CHAPTER 1: INTRODUCTION TO PROGRAMMING AND OBJECT-ORIENTATION

Table of Contents

1 INTRODUCTION TO PROGRAMMING

A programming language is used by a human programmer to direct a computer to *accomplish a specific set of steps* which lead to a desired outcome.

The figure below is intended to illustrate several key points about programming languages. First, programming languages are built upon and relate directly to the underlying computer (hardware). In fact, they are designed to control the operation of the hardware. Second, these programming languages can be divided into **TWO** major groups. They are:

- **Low-level Languages** (Low Level languages are further divided into *Machine language* and *Assembly language*)
- **High-Level Languages**

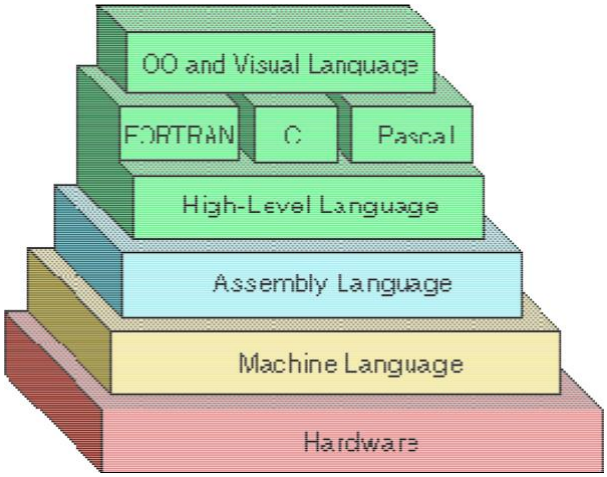


Figure 1: Levels of Programming Languages

1.1 Low Level Languages

The term low level means closeness to the way in which the machine has been built. Low level languages are **machine oriented** and require extensive knowledge of computer hardware and its configuration.

1.1.1 Machine Language

Machine Language is the only language that is directly understood by the computer. It does not need any translator program. We also call it **machine code** and it is written as strings of 1′s (one) and 0's (zero). When this sequence of codes is fed to the computer, it recognizes the codes and converts it in to electrical signals needed to run it. For example, a program instruction may look like this:
1011000111101

It is not an easy language to learn because of its difficult to understand. It is efficient for the computer but very inefficient for programmers. It is considered to be first generation language. It is also difficult to debug the program written in this language.

Advantage of Machine Language

The only advantage is that program of machine language run very fast because no translation program is required for the CPU.

Disadvantages of Machine Language

i. It is very difficult to program in machine language. The programmer has to know details of hardware to write program.
ii. The programmer has to remember a lot of codes to write a program which might easily result in program errors.
iii. It is difficult to debug the program.

1.1.2 Assembly Language

Instead of using a string of binary bits in a machine language, programmers started using English-like words as commands that can be easily interpreted by programmers. In other words, the computer manufacturers started providing English-like words abbreviated as mnemonics that are similar to binary instructions in machine languages. The program is in alphanumeric symbols instead of 1s and 0s. The designer chooses easy symbols that are to be remembered by the programmer, so that the programmer can easily develop the program in assembly language. The alphanumeric symbols are called mnemonics in the assembly language. The ADD, SUB, MUL, DIV, RLC and RAL are some symbols called mnemonics.

No matter how close assembly language is to machine code, the computer still cannot understand it. The assembly-language program must be translated into machine code by a separate program called an **assembler**. The assembler program recognizes the character strings that make up the symbolic names of the various machine operations, and substitutes the required machine code for each instruction. At the same time, it also calculates the required address in memory for each symbolic name of a memory location, and substitutes those addresses for the names. The final result is a machine-language program that can run on its own at any time; the assembler and the assembly-language program are no longer needed. To help distinguish between the "before" and "after" versions of the program, the original assembly-language program is also known as the source code, while the final machine-language program is designated the object code.

Every processor has its own assembly language. For example, 8085 CPU has its own assembly language. CPUs such as 8086, 80186, 80286 have their own assembly languages.

If an assembly-language program needs to be changed or corrected, it is necessary to make the changes to the source code and then re-assemble it to create a new object program..

Advantages of Assembly Language:

i. The symbolic programming of Assembly Language is easier to understand and saves a lot of time and effort of the programmer.
ii. It is easier to correct errors and modify program instructions.
iii. Assembly Language has the same efficiency of execution as the machine level language. Because this is one-to-one translator between assembly language program and its corresponding machine language program.

Disadvantages of Assembly Language:

i. One of the major disadvantages is that assembly language is machine dependent. A program written for one computer might not run in other computers with different hardware configuration i.e. Assembly language programs are not portable.
ii. It is necessary to remember the registers of CPU and mnemonic instructions by the programmer.
iii. Several mnemonic instructions are needed to write in assembly language than a single line in high-level language. Thus, assembly language programs are longer than the high language programs.

Advantages of Low-level Languages

Low-level languages have the advantage that they can be written to take advantage of any peculiarities in the architecture of the central processing unit (CPU) which is the "brain" of any computer. Thus, a program written in a low-level language can be extremely efficient, making optimum use of both computer memory and processing time.

Disadvantages of Low-level Languages

To write a low-level program takes a substantial amount of time, as well as a clear understanding of the inner workings of the processor itself. Therefore, low-level

programming is typically used only for very small programs, or for segments of code that are highly critical and must run as efficiently as possible.

1.2 High-Level Languages

Very early in the development of computers, attempts were made to make programming easier by reducing the amount of knowledge of the internal workings of the computer that was needed to write programs. If programs could be presented in a language that was more familiar to the person solving the problem, then fewer mistakes would be made.

High-level programming languages allow the specification of a problem solution in terms closer to those used by human beings. These languages were designed to make programming far easier, less error-prone and to remove the programmer from having to know the details of the internal structure of a particular computer. These high-level languages were much closer to human language.

Advantages of High-level Languages

High-level languages permit faster development of large programs. The final program as executed by the computer is not as efficient, but the savings in programmer time generally far outweigh the inefficiencies of the finished product. This is because the cost of writing a program is nearly constant for each line of code, regardless of the language. Thus, a high-level language where each line of code translates to 10 machine instructions costs only one tenth as much in program development as a low-level language where each line of code represents only a single machine instruction.

1.3 Translators

A program is a set of instructions for performing a particular task. These instructions are just like English words. The computer interprets the instructions as 1's and 0's. A program can be written in assembly language as well as in high-level language. This written program is called the **source program**. The source program is to be converted to the machine language, which is called an **object program**. A translator is required for such a translation.

Program translator translates source code of programming language into machine language-instruction code. Generally, computer programs are written in languages like COBOL, C, JAVA, BASIC, ASSEMBLY LANGUAGE etc which should be translated into machine language before execution. Programming language translators are classified as shown in the figure below:

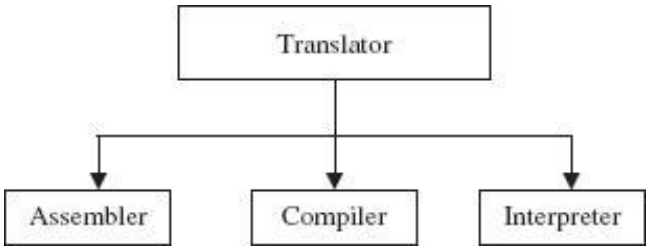


Figure 2: Translators

Translators are as follows:

a) Assembler
b) Compiler
c) Interpreter

Assembler: An assembler translates the symbolic codes of programs of an assembly language into machine language instructions. The symbolic language is translated to the machine code in the ratio of one is to one symbolic instructions to one machine code instructions.

Compiler: It is a program which translates a high level language program into a machine language program. A compiler is more intelligent than an assembler. It checks all kinds of limits, ranges, errors etc. But its program run time is more and occupies a larger part of the memory. It has slow speed. Because a compiler goes through the entire program and then translates the entire program into machine codes.

If a compiler runs on a computer and produces the machine codes for the same computer then it is known as a self compiler or resident compiler. On the other hand, if a compiler runs on a computer and produces the machine codes for other computer then it is known as a cross compiler.

Compilers will show errors and warnings for the statements violating the syntax rules of the language. They also have the ability of linking subroutines of the program.

Interpreter: An interpreter is a program which translates statements of a program into machine code. It translates *only one statement of the program at a time*. It reads only one statement of program, translates it and executes it. Then it reads the next statement of the program again translates it and executes it. In this way it proceeds further till all the statements are translated and executed.

On the other hand, a compiler goes through the entire program and then translates the entire program into machine codes. A compiler is 5 to 25 times faster than an interpreter.

By the compiler, the machine codes are saved permanently for future reference. On the other hand, the machine codes produced by interpreter are not saved. An interpreter is a small program as compared to compiler. It occupies less memory space, so it can be used in a smaller system which has limited memory space.

1.4 What is OOP?

OOP is a design philosophy. It stands for Object Oriented Programming. **O**bject-**O**riented **P**rogramming (*OOP*) uses a different set of programming languages than old procedural programming languages (*C, Pascal*, etc.). Everything in *OOP* is grouped as self sustainable "*objects*". Hence, you gain reusability by means of four main object-oriented programming concepts.

In order to clearly understand the object orientation model, let's take your "hand" as an example. The "*hand*" is a class. Your body has two objects of the type "*hand*", named "left hand" and "right hand". Their main functions are controlled ormanaged by a set of electrical signals sent through your shoulders (through an interface). So the shoulder is an interface that your body uses to interact with your hands. The hand is a well-architected class. The hand is being reused to create the left hand and the right hand by slightly changing the properties of it.

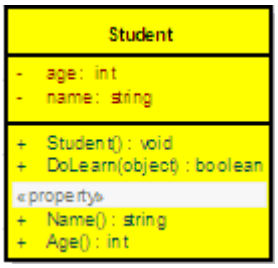
1.4.1 What is an Object?

An object can be considered a "*thing*" that can perform a set of related activities. The set of activities that the object performs defines the object's behavior. An object represents an entity in the real world that can be distinctly identified for example, the Hand (object) can grip something, or a *Student* (object) can give their name or address.

A desk, a circle, a button, and even a loan can all be viewed as objects. An object has a unique identity, **state**, and **behaviors**. The state of an object consists of a set of data fields (also known as properties) with their current values. The behavior of an object is defined by a set of methods.

In pure *OOP* terms an object is an instance of a class.

1.4.2 What is a Class?



A *class* is simply a representation of a type of *object*. It is the blueprint, or plan, or template, which describes the details of an *object*. A class is the blueprint from which the individual objects are created. *Class* is composed of three things: a name, attributes, and operations.

A Java class uses variables to define data fields and methods to define behaviors. A class provides a special type of methods, known as constructors, which are invoked to construct objects from the class.

```
public class Student
{
}
```

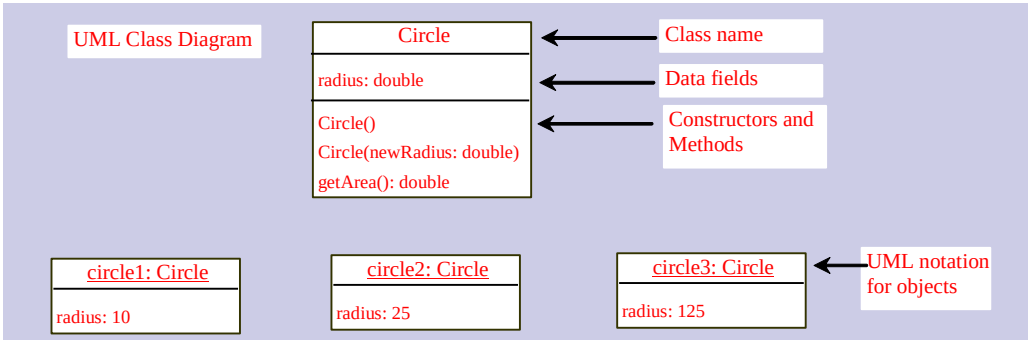
According to the sample given below we can say that the Student object, named objectStudent, has been created out of the Student class.

```
Student objectStudent = new Student();
```

In real world, you'll often find many individual objects all of the same kind. As an example, there may be thousands of other bicycles in existence, all of the same make and model. Each bicycle has built from the same blueprint. In object-oriented terms, we say that the bicycle is an instance of the class of objects known as bicycles.

In the software world, though you may not have realized it, you have already used classes. For example, the TextBox control, is made out of the TextBox class, which defines its appearance and capabilities. Each time you drag aTextBox control, you are actually creating a new instance of the TextBox class.

1.4.3 UML Class Diagram (Unified Modeling Language)



1.4.4 How to identify and design a Class?

This is an art; each designer uses different techniques to identify classes. However according to Object Oriented Design Principles, there are five principles that you must follow when design a class,

- SRP - The Single Responsibility Principle - A class should have one, and only one, reason to change.
- OCP - The Open Closed Principle - Should be able to extend any classes' behaviors, without modifying the classes.
- LSP - The Liskov Substitution Principle- Derived classes must be substitutable for their base classes.
- DIP - The Dependency Inversion Principle- Depend on abstractions, not on concretions.
- ISP - The Interface Segregation Principle- Make fine grained interfaces that are client specific.

Additionally to identify a class correctly, you need to identify the full list of leaf-level functions or operations of the system (granular level use cases of the system). Then you can proceed to group each function to form classes (classes will group same types of functions or operations). However a well-defined class must be a meaningful grouping of a set of functions and should support the reusability, while increasing expandability or maintainability, of the overall system.

In software world the concept of dividing and conquering is always recommended, if you start analyzing a full system at the start, you will find it harder to manage. So the better approach is to identify the module of the system first and then dig deep in to each module separately to seek out classes.

A software system may consist of many classes. When you have many classes, it needs to be managed. Think of a big organization, with its work force exceeding several thousand employees (let's take one employee as one class). In order to manage such a work force, you need to have proper management policies in place. Same technique can be applied to manage classes of your software system. In order to manage the classes of a software system, and to reduce the complexity, system designers use several techniques, which can be grouped under four main concepts named

1. Encapsulation

2. Abstraction

3. Inheritance

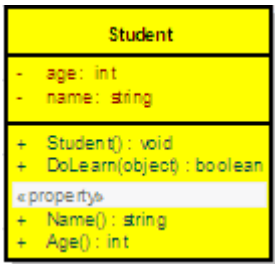
4. Polymorphism.

These concepts are the four main *gods* of *OOP* world and in software term, they are called four main Object Oriented Programming (*OOP*) Concepts.

1.5 Object Oriented Programming (OOP) Concepts

1.5.1 What is Encapsulation?

The encapsulation is the inclusion-within a program object-of all the resources needed for the object to function, basically, the methods and the data. In *OOP* the encapsulation is mainly achieved by creating classes, the classes expose public methods and properties. A class is kind of a container or capsule or a cell, which encapsulate a set of methods, attribute and properties to provide its indented functionalities to other classes. In that sense, encapsulation also allows a class to change its internal implementation without hurting the overall functioning of the system. That idea of encapsulation is to hide how a class does its business, while allowing other classes to make requests of it.



In order to modularize/ define the functionality of a one class, that class can uses functions or properties exposed by another class in many different ways. According to Object Oriented Programming there are several techniques classes can use to link with each other. Those techniques are named association, aggregation, and composition.

There are several other ways that an encapsulation can be used, as an example we can take the usage of an interface. The interface can be used to hide the information of an implemented class.

```
IStudent myLStudent = new LocalStudent();
IStudent myFStudent = new ForeignStudent();
```

According to the sample above (let's assume that both LocalStudent and ForeignStudent classes have implemented the IStudent interface) we can see how LocalStudent and ForeignStudent hide their localize implementing through the IStudentinterface.

As an example in both instances '*myLStudent'* and '*myFStudent'* are of type *IStudent*, but they both carries two separate local and foreign implementation underneath. This way a method call like *'DoLearn*(object)' to *'myLStudent'* and *'myFStudent'*object will trigger their respective foreign and local implementation. This way *'myFStudent*' carrying *'ForeignStudent'* will trigger the respective learning function with foreign syllabus while the other one with '*LocalStudent'* will trigger the learning function with local syllabus.

References

- http://www.ict.griffith.edu.au/arock/oop/students/lecture/
- http://www.codeproject.com/Articles/22769/Introduction-to-Object-Oriented-Programming-Concep
- http://ceng162.cankaya.edu.tr/course.php?page=Lecture%20Notes