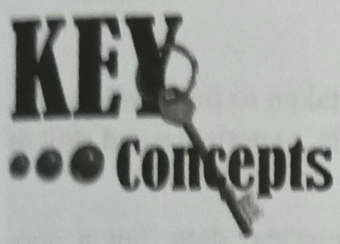
CHAPTER 2

Introduction to Object Oriented Programming (OOP)



• Programming Languages • Programming paradigms • Concepts of OOP • Merits, demerits, and applications of OOP

2.1 COMPUTER PROGRAMMING AND PROGRAMMING LANGUAGES

A *program* is a collection of instructions that tells the computer how to solve a particular problem. We have already written algorithms and pseudocdes and drawn flowcharts that gives a blueprint of the solution (or the program to be written). Computer programming goes a step further in problem solving process. *Programming* is the process of taking an algorithm and writing it in a particular programming language, so that it can be executed by a computer. Programmers can use any of the programming languages that exist to write a program.

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), Python, 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.

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. Python 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.

The selection of language for writing a program depends on the following factors:

- The type of computer hardware and software on which the program is to be executed.
- The type of program.
- The expertise and availability of the programmers.
- Features to write the application.
- It should have built-in features that support the development of software that are reliable and less prone to crash.
- Lower development and maintenance costs.
- Stability and capability to support even more than the expected simultaneous users.
- Elasticity of a language that implies the ease with which new features (or functions) can be added to the existing program.
- Portability.
- Better speed of development that includes the time it takes to write a code, 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.

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. Python, however is a good mix of the best features of all these languages.

2.2 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.

2.2.1 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 (refer to Figure 2.1).

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.

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
                0000
FF55    CF      FF54    CF      FF53    CF      C1      00A9
        FF24    CF      FF27    CF      D2      C7      00CC
                                                        00E4
                                                        010D
                                                        013D
```

Figure 2.1 A machine language program

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 2.1 lists the advantages and disadvantages of machine language.

Table 2.1 Advantages and disadvantages of machine language

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

2.2.2 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 even 4GL to write such codes. Table 2.2 lists the advantages and disadvantages of using assembly language.

Table 2.2 Advantages and disadvantages of assembly language

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

2.2.3 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.

Note Assemblers, linkers, compilers, loaders, and interpreters are all system software, which are discussed in Section `1.13.1`.

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 2.3 provides the advantages and disadvantages of 3GLs.

Table 2.3 Advantages and disadvantages of 3GLs

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

Note Python, Ruby, and Perl are third generation programming languages that combine some 4GL abilities within a general-purpose 3GL environment.

2.2.4 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.

2.2.5 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.

2.3 PROGRAMMING PARADIGMS

A *programming paradigm* is a fundamental style of programming that defines how the structure and basic elements of a computer program will be built. The style of writing programs and the set of capabilities and limitations that a particular programming language has depends on the programming paradigm it supports. While some programming languages strictly follow a single paradigm, others may draw concepts from more than one. The sweeping trend in the evolution of high-level programming languages has resulted in a shift in programming paradigm. These paradigms, in sequence of their application, can be classified as follows:

- Monolithic programming—emphasizes on finding a solution
- Procedural programming—lays stress on algorithms
- Structured programming—focuses on modules
- Object-oriented programming—emphasizes on classes and objects
- Logic-oriented programming—focuses on goals usually expressed in predicate calculus
- Rule-oriented programming—makes use of 'if-then-else' rules for computation
- Constraint-oriented programming—utilizes invariant relationships to solve a problem

Each of these paradigms has its own strengths and weaknesses and no single paradigm can suit all applications. For example, for designing computation intensive problems, procedure-oriented programming is preferred; for designing a knowledge base, rule-based programming would be the best option; and for hypothesis derivation, logic-oriented programming is used. In this book, we will discuss only first four paradigms. Among these paradigms, object oriented paradigms supersede to serve as the architectural framework in which other paradigms are employed.

2.3.1 Monolithic Programming

Programs written using monolithic programming languages such as assembly language and BASIC consist of global data and sequential code. The global data can be accessed and modified (knowingly or mistakenly) from any part of the program, thereby, posing a serious threat to its integrity.

A sequential code is one in which all instructions are executed in the specified sequence. In order to change the sequence of instructions, jump statements or 'goto' statements are used. Figure 2.2 shows the structure of a monolithic program.

As the name suggests, monolithic programs have just one program module as such programming languages do not support the concept of subroutines. Therefore, all the actions required to complete a particular task are embedded within the same application itself. This not only makes the size of the program large but also makes it difficult to debug and maintain.

For all these reasons, monolithic programming language is used only for very small and simple applications where reusability is not a concern.

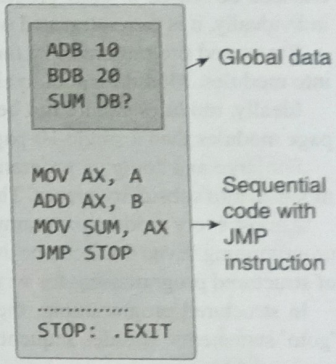


Figure 2.2 Structure of a monolithic program

2.3.2 Procedural Programming

In procedural languages, a program is divided into *n* number of subroutines that access global data. To avoid repetition of code, each subroutine performs a well-defined task. A subroutine that needs the service provided by another subroutine can call that subroutine. Therefore, with 'jump', 'goto', and 'call' instructions, the sequence of execution of instructions can be altered. Figure 2.3 shows the structure of a procedural language.

FORTRAN and COBOL are two popular procedural programming languages.

Advantages

- The only goal is to write correct programs.
- Programs were easier to write as compared to monolithic programming.

Disadvantages

- Writing programs is complex.
- No concept of reusability.
- Requires more time and effort to write programs.
- Programs are difficult to maintain.
- Global data is shared and therefore may get altered (mistakenly).

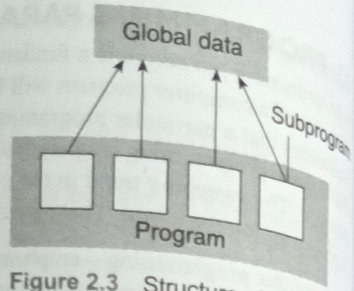


Figure 2.3 Structure of a procedural program

2.3.3 Structured Programming

Structured programming, also referred to as modular programming, was first suggested by mathematicians Corrado Bohm and Guiseppe Jacopini. It was specifically designed to enforce a logical structure on the program to make it more efficient and easier to understand and modify. Structured programming was basically defined to be used in large programs that require a large development team to develop different parts of the same program.

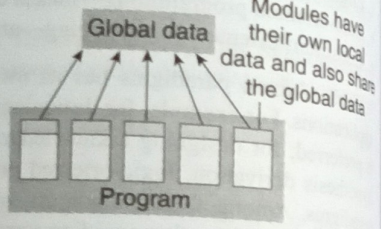


Figure 2.4 Structured program

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 loaded into memory more efficiently and also be reused in other programs. Modules are coded separately and once a module is written and tested individually, it is then integrated with other modules to form the overall program structure (refer to Figure 2.4).

Structured programming is, therefore, based on modularization which groups related statements together into modules. Modularization makes it easier to write, debug, and understand the 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 large and complex programs, the overall program structure may further require the need to break the modules into subsidiary pieces. This process continues until an individual piece of code can be written easily.

Almost every modern programming language similar to C, Pascal, etc. supports the concepts of structured programming. Even OOP can be thought of as a type of structured programming. In addition to the techniques of structured programming for writing modules, it also focuses on structuring its data.

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 as mentioned here.

- 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, end if, or switch, that help to 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, or do... until. Essentially, repetition instructs the program as to how long it needs to continue the function before requesting further instructions.

Advantages

- The goal of structured programming is to write correct programs that are easy to understand and change.
- Modules enhance programmer's productivity by allowing them to look at the big picture first and 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 takes less time to be written than other programs. Modules or procedures written for one program can be reused in other programs as well.
- Each module performs a specific task.
- Each module has its own local data.
- A structured program is easy to debug because each procedure is specialized to perform just one task and every procedure can be checked individually for the presence of any error. In striking contrast, unstructured programs consist of a sequence of instructions that are not grouped for specific tasks. Their logic is cluttered with details and, 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.
- More emphasis is given on the code and the least importance is given to the data.
- Global data may get inadvertently changed by any module using it.
- Structured programs were the first to introduce the concept of functional abstraction.

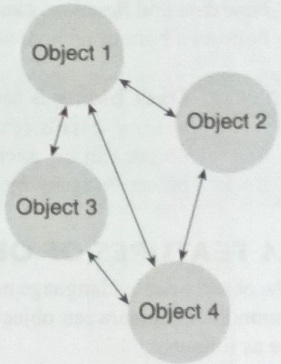
Note Functional abstraction allows a programmer to concentrate on what a function (or module) does and not on how it does.

Disadvantages

- Structured programming is not data-centered.
- Global data is shared and therefore may get inadvertently modified.
- Main focus is on functions.

2.3.4 Object Oriented Programming (OOP)

With the increase in size and complexity of programs, there was a need for a new programming paradigm that could help to develop maintainable programs. To implement this, the flaws in previous paradigms had to be corrected. Consequently, OOP was developed. It treats data as a critical element in the program development and restricts its flow freely around the system. We have seen that monolithic, procedural, and structured programming paradigms are task-based as they focus on the actions the software should accomplish. However, the object oriented paradigm is task-based and data-based. In this paradigm, all the relevant data and tasks are grouped together in entities known as objects (refer to Figure 2.5).



Objects of a program interact by sending messages to each other

Figure 2.5 Object oriented paradigm

For example, consider a list of numbers stored in an array. The procedural or structured 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. For example, to find the largest number or to sort the numbers in the list, we needed specific procedures or functions to do the task. Therefore, the list was a passive entity as it was 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. In this approach, the list is considered an object consisting of the list, along 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, etc.

The striking difference between OOP and traditional approaches is that the program accessing this list need not contain procedures for performing tasks; rather, it 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.

Therefore, 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 2.6 represents a generic object in the object oriented paradigm. Every object contains some data and the operations, methods, or functions that operate on that data. While some objects may contain only basic data types such as characters, integers, and floating types, other objects, on the other hand, may incorporate complex data types such as trees or graphs.

Programs that need the 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.

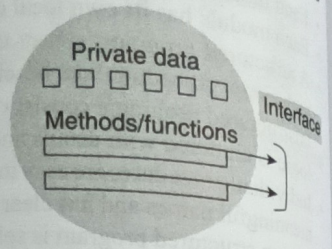


Figure 2.6 Object

For example, the interface for the 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 specify whether the sort should be ascending or descending. Hence, an interface specifies how messages can be sent to the object.

Note OOP is used for simulating real world problems on computers because the real world is made up of objects.

The striking features of OOP include the following:

- The programs are data-centred.
- Programs are divided in terms of objects and not procedures.
- Functions that operate on data are tied together with the data.
- Data is hidden and not accessible by external functions.
- New data and functions can be easily added as and when required.
- Follows a bottom-up approach (discussed in Section 1.16.1) for problem solving.

Note **Data hiding** is technique widely used in object oriented programming (OOP) to hide the internal details (data members) of an object. Data hiding ensures that data members of an object can be exclusively used only by that object. This is especially important to protect object integrity by preventing unintended or intended changes.

2.4 FEATURES OF OBJECT ORIENTED PROGRAMMING

The object oriented language must support mechanisms to define, create, store, manipulate objects, and allow communication between objects. In this section, we will read about the underlying concepts of OOP. These are as follows:

- Classes
- Objects
- Methods
- Message passing
- Inheritance
- Polymorphism
- Containership
- Reusability
- Delegation
- Data Abstraction and Encapsulation

2.4.1 Classes

Almost every language has some basic data types such as int, float, long, and so on, but not all real world objects can be represented using these built-in types. Therefore, OOP, being specifically designed to solve real world problems, allows its users to create user defined data types in the form of classes.

A *class* is used to describe something in the world, such as occurrences, things, external entities, and so on. A class provides a template or a blueprint that describes the structure and behaviour of a set of similar objects. Once we have the definition for a class, 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. The operations that can be performed on its data may include 'getdata', 'setdata', 'editdata', and so on. Figure 2.7 shows the class `Student` with a function `showData()` and attributes namely, `roll_no`, `name`, and `course`. Therefore, we can say that a class describes one or more similar objects.

```
class Student:
def __init__(self, roll_no, name, course):
        self.roll_no = roll_no
        self.name = name
        self.course = course
def showData(self):
        print("ROLL NUMBER : ",self.roll_no)
        print("NAME : ",self.name)
        print("COURSE : ", self.course)
```

Figure 2.7 A sample student class

It must be noted that this data and the set of operations that we have given here can be applied to all students in the class. When we create an instance of a student, we are actually creating an object of class student.

Therefore, once a class is declared, a programmer can create any number of objects of that class.

Note Classes define properties and behaviour of objects.

Therefore, a class is a collection of objects. It is a user-defined data type that behaves same as the built-in data types. This can be realized by ensuring that the syntax of creating an object is same as that of creating an int variable. For example, to create an object (`stud`) of class student, we write

```
student = stud()
```

Note Defining a class does not create any object. Objects have to be explicitly created by using the syntax as follows:
`object-name = class-name()`

2.4.2 Objects

In the previous section, we have taken an example of student class and have mentioned that a class is used to create instances, 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 the objects of the student class. Therefore, all students such as Aditya, Chaitanya, Deepti, and Esha are objects of the class.

Hence, a class can have multiple instances.

Every object contains some data and functions (also called methods) as shown in Figure 2.8. These methods store data in variables and respond to the messages that they receive from other objects by executing their methods (procedures).

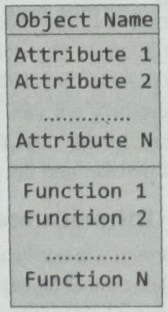


Figure 2.8 Representation of an object

Note While a class is a logical structure, an object is a physical actuality.

2.4.3 Method and Message Passing

A method is a function associated with a class. It defines the operations that the object can execute when it receives a message. In object oriented language, only methods of the class can access and manipulate

the data stored in an instance of the class (or object). Figure 2.9 shows how a class is declared using its data members and member functions.

Every object of the class has its own set of values. Therefore, two distinguishable objects can have the same set of values. Generally, the set of values that the object takes at a particular time is known as the *state* of the object. The state of the object can be changed by applying a particular method. Table 2.4 shows some real world objects along with their data and operations.

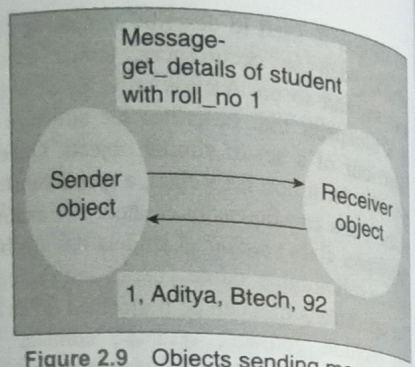


Figure 2.9 Objects sending messages

Table 2.4 Objects with data and functions

Object	Data or attributes	Functions or methods
Person	Name, age, sex	Speak(), walk(), listen(), write()
Vehicle	Name, company, model, capacity, colour	Start(), stop(), accelerate()
Polygon	Vertices, border, colour	Draw(), erase()
Account	Type, number, balance	Deposit(), withdraw(), enquire()
City	Name, population, area, literacy rate	Analyse(), data(), display()
Computer	Brand, resolution, price	Processing(), display(), printing()

Note An object is an instance of a class which 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.

Two objects can communicate with each other through messages. An object asks another object to invoke one of its methods by sending it a message. In Figure 2.9, a sender object is sending 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.

In the figure, sender has asked the receiver to send the details of student having *roll_no 1*. This means that the sender is passing some specific information to the receiver so that the receiver can send the correct and precise information to the sender. The data that is transferred with the message is called *parameters*. Here, *roll_no 1* is the parameter.

Therefore, we can say that messages that are sent to other objects consist of three aspects—the receiver object, the name of the method that the receiver should invoke, and the parameters that must be used with the method.

2.4.4 Inheritance

Inheritance is a concept of OOP in which a new class is created from an existing class. The new class, often known as a subclass, contains the attributes and methods of the parent class (the existing class from which the new class is created).

The new class, known as subclass or derived class, inherits the attributes and behaviour of the pre-existing class, which is referred to as superclass or parent class (refer to Figure 2.10). The inheritance relationship of subclasses and superclasses generates a hierarchy. Therefore, inheritance relation is also called `'is-a'` relation. A subclass not only has all the states and behaviours associated with the superclass but has other specialized features (additional data or methods) as well.

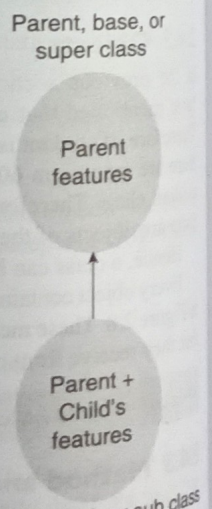


Figure 2.10 Inheritance

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 add the specialized code for the subclass. For example, if we have a class student with following members:

```
Properties: roll_number, name, course and aggregate
Methods: getdata, setdata
```

We can inherit two classes from the class `student`, namely, undergraduate students and postgraduate students (refer to Figure 2.11). These two classes will have all the properties and methods of class students and in addition to that, will have even more specialized members.

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

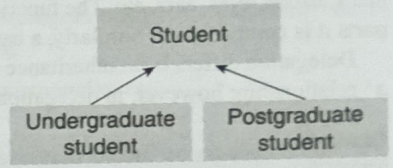


Figure 2.11 Example of Inheritance

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

2.4.5 Polymorphism

Polymorphism, one of the essential concepts of OOP, refers to having several different forms. While inheritance is related to classes and their hierarchy, polymorphism, on the other hand, is related to methods. Polymorphism is a concept that enables the programmers to assign a different meaning or usage to a method in different contexts. In Python, the word 'polymorphism' is often used with inheritance. Polymorphism exists when a number of subclasses is defined which have methods of same name. A function can use objects of any of the polymorphic classes irrespective of the fact that these classes are individually distinct.

Polymorphism can also be applied to operators. For example, we know that operators can be applied only on basic data types that the programming language supports. Therefore, `a + b` will give the result of adding a and b. If `a = 2` and `b = 3`, then `a + b = 5`. When we overload the `+` operator to be used with strings, then `Fraction1 + Fraction2` adds two fractional numbers and returns the result.

Note Binding means associating a function call with the corresponding function code to be executed in response to the call.

2.4.6 Containership

Containership is the ability of a class to contain object(s) of one or more classes as member data. For example, class One can have an object of class Two as its data member. This would allow the object of class One to call the public functions of class Two. Here, class One becomes the container, whereas class Two becomes the contained class.

Containership is also called *composition* because as in our example, class One is composed of class Two. In OOP, containership represents a `'has-a'` relationship.

2.4.7 Reusability

Reusability means developing codes that can be reused either in the same program or in different programs. Python gives due importance to building programs that are reusable. Reusability is attained through inheritance, containership, and polymorphism.

2.4.8 Delegation

To provide maximum flexibility to programmers and to allow them to generate a reusable code, object oriented languages also support delegation. In composition, an object can be composed of other objects and thus, the object exhibits a `'has-a'` relationship.

In delegation, more than one object is involved in handling a request. The object that receives the request for a service, delegates it to another object called its delegate. The property of delegation emphasizes on the ideology that a complex object is made of several simpler objects. For example, our body is made up of brain, heart, hands, eyes, ears, etc. The functioning of the whole body as a system rests on correct functioning of the parts it is composed of. Similarly, a car has a wheel, brake, gears, etc. to control it.

Delegation differs from inheritance in the way that two classes that participate in inheritance share an `'is-a'` relationship; however, in delegation, they have a `'has-a'` relationship.

Note Delegation means that one object is dependent on another object to provide functionalities.

Delegation vs Composition

Delegation is closely related to composition i.e., object of one class acts as a data member in another class. However, in composition, the child cannot exist without the context of the parent.
For example, a school has one or more classes. If we remove the school from existence, the classes cease to exist. This is containership. On the other hand, a school also has a number of students, being instances of another entity person. This represents a delegation because even if school does not exist, students will still exist as a person (that is outside of the context of that school).

2.4.9 Data Abstraction and Encapsulation

Data abstraction refers to the process by which data and functions are defined in such a way that only essential details are revealed 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, as users of television sets, we can switch it on or off, change the channel, set the volume, and add external devices such as speakers and CD or DVD players without knowing the details about how its functionality has been implemented. Therefore, the internal implementation is completely hidden from the external world.

Similarly, in OOP languages, classes provide public methods to the outside world to provide the functionality of the object or to manipulate the object's data. Any entity outside the world does not know about the implementation details of the class or that method.

Data encapsulation, also called data hiding, is the technique of packing data and functions into a single component (class) to hide implementation details of a class from the users. 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) that is 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, explained as follows.

- Any data or function with access level as 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 that class 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.

Note Creating a new data type using encapsulated items that is well suited for an application is called data abstraction.

2.5 MERITS AND DEMERITS OF OBJECT ORIENTED PROGRAMMING LANGUAGE

OOP offers many benefits to program developers and users. It not only provides a solution for many problems associated with software development and its quality, but also enhances programmer productivity and reduces maintenance cost. Some key advantages of OOP include the following:

- Elimination of redundant code through inheritance (by extending existing classes).
- Higher productivity and reduced development time due to reusability of the existing modules.
- Secure programs as data cannot be modified or accessed by any code outside the class.
- Real world objects in the problem domain can be easily mapped to objects in the program.
- A program can be easily divided into parts based on objects.
- The data-centred design approach captures more details of a model in a form that can be easily implemented.
- Programs designed using OOP are expandable as they can be easily upgraded from small to large systems.
- Message passing between objects simplifies the interface descriptions with external systems.
- Software complexity becomes easily manageable.
- With polymorphism, behaviour of functions, operators, or objects may vary depending upon the circumstances.
- Data abstraction and encapsulation hides implementation details from the external world and provides it a clearly defined interface.
- OOP enables programmers to write easily extendable and maintainable programs.
- OOP supports code reusability to a great extent.

However, the downside of OOP include the following:

- Programs written using object oriented languages have greater processing overhead as they demand more resources.
- Requires more skills to learn and implement the concepts.
- Beneficial only for large and complicated programs.
- Even an easy to use software when developed using OOP is hard to be built.
- OOP cannot work with existing systems.
- Programmers must have a good command in software engineering and programming methodology.

2.6 APPLICATIONS OF OBJECT ORIENTED PROGRAMMING

No doubt, the concepts of object oriented technology have changed the way of thinking, analyzing, planning, and implementing software. Software or applications developed using this technology are not only efficient but also easy to upgrade. Therefore, programmers and software engineers all over the world have shown their keen interest in developing applications using OOP. As a result, there has been a constant increase in areas where OOP has been successfully implemented. Some of these areas include the following:

- Designing user interfaces such as work screens, menus, windows, and so on
- Real-time systems
- Simulation and modelling
- Compiler design
- Client server system
- Object oriented databases
- Object oriented distributed database
- Artificial intelligence—expert systems and neural networks
- Parallel programming
- Decision control systems
- Office automation systems
- Networks for programming routers, firewalls, and other devices
- Computer-aided design (CAD) systems

- Computer-aided manufacturing (CAM) systems
- Computer animation
- Developing computer games
- Hypertext and hypermedia

2.7 DIFFERENCES BETWEEN POPULAR PROGRAMMING LANGUAGES

Table 2.5 highlights the differences between popular programming languages.

Table 2.5 Comparison between commonly used programming languages

ATTRIBUTE	C	C++	Java	Python	Smalltalk
Cross platform	Good support	Good support	Better support	Better support	Better support
Simple and Concise	Little Difficult	Difficult	Difficult	Easy	Easy
Reusability	Little	Better	Good	Good	Good
Consistent functional constructs	Less	Less	Better	Good	Good
Object oriented	No	Yes	Yes	Yes	Yes
Popularity	High	Good	High	Good; increased in recent times	Little Less
Use	Application, system, general purpose, low-level operations	Application, system	Application, business, client-side, general, mobile development, server-side, web	Application, general, web, scripting, artificial intelligence, scientific computing	Application, general, business, artificial intelligence, education, web
Functional constructs	No	Yes	Yes	Yes	Yes
Procedural	Yes	Yes	Yes	Yes	Yes
Generic	No	Yes	Yes	No	No
Event driven	No	No	Yes	No	Yes
Other paradigms	NA	NA	Concurrent	Aspect oriented	Concurrent, declarative
Program size	Big	Big	Less Big	Small code that requires less memory	Medium
Effort to write programs	More	More	Little Less	Less	Less
Garbage collection	No	No	Yes	Yes	Yes
Standardized	1989, ANSI C89, ISO C90, ISO C99, ISO C11	1198, ISO/IEC 1998, 2003, 2011, 2014	De facto standard	De facto standard	1998, ANSI