Class findSystemClass(String name);
protected final void resolveClass(Class c);
```

Any Java virtual machine implementation must take care to connect these methods of class `ClassLoader` to the internal class loader subsystem.

The two overloaded `defineClass()` methods accept a byte array, `data[]`, as input. Starting at position `offset` in the array and continuing for `length` bytes, class `ClassLoader` expects binary data conforming to the Java class file format--binary data that represents a new type for the running application -- with the fully qualified name specified in `name`. The type is assigned to either a default protection domain, if the first version of `defineClass()` is used, or to the protection domain object referenced by the `protectionDomain` parameter. Every Java virtual machine implementation must make sure the `defineClass()` method of class `ClassLoader` can cause a new type to be imported into the method area.

The `findSystemClass()` method accepts a `String` representing a fully qualified name of a type. When a user-defined class loader invokes this method in version 1.0 and 1.1, it is requesting that the virtual machine attempt to load the named type via its bootstrap class loader. If the bootstrap class loader has already loaded or successfully loads the type, it returns a reference to the `Class` object representing the type. If it can't locate the binary data for the type, it throws `ClassNotFoundException`. In version 1.2, the `findSystemClass()` method attempts to load the requested type from the system class loader. Every Java virtual machine implementation must make sure the `findSystemClass()` method can invoke the bootstrap (if version 1.0 or 1.1) or system (if version 1.2 or later) class loader in this way.

The `resolveClass()` method accepts a reference to a `Class` instance. This method causes the type represented by the `Class` instance to be linked (if it hasn't already been linked). The `defineClass()` method, described previous, only takes care of loading. (See the previous section, "Loading, Linking, and Initialization" for definitions of these terms.) When `defineClass()` returns a `Class` instance, the binary file for the type has definitely been located and imported into the method area, but not necessarily linked and initialized. Java virtual machine implementations make sure the `resolveClass()` method of class `ClassLoader` can cause the class loader subsystem to perform linking.

The details of how a Java virtual machine performs class loading, linking, and initialization, with user-defined class loaders is given in Chapter 8, "The Linking Model."

Name Spaces

As mentioned in Chapter 3, "Security," each class loader maintains its own name space populated by the types it has loaded. Because each class loader has its own name space, a single Java application can load multiple types with the same fully qualified name. A type's fully qualified name, therefore, is not always enough to uniquely identify it inside a Java virtual machine instance. If multiple types of that same name have been loaded into different name spaces, the identity of the class loader that loaded the type (the identity of the name space it is in) will also be needed to uniquely identify that type.

Name spaces arise inside a Java virtual machine instance as a result of the process of resolution. As part of the data for each loaded type, the Java virtual machine keeps track of the class loader that imported the type. When the virtual machine needs to resolve a symbolic reference from one class to another, it requests the referenced class from the same class loader that loaded the referencing class. This process is described in detail in Chapter 8, "The Linking Model."

The Method Area

Inside a Java virtual machine instance, information about loaded types is stored in a logical area of memory called the method area. When the Java virtual machine loads a type, it uses a class loader to locate the appropriate class file. The class loader reads in the class file--a linear stream of binary data--and passes it to the virtual machine. The virtual machine extracts information about the type from the binary data and stores the information in the method area. Memory for class (static) variables declared in the class is also taken from the method area.

The manner in which a Java virtual machine implementation represents type information internally is a decision of the implementation designer. For example, multi-byte quantities in class files are stored in big- endian (most significant byte first) order. When the data is imported into the method area, however, a virtual machine can store the data in any manner. If an implementation sits on top of a little-endian processor, the designers may decide to store multi-byte values in the method area in little-endian order.

The virtual machine will search through and use the type information stored in the method area as it executes the application it is hosting. Designers must attempt to devise data structures that will facilitate speedy execution of the Java application, but must also think of compactness. If designing an implementation that will operate under low memory constraints, designers may decide to trade off some execution speed in favor of compactness. If designing an implementation that will run on a virtual memory system, on the other hand, designers may decide to store redundant information in the method area to facilitate execution speed. (If the underlying host doesn't offer virtual memory, but does offer a hard disk, designers could create their own virtual memory system as part of their implementation.) Designers can choose whatever data structures and organization they feel optimize their implementations performance, in the context of its requirements.

All threads share the same method area, so access to the method area's data structures must be designed to be thread-safe. If two threads are attempting to find a class named `Lava`, for example, and `Lava` has not yet been loaded, only one thread should be allowed to load it while the other one waits.

The size of the method area need not be fixed. As the Java application runs, the virtual machine can expand and contract the method area to fit the application's needs. Also, the memory of the method area need not be contiguous. It could be allocated on a heap--even on the virtual machine's own heap. Implementations may allow users or programmers to specify an initial size for the method area, as well as a maximum or minimum size.

The method area can also be garbage collected. Because Java programs can be dynamically extended via user-defined class loaders, classes can become "unreferenced" by the application. If a class becomes unreferenced, a Java virtual machine can unload the class (garbage collect it) to keep the memory occupied by the method area at a minimum. The unloading of classes--including the conditions under which a class can become "unreferenced"--is described in Chapter 7, "The Lifetime of a Type."

Type Information

For each type it loads, a Java virtual machine must store the following kinds of information in the method area:

- The fully qualified name of the type
- The fully qualified name of the type's direct superclass (unless the type is an interface or class `java.lang.Object`, neither of which have a superclass)
- Whether or not the type is a class or an interface
- The type's modifiers (some subset of `public`, `abstract`, `final`)
- An ordered list of the fully qualified names of any direct superinterfaces

Inside the Java class file and Java virtual machine, type names are always stored as *fully qualified names*. In Java source code, a fully qualified name is the name of a type's package, plus a dot, plus the type's *simple name*. For example, the fully qualified name of class `Object` in package `java.lang` is `java.lang.Object`. In class files, the dots are replaced by slashes, as in `java/lang/Object`. In the method area, fully qualified names can be represented in whatever form and data structures a designer chooses.

In addition to the basic type information listed previously, the virtual machine must also store for each loaded type:

- The constant pool for the type
- Field information
- Method information
- All class (static) variables declared in the type, except constants
- A reference to class `ClassLoader`
- A reference to class `Class`

This data is described in the following sections.

The Constant Pool

For each type it loads, a Java virtual machine must store a *constant pool*. A constant pool is an ordered set of constants used by the type, including literals (string, integer, and floating point constants) and symbolic references to types, fields, and methods. Entries in the constant pool are referenced by index, much like the elements of an array. Because it holds symbolic references to all types, fields, and methods used by a type, the constant pool plays a central role in the dynamic linking of Java programs. The constant pool is described in more detail later in this chapter and in Chapter 6, "The Java Class File."

Field Information

For each field declared in the type, the following information must be stored in the method area. In addition to the information for each field, the order in which the fields are declared by the class or interface must also be recorded. Here's the list for fields:

- The field's name
- The field's type
- The field's modifiers (some subset of `public`, `private`, `protected`, `static`, `final`, `volatile`, `transient`)

Method Information

For each method declared in the type, the following information must be stored in the method area. As with fields, the order in which the methods are declared by the class or interface must be recorded as well as the data. Here's the list:

- The method's name
- The method's return type (or `void`)
- The number and types (in order) of the method's parameters
- The method's modifiers (some subset of `public`, `private`, `protected`, `static`, `final`, `synchronized`, `native`, `abstract`)

In addition to the items listed previously, the following information must also be stored with each method that is not abstract or native:

- The method's bytecodes
- The sizes of the operand stack and local variables sections of the method's stack frame (these are described in a later section of this chapter)
- An exception table (this is described in Chapter 17, "Exceptions")

Class Variables

Class variables are shared among all instances of a class and can be accessed even in the absence of any instance. These variables are associated with the class--not with instances of the class--so they are logically part of the class data in the method area. Before a Java virtual machine uses a class, it must allocate memory from the method area for each non-final class variable declared in the class.

Constants (class variables declared `final`) are not treated in the same way as non-final class variables. Every type that uses a final class variable gets a copy of the constant value in its own constant pool. As part of the constant pool, final class variables are stored in the method area--just like non-final class variables. But whereas non-final class variables are stored as part of the data for the type that *declares* them, final class variables are stored as part of the data for any type that *uses* them. This special treatment of constants is explained in more detail in Chapter 6, "The Java Class File."

A Reference to Class `ClassLoader`

For each type it loads, a Java virtual machine must keep track of whether or not the type was loaded via the bootstrap class loader or a user-defined class loader. For those types loaded via a user-defined class loader, the virtual machine must store a reference to the user-defined class loader that loaded the type. This information is stored as part of the type's data in the method area.

The virtual machine uses this information during dynamic linking. When one type refers to another type, the virtual machine requests the referenced type from the same class loader that loaded the referencing type. This process of dynamic linking is also central to the way the virtual machine forms separate name spaces. To be able to properly perform dynamic linking and maintain multiple name spaces, the virtual machine needs to know what class loader loaded each type in its method area. The details of dynamic linking and name spaces are given in Chapter 8, "The Linking Model."

A Reference to Class `Class`

An instance of class `java.lang.Class` is created by the Java virtual machine for every type it loads. The virtual machine must in some way associate a reference to the `Class` instance for a type with the type's data in the method area.

Your Java programs can obtain and use references to `Class` objects. One static method in class `Class`, allows you to get a reference to the `Class` instance for any loaded class:

```
// A method declared in class java.lang.Class:
public static Class forName(String className);
```

If you invoke `forName("java.lang.Object")`, for example, you will get a reference to the `Class` object that represents `java.lang.Object`. If you invoke `forName("java.util.Enumeration")`, you will get a reference to the `Class` object that represents the `Enumeration` interface from the `java.util` package. You can use `forName()` to get a `Class` reference for any loaded type from any package, so long as the type can be (or already has been) loaded into the current name space. If the virtual machine is unable to load the requested type into the current name space, `forName()` will throw `ClassNotFoundException`.

An alternative way to get a `Class` reference is to invoke `getClass()` on any object reference. This method is inherited by every object from class `Object` itself:

```
// A method declared in class java.lang.Object:
public final Class getClass();
```

If you have a reference to an object of class `java.lang.Integer`, for example, you could get the `Class` object for `java.lang.Integer` simply by invoking `getClass()` on your reference to the `Integer` object.

Given a reference to a `Class` object, you can find out information about the type by invoking methods declared in class `Class`. If you

look at these methods, you will quickly realize that class `Class` gives the running application access to the information stored in the method area. Here are some of the methods declared in class `Class`:

```
// Some of the methods declared in class java.lang.Class:
public String getName();
public Class getSuperClass();
public boolean isInterface();
public Class[] getInterfaces();
public ClassLoader getClassLoader();
```

These methods just return information about a loaded type. `getName()` returns the fully qualified name of the type. `getSuperClass()` returns the `Class` instance for the type's direct superclass. If the type is class `java.lang.Object` or an interface, none of which have a superclass, `getSuperClass()` returns `null`. `isInterface()` returns `true` if the `Class` object describes an interface, `false` if it describes a class. `getInterfaces()` returns an array of `Class` objects, one for each direct superinterface. The superinterfaces appear in the array in the order they are declared as superinterfaces by the type. If the type has no direct superinterfaces, `getInterfaces()` returns an array of length zero. `getClassLoader()` returns a reference to the `ClassLoader` object that loaded this type, or `null` if the type was loaded by the bootstrap class loader. All this information comes straight out of the method area.

Method Tables

The type information stored in the method area must be organized to be quickly accessible. In addition to the raw type information listed previously, implementations may include other data structures that speed up access to the raw data. One example of such a data structure is a *method table*. For each non-abstract class a Java virtual machine loads, it could generate a method table and include it as part of the class information it stores in the method area. A method table is an array of direct references to all the instance methods that may be invoked on a class instance, including instance methods inherited from superclasses. (A method table isn't helpful in the case of abstract classes or interfaces, because the program will never instantiate these.) A method table allows a virtual machine to quickly locate an instance method invoked on an object. Method tables are described in detail in Chapter 8, "The Linking Model."

An Example of Method Area Use

As an example of how the Java virtual machine uses the information it stores in the method area, consider these classes:

```
// On CD-ROM in file jvm/ex2/Lava.java
class Lava {

    private int speed = 5; // 5 kilometers per hour

    void flow() {
    }
}

// On CD-ROM in file jvm/ex2/Volcano.java
class Volcano {

    public static void main(String[] args) {
        Lava lava = new Lava();
        lava.flow();
    }
}
```

The following paragraphs describe how an implementation might execute the first instruction in the bytecodes for the `main()` method of the `Volcano` application. Different implementations of the Java virtual machine can operate in very different ways. The following description illustrates one way--but not the only way--a Java virtual machine could execute the first instruction of `Volcano`'s `main()` method.

To run the `Volcano` application, you give the name "Volcano" to a Java virtual machine in an implementation-dependent manner. Given the name `Volcano`, the virtual machine finds and reads in file `Volcano.class`. It extracts the definition of class `Volcano` from the binary data in the imported class file and places the information into the method area. The virtual machine then invokes the `main()` method, by interpreting the bytecodes stored in the method area. As the virtual machine executes `main()`, it maintains a pointer to the constant pool (a data structure in the method area) for the current class (class `Volcano`).

Note that this Java virtual machine has already begun to execute the bytecodes for `main()` in class `Volcano` even though it hasn't yet loaded class `Lava`. Like many (probably most) implementations of the Java virtual machine, this implementation doesn't wait until all classes used by the application are loaded before it begins executing `main()`. It loads classes only as it needs them.

`main()`'s first instruction tells the Java virtual machine to allocate enough memory for the class listed in constant pool entry one. The virtual machine uses its pointer into `Volcano`'s constant pool to look up entry one and finds a symbolic reference to class `Lava`. It checks the method area to see if `Lava` has already been loaded.

The symbolic reference is just a string giving the class's fully qualified name: "Lava". Here you can see that the method area must be organized so a class can be located--as quickly as possible--given only the class's fully qualified name. Implementation designers can choose whatever algorithm and data structures best fit their needs--a hash table, a search tree, anything. This same mechanism can be

used by the static `forName()` method of class `Class`, which returns a `Class` reference given a fully qualified name.

When the virtual machine discovers that it hasn't yet loaded a class named "Lava," it proceeds to find and read in file `Lava.class`. It extracts the definition of class `Lava` from the imported binary data and places the information into the method area.

The Java virtual machine then replaces the symbolic reference in `Volcano`'s constant pool entry one, which is just the string "Lava", with a pointer to the class data for `Lava`. If the virtual machine ever has to use `Volcano`'s constant pool entry one again, it won't have to go through the relatively slow process of searching through the method area for class `Lava` given only a symbolic reference, the string "Lava". It can just use the pointer to more quickly access the class data for `Lava`. This process of replacing symbolic references with direct references (in this case, a native pointer) is called *constant pool resolution*. The symbolic reference is *resolved* into a direct reference by searching through the method area until the referenced entity is found, loading new classes if necessary.

Finally, the virtual machine is ready to actually allocate memory for a new `Lava` object. Once again, the virtual machine consults the information stored in the method area. It uses the pointer (which was just put into `Volcano`'s constant pool entry one) to the `Lava` data (which was just imported into the method area) to find out how much heap space is required by a `Lava` object.

A Java virtual machine can always determine the amount of memory required to represent an object by looking into the class data stored in the method area. The actual amount of heap space required by a particular object, however, is implementation-dependent. The internal representation of objects inside a Java virtual machine is another decision of implementation designers. Object representation is discussed in more detail later in this chapter.

Once the Java virtual machine has determined the amount of heap space required by a `Lava` object, it allocates that space on the heap and initializes the instance variable `speed` to zero, its default initial value. If class `Lava`'s superclass, `Object`, has any instance variables, those are also initialized to default initial values. (The details of initialization of both classes and objects are given in Chapter 7, "The Lifetime of a Type.")

The first instruction of `main()` completes by pushing a reference to the new `Lava` object onto the stack. A later instruction will use the reference to invoke Java code that initializes the `speed` variable to its proper initial value, five. Another instruction will use the reference to invoke the `flow()` method on the referenced `Lava` object.

The Heap

Whenever a class instance or array is created in a running Java application, the memory for the new object is allocated from a single heap. As there is only one heap inside a Java virtual machine instance, all threads share it. Because a Java application runs inside its "own" exclusive Java virtual machine instance, there is a separate heap for every individual running application. There is no way two different Java applications could trample on each other's heap data. Two different threads of the same application, however, could trample on each other's heap data. This is why you must be concerned about proper synchronization of multi-threaded access to objects (heap data) in your Java programs.

The Java virtual machine has an instruction that allocates memory on the heap for a new object, but has no instruction for freeing that memory. Just as you can't explicitly free an object in Java source code, you can't explicitly free an object in Java bytecodes. The virtual machine itself is responsible for deciding whether and when to free memory occupied by objects that are no longer referenced by the running application. Usually, a Java virtual machine implementation uses a *garbage collector* to manage the heap.

Garbage Collection

A garbage collector's primary function is to automatically reclaim the memory used by objects that are no longer referenced by the running application. It may also move objects as the application runs to reduce heap fragmentation.

A garbage collector is not strictly required by the Java virtual machine specification. The specification only requires that an implementation manage its own heap *in some manner*. For example, an implementation could simply have a fixed amount of heap space available and throw an `OutOfMemory` exception when that space fills up. While this implementation may not win many prizes, it does qualify as a Java virtual machine. The Java virtual machine specification does not say how much memory an implementation must make available to running programs. It does not say how an implementation must manage its heap. It says to implementation designers only that the program will be allocating memory from the heap, but not freeing it. It is up to designers to figure out how they want to deal with that fact.

No garbage collection technique is dictated by the Java virtual machine specification. Designers can use whatever techniques seem most appropriate given their goals, constraints, and talents. Because references to objects can exist in many places--Java Stacks, the heap, the method area, native method stacks--the choice of garbage collection technique heavily influences the design of an implementation's runtime data areas. Various garbage collection techniques are described in Chapter 9, "Garbage Collection."

As with the method area, the memory that makes up the heap need not be contiguous, and may be expanded and contracted as the running program progresses. An implementation's method area could, in fact, be implemented on top of its heap. In other words, when a virtual machine needs memory for a freshly loaded class, it could take that memory from the same heap on which objects reside. The same garbage collector that frees memory occupied by unreferenced objects could take care of finding and freeing (unloading) unreferenced classes. Implementations may allow users or programmers to specify an initial size for the heap, as well as a maximum and minimum size.

Object Representation

The Java virtual machine specification is silent on how objects should be represented on the heap. Object representation--an integral aspect of the overall design of the heap and garbage collector--is a decision of implementation designers

The primary data that must in some way be represented for each object is the instance variables declared in the object's class and all its superclasses. Given an object reference, the virtual machine must be able to quickly locate the instance data for the object. In addition, there must be some way to access an object's class data (stored in the method area) given a reference to the object. For this reason, the memory allocated for an object usually includes some kind of pointer into the method area.

One possible heap design divides the heap into two parts: a handle pool and an object pool. An object reference is a native pointer to a handle pool entry. A handle pool entry has two components: a pointer to instance data in the object pool and a pointer to class data in the method area. The advantage of this scheme is that it makes it easy for the virtual machine to combat heap fragmentation. When the virtual machine moves an object in the object pool, it need only update one pointer with the object's new address: the relevant pointer in the handle pool. The disadvantage of this approach is that every access to an object's instance data requires dereferencing two pointers. This approach to object representation is shown graphically in Figure 5-5. This kind of heap is demonstrated interactively by the HeapOfFish applet, described in Chapter 9, "Garbage Collection."

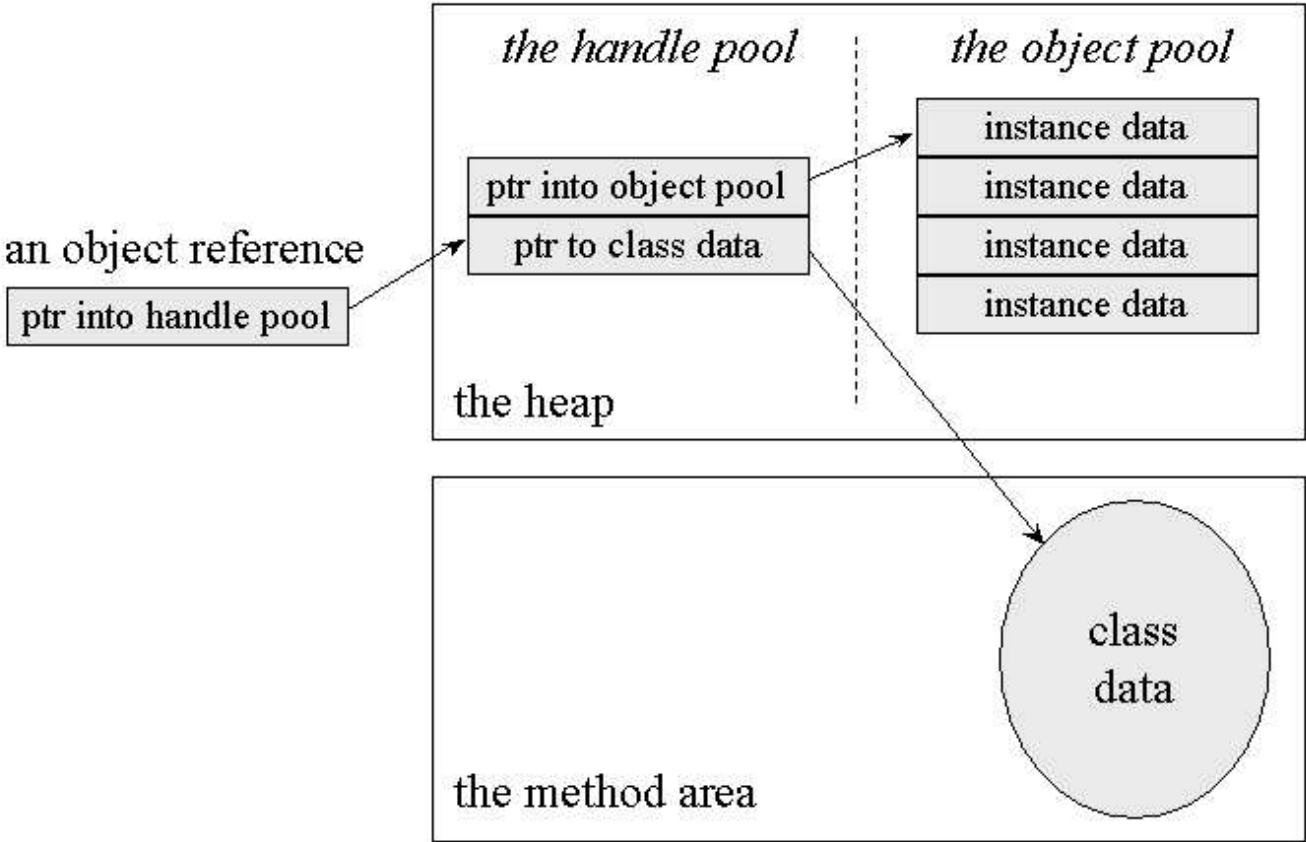
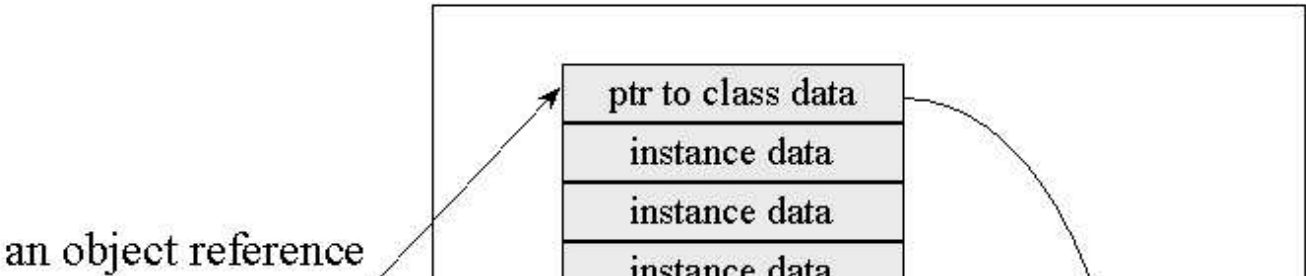


Figure 5-5. Splitting an object across a handle pool and object pool.

Another design makes an object reference a native pointer to a bundle of data that contains the object's instance data and a pointer to the object's class data. This approach requires dereferencing only one pointer to access an object's instance data, but makes moving objects more complicated. When the virtual machine moves an object to combat fragmentation of this kind of heap, it must update every reference to that object anywhere in the runtime data areas. This approach to object representation is shown graphically in Figure 5-6.



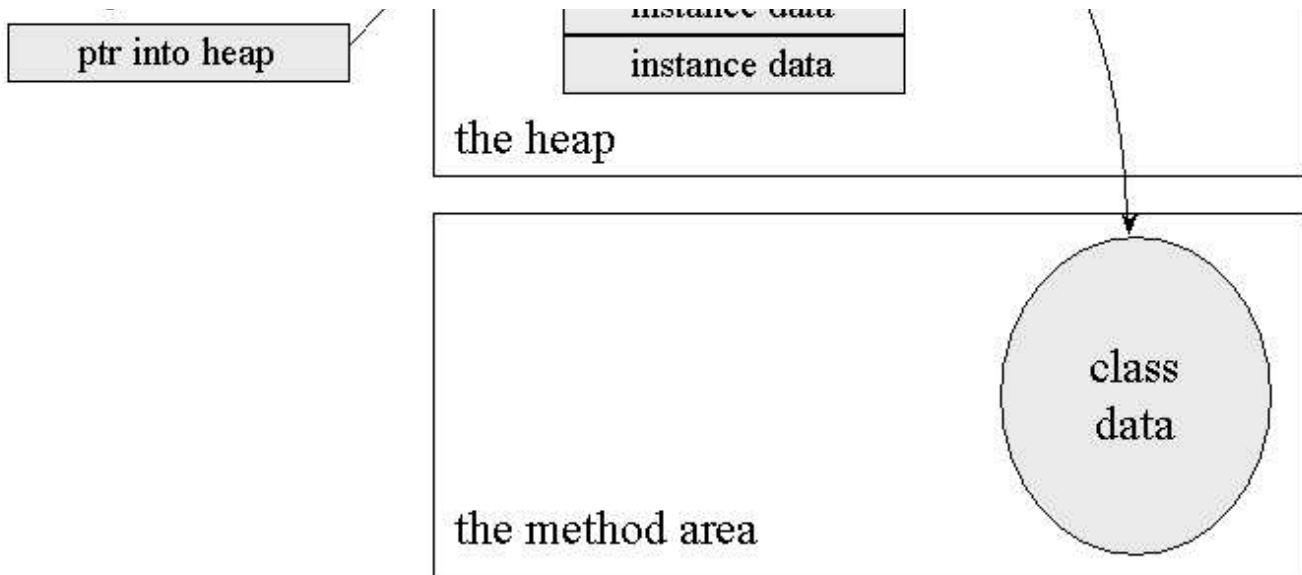


Figure 5-6. Keeping object data all in one place.

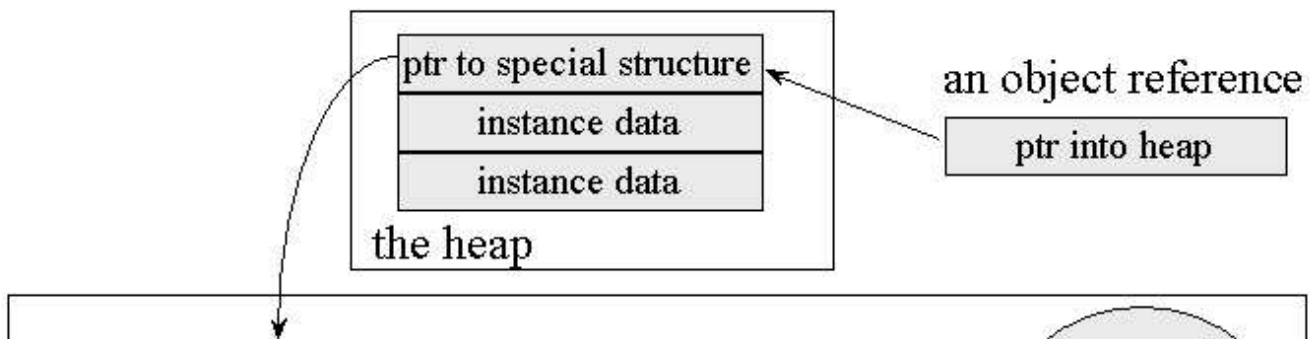
The virtual machine needs to get from an object reference to that object's class data for several reasons. When a running program attempts to cast an object reference to another type, the virtual machine must check to see if the type being cast to is the actual class of the referenced object or one of its supertypes. It must perform the same kind of check when a program performs an `instanceof` operation. In either case, the virtual machine must look into the class data of the referenced object. When a program invokes an instance method, the virtual machine must perform dynamic binding: it must choose the method to invoke based not on the type of the reference but on the class of the object. To do this, it must once again have access to the class data given only a reference to the object.

No matter what object representation an implementation uses, it is likely that a method table is close at hand for each object. Method tables, because they speed up the invocation of instance methods, can play an important role in achieving good overall performance for a virtual machine implementation. Method tables are not required by the Java virtual machine specification and may not exist in all implementations. Implementations that have extremely low memory requirements, for instance, may not be able to afford the extra memory space method tables occupy. If an implementation does use method tables, however, an object's method table will likely be quickly accessible given just a reference to the object.

One way an implementation could connect a method table to an object reference is shown graphically in Figure 5-7. This figure shows that the pointer kept with the instance data for each object points to a special structure. The special structure has two components:

- A pointer to the full the class data for the object
- The method table for the object The method table is an array of pointers to the data for each instance method that can be invoked on objects of that class. The method data pointed to by method table includes:
 - The sizes of the operand stack and local variables sections of the method's stack
 - The method's bytecodes
 - An exception table

This gives the virtual machine enough information to invoke the method. The method table include pointers to data for methods declared explicitly in the object's class or inherited from superclasses. In other words, the pointers in the method table may point to methods defined in the object's class or any of its superclasses. More information on method tables is given in Chapter 8, "The Linking Model."



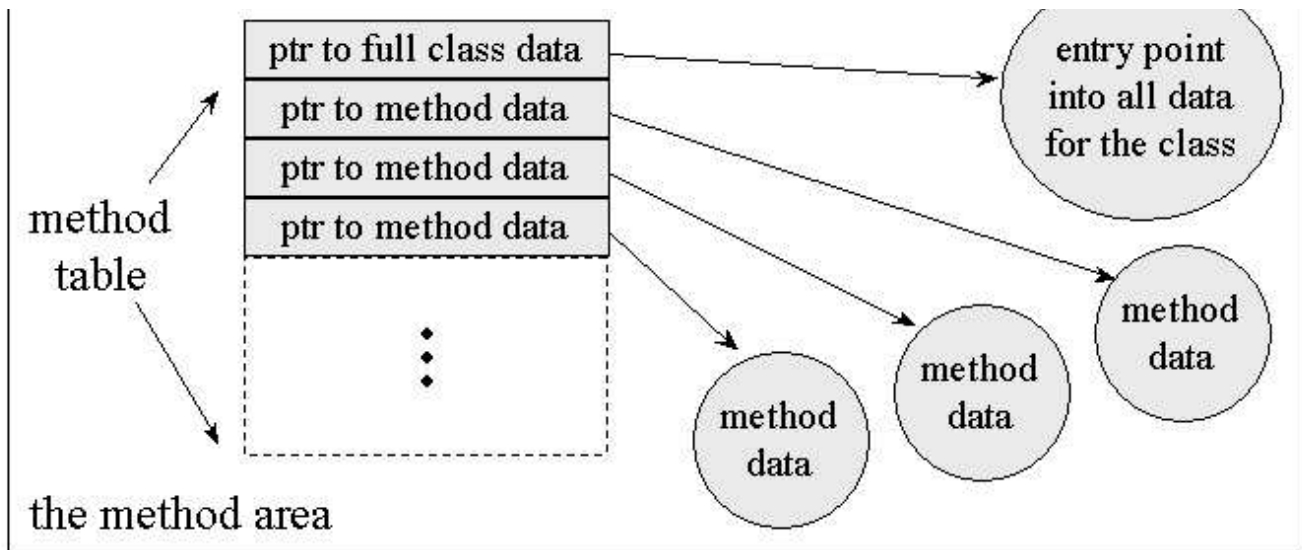


Figure 5-7. Keeping the method table close at hand.

If you are familiar with the inner workings of C++, you may recognize the method table as similar to the VTBL or virtual table of C++ objects. In C++, objects are represented by their instance data plus an array of pointers to any virtual functions that can be invoked on the object. This approach could also be taken by a Java virtual machine implementation. An implementation could include a copy of the method table for a class as part of the heap image for every instance of that class. This approach would consume more heap space than the approach shown in Figure 5-7, but might yield slightly better performance on a systems that enjoy large quantities of available memory.

One other kind of data that is not shown in Figures 5-5 and 5-6, but which is logically part of an object's data on the heap, is the object's *lock*. Each object in a Java virtual machine is associated with a lock (or *mutex*) that a program can use to coordinate multi-threaded access to the object. Only one thread at a time can "own" an object's lock. While a particular thread owns a particular object's lock, only that thread can access that object's instance variables. All other threads that attempt to access the object's variables have to wait until the owning thread releases the object's lock. If a thread requests a lock that is already owned by another thread, the requesting thread has to wait until the owning thread releases the lock. Once a thread owns a lock, it can request the same lock again multiple times, but then has to release the lock the same number of times before it is made available to other threads. If a thread requests a lock three times, for example, that thread will continue to own the lock until it has released it three times.

Many objects will go through their entire lifetimes without ever being locked by a thread. The data required to implement an object's lock is not needed unless the lock is actually requested by a thread. As a result, many implementations, such as the ones shown in Figure 5-5 and 5-6, may not include a pointer to "lock data" within the object itself. Such implementations must create the necessary data to represent a lock when the lock is requested for the first time. In this scheme, the virtual machine must associate the lock with the object in some indirect way, such as by placing the lock data into a search tree based on the object's address.

Along with data that implements a lock, every Java object is logically associated with data that implements a *wait set*. Whereas locks help threads to work independently on shared data without interfering with one another, wait sets help threads to cooperate with one another--to work together towards a common goal.

Wait sets are used in conjunction with wait and notify methods. Every class inherits from `Object` three "wait methods" (overloaded forms of a method named `wait()`) and two "notify methods" (`notify()` and `notifyAll()`). When a thread invokes a wait method on an object, the Java virtual machine suspends that thread and adds it to that object's wait set. When a thread invokes a notify method on an object, the virtual machine will at some future time wake up one or more threads from that object's wait set. As with the data that implements an object's lock, the data that implements an object's wait set is not needed unless a wait or notify method is actually invoked on the object. As a result, many implementations of the Java virtual machine may keep the wait set data separate from the actual object data. Such implementations could allocate the data needed to represent an object's wait set when a wait or notify method is first invoked on that object by the running application. For more information about locks and wait sets, see Chapter 20, "Thread Synchronization."

One last example of a type of data that may be included as part of the image of an object on the heap is any data needed by the garbage collector. The garbage collector must in some way keep track of which objects are referenced by the program. This task invariably requires data to be kept for each object on the heap. The kind of data required depends upon the garbage collection technique being used. For example, if an implementation uses a *mark and sweep* algorithm, it must be able to mark an object as referenced or unreferenced. For each unreferenced object, it may also need to indicate whether or not the object's finalizer has been run. As with thread locks, this data may be kept separate from the object image. Some garbage collection techniques only require this extra data while the garbage collector is actually running. A mark and sweep algorithm, for instance, could potentially use a separate bitmap for marking referenced and unreferenced objects. More detail on various garbage collection techniques, and the data that is required by each of them, is given in Chapter 9, "Garbage Collection."

In addition to data that a garbage collector uses to distinguish between reference and unreferenced objects, a garbage collector needs data to keep track of which objects on which it has already executed a finalizer. Garbage collectors must run the finalizer of any object whose class declares one before it reclaims the memory occupied by that object. The Java language specification states that a garbage collector will only execute an object's finalizer once, but allows that finalizer to "resurrect" the object: to make the object referenced again. When the object becomes unreferenced for a second time, the garbage collector must not finalize it again. Because most objects will likely not have a finalizer, and very few of those will resurrect their objects, this scenario of garbage collecting the same object twice will probably be extremely rare. As a result, the data used to keep track of objects that have already been finalized, though logically part of the data associated with an object, will likely not be part of the object representation on the heap. In most cases, garbage collectors will keep this information in a separate place. Chapter 9, "Garbage Collection," gives more information about finalization.

Array Representation

In Java, arrays are full-fledged objects. Like objects, arrays are always stored on the heap. Also like objects, implementation designers can decide how they want to represent arrays on the heap.

Arrays have a `Class` instance associated with their class, just like any other object. All arrays of the same dimension and type have the same class. The length of an array (or the lengths of each dimension of a multidimensional array) does not play any role in establishing the array's class. For example, an array of three `ints` has the same class as an array of three hundred `ints`. The length of an array is considered part of its instance data.

The name of an array's class has one open square bracket for each dimension plus a letter or string representing the array's type. For example, the class name for an array of `ints` is "`[I`". The class name for a three-dimensional array of `bytes` is "`[[[B`". The class name for a two-dimensional array of `Objects` is "`[[Ljava.lang.Object`". The full details of this naming convention for array classes is given in Chapter 6, "The Java Class File."

Multi-dimensional arrays are represented as arrays of arrays. A two dimensional array of `ints`, for example, would be represented by a one dimensional array of references to several one dimensional arrays of `ints`. This is shown graphically in Figure 5-8.

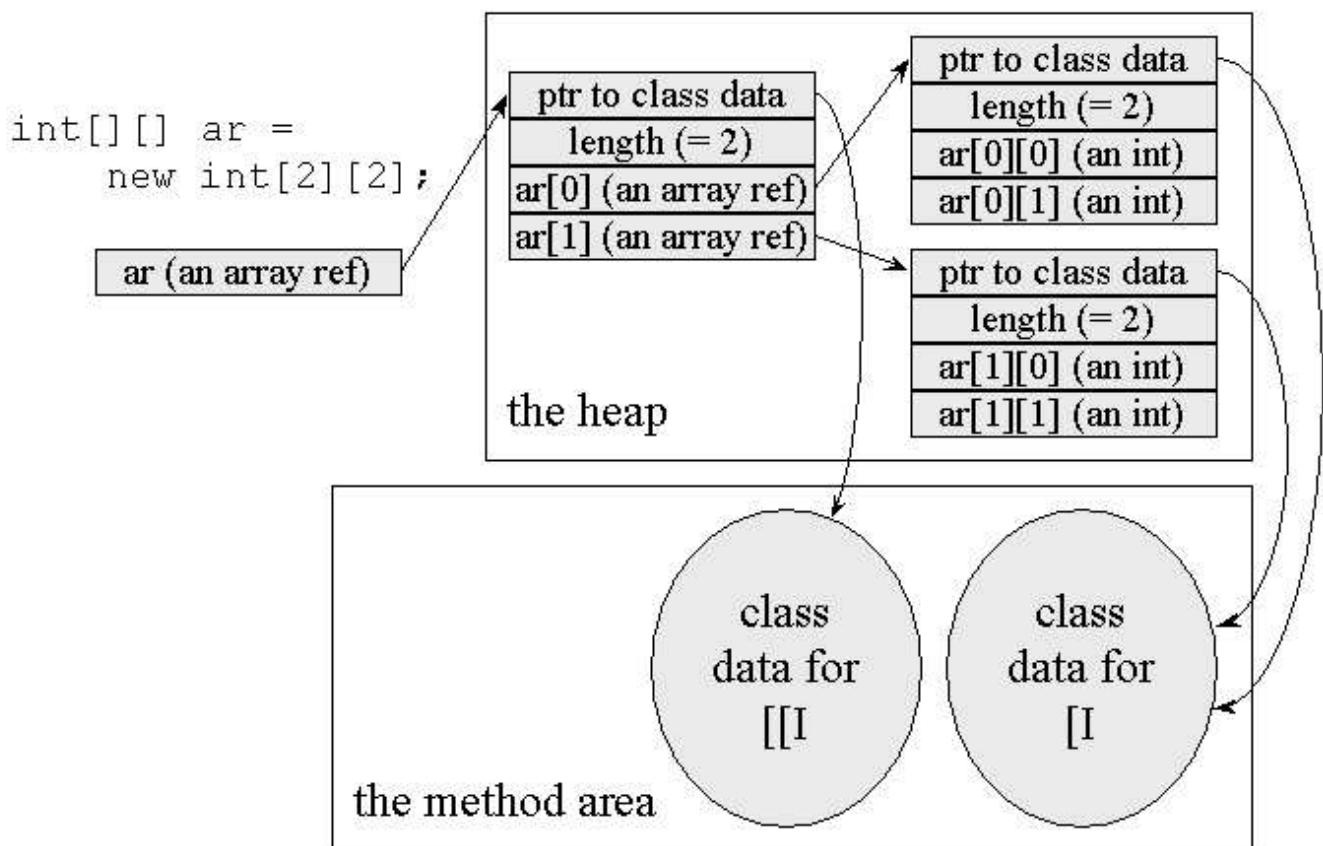


Figure 5-8. One possible heap representation for arrays.

The data that must be kept on the heap for each array is the array's length, the array data, and some kind of reference to the array's class data. Given a reference to an array, the virtual machine must be able to determine the array's length, to get and set its elements by index

(checking to make sure the array bounds are not exceeded), and to invoke any methods declared by `Object`, the direct superclass of all arrays.

The Program Counter

Each thread of a running program has its own pc register, or program counter, which is created when the thread is started. The pc register is one word in size, so it can hold both a native pointer and a `returnAddress`. As a thread executes a Java method, the pc register contains the address of the current instruction being executed by the thread. An "address" can be a native pointer or an offset from the beginning of a method's bytecodes. If a thread is executing a native method, the value of the pc register is undefined.

The Java Stack

When a new thread is launched, the Java virtual machine creates a new Java stack for the thread. As mentioned earlier, a Java stack stores a thread's state in discrete frames. The Java virtual machine only performs two operations directly on Java Stacks: it pushes and pops frames.

The method that is currently being executed by a thread is the thread's *current method*. The stack frame for the current method is the *current frame*. The class in which the current method is defined is called the *current class*, and the current class's constant pool is the *current constant pool*. As it executes a method, the Java virtual machine keeps track of the current class and current constant pool. When the virtual machine encounters instructions that operate on data stored in the stack frame, it performs those operations on the current frame.

When a thread invokes a Java method, the virtual machine creates and pushes a new frame onto the thread's Java stack. This new frame then becomes the current frame. As the method executes, it uses the frame to store parameters, local variables, intermediate computations, and other data.

A method can complete in either of two ways. If a method completes by returning, it is said to have *normal completion*. If it completes by throwing an exception, it is said to have *abrupt completion*. When a method completes, whether normally or abruptly, the Java virtual machine pops and discards the method's stack frame. The frame for the previous method then becomes the current frame.

All the data on a thread's Java stack is private to that thread. There is no way for a thread to access or alter the Java stack of another thread. Because of this, you need never worry about synchronizing multi-threaded access to local variables in your Java programs. When a thread invokes a method, the method's local variables are stored in a frame on the invoking thread's Java stack. Only one thread can ever access those local variables: the thread that invoked the method.

Like the method area and heap, the Java stack and stack frames need not be contiguous in memory. Frames could be allocated on a contiguous stack, or they could be allocated on a heap, or some combination of both. The actual data structures used to represent the Java stack and stack frames is a decision of implementation designers. Implementations may allow users or programmers to specify an initial size for Java stacks, as well as a maximum or minimum size.

The Stack Frame

The stack frame has three parts: local variables, operand stack, and frame data. The sizes of the local variables and operand stack, which are measured in words, depend upon the needs of each individual method. These sizes are determined at compile time and included in the class file data for each method. The size of the frame data is implementation dependent.

When the Java virtual machine invokes a Java method, it checks the class data to determine the number of words required by the method in the local variables and operand stack. It creates a stack frame of the proper size for the method and pushes it onto the Java stack.

Local Variables

The local variables section of the Java stack frame is organized as a zero-based array of words. Instructions that use a value from the local variables section provide an index into the zero-based array. Values of type `int`, `float`, `reference`, and `returnAddress` occupy one entry in the local variables array. Values of type `byte`, `short`, and `char` are converted to `int` before being stored into the local variables. Values of type `long` and `double` occupy two consecutive entries in the array.

To refer to a long or double in the local variables, instructions provide the index of the first of the two consecutive entries occupied by the value. For example, if a `long` occupies array entries three and four, instructions would refer to that `long` by index three. All values in the local variables are word-aligned. Dual-entry longs and doubles can start at any index.

The local variables section contains a method's parameters and local variables. Compilers place the parameters into the local variable array first, in the order in which they are declared. Figure 5-9 shows the local variables section for the following two methods:

```
// On CD-ROM in file jvm/ex3/Example3a.java
class Example3a {

    public static int runClassMethod(int i, long l, float f,
        double d, Object o, byte b) {

        return 0;
    }

    public int runInstanceMethod(char c, double d, short s,
        boolean b) {
```



```

    }
    return 0;
}

```

runClassMethod()			runInstanceMethod()		
index	type	parameter	index	type	parameter
0	int	int i	0	reference	hidden this
1	long	long l	1	int	char c
3	float	float f	2	double	double d
4	double	double d	4	int	short s
6	reference	Object o	5	int	boolean b
7	int	byte b			

Figure 5-9. Method parameters on the local variables section of a Java stack.

Note that Figure 5-9 shows that the first parameter in the local variables for `runInstanceMethod()` is of type `reference`, even though no such parameter appears in the source code. This is the hidden `this` reference passed to every instance method. Instance methods use this reference to access the instance data of the object upon which they were invoked. As you can see by looking at the local variables for `runClassMethod()` in Figure 5-9, class methods do not receive a hidden `this`. Class methods are not invoked on objects. You can't directly access a class's instance variables from a class method, because there is no instance associated with the method invocation.

Note also that types `byte`, `short`, `char`, and `boolean` in the source code become `ints` in the local variables. This is also true of the operand stack. As mentioned earlier, the `boolean` type is not supported directly by the Java virtual machine. The Java compiler always uses `ints` to represent `boolean` values in the local variables or operand stack. Data types `byte`, `short`, and `char`, however, are supported directly by the Java virtual machine. These can be stored on the heap as instance variables or array elements, or in the method area as class variables. When placed into local variables or the operand stack, however, values of type `byte`, `short`, and `char` are converted into `ints`. They are manipulated as `ints` while on the stack frame, then converted back into `byte`, `short`, or `char` when stored back into heap or method area.

Also note that `Object o` is passed as a reference to `runClassMethod()`. In Java, all objects are passed by reference. As all objects are stored on the heap, you will never find an image of an object in the local variables or operand stack, only object references.

Aside from a method's parameters, which compilers must place into the local variables array first and in order of declaration, Java compilers can arrange the local variables array as they wish. Compilers can place the method's local variables into the array in any order, and they can use the same array entry for more than one local variable. For example, if two local variables have limited scopes that don't overlap, such as the `i` and `j` local variables in Example3b, compilers are free to use the same array entry for both variables. During the first half of the method, before `j` comes into scope, entry zero could be used for `i`. During the second half of the method, after `i` has gone out of scope, entry zero could be used for `j`.

```
// On CD-ROM in file jvm/ex3/Example3b.java
class Example3b {

    public static void runtwoLoops() {

        for (int i = 0; i < 10; ++i) {
            System.out.println(i);
        }

        for (int j = 9; j >= 0; --j) {
            System.out.println(j);
        }
    }
}
```

As with all the other runtime memory areas, implementation designers can use whatever data structures they deem most appropriate to represent the local variables. The Java virtual machine specification does not indicate how longs and doubles should be split across the two array entries they occupy. Implementations that use a word size of 64 bits could, for example, store the entire long or double in the lower of the two consecutive entries, leaving the higher entry unused.

Operand Stack

Like the local variables, the operand stack is organized as an array of words. But unlike the local variables, which are accessed via array indices, the operand stack is accessed by pushing and popping values. If an instruction pushes a value onto the operand stack, a later instruction can pop and use that value.

The virtual machine stores the same data types in the operand stack that it stores in the local variables: int, long, float, double, reference, and returnType. It converts values of type byte, short, and char to int before pushing them onto the operand stack.

Other than the program counter, which can't be directly accessed by instructions, the Java virtual machine has no registers. The Java virtual machine is stack-based rather than register-based because its instructions take their operands from the operand stack rather than from registers. Instructions can also take operands from other places, such as immediately following the opcode (the byte representing the instruction) in the bytecode stream, or from the constant pool. The Java virtual machine instruction set's main focus of attention, however, is the operand stack.

The Java virtual machine uses the operand stack as a work space. Many instructions pop values from the operand stack, operate on them, and push the result. For example, the iadd instruction adds two integers by popping two ints off the top of the operand stack, adding them, and pushing the int result. Here is how a Java virtual machine would add two local variables that contain ints and store the int result in a third local variable:

```
iload_0    // push the int in local variable 0
iload_1    // push the int in local variable 1
iadd       // pop two ints, add them, push result
istore_2   // pop int, store into local variable 2
```

In this sequence of bytecodes, the first two instructions, iload_0 and iload_1, push the ints stored in local variable positions zero and one onto the operand stack. The iadd instruction pops those two int values, adds them, and pushes the int result back onto the operand stack. The fourth instruction, istore_2, pops the result of the add off the top of the operand stack and stores it into local variable position two. In Figure 5-10, you can see a graphical depiction of the state of the local variables and operand stack while executing these instructions. In this figure, unused slots of the local variables and operand stack are left blank.

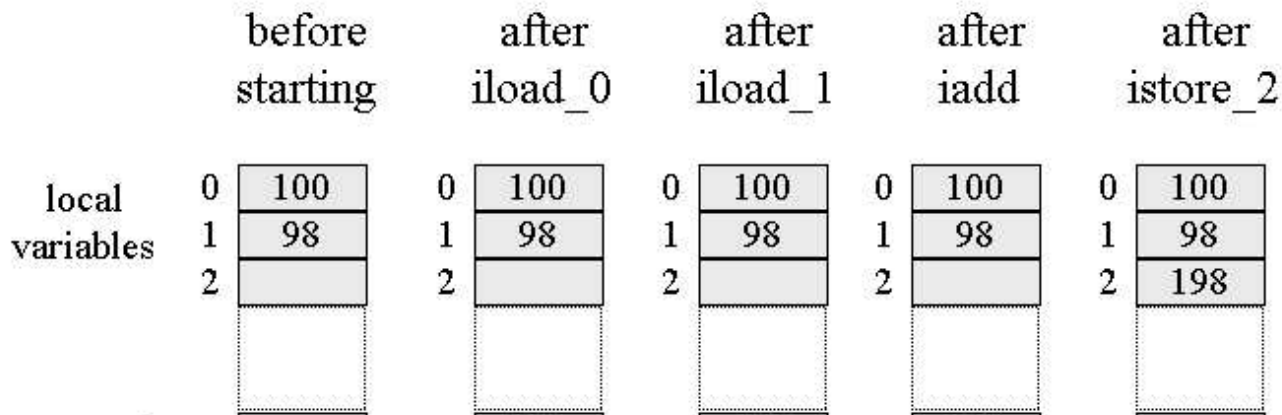




Figure 5-10. Adding two local variables.

Frame Data

In addition to the local variables and operand stack, the Java stack frame includes data to support constant pool resolution, normal method return, and exception dispatch. This data is stored in the *frame data* portion of the Java stack frame.

Many instructions in the Java virtual machine's instruction set refer to entries in the constant pool. Some instructions merely push constant values of type `int`, `long`, `float`, `double`, or `String` from the constant pool onto the operand stack. Some instructions use constant pool entries to refer to classes or arrays to instantiate, fields to access, or methods to invoke. Other instructions determine whether a particular object is a descendant of a particular class or interface specified by a constant pool entry.

Whenever the Java virtual machine encounters any of the instructions that refer to an entry in the constant pool, it uses the frame data's pointer to the constant pool to access that information. As mentioned earlier, references to types, fields, and methods in the constant pool are initially symbolic. When the virtual machine looks up a constant pool entry that refers to a class, interface, field, or method, that reference may still be symbolic. If so, the virtual machine must resolve the reference at that time.

Aside from constant pool resolution, the frame data must assist the virtual machine in processing a normal or abrupt method completion. If a method completes normally (by returning), the virtual machine must restore the stack frame of the invoking method. It must set the pc register to point to the instruction in the invoking method that follows the instruction that invoked the completing method. If the completing method returns a value, the virtual machine must push that value onto the operand stack of the invoking method.

The frame data must also contain some kind of reference to the method's exception table, which the virtual machine uses to process any exceptions thrown during the course of execution of the method. An exception table, which is described in detail in Chapter 17, "Exceptions," defines ranges within the bytecodes of a method that are protected by catch clauses. Each entry in an exception table gives a starting and ending position of the range protected by a catch clause, an index into the constant pool that gives the exception class being caught, and a starting position of the catch clause's code.

When a method throws an exception, the Java virtual machine uses the exception table referred to by the frame data to determine how to handle the exception. If the virtual machine finds a matching catch clause in the method's exception table, it transfers control to the beginning of that catch clause. If the virtual machine doesn't find a matching catch clause, the method completes abruptly. The virtual machine uses the information in the frame data to restore the invoking method's frame. It then rethrows the same exception in the context of the invoking method.

In addition to data to support constant pool resolution, normal method return, and exception dispatch, the stack frame may also include other information that is implementation dependent, such as data to support debugging.

Possible Implementations of the Java Stack

Implementation designers can represent the Java stack in whatever way they wish. As mentioned earlier, one potential way to implement the stack is by allocating each frame separately from a heap. As an example of this approach, consider the following class:

```
// On CD-ROM in file jvm/ex3/Example3c.java
class Example3c {

    public static void addAndPrint() {
        double result = addTwoTypes(1, 88.88);
        System.out.println(result);
    }

    public static double addTwoTypes(int i, double d) {
        return i + d;
    }
}
```

Figure 5-11 shows three snapshots of the Java stack for a thread that invokes the `addAndPrint()` method. In the implementation of the Java virtual machine represented in this figure, each frame is allocated separately from a heap. To invoke the `addTwoTypes()` method, the `addAndPrint()` method first pushes an `int` one and `double` 88.88 onto its operand stack. It then invokes the `addTwoTypes()` method.

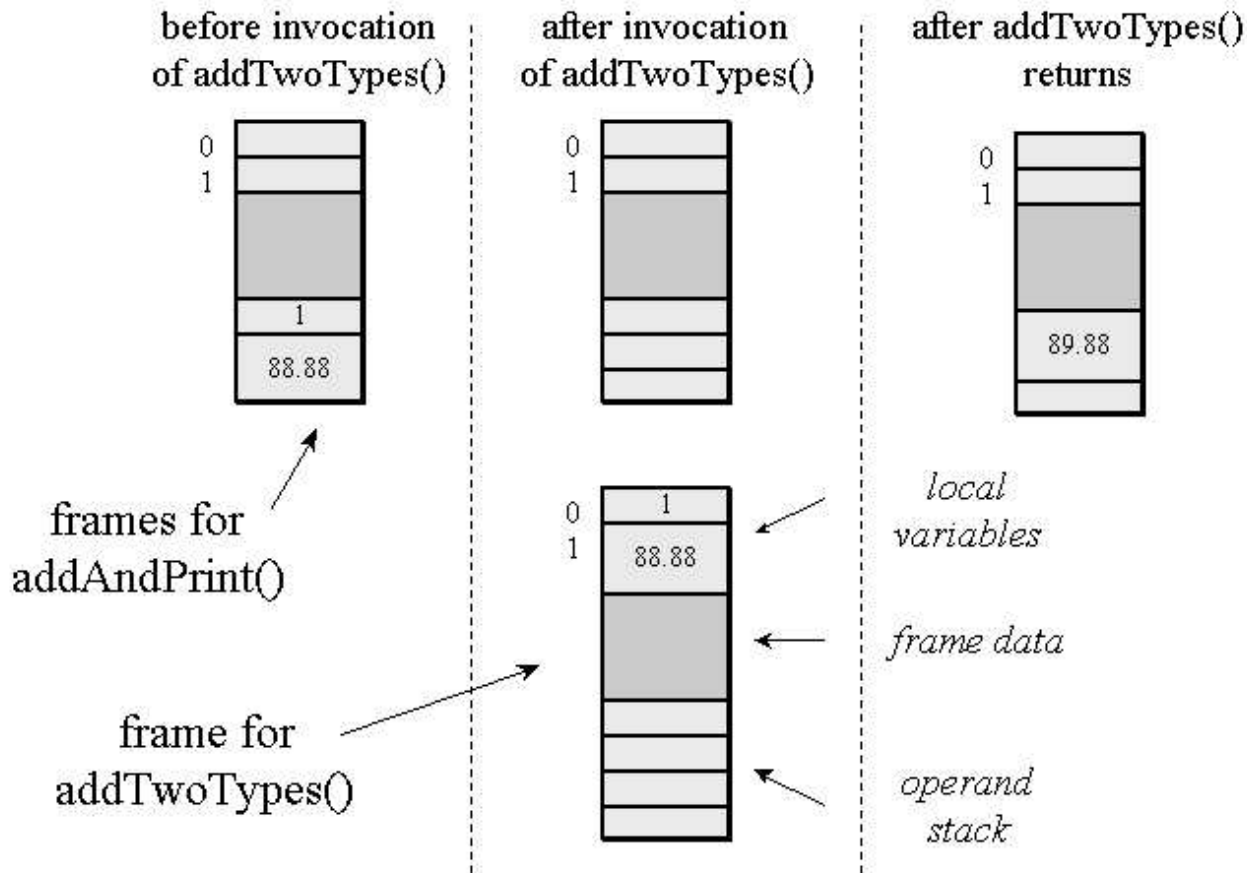


Figure 5-11. Allocating frames from a heap.

The instruction to invoke `addTwoTypes()` refers to a constant pool entry. The Java virtual machine looks up the entry and resolves it if necessary.

Note that the `addAndPrint()` method uses the constant pool to identify the `addTwoTypes()` method, even though it is part of the same class. Like references to fields and methods of other classes, references to the fields and methods of the *same* class are initially symbolic and must be resolved before they are used.

The resolved constant pool entry points to information in the method area about the `addTwoTypes()` method. The virtual machine uses this information to determine the sizes required by `addTwoTypes()` for the local variables and operand stack. In the class file generated by Sun's `javac` compiler from the JDK 1.1, `addTwoTypes()` requires three words in the local variables and four words in the operand stack. (As mentioned earlier, the size of the frame data portion is implementation dependent.) The virtual machine allocates enough memory for the `addTwoTypes()` frame from a heap. It then pops the `double` and `int` parameters (88.88 and one) from `addAndPrint()`'s operand stack and places them into `addTwoTypes()`'s local variable slots one and zero.

When `addTwoTypes()` returns, it first pushes the `double` return value (in this case, 89.88) onto its operand stack. The virtual machine uses the information in the frame data to locate the stack frame of the invoking method, `addAndPrint()`. It pushes the `double` return value onto `addAndPrint()`'s operand stack and frees the memory occupied by `addTwoTypes()`'s frame. It makes `addAndPrint()`'s frame current and continues executing the `addAndPrint()` method at the first instruction past the `addTwoTypes()` method invocation.

Figure 5-12 shows snapshots of the Java stack of a different virtual machine implementation executing the same methods. Instead of allocating each frame separately from a heap, this implementation allocates frames from a contiguous stack. This approach allows the implementation to overlap the frames of adjacent methods. The portion of the invoking method's operand stack that contains the parameters to the invoked method become the base of the invoked method's local variables. In this example, `addAndPrint()`'s entire

operand stack becomes `addTwoTypes()`'s entire local variables section.

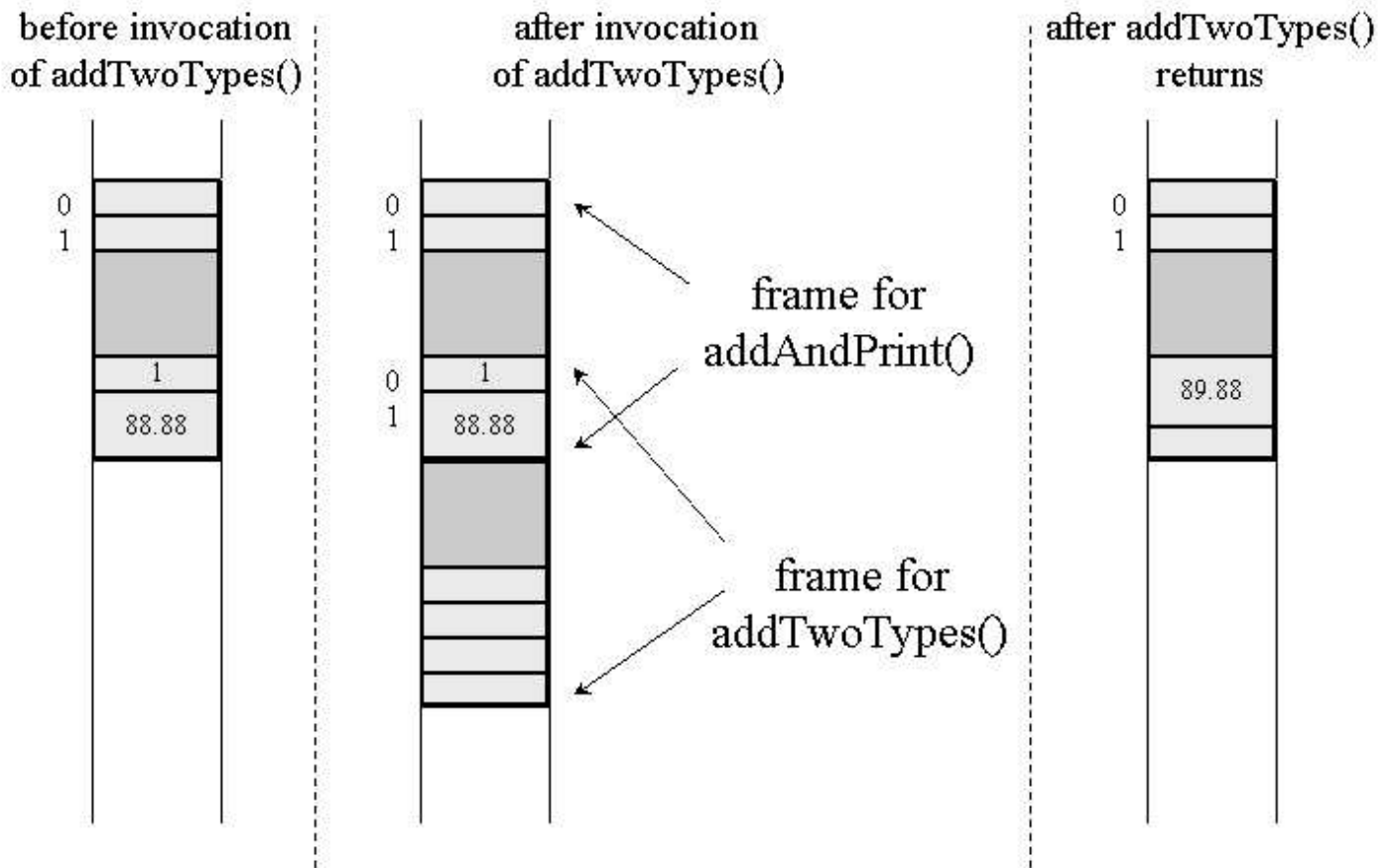


Figure 5-12. Allocating frames from a contiguous stack.

This approach saves memory space because the same memory is used by the calling method to store the parameters as is used by the invoked method to access the parameters. It saves time because the Java virtual machine doesn't have to spend time copying the parameter values from one frame to another.

Note that the operand stack of the current frame is always at the "top" of the Java stack. Although this may be easier to visualize in the contiguous memory implementation of Figure 5-12, it is true no matter how the Java stack is implemented. (As mentioned earlier, in all the graphical images of the stack shown in this book, the stack grows downwards. The "top" of the stack is always shown at the bottom of the picture.) Instructions that push values onto (or pop values off of) the operand stack always operate on the current frame. Thus, pushing a value onto the operand stack can be seen as pushing a value onto the top of the entire Java stack. In the remainder of this book, "pushing a value onto the stack" refers to pushing a value onto the operand stack of the current frame.

One other possible approach to implementing the Java stack is a hybrid of the two approaches shown in Figure 5-11 and Figure 5-12. A Java virtual machine implementation can allocate a chunk of contiguous memory from a heap when a thread starts. In this memory, the virtual machine can use the overlapping frames approach shown in Figure 5-12. If the stack outgrows the contiguous memory, the virtual machine can allocate another chunk of contiguous memory from the heap. It can use the separate frames approach shown in Figure 5-11 to connect the invoking method's frame sitting in the old chunk with the invoked method's frame sitting in the new chunk. Within the new chunk, it can once again use the contiguous memory approach.

Native Method Stacks

In addition to all the runtime data areas defined by the Java virtual machine specification and described previously, a running Java application may use other data areas created by or for native methods. When a thread invokes a native method, it enters a new world in which the structures and security restrictions of the Java virtual machine no longer hamper its freedom. A native method can likely access the runtime data areas of the virtual machine (it depends upon the native method interface), but can also do anything else it wants. It may use registers inside the native processor, allocate memory on any number of native heaps, or use any kind of stack.

Native methods are inherently implementation dependent. Implementation designers are free to decide what mechanisms they will use to enable a Java application running on their implementation to invoke native methods.

Any native method interface will use some kind of native method stack. When a thread invokes a Java method, the virtual machine creates a new frame and pushes it onto the Java stack. When a thread invokes a native method, however, that thread leaves the Java stack behind. Instead of pushing a new frame onto the thread's Java stack, the Java virtual machine will simply dynamically link to and directly invoke the native method. One way to think of it is that the Java virtual machine is dynamically extending itself with native code. It is as if the Java virtual machine implementation is just calling another (dynamically linked) method within itself, at the behest of the running Java program.

If an implementation's native method interface uses a C-linkage model, then the native method stacks are C stacks. When a C program invokes a C function, the stack operates in a certain way. The arguments to the function are pushed onto the stack in a certain order. The return value is passed back to the invoking function in a certain way. This would be the behavior of the of native method stacks in that implementation.

A native method interface will likely (once again, it is up to the designers to decide) be able to call back into the Java virtual machine and invoke a Java method. In this case, the thread leaves the native method stack and enters another Java stack.

Figure 5-13 shows a graphical depiction of a thread that invokes a native method that calls back into the virtual machine to invoke another Java method. This figure shows the full picture of what a thread can expect inside the Java virtual machine. A thread may spend its entire lifetime executing Java methods, working with frames on its Java stack. Or, it may jump back and forth between the Java stack and native method stacks.

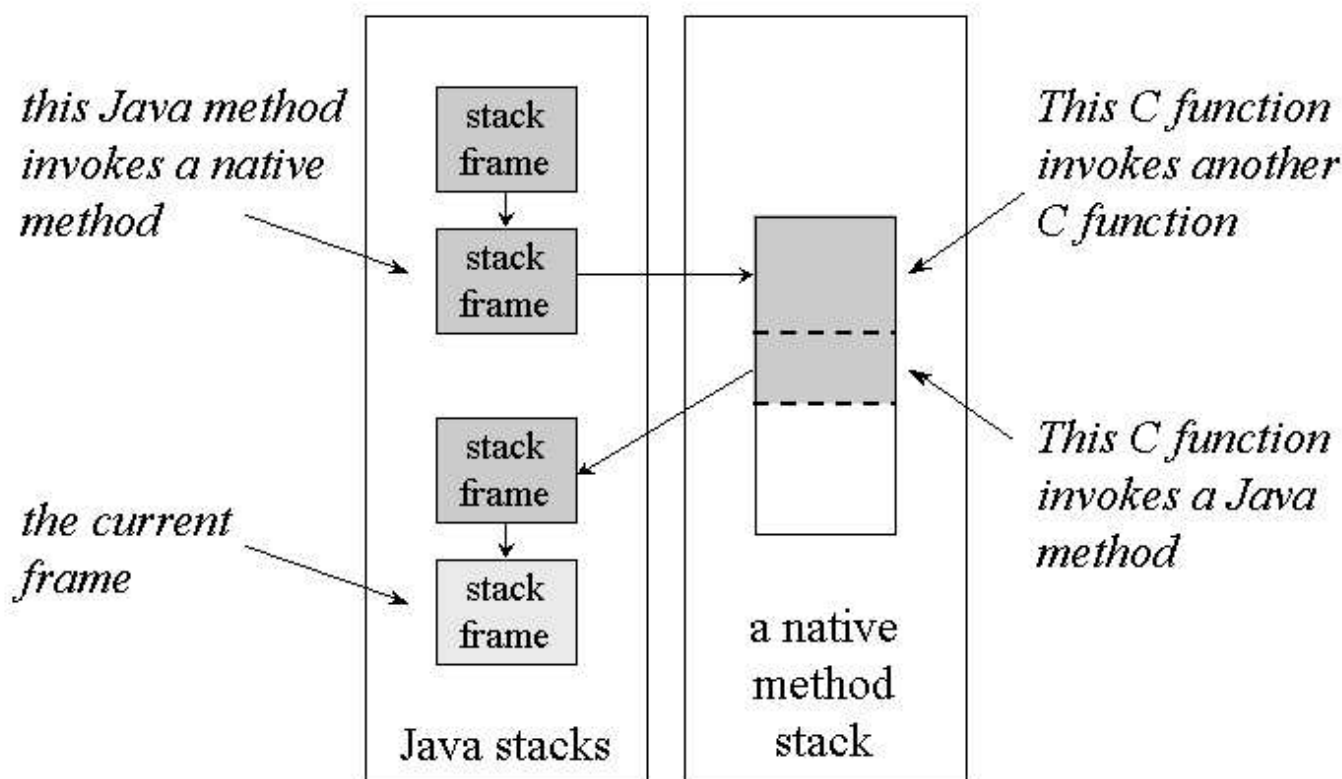


Figure 5-13. The stack for a thread that invokes Java and native methods.

As depicted in Figure 5-13, a thread first invoked two Java methods, the second of which invoked a native method. This act caused the virtual machine to use a native method stack. In this figure, the native method stack is shown as a finite amount of contiguous memory space. Assume it is a C stack. The stack area used by each C-linkage function is shown in gray and bounded by a dashed line. The first C-linkage function, which was invoked as a native method, invoked another C-linkage function. The second C-linkage function invoked a Java method through the native method interface. This Java method invoked another Java method, which is the current method shown in the figure.

As with the other runtime memory areas, the memory they occupied by native method stacks need not be of a fixed size. It can expand and contract as needed by the running application. Implementations may allow users or programmers to specify an initial size for the method area, as well as a maximum or minimum size.

Execution Engine

At the core of any Java virtual machine implementation is its execution engine. In the Java virtual machine specification, the behavior of the execution engine is defined in terms of an instruction set. For each instruction, the specification describes in detail *what* an implementation should do when it encounters the instruction as it executes bytecodes, but says very little about *how*. As mentioned in previous chapters, implementation designers are free to decide how their implementations will execute bytecodes. Their implementations can interpret, just-in-time compile, execute natively in silicon, use a combination of these, or dream up some brand new technique.

Similar to the three senses of the term "Java virtual machine" described at the beginning of this chapter, the term "execution engine" can also be used in any of three senses: an abstract specification, a concrete implementation, or a runtime instance. The abstract specification defines the behavior of an execution engine in terms of the instruction set. Concrete implementations, which may use a variety of techniques, are either software, hardware, or a combination of both. A runtime instance of an execution engine is a thread.

Each thread of a running Java application is a distinct instance of the virtual machine's execution engine. From the beginning of its lifetime to the end, a thread is either executing bytecodes or native methods. A thread may execute bytecodes directly, by interpreting or executing natively in silicon, or indirectly, by just-in-time compiling and executing the resulting native code. A Java virtual machine implementation may use other threads invisible to the running application, such as a thread that performs garbage collection. Such threads need not be "instances" of the implementation's execution engine. All threads that belong to the running application, however, are execution engines in action.

The Instruction Set

A method's bytecode stream is a sequence of instructions for the Java virtual machine. Each instruction consists of a one-byte *opcode* followed by zero or more *operands*. The opcode indicates the operation to be performed. Operands supply extra information needed by the Java virtual machine to perform the operation specified by the opcode. The opcode itself indicates whether or not it is followed by operands, and the form the operands (if any) take. Many Java virtual machine instructions take no operands, and therefore consist only of an opcode. Depending upon the opcode, the virtual machine may refer to data stored in other areas in addition to (or instead of) operands that trail the opcode. When it executes an instruction, the virtual machine may use entries in the current constant pool, entries in the current frame's local variables, or values sitting on the top of the current frame's operand stack.

The abstract execution engine runs by executing bytecodes one instruction at a time. This process takes place for each thread (execution engine instance) of the application running in the Java virtual machine. An execution engine fetches an opcode and, if that opcode has operands, fetches the operands. It executes the action requested by the opcode and its operands, then fetches another opcode. Execution of bytecodes continues until a thread completes either by returning from its starting method or by not catching a thrown exception.

From time to time, the execution engine may encounter an instruction that requests a native method invocation. On such occasions, the execution engine will dutifully attempt to invoke that native method. When the native method returns (if it completes normally, not by throwing an exception), the execution engine will continue executing the next instruction in the bytecode stream.

One way to think of native methods, therefore, is as programmer-customized extensions to the Java virtual machine's instruction set. If an instruction requests an invocation of a native method, the execution engine invokes the native method. Running the native method is how the Java virtual machine executes the instruction. When the native method returns, the virtual machine moves on to the next instruction. If the native method completes abruptly (by throwing an exception), the virtual machine follows the same steps to handle the exception as it does when any instruction throws an exception.

Part of the job of executing an instruction is determining the next instruction to execute. An execution engine determines the next opcode to fetch in one of three ways. For many instructions, the next opcode to execute directly follows the current opcode and its operands, if any, in the bytecode stream. For some instructions, such as `goto` or `return`, the execution engine determines the next opcode as part of its execution of the current instruction. If an instruction throws an exception, the execution engine determines the next opcode to fetch by searching for an appropriate catch clause.

Several instructions can throw exceptions. The `athrow` instruction, for example, throws an exception explicitly. This instruction is the compiled form of the `throw` statement in Java source code. Every time the `athrow` instruction is executed, it will throw an exception. Other instructions throw exceptions only when certain conditions are encountered. For example, if the Java virtual machine discovers, to its chagrin, that the program is attempting to perform an integer divide by zero, it will throw an `ArithmeticException`. This can occur while executing any of four instructions--`idiv`, `ldiv`, `irem`, and `lrem`--which perform divisions or calculate remainders on `ints` or `longs`.

Each type of opcode in the Java virtual machine's instruction set has a mnemonic. In the typical assembly language style, streams of Java bytecodes can be represented by their mnemonics followed by (optional) operand values.

For an example of method's bytecode stream and mnemonics, consider the `doMathForever()` method of this class:

```
// On CD-ROM in file jvm/ex4/Act.java
class Act {

    public static void doMathForever() {
        int i = 0;
        for (;;) {
            i += 1;
            i *= 2;
        }
    }
}
```

The stream of bytecodes for `doMathForever()` can be disassembled into mnemonics as shown next. The Java virtual machine specification does not define any official syntax for representing the mnemonics of a method's bytecodes. The code shown next illustrates the manner in which streams of bytecode mnemonics will be represented in this book. The left hand column shows the offset in bytes from the beginning of the method's bytecodes to the start of each instruction. The center column shows the instruction and any operands. The right hand column contains comments, which are preceded with a double slash, just as in Java source code.

```
// Bytecode stream: 03 3b 84 00 01 1a 05 68 3b a7 ff f9
// Disassembly:
// Method void doMathForever()
// Left column: offset of instruction from beginning of method
// |      Center column: instruction mnemonic and any operands
// |      Right column: comment
0 |  iconst_0          // 03
1 |  istore_0         // 3b
2 |  iinc 0, 1        // 84 00 01
5 |  iload_0          // 1a
6 |  iconst_2         // 05
7 |  imul             // 68
8 |  istore_0         // 3b
9 |  goto 2           // a7 ff f9
```

This way of representing mnemonics is very similar to the output of the `javap` program of Sun's Java 2 SDK. `javap` allows you to look at the bytecode mnemonics of the methods of any class file. Note that jump addresses are given as offsets from the beginning of the method. The `goto` instruction causes the virtual machine to jump to the instruction at offset two (an `iinc`). The actual operand in the stream is minus seven. To execute this instruction, the virtual machine adds the operand to the current contents of the pc register. The result is the address of the `iinc` instruction at offset two. To make the mnemonics easier to read, the operands for jump instructions are shown as if the addition has already taken place. Instead of saying "`goto -7`," the mnemonics say, "`goto 2`."

The central focus of the Java virtual machine's instruction set is the operand stack. Values are generally pushed onto the operand stack before they are used. Although the Java virtual machine has no registers for storing arbitrary values, each method has a set of local variables. The instruction set treats the local variables, in effect, as a set of registers that are referred to by indexes. Nevertheless, other than the `iinc` instruction, which increments a local variable directly, values stored in the local variables must be moved to the operand stack before being used.

For example, to divide one local variable by another, the virtual machine must push both onto the stack, perform the division, and then store the result back into the local variables. To move the value of an array element or object field into a local variable, the virtual machine must first push the value onto the stack, then store it into the local variable. To set an array element or object field to a value stored in a local variable, the virtual machine must follow the reverse procedure. First, it must push the value of the local variable onto the stack, then pop it off the stack and into the array element or object field on the heap.

Several goals--some conflicting--guided the design of the Java virtual machine's instruction set. These goals are basically the same as those described in Part I of this book as the motivation behind Java's entire architecture: platform independence, network mobility, and security.

The platform independence goal was a major influence in the design of the instruction set. The instruction set's stack-centered approach, described previously, was chosen over a register-centered approach to facilitate efficient implementation on architectures with few or irregular registers, such as the Intel 80X86. This feature of the instruction set--the stack-centered design--make it easier to implement the Java virtual machine on a wide variety of host architectures.

Another motivation for Java's stack-centered instruction set is that compilers usually use a stack-based architecture to pass an intermediate compiled form or the compiled program to a linker/optimizer. The Java class file, which is in many ways similar to the UNIX `.o` or Windows `.obj` file emitted by a C compiler, really represents an intermediate compiled form of a Java program. In the case of Java, the virtual machine serves as (dynamic) linker and may serve as optimizer. The stack-centered architecture of the Java virtual machine's instruction set facilitates the optimization that may be performed at run-time in conjunction with execution engines that perform just-in-time compiling or adaptive optimization.

As mentioned in Chapter 4, "Network Mobility," one major design consideration was class file compactness. Compactness is important because it facilitates speedy transmission of class files across networks. In the bytecodes stored in class files, all instructions--except two that deal with table jumping--are aligned on byte boundaries. The total number of opcodes is small enough so that opcodes occupy only one byte. This design strategy favors class file compactness possibly at the cost of some performance when the program runs. In some Java virtual machine implementations, especially those executing bytecodes in silicon, the single-byte opcode may preclude certain optimizations that could improve performance. Also, better performance may have been possible on some implementations if the bytecode streams were word-aligned instead of byte-aligned. (An implementation could always realign bytecode streams, or translate opcodes into a more efficient form as classes are loaded. Bytecodes are byte-aligned in the class file and in the specification of the abstract method area and execution engine. Concrete implementations can store the loaded bytecode streams any way they wish.)

Another goal that guided the design of the instruction set was the ability to do bytecode verification, especially all at once by a data flow analyzer. The verification capability is needed as part of Java's security framework. The ability to use a data flow analyzer on the bytecodes when they are loaded, rather than verifying each instruction as it is executed, facilitates execution speed. One way this design goal manifests itself in the instruction set is that most opcodes indicate the type they operate on.

For example, instead of simply having one instruction that pops a word from the operand stack and stores it in a local variable, the Java virtual machine's instruction set has two. One instruction, `istore`, pops and stores an `int`. The other instruction, `fstore`, pops and stores a `float`. Both of these instructions perform the exact same function when executed: they pop a word and store it. Distinguishing

between popping and storing an `int` versus a `float` is important only to the verification process.

For many instructions, the virtual machine needs to know the types being operated on to know how to perform the operation. For example, the Java virtual machine supports two ways of adding two words together, yielding a one-word result. One addition treats the words as `ints`, the other as `floats`. The difference between these two instructions facilitates verification, but also tells the virtual machine whether it should perform integer or floating point arithmetic.

A few instructions operate on any type. The `dup` instruction, for example, duplicates the top word of a stack irrespective of its type. Some instructions, such as `goto`, don't operate on typed values. The majority of the instructions, however, operate on a specific type. The mnemonics for most of these "typed" instructions indicate their type by a single character prefix that starts their mnemonic. Table 5-2 shows the prefixes for the various types. A few instructions, such as `arraylength` or `instanceof`, don't include a prefix because their type is obvious. The `arraylength` opcode requires an array reference. The `instanceof` opcode requires an object reference.

Type	Code	Example	Description
byte	b	baload	load byte from array
short	s	saload	load short from array
int	i	iaload	load int from array
long	l	laload	load long from array
char	c	caload	load char from array
float	f	faload	load float from array
double	d	daload	load double from array
reference	a	aaload	load reference from array

Table 5-2. Type prefixes of bytecode mnemonics

Values on the operand stack must be used in a manner appropriate to their type. It is illegal, for example, to push four `ints`, then add them as if they were two `longs`. It is illegal to push a `float` value onto the operand stack from the local variables, then store it as an `int` in an array on the heap. It is illegal to push a `double` value from an object field on the heap, then store the topmost of its two words into the local variables as an value of type `reference`. The strict type rules that are enforced by Java compilers must also be enforced by Java virtual machine implementations.

Implementations must also observe rules when executing instructions that perform generic stack operations independent of type. As mentioned previously, the `dup` instruction pushes a copy of the top word of the stack, irrespective of type. This instruction can be used on any value that occupies one word: an `int`, `float`, `reference`, or `returnAddress`. It is illegal, however, to use `dup` when the top of the stack contains either a `long` or `double`, the data types that occupy two consecutive operand stack locations. A `long` or `double` sitting on the top of the operand stack can be duplicated in their entirety by the `dup2` instruction, which pushes a copy of the top two words onto the operand stack. The generic instructions cannot be used to split up dual-word values.

To keep the instruction set small enough to enable each opcode to be represented by a single byte, not all operations are supported on all types. Most operations are not supported for types `byte`, `short`, and `char`. These types are converted to `int` when moved from the heap or method area to the stack frame. They are operated on as `ints`, then converted back to `byte`, `short`, or `char` before being stored back into the heap or method area.

Table 5-3 shows the computation types that correspond to each storage type in the Java virtual machine. As used here, a *storage type* is the manner in which values of the type are represented on the heap. The storage type corresponds to the type of the variable in Java source code. A *computation type* is the manner in which the type is represented on the Java stack frame.

Storage Type	Minimum Bits in Heap or Method Area	Computation Type	Words in the Java Stack Frame
byte	8	int	1
short	16	int	1
int	32	int	1
long	64	long	2
char	16	int	1
float	32	float	1
double	64	double	2
reference	32	reference	1

Table 5-3. Storage and computation types inside the Java virtual machine

Implementations of the Java virtual machine must in some way ensure that values are operated on by instructions appropriate to their type. They can verify bytecodes up front as part of the class verification process, on the fly as the program executes, or some combination of both. Bytecode verification is described in more detail in Chapter 7, "The Lifetime of a Type." The entire instruction set is covered in detail in Chapters 10 through 20.

Execution Techniques

Various execution techniques that may be used by an implementation--interpreting, just-in-time compiling, adaptive optimization, native execution in silicon--were described in Chapter 1, "Introduction to Java's Architecture." The main point to remember about execution techniques is that an implementation can use any technique to execute bytecodes so long as it adheres to the semantics of the Java virtual machine instruction set.

One of the most interesting -- and speedy -- execution techniques is adaptive optimization. The adaptive optimization technique, which is used by several existing Java virtual machine implementations, including Sun's Hotspot virtual machine, borrows from techniques used by earlier virtual machine implementations. The original JVMs interpreted bytecodes one at a time. Second-generation JVMs added a JIT compiler, which compiles each method to native code upon first execution, then executes the native code. Thereafter, whenever the method is called, the native code is executed. Adaptive optimizers, taking advantage of information available only at run-time, attempt to combine bytecode interpretation and compilation to native in the way that will yield optimum performance.

An adaptive optimizing virtual machine begins by interpreting all code, but it monitors the execution of that code. Most programs spend 80 to 90 percent of their time executing 10 to 20 percent of the code. By monitoring the program execution, the virtual machine can figure out which methods represent the program's "hot spot" -- the 10 to 20 percent of the code that is executed 80 to 90 percent of the time.

When the adaptive optimizing virtual machine decides that a particular method is in the hot spot, it fires off a background thread that compiles those bytecodes to native and heavily optimizes the native code. Meanwhile, the program can still execute that method by interpreting its bytecodes. Because the program isn't held up and because the virtual machine is only compiling and optimizing the "hot spot" (perhaps 10 to 20 percent of the code), the virtual machine has more time than a traditional JIT to perform optimizations.

The adaptive optimization approach yields a program in which the code that is executed 80 to 90 percent of the time is native code as heavily optimized as statically compiled C++, with a memory footprint not much bigger than a fully interpreted Java program. In other words, fast. An adaptive optimizing virtual machine can keep the old bytecodes around in case a method moves out of the hot spot. (The hot spot may move somewhat as the program executes.) If a method moves out of the hot spot, the virtual machine can discard the compiled code and revert back to interpreting that method's bytecodes.

As you may have noticed, an adaptive optimizer's approach to making Java programs run fast is similar to the approach programmers should take to improve a program's performance. An adaptive optimizing virtual machine, unlike a regular JIT compiling virtual machine, doesn't do "premature optimization." The adaptive optimizing virtual machine begins by interpreting bytecodes. As the program runs, the virtual machine "profiles" the program to find the program's "hot spot," that 10 to 20 percent of the code that gets executed 80 to 90 percent of the time. And like a good programmer, the adaptive optimizing virtual machine just focuses its optimization efforts on that time-critical code.

But there is a bit more to the adaptive optimization story. Adaptive optimizers can be tuned for the run-time characteristics of Java programs -- in particular, of "well-designed" Java programs. According to David Griswold, Hotspot manager at JavaSoft, "Java is a lot more object-oriented than C++. You can measure that; you can look at the rates of method invocations, dynamic dispatches, and such things. And the rates [for Java] are much higher than they are in C++." Now this high rate of method invocations and dynamic dispatches is especially true in a well-designed Java program, because one aspect of a well-designed Java program is highly factored, fine-grained design -- in other words, lots of compact, cohesive methods and compact, cohesive objects.

This run-time characteristic of Java programs, the high frequency of method invocations and dynamic dispatches, affects performance in two ways. First, there is an overhead associated with each dynamic dispatch. Second, and more significantly, method invocations reduce the effectiveness of compiler optimization.

Method invocations reduce the effectiveness of optimizers because optimizers don't perform well across method invocation boundaries. As a result, optimizers end up focusing on the code between method invocations. And the greater the method invocation frequency, the less code the optimizer has to work with between method invocations, and the less effective the optimization becomes.

The standard solution to this problem is inlining -- the copying of an invoked method's body directly into the body of the invoking method. Inlining eliminates method calls and gives the optimizer more code to work with. It makes possible more effective optimization at the cost of increasing the run-time memory footprint of the program.

The trouble is that inlining is harder with object-oriented languages, such as Java and C++, than with non-object-oriented languages, such as C, because object-oriented languages use dynamic dispatching. And the problem is worse in Java than in C++, because Java has a greater call frequency and a greater percentage of dynamic dispatches than C++.

A regular optimizing static compiler for a C program can inline straightforwardly because there is one function implementation for each function call. The trouble with doing inlining with object-oriented languages is that dynamic method dispatch means there may be multiple function (or method) implementation for any given function call. In other words, the JVM may have many different implementations of a method to choose from at run time, based on the class of the object on which the method is being invoked.

One solution to the problem of inlining a dynamically dispatched method call is to just inline all of the method implementations that may get selected at run-time. The trouble with this solution is that in cases where there are a lot of method implementations, the size of the optimized code can grow very large.

One advantage adaptive optimization has over static compilation is that, because it is happening at runtime, it can use information not available to a static compiler. For example, even though there may be 30 possible implementations that may get called for a particular method invocation, at run-time perhaps only two of them are ever called. The adaptive optimization approach enables only those two to be inlined, thereby minimizing the size of the optimized code.

Threads

The Java virtual machine specification defines a threading model that aims to facilitate implementation on a wide variety of architectures. One goal of the Java threading model is to enable implementation designers, where possible and appropriate, to use native threads. Alternatively, designers can implement a thread mechanism as part of their virtual machine implementation. One advantage to using native threads on a multi-processor host is that different threads of a Java application could run simultaneously on different processors.

One tradeoff of Java's threading model is that the specification of priorities is lowest-common-denominator. A Java thread can run at any one of ten priorities. Priority one is the lowest, and priority ten is the highest. If designers use native threads, they can map the ten Java priorities onto the native priorities however seems most appropriate. The Java virtual machine specification defines the behavior of threads at different priorities only by saying that all threads at the highest priority will get some CPU time. Threads at lower priorities are guaranteed to get CPU time only when all higher priority threads are blocked. Lower priority threads *may* get some CPU time when higher priority threads aren't blocked, but there are no guarantees.

The specification doesn't assume time-slicing between threads of different priorities, because not all architectures time-slice. (As used here, *time-slicing* means that all threads at all priorities will be guaranteed some CPU time, even when no threads are blocked.) Even among those architectures that do time-slice, the algorithms used to allot time slots to threads at various priorities can differ greatly.

As mentioned in Chapter 2, "Platform Independence," you must not rely on time-slicing for program correctness. You should use thread priorities only to give the Java virtual machine hints at what it should spend more time on. To coordinate the activities of multiple threads, you should use synchronization.

The thread implementation of any Java virtual machine must support two aspects of *synchronization*: object locking and thread wait and notify. Object locking helps keep threads from interfering with one another while working independently on shared data. Thread wait and notify helps threads to cooperate with one another while working together toward some common goal. Running applications access the Java virtual machine's locking capabilities via the instruction set, and its wait and notify capabilities via the `wait()`, `notify()`, and `notifyAll()` methods of class `Object`. For more details, see Chapter 20, "Thread Synchronization."

In the Java virtual machine Specification, the behavior of Java threads is defined in terms of *variables*, a *main memory*, and *working memories*. Each Java virtual machine instance has a main memory, which contains all the program's variables: instance variables of objects, components of arrays, and class variables. Each thread has a working memory, in which the thread stores "working copies" of variables it uses or assigns. Local variables and parameters, because they are private to individual threads, can be logically seen as part of either the working memory or main memory.

The Java virtual machine specification defines many rules that govern the low-level interactions of threads with main memory. For example, one rule states that all operations on primitive types, except in some cases `longs` and `doubles`, are atomic. For example, if two threads compete to write two different values to an `int` variable, even in the absence of synchronization, the variable will end up with one value or the other. The variable will not contain a corrupted value. In other words, one thread will win the competition and write its value to the variable first. The losing thread need not sulk, however, because it will write its value the variable second, overwriting the "winning" thread's value.

The exception to this rule is any `long` or `double` variable that is not declared `volatile`. Rather than being treated as a single atomic 64-bit value, such variables may be treated by some implementations as two atomic 32-bit values. Storing a non-volatile `long` to memory, for example, could involve two 32-bit write operations. This non-atomic treatment of `longs` and `doubles` means that two threads competing to write two different values to a `long` or `double` variable can legally yield a corrupted result.

Although implementation designers are not required to treat operations involving non-volatile `longs` and `doubles` atomically, the Java virtual machine specification encourages them to do so anyway. This non-atomic treatment of `longs` and `doubles` is an exception to the general rule that operations on primitive types are atomic. This exception is intended to facilitate efficient implementation of the threading model on processors that don't provide efficient ways to transfer 64-bit values to and from memory. In the future, this exception may be eliminated. For the time being, however, Java programmers must be sure to synchronize access to shared `longs` and `doubles`.

Fundamentally, the rules governing low-level thread behavior specify when a thread may and when it must:

1. copy values of variables from the main memory to its working memory, and
2. write values from its working memory back into the main memory.

For certain conditions, the rules specify a precise and predictable order of memory reads and writes. For other conditions, however, the rules do not specify any order. The rules are designed to enable Java programmers to build multi-threaded programs that exhibit predictable behavior, while giving implementation designers some flexibility. This flexibility enables designers of Java virtual machine implementations to take advantage of standard hardware and software techniques that can improve the performance of multi-threaded applications.

The fundamental high-level implication of all the low-level rules that govern the behavior of threads is this: If access to certain variables isn't synchronized, threads are allowed update those variables in main memory in any order. Without synchronization, your multi-threaded applications may exhibit surprising behavior on some Java virtual machine implementations. With proper use of synchronization, however, you can create multi-threaded Java applications that behave in a predictable way on any implementation of

the Java virtual machine.

Native Method Interface

Java virtual machine implementations aren't required to support any particular native method interface. Some implementations may support no native method interfaces at all. Others may support several, each geared towards a different purpose.

Sun's Java Native Interface, or JNI, is geared towards portability. JNI is designed so it can be supported by any implementation of the Java virtual machine, no matter what garbage collection technique or object representation the implementation uses. This in turn enables developers to link the same (JNI compatible) native method binaries to any JNI-supporting virtual machine implementation on a particular host platform.

Implementation designers can choose to create proprietary native method interfaces in addition to, or instead of, JNI. To achieve its portability, the JNI uses a lot of indirection through pointers to pointers and pointers to functions. To obtain the ultimate in performance, designers of an implementation may decide to offer their own low-level native method interface that is tied closely to the structure of their particular implementation. Designers could also decide to offer a higher-level native method interface than JNI, such as one that brings Java objects into a component software model.

To do useful work, a native method must be able to interact to some degree with the internal state of the Java virtual machine instance. For example, a native method interface may allow native methods to do some or all of the following:

- Pass and return data
- Access instance variables or invoke methods in objects on the garbage-collected heap
- Access class variables or invoke class methods
- Accessing arrays
- Lock an object on the heap for exclusive use by the current thread
- Create new objects on the garbage-collected heap
- Load new classes
- Throw new exceptions
- Catch exceptions thrown by Java methods that the native method invoked
- Catch asynchronous exceptions thrown by the virtual machine
- Indicate to the garbage collector that it no longer needs to use a particular object

Designing a native method interface that offers these services can be complicated. The design needs to ensure that the garbage collector doesn't free any objects that are being used by native methods. If an implementation's garbage collector moves objects to keep heap fragmentation at a minimum, the native method interface design must make sure that either:

1. an object can be moved after its reference has been passed to a native method, or
 2. any objects whose references have been passed to a native method are pinned until the native method returns or otherwise indicates it is done with the objects
- As you can see, native method interfaces are very intertwined with the inner workings of a Java virtual machine.

The Real Machine

As mentioned at the beginning of this chapter, all the subsystems, runtime data areas, and internal behaviors defined by the Java virtual machine specification are abstract. Designers aren't required to organize their implementations around "real" components that map closely to the abstract components of the specification. The abstract internal components and behaviors are merely a vocabulary with which the specification defines the required external behavior of any Java virtual machine implementation.

In other words, an implementation can be anything on the inside, so long as it behaves like a Java virtual machine on the outside. Implementations must be able to recognize Java class files and must adhere to the semantics of the Java code the class files contain. But otherwise, anything goes. How bytecodes are executed, how the runtime data areas are organized, how garbage collection is accomplished, how threads are implemented, how the bootstrap class loader finds classes, what native method interfaces are supported--these are some of the many decisions left to implementation designers.

The flexibility of the specification gives designers the freedom to tailor their implementations to fit their circumstances. In some implementations, minimizing usage of resources may be critical. In other implementations, where resources are plentiful, maximizing performance may be the one and only goal.

By clearly marking the line between the external behavior and the internal implementation of a Java virtual machine, the specification preserves compatibility among all implementations while promoting innovation. Designers are encouraged to apply their talents and creativity towards building ever-better Java virtual machines.

Eternal Math: A Simulation

The CD-ROM contains several simulation applets that serve as interactive illustrations for the material presented in this book. The applet shown in Figure 5-14 simulates a Java virtual machine executing a few bytecodes. You can run this applet by loading `applets/EternalMath.html` from the CD-ROM into any Java enabled web browser or applet viewer that supports JDK 1.0.

The instructions in the simulation represent the body of the `doMathForever()` method of class `Act`, shown previously in the "Instruction Set" section of this chapter. This simulation shows the local variables and operand stack of the current frame, the `pc` register, and the bytecodes in the method area. It also shows an `optop` register, which you can think of as part of the frame data of this particular implementation of the Java virtual machine. The `optop` register always points to one word beyond the top of the operand stack.

The applet has four buttons: Step, Reset, Run, and Stop. Each time you press the Step button, the Java virtual machine simulator will execute the instruction pointed to by the pc register. Initially, the pc register points to an `iconst_0` instruction. The first time you press the Step button, therefore, the virtual machine will execute `iconst_0`. It will push a zero onto the stack and set the pc register to point to the next instruction to execute. Subsequent presses of the Step button will execute subsequent instructions and the pc register will lead the way. If you press the Run button, the simulation will continue with no further coaxing on your part until you press the Stop button. To start the simulation over, press the Reset button.

The value of each register (pc and optop) is shown two ways. The contents of each register, an integer offset from the beginning of either the method's bytecodes or the operand stack, is shown in an edit box. Also, a small arrow (either "pc>" or "optop>") indicates the location contained in the register.

In the simulation the operand stack is shown growing down the panel (up in memory offsets) as words are pushed onto it. The top of the stack recedes back up the panel as words are popped from it.

The `doMathForever()` method has only one local variable, `i`, which sits at array position zero. The first two instructions, `iconst_0` and `istore_0` initialize the local variable to zero. The next instruction, `iinc`, increments `i` by one. This instruction implements the `i += 1` statement from `doMathForever()`. The next instruction, `iload_0`, pushes the value of the local variable onto the operand stack. `iconst_2` pushes an int 2 onto the operand stack. `imul` pops the top two ints from the operand stack, multiplies them, and pushes the result. The `istore_0` instruction pops the result of the multiply and puts it into the local variable. The previous four instructions implement the `i *= 2` statement from `doMathForever()`. The last instruction, `goto`, sends the program counter back to the `iinc` instruction. The `goto` implements the `for (; ;)` loop of `doMathForever()`.

With enough patience and clicks of the Step button (or a long enough run of the Run button), you can get an arithmetic overflow. When the Java virtual machine encounters such a condition, it just truncates, as is shown by this simulation. It does not throw any exceptions.

For each step of the simulation, a panel at the bottom of the applet contains an explanation of what the next instruction will do. Happy clicking.

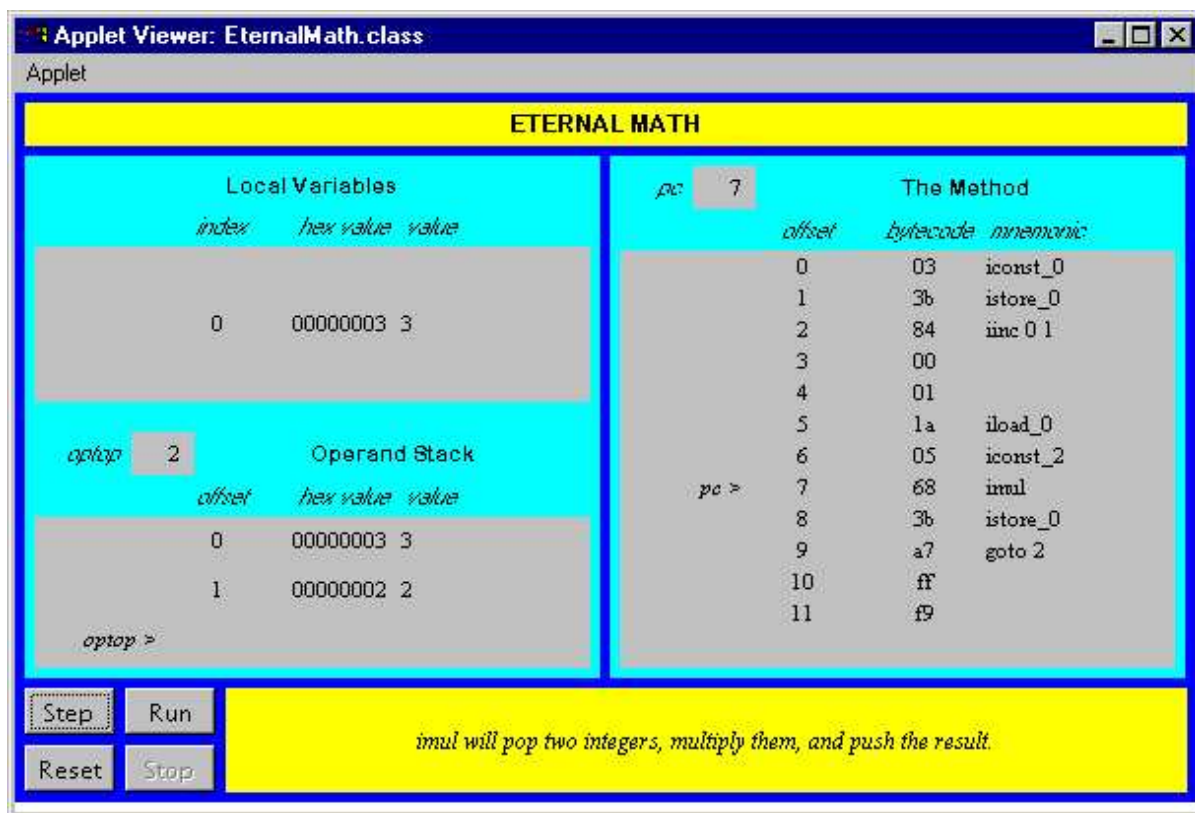


Figure 5-14. The *Eternal Math* applet.

On the CD-ROM

The CD-ROM contains the source code examples from this chapter in the `jvm` directory. The *Eternal Math* applet is contained in a web page on the CD-ROM in file `applets/EternalMath.html`. The source code for this applet is found alongside its class files, in the `applets/JVMSimulators` and `applets/JVMSimulators/COM/artima/jvmsim` directories.

The Resources Page

For links to more information about the Java virtual machine, visit the resources page:
<http://www.artima.com/insidejvm/resources/>

Pavilion Pages

Try [RetroVue](#), the Total Recall journaling debugger from VisiComp.
[Cluster with Coherence](#) for reliability, scalability and performance.
Track issues and change with [TrackStudio Enterprise](#).
Enterprise object relational mapping with [Solarmetric's Kodo JDO](#).
[OrindaBuild](#) writes matching Java to run Oracle PL/SQL procedures.
Visit the [Crystal Developer Zone](#) for Crystal Reports info.
JIRA: [world class issue tracking](#) as used by NASA, BP, 3M & more!

Sponsored Links

Combine [Job Scheduling + JMS Messaging](#) with SwiftMQ.
[Design Objects and Systems](#) with Bruce Eckel and Bill Venners.
Shape the future of Java[tm] technology. [Join the JCP](#) today.

Last Updated: Monday, June 28, 2004
[Copyright](#) © 1996-2004 Artima Software, Inc. All Rights Reserved.

URL: <http://www.artima.com/insidejvm/ed2/jvmP.html>
Trouble with this page? Contact: webmaster@artima.com