# Detailed Contents

# 8 Introduction to Algorithms and Programming Languages

## Learning Objectives

In this chapter, we will learn the technique of writing algorithms and pseudocodes, and drawing a schematic flow of logic in the form of flowcharts. Algorithms, pseudocodes, and flowcharts are used in the design phase of the software/program development process to help the programmers and users to clearly understand the solution to the problem at hand.

This chapter gives a detailed note on several generations of programming languages. The reader will also learn the fundamentals of structured programming languages and the key to designing and implementing correct, accurate, efficient, and maintainable programs.

## 8.1 ALGORITHM

The typical meaning of an algorithm is a formally defined procedure for performing some calculation. If a procedure is formally defined, then it must be implemented using some formal language, and such languages are known as *programming languages*. The algorithm gives the logic of the program, that is, a step-by-step description of how to arrive at a solution.

In general terms, an algorithm provides a blueprint for writing a program to solve a particular problem. It is considered to be an effective procedure for solving a problem in a finite number of steps. Thus, a well-defined algorithm always provides an answer, and is guaranteed to terminate.

Algorithms are mainly used to achieve *software re-use*. Once we have an idea or a blueprint of a solution, we can implement it in any high-level language, such as C, C++, Java, and so on. In order to qualify as an algorithm, a sequence of instructions must possess the following characteristics:

- Be precise
- Be unambiguous
- Not even a single instruction must be repeated infinitely
- After the algorithm gets terminated, the desired result must be obtained

## 8.2 CONTROL STRUCTURES USED IN ALGORITHMS

An algorithm has a finite number of steps and some steps may involve decision-making and repetition. Broadly speaking, an algorithm uses three control structures, namely sequence, decision, and repetition.

### 8.2.1 Sequence

Sequence means that each step of the algorithm is executed in the specified order. An algorithm to add two numbers is given in Figure 8.1. This algorithm performs the steps in a purely sequential order.

```
Step 1: Input the first number as A
Step 2: Input the second number as B
Step 3: Set Sum = A + B
Step 4: Print Sum
Step 5: End
```

**Fig. 8.1** Algorithm to add two numbers

### 8.2.2 Decision

Decision statements are used when the execution of a process depends on the outcome of some condition. For example, `if x = y, then print "EQUAL"`. Hence, the general form of the `if` construct can be given as:

```
if condition then process
```

A condition in this context is any statement that may evaluate either to a true value or a false value. In the preceding example, the variable *x* can either be equal or not equal to *y*. However, it cannot be both true and false. If the condition is true then the process is executed.

A decision statement can also be stated in the following manner:

```
if condition
      then process1
else process2
```

This form is commonly known as the if-else construct. Here, if the condition is true then `process1` is executed, else `process2` is executed. An algorithm to check the equality of two numbers is shown in Figure 8.2.

```
Step 1: Input the first number as A
Step 2: Input the second number as B
Step 3: if A = B
            then print "Equal"
        else
            print "Not equal"
Step 4: End
```

**Fig. 8.2** Algorithm to test the equality of two numbers

### 8.2.3 Repetition

Repetition, which involves executing one or more steps for a number of times, can be implemented using constructs such as the `while`, `do-while`, and `for` loops. These loops execute one or more steps until some condition is true. Figure 8.3 shows an algorithm that prints the first 10 natural numbers.

```
Step 1: [initialize] Set I = 1, N = 10
Step 2: Repeat Steps 3 and 4 while I <= N
Step 3: Print I
Step 4: Set I = I+1
Step 5: End
```

**Fig. 8.3** Algorithm to print the first 10 natural numbers

## 8.3 SOME MORE ALGORITHMS

Let us write some more algorithms.

**Example 8.1** Write an algorithm for interchanging/swapping two values.

*Solution*

```
Step 1: Input first number as A
Step 2: Input second number as B
Step 3: Set temp = A
Step 4: Set A = B
Step 5: Set B = temp
Step 6: print A, B
Step 7: End
```

**Example 8.2** Write an algorithm to find the larger of two numbers.

*Solution*

```
Step 1: Input first number as A
Step 2: Input second number as B
Step 3: if A > B
             then print A
        else if A < B
             then print B
        else
             print "The numbers are equal"
Step 4: End
```

**Example 8.3** Write an algorithm to find whether a number is even or odd.

*Solution*

```
Step 1: Input the number as A
Step 2: if A % 2 = 0
             then print "Even"
        else
             print "Odd"
Step 3: End
```

**Example 8.4** Write an algorithm to print the grade obtained by a student using the following rules:

| Marks | Grade |
|---|---|
| Above 75 | O |
| 60-75 | A |
| 50-60 | B |

| | |
|---|---|
| 40-50 | C |
| Less than 40 | D |

*Solution*

```
Step 1: Enter the marks obtained as M
Step 2: if M > 75
             then print “O”
Step 3: if M >= 60 and M < 75
             then print “A”
Step 4: if M >= 50 and M < 60
             then print “B”
Step 5: if M >= 40 and M < 50
             then print “C”
          else
             print “D”
Step 6: End
```

**Example 8.5** Write an algorithm to find the sum of first N natural numbers.

*Solution*

```
Step 1: Input N
Step 2: Set I = 1, sum = 0
Step 3: Repeat Step 4 while I <= N
Step 4: Set sum = sum + I
          Set I = I + 1
Step 5: print sum
Step 6: End
```

## 8.4 FLOWCHARTS

A flowchart is a graphical or symbolic representation of a process. It is basically used to design and document virtually complex processes to help the viewers to visualize the logic of the process, so that they can gain a better understanding of the process and find flaws, bottlenecks, and other less obvious features within it.

When designing a flowchart, each step in the process is depicted by a different symbol and is associated with a short description. The symbols in the flowchart (refer Figure 8.4) are linked together with arrows to show the flow of logic in the process.

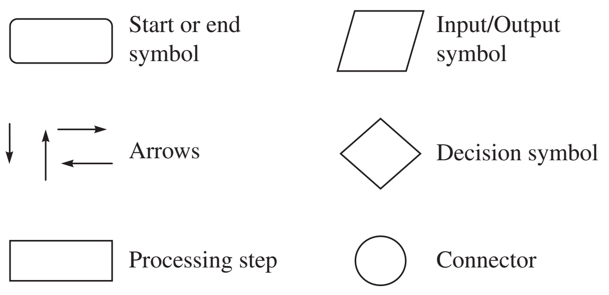


**Fig. 8.4** Symbols of flowchart

The symbols of a flowchart include:

- *Start and end symbols* are also known as the terminal symbols and are represented as circles, ovals, or rounded rectangles. Terminal symbols are always the first and the last symbols in a flowchart.
- *Arrows* depict the flow of control of the program. They illustrate the exact sequence in which the instructions are executed.
- *Generic processing step,* also called as activity, is represented using a rectangle. Activities include instructions such as `add a to b, save the result`. Therefore, a processing symbol represents arithmetic and data movement instructions. When more than one process has to be executed simultaneously, they can be placed in the same processing box. However, their execution will be carried out in the order of their appearance.
- *Input/output symbols* are represented using a parallelogram and are used to get inputs from the users or display the results to them.
- A *conditional or decision symbol* is represented using a diamond. It is basically used to depict a Yes/No question or a True/False test. The two symbols coming out of it, one from the bottom point and the other from the right point, corresponds to Yes or True, and No or False, respectively. The arrows should always be labelled. A decision symbol in a flowchart can have more than two arrows, which indicate that a complex decision is being taken.
- *Labelled connectors* are represented by an identifying label inside a circle and are used in complex or multi-sheet diagrams to substitute for arrows. For each label, the ‘outflow’ connector must have one or more ‘inflow’ connectors. A pair of identically labelled connectors is used to indicate a continued flow when the use of lines becomes confusing.

### 8.4.1 Significance of Flowcharts

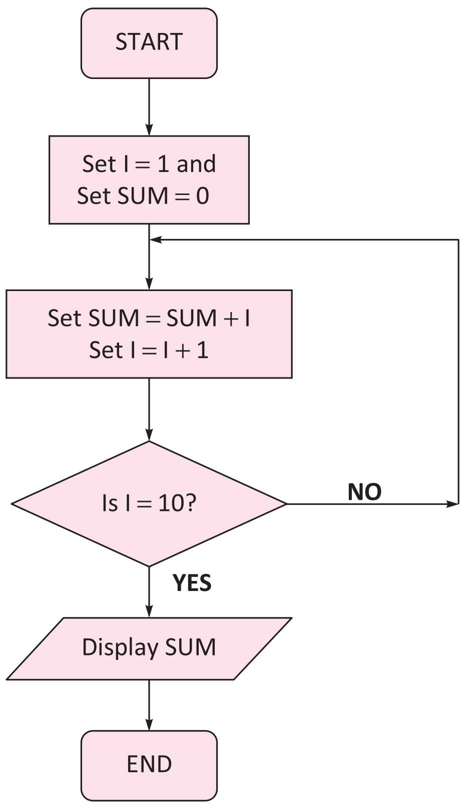
A flowchart is a diagrammatic representation that illustrates the sequence of steps that must be performed to solve a problem. It is usually drawn in the early stages of formulating computer solutions. It facilitates communication between programmers and users. Once a flowchart is drawn, programmers can make users understand the solution easily and clearly.

Flowcharts are very important in the programming of a problem as they help the programmers to understand the logic of complicated and lengthy problems. Once a flowchart is drawn, it becomes easy for the programmers to write the program in any high-level language. Hence, the flowchart has become a necessity for better documentation of complex programs.

A flowchart follows the top-down approach in solving problems.

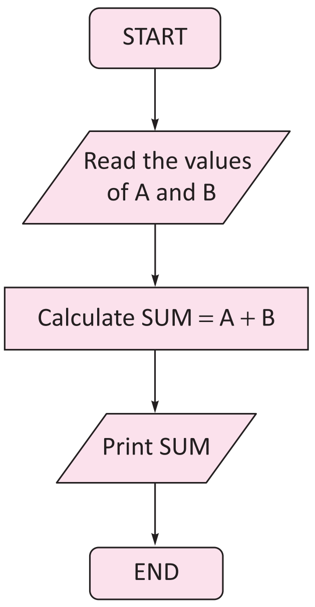
**Example 8.6** Draw a flowchart to calculate the sum of the first 10 natural numbers.

*Solution*



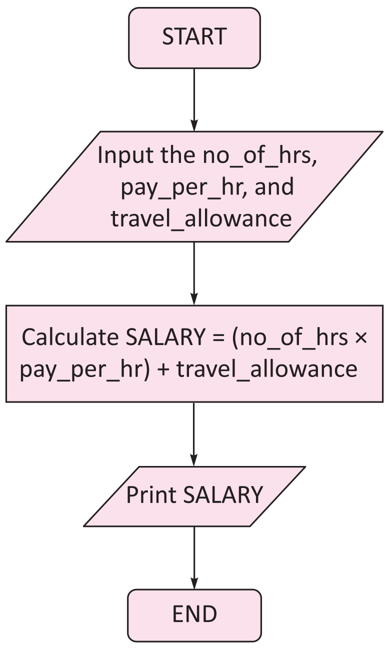
**Example 8.7** Draw a flowchart to add two numbers.

*Solution*



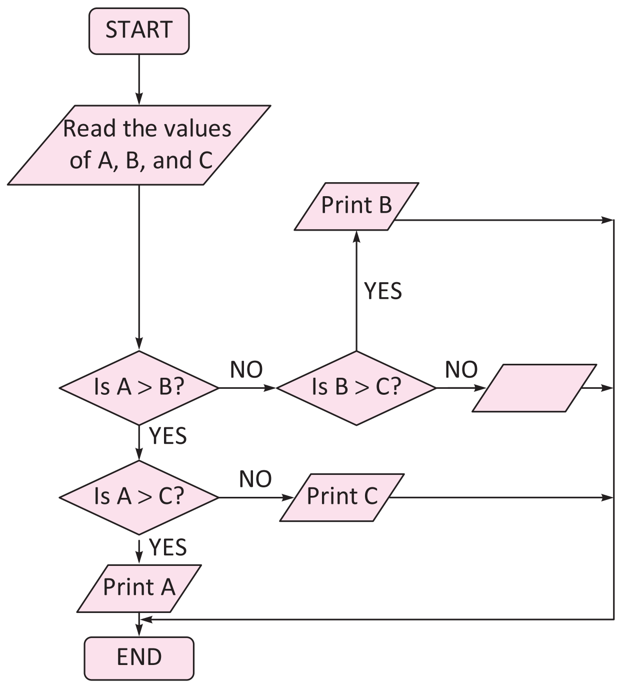
**Example 8.8** Draw a flowchart to calculate the salary of a daily wager.

*Solution*


START
Input the no_of_hrs, pay_per_hr, and travel_allowance
Calculate SALARY = (no_of_hrs × pay_per_hr) + travel_allowance
Print SALARY
END

**Example 8.9** Draw a flowchart to determine the largest of three numbers.

*Solution*



## 8.4.2 Advantages

- Flowcharts are very good communication tools to explain the logic of a system to all concerned. They help to analyse the problem in a more effective manner.
- They are also used for program documentation. They are even more helpful in the case of complex programs.

- They act as a guide or blueprint for the programmers to code the solution in any programming language. They direct the programmers to go from the starting point of the program to the ending point without missing any step in between. This results in error-free programs.
- They can be used to debug programs that have error(s). They help the programmers to easily detect, locate, and remove mistakes in the program in a systematic manner.

### 8.4.3 Limitations

- Drawing flowcharts is a laborious and time-consuming activity. Just imagine the effort required to draw a flowchart of a program having 50,000 statements in it!
- Often, the flowchart of a complex program becomes complex and clumsy.
- At times, a little bit of alteration in the solution may require complete re-drawing of the flowchart.
- The essentials of what is done may get lost in the technical details of how it is done.
- There are no well-defined standards that limit the details that must be incorporated into a flowchart.

## 8.5 PSEUDOCODE

Pseudocode is a form of structured English that describes algorithms. It facilitates designers to focus on the logic of the algorithm without getting bogged down by the details of language syntax. An ideal pseudocode must be complete, describing the entire logic of the algorithm, so that it can be translated straightaway into a programming language.

Pseudocode is a compact and informal high-level description of an algorithm that uses the structural conventions of a programming language. It is basically meant for human reading rather than machine reading, so it omits the details that are not essential for humans. Such details include variable declarations, system-specific code, and subroutines.

Pseudocodes are an outline of a program that can easily be converted into programming statements. They consist of short English phrases that explain specific tasks within a program's algorithm. They should not include keywords in any specific computer language.

The sole purpose of pseudocodes is to enhance human understandability of the solution. They are commonly used in textbooks and scientific publications for documenting algorithms, and for sketching out the program structure before the actual coding is done. This helps even non-programmers to understand the logic of the designed solution. There are no standards defined for writing a pseudocode, because a pseudocode is not an executable program. Flowcharts can be considered as graphical alternatives to pseudocodes, but require more space on paper.

### 8.5.1 Keywords Used while Writing Pseudocodes

For looping and selection, the designer must include the keywords `Do While ... EndDo; Do Until ... EndDo; Case ... EndCase; If ... EndIf; Call ... with (parameters); Call; Return ...; Return; When`, and so on.

#### Parts of Pseudocodes

- Consider the following part of a pseudocode:

```
IF condition THEN
   sequence 1
ELSE
   sequence 2
ENDIF
```

Here, the `ELSE` keyword and `sequence 2` are optional. If the condition is true, `sequence 1` is performed, otherwise `sequence 2` is performed. The following is an example of this construct:

```
IF age >= 18 THEN
   Display Eligible to vote
ELSE
   Display Not Eligible
ENDIF
```

- The `WHILE` construct specifies a loop that tests a condition at the top. The loop is entered only if the condition is true. After each iteration, the condition will be tested, and the loop will continue as long as the condition is true. The beginning and end of the loop are indicated by the keywords `WHILE` and `ENDWHILE`. The general form is

```
WHILE condition
   sequence
ENDWHILE
```

An example of this construct is as follows:

```
WHILE i < 10
   Print i
   Increment i
ENDWHILE
```

- A `CASE` construct indicates a multi-way branch based on conditions that are mutually exclusive. The pseudocode must include keywords such as `CASE`, `OF`, `OTHERS`, and `ENDCASE`. The general form is

```
CASE expression OF
condition 1 : sequence 1
condition 2 : sequence 2
...
condition n : sequence n
OTHERS:
default sequence
ENDCASE
```

Here, the keyword `OTHERS` is optional and specifies the default sequence. The following is an example of the `CASE` construct

```
CASE day OF
         0 : print Sunday
         1 : print Monday
         2 : print Tuesday
         3 : print Wednesday
         4 : print Thursday
         5 : print Friday
         6 : print Saturday
ENDCASE
```

- The `REPEAT` construct is similar to the `WHILE` loop, except that the test is performed at the end of the loop. The keywords used are `REPEAT` and `UNTIL`. The general form is

```
REPEAT
   sequence
UNTIL condition
```

Here, `sequence` will be performed at least once as the test is performed after it is executed. After each iteration, the condition is evaluated, and the loop repeats if the condition is false.

- The `FOR` loop is used for iterating a sequence for a specific number of times. The keywords used are `FOR` and `ENDFOR`. The general form is

```
FOR iteration bounds
   sequence
ENDFOR
```

The following code illustrates a `FOR` loop:

```
FOR each student in the class
   Add 10 as bonus marks
ENDFOR
```

**Example 8.10** Write a pseudocode for calculating the price of a product after adding sales tax to its original price.

*Solution*

```
1. Read the price of the product
2. Read the sales tax rate
3. Calculate sales tax = price of the item × sales
   tax rate
4. Calculate total price = price of the product +
   sales tax
5. Print total price
6. End
Variables: price of the product, sales tax rate, sales
   tax, total price
```

**Example 8.11** Write a pseudocode to calculate the weekly wages of an employee. The pay depends on wages per hour and the number of hours worked. Moreover, if the employee has worked for more than 30 hours, then he or she gets twice the wages per hour, for every extra hour that he or she has worked.

*Solution*

```
1. Read hours worked
2. Read wages per hour
3. Set overtime charges to 0
4. Set overtime hrs to 0
5. IF hours worked > 30 then
      a. Calculate overtime hrs = hours worked – 30
      b. Calculate overtime charges = overtime hrs ×
         (2 × wages per hour)
      c. Set hours worked = hours worked – overtime
         hrs
   ENDIF
6. Calculate salary = (hours worked × wages per hour)
   + overtime charges
7. Display salary
8. End
Variables: hours worked, wages per hour, overtime
      charges, overtime hrs, salary
```

**Example 8.12** Write a pseudocode to read the marks of 10 students. If marks is greater than 50, the student passes, else the student fails. Count the number of students who pass and the number who fail.

*Solution*

```
1. Set pass to 0
2. Set fail to 0
3. Set no of students to 0
4. WHILE no of students < 10
      a. input the marks
      b. IF marks >= 50 then
            Set pass = pass + 1
         ELSE
            Set fail = fail + 1
         ENDIF
   ENDWHILE
5. End
Variables: pass, fail, no of students, marks
```

## 8.6 PROGRAMMING LANGUAGES

A programming language is a language specifically designed to express computations that can be performed by a computer. Programming languages are used to create programs that control the behaviour of a system, to express algorithms, or as a mode of human communication.

Usually, programming languages have a vocabulary of syntax and semantics for instructing a computer to perform specific tasks. The term *programming language* refers to high-level languages such as BASIC (Beginners' All-purpose Symbolic Instruction Code), C, C++, COBOL (COmmon Business Oriented Language), FORTRAN (FORmula TRANslator), Ada, and Pascal, to name a few. Each of these languages has a unique set of keywords (words that it understands) and a special syntax for organizing program instructions.

Though high-level programming languages are easy for humans to read and understand, the computer can understand only machine language, which consists of only numbers. Each type of central processing unit (CPU) has its own unique machine language.

In between machine languages and high-level languages, there is another type of language known as assembly language. Assembly languages are similar to machine languages, but they are much easier to program because they allow a programmer to substitute names for numbers.

However, irrespective of the language that a programmer uses, a program written using any programming language has to be converted into machine language so that the computer can understand it. There are two ways to do this: *compile* the program or *interpret* the program.

The language chosen to write a program depends on the following factors:

- The type of computer on which the program is to be executed
- The type of program
- The expertise of the programmer

For example, FORTRAN is a particularly good language for processing numerical data, but it does not lend itself very well to organizing large programs. Pascal can be used for writing well-structured and readable programs, but it is not as flexible as the C programming language. C++ goes one step ahead of C by incorporating powerful object-oriented features, but it is complex and difficult to learn.

## 8.7 GENERATIONS OF PROGRAMMING LANGUAGES

We now know that programming languages are the primary tools for creating software. As of now, hundreds of programming languages exist in the market, some more used than others and each claiming to be the best. However, in the 1940s when computers were being developed, there was just one language—machine language.

The concept of generations of programming languages (also known as levels) is closely connected to the advances in technology. The five generations of programming languages include machine language, assembly language, high-level language (also known as the third generation language or 3GL), very high-level language (also known as the fourth generation language or 4GL), and fifth generation language that includes artificial intelligence.

### 8.7.1 First Generation: Machine Language

Machine language was used to program the first stored-program computer systems. This is the lowest level of programming language and is the only language that a computer understands. All the commands and data values are expressed using 0s and 1s, corresponding to the *off* and *on* electrical states in a computer.

In the 1950s, each computer had its own native language, and programmers had primitive systems for combining numbers to represent instructions such as *add* and *subtract*. Although there were similarities between each of the machine languages, a computer could not understand programs written in another machine language.

**MACHINE LANGUAGE**

This is an example of a machine language program that will add two numbers and find their average. It is in hexadecimal notation instead of binary notation because that is how the computer presented the code to the programmer. The program was run on a VAX/VMS computer, a product of the Digital Equipment Corporation.

```
                                          000 0000A   0000
                                          000 0000F   0008
                                          000 0000B   0008
                                                      0008
                                                      0058
                                                      00Θ0
FF55    CF    FF54    CF    FF53    CF    C1          00A9
        FF24    CF    FF27    CF    D2    C7          00CC
                                                      00E4
                                                      010D
                                                      013D
```

In machine language, all instructions, memory locations, numbers, and characters are represented in strings of 0s and 1s. Although machine language programs are typically displayed with the *binary* numbers represented in *octal* (base 8) or *hexadecimal* (base 16) number systems, these programs are not easy for humans to read, write, or debug.

The main advantage of machine language is that the execution of the code is very fast and efficient since it is directly executed by the CPU. However, on the downside, machine language is difficult to learn and is far more difficult to edit if errors occur. Moreover, if we want to store some instructions in the memory at some location, then all the instructions after the insertion point would have to be moved down to make room in the memory to accommodate the new instructions. In addition, the code written in machine language is not portable, and to transfer the code to a different computer, it needs to be completely rewritten since the machine language for one computer could be significantly different from that for another computer. Architectural considerations make portability a tough issue to resolve. Table 8.1 lists the advantages and disadvantages of machine language.

Table 8.1 Advantages and disadvantages of machine language

| Advantages | Disadvantages |
|---|---|
| • Code can be directly executed by the computer.<br>• Execution is fast and efficient.<br>• Programs can be written to efficiently utilize memory. | • Code is difficult to write.<br>• Code is difficult to understand by other people.<br>• Code is difficult to maintain.<br>• There is more possibility for errors to creep in.<br>• It is difficult to detect and correct errors.<br>• Code is machine dependent and thus non-portable. |

## 8.7.2 Second Generation: Assembly Language

Second-generation programming languages (2GLs) comprise the assembly languages. Assembly languages are symbolic programming languages that use symbolic notations to represent machine language instructions. These languages are closely connected to machine language and the internal architecture of the computer system on which they are used. Since it is close to machine language, assembly language is also a low-level language. Nearly all computer systems have an assembly language available for use.

Assembly language developed in the mid-1950s was a great leap forward. It used symbolic codes, also known as *mnemonic* codes, which are easy-to-remember abbreviations, rather than numbers. Examples of these codes include ADD for add, CMP for compare, and MUL for multiply.

Assembly language programs consist of a series of individual statements or instructions to instruct the computer what to do. Basically, an assembly language statement consists of a label, an operation code, and one or more *operands*.

Labels are used to identify and refer instructions in the program. The operation code (opcode) is a mnemonic that specifies the operation to be performed, such as *move*, *add*, *subtract*, or *compare*. The operand specifies the register or the location in the main memory where the data to be processed is located.

However, like machine language, the statement or instruction in assembly language will vary from machine to machine, because the language is directly related to the internal architecture of the computer and is not designed to be machine independent. This makes the code written in assembly language less portable, as the code written to be executed on one machine will not run on machines from a different, or sometimes even the same manufacturer.

Nevertheless, the code written in assembly language will be very efficient in terms of execution time and main memory usage, as the language is similar to computer language.

Programs written in assembly language need a translator, often known as the assembler, to convert them into machine language. This is because the computer will understand only the language of 0s and 1s. It will not understand mnemonics such as ADD and SUB.

The following instructions are part of an assembly language code to illustrate addition of two numbers:

```
MOV AX,4     Stores the value 4 in the AX register of the CPU
MOV BX,6     Stores the value 6 in the BX register of the CPU
ADD AX,BX    Adds the contents of the AX and BX registers and
             stores the result in the AX register
```

Although it is much easier to work with assembly language than with machine language, it still requires the programmer to think on the machine's level. Even today, some programmers use assembly language to write those parts of applications where speed of execution is critical; for example, video games, but most programmers have switched to 3GL or 4GL even to write such codes.

Table 8.2 lists the advantages and disadvantages of using assembly language.

Table 8.2 Advantages and disadvantages of assembly language

| Advantages | Disadvantages |
|---|---|
| • It is easy to understand.<br>• It is easier to write programs in assembly language than in machine language.<br>• It is easy to detect and correct errors.<br>• It is easy to modify.<br>• It is less prone to errors. | • Code is machine dependent and thus non-portable.<br>• Programmers must have a good knowledge of the hardware and internal architecture of the CPU.<br>• The code cannot be directly executed by the computer. |

### Assembler

Since computers can execute only codes written in machine language, a special program, called the assembler, is required to convert the code written in assembly language into an equivalent code in machine language, which contains only 0s and 1s. The working of an assembler is shown in Figure 8.5; it can be seen that the assembler takes an assembly language

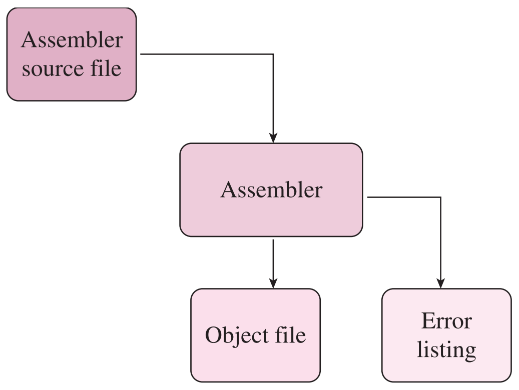


**Fig. 8.5** Assembler

program as input and gives a code in machine language (also called object program) as output. There is a one-to-one correspondence between the assembly language code and the machine language code. However, if there is an error, the assembler gives a list of errors. The object file is created only when the assembly language code is free from errors. The object file can be executed as and when required.

Note An assembler only translates an assembly program into machine language, the result of which is an object file that can be executed. However, the assembler itself does not execute the object file.

### 8.7.3 Third Generation: High-level Language

*Third-generation programming languages* are a refinement of 2GLs. The second generation brought logical structure to software. The third generation was introduced to make the languages more programmer friendly.

The 3GLs spurred the great increase in data processing that occurred in the 1960s and 1970s. In these languages, the program statements are not closely related to the internal characteristics of the computer. Hence, these languages are often referred to as high-level languages.

In general, a statement written in a high-level programming language will expand into several machine language instructions. This is in contrast to assembly languages, where one statement would generate one machine language instruction. 3GLs made programming easier, efficient, and less prone to errors.

High-level languages fall somewhere between natural languages and machine languages. 3GLs include FORTRAN and COBOL, which made it possible for scientists and entrepreneurs to write programs using familiar terms instead of obscure machine instructions.

The widespread use of high-level languages in the early 1960s changed programming into something quite different from what it had been. Programs were written in languages that were more English-like, making them more convenient to use and giving the programmer more time to address a client's problems.

Although 3GLs relieve the programmer of demanding details, they do not provide the flexibility available in low-level languages. However, a few high-level languages such as C and FORTH combine some of the flexibility of assembly languages with the power of high-level languages, but these languages are not well suited to programmers at the beginner level.

Some high-level languages were specifically designed to serve a specific purpose (such as controlling industrial robots or creating graphics), whereas other languages were flexible and considered to be general purpose. Most programmers preferred to use general-purpose high-level languages such as BASIC, FORTRAN, Pascal, COBOL, C++, or Java to write the code for their applications.

Again, a translator is needed to translate the instructions written in a high-level language into the computer-executable machine language. Such translators are commonly known as interpreters and compilers. Each high-level language has many compilers, and there is one for each type of computer.

For example, the machine language generated by one computer's C compiler is not the same as the machine language of some other computer. Therefore, it is necessary to have a C compiler for each type of computer on which the C programs are to be executed.

The 3GLs make it easy to write and debug a program and give a programmer more time to think about its overall logic. Programs written in such languages are portable between machines. For example, a program written in standard C can be compiled and executed on any computer that has a standard C compiler.

Table 8.3 provides the advantages and disadvantages of 3GLs.

**Table 8.3** Advantages and disadvantages of 3GLs

| Advantages | Disadvantages |
|---|---|
| • The code is machine independent.<br>• It is easy to learn and use the language.<br>• There are few errors.<br>• It is easy to document and understand the code.<br>• It is easy to maintain the code.<br>• It is easy to detect and correct errors. | • Code may not be optimized.<br>• The code is less efficient.<br>• It is difficult to write a code that controls the CPU, memory, and registers. |

#### Compiler

A compiler is a special type of program that transforms the source code written in a programming language (the *source language*) into machine language, which uses only two digits—0 and 1 (the *target language*). The resultant code in 0s and 1s is known as the object *code*. The object code is used to create an executable program.

## HOW COMPILERS WORK

Compilers, like other programs, reside on the secondary storage. To translate a source code into its equivalent machine language code, the computer first loads the compiler and the source program from the secondary memory into the main memory. It then executes the compiler along with the source program as its input. The output of this execution is the object file, which is also stored in the secondary storage. Whenever the program is to be executed, the computer loads the object file into the memory and executes it. Thus, it is not necessary to compile the program every time it needs to be executed. Compilation will be needed again only if the source code is modified.

Therefore, a compiler (Figure 8.6) is used to translate the source code from a high-level programming language to a lower-level language (e.g., assembly language or machine code). There is a one-to-one correspondence between the high-level language code and machine language code generated by the compiler.

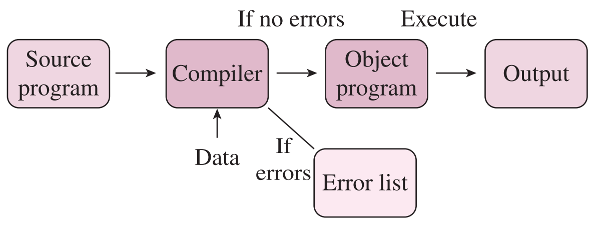


**Fig. 8.6** Compiler

If the source code contains errors, then the compiler will not be able to do its intended task. Errors that limit the compiler in understanding a program are called *syntax errors*. Examples of syntax errors are spelling mistakes, typing mistakes, illegal characters, and use of undefined variables. The other type of error is the logical error, which occurs when the program does not function accurately. Logical errors are much harder to locate and correct than syntax errors. Whenever errors are detected in the source code, the compiler generates a list of error messages indicating the type of error and the line in which the error has occurred. The programmer makes use of this error list to correct the source code.

The work of a compiler is only to translate the human-readable source code into a computer-executable machine code. It can locate syntax errors in the program (if any) but cannot fix it. Unless the syntactical error is rectified, the source code cannot be converted into the object code.

Each high-level language has a separate compiler. A compiler can translate a program in one particular high-level language into machine language. For a program written in some other programming language, a compiler for that specific language is needed.

### Interpreter

Like the compiler, the interpreter executes instructions written in a high-level language. Basically, a program written in a high-level language can be executed in any of the two ways—by compiling the program or by passing the program through an interpreter.

The compiler translates instructions written in a high-level programming language directly into machine language; the interpreter, on the other hand, translates the instructions into an intermediate form, which it then executes. The interpreter takes one statement of high-level code, translates it into the machine level code, executes it, and then takes the next statement and repeats the process until the entire program is translated.

Note An interpreter not only translates the code into machine language but also executes it.

Figure 8.7 shows an interpreter that takes a source program as its input and gives the output. This is in contrast with the compiler, which produces an object file as the output of the compilation process. Usually, a compiled program executes faster than an interpreted program. Moreover, since there is no object file saved for future use, users will have to reinterpret the entire program each time they want to execute the code.

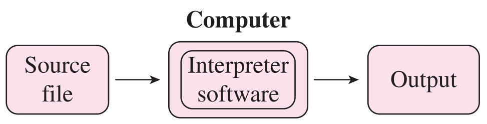


**Fig. 8.7** Interpreter

Overall, compilers and interpreters both achieve similar purposes, but they are inherently different as to how they achieve that purpose. The differences between compilers and interpreters are given in Table 8.4.

### Linker

Software development in the real world usually follows a modular approach (discussed in Section 8.8.2). In this approach, a program is divided into various (smaller) modules as it is easy to code, edit, debug, test, document, and maintain them. Moreover, a module written for one program can also be used for another program. When a module is compiled, an object file of that module is generated.

**Table 8.4** Differences between compilers and interpreters

| Compiler | Interpreter |
|---|---|
| • It translates the entire program in one go.<br>• It generates error(s) after translating the entire program.<br>• Execution of code is faster.<br>• An object file is generated.<br>• Code need not be recompiled every time it is executed.<br>• It merely translates the code.<br>• It requires more memory space (to save the object file). | • It interprets and executes one statement at a time.<br>• It stops translation after getting the first error.<br>• Execution of code is slower as every time reinterpretation of statements has to be done.<br>• No object file is generated.<br>• Code has to be reinterpreted every time it is executed.<br>• It translates as well as executes the code.<br>• It requires less memory space (no object file). |

Once the modules are coded and tested, the object files of all the modules are combined together to form the final executable file. Therefore, a linker, also called a *link editor* or *binder*, is a program that combines the object modules to form an executable program (see Figure 8.8). Usually, the compiler automatically invokes the linker as the last step in compiling a program.

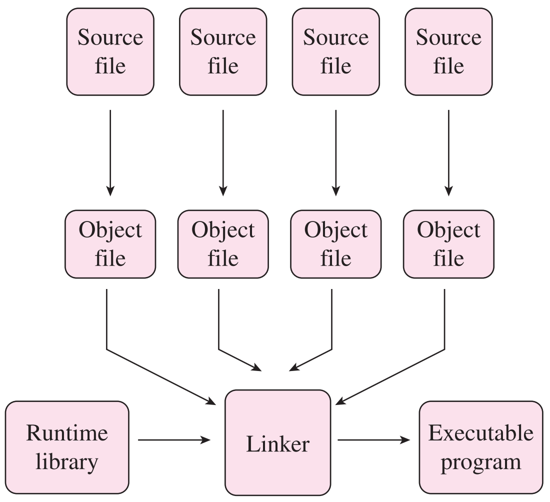


**Fig. 8.8** Linker

### Loader

A loader is a special type of program that copies programs from a storage device to the main memory, where they can be executed. Most loaders are transparent to the users.

Note: Assemblers, linkers, compilers, loaders, and interpreters are all system software.

### 8.7.4 Fourth Generation: Very High-level Languages

With each generation, programming languages started becoming easier to use and more similar to natural languages. 4GLs are a little different from their prior generation because they are non-procedural. While writing a code using a procedural language, the programmer has to tell the computer how a task is done—add this, compare that, do this if the condition is true, and so on—in a very specific step-by-step manner. In striking contrast, while using a non-procedural language, programmers define what they want the computer to do but they do not supply all the details of how it has to be done.

Although there is no standard rule that defines a 4GL, certain characteristics of such languages include the following:

- The instructions of the code are written in English-like sentences.
- They are non-procedural, so users concentrate on the 'what' instead of the 'how' aspect of the task.
- The code written in a 4GL is easy to maintain.
- The code written in a 4GL enhances the productivity of programmers, as they have to type fewer lines of code to get something done. A programmer supposedly becomes 10 times more productive when he/she writes the code using a 4GL than using a 3GL.

A typical example of a 4GL is the query language, which allows a user to request information from a database with precisely worded English-like sentences. A query language is used as a database user interface and hides the specific details of the database from the user. For example, when working with Structured Query Language (SQL), the programmer just needs to remember a few rules of *syntax* and *logic*, and therefore, it is easier to learn than COBOL or C.

Let us take an example in which a report needs to be generated. The report displays the total number of students enrolled in each class and in each semester. Using a 4GL, the request would look similar to the following:

```
TABLE FILE ENROLMENT
SUM STUDENTS BY SEMESTER BY CLASS
```

Thus, we see that a 4GL is very simple to learn and work with. The same task if written in C or any other 3GL would require multiple lines of code.

The 4GLs are still evolving, which makes it difficult to define or standardize them. The only downside of a 4GL is that it does not make efficient use of a machine's resources. However, the benefit of executing a program quickly and easily far outweighs the extra costs of running it.

### 8.7.5 Fifth-generation Programming Language

Fifth-generation programming languages (5GLs) are centred on solving problems using the constraints given to a program rather than using an algorithm written by a programmer. Most constraint-based and logic programming languages and some declarative languages form a part of the 5GLS. These languages are widely used in artificial intelligence research. Another aspect of a 5GL is that it contains visual tools to help develop a program. Typical examples of 5GLs include Prolog, OPS5, Mercury, and Visual Basic.

Thus, taking a forward leap, 5GLs are designed to make the computer solve a given problem without the programmer. While working with a 4GL, programmers have to write a specific code to do a work, but with a 5GL, they only have to worry about what problems need to be solved and what conditions need to be met, without worrying about how to implement a routine or an algorithm to solve them.

In general, 5GLs were generally built upon LISP, many originating on the LISP machine, such as ICAD. There are also many frame languages, such as KL-ONE.

In the 1990s, 5GLs were considered the wave of the future, and some predicted that they would replace all other languages for system development (except the low-level languages). During the period ranging from 1982 to 1993, Japan carried out extensive research on and invested a large amount of money into their fifth-generation computer systems project, hoping to design a massive computer network of machines using these tools. However, when large programs were built, the flaws of the approach became more apparent. Researchers began to observe that given a set of constraints defining a particular problem, deriving an efficient algorithm to solve it is itself a very difficult problem. All factors could not be automated and some still require the insight of a programmer.

However, today the fifth-generation languages are pursued as a possible level of computer language. Software vendors across the globe currently claim that their software meets the visual 'programming' requirements of the 5GL concept.

## 8.8 CATEGORIZATION OF HIGH-LEVEL LANGUAGES

High-level languages can be easily categorized into four groups based on the programming paradigm supported by them (refer Figure 8.9). A programming paradigm refers to the approach the programming language has employed for solving different types of problems.

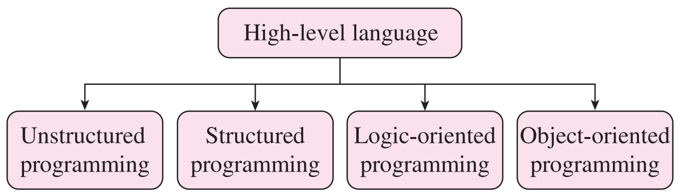


**Fig. 8.9** Categorization of high-level languages

### 8.8.1 Unstructured Programming

In unstructured programming, programmers write small and simple programs consisting only of one main program. Here, `main()` consists of *statements* that modify the data that is *global* throughout the whole program (Figure 8.10). Though this technique is simple, it is not good for writing large programs. For example, if we need to perform a particular task multiple times in the program, then we need to copy the same sequence of statements at different locations within the program. This led to the idea of writing functions or procedures. The new technique of using procedures came to be known as procedural programming.

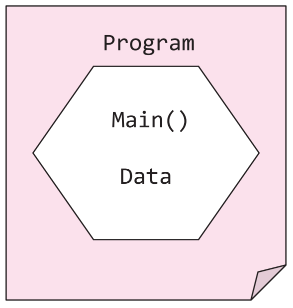


**Fig. 8.10** Unstructured programming

### 8.8.2 Structured Programming Language

The concept of structured programming, also referred to as *modular programming*, was first suggested by the mathematicians Corrado Böhm and Giuseppe Jacopini. It is basically a subset of procedural programming that enforces a logical structure on the program to make it efficient and easy to understand and modify.

Structured programming employs a top-down approach in which the overall program structure is broken down into separate modules. This allows the code to be efficiently loaded into the memory and to be reused in other programs. Modules are coded separately, and once a module is written and tested individually, it is then integrated with the other modules to form the overall program structure.

Structured programming is therefore based on modularization, which groups related statements together (modules). Modularization makes it easy to write, debug, and understand a program. Ideally, modules should not be longer than a page. It is always easy to understand a series of 10 single-page modules than a single 10-page program.

For some large and complex programs, the overall program structure may further require the modules to be broken into subsidiary modules. This process continues until an individual pieces of code can be written easily.

Almost any language can use structured programming techniques to avoid the common pitfalls of unstructured languages. Unstructured programs depend on the programmer's skills to avoid structural problems and are therefore poorly organized. Most modern procedural languages support the concept of structured programming. Even object-oriented programming can be considered a type of structured programming because it uses the techniques of structured programming for program flow and adds more structure for data to the model.

In structured programming, the program flow follows a simple sequence and usually avoids the use of `goto` statements.

Besides sequential flow, structured programming also supports selection and repetition. *Selection* allows for choosing any one of a number of statements to execute based on the current status of the program. Selection statements contain keywords such as `if`, `then`, `endif`, and `switch` to help identify the order as a logical executable. In *repetition*, a selected statement remains active until the program reaches a point where there is a need for some other action to take place. It includes keywords such as `repeat`, `for`, and `do until`. Essentially, repetition instructs the program how long it needs to continue the function before requesting further instructions.

### Advantages

The following are the advantages of structured programming:

- The goal of structured programming is to write correct programs that are easy to understand and modify.
- Modules enhance the programmers' productivity by allowing them to look at the big picture first and then focus on details later.
- With modules, many programmers can work on a single large program, with each working on a different module.
- A structured program can be written in less time than an unstructured program. Modules or procedures written for one program can be reused in other programs as well.
- A structured program is easy to debug. This is because each procedure in a structured program is specialized to perform just one task, and therefore, every procedure can be checked individually for the presence of any error. In contrast, unstructured programs consist of a sequence of instructions that are not grouped for specific tasks. Their logic is cluttered with details and is therefore difficult to follow.
- Individual procedures are easy to change as well as understand. In a structured program, every procedure has meaningful names and has clear documentation to identify the task performed by it. Moreover, a correctly written structured program is self-documenting and can be easily understood by another programmer.

**Example 8.13** Create a program to manage a name and address list of students of your institute.

*Solution*

To perform this task, you will have to first break down the program into modules that perform the following functions:

- Enter new names and addresses.
- Modify existing entries.
- Sort entries.
- Print the list.

Now, each of these modules can be further broken down into smaller modules. For example, 'Enter new names and addresses' module can be subdivided into modules that perform the following tasks:

- Prompt the user to enter new data.
- Read the existing list from the disk.
- Add the name and address to the existing list.
- Save the updated list to the disk.

Similarly, 'Modify existing entries' module can be further divided into modules that perform the following tasks:

- Read the existing list from the disk.
- Modify one or more entries.
- Save the updated list to the disk.

Observe that two submodules—'Read the existing list from the disk' and 'Save the updated list to the disk'—are common to both the modules. Hence, once these submodules are written, they can be used by all the modules that require the same task to be performed. Structured programming method results in a hierarchical or layered program structure, which is depicted in Figure 8.11.

## 8.8.3 Logic-oriented Programming Language

Logic-oriented programming languages employ a programming paradigm that is based on formal predicate logic. The logic paradigm is remarkably different from other programming paradigms. The predicate logic describes the nature of a problem by defining the relationships between rules and facts. These rules together with an inference algorithm form a program. Prolog, LISP, and Datalog are few examples of logic-oriented programming languages.

A logical sentence in a logic program is given in the form of

```
p(X, Y) if q(X) and r(Y)
```

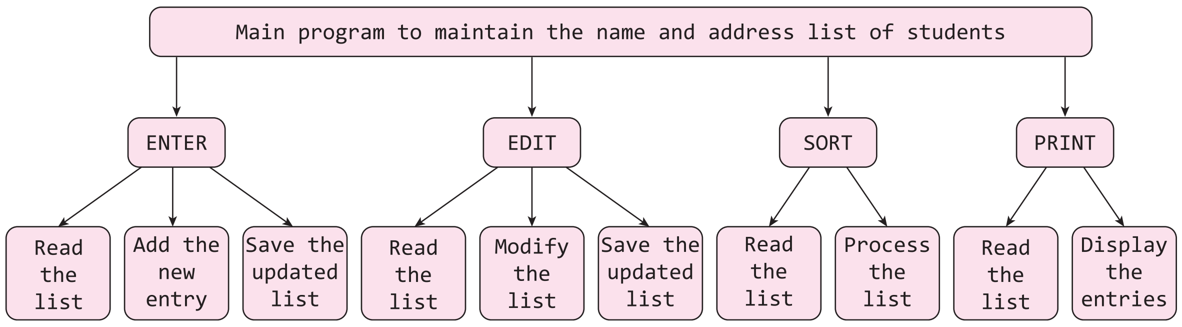


**Fig. 8.11** Layered program structure

For example, the `sister` relation, which can be defined using other simpler relations and properties such as `father`, `mother`, `female`, is shown in Figure 8.12. According to the figure, *A* can be a sister of *B* if both *A* and *B* have the same father and mother and *A* is a female.

```
Sister(A, B):  /* A is the sister of B if there
               are two people F and M for which */
Father(F, A),  /* F is the father of A */
Father(F, B),  /* F is the father of B */
Mother(M, A),  /* M is the mother of A */
Mother(M, A),  /* M is the mother of B */
Female(A)      /* A is a female */
```

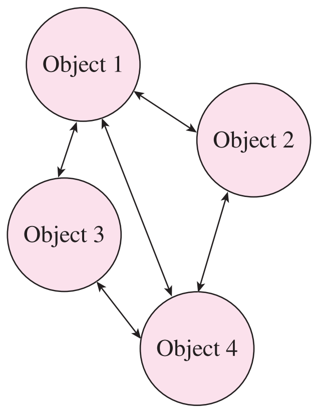
**Fig. 8.12** Predicate logic that defines a sister relationship

We have already seen that mathematical (or Boolean) logic plays a vital role in the design of logic circuits, which form the basis of computer systems. Therefore, in logic-oriented programming languages, a more advanced construct called predicate logic is used. In these languages, different logical assertions about a situation are made, establishing all known facts. Then, queries are made and any deducible solution to the query is returned as output. The role of a computer in this paradigm is thus to maintain data and make logical deductions.

There are two main advantages of using logic-oriented programming. First, the computer solves the problem so that the programs are of minimum statements and complexity. Second, the validity of the solution can be easily proved.

### 8.8.4 Object-oriented Programming

We have seen that unstructured and structured programming paradigms are task-based paradigms as they focus on the actions the software should accomplish. However, the object-oriented paradigm is both task-based and data-based. In this paradigm, all relevant data and tasks are grouped together in entities known as *objects* (Figure 8.13).



Objects of a program interact by sending messages to each other.

**Fig. 8.13** Object-oriented paradigm

For example, consider a list of numbers stored in an array. The procedural or modular programming paradigm considers this list as merely a collection of data. Any program that accesses this list must have some procedures or functions to process this list. If we want to find the largest number or to sort the numbers in the list, we need specific procedures or functions to do the task. Thus, the list is a passive entity as it is maintained by a controlling program rather than having the responsibility of maintaining itself.

However, in the object-oriented paradigm, the list and the associated operations are treated as one entity, known as an *object*. Therefore, in this approach, the list is considered an object consisting of the list together with a collection of routines for manipulating the list. In the list object, there may be routines for adding a number to the list, deleting a number from the list, sorting the list, and so on.

The marked difference between this approach and the traditional approaches is that the program accessing this list does not contain procedures for performing tasks but rather uses the routines provided in the object. In other words, instead of sorting the list as in the procedural paradigm, the program asks the list to sort itself.

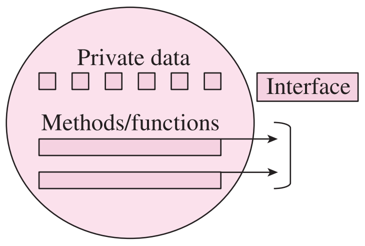


**Fig. 8.14** Object

Thus, we can conclude that the object-oriented paradigm is task-based (as it considers operations) as well as data-based (as these operations are grouped with the relevant data).

Figure 8.14 represents a generic object in the object-oriented paradigm. Every object contains some data as well as the operations, methods, and functions that operate on that data. Some objects contain only basic data types such as characters, integers, and floating types, whereas others incorporate complex data types such as trees or graphs.

Programs that need an object will access the object's methods through a specific interface. The interface specifies how to send a message to the object, that is, a request for a certain operation to be performed.

For example, the interface for a list object may require that any message for adding a new number to the list should include the number to be added. Similarly, the interface might also require that any message for sorting specifies whether the sort should be in ascending or descending order. Hence, an interface specifies how messages can be sent to the object.

Note Object-oriented programming is used for simulating real-world problems on computers because the real world is made of objects.

## Concepts of Object-oriented Programming

An object-oriented language must support mechanisms to define, create, store, and manipulate objects and allow communication between the objects. In this section, we will read about the underlying concepts of object-oriented programming that support the objects. These concepts include the following:

- Class
- Object
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

**Class** A class is used to describe real-world things, such as occurrences, things, and external entities. It provides a template or a blueprint that describes the structure and behaviour of a set of similar objects. Once a class is defined, a specific instance of the class can be easily created. For example, consider a class *student*. A student has attributes such as roll number, name, course, and aggregate marks. The operations that can be performed on the student data may include get_details, set_details, and edit_details. Thus, we can say that a class describes one or more similar objects.

Note that this data and the set of operations that we have mentioned can be applied to all students in the class. When we create an instance of student, we are actually creating an object of the student class.

Note Classes define the properties and behaviour of objects.

**Object** In the previous paragraph, we have taken an example of the student class and said that a class is used to create instances, which are known as objects. Therefore, if student is a class, then all the 60 students in a course (assuming there are maximum 60 students in a particular course) are objects of the student class. Therefore, all students are objects of the class. Thus, a class can have multiple instances.

Every object contains some data and procedures (also called methods). It stores data in variables and responds to messages that it receives from other objects by executing its methods (procedures).

Every object of a class has its own set of values. Therefore, two distinct objects can have the same set of values. In general, the set of values that an object takes at a particular time is known as the *state* of the object.

As mentioned earlier, a class is a combination of properties (data) and methods (functions). The state of an object can be changed by applying a particular method. The possible sequence of state changes of an object is known as the *behaviour* of the object. In other words, the behaviour of an object is defined by the set of methods that can be applied to it.

It is important to understand the difference between classes and objects. Simply put, a class is merely a category of similar objects. All objects of the same class have the same range of potential states and behaviour.

Note An *object* is an instance of a class that can be uniquely identified by its *name*. Every object has a *state*, which is given by the values of its attributes at a particular time.

**Method and messages** A method is a function associated with a class. It defines the operations that an object of the class can execute when it receives a message. In an object-oriented language, only the methods of a class can access and manipulate the data stored in an instance of the class (or object). Objects can communicate with each other through messages. An object asks another object to invoke one of its methods by sending it a *message*. Consider Figure 8.15 in which the sender object sends a message to the receiver object to get the details of a student. In reply to the message, the receiver sends the results of the execution to the sender.

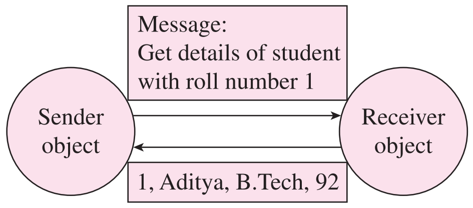


**Fig. 8.15** Objects sending messages

In Figure 8.15, the sender has asked the receiver to send the details of the student having roll number 1. Thus, the sender passes some specific information to the receiver so that the receiver can provide precise information to the sender. The data that is transferred with the message is called the *parameter.* Here, roll number 1 is the parameter. Therefore, messages sent to other objects consist of three parts—the receiver object, the name of the method that the receiver should invoke, and the parameters that must be used with the method.

**Inheritance** This is a concept of object-oriented programming in which a new class is created from an existing class. The new class, known as the *subclass* or derived class, inherits the attributes and behaviour of the pre-existing class, which is referred to as the *superclass* or *parent class*. The inheritance relationship of sub- and superclasses generates a hierarchy. A subclass not only has all the states and behaviour associated with the superclass but also has other more specialized traits.

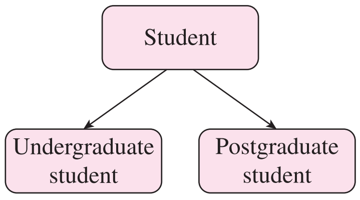


**Fig. 8.16** Inheritance

The main advantage of inheritance is the ability to reuse the code. When we want a specialized class, we do not have to write the entire code for that class from scratch. We can inherit a class from a general class and then add the specialized code for the subclass. For example, assume that we have a class student with the following members:

*Properties*: roll_number, name, course, aggregate_marks
*Methods*: get_details, set_details

We can inherit two classes from the student class, namely undergraduate student and postgraduate student (Figure 8.16). These two classes will have all the properties and methods of the student class, and in addition, they will have even more specialized members.

When a derived class receives a message to execute a method, it first searches for the method in its own class. If it finds that method, then it executes it. If the method is not present, then it searches for it in its superclass. If the method is found, it is executed; otherwise, an error message is reported.

Note A subclass can inherit properties and methods from multiple parent classes. This is called multiple inheritance.

**Polymorphism** Polymorphism means *having several different forms*. It is one of the essential concepts of object-oriented programming. Inheritance is related to classes and their hierarchy, whereas polymorphism is related to methods.

Polymorphism is a concept that enables the programmers to assign a different meaning or usage to a variable, a function, or an object in different contexts. When polymorphism is applied on a variable, the variable with a given name may be allowed to have different forms. The program will then decide which form is to be used at the time of execution. For example, the variable roll_no of the class student may be numeric (numbers alone) or alphanumeric (combination of numbers and letters). The program can be coded to distinguish between the two forms of the variable so that it can be handled in its own way.

Polymorphism can also be applied to a function in such a way that the particular form of function selected for execution varies depending on the parameters given to the function. For example, if the roll number of the student is an integer, then its corresponding function will be executed. In case it consists of alphanumeric characters, then another function with the same name will be executed. This type of polymorphism is called *function overloading*.

Polymorphism can also be applied to operators. For example, we know that operators can be applied only to basic data types that the programming language supports. So, $a + b$ will give the result of adding $a$ and $b$. If $a = 2$ and $b = 3$, then $a + b = 5$. If we overload the + operator to be used with strings, then $str1 + str2$ gives the result $str2$ concatenated with $str1$. Therefore, if $str1$ = 'Oxford' and $str2$ = 'University', then $str1 + str2$ = 'Oxford University'.

**Data abstraction and encapsulation** Data abstraction refers to the process of defining data and functions in such a way that only the essential details are provided to the outside world and the implementation details are hidden. The main focus of data abstraction is to separate the interface and the implementation of a program. For example, we as users of television sets can switch them on or off, change the channel, set the volume, or add external devices such as speakers, CD players, or DVD players without knowing the details about how its functionality is implemented. Thus, the internal implementation is completely hidden from the external world.

Similarly, in object-oriented programming languages, classes provide data abstraction through public methods to the outside world to provide the functionality of the object or to manipulate the object's data. An entity in the outside world will not know about the implementation details of the class or that method.

Data encapsulation, also called data hiding, is the technique of hiding the implementation details of a class from the users. The users are allowed to execute only a restricted set of operations (class methods) on the data members of the class. Therefore, encapsulation organizes the data and methods into a structure that prevents data access by any function (or method) not specified in the class. This ensures the integrity of the data contained in the object.

Encapsulation defines three access levels for data variables and member functions of the class. These access levels specify the access rights.

- Any data or function with access level *public* can be accessed by any function belonging to any class. This is the lowest level of data protection.
- Any data or function with access level *protected* can be accessed only by the class in which it is declared or by any class that is inherited from it.
- Any data or function with access level *private* can be accessed only by the class in which it is declared. This is the highest level of data protection.

## 8.9 SOME POPULAR HIGH-LEVEL LANGUAGES

In this section, we will discuss the features of some popular high-level languages.

### 8.9.1 BASIC

BASIC (Beginner's All-purpose Symbolic Instruction Code) is a general-purpose, high-level programming language developed by John G. Kemeny and Thomas E. Kurtz in 1964. In the 1960s, programming was done only by scientists and mathematicians; hence, BASIC was specifically designed to enable students in fields other than science and mathematics to use computers. It was easy to learn and use and was a very powerful language that was used for a wide range of applications.

BASIC was not only widely used on microcomputers in the 1970s and 1980s but was also shipped with them in the machine's firmware. As it was an easy-to-learn language, it motivated small business owners, professionals, hobbyists, and consultants to develop custom software on the computers they could afford. BASIC was also used in teaching the introductory concepts of programming.

Even today, BASIC is widely being used as Microsoft's Visual Basic, which has added object-oriented programming features and a graphical user interface (GUI) to the standard BASIC. Moreover, these days many variations of BASIC are being used within applications such as Microsoft Word and Microsoft Excel to enable users to write programs to customize and automate these applications.

### 8.9.2 FORTRAN

FORTRAN (Formulas Translation) is one of the oldest general-purpose programming languages. It was developed in 1957 at IBM by a team of programmers led by John Backus. The development of FORTRAN was a remarkable development in the field of programming languages. Previously, programs were written either in machine language or in assembly language. Since software development using low-level programming was cumbersome, Backus wanted to create a machine independent, simple language suitable for a wide variety of applications that combined a form of English shorthand with algebraic equations. The feature of enabling the creation of natural-language programs that ran as efficiently as machine level codes made FORTRAN a popular language in the late 1950s. It was so easy to develop a code in FORTRAN that programmers were able to write programs 500 per cent faster than before with a compromise of execution efficiency of just 20 per cent. It enabled software developers to focus more on the problem-solving aspects of a problem than on the coding aspect.

Note In 1993, John Backus was awarded the National Academy of Engineering's Charles Stark Draper Prize, which is the highest national prize awarded in engineering, for the invention of FORTRAN.

FORTRAN is especially suited to high-performance numeric, scientific, statistical, and engineering computing and is extensively used in areas such as weather prediction, finite element analysis, computational physics, computational chemistry, and computational fluid dynamics. It is also used in programming video games, air traffic control systems, payroll calculations, military applications, factory automation, parallel computer research, storm drainage systems, and design of bridges and airplane structures.

The new versions of FORTRAN have built-in support for features such as structured programming, generic programming, object-oriented programming, concurrent programming, and array programming.

Note FORTRAN is so named because it was specifically designed to allow easy translation of mathematical formulas into code.

### 8.9.3 Pascal

Pascal, which was named after the mathematician Blaise Pascal, is a procedural programming language developed in the late 1960s by Niklaus Wirth. It is a small and efficient language specifically designed to encourage good programming practices using structured programming and data structuring. Although Pascal can be used for technical problems, it was widely used as a teaching language to help people understand the basics of programming.

The structure and syntax of Pascal is similar to that of C. Pascal provided many features that were lacking in other languages then. It enabled programmers to develop well-structured and well-organized programs that were efficient to implement and run. Pascal contains built-in data types (e.g., integers and characters), user-defined data types, and a defined set of data structures (e.g., arrays and files).

With the growing popularity of object-orient programming, a derivative known as Object Pascal was designed and released in 1985. However, despite its success in academia, Pascal found very little success in the business world because of its inflexibility and lack of tools for developing large applications.

### 8.9.4 C

C is a high-level general-purpose programming language that was developed in the 1970s by Dennis Ritchie at Bell Labs. Although it was originally designed for system programming, today C is considered a powerful and flexible language that can be used for a wide range of applications varying from business programs to engineering. The following are some of the reasons that led to the popularity of C even in personal computers:

- It is relatively small and thus requires comparatively less memory than other languages.
- It is easy to understand.
- Codes written in C are easy to maintain.
- Programs written in C are easily portable.
- It is considered closer to assembly language than other high-level languages. Since C supports some low-level features, it is extensively used for writing efficient codes for operating systems, compilers, and so on.
- It supports procedures and modularization of programs.

Note The first major program written in C was the Unix operating system.

### 8.9.5 C++

C++ is a general-purpose programming language developed by Bjarne Stroustrup starting in 1979 at Bell Labs. Like C, C++ is considered an intermediate-level language because it comprises both high-level and low-level language features.

C++ is a very popular programming language and can be implemented on a wide variety of hardware and operating system platforms. It is a powerful language for high-performance applications, including operating systems, system software, application software, device drivers, embedded software, high-performance client and server applications, software engineering, graphics, and games and animation software.

C++ is a superset of the C language. Hence, C++ supports all features of C and also includes other new features such as classes, objects, polymorphism, inheritance, data abstraction, encapsulation, single-line comments using two forward slashes, and strong type checking. C++ is an object-oriented programming language and facilitates design, reuse, and maintenance for complex software. It has an extensive library to enable programmers to reuse existing code. In general, the number of instructions required to perform a task in C++ is comparatively less than that required in other high-level languages. It is easy to write, debug, and modify a code in C++. However, the main drawback of this language is that the C++ compiler does not issue warning or error messages for mistakes such as when an array index is out of range and when an uninitialized variable is used.

### 8.9.6 Java

Java is a general-purpose, object-oriented programming language released by Sun Microsystems in 1995. Though the syntax and semantics of Java is similar to that of C++, it is more powerful than not only C++ but also many other high-level languages. However, unlike C and C++, Java has less support for low-level features.

Programs written in Java are robust, secure, and reliable. Java was the first language to bring animation and interactivity to Internet-based applications. Today, a number of applications and websites will not work unless Java is installed on the computer. Java has marked its presence on a variety of computer systems ranging from laptops to data centres, game consoles to scientific supercomputers, and cell phones to the Internet.

Java is used in the following areas:

- Internet applications such as playing online games and chatting with friends and relatives across the world
- Desktop applications such as viewing images in three dimensions
- Embedded systems applications to be used in devices such as set-top boxes, hand-held devices, and phones
- Intranet applications and other e-business solutions, which have now become an integral part of corporate computing

Note Small Java applications called applets can be downloaded from the Internet and run on the computer by a Java-compatible Web browser such as Netscape Navigator or Microsoft Internet Explorer.

Java programs follow the *write once, run anywhere* (WORA) concept. According to this concept, programmers have to write a code only once and the same code can be executed on any platform without any modification (and thus re-compilation). Basically, Java source code files (having a *.java* extension) are compiled into a *bytecode* (file having a *.class* extension), which can then be executed by a Java interpreter. The already-compiled Java code can run on most computers because Java interpreters and runtime environments, called *Java virtual machines* (JVMs), are available for almost all popular operating systems such as Unix, Mac OS X, and Windows.

Note Java has been simplified to eliminate features that cause common programming errors.

### 8.9.7 LISP

LISP, an acronym for *list processing*, is one of the oldest programming languages still widely used by programmers all over the world. It was developed by John McCarthy in 1959. The main idea behind developing LISP was the need to have a language that could easily manipulate non-numeric data such as symbols and strings of text. LISP's ability to manipulate symbolic expressions rather than numbers makes it convenient for artificial intelligence applications and for simulation of games.

LISP is a functional programming language in which all computations are accomplished by applying functions to arguments. All programs are written as function calls or parenthesized lists or as a list with the function's name followed by arguments. For example, a function *f* taking three arguments can be written as (*f arg1 arg2 arg3*).

Although LISP is not a general-purpose programming language, it has still pioneered many ideas in the field of computer science, including conditionals, tree data structures, higher-order functions, recursion, automatic storage management, dynamic typing, and self-hosting compiler.

The main advantage of using LISP is that even complex functions can be easily written and understood by others. However, the disadvantage of LISP is that it supports neither low-level features nor any object-oriented programming concept. LISP is often used as a scripting language for other applications such as AutoLisp for AutoCAD or for teaching abstract concepts in computer science.

Note LISP was used to develop complex applications such as Emacs editor.

## 8.10 FACTORS AFFECTING SELECTION OF PROGRAMMING LANGUAGE

When planning a software solution, the software development team often faces a common question—which programming language to use? Many programming languages are available today and each one has its own strengths and weaknesses.

C can be used to write an efficient code, whereas a code in BASIC is easy to write and understand; some languages are compiled, whereas others are interpreted; some languages are well known to the programmers, whereas others are completely new. Selecting the perfect language for a particular application at hand is a daunting task. In this section, we will discuss some parameters that influence the selection of a programming language for a project.

**Organizational policies** In the computing industry, most organizations have policies that dictate which computer hardware and software they should use. For example, many organizations have Java as the default programming language.

**Suitability** The programming language must be able to work on the platform being used. In addition, it must have the features to write the application. For example, if Internet applications have to be developed, then Java would be a good choice. If device drivers have to be made, then C would be a better choice.

**Availability of programmers** The choice of programming language also depends on the programmer's experience and expertise. If a language that is new to the programmer is chosen, then it would demand more investment in terms of time and money because either the programmer will have to be trained in the language or some new programmer having a sound knowledge of that language would be hired. In both cases, extra time and money will be required.

**Reliability** Some programming languages have built-in features that support the development of software that is reliable and less prone to crash. A reliable code can withstand even stress conditions. For example, Ada is an object-oriented high-level programming language that had been extended from Pascal and various other languages. It ensures reliability in mission critical applications, for example, in safety critical systems such as the fly-by-wire control system of the Boeing 777 aircraft.

**Development and maintenance costs** Development cost should be considered while choosing a programming language. Some languages support reusable components or an extensive set of standard libraries that makes the code quick and easy to develop and maintain, which results in reduced development cost. Maintaining a program also incurs costs, especially when bugs and errors have to be fixed or the code must be updated to meet the current requirements. A language that is easy to understand generally costs less. Moreover, coding with open-source languages (which are free) is more economical than coding with languages for which licences have to be purchased. This is why PHP applications are more economical than ASP.NET applications.

**Expandability** Software applications that are designed for interactive websites are expected to support a large number of users at the same time without crashing. Hence, the programming language chosen for such applications must be stable and capable enough to support even more than the expected simultaneous users. For example, PHP is a programming language that supports expandability.

**Speed of development** Speed of development is a factor that not only includes the time it takes to write a code but also considers the time taken to find a solution to the problem at hand, time taken to find the bugs, availability of development tools, experience and skill of the programmers, and testing regime.

**Object orientation** In some situations, using object-oriented programming to code a solution is far more beneficial than coding with a traditional language. This is because it not only speeds up the development process because of the existing code that can be reused but also entails the development of classes that can in future be reused while writing other codes.

**Portability** Most of the programming languages are dependent on some hardware constraints. Portability is therefore a serious issue that depends on the underlying platform. Java is an example of a popular language that has good portability. This is because the bytecode generated can run the program on any machine in which a JVM is installed. In contrast, a C or C++ code would run on with two different types of compilers and would thus produce two different types of executable files when run on Linux and Windows platforms.

**Elasticity** The elasticity of a language implies the ease with which new features (or functions) can be added to the existing program.

**Performance** The performance of a language is a serious consideration, especially when the target environment does not offer much scope for scaling (e.g., in hand-held devices).

**Support and community** A programming language should have a strong community support behind it. A language with an active forum, additional libraries, and extensive tutorials is likely to be more popular than a better language that does not have any user support. Perl is a good example that indicates the importance of community.

**Speed requirements** Different languages take different times to execute. The time to execute a code also depends on whether the language in which it is written is compiled, assembled, or interpreted. For example, a code written in an assembly language will execute faster than a code written in a high-level language such as Visual Basic. Therefore, if execution speed is the main concern of program development, then a low-level language is a good choice.

**GUI requirements** Some languages have an in-built support for GUI, whereas others either do not have or have very little support for GUI. If a program that needs GUI is written using a language that has little support for GUI, then the code will be very lengthy and complex. For example, creating a GUI in C will be more complex than creating a GUI in Visual Basic. Therefore, if an application supporting GUI is required, then the language should be appropriately chosen.

## SUMMARY

- Algorithm gives the logic of a program, that is, a step-by-step description of how to arrive at a solution. Algorithms are implemented using a programming language.
- A flowchart is a diagrammatic representation that illustrates the sequence of steps that must be performed to solve a problem. Flowcharts are usually drawn in the early stages of formulating computer solutions. They facilitate communication between programmers and users.
- Pseudocode is a form of structured English that describes algorithms. It facilitates designers to focus on the logic of the algorithm without getting bogged down by the details of language syntax. Pseudocodes should not include keywords in any specific computer language.
- Programming languages are used to create programs that control the behaviour of a system, to express algorithms, or as a mode of human communication. Every programming language has a vocabulary of syntax and semantics for instructing a computer to perform specific tasks.
- Though high-level programming languages are easy for humans to read and understand, the computer can understand only machine language, which consists of only numbers.
- Second-generation programming languages comprise the assembly languages. Assembly languages are symbolic programming languages that use symbolic notation to represent machine language instructions.
- An assembly language statement consists of a label, an operation code, and one or more operands. Labels are used to identify and refer instructions in the program. The operation code (opcode) is a mnemonic that specifies the operation that has to be performed, such as *move*, *add*, *subtract*, or *compare*. The operand specifies the register or the location in the main memory where the data to be processed is located.
- Once the modules are coded and tested, the object files of all the modules are combined together by the linker to form the final executable file.
- Third-generation programming languages fall between natural languages and machine languages. They include FORTRAN and COBOL, which made it possible for scientists and business people to write programs using familiar terms instead of obscure machine instructions.
- While working with 4GLs, programmers define only what they want the computer to do, without supplying all the details of how it has to be done.
- Fifth-generation programming languages are centred on solving problems using the constraints given to the program rather than using an algorithm written by a programmer. They are widely used in artificial intelligence research.
- In unstructured programming, programmers write small and simple programs consisting of only one main program.
- Structured programming employs a top-down approach in which the overall program structure is broken down into separate modules.

## GLOSSARY

**Algorithm** A formally defined procedure for performing some calculation and provides a blueprint to write a program that solves a particular problem.

**Assembler** System software that translates a code written in assembly language into machine language.

**Assembly language** Symbolic programming languages that use symbolic notation to represent machine language instructions.

**Compiler/Interpreter** System software that translates the source code from a high-level programming language to a lower-level language.

**Flowchart** A graphical or symbolic representation of a process.

**Loader** System software that copies programs from a storage device to the main memory, where they can be executed.

**Machine language** The lowest level of programming that was used to program the first stored-program computer systems and is the only language that the computer understands.

**Pseudocode** A compact and informal high-level description of an algorithm that uses the structural conventions of a programming language.

**Programming language** A language specifically designed to express computations that can be performed by the computer.

**Structured programming** A programming approach that employs a top-down approach in such a way that the overall program structure is broken down into separate modules, and thereby allows the code to be loaded into memory more efficiently and also to be reused in other programs.

EXERCISES

## Fill in the Blanks

1. ________ is a formally defined procedure for performing some calculation.
2. ________ statements are used when the outcome of the process depends on some condition.
3. Repetition can be implemented using constructs such as ________, ________, and ________.
4. ________ symbol is always the first and the last symbol in a flowchart.
5. ________ is a form of structured English that describes algorithms.
6. ________ is used to express algorithms and as a mode of human communication.
7. ________ is a good language for processing numerical data.
8. An assembly language statement consists of a ________, ________, and one or more ________.
9. ________ is used to convert an assembly level program into machine language.
10. ________ and ________ are used to translate the instructions written in a high-level language into computer-executable machine language.
11. Fifth-general programming languages are widely used in ________.
12. The object file is created when ________.
13. Classes define ________ and ________ of objects.
14. An object can be uniquely identified by its ________.
15. Messages consist of ________, ________, and ________.
16. ________ is extensively used for writing efficient codes for operating systems and compilers.
17. Programs written in ________ are robust, secure, and reliable.
18. Small Java applications are called ________.

## Multiple Choice Questions

1. A graphical or symbolic representation of a process is:
   (a) Algorithm (b) Flowchart
   (c) Pseudocode (d) Program
2. The symbol that is represented using a rectangle in a flowchart is:
   (a) Terminal (b) Decision
   (c) Activity (d) Input/output
3. The details that are omitted in pseudocodes are:
   (a) Variable declaration
   (b) System specific code
   (c) Subroutines
   (d) All of these
4. The language that is used to program the first stored-program computer systems is:
   (a) Machine language (b) Assembly language
   (c) Pascal (d) Fortran
5. The register or location in main memory from where the data to be processed is located is specified by:
   (a) Label (b) Opcode
   (c) Operand(s) (d) None of these
6. The generation to which COBOL belongs is:
   (a) First generation
   (b) Second generation
   (c) Third generation
   (d) Fourth generation
7. The language that concentrates on the 'what' instead of the 'how' aspect of the task is:
   (a) First generation
   (b) Second generation
   (c) Third generation
   (d) Fourth generation
8. Of the following, a 5GL is:
   (a) Prolog (b) OPSS
   (c) Mercury (d) LISP
9. The advantages of modularization are:
   (a) Reusability
   (b) Enhanced productivity
   (c) Less time to develop
   (d) All of these
10. The code in 0s and 1s is:
    (a) Source code (b) Object code
    (c) Executable code (d) None of these
11. The system software that creates the final executable file is:
    (a) Assembler (b) Compiler
    (c) Loader (d) Linker
12. The type of high-level language that uses predicate logic is:
    (a) Unstructured
    (b) Procedure oriented
    (c) Logic oriented
    (d) Object oriented
13. The type of data that can be accessed only by the class in which it is declared or by any class that is inherited from it is:
    (a) Public (b) Private
    (c) Protected (d) All of these
14. The high-level language that is used for numeric, scientific, statistical, and engineering computations is:
    (a) C (b) Basic
    (c) Java (d) FORTRAN

EXERCISES

15. The most portable language is:
    (a) C (b) Basic
    (c) Java (d) FORTRAN
16. The language that should not be used for organizing large programs is:
    (a) C (b) C++
    (c) COBOL (d) FORTRAN

**State True or False**

1. Algorithm solves a problem in a finite number of steps.
2. Repetition means that each step of the algorithm is executed in a specified order.
3. Terminal symbol depicts the flow of control of the program.
4. Labelled connectors are square in shape.
5. Flowcharts are drawn in the early stages of formulating computer solutions.
6. The main focus of pseudocodes is on the details of the language syntax.
7. Assembly language is a low-level programming language.
8. C and Pascal can be used for writing well-structured and readable programs.
9. A code written in machine language is highly portable.
10. Assembly language is directly related to the internal architecture of the computer.
11. Fourth-generation programming languages are nonprocedural languages.
12. It takes less time to write a structured program than other programs.
13. Algorithms are implemented using a programming language.
14. Logic errors are much harder to locate and correct than syntax errors.
15. An interpreter translates the code and also executes it.
16. A subclass can inherit properties and methods from multiple parent classes.
17. A Private data can be accessed only by the class in which it is declared or by any class that is inherited from it.
18. Visual Basic is an object-oriented programming language.
19. Ada is an object-oriented high-level programming language.

**Review Questions**

1. Define an algorithm. How is it useful in the context of software development?
2. Explain sequence, repetition, and decision statements. Also give the keywords used in each type of statement.
3. With the help of an example, explain the use of a flowchart.
4. How is a flowchart different from an algorithm? Do we need to have both of them for program development?
5. What do you understand by the term *pseudocode*?
6. Differentiate between algorithm and pseudocodes.
7. Define the term *programming language*. Give examples of such languages.
8. State the factors that a user should consider to choose a particular programming language.
9. What is machine language? Do we still use it?
10. Write a short note on assembly language.
11. What is an assembler?
12. Differentiate between an assembler and an interpreter.
13. A code written in machine language is efficient and fast to execute. Comment.
14. How is a third-generation programming language better than its predecessors?
15. Explain the significance of assemblers, interpreters, and compilers.
16. A 4GL code enhances the productivity of the programmers. Justify.
17. Write a short note on structured programming.
18. What is modularization? Give its advantages.
19. How can you categorize high-level languages?
20. Differentiate between a procedural language and an object-oriented language.
21. Differentiate between a class and an object.
22. Explain the main features of an object-oriented programming language.
23. Write a short note on some popularly used programming languages.
24. If given a program to write, how will you select the programming language to write the code?