CHAPTER 1

INTRODUCTION TO JAVA

1.1. INTRODUCTION TO JAVA

In the early days of programming, languages were structured. With the popularity of C++, there has been a new programming paradigm introduced i.e. object oriented programming. It involves building a program or a system from objects. In the words of Grady Brooch "OOP is a method of programming in which programs are organized as cooperative collection of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationship."

After the success of C^{++} in 1991 a language was needed for the internet which can be used both as console based language and for internet based programming. Various stages of development of java language have been described in history of java.

Java is one object oriented programming language, which is platform independent and very secure for use on internet.

History of Java

Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in June 1991. The team is known as Green project team. Java was not designed for the internet. The objective behind the development of java is to create a common development environment for consumer electronic devices which was easily portable from one device to another. Java was originally designed for interactive television, but it was too advanced for the digital cable television industry at the time. The language was initially called Oak after an oak tree that stood outside Gosling's office; it went by the name Green later, and was later renamed Java, from Java coffee, said to be consumed in large quantities by the language's creators. Gosling aimed to implement a virtual machine and a language that had a familiar C/C++ style of notation

The java development begun by using C++ as their model but the trouble with C and C++ (and most other languages) is that they are designed to be compiled for a specific target. Although it is possible to compile a C++ program for just about any type of CPU, to do so requires a full C++ compiler targeted for that CPU. The problem is that compilers are expensive and time-consuming to create. An easier-and more cost-efficient-solution was needed. In an attempt to find such a solution, Gosling and others began work on a portable, platform-independent language that could be used to produce code that would run on a variety of CPUs under differing environments. This effort ultimately led to the creation of Java. Originally it called "Oak".

In 1993 the www appeared on the Internet and transformed the text based internet into a graphical - rich environment .The Green Project team came up with the idea of developing web page called web applet using new language that could run on all types of computer s connected to Internet. The team developed a web browser called "Hot Java".

Scanned by CamScanner

In 1995 the Oak was renamed as Java. Sun Microsystems released the first public implementation as Java 1.0 in 1995. It promised "Write Once, Run Anywhere" (WORA), providing no-cost run-times on popular platforms. The first beta release of Java was made available for download in November 1995.

Finally On November 13, 2006, Sun released Java as free and open source software, (FOSS). On May 8, 2007, Sun finished the process, making all of Java's core code available under free software/open-source distribution terms, aside from a small portion of code to which Sun did not hold the copyright.

1.2. FEATURES OF JAVA

The inventor of java wanted to design a language which could offer solution to some of the problems encountered in modern internet programming. They wanted the language to be not only reliable, portable and distributed but also simple, compact and interactive. Sun Microsystems officially launched Java with the following attributes:

1. Simple

Java was developed by taking the best points from other programming languages, primarily C and C++. Java therefore utilizes algorithms and methodologies that are already proven.

Error prone tasks such as pointers and memory management have either been eliminated or are handled by the Java environment automatically rather than by the programmer.

Since Java is primarily a derivative of C++ which most programmers are conversant with, it implies that Java has a familiar feel rendering it easy to use. With the above discussion we can also say that java doesn't have any surprising feature therefore it can be termed as simple language.

2. Portable

For internet programming we were in need of that language which is easily downloadable. In simple terms a language which is portable i.e. represents more information in less code, solution to which was provided by the java language.

3. Secure

Java deals with the vast security issues by putting in place multilevel system of security.

Maximum viruses and infected files are formed with the help of pointers, whereas in java, pointers are not allowed and memory management is also under the control of compiler of java. Java provides a firewall between a java program and the remaining computer system. Neither this firewall allows the java program to interfere in the remaining computer system nor does this allow the remaining computer system to interfere in the Java runtime environment.

Important note: Portability and Security are two co related words and we will throw light on the matter of the fact that both characteristics have same solution achieved from same mechanism i.e. BYTECODE.

Bytecode is a highly optimized set of instructions designed to be executed by the Java run time system called as *JVM(Java Virtual Machine)* produced by the compilation of source code.

After the translation of source code into bytecode, execution of Java programs in varied environment becomes easy, as now only JVM needs to be implemented for each platform. At compile time pointers and memory allocation are removed, even though the java compiler produces only correct java code there is still the possibility of the code being tempered with between compilation and runtime. Java guards against this by using bytecode.

Bytecode helps in achieving following things that the code does not do any of the following.

(*a*) Forge pointers (*b*) Violate access restrictions (*c*) Incorrectly access classes (*d*)Overflow or underflow operand stack (*e*) Use incorrect parameters of bytecode instructions (*f*)Use illegal data conversions

At run time the java interpreter further ensures that Java program loaded do not access the remaining file system of the computer except that permitted.

4. Object Oriented

In OOP data is treated as objects to which methods are applied. Java's basic execution unit is the class. The object model in java is simple and easy to extend as it also provides reusability of code and dynamic applications. Although Java is not purely OOPs based language still it is strictly a OOPs based language. As simple data types like integers are treated as non-objects.

5. Robust(strong)

Java is a robust language. It provides many safeguards to ensure reliable code. The ability to create robust programs was given a high priority in the design of Java. Java is a strictly typed language; it checks the code at compile time. However, it also checks your code at run time. In fact, many hard-to-track-down bugs that often turn up in hard-to-reproduce run-time situations are simply impossible to create in Java.

There are two reasons for program failure

1. Memory Management mistakes
2. Mishandled Exceptional conditions

Java manages memory automatically by using garbage collector. Exception conditions are handled in java by providing object exception handling. In the well written java program all runtime errors can and should be managed by the program.

6. Multithreaded

Multithreading is special form multiprocessing which helps in executing a java program with multiple threads in parallel. Java is basically designed to meet the real world programming requirement of creating various networked programs which can interact with each other. To accomplish this java support multithreaded programming which allows user to write programs that does multiple things simultaneously.

7. Architecture neutral

Java compiler compiles source code to a stage which is intermediate between source and native machine code. This intermediate stage is known as the bytecode which is neutral. In easy word it can be said that "write once run anywhere, anytime, forever". Neutral word here means that change in architecture, design, or platform doesn't have affect on the execution of program.

Important note: Java is also platform independent and machine independent

8. Interpreted

When java code is compiled by the compiler output of that is java bytecode which is an executable code for the Java virtual machine (JVM). The bytecode is then run through a Java interpreter on any given platform that has the interpreter parted to it. The interpreter converts the code to the target hardware and executes it.

9. High performance

The Java language supports many high-performance feature such as multithreading, just-in-time compiling, and native code usage. In Java creation of cross-platform programs is made possible with the help of bytecode.

10. Distributed

Java is designed for the internet as it can handle TCP\IP protocol. Internet programmers can call on the functions through the supplied libraries and be able to access files on the internet as easily, as writing to a local file system. Java also allows objects on two different computers to execute procedures remotely. Java has recently revised these interfaces in a package called Remote Method Invocation (RMI). Internet is purely a distributed system and Java is designed for this so java is also Distributed in nature.

Scanned by CamScanner

1. Dynamic

By connecting to the internet a user immediately has access to thousands of programs and other computers. During the execution of a program java can dynamically load classes that it requires either from the local hard drive from another computer on the local area network or from a computer somewhere on the internet.

1.3. JAVA PLATFORM OVERVIEW

1.3.1. The Java Language

Java is a general-purpose, concurrent, class-based, object-oriented computer programming language that is specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere" (WORA), meaning that code that runs on one platform does not need to be recompiled to run on another. Java applications are typically compiled to bytecode (class file) that can run on any Java virtual machine (JVM) regardless of computer architecture.

1.3.2. The Java Complier

In Java, a source file is officially called a compilation unit. It is a text file that contains one or more class definitions. The Java compiler requires that a source file use the .Java filename extension. Notice that the file extension is four characters long. As you might guess, your operating system must be capable of supporting long filenames. This means that DOS and Windows 3.1 are not capable of supporting Java. However, Windows 95/98 and Windows NT/2000/XP work just fine.

In Java, all code must reside inside a class. By convention, the name of that class should match the name of the file that holds the program. You should also make sure that the capitalization of the filename matches the class name. The reason for this is that Java is case-sensitive.

1.3.3. The JVM

Java is an interpreted language: the Java compiler generates byte-codes for the Java Virtual Machine (JVM), rather than native machine code. To actually run a Java program, you use the Java interpreter to execute the compiled byte-codes because Java byte-codes are platform-independent.

1.3.4. The Java Environment

Java environment includes a large number of development tools and hundreds of classes and methods. The development tools are of the system known as :

1.3.4.1. Java Development Kit

The Java Development Kit comes with a collection of tools that are used for developing and running ava programs. They include:

Tool	*Description*
appletviewer	Enables to run Java applets(without actually using a Java-compatible browser).
ava	Java interpreter, which runs applets and application by reading and interpreting bytecode files.
avac	Java Compiler, which translates Java source code to Bytecode files that the interpreter can understand
vadoc	Creates HTML format documentation from Java source Code files.
ah	Produces header files for use with native methods.
ap	Java disassembler, which enables to covert bytecode Files into a program description.
	Java debugger, which helps us to find errors in our programs

To create Java Program, we need to create a source file using a text editor. The source code is then compiled using the Java compiler Javac and executed using the Java interpreter Java The Java debugger jdb is used to find errors,if any in the source code.A compiled Java Program can be converted into a source code with the help of Java disassembler Javap. Following figure shows the complete flow of Java program.

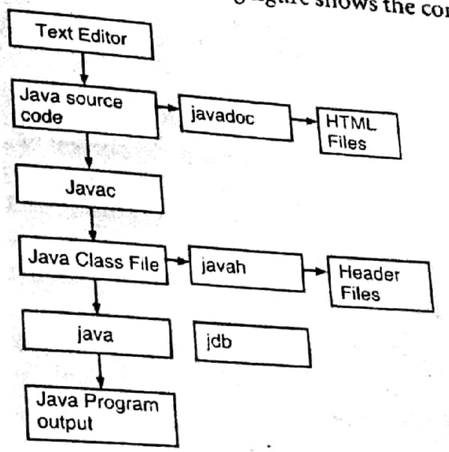

Fig. 1.1. Flow of Java Program

1.3.4.2 Packages in Java Program

The Java Standard Library(or **API – Application Programming Interface**) includes hundreds of classes and methods grouped into several functions. Most commonly used packages are:

- **Language Support Package(Java.lang):** A collection of classes and methods required for implementing basic features of Java. It is a default package which automatically imported to the Java program
- **Utilities Package (Java.util):** A collection of classes to provide utility functions such as date and time functions.
- **Input/Output Package (Java.io):** A collecton of classes required for input/output manipulation.
- **Networking Package (Java.net):** A collection of classes for communicating with other computers via Internet.
- **AWT Package (Java.awt):** The abstract Window ToolKit Package contains classes that implements platform-independents graphical user interface.
- **Applet Package (Java.applet) :** This is includes a set of classes that allows us to create Java applets.

1.3.4.3. The CLASSPATH environment variable and JAR files

After installing the software, the JDK directory will have the structure shown below.

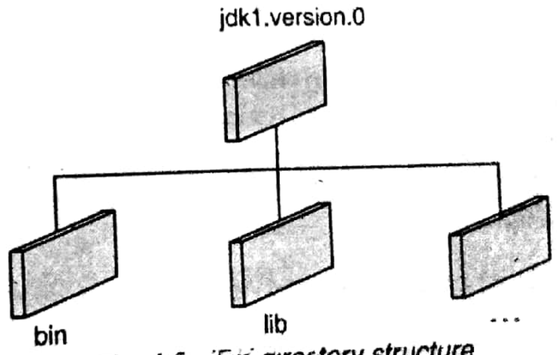

Fig. 1.2. JDK directory structure

Scanned by CamScanner

The bin directory contains both the compiler and the launcher.

When the Java VM runs your program it searches for application .class files using the paths listed in your CLASSPATH environment variable. There are three possibilities -

- To run a simple Java application, you do not need to set a CLASSPATH environment variable; the Java VM looks for classes in the current working directory by default. So either your class file should be in the same directory (bin) or you need to specify the full path to the executable every time you run it, such as:

```
C:> Javac C:\Java\jdk1.7.0\bin\Javac MyClass.Java
```

- But, if your program uses classes that are not in the current working directory, then you need to explicitly set your CLASSPATH environment variable to list all the directories containing classes used by this application which can be done either by using **SET PATH** command on command prompt or by using **CONTORL PANEL** as -

(a) SET PATH Command -

```
C:> set path=c:\jdk1.7.0\bin
C:> Javac MyClass.Java
```

(b) Using Control Panel -

Windows XP

1. Select **Start**, select **Control Panel**, **select Performance and maintenance option and** double click **System,** and select the **Advanced** tab.
2. Click **Environment Variables.** In the section System Variables, find the PATH environment variable and select it. Click Edit. If the PATH environment variable does not exist, click New.
3. In the **Edit System Variable** (or **New System Variable)** window, specify the value of the PATH environment variable. Click **OK.** Close all remaining windows by clicking **OK.**

Windows Vista:

1. From the desktop, right click the My Computer icon.
2. Choose Properties from the context menu.
3. Click the Advanced tab (Advanced system settings link in Vista).
4. Click **Environment Variables.** In the section **System Variables,** find the PATH environment variable and select it. Click Edit. If the PATH environment variable does not exist, click New.
5. In the Edit System Variable (or New System Variable) window, specify the value of the PATH environment variable. Click **OK.** Close all remaining windows by clicking OK.

Windows 7:

1. From the desktop, right click the Computer icon.
2. Choose **Properties** from the context menu.
3. Click the **Advanced system settings** link.
4. Click **Environment Variables.** In the section System Variables, find the PATH environment variable and select it. Click Edit. If the PATH environment variable does not exist, click New.
5. In the **Edit System Variable** (or **New System Variable**) window, specify the value of the PATH environment variable. Click **OK.** Close all remaining windows by clicking OK.

- In addition, if your program uses classes contained in a JAR (Java ARchive) file, then the JAR file must be listed in your CLASSPATH. A JAR file is a single file that contains many .class files.

setenv CLASSPATH

```
./:/home/newhall/public_html/cs35/book_classes/lib/jdsl_general.jar
```


Each path is separated by a ':'. In this example, CLASSPATH is set to two directories: the current directory (./) and the dsl_general.jar file. If you have a CLASSPATH environment variable defined, then the VM uses the paths listed in the CLASSPATH environment variable rather than searching using its default paths. As a result, if you set this environment variable rather than searching using its the VM will not find classes in the current working directory.

1.3.4.4. The Garbage Collector

The Java runtime environment has a garbage collector that periodically frees the memory used by objects that are no longer needed. The Java garbage collector can be a mark-sweep garbage collector that scans Java's dynamic memory areas for objects, marking those that are referenced. After all the objects are investigated, the objects that are not marked (not referenced) are assumed to be garbage and their memory is reclaimed. The garbage collector runs either synchronously or asynchronously in a low priority daemon thread. The garbage collector executes synchronously when the system runs out of memory or asynchronously when the system is idle. The garbage collector can be invoked to run at any time by calling System.gc(). But asking the garbage collector to run does not guarantee that your objects will be garbage collected.

2. OOPS Concepts: Object Oriented Programming or OOP is the technique to create programs based on the real world. Unlike procedural programming, here in the OOP programming model programs are organized around objects and data rather than actions and logic. Objects represent some concepts or things and like any other objects in the real Objects in programming language have certain behavior, properties, type, and identity.

3. OOPS Features- There are some basic features of object oriented programming as follows:

1. **Object:** Objects are user defined variables/runtime entities in object oriented method. Objects are the basic unit of object orientation with behavior, identity. An object is expressed by the variable and methods within the objects.

For example:

Object: STUDENT
DATA Name Address Marks
METHODS Total() Average()

Each object holds data and code to operate the data. Object can interact without having to identify the details of each other's data or code. It is sufficient to identify the type of message received and the type of reply returned by the objects.

2. Classes

It is the central point of OOP and that contains data and codes with behavior. In Java everything happens within class and it describes a set of objects with common behavior. The class definition describes all the properties, behavior, and identity of objects present within that class. A class is a set of objects with similar properties (attributes), common behavior (operations), and common link to other objects.

The complete set of data and code of an object can be made a user defined data type with the help of class. The objects are variable of type class. Classes are user defined data types and work like the build in type of the programming language. Once the class has been defined, we can make any number of object belonging to that class. Each object is related with the data of type class with which they are formed.

Scanned by CamScanner

3. Data Abstraction

The process of abstraction in Java is used to hide certain details and only show the essential features of the object or we can say without including the background details or explanations. For ex a car sales man will not tell the customer how many wires are there inside the car, will only tell the important features in which customers are interested.

4. Data Encapsulation

Data Encapsulation means wrapping of data and functions into a single unit (i.e. class). It is most useful feature of class. The data is not easy to get to the outside world and only those functions which are enclosed in the class can access it.

These functions provide the boundary between Object's data and program. This insulation of data from direct access by the program is called as Data hiding.

For example:

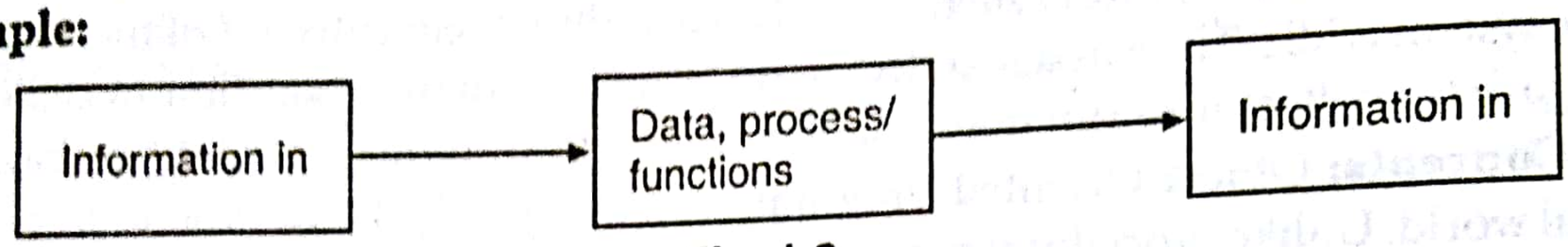

Fig. 1.3.

5. Inheritance

Inheritance is the process by which objects of one class can get the properties of objects of another class. Inheritance means one class of objects inherits the data and behaviors from another class. Inheritance maintains the hierarchical classification in which a class inherits from its parents.

Inheritance provides reusability. That means we can include additional characteristics to an existing class without modification. This is possible deriving a new class from existing one.

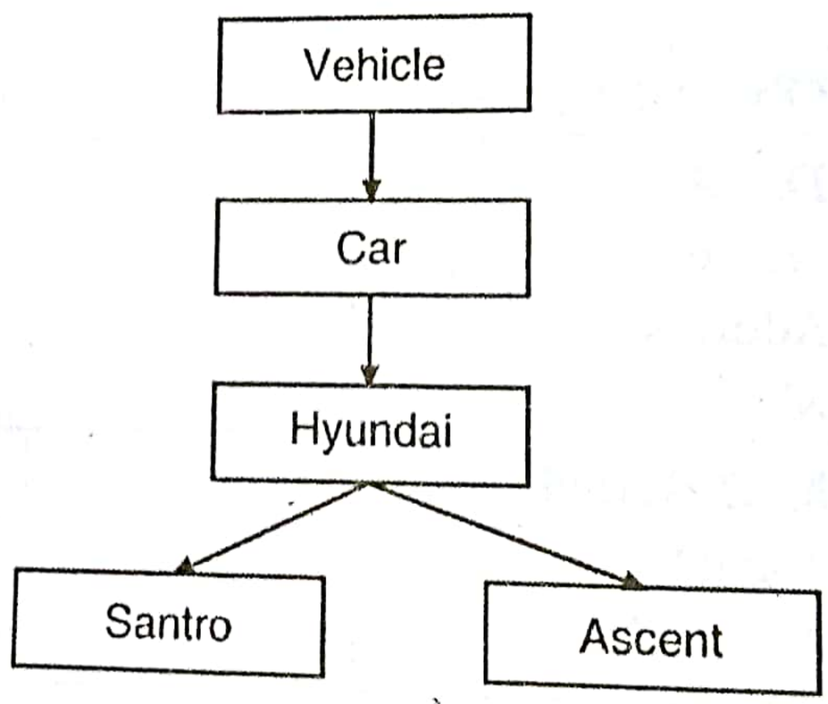

Fig. 1.4.

In the above Figure, the Santro is a part of the class Hyundai which is again part of the class car and car is the part of the class vehicle. That means vehicle class is the parent class.

6. Polymorphism

(Poly means—many and morph means —form). Polymorphism means the ability to take more than one form. Polymorphism plays a main role in allocate objects having different internal structures to share the same external interface.

It means objects that can take on or assume many different forms. Polymorphism means that the same operations may behave differently on different classes. Polymorphism allows us to write generic, reusable code more easily. As shown in following fig 1.5.

Scanned by CamScanner

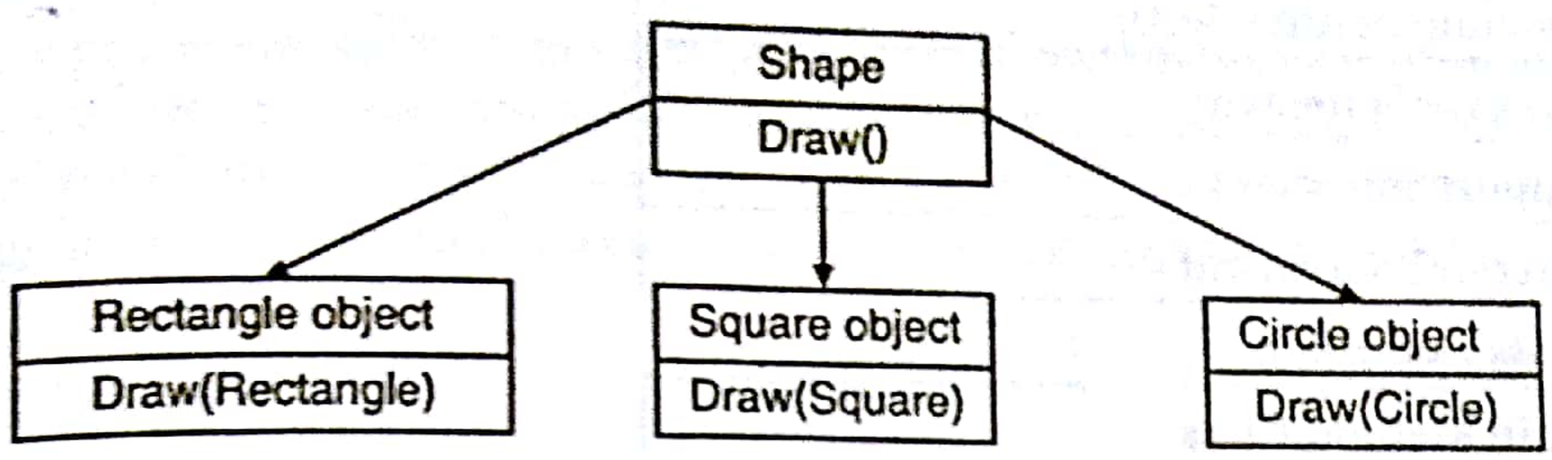

Fig. 1.5.

7. Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means that the code related with a given procedure call is not known until the time of the call at run time.

4. Java Program Structure

Java is a general-purpose, object-oriented programming language which supports two different types of Programs.

- Stand-alone application (Java Application)
- Web applications (Java Applets)

Stand- alone application (Java Application)

These applications are the programs written in Java to carry out certain tasks on a stand-alone local machine. Execution of a stand-alone Java program involves two steps.

1. **Compiling source code into bytecode using Javac compiler**
2. **Executing the bytecode program using Java interpreter**

Following figure shows the flow of Java program for both stand alone program and applet.

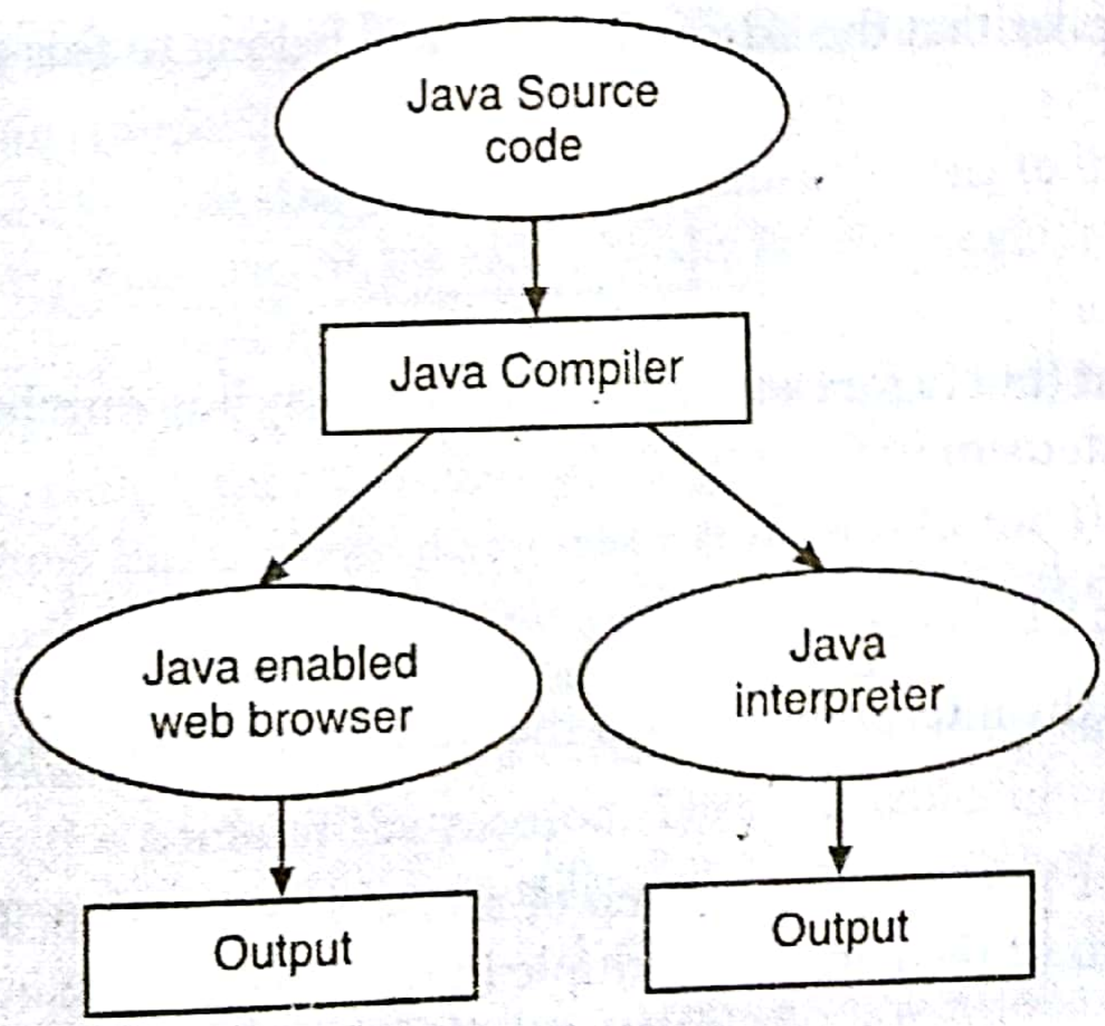

Fig. 1.6. Flow of stand-alone & applet program

Java Application Structure

A Java program may contain many classes of which only one class defines a main method. Classes contain data members and methods that operate on the data members of the class Method may contain data type declaration and executable statement. To write a Java program we first define classes and then put them together. A Java program may contain one or more section

Scanned by CamScanner

Documentation Section
Package Statement
Import Statement
Interface Statement
Class Definition

Main Method Class
{
main Method Definition
}

Documentation Section

The documentation section comprises a set of comment lines giving the name of the program, author and other details, which the programmer would like to refer to at a later stage. Comments may explain why and what of classes and hoe of algorithms.

Single Line Comments - As used in C++

//

Multiple Line Comments - As in C++

```
/* ------------
----------
-----*/
```

Documentation Comments – Java also uses a third style of comment /**.....*/ known as documentation comment. This form of comment is used for generating documentation automatically.

Package Statement

The first statement allowed in a Java file is a Package statement. This statement declares a Package name and informs the compiler that the classes defined here belong to this package. The package statement is optional. Example

```
package p1;
```

Import Statement

After a package statement (but before any class definition)may be a number of import statement. This is similar to the **#include** statement in C

Example

```
import Java.util.Date;
```

This statement instructs the interpreter to load the Date class contained in the package util.

Interface Statement

An interface is like a class but includes a group of method declaration. This is also optional section and is used only when wish to implement the multiple inheritance feature in the program. Interface is new concept in Java.

Class Definitions

A Java program may contain multiple class definition. Classes are the primary and essential elements of a Java Program. These classes are used to map the objects of real world problems. The number of classes used depends on the complexity of the problems

Main Method Class

Since every Java stand alone program required a main method as its starting point this class is the essential part of a Java program. A simple Java program may contain only this part. The main method creates objects of various classes and establishes communication between them on reaching the end of main the program terminates and the control passes back to the operating system.

Ex,

```
class Program
{
    public static void main(String args[])
    {
        System.out.println("Hello");
    }
}
```

Here, the class program contains the main () method, and print the message. You have to type this code in notepad and the name of Java source file and the main method class name should be same.

Save source file : Program.Java
Complile source file : Javac Program.Java - and convert in .class file and create a file as Program.class
Interpret bytecode file : Java Program

main() method signature

```
public static void main(String args[])
{
}
```

The **public** keyword is an access *specifier*, which allows the programmer to control the visibility of class members. When a class member is preceded by **public,** then that member may be accessed by code outside the class in which it is declared.

The keyword **static** allows **main( )** to be called without having to instantiate a particular instance of the class. This is necessary since **main( )** is called by the Java interpreter before any objects are made.

The keyword void simply tells the compiler that main( ) does not return a value.

main( ) is the method called when a Java application begins. The Java is case-sensitive. Thus, Main is different from **main**. It is important to understand that the Java compiler will compile classes that do not contain a **main( )** method. But the Java interpreter has no way to run these classes. So, type Main instead of main, the compiler would still compile your program. However, the Java interpreter would report an error because it would be unable to find the **main( )** method.

Any information that needs to pass to a method is received by variables specified within the set of parentheses that follow the name of the method. These variables are called parameters. If there are no parameters required for a given method, you still need to include the empty parentheses. In **main( )**, there is only one parameter, although a complicated one. **String args[ ]** declares a parameter named **args**, which is an array of instances of the class **String.** (*Arrays* are collections of similar objects.) Objects of type String store character strings. In this case, args receives any command-line arguments present when the program is executed.

```
System.out.println("Hello java.");
```

This line outputs the string "Hello java." followed by a new line on the screen. Output is actually accomplished by the built-in println( ) method. In this case, println( ) displays the string which is passed

Scanned by CamScanner

to it. **println( )** can be used to display other types of information, too. System is a predefined class that provides access to the system, and out is the output stream that is connected to the console.

Web application (Java Applets)

Applets are small/tiny downloadable Java programs developed for Internet application. An applet located on a distant computer (Server) can be downloaded via Internet and executed on a local computer (Client) using a Java -capable browser. Since applets are embedded in an HTML (Hypertext Markup Language) document and run inside a Web page, creating and running applets are more complex than creating an application.

1.4. JAVA PROGRAM COMPILATION AND EXECUTION PROCESS

Compilation and execution of a Java program is two step processes. During compilation phase Java compiler compiles the source code and generates bytecode. This intermediate *bytecode* is saved in form of a .class file.

In second phase, Java virtual machine (JVM) also called Java interpreter takes the .class as input and generates output by executing the *bytecode*.

Java is an object oriented programming language; therefore, a program in Java is made of one or more classes. No matter how trivial a Java program is, it must be written in form of a class.

1.4.1. Develop Java Program

To demonstrate compilation and execution of a Java program we create a simple HelloWorld program. While writing HelloWorld program we must keep in mind that the file name and the name of the class that contains main method must be identical. Second, a file can contain only one public class at a time; therefore, if a file contains more than one class, the only class can be declared public at a time. By keeping above rules in mind we create a Java program HelloWorld.java inserting the following piece of code into a plain text file.

```
/* HelloWorld.java */
class HelloWorld
{
        public static void main(String[] args)
        {
                // prints Hello World! on console
                System.out.println("Hello World!");
        }
}
```

1.4.2. Compile Java Program

Once the Java program is written and saved, first, it has to be compiled. To compile a Java program from command line we need to invoke the Java compiler by supplying javac command. Java compiler comes with JDK (Java Development Kit). JDK is a bundle of software needed for developing Java applications. It includes the JRE (Java Runtime Environment), set of API classes,

Now, compile HelloWorld.java as follows:

```
javac HelloWorld.java
```

The `javac` compiler creates a file called `HelloWorld.class` that contains the bytecode version of the program. The Java bytecode is the intermediate representation of HelloWorld.

Scanned by CamScanner

of information, too. System is a predefined class that
stream that is connected to the console.

ams developed for Internet application. An applet
aded via Internet and executed on a local computer
e embedd g an HTML (Hypertext Markup Lan-
d run pplets are more complex than creat-

OCESS

g compilation phase Java com-
e bytecode is saved in form of a

r takes the .class as input and

n in Java is made of one or more
rm of a class.

e create a simple HelloWorld program.
e file name and the name of the class that
tain only one public class at a time; there-
eclared public at a time. By keeping above
g the following piece of code into a plain

```
args)
e
ld!");
```


contains instructions the Java interpreter will execute. The Java compiler doesn't execute the Java program - that is the job of the Java virtual machine. However, the Java virtual machine cannot execute .java files directly. The compiler's job is to translate Java source files into "class files" that the virtual machine can execute.

Note that the Java compiler (javac) also facilitates to compile multiple .java files together. If there are more than one Java source files in the same directory, you can either list the file names separated by spaces, or use the wildcard characters, for example,

```
javac HelloWorld.java one.java two.java
or
javac *.java
```

1.4.3. Run Java Program

After successful compilation of `HelloWorld.java` to `HelloWorld.class` to actually run the program, we use the Java interpreter, called java. To do so, pass the class name HelloWorld as a command-line argument, as shown follows:

```
java HelloWorld
```

The message `Hello World!` will be printed on the screen as a result of the above command.

It is important to note that in above command we have omitted the .class suffix of the byte-code file name (that is HelloWorld.class in our case).

The java command invokes the *Java Virtual Machine (JVM)*. JVM then loads the specified class mentioned at command line and invokes the method main of this class and start executing it, passing it a single argument that is an array of strings. The array of strings passed to main is to receive command line arguments. The JVM generally takes the following steps in order to run a Java program.

Loading of Classes and Interfaces

Loading refers to the process of finding the binary form of a class or interface type with a particular name computed from source code by a Java compiler. This is simply the bytecode.

Linking of Classes and Interfaces

Linking is the process of taking a binary form of a class or interface type and combining it into the run-time state of the Java virtual machine, so that it can be executed. A class or interface type is always loaded before it is linked.

QUESTIONS WITH ANSWERS

Scanned by CamScanner

CHAPTER

2 JAVA VIRTUAL MACHINE

2.1. INTRODUCTION TO JAVA VIRTUAL MACHINE

All language compilers translate source code into machine code for a specific computer. Java compiler al does the same thing. Then, how does Java achieve architecture neutrality? The answer is that the Ja compiler produces an intermediate code known as bytecode for a machine that does not exists. This m chine is called the Java Virtual Machine and it exists only inside the computer memory. It is a simulat within the computer and does all major functions of a real computer. The Figure 2.1 illustrates the proc of compiling a Java Program into bytecode which is also referred to as virtual machine code.

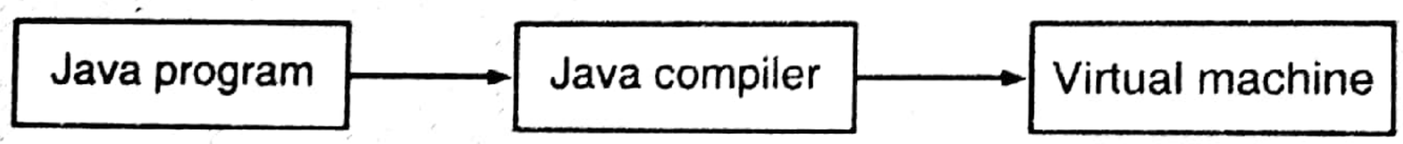

Fig. 2.1.

The Virtual machine code is not machine specific. The machine specific code (known as mach code) is generated by the Java interpreter by acting as an intermediary between the virtual machine and real machine as shown in Figure 2.2. Remember that the interpreter is different for different machines

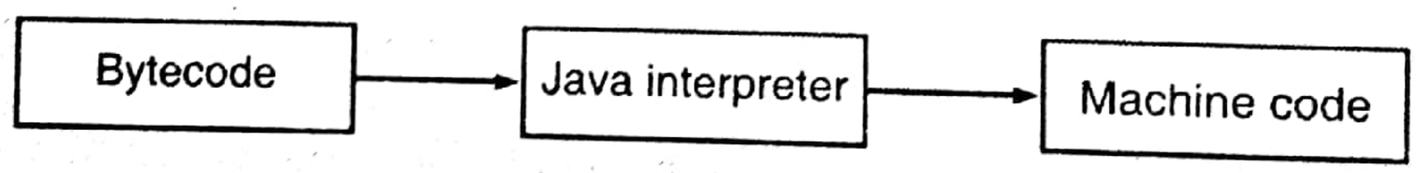

Fig. 2.2.

The Figure 2.3 shows whole Java works on a typical computer The Java Object framework (Java A acts as the intermediary between the user programs and the virtual machine which in turn acts as intermediary between the operating system and the Java object framework.

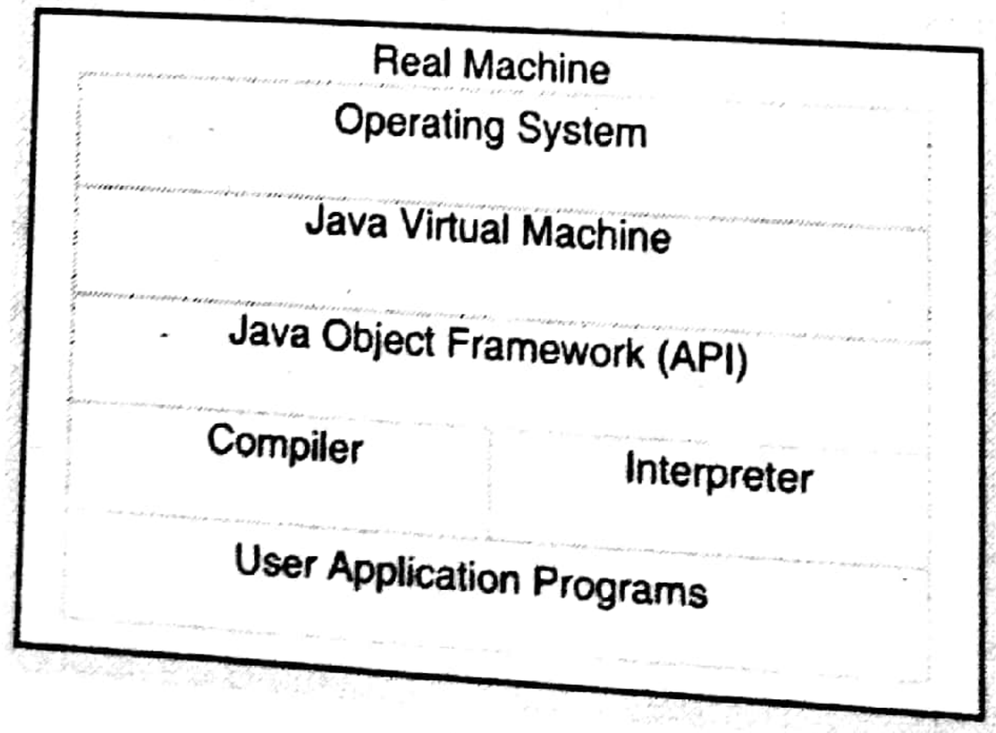

Fig. 2.3.

Scanned by CamScanner

2.2. JVM ARCHITECTURE

The Figure 2.4 explains the architecture of Java Virtual Machine.

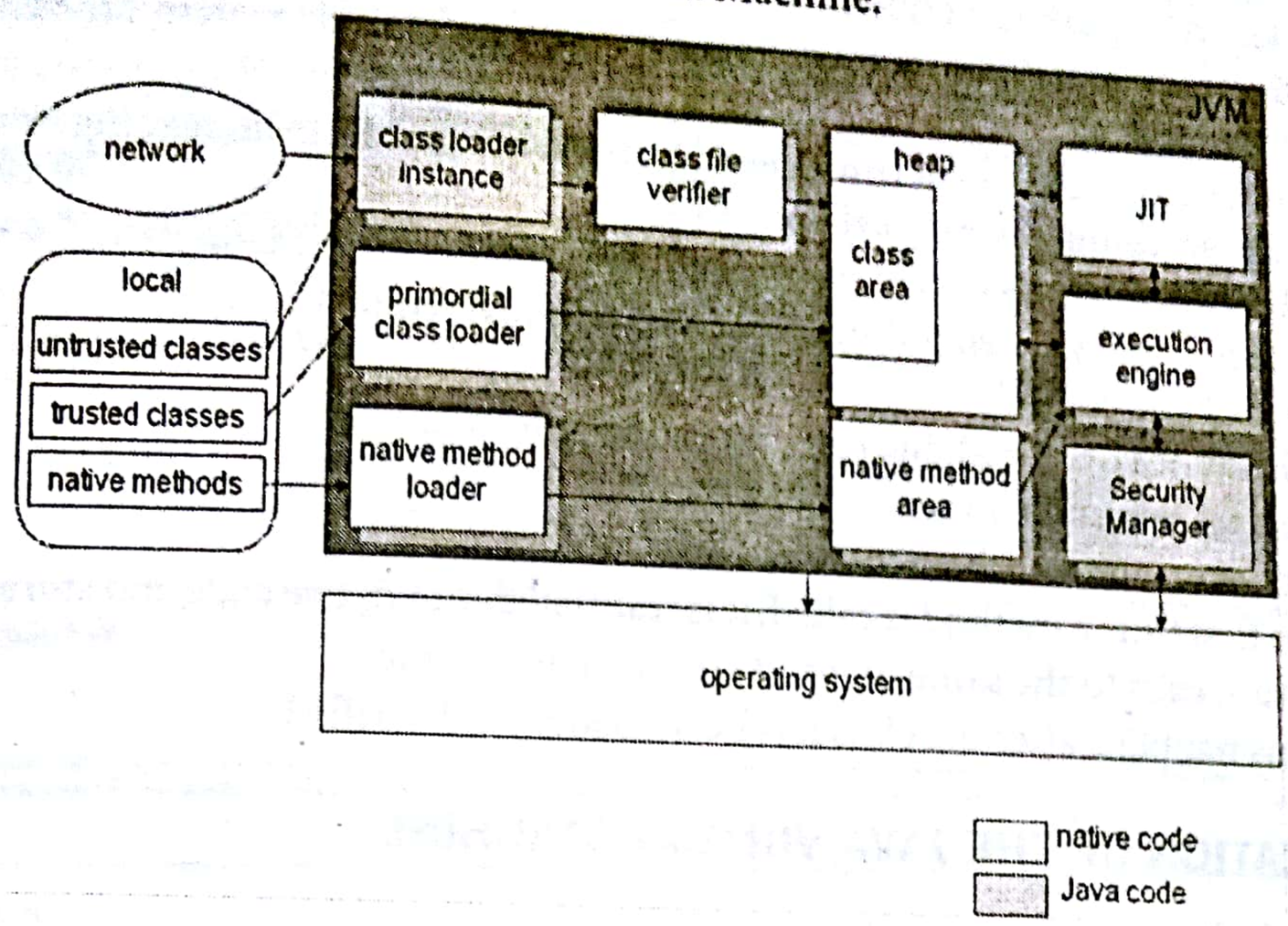

Fig. 2.4. The component of Java / The Execution Environment of JVM

Components of JVM

1. class loaders
   (*a*) locate and load classes into the JVM
   (*b*) primordial class loader
      (*i*) loads trusted classes (system classes found on the boot class path)
      (*ii*) integral part of the JVM
   (*c*) class loader instances
      (*i*) instances of Java.net.URLClassLoader (which extends SecureClassLoader)
      (*ii*) load untrusted classes from the local file system or from the network and passes them to the class file verifier
   (*d*) application developers can implement their own class loaders
2. class file verifier
   (*a*) checks untrusted class files
   (*b*) size and structure of the class file
   (*c*) bytecode integrity (references, illegal operations, ...)
   (*d*) some run-time characteristics (e.g., stack overflow)
   Note a class is accepted only if it passes the test
3. native method loader
   (*a*) native methods are needed to access some of the underlying operating system functions (e.g., graphics and networking features)
   (*b*) once loaded, native code is stored in the native method area for easy access
4. the heap
   (*a*) memory used to store objects during execution
   (*b*) how objects are stored is implementation specific

5. execution engine
   (a) a virtual processor that executes bytecode
   (b) has virtual registers, stack, etc.
   (c) performs memory management, thread management, calls to native methods, etc.
6. Security Manager
   (a) enforces access control at run-time (e.g., prevents applets from reading or writing to the file system, accessing the network, printing, ...)
   (b) application developers can implement their own Security Manager
   (c) or use the policy based SM implementation provided by the JDK
7. JIT - Just In Time compiler
   (a) performance overhead due to interpreting bytecode
   (b) translates bytecode into native code on-the-fly
      - works on a method-by-method basis
      - the first time a method is called, it is translated into native code and stored in the class area
      - future calls to the same method run the native code
   (c) all this happens after the class has been loaded and verified

2.3. ORGANIZATION OF THE JAVA VIRTUAL MACHINE

The JVM is divided into three conceptual data spaces:

- **Class area,** where the code and constants are kept
- **Java stack,** which keeps track of which methods have been called and the data associated with each method invocation
- **Heap,** where objects are kept

2.3.1. Class Area

The class area stores the classes that are loaded into the system. Method implementations are kept in a space called the method area, and constants are kept in a space called the constant pool. The class definitions also serve as templates for objects. Objects are stored in the heap. (See section 2.3.3).

A class loader introduces new classes into the class area. When a class is no longer used, it may be garbage collected.

Classes have several properties:

- Superclass
- List of interfaces (possibly empty)
- List of fields
- List of methods and implementations, stored in the method area
- List of constants, stored in the constant pool

All properties of classes are immutable. That is, there's no way to change any property of a class once it has been brought into the system. This makes the machine more stable, since you know that a method will have the same code each time it's invoked, and each object of a particular class has the same fields as every other object of that class.

If a class is no longer used, then it may be unloaded and reloaded at a later time. The new definition may be different. However, this does not cause inconsistencies between objects, because the original definition of the class would not be unloaded if instances of it still existed.

Fields come in two varieties: static and nonstatic. For nonstatic fields, there is a copy of the field for each object. For static fields, there is a single copy of the field for the entire class.

Scanned by CamScanner

See Figure 2.5 for a picture of a class area. It depicts two classes: GamePlayer and ChessPlayer. Each has fields and methods. Each field and method has a descriptor and a list of properties. The descriptor tells which type of values the field may hold or the parameters and return type of a method. The constant pool has been omitted from the figure. For methods that don't have the abstract property, there is a method implementation. Method implementations are defined in terms of instructions

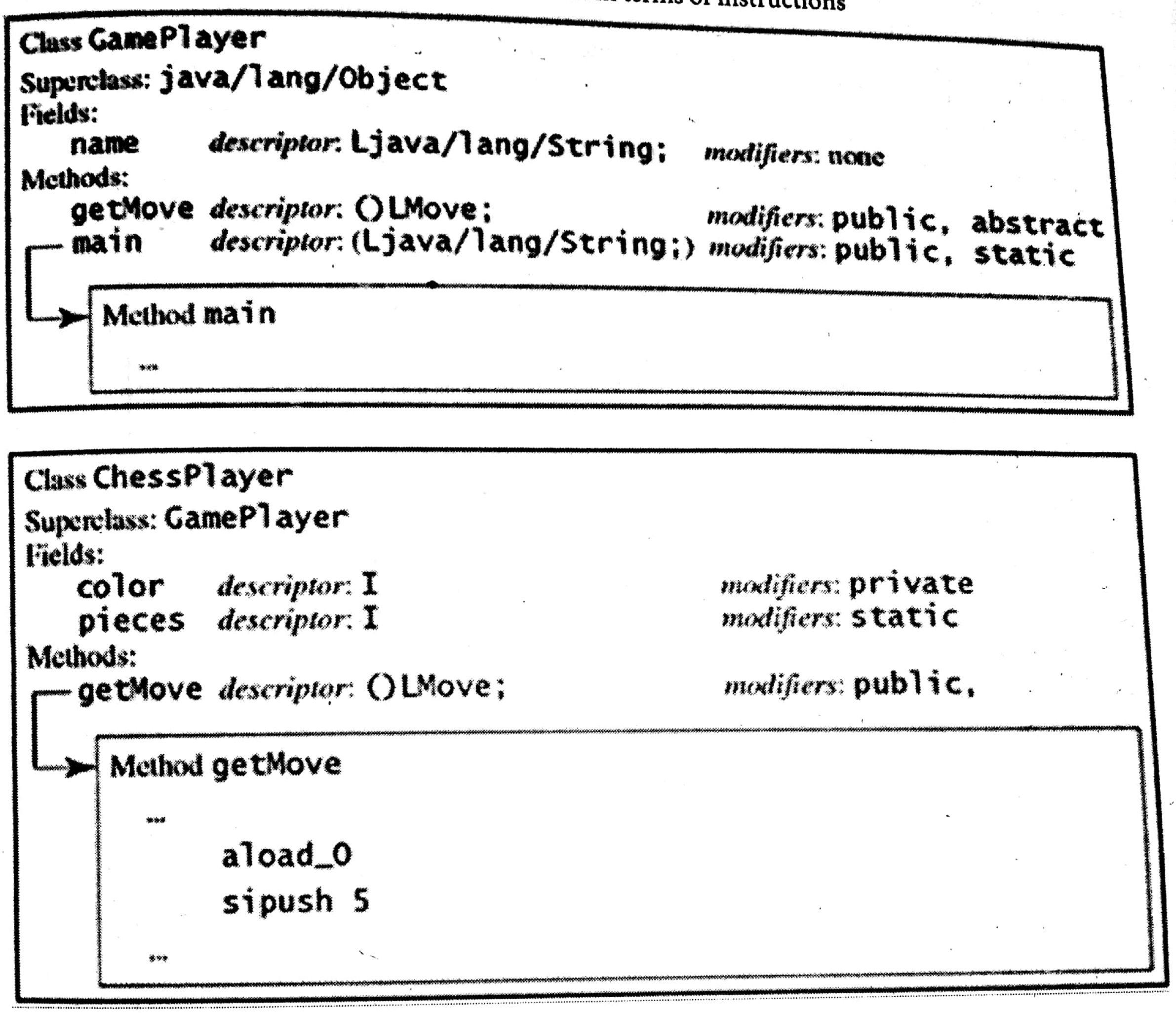

Fig. 2.5. Class area

2.3.2. Java Stack

Each time a method is invoked, a new data space called a stack frame is created. Collectively, the stack frames are called the Java stack. The stack frame on top of the stack is called the active stack frame.

Each stack frame has an operand stack, an array of local variables, and a pointer to the currently executing instruction. This instruction pointer is called the program counter (PC). The program counter points into the method area. It points to the current instruction. Ordinarily, the program counter moves from one instruction to the subsequent instruction, but some instructions (like goto) cause the program counter to move to some other place within the method.

Scanned by CamScanner

The top frame of the Java stack shows the current place of execution. It is called the active frame. Only the operand stack and local variable array in the active stack frame can be used. When a method is invoked, a new Java stack frame is created and that becomes the top of the Java stack. The program counter is saved as part of the old Java stack frame. The new Java stack frame has its own program counter, which points to the beginning of the called method.

When the newly called method returns, the active stack frame disappears and the stack frame below it becomes the active frame again. The program counter is set to the instruction after the method call, and the method continues.

Figure 2.6 shows a Java stack that has two stack frames. The first entry on the stack is at the bottom. It shows a call to the method main in the class GamePlayer. The program counter points to the instruction nine bytes from the beginning of the method.

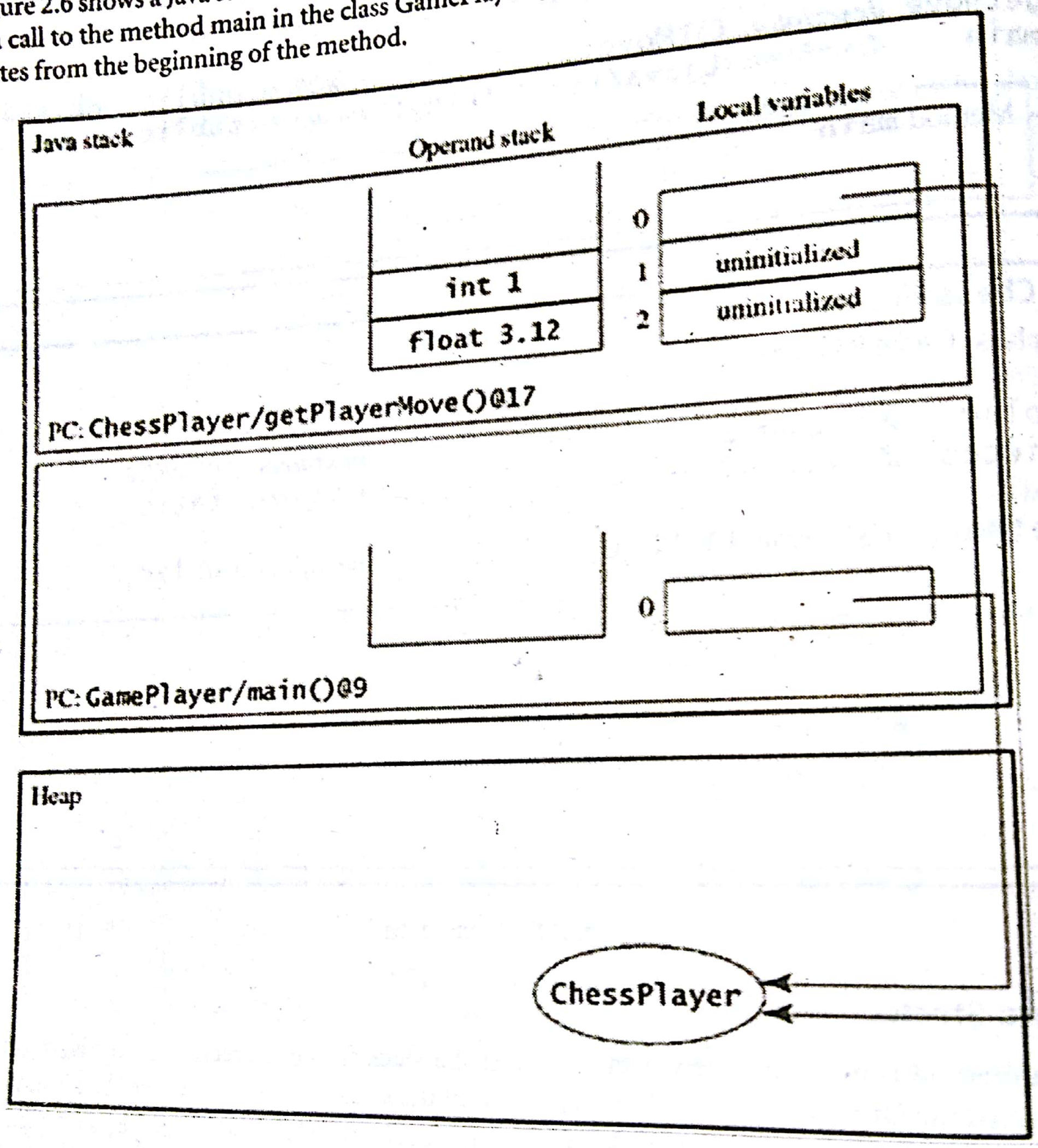

Fig. 2.6. Java stack and heap

Above that is the active stack frame. The active stack frame is a method called getPlayerMove in the class ChessPlayer, currently at instruction 17. It has two items on its operand stack and three slots in its local variable array. Two of those slots are uninitialized. Slot 0 contains a reference to an object in the heap. The references are represented by arrows that point to the objects.

2.3.3. Heap

Objects are stored in the heap. Each object is associated with a class in the class area. Each object also has a number of slots for storing fields; there is one slot for each nonstatic field in the class, one for each nonstatic field in the superclass, and so on. An example is shown in Figure 2.7. This heap contains a chess player named Pooky. A real heap has thousands or millions of objects in it.

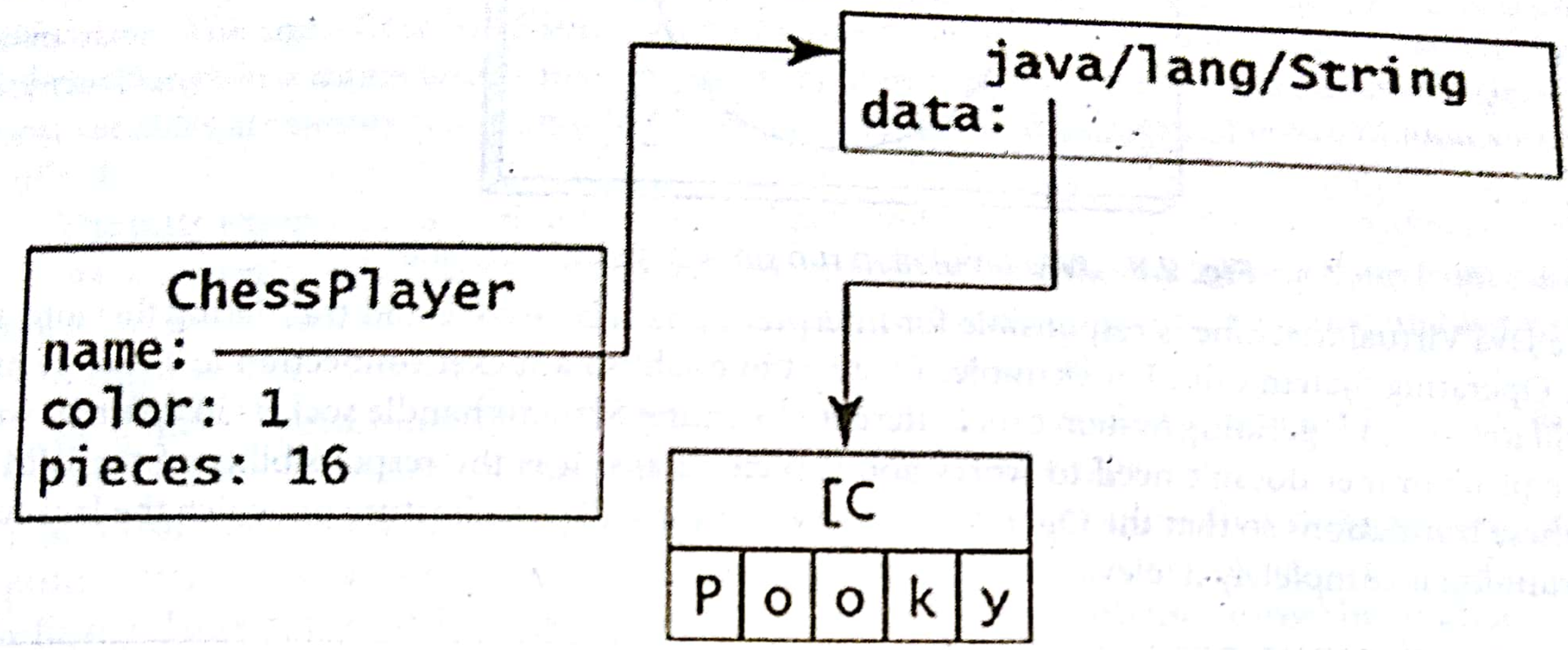

Fig. 2.7. Heap

The chess player is an object of class `ChessPlayer`. The name field is a reference to an object whose class is `java/lang/String`. This string contains a field called data, which points to an array of characters. ([C is the descriptor for an array of characters. This array is five characters long, containing the letters `P`, `o`, `o`, `k`, and `y`.

Another field of `ChessPlayer` is color, which is an `int`. This doesn't point to another object. Instead, it just holds the number 1.

2.4. JVM AS AN EMULATOR

The JVM is an emulator that emulates the Java Bytecode. Java compiler does not directly convert high-level language such as C/C++ to the machine language (direct CPU instruction); it converts the Java language that the developer understands to the Java Bytecode that the JVM understands. Since Java bytecode has no platform-dependent code, it is executable on the hardware where the JVM has been installed, even when the CPU or OS is different. The size of the compiled code is almost identical to the size of the source code, making it easy to transfer and execute the compiled code via the network.

The difficult part of creating Java byte code is that the source code is compiled for a machine that does not exist. This machine is called the Java Virtual Machine, and it exists only in the memory of our computer. Fooling the Java compiler into creating byte code for a nonexistent machine is only one-half of the ingenious process that makes the Java architecture neutral. The Java interpreter must also make our computer and the byte code file believe they are running on a real machine. It does this by acting as the intermediary between the Virtual Machine and our real machine.

Scanned by CamScanner

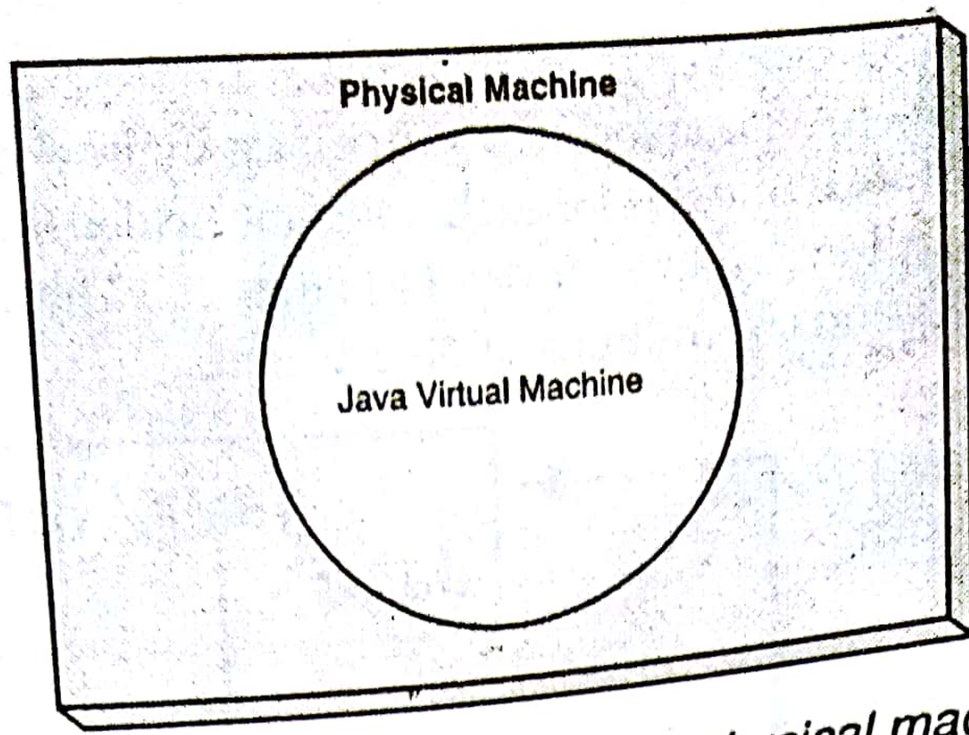

Fig. 2.8. JVM emulation run on a physical machine

The Java Virtual Machine is responsible for interpreting Java byte code and translating this into actions or Operating System calls. For example, a request to establish a socket connection to a remote machine will involve an Operating System call. Different Operating Systems handle sockets in different ways - but the programmer doesn't need to worry about such details. It is the responsibility of the JVM to handle these translations so that the Operating System and the CPU architecture on which the Java software is running is completely irrelevant to the developer.

2.5. JVM AS AN INTERPRETER

An interpreter is a program that acts much like a CPU, with a kind of fetch-and-execute cycle. In order to execute a program, the interpreter runs in a loop in which it repeatedly reads one instruction from the program, decides what is necessary to carry out that instruction, and then performs the appropriate machine-language commands to do so. One use of interpreters is to execute high-level language programs. However, interpreters have another purpose: they can let you use a machine-language program meant for one type of computer on a completely different type of computer. For example, one of the original home computers was the Commodore 64 or "C64". While you might not find an actual C64, you can find programs that run on other computers -- or even in a web browser -- that "emulate" one. Such an emulator can run C64 programs by acting as an interpreter for the C64 machine language.

The designers of Java chose to use a combination of compilation and interpreting. Programs written in Java are compiled into machine language, but it is a machine language for a computer that doesn't really exist. This so-called "virtual" computer is known as the Java Virtual Machine, or JVM. The machine language for the Java Virtual Machine is called Java bytecode. There is no reason why Java bytecode couldn't be used as the machine language of a real computer, rather than a virtual computer. But in fact the use of a virtual machine makes possible one of the main selling points of Java: the fact that it can actually be used on any computer. All that the computer needs is an interpreter for Java bytecode. Such an interpreter simulates the JVM in the same way that a C64 emulator simulates a Commodore 64 computer. (The term JVM is also used for the Java bytecode interpreter program that does the simulation, so we say that a computer needs a JVM in order to run Java programs. Technically, it would be more correct to say that the interpreter implements the JVM than to say that it is a JVM.) Of course, a different Java bytecode interpreter is needed for each type of computer, but once a computer has a Java bytecode interpreter, it can run any Java bytecode program, and the same program can be run on any computer that has such an interpreter. This is one of the essential features of Java: the same compiled program can be run on many different types of computers.

Scanned by CamScanner

2.6. THE JAVA VIRTUAL MACHINE INSTRUCTIONS SET

A Java virtual machine instruction consists of an opcode specifying the operation to be performed, followed by zero or more operands embodying values to be operated upon. The Java virtual machine checks that the code satisfies the static and structural constraints at link time using a class file verifier.

2.6.1. Types of Java Virtual Machine Instructions set

The JVM is a stack-oriented interpreter that creates a local stack frame of fixed size for every method invocation. The size of the local stack has to be computed by the compiler. Values may also be stored intermediately in a frame area containing local variables which can be used like a set of registers. These local variables are numbered from 0 to 65535, i.e., you have a maximum of 65536 of local variables per method.

The instruction set can be roughly grouped as follows:

Stack operations: Constants can be pushed onto the stack either by loading them from the constant pool with the ldc instruction or with special "short-cut" instructions where the operand is encoded into the instructions, e.g., `iconst_0 or bipush` (push byte value).

Arithmetic operations: The instruction set of the Java Virtual Machine distinguishes its operand types using different instructions to operate on values of specific type.

Control flow: There are branch instructions like `if_icmpeq`, which compares two integers for equality. There is also a `jsr` (jump to sub-routine) and ret pair of instructions that is used to implement the finally clause of try-catch blocks. Exceptions may be thrown with the athrow instruction.

Field access: The value of an instance field may be retrieved with getfield and written with putfield. For static fields, there are getstatic and putstatic counterparts.

Method invocation: Static Methods may either be called via invokestatic or be bound virtually with the invokevirtual instruction. Super class methods and private methods are invoked with invokespecial. A special case is interface methods which are invoked with invokeinterface.

Object allocation: Class instances are allocated with the new instruction, arrays of basic type like int[] with `newarray`, arrays of references like `String[ ][ ]` with `anewarray` or `multianewarray`.

Conversion and type checking: For stack operands of basic type there exist casting operations like f2i which converts a float value into an integer. The validity of a type cast may be checked with checkcast and the instanceof operator can be directly mapped to the equally named instruction.

2.6.2. Format of Instruction Descriptions

The instruction format of Java language is represented as under: - Mnemonic means a symbol.

Operation : Short description of the instruction Format

mnemonic

operand 1

operand 2

- - - -

Forms

mnemonic = opcode- Its opcode is its numeric representation and is given in both decimal and hexa-decimal forms. Only the numeric representation is actually present in the Java virtual machine code in a class file.

Operand Stack - A longer description detailing constraints on operand stack contents or constant pool entries, the operation performed the type of the results, etc.

Scanned by CamScanner

..., *value1, value2* ..., *result* shows an operation that begins by having value2 on top of the operand stack with value1 just beneath it.

Linking Exceptions : If any linking exceptions may be thrown by the execution of this instruction, they are set off one to a line, in the order in which they must be thrown.

Runtime Exceptions : If any runtime exceptions can be thrown by the execution of an instruction, they are set off one to a line, in the order in which they must be thrown.

2.7. CLASS FILE FORMAT

The Figure 2.9 shows a simplified example of the contents of a Java class file: It starts with a header containing a "magic number" (`0xCAFEBABE`) and the version number, followed by the constant pool, which can be roughly thought of as the text segment of an executable, the *access rights* of the class encoded by a bit mask, a list of interfaces implemented by the class, lists containing the fields and methods of the class, and finally the class attributes, e.g., the `SourceFile` attribute telling the name of the source file. Attributes are a way of putting additional, user-defined information into class file data structures. For example, a custom class loader may evaluate such attribute data in order to perform its transformations. The JVM specification declares that unknown, i.e., user-defined attributes must be ignored by any Virtual Machine implementation.

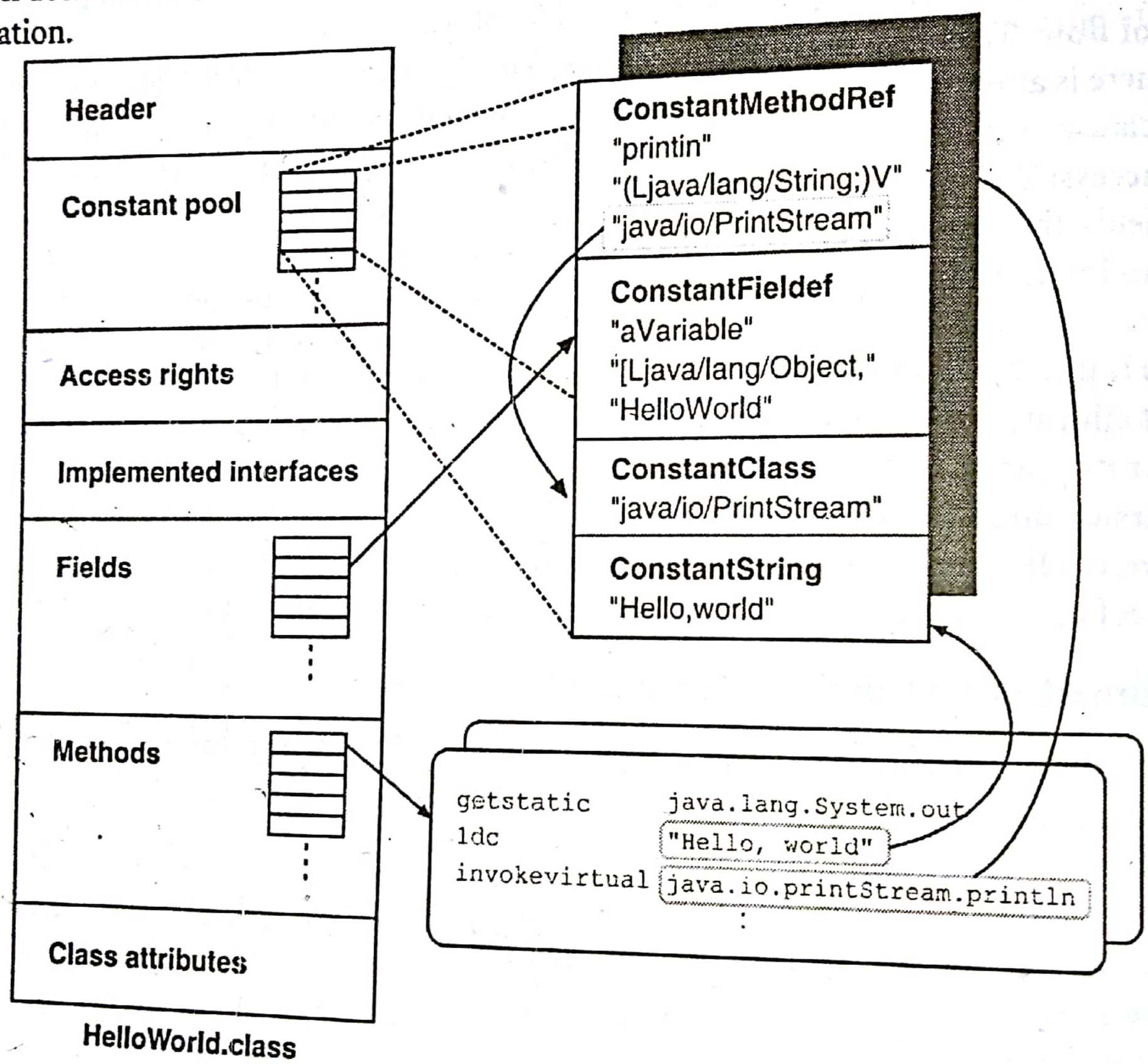

Fig. 2.9. Java class file format

2.7.1. Instrumentation of a .class file

Each class file contains the definition of a single class or interface. A class file consists of a stream of 8-bit bytes. All 16-bit, 32-bit, and 64-bit quantities are constructed by reading in two, four, and eight consecutive 8-bit bytes, respectively.

A class file consists of a single ClassFile structure:

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

The items in the ClassFile structure are as follows:

magic - The magic item supplies the magic number identifying the class file format; it has the value 0xCAFEBABE.

minor_version, major_version - The values of the minor_version and major_version items are the minor and major version numbers of this class file. Together, a major and a minor version number determine the version of the class file format. If a class file has major version number M and minor version number m, we denote the version of its class file format as M.m. Thus, class file format versions may be ordered lexicographically, for example, 1.5 < 2.0 < 2.1.

constant_pool_count - The value of the constant_pool_count item is equal to the number of entries in the constant_pool table plus one. A constant_pool index is considered valid if it is greater than zero and less than constant_pool_count, with the exception for constants of type long and double.

constant_pool[] - The constant_pool is a table of structures representing various string constants, class and interface names, field names, and other constants that are referred to within the ClassFile structure and its substructures. The format of each constant_pool table entry is indicated by its first "tag" byte.

The **constant_pool** table is indexed from 1 to constant_pool_count-1.

access_flags - The value of the access_flags item is a mask of flags used to denote access permissions to and properties of this class or interface.

2.7.2. Byte code engineering libraries

The Byte Code Engineering Library (formerly known as JavaClass) is intended to give users a convenient possibility to analyze, create, and manipulate (binary) Java class files (those ending with .class). Classes are represented by objects which contain all the symbolic information of the given class: methods, fields and byte code instructions.

Scanned by CamScanner

Such objects can be read from an existing file, be transformed by a program (e.g. a class loader at ru time) and dumped to a file again. An even more interesting application is the creation of classes fro scratch at run-time. BCEL is already being used successfully in several projects such as compilers, optimize obsfuscators, bytecode verifiers and analysis tools.

BCEL contains a byte code verifier named JustIce, which usually gives you much better informati about what's wrong with your code than the standard JVM message.

2.8. VERIFICATION

In order to ensure that certain parts of the machine are kept safe from tampering, the JVM has a verific tion algorithm to check every class. The purpose of verification is to ensure that programs follow a set rules that are designed to protect the security of the JVM.

Programs can try to subvert the security of the JVM in a variety of ways. For example, they might t overflowing the stack, hoping to corrupt memory they are not allowed to access. They might try to cast object inappropriately, hoping to obtain pointers to forbidden memory. The verification algorithm e sures that this does not happen by tracing through the code to check that objects are always used accordi to their proper types.

The verification algorithm is applied to every class as it is loaded into the system, before instances a created or static properties used. This allows the JVM implementation to assume that the class has certa safety properties, which permit the implementation to make optimizations based on that assumption.

The verification algorithm makes it possible to safely download Java applets from the Internet. I web browser, a class called the SecurityManager ensures that some JVM capabilities are available only authorized programs, and the verification algorithm makes sure that the programs don't do anything t would enable them to circumvent the SecurityManager.

2.8.1. How the Verification Algorithm Works

The Java Virtual Machine Specification contains a large set of rules that programs must follow if they w to run in the machine. It is the job of the verification algorithm to prove that each and every one of th rules has been followed. These rules were designed to be verifiable by just examining each class, with resorting to running the class.

The verification algorithm works by asking a set of questions about the class file. These questions into five general categories.

1. Is it a structurally valid class files?
2. Are all constant references correct?
3. Are the instructions valid?
4. Will the stack and local variables always contain values of the appropriate types?
5. Do the classes used really exist, and do they have the necessary methods and fields?

These questions can be answered statically, that is, from looking at the bytecodes without execu the program. This enables the virtual machine to stop a badly behaving program before it starts. It permits faster execution, because the virtual machine doesn't have to check some kinds of errors as program runs. If the program has passed verification, then these errors cannot occur.

2.8.2. Java Language and Verification Algorithm

The most common programming language for writing Java virtual machine programs is Java. One t the language guarantees is that anything you can say in Java can be translated into virtual machine c that will pass the verification algorithm.

The verification rules state that the arithmetic instructions (iadd, ddiv, etc.) always take two values of the same type (two ints, two doubles, etc.). Say you have the following Java program fragment:

```
int i = 70;              // Call this variable 1
float j = 111.1;         // Call this variable 2
double k = i+j;          // Call this variable 3 (and 4)
```

A naïve (and invalid) translation into bytecodes would be

```
bipush 70        ; Initialize i (variable 1) to 70
istore_1
ldc 111.1        ; Initialize j (variable 2) to 111.1
fstore_1
iload_1          ; Push the integer 70
fload_2          ; Push the float 111.1
fadd             ; Add two floats (This is an error!)
dstore_3         ; Store it in variables 3 and 4 (Another error!)
```

The verification algorithm rejects this, because it attempts to apply a fadd instruction when the top of the stack contains an int and a float instead of two floats. The attempt to store a float value with a dstore instruction is also invalid.

The Java language has a complex set of implicit and explicit type conversion rules, described in chapter 5 of The Java Programming Language Specification. One of those rules says that when you apply an arithmetic operator to a float and an int, you must first convert the int into a `float`. This is done automatically by the compiler. A proper Java compiler would instead generate

```
bipush 70        ; Initialize i (variable 1) to 70
istore_1
ldc 111.1        ; Initialize j (variable 2) to 111.1
fstore_1
iload_1          ; Push the integer 70
i2f              ; Convert the integer to a float
fload_2          ; Push the float 111.1
fadd             ; Now adding two floats is legal
f2d              ; Convert the result to a double
dstore_3         ; Store it in variable 3
```

The verification algorithm has no problem with this code.

The verification rules also require that no matter how you get to a particular instruction, whether by falling through from the previous instruction or by use of a goto or one of the if instructions, the stack will always have the same structure. A Java compiler can make sure that this is true by following two rules of its own:

- Each Java statement translates so that it leaves nothing on the stack.
- You can branch only to the beginning of a statement.

Scanned by CamScanner

To see how these two rules help the compiler enforce the JVM verification algorithm, consider the following code from a class called Parser:

```
if(isLegalToken)
    getNextToken();
else
    printErrorMessage("Invalid token");
```

with the following declarations

```
static int getNextToken();
static void printErrorMessage();
```

A Java compiler generates code like

```
line1: getstatic Parser/isLegalToken Z
                                ; if(isLegalToken)
line2: ifeq 16
                                ; getNextToken();
line3: invokestatic Parser/getNextToken ()I
line4: pop
line5: goto 18
line6: ldc "Invalid token"      ; else
                                ; printErrorMessage("Invalid token");
line7: invokestatic Parser/printErrorMessage
            (Ljava/lang/String;)V
line8:                          ; The method continues here
```

The statement `getnextToken();` translates into `line3` and `line4`. The statement `printErrorMessage("Invalid Token")` translates into lines `line6` and `line7`.

The call to getNextToken leaves its result on the stack (an int). The instruction at line4 pops that value off the stack, since the program doesn't use the value. The program pushes a message onto the stack on line6, but the call to `printErrorMessage` pops it off and leaves nothing in its place. In each case, the net effect is an empty stack.

Both branches of the if end up at line8 with nothing on the stack. That means that the if statement as a whole also works as a statement. You could put this if statement inside another if or inside a for loop. Since its net effect on each statement on the stack is zero, you'll never violate the virtual machine's rules.

A Java compiler is free to generate any virtual machine code it wants to, as long as the resulting code does the same thing as the original Java program. By breaking these rules in carefully controlled ways a compiler can generate much faster code with the same thing, without compromising safety.

2.9. JAVA HEAP & STACK

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 2-10 for a graphical depiction of these memory areas.

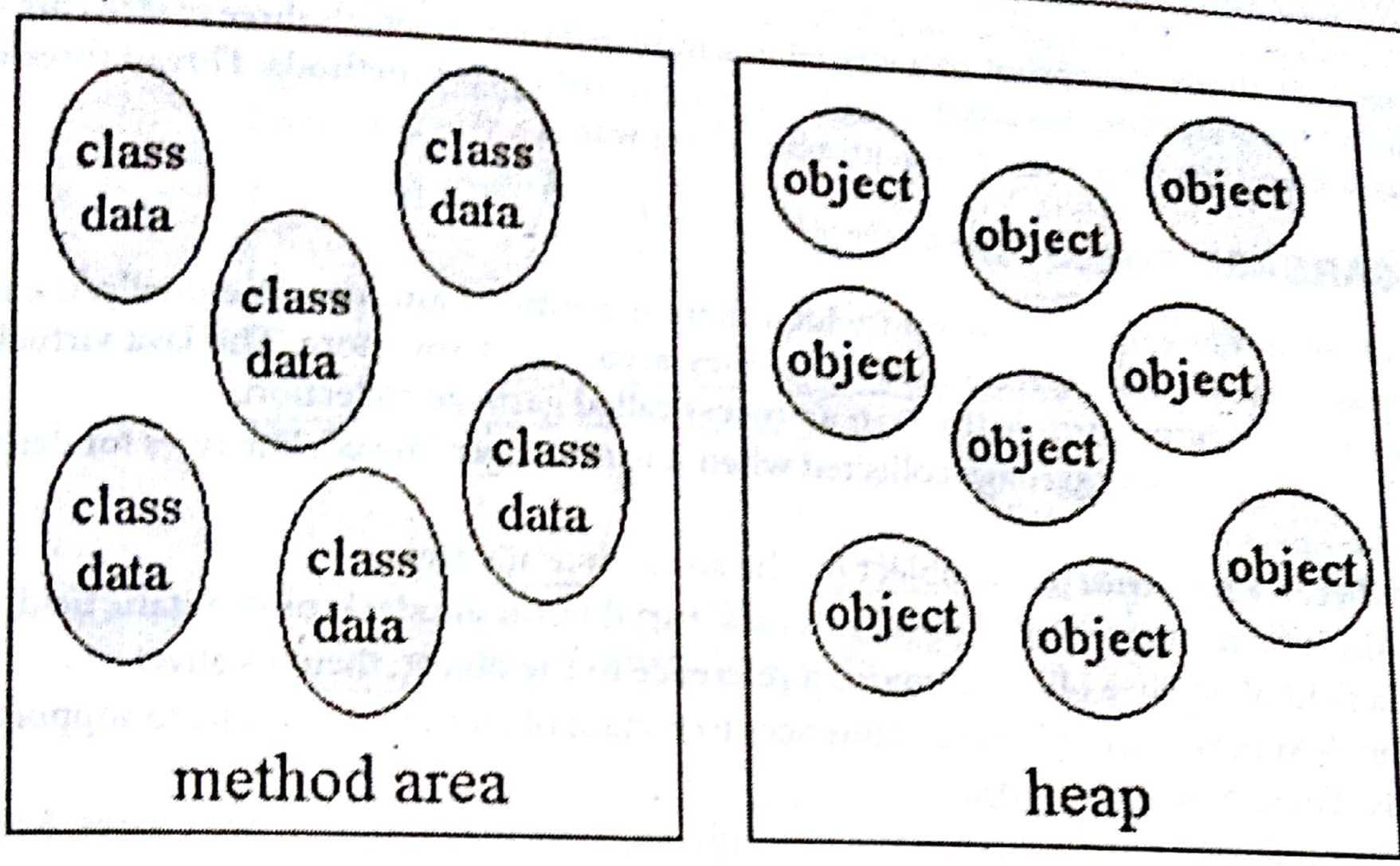

Fig. 2.10. Runtime data areas shared among all threads

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.

See Figure 2-11 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.

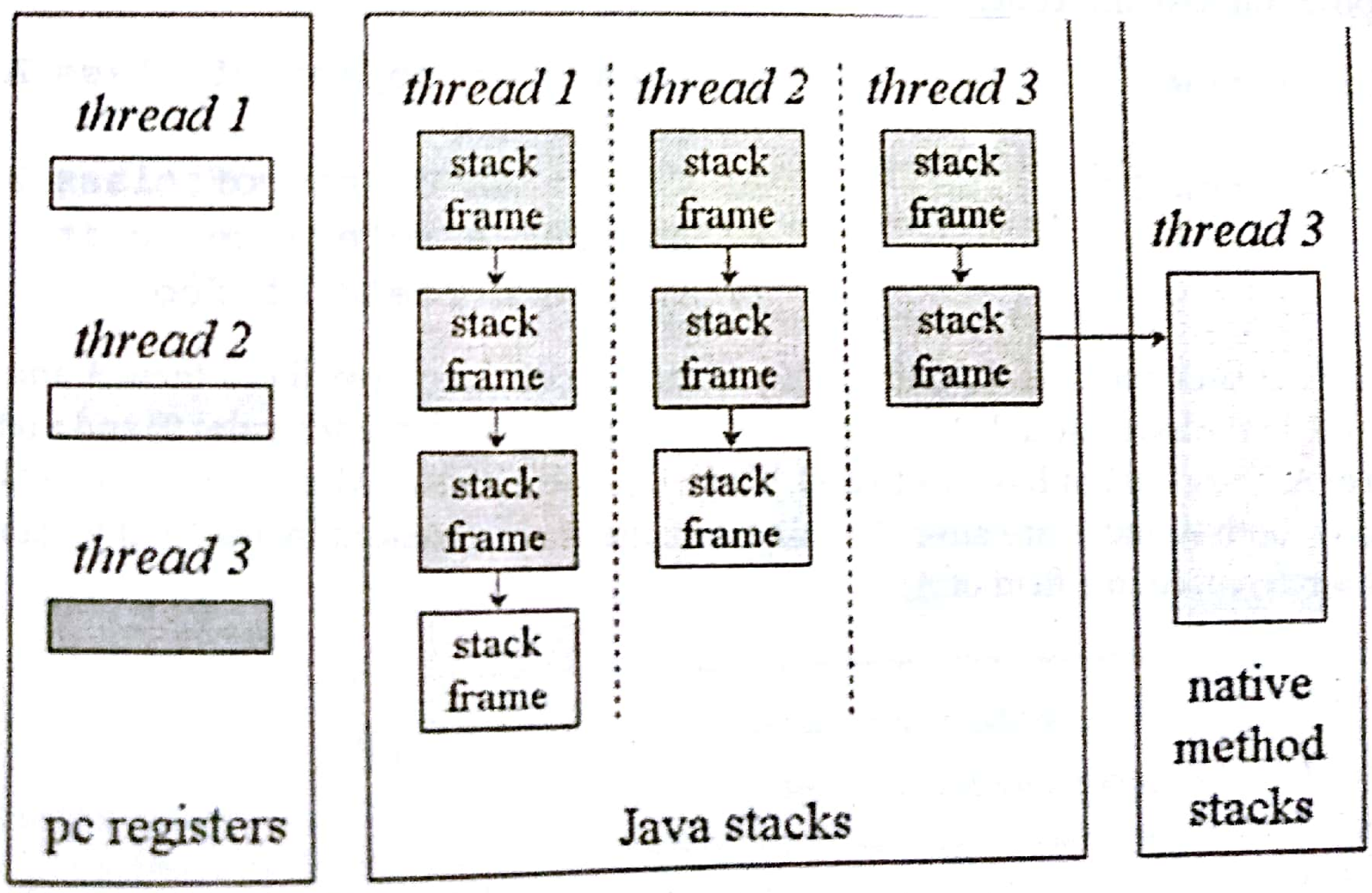

Fig. 2.11. Runtime data areas exclusive to each thread

Figure 2-11 shows a snapshot of a virtual machine instance in which three threads are executing the instant of the snapshot, threads one and two are executing Java methods. Thread three is executing native method and the stacks are shown growing downwards.

2.10. GARBAGE COLLECTION

Each object consumes some memory, of which there is a limited amount. Eventually, the memory allocated to these objects must be reclaimed when they aren't used any more. The Java virtual machine claims these objects automatically through a process called garbage collection.

An object is ready to be garbage collected when it is no longer "alive." The rules for determining if object is alive follow.

- If there's a reference to the object on the stack, then it's alive.
- If there's a reference to the object in a local variable, on the stack, or in a static field, then it's alive.
- If a field of an alive object contains a reference to the object, then it's alive.
- The JVM may internally keep references to certain objects, for example, to support native methods. These objects are alive.

Any object that isn't alive is dead. As an example, suppose that you have two classes, A and B, like this:

```
class A
{
   B b;                    // A field pointing to an object
                           // of type B
}
class B
{
   A a;                    // A field pointing to an object
                           // of type A
}
```

Suppose you have this code:

```
A foo = new A();           // Create an object of class A
                           // called foo
foo.b = new B();           // Create an object of class B
                           // and put a reference to it
                           // in the b field of foo
```

The code creates two objects, one of class A and one of class B (we'll call these A and B). There is a reference to A in the local variable foo (which is stored in local variable array slot 0) and a reference to B in the b field of A. The a field of B points to null, which is no object at all. Memory now looks like Figure 2-12. In this figure, both A and B are alive. A is alive because it is referenced by the local variable 0. B is alive because it is referenced in a field of A.

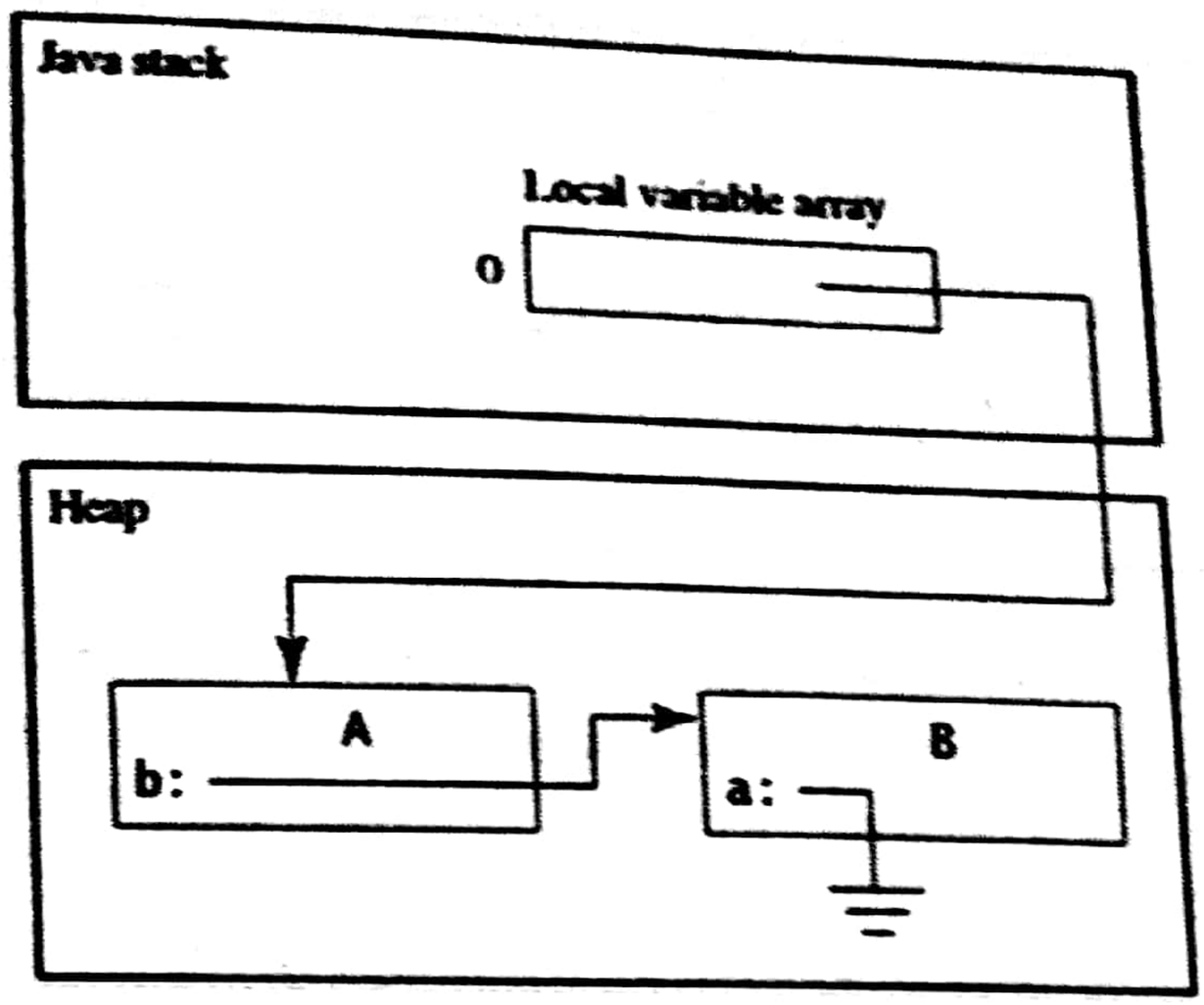

Fig. 2.12. Two live objects

Suppose we break the link like this:

```
foo.b = null;
```

This sets the b field to point to no object at all. Now memory looks like Figure 2.13. Now B doesn't meet any of the four conditions for aliveness; there are no arrows pointing to it at all.

This means that B is now dead, and it can be garbage collected. A is still alive, so it must not be garbage collected.

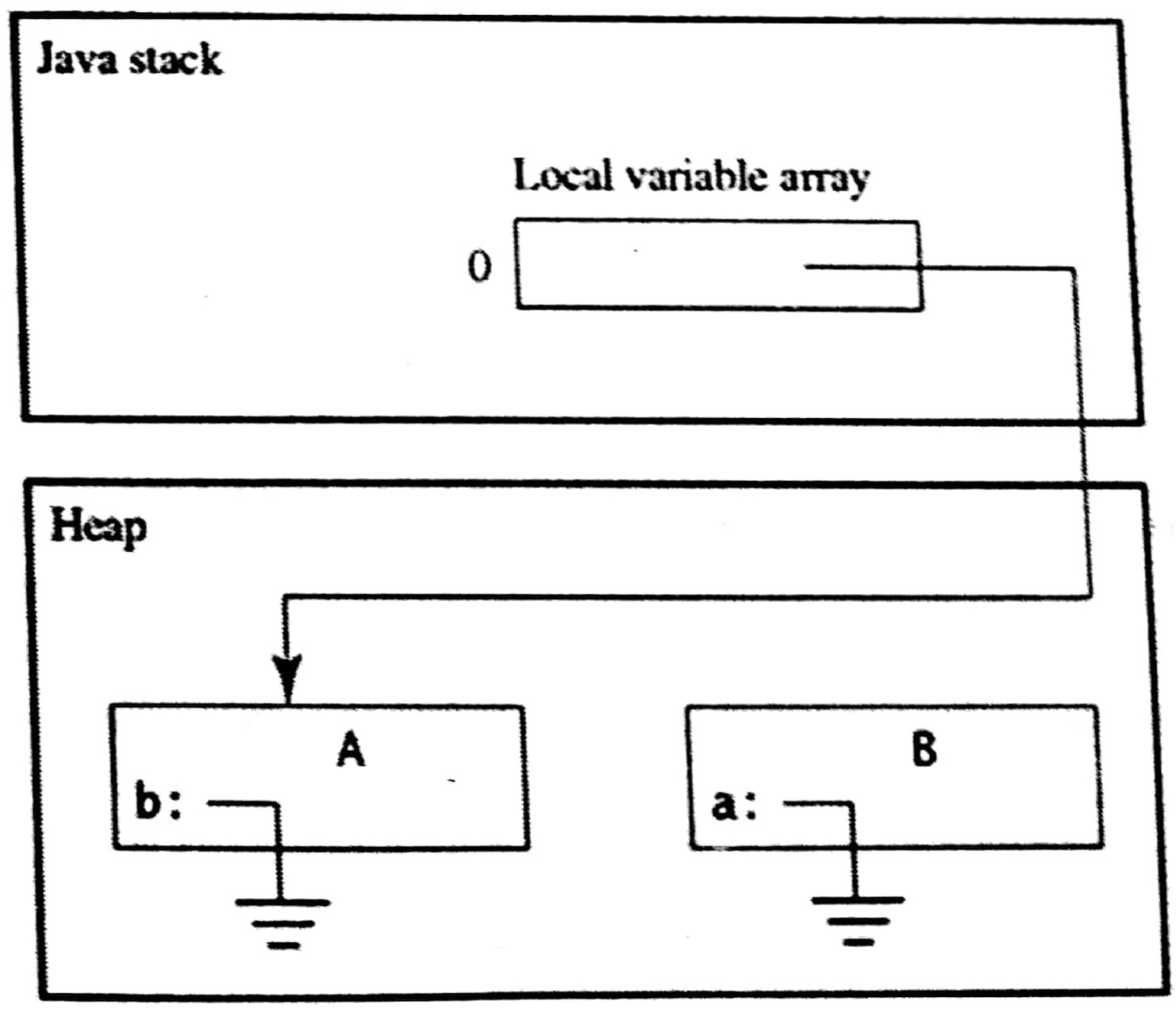

Fig. 2.13. The object on the right is dead

One thing can be a little confusing about the third rule: only a reference from a live object counts, not a reference from a dead one. Suppose there are objects like Figure 2.14. Even though these objects point to each other, neither is alive because one isn't alive. An object is dead unless it can prove that it's alive. Since the only claim that each has on life is the other one, they're both dead.

Scanned by CamScanner

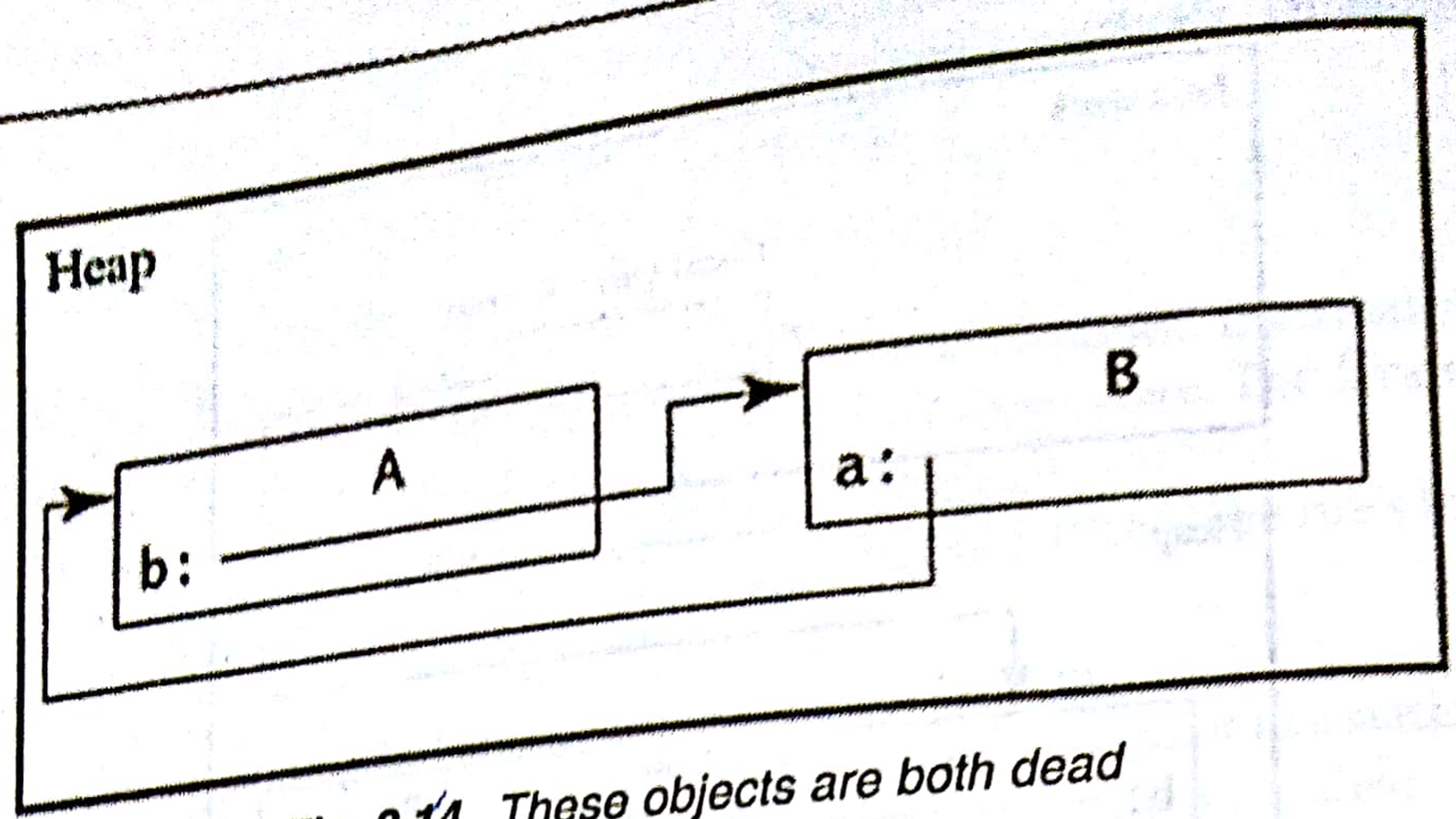

Fig. 2.14. These objects are both dead

2.10.1. Finalization

In some cases, it is helpful to know when an object is no longer used so that you can do some final cleanin up. For example, if you write a class where each instance represents a connection to a database, a databa connection object that is no longer used can inform the database that the connection is no longer nece sary.

To handle this, provide a method called finalize in your class. This method is called when the garba collector notices that the object is dead. The finalize method is defined in the class java.lang.Object to d nothing. You may override this method to do any last-minute chores your object should do before it garbage collected.

To close the database connection, you might provide this method:

```
public void finalize()
{
 this.close();                    // Close the connection
}
```

When this object becomes dead, the finalize method will be called and it will close the connection.

Note that you can't be sure when the connection will be closed, because the garbage collector may tak some time before it notices that the object is dead. In fact, the garbage collector may never notice that th object is dead, or finalization may never occur for other reasons. Therefore, anybody using the connectio should explicitly close the connection, not depend on finalize. The `finalize` method can be used as backup, but in that case there is still the possibility that the connection will never be closed.

One trick you can play is to "resurrect" your object. The `finalize` method was called because the was no way to reach the object, but in the body of the finalize method there is a way to reach the object: it available through this.

You can store the value of this into a field, like this:

```
static Vector resurrected_objects = new Vector();
void finalize()
{
 resurrected_objects.addElement(this);
}
```

This resurrects the object because now there is a way to get to it: it is stored in resurrected_objects which maintains a reference to the object. The garbage collector will notice this, and the object is n

reclaimed. The body of finalize may resurrect other objects that were presumed dead, or it may eliminate the last link to an object so that it is now dead. The garbage collector has to be very smart to correctly handle all these cases. You can be sure that an object will never be garbage collected until there is absolutely no way to resurrect it.

At some point in the future, the object may become dead again. The object will be garbage collected, but finalize will not run again. If you depend on finalize to clean things up, you must be very careful if you choose to resurrect the object during finalization: you get only one shot at it.

2.10.2 Timing of Garbage Collection

You shouldn't design your system to depend on finalization or garbage collection to run at any particular time or in any particular order. In fact, there is no guarantee that finalization will ever occur. Therefore, you should never write your programs to depend on finalization for correctness.

The garbage collector may run when you get low on memory. It may also run when the system is otherwise idle and the JVM decides to use the free time to tidy itself up. It may also run at the user's explicit command by calling the gc method in the class java.lang.System.

The garbage collector may be run automatically by the system at any time. The system can also undertake garbage collection at the explicit request of a program. Even if you knew when it was going to run, you couldn't be sure that it would collect all dead objects. Some garbage collectors take a quick-and-dirty look at the system to collect obviously dead objects and delay collection of objects that have been around for a long time until it has time to do a thorough examination of the state of memory.

2.11 JAVA SECURITY

Java is a secure language. Some of the details is as follows:

2.11.1 The need of Java Security

The term "security" is somewhat vague unless it is discussed in some context; different expectations of the term "security" might lead us to expect that Java programs would be:

- *Safe from malevolent programs*: Programs should not be allowed to harm a user's computing environment. This includes Trojan horses as well as harmful programs that can replicate themselves--computer viruses.
- *Non-intrusive*: Programs should be prevented from discovering private information on the host computer or the host computer's network.
- *Authenticated*: The identity of parties involved in the program should be verified.
- *Encrypted*: Data that the program sends and receives should be encrypted.
- *Audited*: Potentially sensitive operations should always be logged.
- *Well-defined*: A well-defined security specification would be followed.
- *Verified*: Rules of operation should be set and verified.
- *Well-behaved*: Programs should be prevented from consuming too many system resources.

The nature of the Internet created a new and largely unprecedented requirement for programs to be free of viruses and Trojan horses. But widespread downloading also led to a pervasive problem of malevolent attributes. For Java to succeed, it needed to circumvent the virus/trojan horse problems that plagued other models of software distribution.

Scanned by CamScanner

2.11.2 Security and the Virtual Machine

The first popular use of Java was to permit web surfers to download applets into their web browsers. This greatly extended the capabilities of web browsers, but some people were concerned that the capabilities had been extended too far, allowing malicious applet writers to wreak havoc on the computer of anybody using the applets. They pointed out that Java is a full-featured language, which would allow applet writers to do just about anything. It is extremely difficult to prove things about computer programs written in conventional languages, and it is usually easy to hide malicious code in an innocent-looking program.

The Java platform provides a solution to these security fears. The platform contains checks designed to prevent unfriendly behavior by limiting the capabilities of applets, placing potentially harmful operations off-limits. To ensure that these checks cannot be circumvented, the Java virtual machine contains a verification algorithm that eliminates the possibility of behavior that could be used to bypass the security checks.

2.11.3 Security Promises of the JVM

Here are some of the promises the Java virtual machine makes about programs that have passed the verification algorithm:

- Every object is constructed exactly once before it is used.
- Every object is an instance of exactly one class, which does not change through the life of the object.
- If a field or method is marked private, then the only code that ever accesses it is found within the class itself.
- Fields and methods marked protected are used only by code that participates in the implementation of the class.
- Every local variable is initialized before it is used.
- Every field is initialized before it is used.
- It is impossible to underflow or overflow the stack.
- It is impossible to read or write past the end of an array or before the beginning of the array.
- It is impossible to change the length of an array once it has been created.
- Final methods cannot be overridden, and final classes cannot be subclassed.
- Attempts to use a null reference as the receiver of a method invocation or source of a field cause a NullPointerException to be thrown.

2.11.4 Security Architecture and Security Policy

The Java platform builds a security architecture on top of the protections promised by the JVM. A security architecture is a way of organizing the software that makes up the Java platform so that potentially harmful operations are isolated from unprivileged code but available to privileged code. Most code is unprivileged; only carefully selected pieces of code are privileged to perform potentially dangerous operations. The security architecture is responsible for making sure that unprivileged code does not masquerade as privileged code.

The core of the Java platform security architecture under Java platforms 1.0 and 1.1 is the SecurityManager class. This class decides which pieces of code can perform certain operations and which cannot. Collectively, these decisions are called the security policy. The security policy is enforced by the Java platform classes, which check the `SecurityManager` before proceeding with any operation under the control of the `SecurityManager`.

On the Java 2 platform, the core of the security architecture is shifted to a class called `AccessController`, which falls outside the scope of this book. See http//java.sun.com for more information.

Scanned by CamScanner

Only one instance of the `SecurityManager` can be installed, and once it is installed it cannot be removed. It is called the security manager. By default, there is no security manager, and all operations are permitted. The class java.lang.System is responsible for ensuring that there is only one security manager. It provides the static methods `getSecurityManager` and `setSecurityManager` to get and set the security manager.

The `SecurityManager` class has a set of methods that are called by the Java platform code before proceeding with certain potentially harmful operations. These methods throw a `SecurityException` if the operation is forbidden. If no exception is thrown, then the caller may assume that the operation is permitted, and it can proceed with the operation. The security manager uses a variety of factors to determine whether an operation is permitted or not. These factors include the source of the code attempting the operation and the preferences of the user.

Following is an example of how the security architecture and the security policy interact.

Example

The security architecture demands that before a file is opened for reading, the security manager will be checked to see whether the code requesting the file is allowed to open the file. This is a sensible policy, since the ability to read files gives a program access to potentially important and confidential information.

The most important ways to read files are through the `FileInputStream` or `FileReader` classes in the `java.io` package. These classes include native methods that are implemented in a machine-dependent way to interact with the computer's file system. Before these methods can be used, an instance of FileInputStream or `FileReader` must be constructed. Following the security architecture, all the publicly accessible constructors of `FileInputStream` and `FileReader` begin with a call to the checkRead method of `SecurityManager`:

2.11.4.1 Basic Applet Security Policy

For the writers of web browsers, security is of tremendous concern, since a malicious or mistaken applet could cause problems for many, many web surfers. For this reason, the Java applet security policy is very restrictive. The security manager discriminates between two kinds of classes: system classes (loaded by the JVM automatically) and applets (loaded by a custom class loader from over the network). The web browser implements its own security manager, which subclasses `SecurityManager`.

The security manager needs to know which class is invoking the operation. For example, suppose an applet called NastyApplet tries to create a FileInputStream so that it can read the system's password file. In order to do that, it must call the constructor for FileInputStream, or the verification algorithm will reject the applet before it gets a chance to run.

2.12 THE JAVA SANDBOX SECURITY MODELS

Discussions of Java's security model often centered on the idea of a sandbox model. The idea behind this model is that when you allow a program to be hosted on your computer, you want to provide an environment where the program can play (i.e., run) and limit the resources that can be accessed by applets, but you want to confine the program's play area within certain bounds. You may decide to give the program certain toys to play with (i.e., you may decide to let it have access to certain system resources), but in general, you want to make sure that the program is confined to its sandbox.

The Java sandbox is responsible for protecting a number of resources, and it does so at a number of levels. Consider the resources of a typical machine as shown in. The user's machine has access to many things:

Scanned by CamScanner

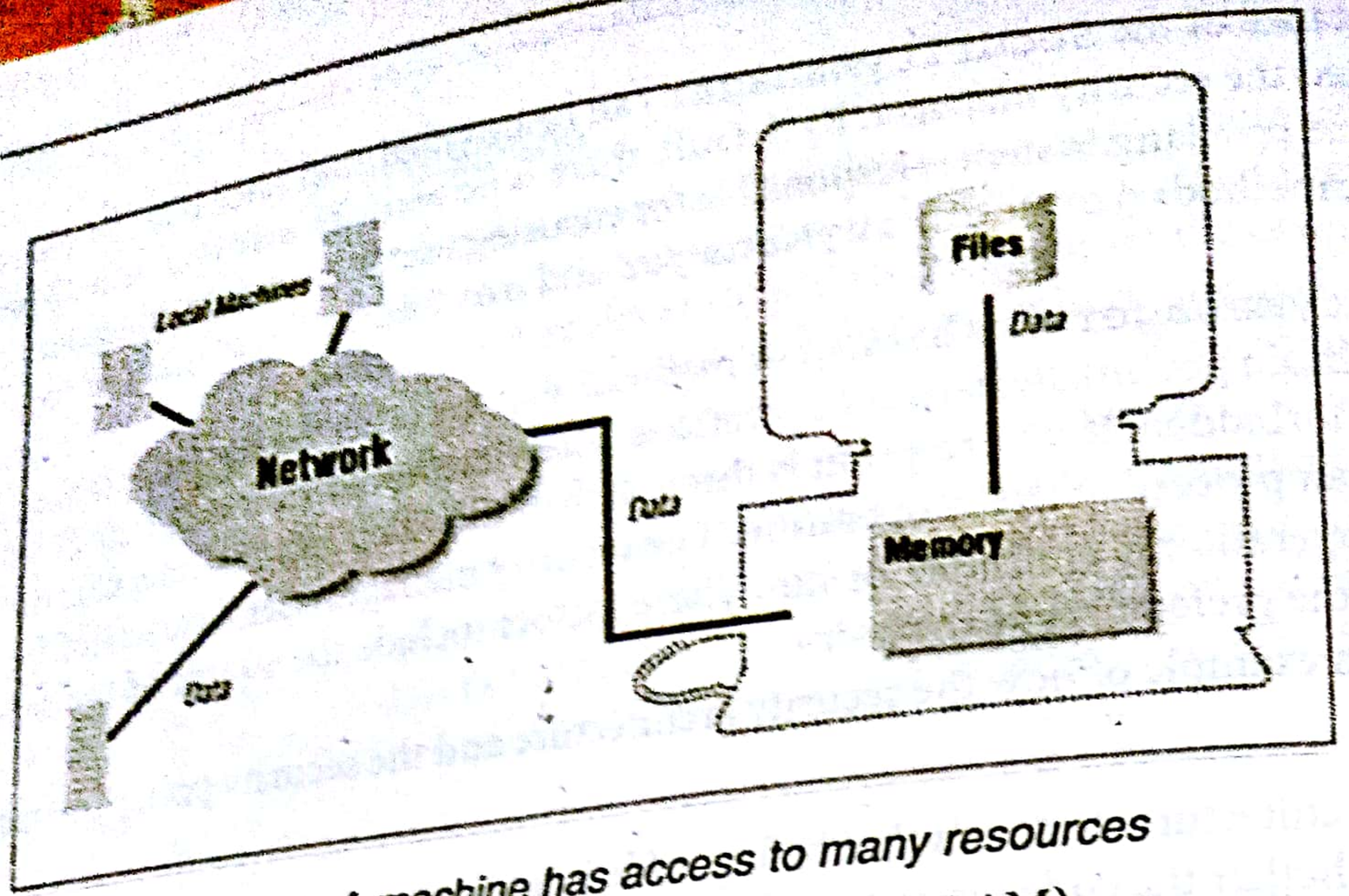

Fig. 2.15. A machine has access to many resources

- Internally, it has access to its local memory (the computer's RAM).
- Externally, it has access to its filesystem and to other machines on the local network.
- For running applets, it also has access to a web server, which may be on its local (private) net, or may be on the Internet.
- Data flows through this entire model, from the user's machine through the network and (possibly) to disk.
- Each of these resources needs to be protected, and those protections form the basis of Java's security model.

We can imagine a number of different-sized sandboxes in which a Java program might run:

- A sandbox in which the program has access to the CPU, the screen, keyboard, and mouse, and to its own memory. This is the minimal sandbox--it contains just enough resources for a program to run.
- A sandbox in which the program has access to the CPU and its own memory as well as access to the web server from which it was loaded. This is often thought of as the default state for the sandbox.
- A sandbox in which the program has access to the CPU, its memory, its web server, and to a set of program-specific resources (local files, local machines, etc.). A word-processing program, for example, might have access to the docs directory on the local Filesystem, but not to any other files.
- An open sandbox, in which the program has access to whatever resources the host machine normally has access to.

Fine grained access control (Java 2) - This model is based on the idea that every code (remote or local) has access to the system resources based on what is defined in a policy file. Features of this model are -

- introduced in Java 2
- a protection domain is an association of a code source and the permissions granted
- the code source consists of a URL and an optional signature
- permissions granted to a code source are specified in the policy file grant CodeBase "http://Java.sun.com", SignedBy "Sun" { permission Java.io.FilePermission "${user.home}${/}*", "read, write"; permission Java.net.SocketPermission "localhost:1024-", "listen";};

2.13 OV

As shown i
resources a
intercept th
mations be

During
the JVM a
tion to th
JVM's rul

Fig
memor
mecha
chine a
the me

W
and o
metho
chine

A
tion,
areas

Scanned by CamScanner

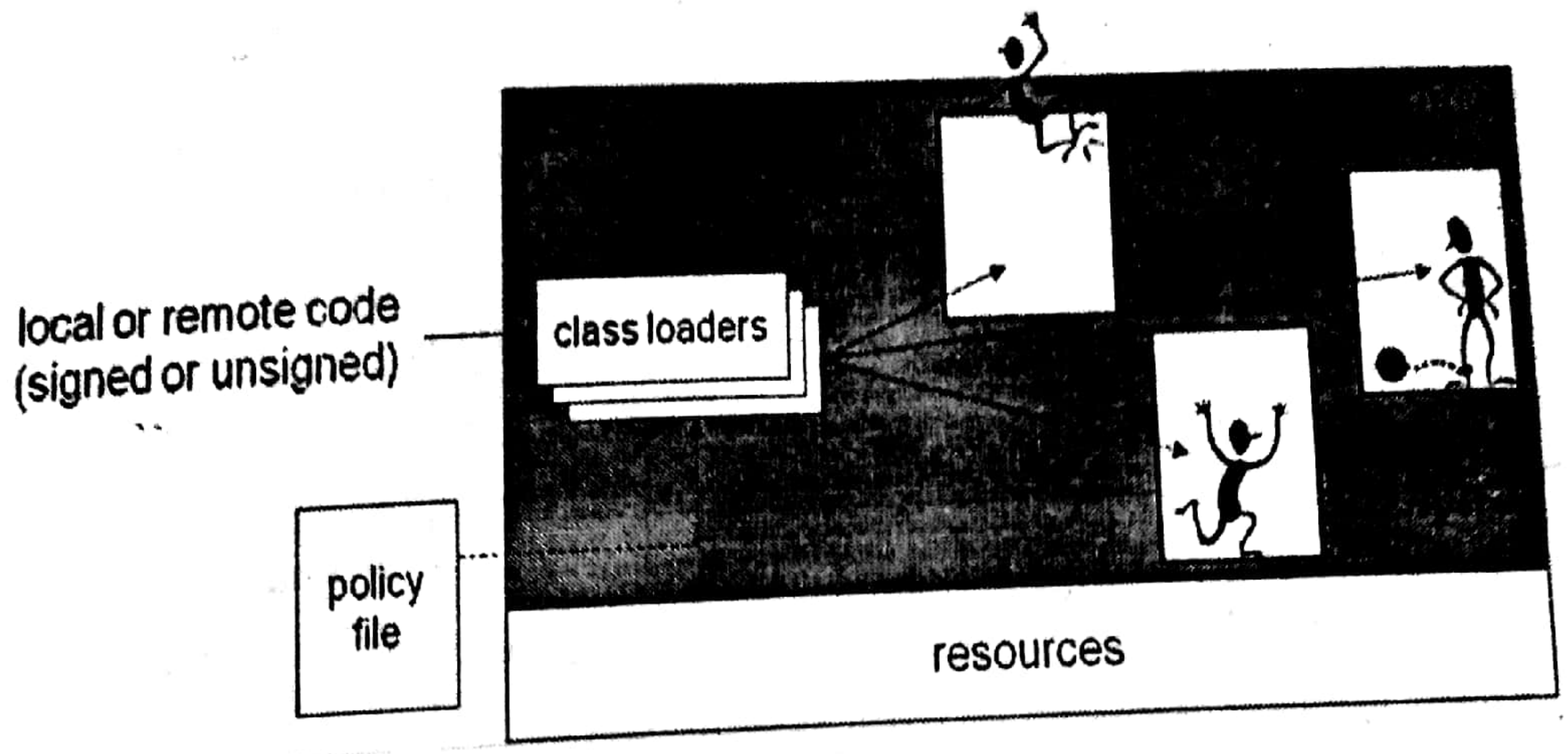

Fig. 2.16.

2.13 OVERVIEW OF CLASS LOADER

As shown in fig 2.17 Class loaders are responsible for loading class files from the file system or other resources and passing the byte code to the Virtual Machine. A custom ClassLoader object may be used to intercept the standard procedure of loading a class, i.e. the system class loader, and perform some transformations before actually passing the byte code to the JVM.

During run-time the Virtual Machine requests a custom class loader to load a given class. But before the JVM actually sees the byte code, the class loader makes a "side-step" and performs some transformation to the class. To make sure that the modified byte code is still valid and does not violate any of the JVM's rules it is checked by the verifier before the JVM finally executes it.

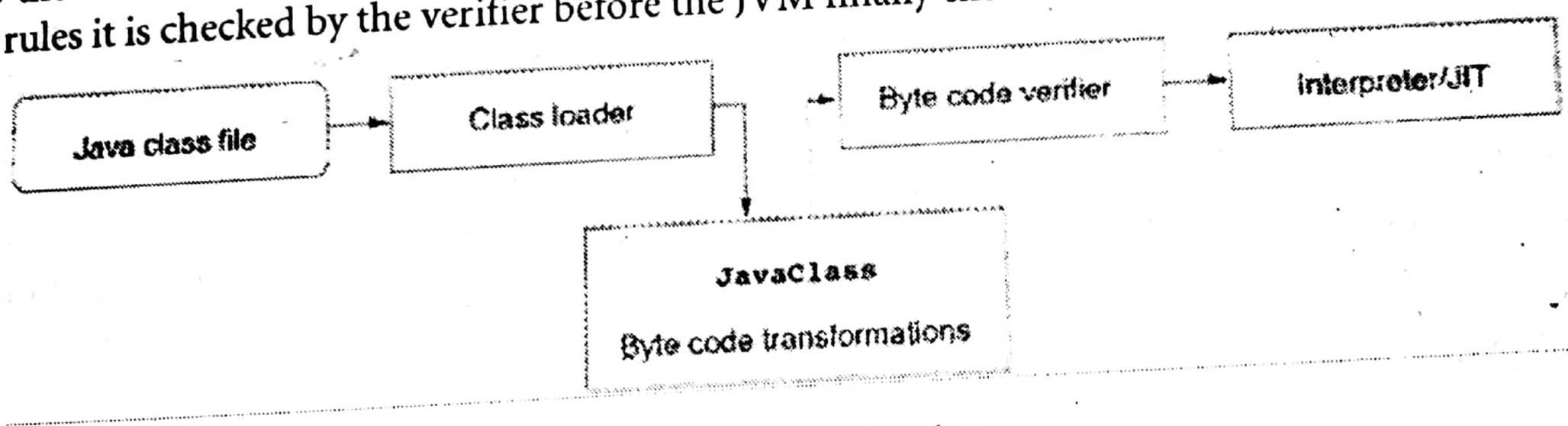

Fig. 2.17. Class loaders

Figure 2.18 shows a block diagram of the Java virtual machine that includes the major subsystems and memory areas described in the specification. 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.

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.

Scanned by CamScanner

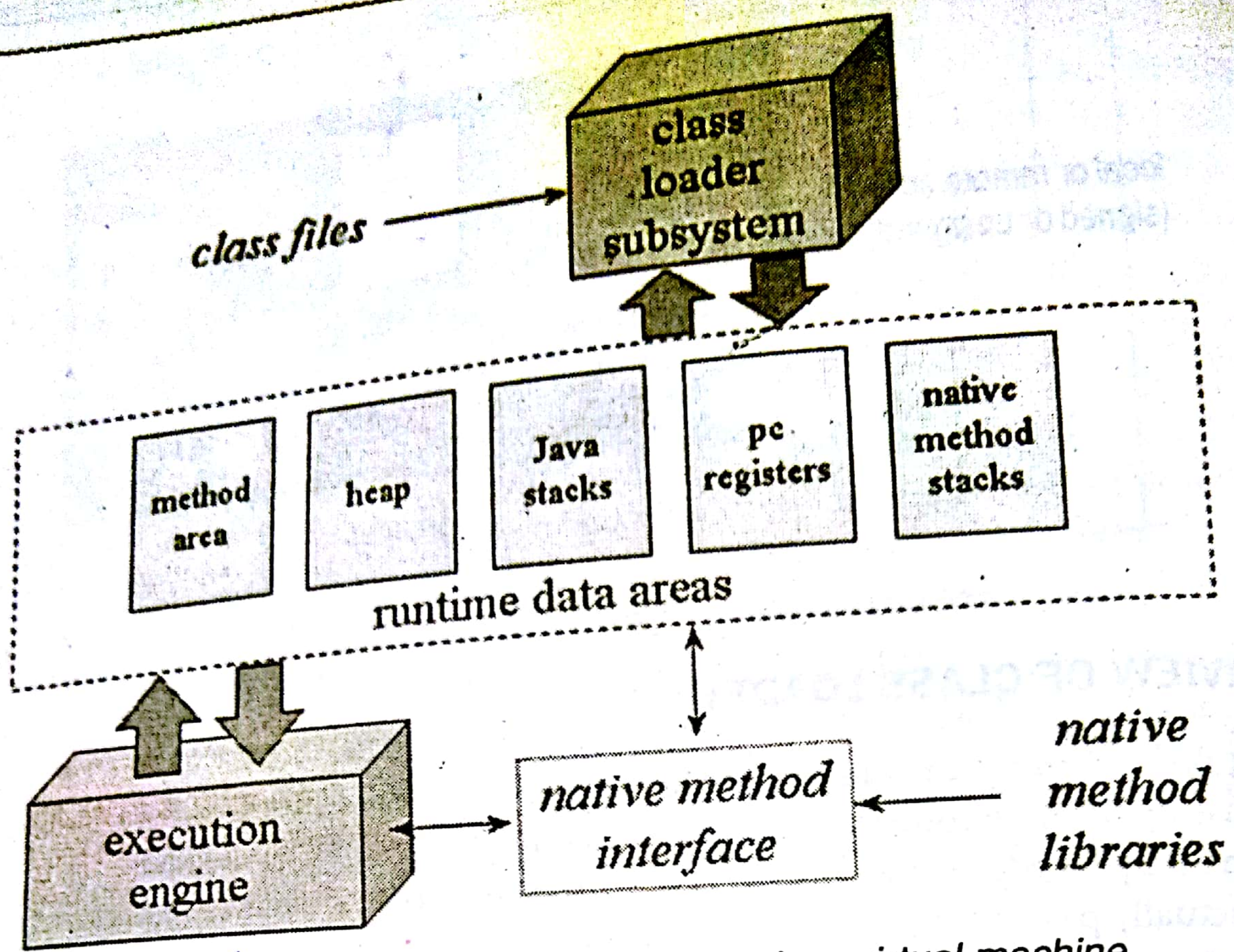

Fig. 2.18. The internal architecture of the Java virtual machine

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.

2.13.1 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 ClassLo to the internal class loader subsystem.