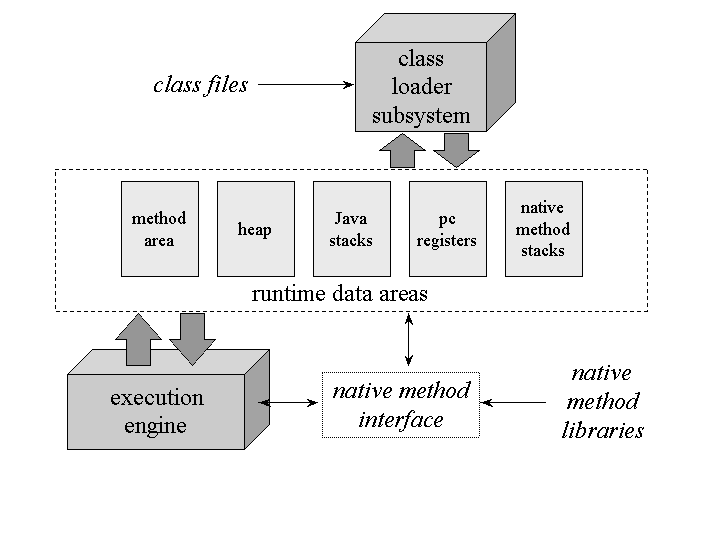
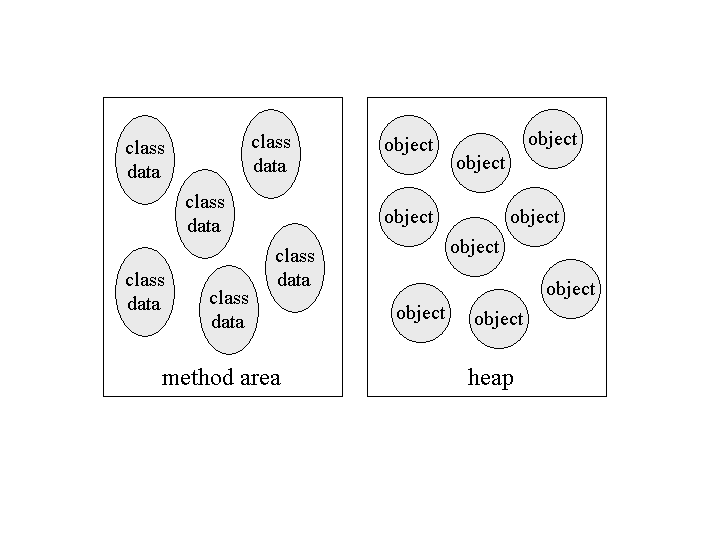
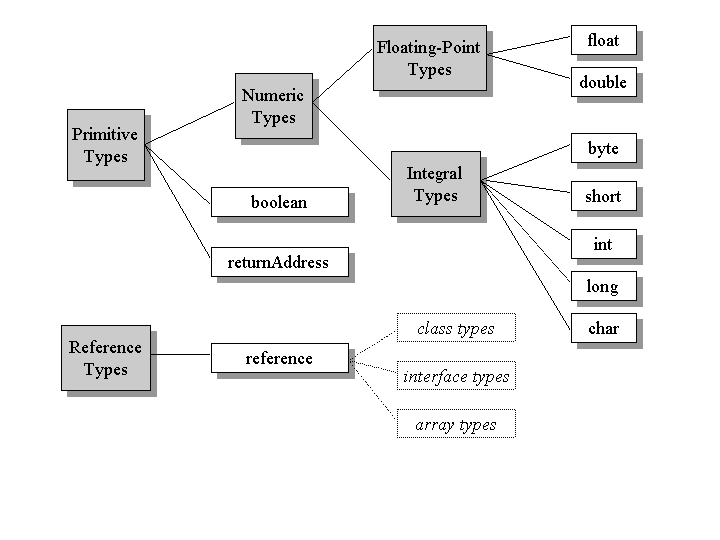
The Java Virtual Machine







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.