CHAPTER 1

Introduction to Data Structures

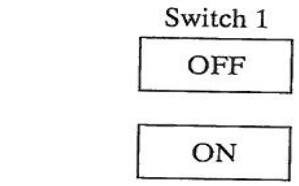
A computer is a machine that manipulates information. The study of computer science includes the study of how information is organized in a computer, how it can be manipulated, and how it can be utilized. Thus it is exceedingly important for a student of computer science to understand the concepts of information organization and manipulation.

1.1 INFORMATION AND MEANING

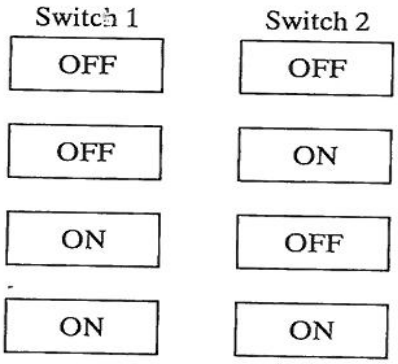
If computer science is fundamentally the study of information, the first question that arises is: what is information? Unfortunately, although the concept of information is the bedrock of the entire field, this question cannot be answered precisely. In this sense, the concept of information in computer science is similar to the concepts of point, line, and plane in geometry: they are all undefined terms about which statements can be made but which cannot be explained in terms of more elementary concepts.

In geometry, it is possible to talk about the length of a line despite the fact that the concept of a line is itself undefined. The length of a line is a measure of quantity. Similarly, in computer science, we can measure quantities of information. The basic unit of information is the ***bit***, whose value asserts one of two mutually exclusive possibilities. For example, if a light switch can be in one of two positions but not in both simultaneously, the fact that it is either in the "on" position or the "off" position is one bit of information. If a device can be in more than two possible states, then the fact that it is in a particular state is more than one bit of information. For example, if a dial has eight possible positions, then the fact that it is in position four rules out seven other possibilities, whereas the fact that a light switch is on rules out only one other possibility.

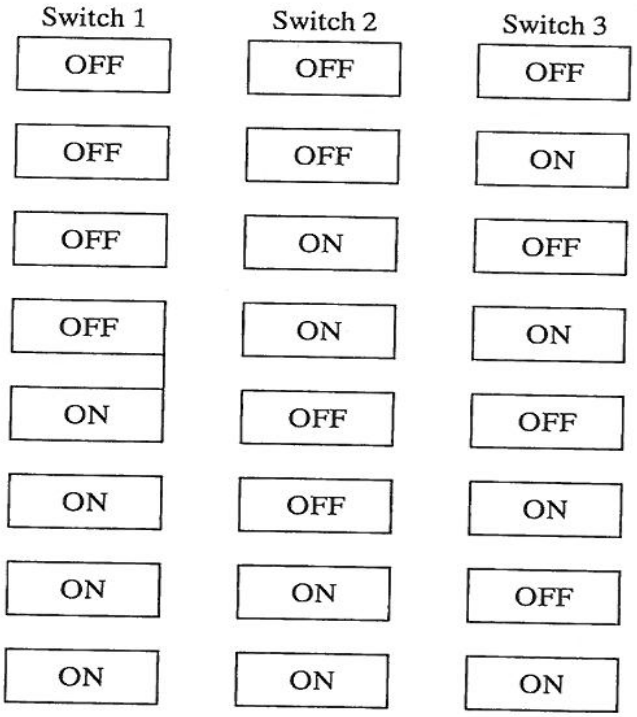
Another way of thinking of this phenomenon is as follows. Suppose we had only two-way switches, but could use as many of them as we needed. How many switches would be necessary to represent a dial with eight positions? Clearly, one switch can represent only two positions (see Figure 1.1.1a). Two switches can represent four different



(a) One switch (two possibilities).



(b) Two switches (four possibilities).



(c) Three switches (eight possibilities).

FIGURE 1.1.1

positions (Figure 1.1.1b), and three switches are required to represent eight different positions (Figure 1.1.1c). In general, n switches can represent 2^n different possibilities.

The binary digits 0 and 1 are used to represent the two possible states of any bit (in fact, the word "bit" is a contraction of the words "binary digit"). Given n bits, a string of n ones and zeros is used to represent their settings. For example, the string 101011 represents six switches, the first of which is "on" (one), the second of which is "off" (zero), the third on, the fourth off, and the fifth and sixth on.

We have seen that three bits are sufficient to represent eight possibilities. The eight possible configurations of these three bits (000, 001, 010, 011, 100, 101, 110, and 111) can be used to represent the integers 0 through 7. However, there is nothing intrinsic about these bit settings that implies that a particular setting represents a particular integer. Any assignment of integer values to bit settings is equally valid as long as no two integers are assigned to the same bit setting. Once such an assignment has been made, a particular bit setting can be unambiguously interpreted as a specific integer. Let us examine several widely used methods for interpreting bit settings as integers.

Binary and Decimal Integers

The most widely used method for interpreting bit settings as nonnegative integers is the ***binary number system***. In this system each bit position represents a power of 2. The rightmost bit position represents 2^0, which equals 1, the next position to the left represents 2^1, which is 2, the next bit position represents 2^2, which is 4, and so on. An integer is represented as a sum of powers of 2. A string of all zeros represents the number 0. If a 1 appears in a particular bit position, then the power of 2 represented by that bit position is included in the sum, but if a 0 appears, then that power of 2 is not included in the sum. For example, the group of bits 00100110 has ones in positions 1, 2, and 5 (counting from right to left with the rightmost position counted as position 0). Thus 00100110 represents the integer $2^1 + 2^2 + 2^5 = 2 + 4 + 32 = 38$. Under this interpretation, any string of bits of length n represents a unique nonnegative integer between 0 and $2^n - 1$, and any nonnegative integer between 0 and $2^n - 1$ can be represented by a unique string of bits of length n.

There are two widely used methods for representing negative binary numbers. In the first method, called ***ones complement notation***, a negative number is represented by changing each bit in its absolute value to the opposite bit setting. For example, since 00100110 represents 38, 11011001 is used to represent -38. This means that the leftmost bit of a number is no longer used to represent a power of 2, but is reserved for the sign of the number. A bit string starting with a 0 represents a positive number, while a bit string starting with a 1 represents a negative number. Given n bits, the range of numbers that can be represented is $-2^{(n-1)} + 1$ (a 1 followed by $n - 1$ zeros) to $2^{(n-1)} - 1$ (a 0 followed by $n - 1$ ones). Note that under this representation, there are two representations for the number 0, a "positive 0" consisting of all zeros, and a "negative 0" consisting of all ones.

The second method of representing negative binary numbers is called ***twos complement notation***. In this notation, 1 is added to the ones complement representation of a negative number. For example, since 11011001 represents -38 in ones complement notation, 11011010 is used to represent -38 in twos complement notation. Given n bits,

the range of numbers that can be represented is $-2^{(n-1)}$ (a 1 followed by $n-1$ zeros) to $2^{(n-1)}-1$ (a 0 followed by $n-1$ ones). Note that $-2^{(n-1)}$ can be represented in twos complement notation but not in ones complement notation. However, its absolute value, $2^{(n-1)}$, cannot be represented in either notation using n bits. Note also that there is only one representation for the number 0 using n bits in twos complement notation. To see this, consider 0 using eight bits: 00000000. The ones complement is 11111111, which is negative 0 in that notation. Adding one to produce the twos complement form yields 100000000, which is nine bits long. Since only eight bits are allowed, the leftmost bit (or "overflow") is discarded, leaving 00000000 as minus 0.

The binary number system is by no means the only method by which bits can be used to represent integers. For example, a string of bits may be used to represent integers in the decimal number system, as follows. Four bits can be used to represent a decimal digit between 0 and 9 in the binary notation described above. A string of bits of arbitrary length may be divided into consecutive sets of four bits where each set represents a decimal digit. The string then represents the number that is formed by those decimal digits in conventional decimal notation. For example, in this system, the bit string 00100110 is separated into two strings of four bits each: 0010 and 0110. The first of these represents the decimal digit 2, and the second represents the decimal digit 6, so that the entire string represents the integer 26. This representation is called ***binary coded decimal***.

One important feature of the binary coded decimal representation of nonnegative integers is that not all bit strings are valid representations of a decimal integer. Four bits can be used to represent one of sixteen different possibilities since there are sixteen possible states for a set of four bits. However, in the binary coded decimal integer representation, only ten of those sixteen possibilities are used. That is, codes such as 1010 and 1100, whose binary values are ten or larger, are invalid in a binary coded decimal number.

Real Numbers

The usual method used by computers to represent real numbers is ***floating-point notation***. There are many varieties of floating-point notation, and each has individual characteristics. The key concept is that a real number is represented by a number, called a ***mantissa***, times a ***base*** raised to an integer power, called an ***exponent***. The base is usually fixed, and the mantissa and exponent vary to represent different real numbers. For example, if the base is fixed at 10, the number 387.53 could be represented as 38753×10^{-2}. (Recall that 10^{-2} is .01.) The mantissa is 38753, and the exponent is -2. Other possible representations are $.38753 \times 10^{3}$ and 387.53×10^{0}. We choose the representation in which the mantissa is an integer with no trailing zeros.

In the floating-point notation that we describe (which is not necessarily implemented on any particular machine exactly as described), a real number is represented by a 32-bit string consisting of a 24-bit mantissa followed by an 8-bit exponent. The base is fixed at 10. Both the mantissa and the exponent are twos complement binary integers. For example, the 24-bit binary representation of 38753 is 000000001001011101100001, and the 8-bit twos complement binary representation of -2 is 11111110; so the representation of

387.53 is 00000000100101110110000111111110. Other real numbers and their floating point representations are:

0	00000000000000000000000000000000
100	00000000000000000000000100000010
.5	00000000000000000000010111111111
.000005	00000000000000000000010111111010
12000	00000000000000000000110000000011
−387.53	11111111011010001001111111111110
−12000	11111111111111111111010000000011

The advantage of floating-point notation is that it can be used to represent numbers with extremely large or extremely small absolute values. For example, in the notation presented above, the largest number that can be represented is $(2^{23-1}) \times 10^{127}$, which is a very large number indeed. The smallest positive number that can be represented is 10^{-128}, which is quite small. The limiting factor on the precision with which numbers can be represented on a particular machine is the number of significant binary digits in the mantissa. Not every number between the largest and the smallest can be represented. Our representation allows only twenty-three significant bits. Thus a number such as 10 million and 1, which requires twenty-four significant binary digits in the mantissa, would have to be approximated by 10 million (1×10^7), which only requires one significant digit.

Character Strings

As we all know, information is not always interpreted numerically. Items such as names, job titles, and addresses must also be represented in some fashion by a computer. To enable the representation of such nonnumeric objects, still another method of interpreting bit strings is necessary. Such information is usually represented in character string form. For example, in some computers, the eight bits 00100110 are used to represent the character '&'. A different eight-bit pattern is used to represent the character 'A', another to represent 'B', another to represent 'C', and still another for each character that has a representation in a particular machine. A Russian machine uses bit patterns to represent Russian characters, whereas an Israeli machine uses bit patterns to represent Hebrew characters.

If eight bits are used to represent a character, up to 256 different characters can be represented, because there are 256 different eight-bit patterns. If the string 11000000 is used to represent the character 'A', and 11000001 is used to represent the character 'B', then the character string "AB" would be represented by the bit string 1100000011000001. In general, a character string is represented by the concatenation of the bit strings that represent the individual characters of the string. One common 8-bit pattern is known as the extended ASCII code (American Standard Code for Information Interchange).

In order to be able to represent a greater number of characters, Java has adopted an international 16-bit-based code known as ***Unicode UTF-16***. Using sixteen bits allows for up to 2^{16}, or 65,536, characters to be represented.

As in the case of integers, there is nothing intrinsic about a particular bit string that makes it suitable for representing a specific character. The assignment of bit

strings to characters may be entirely arbitrary, but it must be adhered to consistently. It may be that some convenient rule is used in assigning bit strings to characters. For example, two bit strings may be assigned to two letters so that the one with a smaller binary value is assigned to the letter that comes earlier in the alphabet. However, such a rule is merely a convenience; it is not mandated by any intrinsic relation between characters and bit strings. In fact, computers even differ over the number of bits used to represent a character. Some computers use seven bits (and therefore allow only up to 128 possible characters), some use eight (up to 256 characters), some use ten (up to 1024 possible characters), while the Java Virtual Machine uses sixteen (up to 65,536). The number of bits necessary to represent a character in a particular computer is called the ***byte size***, and a group of bits of that number is called a ***byte***.

Note that using sixteen bits to represent a character means that 65,536 possible characters can be represented. It is not very often that one finds a computer that uses so many different characters (although it is conceivable for a computer to include upper- and lower-case letters, special characters, international alphabets, italics, boldface, and other type characters), so that many of the 16-bit codes are not used to represent characters.

Thus we see that information itself has no meaning. Any meaning can be assigned to a particular bit pattern, as long as it is done consistently. It is the interpretation of a bit pattern that gives it meaning. For example, the bit string 00100110 can be interpreted as the number 38 (binary), the number 26 (binary coded decimal), or the character '&' (ASCII). A method of interpreting a bit pattern is often called a ***data type***. We have presented several data types: binary integers, binary coded decimal non-negative integers, real numbers, and character strings. The key questions are how to determine what data types are available to interpret bit patterns and what data type to use in interpreting a particular bit pattern.

Hardware and Software

The ***memory*** (also called ***storage*** or ***core***) of a computer is simply a group of bits (switches). At any instant of the computer's operation, any particular bit in memory is either 0 or 1 (off or on). The setting of a bit is called its ***value*** or its ***contents***.

The bits in a computer memory are grouped together into larger units such as bytes. In some computers, several bytes are grouped together into units called ***words***. Each unit (byte or word, depending on the machine) is assigned an ***address***; that is, a name identifying a particular unit among all the units in memory. This address is usually numeric, so that we may speak of byte 746 or word 937. An address is often called a ***location***, and the contents of a location are the values of the bits that make up the unit at that location.

Every computer has a set of "native" data types. This means that it is constructed with a mechanism for manipulating bit patterns consistent with the objects they represent. For example, suppose a computer contains an instruction to add two binary integers and place their sum at a given location in memory for subsequent use. Then there is a mechanism built into the computer to:

1. Extract operand bit patterns from two given locations.
2. Produce a third bit pattern representing the binary integer that is the sum of the two binary integers represented by the two operands.
3. Store the resultant bit pattern at a given location.

The computer "knows" that the bit patterns at the given locations are to be interpreted as binary integers because the hardware that executes that particular instruction is designed to do so. This is akin to a light "knowing" that it is to be on when the switch is in a particular position.

If the same machine also has an instruction to add two real numbers, then there is a separate built-in mechanism to interpret operands as real numbers. Two distinct instructions are necessary for the two operations, and each instruction carries within itself an implicit identification of the types of its operands as well as their explicit locations. Therefore, it is the programmer's responsibility to know which data type is contained in each location that is used. It is the programmer's responsibility to choose between using an integer or real addition instruction to obtain the sum of two numbers.

A high-level programming language aids in this task considerably. For example, if a Java programmer declares

```
int x, y;
float a, b;
```

space is reserved at four locations for four different numbers. These four locations may be referenced by the ***identifiers*** x, y, a, and b. An identifier is used instead of a numerical address to refer to a particular memory location because of its convenience for the programmer. The contents of the locations reserved for x and y will be interpreted as integers, while the contents of a and b will be interpreted as floating point numbers. The compiler that is responsible for translating Java programs into machine language will translate the "+" in the statement

```
x = x + y;
```

into integer addition, and will translate the "+" in the statement

```
a = a + b;
```

into floating point addition. An operator such as "+" is really a ***generic*** operator because it has several different meanings depending on its context. The compiler relieves the programmer of specifying the type of addition that must be performed by examining the context and using the appropriate version.

It is important to recognize the key role played by declarations in a high-level language. It is by means of declarations that the programmer specifies how the contents of the computer memory are to be interpreted by the program. In doing this, a declaration specifies how much memory is needed for a particular entity, how the contents of that memory are to be interpreted, and other vital details. Declarations also specify to the compiler exactly what is meant by the operation symbols that are subsequently used.

Concept of Implementation

Thus far, we have been viewing data types as a method of interpreting the memory contents of a computer. The set of native data types that a particular computer can support is determined by what functions have been wired into its hardware. However, we can view the concept of "data type" from a completely different perspective; not in terms of what a computer can do, but in terms of what the user wants done. For example,

if one wishes to obtain the sum of two integers, one does not care very much about the detailed mechanism by which that sum will be obtained. One is interested in manipulating the mathematical concept of an "integer," not in manipulating hardware bits. The hardware of the computer may be used to represent an integer and is useful only insofar as the representation is successful.

Once the concept of "data type" is divorced from the hardware capabilities of the computer, a limitless number of data types can be considered. A data type is an abstract concept defined by a set of logical properties. Once an abstract data type is defined and the legal operations involving it are specified, we may ***implement*** that data type (or a close approximation to it). An implementation may be a ***hardware implementation*** in which the circuitry necessary to perform the required operations is designed and constructed as part of a computer. Or it may be a ***software implementation*** in which a program consisting of already existing hardware instructions is written to interpret bit strings in the desired fashion and to perform the required operations. Thus, a software implementation includes a specification of how an object of the new data type is represented by objects of previously existing data types, as well as a specification of how such an object is manipulated in conformance with the operations defined for it.

Often a particular machine may be entirely simulated through software. A computer that is completely simulated by software, known as a ***virtual machine***, may be implemented on any number of actual hardware platforms. Programs that are written for the virtual machine will run on any computer for which the virtual machine has been implemented. This is the approach taken by Java, which contributes to the high degree of portability of Java applications. Throughout the remainder of this text, the term "implementation" is used to mean "software implementation."

Example

We illustrate these concepts with an example. Suppose the hardware of a computer contains an instruction

```
MOVE (source, dest, length)
```

that copies a character string of *length* bytes from an address specified by *source* to an address specified by *dest*. (We present hardware instructions and locations using uppercase letters. The length must be specified by an integer. *source* and *dest* can be specified by identifiers that represent storage locations.) An example of this instruction is MOVE(a, b, 3), which copies the three bytes starting at location a to the three bytes starting at location b.

Note the different roles played by the identifiers a and b in this operation. The first operand of the MOVE instruction is the contents of the location specified by the identifier a. The second operand, however, is not the contents of location b, since they are irrelevant to the execution of the instruction. Rather, the location itself is the operand, since the location specifies the destination of the character string. Although an identifier always stands for a location, it is common for an identifier to be used to reference the contents of that location. It is always apparent from the context whether an identifier is referencing a location or its contents. The identifier appearing as the first operand of a MOVE instruction refers to the contents of memory, while the identifier appearing as the second operand refers to a location.

We also assume that the computer hardware contains the usual arithmetic and branching instructions, which we indicate by using Java-like notation. For example, the instruction

```
z = x + y;
```

interprets the contents of the bytes at locations x and y as binary integers, adds them, and inserts the binary representation of their sum into the byte at location z. (We do not operate on integers greater than one byte in length and ignore the possibility of overflow.) Here again, x and y are used to reference memory contents, while z is used to reference a memory location, but the proper interpretation is clear from the context.

Sometimes, it is desirable to add a quantity to an address to obtain another address. For example, if a is a location in memory, we might want to reference the location 4 bytes beyond a. We cannot refer to this location as $a + 4$ because that notation is reserved for the integer contents of location a plus 4. We therefore introduce the notation $a[4]$ to refer to this location. We also introduce the notation $a[x]$ to refer to the address given by adding the binary integer contents of the byte at x to the address a.

The MOVE instruction requires the programmer to specify the length of the string to be copied. Thus, its operand is a fixed-length character string (i.e., the length of the string must be known). A fixed-length string and a byte-sized binary integer may be considered native data types of this particular machine.

Suppose we wished to implement varying-length character strings on this machine. That is, we want to enable programmers to use the instruction

```
MOVEVAR(source, dest)
```

to move a character string from location *source* to location *dest* without being required to specify any length.

To implement this new data type, we must first decide on how it is to be represented in the memory of the machine and then indicate how that representation is to be manipulated. Clearly, it is necessary to know how many bytes must be moved in order to execute the instruction. Since the MOVEVAR operation does not specify the number, it must be contained within the representation of the character string itself. A varying-length character string of length l may be represented by a contiguous set of $l + 1$ bytes ($l < 65{,}536$). The first byte contains the binary representation of the length l, and the remaining bytes contain the representations of the characters in the string. Representations of three such strings are illustrated in Figure 1.1.2 (Note that the digits 5 and 9 in these figures do not stand for the bit patterns representing the characters '5' and '9' but for the patterns 00000000 00000101 and 00000000 00001001 [assuming sixteen bits per character], which represent the integers five and nine. Similarly, 14 in Figure 1.1.2c stands for the bit pattern 00000000 00001110. Note also that this representation may be different from the way character strings are actually implemented in Java.)

The program to implement the MOVEVAR operation can be written as follows (i is an auxiliary memory location):

```
MOVE(source, dest, 1);
for (i = 1; i < dest; i++)
MOVE(source[i], dest[i], 1);
```

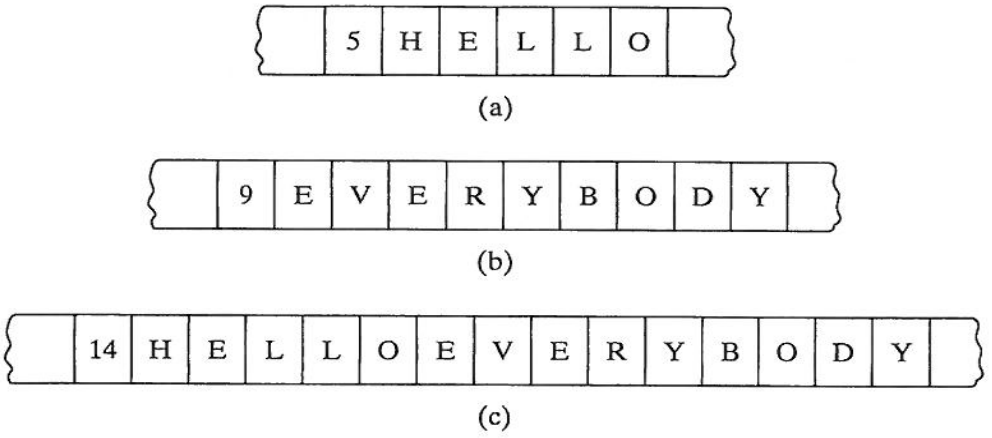


FIGURE 1.1.2 Varying-length character strings.

Similarly, we can implement an operation CONCATVAR($c1$, $c2$, $c3$) to concatenate two varying-length character strings at locations $c1$ and $c2$ and place the result at $c3$. Figure 1.1.2c illustrates the concatenation of the two strings in Figures 1.1.2a and b:

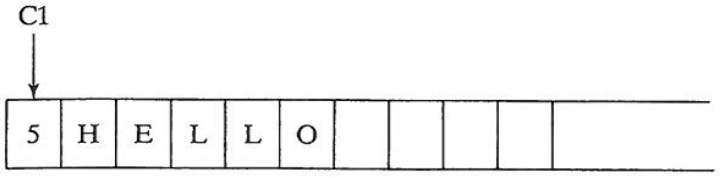
```
// move the length
z = c1 + c2;
MOVE(z, c3, 1);
// move the first string
for (i = 1; i <= c1) MOVE(c1[i], c3[i], 1)
  ;
// move the second string
for (i = 1; i <= c2) {
  x = c1 + i;
  MOVE(c2[i], c3[x], 1);
}
```

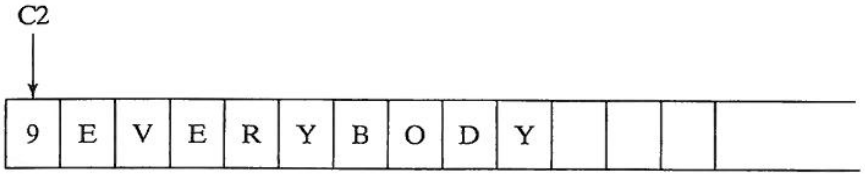
However, once the operation MOVEVAR has been defined, CONCATVAR can be implemented using MOVEVAR as follows:

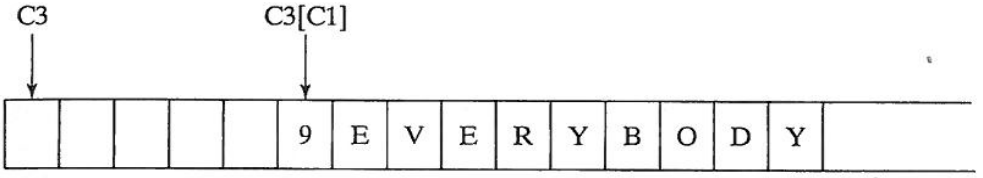
```
MOVEVAR(c2, c3[c1]);          // move the second string
MOVEVAR(c1, c3);              // move the first string
z = c1 + c2;                  // update the length of the result
MOVE(z, c3, 1);
```

Figure 1.1.3 illustrates phases of this operation on the strings of Figure 1.1.2. Although this latter version is shorter, it is not really more efficient, since all the instructions used in implementing MOVEVAR are performed each time it is used.

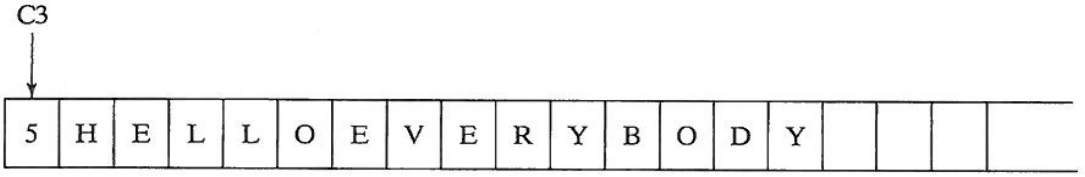
The statement $z = c1 + c2$ in both of the above algorithms is of particular interest. The addition instruction operates independently of the use of its operands (in this case, parts of varying-length character strings). The instruction is designed to treat its operands as single-byte integers regardless of any other use that the programmer has



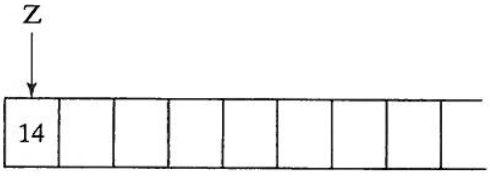


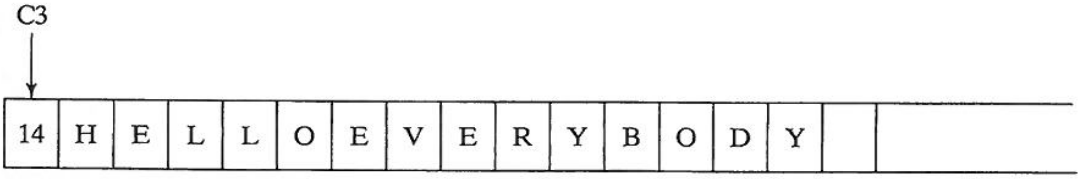


(a) MOVEVAR (C2, C3[C1]);



(b) MOVEVAR (C1, C3);





(c) Z = C1 + C2; MOVE (Z, C3, 1);

FIGURE 1.1.3 CONCATVAR operations.

for them. Similarly, the reference to $c3[c1]$ is to the location whose address is given by adding the contents of the byte at location $c1$ to the address $c3$. Thus the byte at $c1$ is treated as holding a binary integer, although it is also the start of a varying-length character string. This illustrates the fact that a data type is a method of treating the contents of memory and that those contents have no intrinsic meaning.

Note that this representation of varying-length character strings allows only strings whose length is less than or equal to the largest binary integer that fits into a single byte. If a byte is sixteen bits, this means that the largest such string is 65,535 (i.e., $2^{16} - 1$) characters long. To allow for longer strings, a different representation must be chosen and a new set of programs must be written. If we use this representation of varying-length character strings, then the concatenation operation is invalid if the resulting string is more than 65,535 characters long. Since the result of such an operation is undefined, a wide variety of actions can be implemented if that operation is attempted. One possibility is to use only the first 65,535 characters of the result. Another possibility is to ignore the operation entirely and not move anything to the result field. There is also the choice of printing a warning message or of assuming that the user wants to achieve whatever result the implementer decides on.

As was pointed out earlier, there may be several different ways of implementing an abstract data type. For example, the C language uses an entirely different implementation of character strings that avoids this limitation on the length of the string. In C, all strings are terminated by the special character '\0'. This character, which never appears within a string, is automatically placed by the C compiler at the end of every string. Since the length of the string is not known in advance, all string operations must proceed one character at a time until '\0' is encountered.

The algorithm to implement the MOVEVAR operation, under this implementation can be written as follows:

```
i = 0;
while (source[i] != '\0') {
    MOVE(source[i], dest[i], 1);
    i++;
}
dest[i] = '\0';      //terminate the destination string with '\0'
```

To implement the concatenation operation, CONCATVAR($c1, c2, c3$) we may write

```
i = 0;
// move the first string
while (c1[i] != '\0') {
    MOVE(c1[i], c3[i], 1);
    i++;
}
// move the second string
j = 0;
while (c2[j] != '\0')
    move(c2[j++], c3[i++], 1);
// terminate the destination string with a '\0'
c3[i] = '\0';
```

A disadvantage of the C implementation of character strings is that the length of a character string is not readily available without advancing through the string one character at a time until '\0' is encountered. This is more than offset by the advantage of not having an arbitrary limit placed on the length of the string.

It is important to note that implementation details are often hidden from the user. This is the approach used by the Java language. The implementer of the Java *String* class (or any other class, for that matter) need only publish the behavior of the class via its set of supported methods—the internal implementation is of no concern or use to the user of the class. Furthermore, this approach to language design leaves it open to the class implementer to select the exact representation to be used. One Java implementation could use null-terminated character arrays, a second could use a length field, and a third could use some other representation entirely.

Once a representation has been chosen for objects of a particular data type, and routines have been written to operate on those representations, the programmer is free to use that data type to solve problems. The original hardware of the machine plus the programs for implementing more complex data types than those provided by the hardware can be thought of as a "better" machine than the one consisting of the hardware alone. The programmer of the original machine need not worry about how the computer is designed and what circuitry is used to execute each instruction. The programmer need know only what instructions are available and how they can be used. Similarly, the programmer who uses the "extended" machine (which consists of hardware and software), or "virtual computer," as it is sometimes known, need not be concerned with the details of how various data types are implemented. All the programmer needs to know is how they can be manipulated.

Abstract Data Types

A useful tool for specifying the logical properties of a data type is the ***abstract data type***, or ***ADT***. Fundamentally, a data type is a collection of values and a set of operations on those values. The collection and the operations form a mathematical construct that may be implemented using a particular hardware or software data structure. The term "abstract data type" refers to the basic mathematical concept that defines the data type.

In defining an abstract data type as a mathematical concept, we are not concerned with space or time efficiency. Those are implementation issues. In fact, the definition of an ADT is not concerned with implementation details at all. It may not even be possible to implement a particular ADT on a particular piece of hardware or using a particular software system. For example, we have already seen that the ADT *integer* is not implementable in full generality. Nevertheless, by specifying the mathematical and logical properties of a data type or structure, the ADT is a useful guideline to implementers and a useful tool to programmers who wish to use the data type correctly.

There are a number of methods for specifying an ADT. The method we use is semiformal and borrows heavily from Java notation but extends it where necessary. To illustrate the concept of an ADT and our specification method, consider the ADT *RATIONAL*, which corresponds to the mathematical concept of a rational number. The operations we define on rational numbers are creation of a rational number from

two integers, addition, multiplication, and testing for equality. The following is an initial specification of this ADT:

```
abstract class RATIONAL <integer, integer> {
  // value definition
  condition RATIONAL[1] != 0;

  // method definition
abstract RATIONAL makeRational(int a, int b)
precondition    b != 0;
postcondition   makeRational[0] == a;
                makeRational[1] == b;

abstract RATIONAL add(RATIONAL a, RATIONAL b)    // written a + b
postcondition   add[1] == a[1] * b[1];
                add[0] == a[0] * b[1] + b[0] * a[1];

abstract RATIONAL mult(RATIONAL a, RATIONAL b)   // written a * b
postcondition   mult[0] == a[0] * b[0];
                mult[1] == a[1] * b[1];

abstract boolean equals(RATIONAL a, RATIONAL b) // written a == b
postcondition equals == (a[0]*b[1] == b[0]*a[1]);
}
```

An ADT consists of two parts: a value definition and a method definition. The value definition defines the collection of values for the ADT and consists of two parts: a definition clause and a condition clause. For example, the value definition for the ADT *RATIONAL* states that a *RATIONAL* value consists of two integers, where the second does not equal zero. Of course, the two integers that comprise a rational number are the numerator and the denominator. We use array notation (square brackets) to indicate the parts of an abstract type.

The keywords ***abstract class*** introduce a value definition, and the keyword ***condition*** is used to specify any conditions on the newly defined type. In this definition, the condition specifies that the denominator may not be zero. The definition clause is required, but the condition clause may not be necessary for every ADT.

Immediately following the value definition comes the method specification. Each method is defined as an abstract function with three parts: a header, the optional preconditions, and the postconditions. For example, the method definition of the ADT *RATIONAL* includes the methods of creation (*makeRational*), addition (*add*), and multiplication (*mult*), as well as a test for equality (*equal*). Let us consider the specification for multiplication first, since it is the simplest. It contains a header and postconditions, but no preconditions.

```
abstract RATIONAL mult(RATIONAL a, RATIONAL b)    // written a * b
postcondition   mult[0] == a[0] * b[0];
                mult[1] == a[1] * b[1];
```

The header of this definition is the first line, which is just like a Java method header. The keyword ***abstract*** indicates that this is not a Java method but an ADT method definition.

The comment beginning with the new keyword ***written*** indicates an alternative way of writing the method.

The postcondition specifies what the method does. In a postcondition, the name of the method (in this case, *mult*) is used to denote the result of the operation. Thus, *mult*[0] represents the numerator of the result, and *mult*[1] the denominator of the result. That is, it specifies what conditions become true after the operation is executed. In this example, the postcondition specifies that the numerator of the result of a rational multiplication equals the integer product of the numerators of the two inputs, and that the denominator equals the integer products of the two denominators.

The specification for addition (ADD) is straightforward and simply states that

$$\frac{a0}{a1} + \frac{b0}{b1} = \frac{a0 \times b1 + a1 \times b0}{a1 \times b1}$$

The creation operation (*makeRational*) creates a rational number from two integers and contains the first example of a precondition. In general, preconditions specify any restrictions that must be satisfied before the operation can be applied. In this example, the precondition states that *makeRational* cannot be applied if its second parameter is 0.

The specification for equality (*equal*) is more significant and conceptually more complex. In general, any two values in an ADT are "equal" if and only if the values of their components are equal. Indeed, it is usually assumed that an equality (and inequality) method exists and is defined that way, so that no explicit equal method definition is required. The assignment method (setting the value of one object to the value of another) is another example of a method that is often assumed for an ADT and is not specified explicitly.

However, for some data types, two values with unequal components may be considered equal. Indeed, such is the case with rational numbers, where, for example, the rational numbers 1/2, 2/4, 3/6, and 18/36 are all equal despite the inequality of their components. Two rational numbers are considered equal if their components are equal when the numbers are reduced to lowest terms (i.e., when their numerators and denominators are both divided by their greatest common divisor). One way of testing for rational equality is to reduce the two numbers to lowest terms and then test for equality of numerators and denominators. Another way of testing for rational equality is to check whether the cross-products (i.e., the numerator of one times the denominator of the other) are equal. This is the method that we used in specifying the abstract *equal* method.

The abstract specification illustrates the role of an ADT as a purely logical definition of a new data type. As collections of two integers, two ordered pairs are unequal if their components are not equal; yet as rational numbers, they may be equal. It is unlikely that any implementation of rational numbers would implement a test for equality by actually forming the cross-products; they might be too large to represent as machine integers. Most likely, an implementation would first reduce the inputs to lowest terms and then test for component equality. Indeed, a reasonable implementation would insist that *makeRational*, *add*, and *mult* only produce rational numbers in lowest terms. However, mathematical definitions such as abstract data type specifications need not be concerned with implementation details.

In fact, the realization that two rationals can be equal even if they are componentwise unequal forces us to rewrite the postconditions for *makeRational*, *add*, and *mult*. That is, if

$$\frac{m0}{m1} == \frac{a0}{a1} \times \frac{b0}{b1}$$

it is not necessary that $m0$ equal $a0 * b0$ and that $m1$ equal $a1 * b1$, only that $m0 * a1 * b1$ equal $m1 * a0 * b0$. A more accurate ADT specification for *RATIONAL* is the following:

```
abstract class RATIONAL <integer, integer> {
  // value definition
  condition RATIONAL[1] != 0;

  // method definition
  abstract boolean equals(RATIONAL a, RATIONAL b) // written a == b
  postcondition equals == (a[0]*b[1] == b[0]*a[1]);

  abstract RATIONAL makeRational(int a, int b)    // written [a, b]
  precondition    b != 0;
  postcondition   makeRational[0]*b == a*makeRational[1];

  abstract RATIONAL add(RATIONAL a, RATIONAL b)   // written a + b
  postcondition   add == [a[0]*b[1] + b[0]*a[1], a[1]*b[1]];

  abstract RATIONAL mult(RATIONAL a, RATIONAL b)  // written a * b
  postcondition   mult == [a[0]*b[0], a[1]*b[1]];
}
```

Here, the *equals* method is defined first and the method == is extended to rational equality using the ***written*** clause. That operator is then used to specify the results of subsequent rational methods (*add* and *mult*).

The result of the *makeRational* operation on the integers a and b produces a rational that equals a/b, but the definition does not specify the actual values of the resulting numerator and denominator. The specification for *makeRational* also introduces the notation $[a, b]$ for the rational formed from integers a and b, and this notation is then used in defining *add* and *mult*.

The definitions of *add* and *mult* specify that their results equal the unreduced results of the corresponding operation, but the individual components are not necessarily equal.

Note, again, that in defining these methods we are not specifying how they are to be computed, only what their result must be. How they are computed is determined by their implementation, not by their specification.

Sequences as Value Definitions

In developing the specifications for various data types, we often use set-theoretic notation to specify the values of an ADT. In particular, it is helpful to use the notation of mathematical sequences that we now introduce.

A ***sequence*** is simply an ordered set of elements. A sequence S is sometimes written as the enumeration of its elements, such as

$$S = < s_0, s_1, \ldots, s_{n-1} >$$

If S contains n elements, then S is said to be of length n. We assume the existence of a length method *length* such that *S.length*() is the length of the sequence S. We also assume the methods *S.first*(), which returns the value of the first element of S (s_0 in the example above), and *S.last*(), which returns the value of the last element of S (s_{n-1} in the example above). There is a special sequence of length zero, called *nullSeq*, that contains no elements. *nullSeq.first*() and *nullSeq.last*() are undefined.

We wish to define an ADT *sequence*1 whose values are sequences of elements. If the sequences can be of arbitrary length and consist of elements all of which are of the same type, *type*, then *sequence*1 can be defined by

```
abstract class <<type> sequence1 { ... }
```

Alternatively, we may wish to define an ADT *sequence*2 whose values are sequences of fixed length whose elements are of specific types. In such a case, we would specify the definition

```
abstract class <<type0, type1, type2, ..., typen> sequence2 { ... }
```

Of course, we may want to specify a sequence of fixed length, all of whose elements are of the same type. We could then write

```
abstract class <<type, n> sequence3 { ... }
```

In this case *sequence*3 represents a sequence of length n, all of whose elements are of type *type*.

For example, using the foregoing notation we could define the following types:

```
abstract class <<int> intSeq { ... }
                                  // sequence of integers of any length
abstract class <<int, char, float> seq3 { ... }
                                  // sequence of length 3 consisting of
                                  // an integer, a character, and a
                                  // floating-point number
abstract class <<int, 10> intSeq { ... }
                                  // sequence of 10 integers
abstract class <<, 2> pair { ... }
                                  // arbitrary sequence of length 2
```

Two sequences are *equal* if each element of the first is equal to the corresponding element of the second. A ***subsequence*** is a contiguous portion of a sequence. If S is a sequence, then the method *S.sub*(i,j) refers to the subsequence of S starting at position i in S and consisting of j consecutive elements. Thus if T equals *S.sub*(i,k), then T is the sequence $<t_0, t_1, \ldots, t_{k-1}>$, $t_0 = s_i$, $t_1 = s_{i+1}$, $\ldots$, $t_{k-1} = s_{i+k-1}$. If i is not between 0 and *S.length*() $-k$, then *S.sub*(i,k) is defined as *nullSeq*.

The concatenation of two sequences, written $S + T$, is the sequence consisting of all the elements of S followed by all the elements of T. It is sometimes desirable to specify the insertion of an element in the middle of a sequence. *S.place(i,x)* is defined as the sequence S with the element x inserted immediately following position i (or into the first element of the sequence if i is -1). All subsequent elements are shifted by one position. That is, *S.place(i,x)* equals $S.sub(0,i) + <x> + S.sub(i + 1, S.length() - i - 1)$.

Deletion of an element from a sequence can be specified in one of two ways. If x is an element of sequence S, then $S - <x>$ represents the sequence S without all occurrences of element x. The sequence *S.delete(i)* is equal to S with the element at position i deleted. *delete* can also be written in terms of other methods as $S.sub(0,i - 1) + S.sub(i + 1, S.length() - i - 1)$.

ADT for Varying-Length Character Strings

As an illustration of the use of sequence notation in defining an ADT, we develop an ADT specification for the varying-length character string. There are four basic methods (aside from equality and assignment) normally included in systems that support such strings:

length	a method that returns the current length of the string
concat	a method that returns the concatenation of its two input strings
substring	a method that returns a substring of a given string
indexOf	a method that returns the first position of one string as a substring of another.

```
abstract class STRING << char >> {
  abstract int length(STRING s)
  postcondition length == s.length();

  abstract STRING concat(STRING s1, STRING s2)
  postcondition  concat == s1 + s2;

  abstract STRING substring(STRING s1, int i, int j)
  precondition  0 <= i < s1.length();
                0 <= j <= s1.length() - i;
  postcondition substring == s1.sub(i, j);

  abstract int indexOf(STRING s1, STRING s2)
  postcondition  // lastPos = s1.length() - s2.length()
                 (indexOf == -1 && for (i = 0; i <= lastPos; i++)
                 (s2 != s1.sub(i, s2.length())))
                 ||
                (indexOf >= 0 && indexOf <= lastPos
                          && s2 == str1.sub(indexOf, s2.length())
                          && for (i = 1; i < indexOf; i++)
                                  (s2 != s1.sub(i, s2.length())))
}
```

The postcondition for *indexOf* is complex and introduces some new notation, so we review it here. First, note that the content of the initial comment has the form of a Java assignment statement. This merely indicates that we wish to define the symbol *lastPos* as representing the value of *s1.length()* − *s2.length()* for use in the postcondition to simplify the appearance of the condition. Here, *lastPos* represents the maximum possible

value of the result (i.e., the last position of $s1$ where a substring whose length equals that of $s2$ can start). *lastPos* is used twice the postcondition. The longer expression $s1.length() - s2.length$ could have been used in both cases, but we choose to use a more compact symbol (*lastPos*) for clarity.

The postcondition itself states that one of two conditions must hold. The two conditions, which are separated by the || operator, are:

1. The method's value (*indexOf*) is −1, and $s2$ does not appear as a substring of $s1$.
2. The method's value is between 0 and *lastPos*, $s2$ does appear as a substring of $s1$ beginning at the method value's position, and $s2$ does not appear as a substring of $s1$ in any earlier position.

Note the use of a pseudo-***for*** loop in a condition. The condition

```
for (i = x; i <= y; i++)
    (condition(i))
```

is true if *condition*(i) is true for all i from x to y inclusive. It is also true if $x > y$. Otherwise, the entire ***for***-condition is false.

Data Types in Java

The Java language contains eight ***primitive data types***: *boolean*, *char*, *byte*, *short*, *int*, *long*, *float*, and *double*. We have already seen how integers, floats, and characters can be implemented in hardware. A *boolean* variable occupies a single bit and may be only be assigned the values ***true*** or ***false***. As discussed earlier in this section, variables of type *char* may contain any character represented by sixteen bits (using Unicode UTF-16). Four of these types are used to represent the ***integral*** types:

Type	Size	Value
byte	8 bits	−128 to 127
short	16 bits	−32,768 to 32,767
int	32 bits	−2,147,483,648 to 2,147,483,647
long	64 bits	−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

float and *double* variables are used to represent ***floating-point*** numbers occupying thirty two and sixty four bits respectively. Several special values (and constants) are defined, including positive infinity (POSITIVE_INFINITY), negative infinity (NEGATIVE_INFINITY), and not-a-number (NaN). These may be the result of certain floating-point operations and are defined as part of the IEEE 754 floating-point standard used by the Java language. *float* variables have a precision of eight decimal digits, while *double* carries a precision of seventeen decimal digits.

Type	Size	Max Value	Min Value
float	32 bits	±3.40282347E+38	±1.40239846E−45
double	64 bits	±1.79769313486231570E+308	± 4.94065645841246544E−324

A variable declaration in Java specifies two things. First, it specifies the amount of storage that must be set aside for objects declared with that type. For example, a variable of type *int* must have enough space to hold the largest possible integer value. Second, it specifies how data represented by strings of bits are to be interpreted. The same bits at a specific storage location can be interpreted as an integer of a floating-point number, yielding two completely different numeric values.

A variable declaration specifies that storage is to be set aside for an object of the specified type, and that the object at that storage location can be referenced with the specified variable identifier.

Objects and Java

A Java programmer can think of the Java language as defining a new machine with its own capabilities, data types, and operations. The user can state a problem solution in terms of the more useful Java constructs rather than in terms of lower-level machine language constructs. Thus problems can be solved more easily because a larger set of tools is available.

Java is an ***object-oriented*** language. Recall that when we defined the ADT *RATIONAL* it was necessary to describe its behavior in terms of both of its data components and its method components. Thus we specified that a rational number consists of a numerator and denominator, as well as a series of methods, such as *add*, *mult*, and *equal*. Such a definition is known in Java as a ***class.*** A class is a collection of data, which may be of both primitive and non-primitive type, and methods that operate on the data. A particular instance of a class (i.e., data interpreted according to the definition of the class) is known as an ***object***. In Section 1.3 we define the *Rational* class and illustrate how it may be used to process rational numbers.

As a simple example of the use of a class, consider an application that processes geometric figures such as circles. In order to specify a circle, we need to specify the x and y coordinates of its center as well as its radius. Using the constant π, we may calculate its area and circumference. We can think of a circle as a collection of the data representing its center and radius, π, and methods to calculate the area and circumference. We adopt the Java convention that class names begin with a capital letter, variables begin with a lower-case letter, and constants are written with all capital letters. The *Circle* class may thus be defined by

```
public class Circle {
  float x, y;
  double radius;

  public double circumference () {
      return 2 * Math.PI * radius;
  } // end circumference method

  public double area () {
      return Math.PI * radius * radius;
  } // end area method
} // end Circle class
```

Once the *Circle* class has been defined, individual instances of the *Circle* class, known as ***objects of the class***, may be created by writing

```
Circle c = new Circle();
```

In this declaration, the variable *c* is declared to be a reference to an object of the *Circle* class. The ***new*** keyword causes a new instance of a *Circle* object to be created and a reference to it is placed in the variable *c*. A reference may be thought of as a location. That is, the contents of the location named *c* contain the location of an object whose type is *Circle*. Often variables of a class are declared, without having them refer to a specific object of the class. Thus the statement

```
Circle c1, c2;
```

declares *c*1 and *c*2 to be variables of the *Circle* class without any objects of the class being created. *c*1 and *c*2 have not been initialized with location (reference) values.

Value and Reference Semantics

An important distinction must be made between the Java primitive types described above and the ***nonprimitive*** types: *Arrays* and *Objects*. Upon assigning a value to a primitive type, a copy of the value stored in the variable appearing on the right side of the assignment operator is assigned to the variable that appears to the left of the assignment operator. Similarly, when passing a primitive argument to a parameter of a method, a copy of the original value is stored at the location represented by the parameter. This behavior is often referred to as a ***call by value***.

A nonprimitive type, however, contains a reference to the object being assigned rather than the object itself. It is therefore quite possible that two nonprimitive variables refer to the same object. Assume the following sequence of statements:

```
Circle c1, c2;          // Define c1 and c2 to be reference
                        // variables to Circle objects

c1 = new Circle();      // Assign a reference to a newly
                        // instantiated object of the Circle
                        // class to the variable c1

c1.x = 0;               // Initialize the data components of the
c1.y = 0;               // Circle object referenced by c1
c1.radius = 1;

c2 = c1;                // c2 and c1 refer to the same object

c2.radius = 10;         // The radius of the object referred to by
                        // both c1 and c2 is now 10.
```

Now compare this with the following statements involving primitive types:

```
int x, y;               // Declare x and y to be variables which
                        // can contain integer values
```

```
y = 0;                    // Place the value 0 into the storage
                          // allocated for the variable y

x = y;                    // Copy the value in the variable y into
                          // the variable x

y = 1;                    // Place the value 1 into the storage
                          // allocated for the variable y.
                          // x retains the value 0
```

Nonprimitive variables are often said to have ***reference*** behavior, and primitive values are said to exhibit ***value*** behavior.

An exception to the reference behavior of Java objects occurs when passing parameters to methods. All parameters are passed to a Java method by value. That is, the values being passed are copied into the parameters of the called method at the time the method is invoked. This is true even for reference variables. The parameter is initially a copy of the reference that refers to the same object as the argument sent to the method when it is called. When a new value is assigned to the parameter, the parameter is changed to reference the new value, but the value of the original argument reference remains unchanged. If the value of a parameter is changed within the method, the value in the calling application is not changed.

For example, consider the following application segment and method (the line numbers are for reference only):

```
1   int x = 5;
2   System.out.println("\n" + x);
3   method(x);
4   System.out.println("\n" + x);
      ...
5   void method(int y) {
6     ++y;
7     System.out.println("\n" + y);
8   } // end method
```

Line 2 prints 5 and then line 3 invokes *method*. The value of x, which is 5, is copied into y, and *method* begins execution. Line 7 then prints 6 and *method* returns. However when line 6 incremented the value of y, the value of x remained unchanged. Thus line 4 prints 5. x and y refer to two different variables that happen to have the same value at the beginning of *method*. y can change independently of x. This is true even for the case in which x and y refer to objects. In summary, parameters that are changed within a method may never modify their arguments.

Data Structures and Java

The study of data structures involves two complementary goals. The first goal is to identify and develop useful mathematical entities and operations and to determine what classes of problems can be solved by using them. The second goal is to determine representations for those abstract entities and to implement the abstract operations on

the concrete representations. The first of these goals views a high-level data type as a tool that can be used to solve other problems, while the second views the implementation of such a data type as a problem to be solved using already existing data types. In determining representations for abstract entities, we must be careful to specify what facilities are available for constructing such representations. For example, it must be stated whether the full Java language is available or we are restricted to the hardware facilities of a particular machine.

In Sections 1.2 and 1.3 we examine several data structures that already exist in Java: the array, the string, and the class. We describe the facilities that are available in Java for utilizing these structures. We also focus on the abstract definitions of these data structures and how they can be useful in problem solving. Finally, we examine how they could be implemented if Java were not available (although a Java programmer can simply use the data structures as defined in the language without being concerned about most of these implementation details).

In the remainder of the book, we develop more complex data structures and show their usefulness in problem solving. We also show how to implement these data structures using the data structures that are already available in Java. Since the problems that arise in the course of attempting to implement high-level data structures are quite complex, this will also allow us to investigate the Java language more thoroughly and to gain valuable experience in its use.

Often, no implementation, hardware or software, can model a mathematical concept completely. For example, it is impossible to represent arbitrarily large integers on a computer because the size of such a machine's memory is finite. Thus it is not the data type "integer" that is represented by the hardware but rather the data type "integer between x and y", where x and y are the smallest and largest integers representable by that machine.

It is important to recognize the limitations of a particular implementation. Often it will be possible to present several implementations of the same data type, each with its own strengths and weaknesses. One particular implementation may be better than another for a specific application, and the programmer must be aware of the possible tradeoffs that might be involved.

One important consideration in any implementation is its efficiency. In fact, the reason that the high-level data structures we discuss are not built into Java is the significant overhead they would entail. There are languages of significantly higher level than Java that have many of these data types already built into them, but many of these are inefficient and thus are not in widespread use.

Efficiency is usually measured by two factors: time and space. If an application is heavily dependent on manipulating high-level data structures, then the speed at which those manipulations can be performed will be the major determinant of the speed of the entire application. Similarly, if a program uses a large number of such structures, then an implementation that uses an inordinate amount of space to represent the data structure will be impractical. Unfortunately, there is usually a tradeoff between these two efficiencies, so that an implementation that is fast uses more storage than one that is slow. The choice of implementation in such cases involves a careful evaluation of the tradeoffs among the various possibilities.

EXERCISES

1.1.1 In the text, an analogy is made between the length of a line and the number of bits of information in a bit string. In what ways is this analogy inadequate?

1.1.2 Determine what hardware data types are available on the computer at your particular installation and what operations can be performed on them.

1.1.3 Prove that there are 2^n different settings for n two-way switches. Suppose we wanted to have m settings. How many switches would be necessary?

1.1.4 Interpret the following bit settings as binary positive integers, binary integers in twos complement, and binary coded decimal integers. If a setting cannot be interpreted as a binary coded decimal integer, explain why.

a. 10011001

b. 1001

c. 000100010001

d. 01110111

e. 01010101

f. 100000010101

1.1.5 Write Java methods *add, subtract*, and *multiply* that read two strings of 0s and 1s representing binary nonnegative integers and print the string representing their sum, difference, and product respectively.

1.1.6 Assume a ternary computer in which the basic unit of memory is a "trit" (ternary digit) rather than a bit. A trit can have three possible settings (0, 1, and 2) rather than just two (0 and 1). Show how nonnegative integers can be represented in ternary notation using trits by a method analogous to binary notation using bits. Is there any nonnegative integer that can be represented using ternary notation and trits that cannot be represented using binary notation and bits? Are there any that can be represented using bits that cannot be represented using trits? Why are binary computers more common than ternary computers?

1.1.7 Write a Java applet to read a string of 0s and 1s representing a positive integer in binary and to print a string of 0s, 1s, and 2s representing the same number in ternary notation (see the preceding exercise). Write another Java applet to read a ternary number and print the equivalent in binary.

1.1.8 Write an ADT specification for complex numbers $a + bi$, where $abs(a + bi)$ is $sqrt(a^2 + b^2)$, $(a + bi) + (c + di)$ is $(a + c) + (b + d)i$, $(a + bi)*(c + di)$ is $(a*c-b*d) + (a*d + b*c)i$, and $-(a + bi)$ is $(-a) + (-b)i$.

1.2 ARRAYS, STRINGS, AND VECTORS IN JAVA

In this section and the next, we examine several data structures that are an invaluable part of the Java language. We will see how to use these structures and how they can be implemented. These structures are ***composite*** or ***structured*** data types; that is, they are made up of simpler data structures that exist in the language. The study of these data structures involves an analysis of how simple structures combine to form the composite

and how to extract a specific component from the composite. We expect that you have already seen these data structures in an introductory Java programming course and are aware of how they are defined and used in Java. In these sections, therefore, we will not dwell on the many details associated with these structures but instead will highlight those features that are interesting from a data structure point of view.

The first of these data types is the ***array***. The simplest form of an array is a ***one-dimensional array*** that may be defined abstractly as a finite ordered set of homogeneous elements. By "finite" we mean that there is a specific number of elements in the array. This number may be large or small, but it must exist. By "ordered" we mean that the elements of the array are arranged so that there is a zeroth, first, second, third, etc. By "homogeneous" we mean that all the elements in the array must be of the same type. For example, an array may contain all integers or all characters but not both. A Java array may also contain references to Java objects.

However, specifying the form of a data structure does not completely describe the structure. We must also specify how the structure is accessed. For example, the Java declaration

```
type a[ ] = new type[100];
```

creates an array of one hundred items of type *type* and initializes each item to the default value for that type (0 for ***int***, the null character for ***char***, ***false*** for ***boolean***, 0.0 for ***float*** and ***double***, and ***null*** for objects). The variable *a* is a reference to the newly created array. Alternatively, one may declare an array by specifying a series of intial values. For example, the declaration

```
char b[ ] = {'a', 'b', 'c', 'd', 'e'};
```

creates a reference variable *b* which refers to an array of five characters initialized with the characters 'a', 'b', 'c', 'd', 'e'.

The two basic operations that access an array are ***extraction*** and ***storing***. The extraction operation is a method that accepts an array, *a*, and an index, *i*, and returns an element of the array. In Java, the result of this operation is denoted by the expression $a[i]$. The storing operation accepts an array, *a*, an index, *i*, and an element, *x*. In Java, this operation is denoted by the assignment statement $a[i] = x$. The operations are defined by the rule that after the assignment statement has been executed, the value of $a[i]$ is the value of *x*.

The smallest element of an array's index is called its ***lower bound*** and in Java is always zero, while the highest element is called its ***upper bound***. If *lower* is the lower bound of an array and *upper* the upper bound, the number of elements in the array, called its ***length*** or ***range***, is given by $upper - lower + 1$. For example, in the array *a* declared above, the lower bound is 0, the upper bound is 99, and the length is 100. The length of any array in Java may be obtained by appending *.length* to its name. For example, in the array *b* declared above, the value of *b.length* is 5.

An important feature of a Java array is that once an array has been declared, neither the upper bound nor the lower bound (and hence the range as well) may be changed during a program's execution. The lower bound is always fixed at 0, and the upper bound is fixed at the time the array is created. It is important to distinguish between an array reference and the array itself. An array, once created, is fixed in size, but

an array reference may refer to different arrays throughout an application. For example, suppose a programmer were to write

```
double x[ ];              // Creates a reference variable x, which
                          // may later be used to refer to an array
                          // containing double elements.
x = new double[10];       // Creates an array of 10 double elements
                          // referenced by the variable x.
x = new double[50];       // Creates an array of 50 double elements
                          // referenced by the variable x. The
                          // previous array of 10 double elements
                          // is no longer accessible.
```

One very useful technique is to use the *length* construct, which produces the size of an array, to minimize the work needed when an array size is changed. For example, consider the following application segment to declare and initialize an array of characters to the capital letters of the English alphabet (65 is the numeric value of the bits representing the letter 'A' in Unicode):

```
char c[ ] = new char[26];
for (int i = 0; i<26; i++)
  c[i] = (char) (65+i);
```

In order to change the array to a larger (or smaller) size, the constant 26 must be changed in two places: once in the declarations and once in the ***for*** statement. Consider the following equivalent alternative:

```
char c[ ] = new char[26];
for (int i = 0; i< c.length; i++)
  c[i] = (char) (65+i);
```

Now, only a single change in the array size is needed to change the upper bound.

The Array as an ADT

We can represent an array as an abstract data type with a slight extension of the conventions and notation discussed earlier. We assume the method *type*(*arg*), which returns the type of its argument *arg*. In fact, if *arg* is a Java object, we can get its class (which is its type) by *arg.getClass*(). However, we cannot do this for primitive variables such as integers, booleans, and so on. Since we are not concerned here with implementation, but rather with specification, the use of such a function is permissible.

Let *ARRTYPE*(*ub, elType*) denote the ADT corresponding to the Java array type *elType array*[*ub*]. This is our first example of a parameterized ADT, in which the precise ADT is determined by the values of one or more parameters. In this case, *ub* and *elType* are the parameters; note that *elType* is a type indicator, not a value. We may now view any one-dimensional array as an entity of the type *ARRTYPE*. For example,

ARRTYPE(10, *int*) would represent the type of the array in the designator *int* [10]. We may now view any one-dimensional array as an entity of the type *ARRTYPE*. The specification follows:

```
abstract class <<elType, ub>> ARRTYPE(ub, elType) {
  condition type(ub) == int;

  abstract elType extract(a, i)              // written a[i]
  ARRTYPE(ub, elType) a;
  int i;
  precondition 0 <= i < ub;
  postcondition extract == a_i

  abstract store(a, i, elt)                  // written a[i] = elt
  ARRTYPE (ub, elType) a;
  int i;
  elType elt;
  precondition 0 <= i < ub;
  postcondition a[i] == elt;
}
```

The *store* operation is our first example of an operation that modifies one of its parameters, in this case the array *a*. This is indicated in the postcondition by specifying the value of the array element to which *elt* is being assigned. The postcondition states that when the *store* operation is completed, the result of the *extract* operation applied to *a* and *i* (which is written *a*[*i*]) equals the value of *elt*. Unless a modified value is specified in a postcondition, we assume that all the parameters retain the same value after the operation is applied in a postcondition as before. It is not necessary to specify that such values remain unchanged. Thus, in this example, all the array elements other than the one to which *elt* is assigned retain the same values.

Note that once the operation *extract* has been defined together with its bracket notation *a*[*i*], that notation can be used in the postcondition for the subsequent *store* operation specification. Within the postcondition of *extract*, however, subscripted sequence notation must be used because the array bracket notation itself is being defined.

Using One-Dimensional Arrays

A one-dimensional array is used when it is necessary to keep a large number of items in memory and reference them in a uniform manner. Let us see how these two requirements apply to practical situations.

Suppose we wish to generate one hundred random integers between 1 and 100, find their average, and determine by how much each integer deviates from that average. The following application accomplishes this:

```
public class Average {
  public static void main(String[ ] args) {
      int num[ ] = new int[100];      // array of numbers
      int total;                      // sum of the numbers
      int avg;                        // average of the numbers
      int diff;                       // difference between each
                                      // number and the average
```

```
        total = 0;
        for (int i = 0; i < num.length; i++) {
          // Generate random numbers from 1 to 100
          // place the numbers into the array, and add them
          num[i] = 1 + (int) Math.round((99.0 * Math.random()));
          total += num[i];
        }
        avg = total / num.length;                    // compute the average
        System.out.println("number difference"); // print heading

        // print each number and its difference
        for (int i = 0; i < num.length; i++) {
            diff = num[i] - avg;
            System.out.println("" + num[i] + " " + diff + "\n");
        }
    System.out.println("average is: " + avg);
    } // end main method
} // end Average class
```

This application uses two groups of one hundred numbers. The first group is the set of random integers and is represented by the array *num*, and the second group is the set of differences that are the successive values assigned to the variable *diff* in the second loop. Here a question arises: Why is an array used to hold all the values of the first group simultaneously, but only a single variable to hold one value at a time of the second group?

The answer is quite simple. Each difference is computed and printed and is never needed again. Thus the variable *diff* can be reused for the difference of the next integer and the average. However, the original integers that are the values of the array *num* must all be kept in memory. Although each can be added into *total* as it is input, it must be retained until after the average is computed in order for the program to compute the difference between it and the average. Therefore, an array is used.

Of course, one hundred separate variables could have been used to hold the integers. The advantage of an array, however, is that it allows the programmer to declare only a single identifier and yet obtain a large amount of space. Furthermore, in conjunction with the ***for*** loop, it also allows the programmer to reference each element of the group in a uniform manner instead of having to code a series of statements such as

```
num0  = 1 + (int) (99.0 * Math.random());
num1  = 1 + (int) (99.0 * Math.random());
num2  = 1 + (int) (99.0 * Math.random());
        ...
num99 = 1 + (int) (99.0 * Math.random());
```

A particular element of an array may be retrieved through its index. For example, suppose a company is using an applet where an array is declared by

```
int sales[ ] = new int[10];
```

The array will hold sales figures for a ten-year period. Suppose that two input fields, *input*1 and *input*2, are placed on the applet: the first contains an integer from 0 to 9 representing a year, and the second contains the sales figure for that year. It is desired to

read the sales figure into the appropriate element of the array. This can be accomplished by executing the statements

```
yr = Integer.parseInt(input1.getText());
sales[yr] = Integer.parseInt(input2.getText());
```

within the action method of the applet. In this statement, a particular element of the array is accessed directly by using its index. Consider the situation if ten variables $s0, s1, \ldots, s9$ had been declared. Then, even after executing *yr = Integer.parseInt(input1.getText())* to set *yr* to the integer representing the year, the sales figure could not be read into the proper variable without coding something like:

```
switch (yr) {
  case 0:        s0 = Integer.parseInt(input2.getText());   break;
  case 1:        s1 = Integer.parseInt(input2.getText());   break;
    .
    .
    .
  case 9:        s9 = Integer.parseInt(input2.getText());   break;
}
```

This is bad enough with ten elements—imagine the inconvenience if there were a hundred or a thousand.

Implementing One-Dimensional Arrays

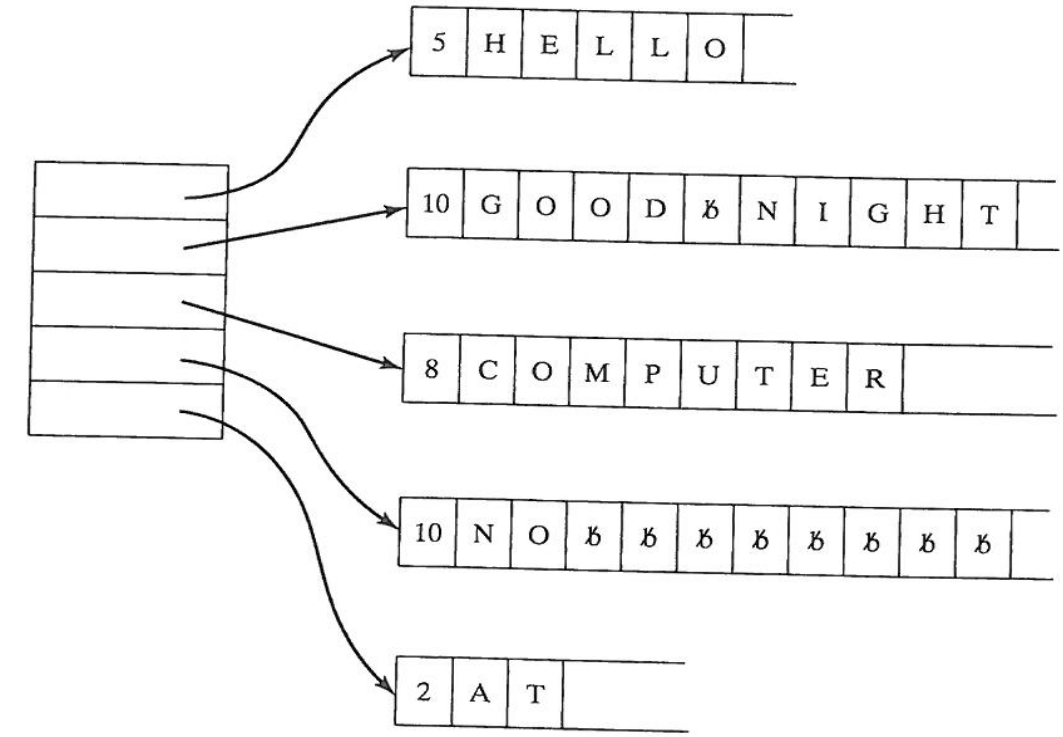
A one-dimensional array can be implemented easily. The Java declaration

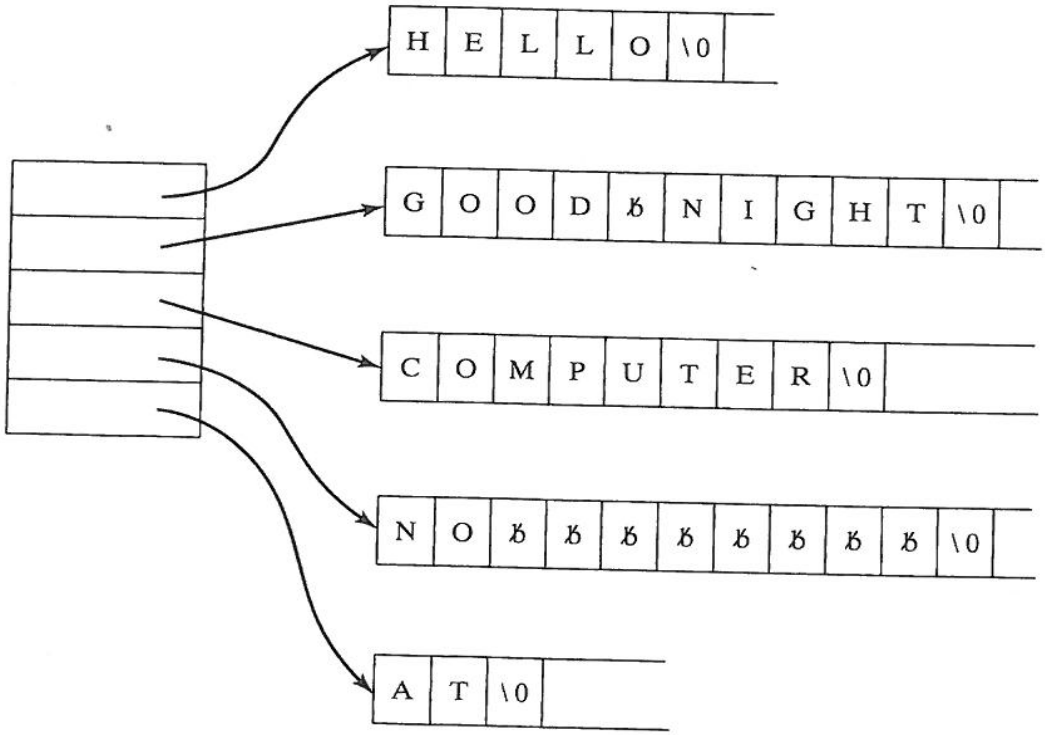
```
int b[ ] = new int[100];
```

reserves one hundred successive memory locations, each large enough to contain a single integer. We call the address of the first of these locations the ***base address*** of the array *b*, and we denote it by *base(b)*. Suppose that the size of each individual element of the array is *esize*. Then a reference to the element *b*[0] is to the element at location *base(b)*, a reference to *b*[1] is to the element at *base(b) + esize*, a reference to *b*[2] is to the element *base(b) + 2*esize*. In general, a reference to *b*[*i*] is to the element at location *base(b) + i*esize*. Thus it is possible to reference any element in the array, given its index.

In Java, all elements of any particular array have the same fixed, predetermined size. Some programming languages, however, allow arrays of objects of differing sizes. For example, a language might allow arrays of varying-length character strings. In such cases, the above method cannot be used to implement the array. This is because this method of calculating the address of a specific element of the array depends upon knowing the fixed size *esize* of each preceding element. If not all of the elements have the same size, a different implementation must be used.

One method of implementing an array of varying-sized elements is to reserve a contiguous set of memory locations, each of which holds a reference to another location. The contents of each such memory location is the address of the varying-length array element in some other portion of memory. For example, Figure 1.2.1a illustrates an array of five varying-length character strings under the two implementations of





(a)

FIGURE 1.2.1 Implementations of an array of varying-length strings.

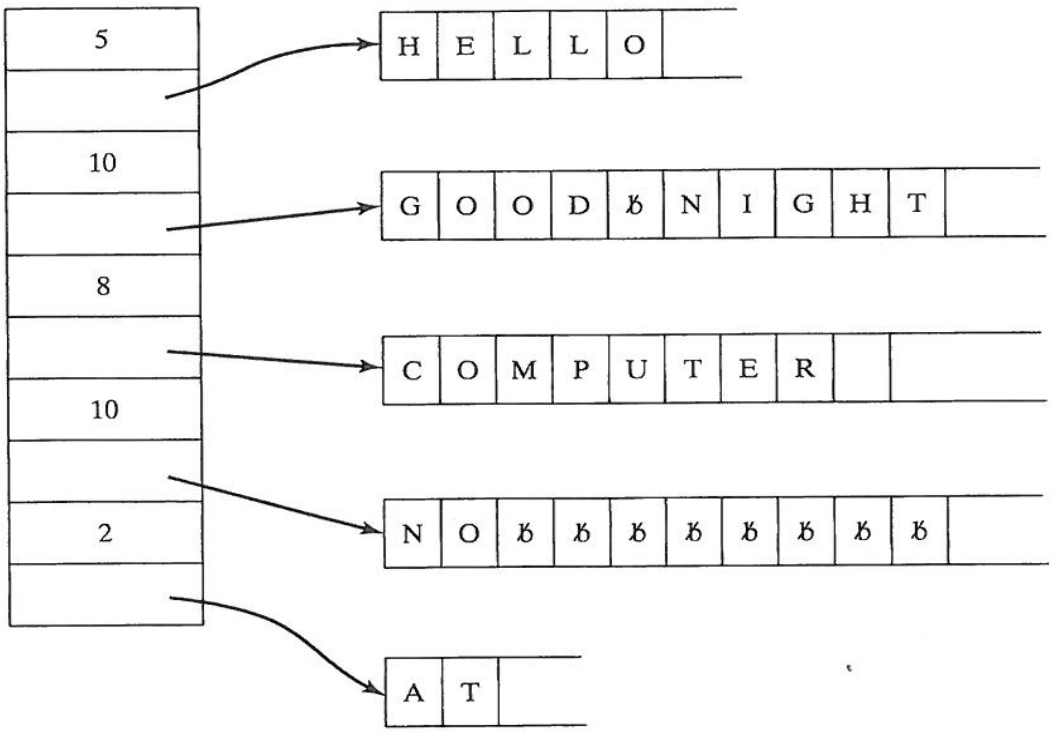


FIGURE 1.2.1 (*Continued*)

varying-length integers presented in Section 1. The arrows in the diagram indicate the addresses of other portions of memory. The character '$\flat$' indicates a blank.

Since the length of each address is fixed, the location of the address of a particular element can be computed in the same way that the location of a fixed-length element was computed in the previous examples. Once this location is known, its contents can be used to determine the location of the actual array element. This, of course, adds an extra level of indirection to referencing an array element by involving an extra memory reference, which in turn decreases efficiency. However, this is a small price to pay for the convenience of being able to maintain such an array.

A similar method for implementing an array of varying-sized elements is to keep all the fixed-length portions of the elements in the contiguous array area, in addition to keeping the address of the varying-length portion in the contiguous area. For example, in the implementation of varying-length character strings presented in the previous section, each such string contains a fixed-length portion (a 1-byte length field) and a variable-length portion (the character string itself). One implementation of an array of varying-length character strings keeps the length of the string together with the address, as shown in Figure 1.2.1b. The advantage of this method is that those parts of an element that are of fixed length can be examined without an extra memory reference. For example, a method to determine the current length of a varying-length character string can be implemented with a single memory lookup. The fixed-length information

for an array element of varying length stored in the contiguous memory area of the array is often called a ***header***.

Arrays as Parameters

Every parameter of a Java method must be declared within the method. However, the size of a one-dimensional array parameter is only specified in the main program. This is because new storage is not allocated for an array parameter in Java. Rather, the parameter refers to the original array that was allocated in the main program. For example, consider the following method to compute the average of the elements of an array:

```
public double avg(double a[ ]) {
  double sum = 0;

    for (int i = 0; i < a.length; i++)
      sum += a[i];
  return sum / a.length;
}
```

In the main application, we might have written

```
public static final int ARANGE = 100;
double a[ ] = new double[ARANGE];
    ...
average = avg(a);
```

Note that the length of the array is not passed to the method—the length of the array may always be obtained by using *length*.

Since an array variable in Java is a reference to the actual storage locations currently allocated for the array, array parameters are passed ***by reference*** rather than by value. That is, unlike simple variables that are passed by value, an array's contents are not copied when it is passed as a parameter in Java. Instead, a reference to the array is passed. If a calling method contains the call *method*(a), where a is an array and the method *method* has the header

```
public int method(int b[ ])
```

then the statement

```
b[i] = x;
```

inside *method* modifies the value of $a[i]$ inside the calling method. b inside *method* references the same array of locations as a in the calling function.

Passing an array by reference rather than by value is more efficient in both time and space. The time that would be required to copy an entire array on invoking a function is eliminated. Also, the space that would be needed for a second copy of the array in the called function is reduced to space for only a single pointer variable.

Two-Dimensional Arrays

The component type of an array can be another array. For example, we may define

```
int a[ ] = new int [3][5];
```

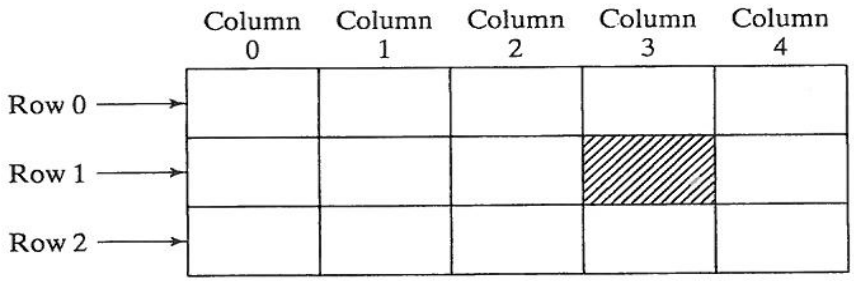


FIGURE 1.2.2 Two-dimensional arrays.

This defines a new array containing three elements. Each of these elements is itself an array containing five integers. Figure 1.2.2 illustrates such an array. An element of this array is accessed by specifying two indices: a row number and a column number. For example, the element that is darkened in Figure 1.2.2 is in row 1 and column 3 and may be referenced as $a[1][3]$. Such an array is called a ***two-dimensional*** array. The number of rows or columns is called the ***range*** of the dimension. In the array a above, the range of the first dimension is 3 and the range of the second dimension is 5. Thus the array a has three rows and five columns.

A two-dimensional array clearly illustrates the differences between a ***logical*** and a ***physical*** view of data. A two-dimensional array is a logical data structure that is useful in programming and problem solving. For example, such an array is useful in describing an object that is physically two-dimensional, such as a map or a checkerboard. It is also useful in organizing a set of values that are dependent upon two inputs. For example, a program for a department store that has twenty branches each of which sells thirty items might include a two-dimensional array declared by

```
int sales[ ][ ] = new int[20][30];
```

Each element *sales*[*i*][*j*] represents the amount of item *j* sold in branch *i*.

However, although it is convenient for the programmer to think of the elements of such an array as being organized in a two-dimensional table, and programming languages do indeed include facilities for treating them as a two-dimensional array, the hardware of most computers has no such facilities. An array must be stored in the memory of a computer, and that memory is usually linear. By this we mean that the memory of a computer is essentially a one-dimensional array. A single address (which may be viewed as a subscript of a one-dimensional array) is used to retrieve a particular item from memory. In order to implement a two-dimensional array, it is necessary to develop a method of ordering its elements in a linear fashion and of transforming a two-dimensional reference to the linear representation.

One method of representing a two-dimensional array in memory is the ***row-major*** representation. Under this representation, the first row of the array occupies the first set of memory locations reserved for the array, the second row occupies the next set, and so forth. There may also be several locations at the start of the physical array that serve as a header and contain the upper and lower bounds of the two dimensions. (This header should not be confused with the headers discussed earlier. This header is for the entire array, whereas the headers mentioned earlier are headers for the individual array

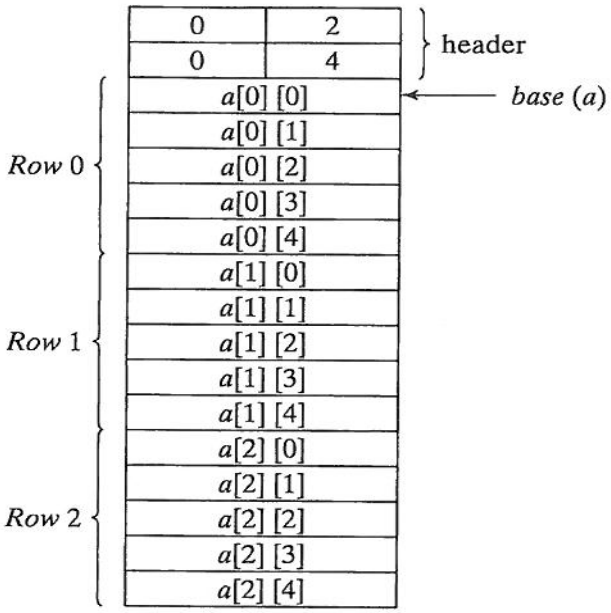


FIGURE 1.2.3 Representing a two-dimensional array.

elements.) Figure 1.2.3 illustrates the row-major representation of the two-dimensional array *a* declared above and illustrated in Figure 1.2.2. Alternatively, the header need not be contiguous to the array elements, but could instead contain the address of the first element of the array. Additionally, if the elements of the two-dimensional array are variable-length objects, the elements of the contiguous area could themselves contain the addresses of those objects in a form similar to those of Figure 1.2.1 for linear arrays.

Let us suppose that a two-dimensional integer array is stored in row-major sequence, as in Figure 1.2.3, and let us suppose that, for an array *ar*, *base*(*ar*) is the address of the first element of the array. That is, if *ar* is declared by

```
int ar[ ][ ] = new int[r1][r2];
```

where *r*1 and *r*2 are the ranges of the first and second dimensions respectively, then *base*(*ar*) is the address of *ar*[0][0]. We also assume that *esize* is the size of each element in the array. Let us calculate the address of an arbitrary element, *ar*[*i*1][*i*2]. Since the element is in row *i*1, its address can be calculated by computing the address of the first element of row *i*1 and adding the quantity *i*2**esize* (this quantity represents how far into row *i*1 the element at column *i*2 is). But in order to reach the first element of row *i*1 (i.e., the element *ar*[*i*1][0]), it is necessary to pass through *i*1 complete rows each of which contains *r*2 elements (since there is one element from each column in each row), so that the address of the first element of row *i*1 is at *base*(*ar*) + *i*1**r*2**esize*. Therefore, the address of *ar*[*i*1][*i*2] is at

```
base(ar) + (i1 * r2 + i2)* esize
```

As an example, consider the array *a* of Figure 1.2.2 whose representation is illustrated in Figure 1.2.3. In this array, *r*1 = 3, *r*2 = 5, and *base*(*a*) is the address of *a*[0][0]. Let us also suppose that each element of the array requires a single unit of storage, so that

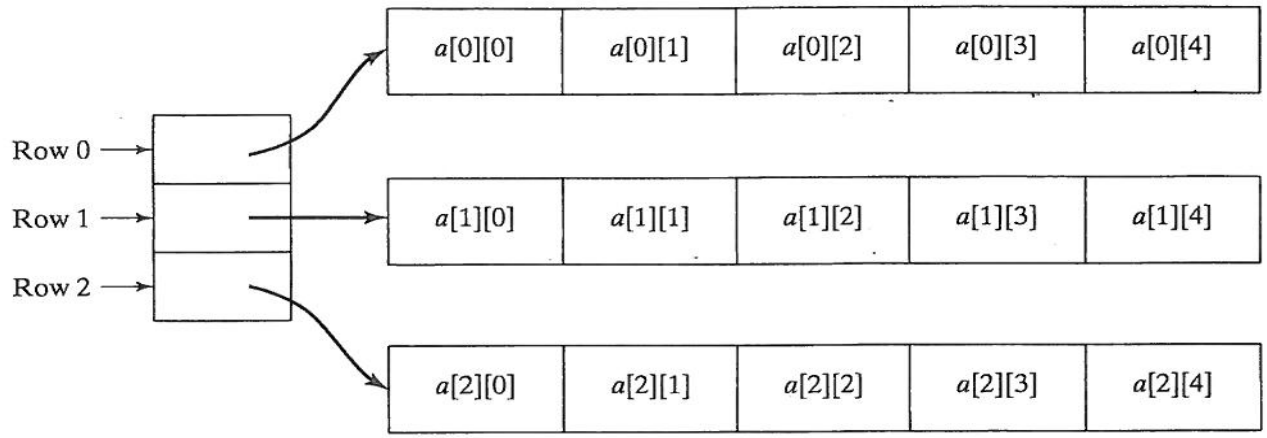


FIGURE 1.2.4 Alternative implementation of a two-dimensional array.

esize equals 1. (This is not necessarily true, because *a* was declared as an array of integers, and an integer may need more than one unit of memory on a particular machine. For simplicity, however, we accept this assumption.) Then the location of *a*[2][4] can be computed by

```
base[a] + 2*5 + 4*1
```

that is,

```
base(a) + 14
```

You may confirm the fact that *a*[2][4] is fourteen units past *base*(*a*) in Figure 1.2.3.

Another possible implementation of a two-dimensional array is as follows. An array *ar* declared with upper bounds *u*1 and *u*2 consists of *u*1 one-dimensional arrays. The first is an array *ap* of *u*1 pointers. The *i*th element of *ap* is a pointer to a one-dimensional array whose elements are the elements of the two-dimensional array *ar*. For example, Figure 1.2.4 illustrates such an implementation for the array *a* of Figure 1.2.2, where *u*1 is 3 and *u*2 is 5.

To reference *ar*[*i*][*j*], the array *ar* is first accessed to obtain the pointer *ar*[*i*]. The array at that pointer location is then accessed to obtain *a*[*i*][*j*].

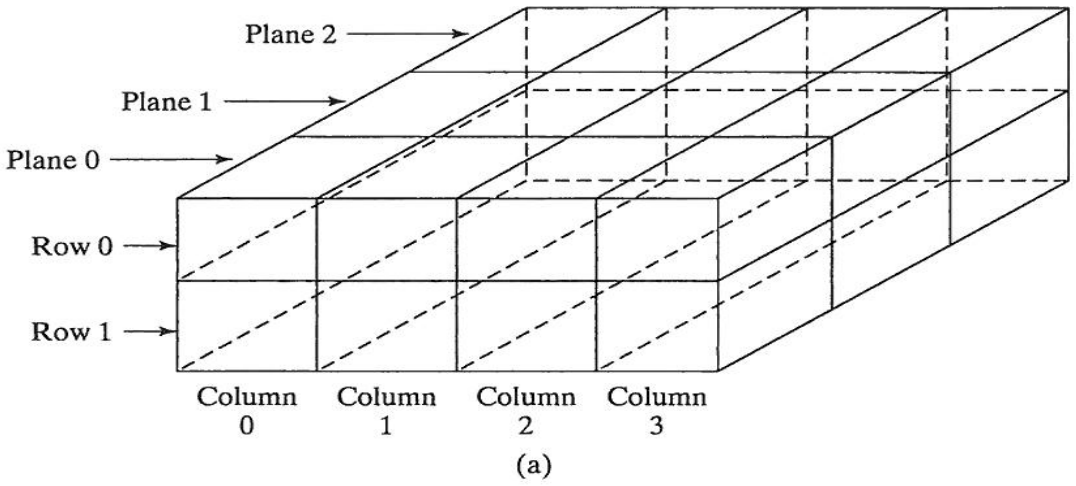
This second implementation is the simpler and most straightforward of the two. However, the *u*1 arrays *ar*[0] through *ar*[*u*1 − 1] would usually be allocated contiguously, with *ar*[0] immediately followed by *ar*[1], and so on. The first implementation avoids allocating the extra pointer array, *ap*, and computing the value of an explicit pointer to the desired row array. It is therefore more efficient in both space and time.

Multidimensional Arrays

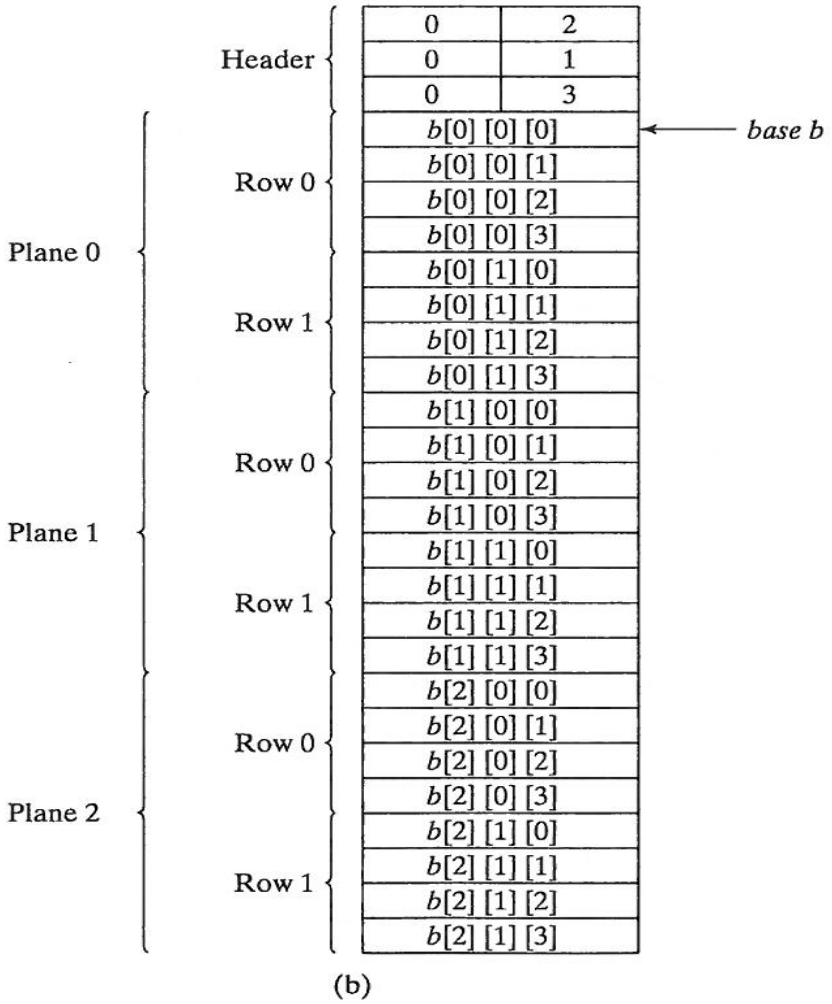
Java also allows arrays with more than two dimensions. For example, a three-dimensional array may be declared by

```
int b[ ][ ][ ] = new int[3][2][4];
```

and is illustrated in Figure 1.2.5a. An element of this array is specified by three subscripts, such as *b*[2][0][3]. The first subscript specifies a plane number, the second subscript, a row number, and the third, a column number. Such an array is useful when



(a)



(b)

FIGURE 1.2.5 Three-dimensional array.

a value is determined by three inputs. For example, an array of temperatures might be indexed by latitude, longitude, and altitude.

For obvious reasons, the geometric analogy breaks down when we go beyond three dimensions. However, Java does allow an arbitrary number of dimensions. For example, a six-dimensional array may be declared by

```
int c[ ][ ][ ][ ][ ][ ] = new int[7][15][3][5][8][2];
```

Referencing an element of this array would require six subscripts, such as *c*[2][3][0][1][6][1]. The number of different subscripts that are allowed in a particular position (the range of a particular dimension) equals the upper bound of that dimension. The number of elements in an array is the product of the ranges of all its dimensions. For example, the array *b* above contains 3*2*4 = 24 elements, while the array *c* contains 7*15*3*5*8*2 = 25,200 elements.

The row-major representation of arrays can be extended to arrays of more than two-dimensions. Figure 1.2.5b illustrates the representation of the array *b* of Figure 1.2.5a. The elements of the six dimensional array *c* described above are ordered as follows:

```
C[0][0][0][0][0][0]
C[0][0][0][0][0][1]
C[0][0][0][0][1][0]
C[0][0][0][0][1][1]
C[0][0][0][0][2][0]
    ...
    ...
C[6][14][2][4][5][0]
C[6][14][2][4][5][1]
C[6][14][2][4][6][0]
C[6][14][2][4][6][1]
C[6][14][2][4][7][0]
C[6][14][2][4][7][1]
```

That is, the last subscript varies most rapidly, and a subscript is not increased until all possible combinations of the subscripts to its right have been exhausted. This is similar to the odometer (mileage indicator) of a car, where the rightmost digit changes most rapidly.

What mechanism is needed to access an element of an arbitrary multidimensional array? Suppose that *ar* is an *n*-dimensional array declared by:

```
int ar[ ][ ]...[ ] = new int[r1][r2]...[rn];
```

and stored in row-major order. Each element of *ar* is assumed to occupy *esize* storage locations, and *base*(*ar*) is defined as the address of the first element of the array (i.e., *ar*[0][0]...[0]). Then, to access the element

```
ar[i1][i2]...[in];
```

it is first necessary to pass through *i*1 complete "hyper-planes," each consisting of *r*2*r3*...*rn* elements, to reach the first element of *ar* whose first subscript is *i*1. Then it is necessary to pass through an additional *i*2 groups of *r*3*r4*...*rn* elements in

order to reach the first element of *ar* whose first two subscripts are *i*1 and *i*2, respectively. A similar process must be carried out through the other dimensions until the first element whose first $n - 1$ subscripts match those of the desired element is reached. Finally, it is necessary to pass through *in* additional elements to reach the element desired.

Thus the address of $ar[i1][i2]\ldots[in]$ may be written as $base(ar) + esize*[i1*r2*\ldots*rn + i2*r3*\ldots*rn + \ldots + (i(n-1)*rn + in)]$, which can be evaluated more efficiently by using the equivalent formula:

```
base(ar) + esize * [in + rn * (i(n - 1) + r(n - 1)
* (... + r3 * (i2 + r2 * i1) ...) ) ]
```

This formula may be evaluated by the following algorithm that computes the address of the array element and places it into *addr* (assuming arrays *i* and *r* of size *n* to hold the indices and the ranges, respectively):

```
offset = 0;
for (j = 0; j < n; j++)
  offset = r[j] * offset + i[j];
addr = base(ar) + esize * offset;
```

Character Strings in Java

It is tempting to think of a string as a one-dimensional array of characters. Although a Java string shares several common characteristics with a Java array, a string in Java is actually an instance of the Java *String* class. A string constant or ***literal*** is denoted by any set of characters included in double quote marks. Various escape sequences can be used to represent special characters within a string: \n for a new-line character, \t for a tab character, \b for a backspace character, \" for the double quote character, \\ for the backslash character, \' for the single quote character, \r for the carriage return character, and \f for the form feed character. Thus, for example, "I DON\'T KNOW\n" represents the string I DON'T KNOW followed by the new-line character.

The method *length* may be used to obtain the length of a string. For example, suppose a programmer declares

```
String s1, s2;
s1 = new String ("HELLO THERE");
s2 = new String ("I DON\'T KNOW HIM");
```

then s1.*length*() is 11, and *s*2.*length*() is 16 (the ***escape sequence*** \' represents the single quote character).

The compiler creates a new *String* object each time it encounters a string literal. Thus the statement

```
s1 = "HELLO THERE";
```

has the same effect as the declaration of *s*1 shown above.

It is important to note that all strings in Java are immutable; that is, once a string is created, there is no way to change it. (Objects of a related class, *StringBuffer*, may be modified.) Recall that nonprimitive variables in Java are actually references to the object

being assigned. Thus assigning one string variable to another sets both variables pointing to the identical string. For example,

```
String s1, s2;
s1 = "HELLO THERE";      // s1 refers to a newly created String object
s2 = s1;                 // s1 and s2 refer to the identical String
                         // object
```

In order to determine whether or not two *String* variables refer to the same *String* object, we make use of the logical==operator. It is quite possible that two strings appear to be identical and yet are different. Consider the following statements:

```
String s1, s2;
s1 = "HELLO THERE";      // s1 refers to a newly created String object
s2 = "HELLO THERE";      // s2 refers to a newly created String
                         // object
```

The Boolean expression *s*1==*s*2 is ***false*** because *s*1 and *s*2 refer to different *String* objects. In order to compare two different *String* objects for equality, we may use the *String* method *equals*. The *String* method *compareTo* compares two strings lexicographically (i.e., in alphabetic order; for nonalphabetic characters, the UNICODE representation order is used). The *String* method *compareTo* returns 0 if the strings are lexicographically the same, a negative number if the string that invoked the *compareTo* method is lexicographically less than its parameter, and a positive number if the string that invoked the *compareTo* method is lexicographically greater than its parameter. The table below summarizes these relationships.

```
String s1, s2, s3, s4;
s1 = "HELLO THERE";
s2 = "HELLO THERE";
s3 = "GOOD BYE";
s4 = s1;
```

Expression	Value
s1==s2	false
s1==s4	true
s1.equals(s2)	true
s1.equals(s3)	false
s1.equals(s4)	true
s1.compareTo(s2)	0
s1.compareTo(s3)	Positive value
s3.compareTo(s1)	Negative value

To illustrate the immutability of strings and the fact that a string variable merely contains a reference to an actual string of characters, consider the following example. The lines are numbered for reference.

1. String s1 = "HELLO";
2. String s2 = "BYE";

3. String s3 = "GOOD";
4. s1 = s3;
5. s3 = s3 + s2;

Statements 1–3 set up three variables, *s*1, *s*2, and *s*3, and initialize them to reference newly allocated strings of characters, as shown in Figure 1.2.6a. Statement 4 results in Figure 1.2.6b. *s*1 and *s*3 now both reference the storage allocated for *s*3. Since nothing references the string "HELLO" any longer, that storage is ripe for garbage collection. Statement 5 creates new storage for "GOODBYE" and resets the reference in *s*3; the string "GOOD" is now only referenced by *s*1, it is not modified by the addition of "BYE". See Figure 1.2.6c.

Character String Operations

The Java *String* class implements a wide variety of methods operating on character strings. In addition to the *equals, compareTo*, and *length* methods presented earlier, the following methods are often used in this text. The reader is urged to consult a Java reference for other useful *String* methods.

- *str.charAt(int)* Returns the character at the specified index, *int*, of string *str*.
- *str1.concat(str2)* Concatenates *str*2 to the end of *str*1. Equivalent to *str*1 = *str*1 + *str*2;
- *str1.indexOf(str2)* Returns the index within *str*1 of the first occurrence of *str*2.
- *str1.indexOf(str2, int)* Returns the index within *str*1 of the first occurrence of *str*2, starting at the specified index.
- *valueOf(boolean)* Returns the *String* representation of the Boolean argument.
- *valueOf(char[ ])* Returns the *String* representation of the *char array* argument.
- *valueOf(char[ ], int, int)* Returns the *String* representation of a specific subarray of the *char array* argument. The second parameter represents the initial offset into the value of the string, and the third parameter represents the length of the value of the string.
- *valueOf(double)* Returns the *String* representation of the *double* argument.
- *valueOf(float)* Returns the *String* representation of the *float* argument.
- *valueOf(int)* Returns the *String* representation of the *int* argument.
- *valueOf(long)* Returns the *String* representation of the *long* argument.

Vectors in Java

The ***vector*** is one of the most useful of the so-called ***container*** classes that are available to the Java programmer. In many ways a vector is similar to an array, with two significant exceptions. First, unlike an array, a vector can grow and shrink automatically as items are inserted and deleted. Second, while an array, as a feature of the Java language,

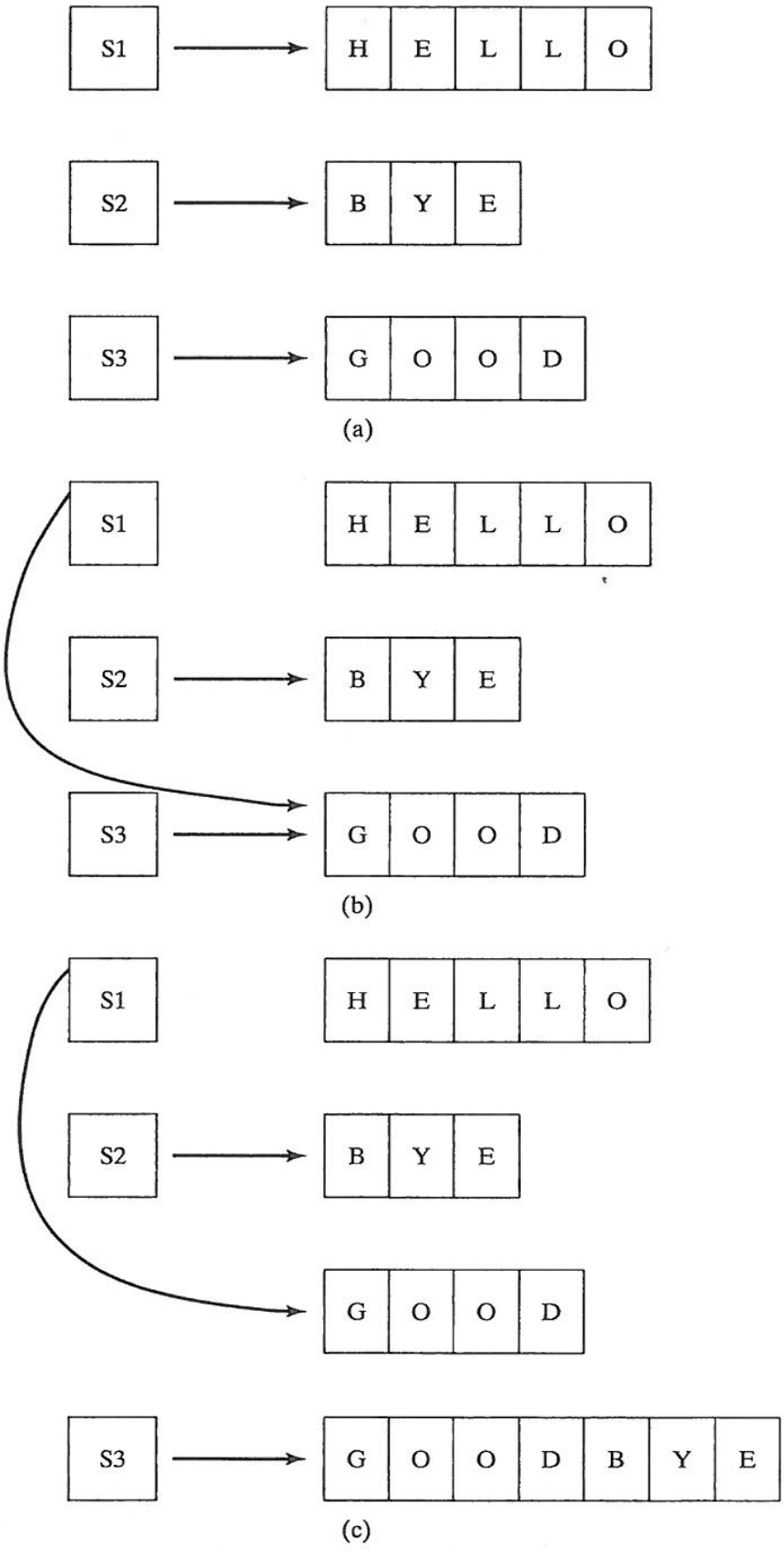


FIGURE 1.2.6 String assignments.

can contain elements of any type, vectors, because defined in the *java.util* package, may only contain elements of type *Object*, which we introduce in Section 1.3. We defer discussion of vectors to the end of the next section

EXERCISES

1.2.1 a. The ***median*** of an array of numbers is the element m of the array such that half the remaining numbers in the array are greater than or equal to m, and half are less than or equal to m, if the number of elements in the array is odd. If the number of elements is even, then the median is the average of the two elements $m1$ and $m2$ such that half the remaining elements are greater than or equal to $m1$ and $m2$, and half the elements are less than or equal to $m1$ and $m2$. Write a Java method that accepts an array of numbers and returns the median of the numbers in the array.

b. The ***mode*** of an array of numbers is the number m in the array that is repeated most frequently. If more than one number is repeated with equal maximal frequency, then there is no mode. Write a Java method that accepts an array of numbers and returns the mode or an indication that the mode does not exist.

1.2.2 Write a Java application that can do the following: Read a group of temperature readings (a reading consists of two numbers, an integer between −90 and 90 representing the latitude at which the reading was taken and the observed temperature at that latitude). Print a table consisting of each latitude and the average temperature at that latitude. If there are no readings at a particular latitude, print "NO DATA" instead of an average. Then print the average temperature in the northern and southern hemispheres (the northern consists of latitudes 1 through 90, and the southern, of latitudes −1 through −90). (This average temperature should be computed as the average of the averages, not the average of the original readings.) Also determine which hemisphere is warmer. In making the determination, take the average temperatures in all latitudes of each hemisphere for which there are data for both that latitude and the corresponding latitude in the other hemisphere. (For example, if there are data for latitude 57 but not for latitude −57, then the average temperature for latitude 57 should be ignored in determining which hemisphere is warmer.)

1.2.3 Write an application for a chain of twenty department stores, each of which sells ten different items. Every month, each store manager submits a data card for each item consisting of a branch number (from 1 to 20), an item number (from 1 to 10), and a sales figure (less than $100,000) representing the amount of sales for that item in that branch. However, some managers may not submit cards for some items (e.g., not all items are sold in all branches). Write a Java application that reads these data cards and prints a table with twelve columns. The first column should contain the branch numbers from 1 to 20 and the word "TOTAL" in the last line. The next ten columns should contain the sales figures for each of the ten items for each of the branches, with the total sales of each item in the last line. The last column should contain the total sales of each of the twenty branches for all items, with the grand total sales figure for the chain in the lower right-hand corner. Each column should have an appropriate heading. If no sales were reported for a particular branch and item, assume zero sales. Do not assume that your input is in any particular order.

1.2.4 Show how a checkerboard can be represented by a Java array. Show how to represent the state of a game of checkers at a particular instant. Write a Java method that accepts an array representing such a checkerboard and prints all possible moves that black can make from that position.

1.2.5 Write a method *printArray*(*a*) that accepts an *m* by *n* array *a* of integers and prints the values of the array on several pages as follows: Each page is to contain fifty rows and twenty columns of the array. Headings "COL 0", "COL 1", and so on, should be printed along the top of each page, and headings "ROW 0", "ROW 1", and so on, along the left margin of each page. The array should be printed by subarrays. For example, if *a* were a 100 by 100 array, the first page contains *a*[0][0] through *a*[49][19], the second page contains *a*[0][20] through *a*[49][39], the third page contains *a*[0][40] through *a*[49][59], and so on, until the fifth page contains *a*[0][80] through *a*[49][99], the sixth page contains *a*[50][0] through *a*[99][19], and so on. The entire printout occupies ten pages. If the number of rows is not a multiple of 50, or the number of columns is not a multiple of 20, the last pages of the printout should contain fewer than one hundred numbers.

1.2.6 Assume that each element of an array *a* stored in row-major order occupies four units of storage. If *a* is declared by each of the following, and the address of the first element of *a* is 100, find the address of the indicated array element:

a.	**int** a[ ] = **new int**[100];	address of	a[10]
b.	**int** a[ ] = **new int**[200];	address of	a[100]
c.	**int** a[ ][ ] = **new int**[10][20];	address of	a[0][0]
d.	**int** a[ ][ ] = **new int**[10][20];	address of	a[2][1]
e.	**int** a[ ][ ] = **new int**[10][20];	address of	a[5][1]
f.	**int** a[ ][ ] = **new int**[10][20];	address of	a[1][10]
g.	**int** a[ ][ ] = **new int**[10][20];	address of	a[2][10]
h.	**int** a[ ][ ] = **new int**[10][20];	address of	a[5][3]
i.	**int** a[ ][ ] = **new int**[10][20];	address of	a[9][19]

1.2.7 Write a Java method *listOff* that accepts two one-dimensional array parameters of the same size: *range* and *sub*. *range* represents the range of an integer array. For example if the elements of *range* are

```
3       5       10      6       3
```

range represents an array *a* declared by

```
int a[ ][ ][ ][ ][ ] = new int[3][5][10][6][3];
```

The elements of *sub* represent subscripts to the above array. If *sub*[*i*] does not lie between 0 and *range*[*i*] − 1, then all the subscripts from the *i*th onwards are missing. In the above example, if the elements of *sub* are

```
1       3       1       2       3
```

sub represents the one-dimensional array *a*[1][3][1][2]. The method *listOff* should print the offsets from the base of the array *a* represented by *range* of all the elements of *a* that are included in the array (or the offset of the single element if all the subscripts are within bounds) represented by *sub*. Assume that the size (*esize*)

of each element of a is 1. In the foregoing example, *listOff* would print the values 4, 5, and 6.

1.2.8 a. A ***lower triangular*** array a is an n by n array in which $a[i][j] == 0$ if $i < j$. What is the maximum number of nonzero elements in such an array? How can these elements be stored sequentially in memory? Develop an algorithm for accessing $a[i][j]$ where $i >= j$. Define an ***upper triangular*** array in an analogous manner, and do the same as above for such an array.

b. A ***strictly lower triangular array*** a is an n by n array in which $a[i][j] == 0$ if $i <= j$. Answer the questions in part (a) for such an array.

c. Let a and b be two n by n lower triangular arrays. Show how an n by $n + 1$ array c can be used to contain the nonzero elements of the two arrays. Which elements of c represent the elements $a[i][j]$ and $b[i][j]$, respectively?

d. A ***tridiagonal*** array a is an n by n array in which $a[i][j] == 0$ if the absolute value of $i - j$ is greater than 1. What is the maximum number of nonzero elements in such an array? How can these elements be stored sequentially in memory? Develop an algorithm for accessing $a[i][j]$ if the absolute value of $i - j$ is 1 or less. Do the same for an array a in which $a[i][j] == 0$ if the absolute value of $i - j$ is greater than k.

1.2.9 Write a Java method *numOccur* that accepts two strings and returns the number of times the second occurs as a substring of the first.

1.2.10 Write a Java method *reverse* that accepts a string $s1$ and returns the string $s2$ consisting of the reversal of $s1$ (i.e., the last character of $s1$ is the first character of $s2$, the next-to-last character of $s1$ is the second character of $s2$, and so on).

1.3 CLASSES AND OBJECTS IN JAVA

In this section, we introduce the concept of Java classes. We assume that you are familiar with basic object-oriented programming concepts from an introductory course. We review some highlights of this approach to programming and point out some interesting and useful features needed for a more general study of data structures.

Object-oriented programming (OOP) and design is a comparatively new paradigm in computer science that has slowly become the predominant method of software development. The basic concept of object orientation is to view data rather than procedure as the central focus of problem solution. Under this approach, applications model the problem at hand by describing each object in terms of a class that defines its contents and capabilities.

A ***class*** embodies the concept of an abstract data type by defining both the set of values of a given type and the set of operations that can be performed on them. A variable of a class type is known as an ***object***, and the operations on it are called ***methods***. When one object A invokes a method m on another object B, we sometimes say "A is sending message m to B." B is viewed as receiving the message and carrying out a transformation in response to it.

To illustrate these concepts, let us assume that a programmer wishes to describe a student attending a university. Typically, we would need to keep track of the student's identification number, name, grade point index, number of credits, and date of

admittance. In addition, we would often wish to print the student's grade point average and determine whether or not the student has the minimum number of credits necessary for graduation. Consider the following declaration:

```
public class Student {
  static final int GRADUATION = 120;
  static int totalStudents = 0;

  String idNumber;
  String firstName, lastName;
  double gpIndex;
  int credits;
  Date dateAdmitted;

  public boolean readyToGraduate () {
        return credits >= GRADUATION ? true: false ;
  } // end readyToGraduate method

  public void printGPIndex () {
        System.out.println("Grade Point Average is " + gpIndex);
  } // end printGPIndex method

  public static void printTotalStudents() {
        System.out.println("The total number of students is " +
        totalStudents);
  } // end printTotalStudents method

...       // other class methods

} // end Student class
```

The *Student* class defined above includes data elements, or ***members,*** of the class: *GRADUATION*, *totalStudents*, *idNumber*, *firstName*, *gpIndex*, *credits*, and *dateAdmitted*; as well as class methods: *readyToGraduate*, *printGPIndex*, and *printTotalStudents*. Note that *dateAdmitted* is defined in terms of the *Date* class. We leave the definition of the *Date* class as an exercise for the reader. Now suppose that an application contains the declaration

```
Student person1 = new Student();
Student person2 = new Student();
```

person1 and *person2* are reference variables that refer to newly created ***instances*** of objects of the *Student* class. *idNumber*, *firstName*, *gpIndex*, *credits*, and *dateAdmitted* are known as ***instance variables*** because each object of the class, namely *person*1 and *person*2, has its own copy of these variables. On the other hand, the ***static*** keyword specifies that *GRADUATION* and *totalStudents* are common to the entire class. Such variables are known as ***class variables***. The ***final*** keyword indicates that *GRADUATION* is a class constant and may not be assigned another value.

An object's instance variables may be assigned values by writing statements like the one shown below.

```
person1.gpIndex = 4.0;
```

We may think of this statement as representing a message to the object referred to by *person*1 to set its *gpIndex* to 4.0.

Similarly, *ReadyToGraduate* and *printGPIndex* are known as ***instance methods*** because each object of the *Student* class invokes these methods on its own instance variables. Specifying

```
person1.printGPIndex ();
```

prints the value contained in the *gpIndex* referred to by *person*1, while

```
person2.printGPIndex ();
```

prints the value contained in the *gpIndex* of the *person*2 object. By analogy to class variables, the ***static*** keyword in the header of the *printTotalStudents* method indicates that *printTotalStudents* is a ***class method***. Class methods are invoked by reference to the class (in this case *Student*). Thus, to print the total number of instantiated student objects we would write

```
Student.printTotalStudents();
```

A class method cannot reference a nonstatic data member of a class, because nonstatic data methods have individual values for each object of the class, whereas a static data member has a single value for the entire class.

Constructors

Each time a new object is created, Java automatically initializes all instance variables to their default values. **int**, **byte**, **short**, **long**, **float**, and **double** are initialized to 0, **char** to \u0000, **boolean** to **false**, and *String* to **null**.

Often it is desirable to instantiate an object with specific initial values. A ***constructor*** is a special method of a class that is invoked whenever an object of that class is created. A constructor is always named with the same name as the class itself. In our example above, a constructor is not specified; therefore, the ***default constructor*** is used and each of the object's instance variables is given its appropriate default initial value.

The class variable *totalStudents* is initialized to zero when the class is first accessed. Whenever an object of the class is created, it would be reasonable to increment this counter. This can be accomplished by including the following constructor in the class definition:

```
public Student() {
   totalStudents++;
} // end constructor
```

The constructor *Student* is invoked each time a new object is created.

Suppose we also wanted to initialize an object with a student's first and last names. We could define another constructor method for the *Student* class:

```
public Student(String first, String last) {
   firstName = first;
   lastName = last;
   totalStudents++;
} // end constructor
```

This constructor is invoked whenever a statement of the form

```
Student person3 = new Student("Boris", "Sery");
```

is used. The string arguments are passed to the constructor and assigned to the newly created object's instance variables.

The reader will note that we have defined two methods that have the same name but different numbers of parameters. Such methods are said to be ***overloaded***. That is, the same method name, in this case the class constructor, can apply to different methods if their parameters are of different types. There is no limit to the number of overloaded methods or constructors that may be defined. The compiler chooses the constructor that matches the arguments provided when the object is created. Overloaded methods are useful whenever the programmer wishes to accomplish the same basic tasks but with different numbers or types of parameters.

Representing Other Data Structures

Throughout the remainder of this text, classes are used to represent more complex data structures. Aggregating data and methods into a class is useful because it enables us to group objects within a single entity and to name each object appropriately according to its function.

As examples of how classes can be used in this fashion, let us consider the problem of representing rational numbers.

Rational Numbers

In Section 1.1 we presented an ADT for rational numbers. Recall that a ***rational number*** is any number that can be expressed as the quotient of two integers. Thus 1/2, 3/4, 2/3, and 2 (i.e., 2/1) are all rational numbers, but $\sqrt{2}$ and π are not. A computer usually represents a rational number by its decimal approximation. If we instruct the computer to print 1/3, it responds with .333333. Although this is close enough (the difference between .333333 and one-third is only one third of a million), it is not exact. If we were to ask for the value of 1/3 + 1/3, the result would be .666666 (which equals .333333 + .333333), while the result of printing 2/3 might be .666667. This would mean that the result of the test 1/3 + 1/3 == 2/3 would be false! In most instances, the decimal approximation is good enough, but sometimes it is not. It is therefore desirable to implement a representation of rational numbers for which exact arithmetic can be performed.

How can we represent a rational number exactly? Since a rational number consists of a numerator and a denominator, we can represent a rational number by defining a *Rational* class, as follows:

```
public class Rational {

  private long numerator, denominator;
  public Rational() {
        numerator = 0;
        denominator = 1;
  }
```

```
    public Rational (long i) {
          numerator = i;
          denominator = 1;
    }

    public Rational (long num, long denom) {
                              // We will modify this definition shortly
      numerator = num;
      denominator = denom;
    }

    // class methods to be developed below
    private Rational reduce () {      ...

    public boolean equals(Rational rat) {      ...

    public Rational multiply(Rational rat) {      ...

    public Rational add(Rational rat) {      ...

    public Rational divide(Rational rat) {      ...

    public String toString() {      ...

} // end Rational class
```

Then, when we declare an object to be a *Rational*, the appropriate constructor is invoked. The operator ***new*** in Java allocates a new object of the given type and returns a reference to it. When ***new*** is called, the constructor is invoked automatically. Thus the statement

```
Rational r = new Rational ();
```

declares a reference variable to an object of the *Rational* class, allocates a new object of the *Rational* class, initializes it to the rational zero (0/1) (since that is what the constructor *Rational* with no parameters does), and sets *r* pointing to it. The declaration

```
Rational r = new Rational (3)
```

sets *r* to the rational 3/1, since it invokes the second version of the constructor. Finally, the declaration

```
Rational r = new Rational (2, 5)
```

sets *r* to the rational 2/5, invoking the third version of *Rational*, with two parameters.

The instance variables *numerator* and *denominator* and the method *reduce* are ***private***. That is, they can be referenced only from within the methods of the *Rational* class. The methods *equals, multiply, add, divide*, and *toString*, by contrast, are ***public***. This means they can be referenced outside the methods of the *Rational* class.

The reasons for doing this are simple. We do not want "outsiders" manipulating either the *numerator* or *denominator* member. They are merely a way of implementing a rational number and are to be used solely for that purpose. An external method manipulates a *Rational*; only within the internal methods of *Rational* should we be able to access *numerator* and *denominator*. Similarly, the method *reduce* is a method to reduce

the internal representation of the *Rational* (i.e., the numerator and denominator) to lowest terms. We intend to use *reduce* to ensure that every rational number is kept in lowest terms. The outside world has no cause to call *reduce*. Every method that manipulates the internal numerator and denominator (i.e., *equals, add, multiply*, and *divide*) is a member of the *Rational* class and will automatically ensure that the resulting number is in reduced form by calling *reduce*. We will see this when we present the implementations of these methods. There is no need for anyone else to call *reduce*, and therefore *reduce* is defined as private.

On the other hand, the methods *equals, add, multiply, divide*, and *toString* are public. These functions form the public interface for the *Rational* class. That is, they are the methods by which the outside world can manipulate and use objects of type *Rational*. The ability to restrict access to certain members of the class to methods of the class itself is known as ***information hiding***.

We now turn our attention to the implementation of the class methods. You might think that we are now ready to define rational number arithmetic for our new representation, but there is one significant problem. Suppose we defined two rational numbers *r*1 and *r*2 and gave them values. How can we test whether the two numbers are the same? Perhaps you might want to code

```
if (r1.numerator == r2.numerator && r1.denominator == r2.denominator)
   ...
```

That is, if both numerators and denominators are equal, then the two rational numbers are equal. However, it is possible for both numerators and denominators to be unequal, yet the two rational numbers are the same. For example, the numbers 1/2 and 2/4 are indeed equal although their numerators (1 and 2) as well as their denominators (2 and 4) are unequal. We therefore need a new way of testing equality under our representation.

Well, why are 1/2 and 2/4 equal? The answer is that they both represent the same ratio. One out of two and two out of four are both one-half. In order to test rational numbers for equality, we must first reduce them to lowest terms. Once both numbers have been reduced to lowest terms, we can then test for equality by simple comparison of their numerators and denominators.

Define a ***reduced rational number*** as a rational number for which there is no integer that evenly divides both the denominator and numerator. Thus 1/2, 2/3, and 10/1 are all reduced, while 4/8, 12/18, and 15/6 are not. In our example, 2/4 reduced to lowest terms is $\frac{1}{2}$, so the two rational numbers are equal.

A procedure known as Euclid's algorithm can be used to reduce any fraction of the form *numerator/denominator* into its lowest terms. This procedure may be outlined as follows:

1. Let a be the larger of the *numerator* and *denominator*, and let b be the smaller.
2. Divide b into a, finding a quotient q and a remainder r (i.e., $a = q*b + r$).
3. Set $a = b$ and $b = r$.
4. Repeat steps 2 and 3 until b is zero.
5. Divide both the *numerator* and the *denominator* by the value of a.

As an illustration, let us reduce 1032/1976 to its lowest terms.

step 0	*numerator* = 1032		*denominator* = 1976	
step 1	$a = 1976$	$b = 1032$		
step 2	$a = 1976$	$b = 1032$	$q = 1$	$r = 944$
step 3	$a = 1032$	$b = 944$		
step 4 and 2	$a = 1032$	$b = 944$	$q = 1$	$r = 88$
step 3	$a = 944$	$b = 88$		
step 4 and 2	$a = 944$	$b = 88$	$q = 10$	$r = 64$
step 3	$a = 88$	$b = 64$		
step 4 and 2	$a = 88$	$b = 64$	$q = 1$	$r = 24$
step 3	$a = 64$	$b = 24$		
step 4 and 2	$a = 64$	$b = 24$	$q = 2$	$r = 16$
step 3	$a = 24$	$b = 16$		
step 4 and 2	$a = 24$	$b = 16$	$q = 1$	$r = 8$
step 3	$a = 16$	$b = 8$		
step 4 and 2	$a = 16$	$b = 8$	$q = 2$	$r = 0$
step 3	$a = 8$	$b = 0$		
step 5	1032/8 = 129		1976/8 = 247	

Thus 1032/1976 in lowest terms is 129/247.

Let us write a method to reduce a rational.

```
private Rational reduce () {
  long a, b, remainder;
  if (numerator > denominator) {
          a = numerator;
          b = denominator;
  }
  else {
          a = denominator;
          b = numerator;
  }
  while (b != 0) {
          remainder = a % b;
          a = b;
          b = remainder;
  }
  return new Rational(numerator / a, denominator / a);
} // end reduce method
```

Using the method *reduce*, we can write another method *equals* that determines whether or not two rational numbers *r*1 and *r*2 are equal. If they are, the method returns ***true***; otherwise, the method returns ***false***.

```
public boolean equals(Rational rat) {
        Rational r1, r2;
        r1 = reduce();
        r2 = rat.reduce();
        if (r1.numerator == r2.numerator && r1.denominator ==
            r2.denominator)
                return true;
        else
                return false;
} // end equals method
```

Note, at this point we may want to redefine the constructor *Rational*(***long*** *num*, ***long*** *denom*) to produce only rationals in reduced form. That is, it produces the rational number in reduced form that is equal to *num*/*denom*, as follows:

```
public Rational (long num, long denom) {
  long a, b, rem;

  if (num > denom) {
    a = num;
    b = denom;
  }
  else {
    a = denom;
    b = num;
  }
  while (b != 0) {
    rem = a % b;
    a = b;
    b = rem;
  }
  numerator = num / a;
  denominator = denom / a;
} // end Rational constructor
```

In this way, we are sure that all rationals produced by the constructors are in reduced form. We have to make sure that all other routines producing rationals (e.g. *add, multiply*) also produce rationals in reduced form. We can then assume that every *Rational* is in reduced form.

We may now write methods to perform arithmetic on rational numbers. We present a first try at a method to multiply two rationals.

```
public Rational multiply(Rational rat) {
  return new Rational(numerator * rat.numerator, denominator *
  rat.denominator);
} // end multiply method
```

In this method, two reduced rationals a/b and c/d, are multiplied by computing $(a*c)/(b*d)$. However, in this method, there is the danger that the product of the two numerators and two denominators may be too large even for a ***long*** variable. We would like the products to be as small as possible. The solution is to reduce a/d and c/b. In that way, since we assume that the input rationals a/b and c/d are in reduced form, we are certain that $a*c$ has no terms in common with $b*d$ and that the products are as small as possible.

Here is the method *multiply* implementing these ideas:

```
public Rational multiply(Rational rat) {
  Rational r1, r2;

  r1 = new Rational(numerator, rat.denominator);
  r2 = new Rational (rat.numerator, denominator);
  return new Rational(r1.numerator * r2.numerator,
                                    r1.denominator * r2.denominator);
} // end multiply method
```

The method *divide* simply multiplies by a reciprocal.

```
public Rational divide(Rational rat) {
  Rational rnl;

  rnl = new Rational(rat.denominator, rat.numerator);
  return  multiply(rnl);
} // end divide method
```

The method *equals* assumes that both rationals being tested for equality are in reduced form.

```
public boolean equals(Rational rat) {
  if (numerator == rat.numerator && denominator == rat.denominator)
    return true;
  else
    return false;
} // end equals method
```

Just as with the *multiply* method, to add two rational numbers we could first reduce each to lowest terms, then multiply the two denominators to produce a resulting denominator, then multiply each numerator by the denominator of the other rational number and add the two products to produce the numerator. The result can then be reduced to lowest terms. However, this too entails the danger that the product of the two denominators may be too large even for a ***long*** variable.

Instead, we use the following algorithm to add a/b to c/d. We assume that the value $rden(x, y)$ denotes the denominator of x/y reduced to lowest terms:

```
k = rden(b, d);
denom = b*k;                          // the resulting denominator
num = a*k +c*(denom / d);             // the resulting numerator
```

num is the numerator of the sum, *denom* is the denominator. We leave it as an exercise to show that this algorithm is correct.

Implementing this algorithm in the context of the *Rational* class provides the following definition of the method *add*:

```
public Rational add(Rational r) {
  Rational rnl;
  long k, denom, num;

  // implement the line k = rden(b, d); of the algorithm
  rnl = new Rational(denominator, rat.denominator);
  k = rnl.denominator;

  // compute the denominator of the result;
  // algorithm line denom = b*k;

  denom = denominator * k;
  // compute the numerator of the result
  // algorithm line num = a*k + c*a(denom/r.denominator);
  num = numerator * k + rat.numerator * (denom/rat.denominator);
  // form a Rational from the result and reduce the result to
  // lowest terms

  return rnl = new Rational(num, denom);
} // end add method
```

Most Java classes provide a method *toString* that converts an object to an appropriate *String* object suitable for printing. To implement the method *toString* we first must decide on a format for the output. A reasonable format might be to print the numerator followed by a slash followed by the denominator. We adopt this format in the routine below:

```
public String toString() {
   return numerator + "/" + denominator;
} // end toString method
```

Using the class *Rational*

The Java language provides two mechanisms for executing programs: ***applications*** and ***applets***. Applications are classes that are designed to be self-standing. Each application class contains a method of the form:

```
public static void main(String[ ] args) ... {
} // end main method
```

which is invoked by the Java interpreter. Statements contained in the *main* method are executed sequentially. Upon reaching the last statement in the method, control is returned to the operating system and the application terminates. Applications may invoke other methods, call upon other classes, read and write to files, and perform any operation defined by the Java language specification.

Often, however, it is desired to invoke programs in the context of a ***browser***. These programs, known as applets, are frequently downloaded from the Internet, and their origin may not be known. Security is often a prime concern. The Java language provides a mechanism by which applets are constrained to function within the confines

of the ***sandbox***, from which they can do no harm to the local environment. The most important restriction on an applet is that it has no access to the local file system. The reader is urged to consult a Java reference text for additional restrictions placed upon applets.

Applets are invoked when the browser encounters an HTML statement containing the *applet* tag. The *applet* tag may optionally contain *height* and *width* parameters that control the amount of space on the browser page allocated to the applet.

```
<applet code=appletClass height = x width = y> </applet>
```

Applets are often ***event-driven***, that is, they react to external events, such as mouse clicks, system events, and other forms of user-computer interaction. The browser invokes the applet, which in turn invokes the methods defined by the programmer. Many methods defined in the *java.applet* package are designed to allow the applet to interact with the user. Java's *abstract windowing toolkit* (AWT) provides a powerful framework for working with graphics in a windows environment. A graphical user interface, or ***gui***, may be designed by using the methods present in the *java.awt* and *java.awt.event* packages.

In the following example, we illustrate the use of the *Rational* class by means of an applet designed to display the user interface shown in Figure 1.3.1. Although the reader should be familiar with the basics of applet programming before continuing with the following example, detailed knowledge of Java applet programming is not necessary for an understanding of data structures.

We want to write an applet that allows the user to input two rational numbers, select the arithmetic operation from the keyboard, and indicate that the computation

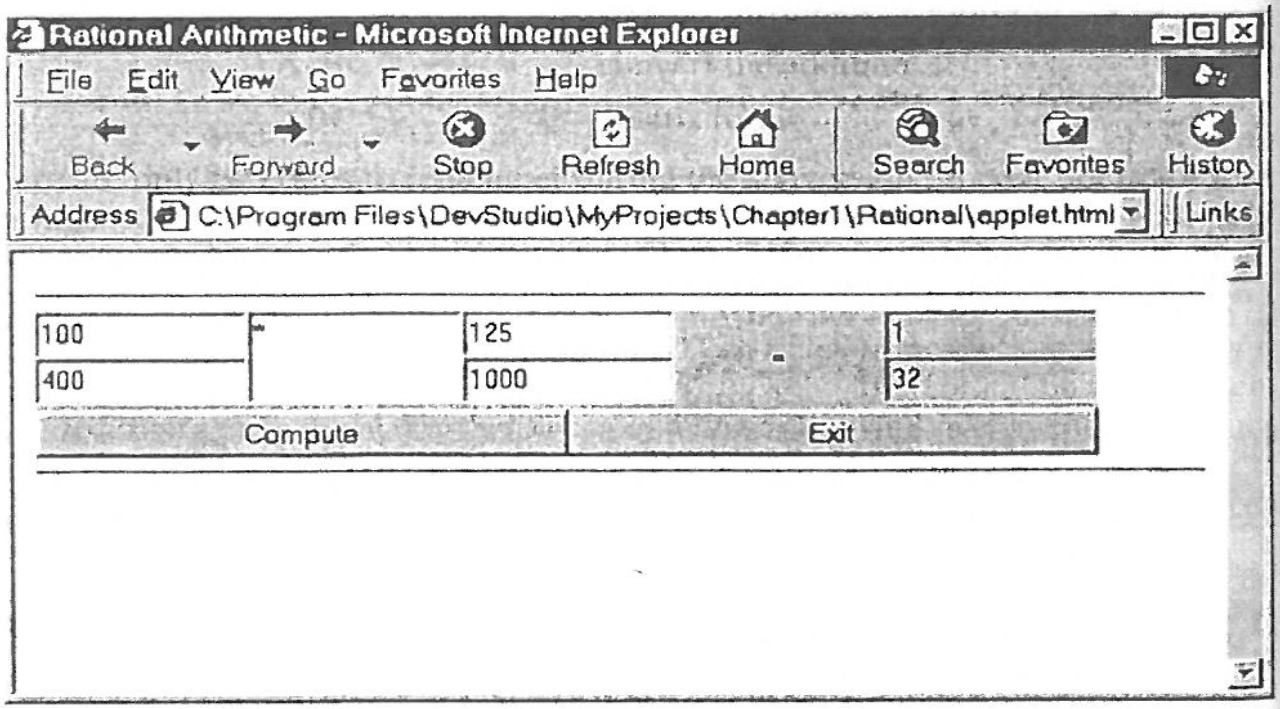


FIGURE 1.3.1 An applet displaying the result of the ***RationalApplet class***.

should be performed by means of a mouse click. The applet class, which we will call *RationalApplet*, may be invoked by the following HTML code:

```
<html>
   <head>
        <title>Rational Arithmetic</title>
   </head>
   <body>
        <hr>
        <applet code=RationalApplet height=75 width = 500> </applet>
        <hr>
   </body>
</html>
```

The *RationalApplet* class consists of three methods:

1. *init* is responsible for laying out the graphical user interface and is invoked when the applet is initilized by the browser.
2. *addButton* is responsible for instantiating a new mouse button and assigning an action to be performed when a mouse click is detected.
3. *actionPerformed* is responsible for performing the appropriate arithmetic operation as input by the user.

The applet is as follows:

```
import java.io.IOException;
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class RationalApplet extends Applet implements
ActionListener {
  private TextField num1numerator, num1denominator;
  private TextField num2numerator, num2denominator;
  private TextField ansnumerator, ansdenominator;
  private TextField operation;

  public void init() {
    setLayout(new BorderLayout());

    Panel p1=new Panel();
    p1.setLayout(new GridLayout(2,1));

    num1numerator=new TextField("",8);
    num1numerator.setEditable(true);
    p1.add(num1numerator, "North");

    num1denominator = new TextField("",8);
    num1denominator.setEditable(true);
    p1.add(num1denominator, "South");
```

```
        Panel p2=new Panel();
        p2.setLayout(new GridLayout(2,1));

        num2numerator=new TextField("",8);
        num2numerator.setEditable(true);
        p2.add(num2numerator, "North");

        num2denominator = new TextField("",8);
        num2denominator.setEditable(true);
        p2.add(num2denominator, "South");

        Panel p3=new Panel();
        p3.setLayout(new GridLayout(1,1));

        operation=new TextField("", 2);
        p3.add(operation,"Center");

        Panel p4=new Panel();
        p4.setLayout(new GridLayout(1,1));

        Label l=new Label("=",Label.CENTER);
        p4.add(l,"Center");

        Panel p5=new Panel();
        p5.setLayout(new GridLayout(2,1));

        ansnumerator=new TextField("",8);
        ansnumerator.setEditable(false);
        p5.add(ansnumerator, "North");

        ansdenominator = new TextField("",8);
        ansdenominator.setEditable(false);
        p5.add(ansdenominator, "South");

        Panel p=new Panel();
        p.setLayout(new GridLayout(1,5));
        p.add(p1);
        p.add(p3);
        p.add(p2);
        p.add(p4);
        p.add(p5);
        add(p, "North");

        Panel mainpanel=new Panel();
        mainpanel.setLayout(new GridLayout(1,2));
        addButton(mainpanel,"Compute");
        addButton(mainpanel,"Exit");
        add (mainpanel, "South");
    } // end init method

    public void addButton(Container c, String s) {
        Button b=new Button(s);
        c.add(b);
        b.addActionListener(this);
    } // end addButton method
```

```
public void actionPerformed(ActionEvent evt) {
    Rational result=new Rational();
    String s = evt.getActionCommand();
    if (s.equals("Compute")) {
          Rational r1 = new Rational(Long.parseLong(num1numerator.
          getText()),
                    Long.parseLong(num1denominator.getText()));
          Rational r2 = new
          Rational(Long.parseLong(num2numerator.getText()),
                    Long.parseLong(num2denominator.getText()));
    char operator=(operation.getText()).charAt(0);
    switch (operator) {
      case '+':     result = r1.add(r2);          break;
      case '*':     result = r1.multiply(r2);     break;
      case '/':     result = r1.divide(r2);       break;
      default:      System.out.println("Undefined operation");
    }
    String str = result.toString();
    int i = str.indexOf("/");
    ansnumerator.setText(str.substring(0,i));
    ansdenominator.setText(str.substring(i + 1));
  }
  else {
    System.exit(0);
  }
 } // end actionPerformed method
} // end RationalApplet class
```

Allocation of Storage and Scope of Classes

Until now we have been concerned with the declaration of variables, that is, the description of a variable's type or attribute and the methods that manipulate them. Two important questions, however, remain to be answered. At what point is a class or object (and its data members) associated with actual storage (i.e., ***storage allocation***)? At what point in a program may a particular class or object (and its data members) be referenced (i.e., ***scope*** or ***visibility*** of classes)?

In the previous section we distinguished between instance variables, which "belong" to each instance of the class, and class variables, which are "common" to the entire class. In Java, class variables (i.e., those with the ***static*** modifier) are automatically allocated storage at the time that the class is first loaded. If initial values are provided, they are assigned to the variables at the time of creation; otherwise the class variables are initialized to their default values. Instance variables, on the other hand, are allocated storage each time a new object is created. The programmer creates a new object (and associates storage with the object's variables) each time the ***new*** keyword is invoked. At that time the default class constructor is invoked and the object's variables are assigned their initial values. As we have seen, the programmer may override the default constructor with one or more class constructors that may be used to initialize the object and its data members. If more than one constructor is defined, the ***parameter signature*** is used to choose among them.

It is also possible to define a ***static initializer*** to initialize class variables. The static initializer is invoked only the first time any object of that class comes into existence. The programmer may specify any number of initial actions that are invoked at the time that the class is first loaded by preceding them with the ***static*** keyword. Static initalizers have the form:

```
static {        // static initializer
  ...
} // end initializer
```

We consider static initalizers further in Chapter 4.

Recall from Section 1.1 that Java always passes arguments to parameters of a method by a "call by value" mechanism. Storage for parameters declared within a method are allocated storage when the method is invoked. When the method terminates, storage assigned to those variables is deallocated. These parameters exist only as long as the method is active. Thus any change made to the parameter within the method does not affect the argument's initial value.

Once storage is allocated for a class or object, it remains allocated until the Java interpreter determines that it is no longer needed. At that time, a process known as ***garbage collection*** reclaims, or deallocates, the storage so that it may be reused. The allocation and deallocation of storage runs automatically in the background, freeing the programmer from having to worry about the details of the storage allocation process.

We now turn to the second of our questions, namely, the scope or visibility of an object. ***Encapsulation***, which is the ability to restrict access to certain members of the class to methods of the class itself, is an important component of modern programming. Were it not for the ability to restrict access to variables, it would be possible for programmers to inadvertently change the value of a class member by assigning a value to a similarly named variable. Such errors, known as ***side-effect*** errors, are very difficult to correct. Furthermore, programmers often wish to hide the details of the implementation of a class from other classes that may use it. This allows the programmer to change the implementation without having to worry that other classes that refer to it will "break" under the new implementation.

The Java language provides three keywords that allow the programmer to limit or extend the visibility of a class, its methods, and its data members: ***public***, ***private***, and ***protected***. If the keywords are not specified, the class (or its methods and data members) is said to have ***package visibility***. In order to understand how these concepts are applied, we first discuss the organization of a Java class and its relationship to the file and directory in which it is defined as well as its relationship to other classes.

A class is usually defined in a file having the same name as the class. Thus a class named *ClassA* might be defined in a file named *ClassA.java* (*ClassA.class* after it is compiled). All classes defined in the same directory are said to be in the same ***default package***. It is often useful for one class to be able to refer to a group of related classes that reside in a different directory. In such cases, the package is given a name. Each class of the package is placed in the same directory as other classes of the package and is prefaced by a header that identifies the name of the package. For example, suppose it was desired to define a package *packageA* which consists of two

related classes: *ClassB* and *ClassC*. *ClassB* and *ClassC* would both begin with the statement

```
package packageA;
```

and would be stored in a directory called *packageA*. If *ClassA* wanted to refer to the methods and data members of the classes defined in *packageA*, it would begin with the statement

```
import packageA.*;
```

in which the asterisk specifies that all classes of the package should be imported. The ability to import classes allows a programmer to make use of the large number of predefined classes provided by the Java language. In addition, the use of packages allows the Java programmer to organize a large number of classes by function and facilitates the reuse of code.

In Section 2.3 we use the concept of inheritance in which one or more independently defined classes are said to ***inherit*** the methods and data elements of the class upon which they are based. Such classes are known as ***subclasses*** and can be identified by the use of the keyword ***extends***. For example, if *ClassD* wished to inherit the methods and data members of *ClassA* (in order to extend the capabilities of *ClassA*), it would begin with the header

```
public class ClassD extends ClassA {
```

ClassA is known as the ***superclass*** of *ClassD*. The reader is urged to review the concept of inheritance before proceeding.

Ordinarily, only one class is defined per file. Java 1.1 extended the Java language by allowing classes to be nested within other classes. Such ***inner classes*** are used as auxiliaries, or "helpers," by their container classes and are not intended for use by other "top-level" classes. During compilation, the Java compiler locates each inner class and creates a separate *.class* file.

To summarize, the organizational structure of a Java application or applet consists of packages, classes, subclasses, and inner classes; and four visibility categories: public, private, protected, and package. A private method or member may only be referenced from within the class in which it is declared. No other class may access the private methods or members of another, nor are they inherited by any subclass that extends the class in which they are declared. Public elements, on the other hand, are visible to any class that wishes to use them. Protected methods and members are visible to all classes within a package, but may not be accessed by classes that lie outside the package. Protected methods are also visible to subclasses of the class in which they are defined, even if those classes are not in the same package. By default, (i.e., if neither ***public, private***, nor ***protected*** is specified), the methods and members have package visibility, that is, they are visible only within the package in which they are defined and may not be used by classes or even subclasses that are defined outside the package.

To illustrate these rules, consider the following application (the numbers to the left of each line are for reference purposes). Figure 1.3.2 illustrates the directory structure of the files in which the various classes are defined.

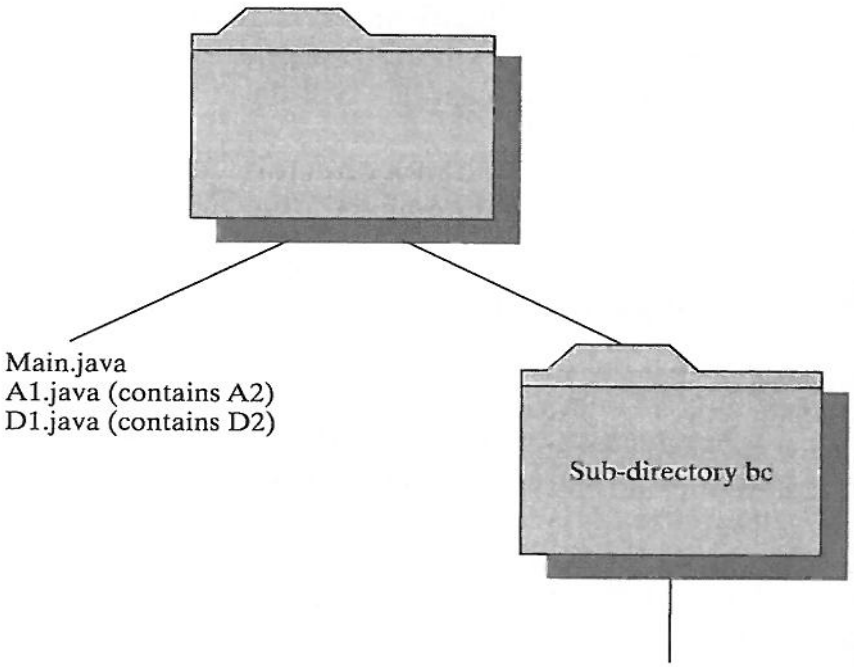


FIGURE 1.3.2 File layout for scope example.

```
// source Main.java

1   import bc.*;
2   public class Main {
3         public static void main (String[ ] args) {
4               A1 a1main = new A1();
5               a1main.print();
6               System.out.print(a1main.x + "    ");
7               System.out.print(a1main.y + "    ");
8               System.out.println(a1main.z);
9               A1.A2 a2main = new A1.A2();
10              a2main.print();
11              bc.B bmain = new bc.B();
12              bmain.print();
13              bc.C cmain = new bc.C();
14              cmain.print();
15              System.out.print(cmain.x + "    ");
16              System.out.print(cmain.y + "    ");
17              System.out.println(cmain.z);
18              D2 d2main = new D2();
19              d2main.println();
20        } // end main
21    } // end class Main

// end of source Main.java
```

```
// source A1.java

22  public class A1 {
23        public static int x, y, z;
24        private static int a, b;

25        public A1() {
26              x = 1;
27              y = 2;
28              z = 3;
29              a = 1;
30              b = 2;
31        } // end constructor A1

32        public void print() {
33              System.out.print(x + "    ");
34              System.out.print(y + "    ");
35              System.out.print(z + "    ");
36              System.out.print(a + "    ");
37              System.out.println(b + "    ");
38        } // end print
39        public static class A2 {
40              private int a, b;

41              public A2() {
42                    a = 4;
43                    b = 5;
44              } // end constructor A2

45              public void print() {
46                    System.out.print(A1.x + "    ");
47                    System.out.print(A1.y + "    ");
48                    System.out.print(A1.z + "    ");
49                    System.out.print(A1.a + "    ");
50                    System.out.println(A1.b);
51                    System.out.print(a + "    ");
52                    System.out.println(b);
53              } // end print
54        }  // end inner class A2
55  }  // end outer class A1

// end of source A1.java

// source D1.java

56  public class D1 {
57        protected int n;
58  } // end class D1

// end source of D1.java
```

```
// source D2.java

59  class D2 extends D1 {
60        public D2() {
61              n = 3;
62        } // end constructor D2

63        public void println() {
64              System.out.println(n);
65        } // end println
66  } // end class D2

// end of source D2.java

// subdirectory bc

// source B.java

67      package bc;
68      public class B {
69          int b;

70          public B() {
71              b = 7;
72          } // end constructor B

73          public void print() {
74              System.out.println(b);
75          } // end print
76      } // end class B

// end of source B.java

// source C.java

77  package bc;
78  public class C {
79        public int x, y, z;
80        B bnum = new B();

81        public C() {
82              x = 10;
83              y = 20;
84              z = 30;
85        } // end constructor C

86        public void print() {
87              System.out.print(x + "   ");
88              System.out.print(y + "   ");
89              System.out.println(z);
90              System.out.println(bnum.b);
91        } // end print
92  } // end class C

// end of source C.java
```

Execution of the application yields the following results:

```
a  1  2  3  1  2      // public variables x, y, z and private
                      // variables a, b are printed by using the
                      // print method of class A1
b  1  2  3            // public variables x, y, z are printed by main
c  1  2  3  1  2      // public variables x, y, z and private
                      // variables a, b of class A1 are printed by
                      // inner class A2
d  4  5               // public variables a, b of inner class A2 are
                      // printed by class A2
e  7                  // package variable b is printed by class B
f  10  20  30         // public variables x, y, z are printed by
                      // class C
g  7                  // package variable b of class B is printed by
                      // class C
h  10  20  30         // public variables x, y, z of class C are
                      // printed by main
i  3                  // protected variable n created by class D1 is
                      // printed by class D2 extending D1
```

Let us trace through the application. Execution begins with line 1, in which the *Main* class imports all the public methods and data members of *package bc*. Note that *package bc*, defined in lines 67–90, is in a subdirectory of the current directory called *bc* (see Figure 1.3.2) and contains classes *B* and *C*. Both class *B* and class *C* are members of *package bc*, because they are identified as members of the package in lines 67 and 77, respectively.

Execution then continues at line 2, which defines the class *Main*. Every application must have one *Main* class containing a *main* method (line 4) that marks the entry point into the application. The method *main* defines an object *a1main* of the class *A1* and instantiates the object by invoking its constructor (lines 25–31), allocating storage for its data members in line 4. Examination of the definition of class *A1* (lines 22–55) reveals three ***public*** data members [*x*, *y*, *z* (line 23)] and two ***public*** methods [constructor *A1* (lines 25–31) and *print* (lines 32–38)] which will be available to all classes that wish to use them. The ***private*** data members, [*a* and *b* (line 24)] will only be available to the methods of their own class. Class *A1* also contains an inner class, *A2* (lines 39–54), which will be discussed later.

The *main* method then invokes the *print* method of class *A1* (line 5), resulting in the output shown on line a. Since this *print* method is defined as ***public*** (line 32), it may be called by any method in the application, including the *main* method. Because data members *x*, *y*, and *z* are ***public***, they may be printed either by the *print* method of class *A1* (line 5) or by the *print* (lines 6 and 7) and *println* (line 8) methods of the default *System.out* object (output line b). However, ***private*** data members *a* and *b* may only be accessed through the methods of the class in which they are defined.

Line 9 then defines a new object *a2main* of the inner class *A2* (lines 39–54) and instantiates the object by invoking its constructor (lines 41–44). Inner classes are hidden within the outer class and are only accessible through the outer class. The ***static*** keyword (line 39) restricts class *A2* from directly accessing any of the data members of the outer class *A1*. Once the *main* method has created an object (*a2main*) of the *A2*

class, its public methods may be used. Line 10 then calls upon the ***public*** *print* method of *A*2 to print the ***public*** variables *x*, *y*, *z*, and the ***private*** variables *a*, *b* of its outer class *A*1 (output line c) and its own ***private*** variables *a* and *b*, producing the output of line d. Note that when the *print* method of *A*2 refers to the variables *a* and *b* without specifying a class, it refers to the data members of *A*2, while under the same circumstances, the *print* method of *A*1 refers to the *a* and *b* of *A*1.

Line 11 of *main*, referring to the package *bc*, defines a new object *cmain* of the class *B*, thus instantiating an integer variable *b* with an initial value of 7 by invoking its constructor (lines 70–72). Class *B*'s *print* method (lines 73–75) is invoked on line 12, resulting in output line e.

Note that variable *b* of class *B* is declared to be neither ***public*** nor ***private*** (line 69). Therefore, *b* has package visibility and is accessible to all classes in the package.

Execution then continues with line 13, which defines a new object *cmain* of the class *C*. When *cmain* is instantiated, a new object *bnum* of class *B* is created within class *C* (line 80), which upon creation invokes *B*'s constructor (line 70–72). Class *C*'s constructor (lines 81–85) is also invoked, thus initializing the ***public*** variables *x*, *y* and *z*. Of course, these variables *x*, *y*, and *z* belong to the object *cmain,* and should not be confused with the variables *x*, *y*, and *z* which belong to *a1main*. Line 14 then invokes the ***public*** *print* method of class *C,* resulting in output lines f and g. Note that *C*'s *println* may access *bnum.b* (line 90) because both *B* and *C* are in the same package and *b* has package visibility. Lines 15, 16, and 17 use the default *System.out.print* and *System.out.println* to print the *cmain* ***public*** variables *x*, *y*, and *z*, resulting in output line h.

Returning to line 18 of the *main* method, a new object *d2main* of class *D*2 is created. Since class *D*2 ***extends*** class *D*1 (line 59), class *D*2 inherits all of *D*'s methods and data members. The subclass *D*2 behaves as if all of the methods and data members of *D*1 were defined within it. Therefore, when the constructor for *D*2 (lines 60–62) is invoked at the time of the creation of *d2main*, the variable *n* of the *D*1 class is initialized. *n* is ***protected*** (line 57) and thus is available to all subclasses and package members of the class in which it is created. Finally, line 19 of the *main* method invokes *D2.println*, resulting in output line i.

Vectors in Java

One of the major limitations of arrays is their static nature. Once an array is created it is of a fixed size. Thus the programmer is often faced with the following dilemma: If the amount of storage necessary for the application has been overestimated, the array, once created, may contain a significant amount of wasted storage; on the other hand, if the amount of required storage is underestimated, a new array of larger size will have to be created, and all the information from the first array will need to be copied over to the second array. The *java.util* package addresses these issues by defining an object known as a ***vector*** that can change its size dynamically.

In order to illustrate how a vector is used, we will revisit the *Student* class described earlier in this section. We repeat here, for the reader's convenience, the definition of this class.

```
public class Student {
    static final int GRADUATION = 120;
    static int totalStudents = 0;
```

```
    String idNumber;
    String firstName, lastName;
    double gpIndex;
    int credits;
    Date dateAdmitted;
  ...       // other class methods
} // end Student class
```

Suppose it was desired to store information on a number of students. Since the number of students will change from semester to semester, it is appropriate to store the students' information in a vector. A vector *classRecords*, with an initial capacity of twenty-five student records, may be defined by:

```
Vector classRecords = new Vector(25);
```

This defines a vector which has the potential of holding up to twenty-five records. However, unlike an array, if you attempt to add a twenty-sixth member, the vector will automatically double in size (with room for a total of fifty records). Should you once again exceed the vector's capacity by attempting to insert a fifty-first record, the vector will once again double its size (with room for a total of one hundred records), and so on. In general, a vector *v* may be defined by the statement:

```
Vector v = new Vector(m, n);
```

where *m* represents the amount of initial storage, and *n* represents the amount of storage that is added to the vector when its current size is exceeded. If *n* is omitted, the vector doubles in size; if both *m* and *n* are omitted, a vector with the potential for ten objects is created by default. The current size of a vector may be obtained by using the *v.size* method. Vectors, like arrays, are indexed from zero to *v.size*() - 1.

In order to insert an object into a vector, the *add* method of the *Vector* class is used. For example, using the *Student* class constructor presented earlier in this section, the statements

```
classRecords.add(new Student("Boris", "Sery"));
classRecords.add(new Student("Edward", "Mardakhaev"));
classRecords.add(new Student("Marina", "Marchenko"));
```

would add three students into the first three positions of the *classRecords* vector. It is also possible to insert an object into the middle of a vector. Thus the statement

```
classRecords.add(1, new Student("Shalva", "Landy"));
```

would add the student whose name is "Shalva Landy" to position one of the *classRecords* vector, shifting the remaining elements up.

An object may be removed from a vector by invoking the *remove* method. Thus the statement

```
Student s = (Student) classRecords.remove(2);
```

would remove the student whose name is "Edward Mardakhaev" from the vector, shifting the remaining elements down by one, and assigning the object returned by the *remove* method to the variable *s*. Note that the object must be cast as an object of the

Student class before it is assigned to the variable *s*. This is because a vector can only contain items of the *Object* class, and *Objects* must be cast into a compatible type before they may be assigned to a class object.

Java 2 added the *v.get(i)* and *v.set(i, x)* methods to the *Vector* class. The *get* method retrieves the object stored in location *i* of vector *v* without removing it from the vector. It should be noted that the *set* method overwrites the contents of location *i* with the object *x*, while the *add* method inserts object *x* at the specified location, shifting all items in the vector up to make room for the newly inserted item.

Unlike an array, vector elements do not exist until objects are added to the vector. Thus, assuming the sequence of statements above, the statement

```
classRecords.add(10, new Student("Alexander", "Kaplan"));
```

yields an "*ArrayIndexOutOfBoundsExeception* 10 > 3", since the vector *classRecords* currently contains only three elements. The *setSize(n)* method allows the programmer to set the size of the vector to *n* elements. If *n* is greater than the current size of the vector, this method increases the vector to size *n*, setting the additional elements to ***null***. If *n* is less than the current size of the vector, the vector is trimmed to size *n*, and the elements that were above the $(n-1)^{th}$ position are discarded. Thus the statements:

```
classRecords.setSize(15);
classRecords.add(10, new Student("Alexander", "Kaplan"));
```

would set the size of the *classRecords* vector to fifteen, then increase the size to sixteen, inserting "Alexander Kaplan" in the vector at index ten, and setting the objects in index three through nine and eleven through fifteen to ***null***.

The reader is urged to consult the Java documentation in order to learn about the other vector methods defined in the *java.util* package.

EXERCISES

1.3.1 Implement complex numbers, as specified in Exercise 1.1.8, using structures with real and complex parts. Write routines to add, multiply, and negate such numbers.

1.3.2 Suppose a real number is represented by a Java class, such as:

```
public class Real {
      private long left, right;
...
```

where *left* and *right* represent the digits to the left and right of the decimal point, respectively. If *left* is a negative integer, then the represented real number is negative.

a. Write a routine to input a real number, and create an object representing that number.

b. Write a function that accepts such a structure and returns the real number represented by it.

c. Write routines *add*, *subtract*, and *multiply* that accept two such structures, and set the value of a third structure to represent the number that is the sum, difference, and product, respectively, of the two input records.

1.3.3 Assume two arrays, one of student records, the other of employee records. Each student record contains members for a last name, first name, and grade point index. Each employee record contains members for a last name, first name, and salary. Both arrays are ordered in alphabetical order by last name and first name. Two records with the same last name/first name do not appear in the same array. Write a Java method to give a 10 percent raise to every employee who has a student record and whose grade point index is higher than 3.0.

1.3.4 Write a method as in the preceding exercise, but assume that the employee and student records are kept in a vector rather than in two ordered arrays.

1.3.5 Write a method as in Exercise 1.3.3, but assuming that the employee and student records are kept in two ordered external files rather than in two ordered arrays.

1.3.6 Using the rational number representation given in the text, write routines to add, subtract, and divide such numbers.

1.3.7 Write a method *negate* for the class *Rational* that returns the negative of a rational number.

1.3.8 The text presents a method *equals* that determines whether or not two rational numbers *r*1 and *r*2 are equal by first reducing *r*1 and *r*2 to lowest terms and then testing for equality. An alternative method would be to multiply the denominator of each by the numerator of the other and testing the two products for equality. Write a method *equal*2 to implement this algorithm. Which of the two methods is preferable?

1.3.9 Define a class *NewString* that represents a string by a length and a reference to an array of characters.

a. Write a constructor for *NewString* that allocates appropriate storage for it and initializes it to a given Java string.

b. Write a constructor for *NewString* that allocates storage of a given size for the string but does not initialize its characters.

c. Write a method *concat* that concatenates one *NewString* with another.

1.3.10 Implement a class *NewString*, as in the previous exercise, using a vector.