# FEDERAL UNIVERSITY OF TECHNOLOGY, MINNA, NIGERIA


## CENTRE FOR OPEN DISTANCE AND e-LEARNING (CODeL)

## B.TECH. COMPUTER SCIENCE PROGRAMME

**COURSE TITLE** ORGANIZATION AND DESIGN OF PROGRAMMING LANGUAGES

**COURSE CODE** CPT 213

**CREDIT UNIT:** 3

## COURSE DEVELOPMENT TEAM

# CPT 213

# ORGANIZATION AND DESIGN OF PROGRAMMING LANGUAGES


**Course Developer/Writers**
Dr. ( Mrs.) Ogwueleka
Computer Science Department,
Federal University of Technology, Minna, Nigeria.

**Course Editors**
Dr. Abraham, O.
Dr. Agboola, A. K.
Computer Science Department
Federal University of Technology, Minna, Nigeria.

**Programme Coordinator**
Mrs O. A. Abisoye
Computer Science Department
Federal University of Technology, Minna, Nigeria.

**Instructional Designers**
Dr. Gambari, Amosa Isiaka
Mr. Falode, Oluwole Caleb
Centre for Open Distance and e-Learning,
Federal University of Technology, Minna, Nigeria.

**Editor**
Chinenye Priscilla Uzochukwu
Centre for Open Distance and e-Learning,
Federal University of Technology, Minna, Nigeria.

**Director**
Prof. J. O. Odigure
Centre for Open Distance and e-Learning,
Federal University of Technology, Minna, Nigeria

# STUDY GUIDE

**CPT 213: ORGANIZATION AND DESIGN OF PROGRAMMING LANGUAGES**

## 1.0 INTRODUCTION

CPT 311: Organization and Design of programming languages is a 3 credit unit course for students studying towards acquiring a Bachelor of Science in Computer Science and other related disciplines. In this course we introduce the major principles and concepts underlying all programming languages without concentrating on one particular programming language. Specific languages are used as examples and illustrations. It is not necessary for the reader to be familiar with all these languages, or even any of them to understand the concept being illustrated. At most the reader is required to be experienced in only one programming language and to have some general knowledge of data structures, algorithms and computational processes.

## 2.0 COURSE GUIDE

The course guide introduces to you what you will learn in this course and how to make the best use of the material. It brings to your notice the general guidelines on how to navigate through the course and on the expected actions you have to take for you to complete this course successfully. Also, the guide will hint you on how to respond to your SELF ASSESSMENT EXERCISEand Tutor-Marked Assignments.

## 3.0 COURSE CONTENTS

1. History of programming languages;
2. Brief survey of programming paradigms (distinguishing characteristics, tradeoffs between different paradigms, safety and power of expression and particular language supporting each paradigm);
3. Procedural languages, Object-oriented languages, Functional languages, Declarative, non-algorithmic languages, Scripting languages;
4. The effects of scale on programming methodology;
5. General principles of language design;
6. Design goals. Typing regimes (Data type as set of values with set of operations, Data types , Elementary types, user-defined types, Abstract data types);
7. Data structure models;
8. Abstraction mechanisms (Procedures, functions); and
9. Control structure models, specifications and their implementations.

## 4.0 COURSE AIM

The aim of this course is to introduce the students to the general organization and Design principles of programming languages.

To help the student gain an understanding of the basic structure of programming languages.To help the students learn the principles underlying all programming languages, So that it is easier to learn new languages.

To study different language paradigms, so that one can be able to select an appropriate language for a task

## 5.0 COURSE OBJECTIVES

At the end of this course, you should be able to:

1. demonstrate understanding of the evolution of programming languages and relate how this history has led to the paradigms available today;
2. identify at least one outstanding and distinguishing characteristic for each of the programming paradigms covered in this unit;
3. be able to explain the differences between programming languages and programming paradigms;
4. be able to differentiate between low-level and high-level programming languages and their associated advantages and disadvantages;
5. be able to list four programming paradigms and describe their strengths and weaknesses;
6. evaluate the tradeoffs between the different paradigms, considering such issues as space efficiency, time efficiency (of both the computer and the programmer), safety, and power of expression;
7. identify at least one outstanding and distinguishing characteristic for each of the programming paradigms covered in this unit;
8. be able to list four properties of programming language and describe them;
9. evaluate the tradeoffs between the different programming principles considering such issues as efficiency and regularity;
10. evaluate the tradeoffs between the different paradigm;
11. be able to list the basic data types;
12. be able to state the difference between Primitive data types and Abstract data types; and
13. describe the importance of Data Abstraction in programming.

## 6.0 Working through This Course

To complete this course, you are required to study all the units, the recommended text books, and other relevant materials. Each unit contains some self assessment exercises and tutor marked assignments, and at some point in this course, you are required to submit the tutor marked assignments. There is also a final examination at the end of this course. Stated below are the components of this course and what you have to do.

## 7.0 Course Materials

The major components of the course are:
1. Course Guide
2. Study Units
3. Text Books
4. Assignment File
5. Presentation Schedule

## 8.0 Study Units

There are 22 study units and 5 modules in this course. They are:

## Module 1: Programming Languages

UNIT 1: Concept of Programming Language
UNIT 2: Categories of Programming Language
UNIT 3: History of Programming Language

## Module 2: Programming Paradigms

UNIT 1: Programming Paradigms
UNIT 2: Classification of Programming Languages Against The Main Paradigms
UNIT 3: Pseudocode Example

## Module 3: Scaling

Unit 1: Decimal Numbers
Unit 2: Storage/Implementation Of Decimal Numbers
Unit 3: Binary Number System
Unit 4: Converting Fractions
Unit 5: Converting A General Decimal Number
Unit 6: Binary Representation and Precision
Unit 7: Float Representation
Unit 8: Storage Error

## Module 4: Language Design

Unit 1: Programming Language Properties
Unit 2: The Goal of Programming Language
Unit 3: History And Design Criteria
Unit 4: Language Design Principles

## Module 5: Data Types

Unit 1: Data Types
Unit 2: Abstract Data Types
Unit 3: Stacks and Queues
Unit 4: Data Abstraction

## 9.0 Recommended Textbooks

1. Principles of Programming Languages by Macclennan, 3rd edition, 1999, Oxford
2. Concept of Programming Languages, 8th edition by Robert W. Sebesta, Addison Wesley, ISBN -13:978-0321-49362-0, ISBN-10:0-321-49362-1

## 10.0 Assignment File

The assignment file will be given to you in due course. In this file, you will find all the details of the work you must submit to your tutor for marking. The marks you obtain for these assignments will count towards the final mark for the course. Altogether, there are tutor marked assignments for this course.

## 11.0 Presentation Schedule

The presentation schedule included in this course guide provides you with important dates for completion of each tutor marked assignment. You should therefore endeavour to meet the deadlines.

## 12.0 Assessment

There are two aspects to the assessment of this course. First, there are tutor marked assignments; and second, the written examination. Therefore, you are expected to take note of the facts, information and problem solving gathered during the course. The tutor marked assignments must be submitted to your tutor for formal assessment, in accordance to the deadline given. The work submitted will count for 40% of your total course mark.At the end of the course, you will need to sit for a final written examination. This examination will account for 60% of your total score.

## 13.0 Tutor Marked Assignments (TMA)

There are TMAs in this course. You need to submit all the TMAs. The best 10 will therefore be counted. When you have completed each assignment, send them to your tutor as soon as possible and make certain that it gets to your tutor on or before the stipulated deadline. If for any reason you cannot complete your assignment on time, contact your tutor before the assignment is due to discuss the possibility of extension. Extension will not be granted after the deadline, unless on extraordinary cases.

## 14.0 Final Examination and Grading

The final examination for CPT 311 will be of last for a period of 2 hours and have a value of 60% of the total course grade. The examination will consist of questions which reflect the self assessment exercise and tutor marked assignments that you have previously encountered. Furthermore, all areas of the course will be examined. It would be better to use the time between finishing the last unit and sitting for the examination, to revise the entire course. You might find it useful to review your TMAs and comment on them before the examination. The final examination covers information from all parts of the course.

## 15.0 The Following Are Practical Strategies For Working Through This Course

1. Read the course guide thoroughly.

2. Organize a study schedule. Refer to the course overview for more details. Note the time you are expected to spend on each unit and how the assignment relates to the units. Important details, e.g. details of your tutorials and the date of the first day of the semester are available. You need to gather together all these information in one place such as a diary, a wall chart calendar or an organizer. Whatever method you choose, you should decide on and write in your own dates for working on each unit.
3. Once you have created your own study schedule, do everything you can to stick to it. The major reason that students fail is that they get behind with their course works. If you get into difficulties with your schedule, please let your tutor know before it is too late for help.
4. Turn to Unit 1 and read the introduction and the objectives for the unit.

5. Assemble the study materials. Information about what you need for a unit is given in the table of content at the beginning of each module. You will almost always need both the study unit you are working on and one of the materials recommended for further readings, on your desk at the same time.
6. Work through the unit, the content of the unit itself has been arranged to provide a sequence for you to follow. As you work through the unit, you will be encouraged to read from your set books.
7. Keep in mind that you will learn a lot by doing all your assignments carefully. They have been designed to help you meet the objectives of the course and will help you pass the examination.
8. Review the objectives of each study unit to confirm that you have achieved them.
If you are not certain about any of the objectives, review the study material and consult your tutor.
9. When you are confident that you have achieved a unit's objectives, you can start on the next unit. Proceed unit by unit through the course and try to pace your study so that you can keep yourself on schedule.
10. When you have submitted an assignment to your tutor for marking, do not wait for its return before starting on the next unit. Keep to your schedule. When the assignment is returned, pay particular attention to your tutor's comments, both on the tutor marked assignment form and also written on the assignment. Consult your tutor as soon as possible if you have any questions or problems.
11. After completing the last unit, review the course and prepare yourself for the final examination. Check that you have achieved the unit objectives (listed at the beginning of each unit) and the course objectives (listed in this course guide).

## 16.0 TUTORS AND TUTORIALS

There are 8 hours of tutorial provided in support of this course. You will be notified of the dates, time and location together with the name and phone number of your tutor as soon as you are allocated a tutorial group. Your tutor will mark and comment on your assignments, keep a close watch on your progress and on any difficulties you might encounter and provide assistance to you during the course. You must mail your tutor-marked assignment to your tutor well before the

due date. At least two working days are required for this purpose. They will be marked by your tutor and returned to you as soon as possible.

Do not hesitate to contact your tutor by telephone, e-mail or discussion board if you need help. The following might be circumstances in which you would find help necessary: contact your tutor if:

• You do not understand any part of the study units or the assigned readings.
• You have difficulty with the self test or exercise.
• You have questions or problems with an assignment, with your tutor's comments on an assignment or with the grading of an assignment.

You should endeavor to attend the tutorials. This is the only opportunity to have face to face contact with your tutor and ask questions which are answered instantly. You can raise any problem encountered in the course of your study. To gain the maximum benefit from the course tutorials, have some questions handy before attending them. You will learn a lot from participating actively in discussions.

**GOODLUCK!**

# Table of Contents

# Module 1

## PROGRMMING LANGUAGES

# Unit 1

## CONCEPT OF PROGRAMMING LANGUAGE

Contents

# 1.0 Introduction

A **programming language** is an artificial language designed to communicate instructions to a machine, particularly a computer. Programming languages can be used to create programs that control the behavior of a machine and/or to express algorithms precisely.

The functions of a computer system are controlled by *computer programs; a* computer program is a clear, step-by-step, finite set of instructions. A computer program must be clear so that only one meaning can be derived from it. A computer program is written in a computer language called a *programming language.*

# 2.0 Objectives

After studying this module you shall:

1. be able to define programming language;
2. be able to differentiate between low-level and high-level programming languages and their associated advantages and disadvantages;
3. be able to narrate the history of programming language and list the important languages developed within each period; and
4. Demonstrate understanding of the evolution of programming languages and relate how this history has led to the paradigms available today.

# 3.0 Learning Content

## 3.1 Categories of Programming Language

There are three categories of programming languages:

1. Machine languages;
2. Assembly languages; and
3. High-level languages.

Machine languages and assembly languages are also called low-level languages. A Machine language program consists of a sequence of zeros and ones. Each kind of CPU has its own machine language.

**Advantages**

1. Fast and efficient
2. Machine oriented
3. No translation required

**Disadvantages**

1. Not portable
2. Not programmer friendly

**Assembly language programs:** use mnemonics to represent machine instructions. Each statement in assembly language corresponds to one statement in machine language. Assembly language programs have the same advantages and disadvantages as machine language programs. Compare the following machine language and assembly language programs:

| 8086 Machine language program for<br><br>var1 = var1 + var2 | 8086 Assembly program for<br><br>var1 = var1 + var2 |
|---|---|
| 1010 0001 0000 0000 0000 0000 | MOV AX , var1 |
| 0000 0011 0000 0110 0000 0000 0000 0010 | ADD AX , var2 |
| 1010 0011 0000 0000 0000 0000 | MOV var1 , AX |

A high-level language (HLL) has two primary components:

1. a set of built-in **language primitives** and **grammatical rules;** and

2. a **translator**

***A HLL language program*** consists of English-like statements that are governed by a strict syntax.

**Advantages**

1. Portable or machine independent;
2. Programmer-friendly.

**Disadvantages**

1. Not as efficient as low-level languages;
2. Need to be translated.

Examples: C, C++, Java, FORTRAN, Visual Basic, and Delphi.

## Self-Assessment Question(s)

## Self-Assessment Answer

Please insert relevant answers

# 4.0 History of Programming Languages

Before high level programming languages existed, computers were programmed one instruction at a time using binary or hex. This was a tedious job and there were a lot of errors. Programs were difficult to read, and modification was extremely difficult because all programs had to be written using absolute addressing. Obviously, this job did not attract many people, so there was a shortage of programmers. Expensive computers sat idle for long periods of time while software was being developed. Software often cost two to four times as much as the computer. This led to the development of assemblers and assembly languages. Programming became somewhat easier, but many users still wanted floating point numbers and array indexing. Since these capabilities were not supported in hardware, high level languages had to be developed to support them.

The original computer programming languages, as mentioned above, were so-called machine languages: the human and computer programmed in same language. Machine language is great for computers but not so great for humans since the instructions are each very simple and so many, many instructions arerequired. High-level languages were introduced for ease of programmability by humans. FORTRAN was the first high-level language, developed in 1957 by a team led by Backus at IBM. FORTRAN programs were translated (compiled) into machine language to be executed. They didn't run as fast as hand-codedmachine language programs, but FORTRAN nonetheless caught on very quickly because FORTRAN programmers were much more productive. A swarm of early languages followed: ALGOL in '58, Lisp in the late50's, PL/1 in the early 60's, and BASIC in 1964.

**The 1950s and 1960s**

In the 1950s, the first the modern programming languages whose descendants are still in widespread use today were designed, these programming languages lets the programmer concentrate on solving a problem without having to worry so much about the computer's hardware. A drawback though is the slowercomputing time, as the code will first be translated into machine code and then be run.

**FORTRAN - 1957**

The first big language was Fortran (FORmula TRANslation) which was mainly intended and used for scientific computing, the military projects and later the Space Program being prominent examples. It was the first language to introduce data types, such as boolean, integer, real and double-precision numbers. It spawned a whole lot of dialects which adapted the language and kept it up to date in the course of the years and is widely used up until today.

**COBOL - 1959**

Whilst Fortran was great for number crunching it did lack a proper I/O, and in 1959 the Conference on Data Systems and Languages (CODASYL) got together and agreed on Cobol as a language for the business market. It only recognized numbers and strings of text, which were grouped together into arrays providing for good data handling.
Although widely criticized for being just a short term solution to problems perceived in the 1950's and significantly better languages existing on the market today, the language is still in widespread use, due to legacy systems having been dragged along from the very beginning.

**ALGOL – 1958**

**ALGOL is shor for ALGO**rithmic **L**anguage, Implemented in a formal grammar, the Backus-Naur-Form, Algol is the ancestor of pretty much every programming language in use today, in that it is the first block-structured language. Originally intended for scientific computing it soon got replaced by the languages it
Spawned.

**Lisp - 1958**

Lisp is short for List Processing and is quite different from the other languages of that time, or any time even, in that it relies heavily on linked lists as the main data structure. Even the source code itself is made up of lists, thus making the language easy to manipulate.
Lisp is closely linked to AI research.

Some important languages that were developed in this period include:

- 1951 - Regional Assembly Language
- 1952 - Auto code
- 1954 - IPL (forerunner to LISP)
- 1955 - FLOW-MATIC (forerunner to COBOL)
- 1957 - FORTRAN (First compiler)

- 1957 - COMTRAN (forerunner to COBOL)
- 1958 - LISP
- 1958 - ALGOL 58
- 1959 - FACT (forerunner to COBOL)
- 1959 - COBOL
- 1959 - RPG
- 1962 - APL
- 1962 - Simula
- 1962 - SNOBOL
- 1963 - CPL (forerunner to C)
- 1964 - BASIC
- 1964 - PL/I
- 1967 - BCPL (forerunner to C)

**The Period From 1967-1978:**

The period from the late 1960s to the late 1970s brought a major flowering of programming languages. Most of the major language paradigms now in use were invented in this period.The 1960s and 1970s also saw considerable debate over the merits of "structured programming", which essentially meant programming without the use of Go to. This debate was closely related to language design: some languages did not include GOTO, which forced structured programming on the programmer. Although the debate raged hotly at the time, nearly all programmers now agree that, even in languages that provide GOTO, it is bad programming style to use it except in rare circumstances. As a result, later generations of language designers have found the structured programming debate tedious and even bewildering.

Some important languages that were developed in this period include:

- 1968 - Logo
- 1969 - B (forerunner to C)
- 1970 - Pascal
- 1970 - Forth
- 1972 - C
- 1972 - Smalltalk
- 1972 - Prolog
- 1973 - ML
- 1975 - Scheme
- 1978 - SQL (initially only a query language, later extended with programming constructs

**The 1980s**

The 1980s were years of relative consolidation in imperative languages. Rather than inventing new paradigms, all of these movements elaborated upon the ideas invented in the previous decade. Although major new paradigms for imperative programming languages did not appear, many researchers expanded on the ideas of prior languages and adapted them to new contexts.

Some important languages that were developed in this period include:

- 1980 - C++ (as C with classes, name changed in July 1983)
- 1983 - Ada
- 1984 - Common Lisp
- 1984 - MATLAB
- 1985 - Eiffel
- 1986 - Objective-C
- 1986 - Erlang
- 1987 - Perl
- 1988 - Tcl
- 1988 - Mathematica
- 1989 - FL (Backus);

**The 1990s: the Internet age**

The rapid growth of the Internet in the mid-1990s was the next major historic event in programming languages. By opening up a radically new platform for computer systems, the Internet created an opportunity for new languages to be adopted. In particular, the Java programming language rose to popularity because of its early integration with the Netscape Navigator web browser, and various scripting languages achieved widespread use in developing customized application for web servers.

The 1990s saw no fundamental novelty in imperative languages, but much recombination and maturation of old ideas. This era began the spread of functional languages. A big driving philosophy was programmer productivity. Many "rapid application development" (RAD) languages emerged, which usually came with an IDE, garbage collection, and were descendants of older languages. All such languages were object-oriented. These included Object Pascal, Visual Basic, and Java. Java in particular received much attention.

More radical and innovative than the RAD languages were the new scripting languages. These did not directly descend from other languages and featured new syntaxes and more liberal incorporation of features. Many consider these scripting languages to be more productive than even the RAD languages, but often because of choices that make small programs simpler but large programs more difficult to write and maintain. Nevertheless, scripting languages came to be the most prominent ones used in connection with the Web.

Some important languages that were developed in this period include:

- 1990 - Haskell
- 1991 - Python
- 1991 - Visual Basic
- 1991 - HTML
- 1993 - Ruby
- 1993 - Lua
- 1994 - CLOS (part of ANSI Common Lisp)
- 1995 - Java
- 1995 - Delphi (Object Pascal)
- 1995 - JavaScript
- 1995 - PHP
- 1996 - WebDNA
- 1997 - Rebol
- 1999 –D

### Self-Assessment Question(s)

### Self-Assessment Answer

Please insert relevant answers

## 5.0 CONCLUSION

In this unit we took a look at an overview of programming language history to prepare us for the work in this course program organization and design

## 6.0 SUMMARY

In introducing this module, it was stated that **programming language** is an artificial language designed to communicate instructions to a machine, particularly a computer. Programming languages can be used to create programs that control the behavior of a machine and/or to express algorithms precisely.Andthe following were discussed:

1. categories of programming language
2. history of programming language

## 7.0 TUTOR-MARKED ASSIGNMENT

1. List two advantages and two disadvantages of low-level languages.
2. Explain the similarities and differences between an assembly language and a machine language.
3. List five important languages that were developed within the period of 1950s and 1960s

## 8.0 REFERENCES/FURTHER READING

Thomas J. Bergin and Richard G. Gibson. History of Programming Languages (Pro- ceedings). Addison-Wesley, 1993

Richard L. Wexelblat. History of Programming Languages (Proceedings). Academic Press, 1981.

# Module 2

## Programming Paradigms

# Unit 2

## Categories of Programming Language

Contents



## 1.0 Introduction

This implies main styles of programming, Programming languages are classified in accordance with the main style and techniques supported. Some programming languages are specifically designed for use in certain applications. Different programming languages follow different

approaches to solving programming problems. A programming paradigm is an approach to solving programming problems.

## 2.0 Learning Outcome

At the end of this unit, you should be able to:

1. be able to list four programming paradigms and describe their strengths and weaknesses;
2. evaluate the tradeoffs between the different paradigms, considering such issues as space efficiency, time efficiency (of both the computer and the programmer), safety, and power of expression;
3. identify at least one outstanding and distinguishing characteristic for each of the programming paradigms covered in this unit; and
4. Evaluate the tradeoffs between the different paradigms.

## 3.0 Learning Content

## 3.1 Classification of Programming Paradigms

A programming paradigm may consist of many programming languages. There are four main paradigms, of the four, functional and object-oriented are the most common, with imperative creeping its way into almost all modern languages as well. Logic programming is something that is less seen except in the AI (Artificial Intelligence) realm. We can classify languages according to paradigms of their kernels.

**The following is a classification of several famous languages against the main paradigms:**

1. Imperative paradigm: Algol, Pascal, C, Ada; COBOL, PL/1;
2. Functional paradigm: Lisp, Refal, Planner, Scheme, ML, APL;
3. Logic paradigm: Prolog;
4. Object-oriented paradigm: Smalltalk, Eiffel, Simula, C++, Java

Other possible programming paradigms

1. The visual paradigm

2. One of the parallel paradigms
3. The constraint based paradigm

We could notice that Smalltalk, the first object-oriented language, is not popular because of complexity of its syntax and dynamic semantics. But its basic object ideas (abstraction of object's state and behavior, encapsulation and inheritance of state and behavior, polymorphism of operations) are easily integrated with the principles of programming languages of the other styles. For this reason, the object-oriented paradigm became widespread as soon as it was combined with traditional imperative paradigm. To be more precise, it became widespread when it was embedded into the popular imperative languages C and Pascal, thereby giving imperative object-oriented languages C++ and Object Pascal.

### 3.1.1 Imperative Programming Paradigm

The ***imperative*** (procedural) programming paradigm is the oldest and the most traditional one. It has grown frommachine and assembler languages, whose main features reflect the John von Neuman's principles of computerarchitecture. An imperative program consists of explicit commands (instructions) and calls of procedures(subroutines) to be consequently executed; they carry out operations on data and modify the values of programvariables (by means of assignment statements), as well as external environment. Within this paradigm, variables
are considered as containers for data similar to memory cells of computer memory.
In this paradigm, a program is a series of *statements* containing *variables*.
Program execution involves changing the memory contents of the computer continuously.
Example of imperative languages are: C, FORTRAN, Pascal, COBOL etc

**Advantages**
1. Low memory utilization
2. Relatively efficient
The most common form of programming in use today.

**Disadvantages**
1. Difficulty of reasoning about programs
2. Difficulty of parallelization.
3. Tend to be relatively low level.

# 3.1.2 Functional Paradigm

The ***functional*** paradigm is in fact an old style too, since it has arisen from evaluation of algebraic formulae, andits elements were used in first imperative algorithmic languages such as Fortran. Pure functional program is acollection of mutually related (and possibly recursive) functions. Each function is an expression for computing avalue and is defined as a composition of standard (built-in) functions. Execution of functional program is simplyapplication of all functions to their arguments and thereby computation of their values.
A program in this paradigm consists of *functions* and uses functions in a similar way as used in mathematics. Program execution involves functions calling each other and returning results. There are no variables in functional languages.

Example functional languages include: ML, Miranda™, Haskell

**Advantages**

1. Small and clean syntax.
2. Better support for reasoning about programs.
3. They allow functions to be treated as any other data values.
4. They support programming at a relatively higher level than the imperative languages.

**Disadvantages**

1. Difficulty of doing input-output.
2. Languages use more storage space than their imperative cousins.

## Self-Assessment Question(s)

## Self-Assessment Answer

Please insert relevant answers

## 3.1.3 Logic Paradigm

Within the ***logic*** paradigm, program is thought of as a set of logic formulae: axioms (facts and rules) describingproperties of certain objects, and a theorem to be proved. Program execution is a process of logic proving(inference) of the theorem through constructing the objects with the described properties.

The essential difference between these three paradigms concerns not only the concept of program and itsexecution, but also the concept of program variable. In contrast with imperative programs, there are neitherexplicit assignment statements nor side effects in pure functional and logic programs. Variables in such aprogram are similar to those in mathematics: they denote actual values of function arguments or denote objectsconstructed during the inference. This peculiarity explains why functional and logic paradigms are considered asnon-traditional.

A program in the logic paradigm consists of a set of *predicates* and *rules of inference*.
Predicates are statements of fact like the statement that says: water is wet.
Rules of inference are statements like: If X is human then X is mortal.
The predicates and the rules of inference are used to prove statements that the programmer Supplies. Example: Prolog

**Advantages**

**1.** Good support for reasoning about programs
2. Can lead to concise solutions to problems

**Disadvantages**

**1.** Slow execution.
**2.** Limited view of the world.
**3.** That means the system does not know about facts that are not its predicates and rules of inference.
**4.** Difficulties in understanding and debugging large programs.

### 3.1.4 Object Oriented Paradigm

Within the ***object-oriented*** paradigm, a program describes the structure and behavior of so called objects and classes of objects. An object encapsulates passive data and active operations on these data: it has a storage fixing its state (structure) and a set of methods (operations on the storage) describing behavior of the object.

Classes represent sets of objects with the same structure and the same behavior. Generally, descriptions of classes compose an inheritance hierarchy including polymorphism of operations. Execution of an object-oriented program is regarded as exchange of messages between objects, modifying their states.
A program in this paradigm consists of *objects* which communicate with each other by *sending messages*

Example object oriented languages include: Java, C#, Smalltalk, etc

**Advantages**

1. Conceptual simplicity

2. Models computation better.

3. Increased productivity.

**Disadvantages**

1. Can have a steep learning curve, initially.

2. Doing I/O can be cumbersome

Programming techniques elaborated within corresponding programming style and programming languages have its own scope of adequate applications. Functional programming is preferable for symbolic processing, while logic programming is useful for deductive databases and expert systems, but both of them are not suitable for interactive tasks or event-driving applications. Imperative languages are equally convenient for numeric and symbolic computations, giving up to most of the functional languages and Prolog in the power of symbolic processing techniques. The object paradigm is useful for creating large programs (especially interactive) with complicated behavior and with various types of data.

We have outlined main programming paradigms, as well as programming techniques and programming languages elaborated within them. Programming techniques of traditional imperative paradigm essentially differ from techniques of nontraditional ones – functional and

logic. They have different scopes of applicability, and for this reason necessity to integrate techniques of different paradigms often arises in programming projects.
The question then is which of these paradigms is the best?

1. The most accurate answer is that there is no best paradigm. No single paradigm will fit all problems well.

2. Human beings use a combination of the models represented by these paradigms.
3. Languages with features from different paradigms are often too complex.
4. So, the search of the ultimate programming language continue.

## 3.2 Pseudocode examples comparing two paradigms

A pseudocode comparison of imperative and object oriented approaches used to calculate the area of a circle ($\pi r^2$.)

| Imperative | Object Oriented |
|---|---|
| load r;<br>r2 = r * r;<br>**result = r2 * "3.142";** | circle.area method(r2):<br>push stack<br>load r2;<br>r3 = r2 * r2;<br>res = r3 * "3.142";<br>pop stack<br>**return(res);** 12,13<br>.....<br>main proc:<br>load r; 1<br>result = circle.area(r);<br>+allocate heap storage; 2[See 1]<br>+copy r to message; 3<br>+load p = address of message; 4<br>+load v = addr. of method 'circle.area' 5<br>+goto v with return; 6<br>.<br>.<br>.... storage .....<br>result variable (assumed pre-allocated)<br>immutable variable "3.142" (final)<br>(heap) message variable for circle method call<br>vtable(==>area)<br>stack storage |

Note that the actual arithmetic operations used to compute the area of the circle are the same in the two paradigms, with the difference being that object-oriented paradigms wrap those operations in a subroutine call that makes the computation general and reusable.

### Self-Assessment Question(s)

### Self-Assessment Answer

Please insert relevant answers

## 4.0 CONCLUSION

In this unit we took a look at an overview of programming language paradigms So that the students can be able to select an appropriate language for a task.

## 5.0 SUMMARY

In introducing this module, it was stated that **programming language** paradigm implies main styles of programming, *Programming languages are classified in accordance with the main style and techniques* supported. And the following were discussed :categories of programming language paradigms.

1. Imperative paradigm
2. Functional paradigm
3. Logic Paradigm
4. Object- Oriented paradigm
5. Pseudo code example

## 6.0 TUTOR-MARKED ASSIGNMENT

1. Mention the programming paradigm to which each of the following languages belongs: Visual Basic, Java, C#, Haskell, Lisp, Prolog, Pascal.

2. Which programming paradigms give better support for reasoning about programs?
3. Which programming paradigms give better support for doing I/O.

## 7.0 REFERENCES/FURTHER READING

[Field, 1988] Field, A., Harrison P. Functional Programming. Addison-Wesley, 1988.
[Finkel, 1996] Finkel, R.A. Advanced Programming Language Design. Addison-Wesley Publ. Comp., 1996.
Steele, 1990] Steele, G. L. Common Lisp – the Language, 2nd edit. Digital Press, 1990.
Stroustrup, 1997] Stroustrup, B. The C++ Programming Language, 3rd edition. Addison-Wesley. 1997

# Module 3

## SCALING

# Unit 3

## History of Programming Language

Contents

## 1.0 Introduction

In early computers decimal floating-point in hardware was unstandardized and relatively rare. As a result, programming languages with decimal types describe a decimal number as an integer which is scaled (divided) by a power of ten (in other words, effectively encoding decimal values as rational numbers). The number 2.50, for example, is held as the integer 250 with a scale of 2; the scale is therefore simply a negative exponent. Precision is the number of digits in a number. Scale is the number of digits to the right of the decimal point in a number. For example, the number 123.45 has a precision of 5 and a scale of 2.

Despite the widespread use of binary arithmetic, decimal computation remains essential for many applications. Not only is it required whenever numbers are presented for human inspection, but it is also often a necessity when fractions are involved. Decimal fractions (rational numbers whose denominator is a power of ten) are pervasive in human endeavours, yet most cannot be represented by binary fractions; the value 0.1, for example, requires an infinitely recurring binary number. If a binary approximation is used instead of an exact decimal fraction, results can be incorrect even if subsequent arithmetic is exact.

## 2.0 Learning Outcome

By the end of this unit, you should be able to:

1. be able to describe decimal and binary digits;
2. be able to state how numbers are stored in computer;
3. be able to convert integer decimal to binary and binary to decimal;
4. be able to convert fractions to binary;
5. state how a float is represented in computer; and
6. describe the word storage error.

## 3.0 Learning content

## 3.1 Decimal Numbers

Since the objective of most computer systems is to process data, it is important to understand how data is stored and interpreted by the software. Decimal data accept two optional parameters, *precision* and *scale*, enclosed within parentheses and separated by a comma:

*datatype* [(*precision* [, *scale* ])]

Each combination of **precision** and **scale** is defined as a distinct data type. For example, *numeric*(10,0) and *numeric*(5,0) are two separate data types. The precision and scale determine the range of values that can be stored in a *decimal* or *numeric* column:

1. **precision** specifies the maximum number of decimal digits that can be stored in the column. It includes all digits to the right or left of the decimal point. You can specify a precision of 1 – 38 digits or use the default precision of 18 digit; and
2. **scale** specifies the maximum number of digits that can be stored to the right of the decimal point. The scale must be less than or equal to **precision**. You can specify a scale of 0 – 38 digits or use the default scale of 0 digits.

Exact numeric types with a scale of 0 displays without a decimal point. You cannot enter a value that exceeds either the precision or the scale for the column.

The storage size for a *numeric* or *decimal* column depends on its precision. The minimum storage requirement is 2 bytes for a 1- or 2-digit column. Storage size increases by 1 byte for each additional 2 digits of precision, to a maximum of 17 bytes.

### 3.1.1 Storage/Manipulation of Decimal Numbers

Traditionally, calculation with decimal numbers has used exact arithmetic, where the addition of two numbers uses the largest scale necessary, and multiplication results in a number whose scale is the sum of the scales of the operands (1.25 × 3.42 gives 4.2750, for example). However, as applications and commercial software products have become increasingly complex, simple rational arithmetic of this kind has become inadequate.

Repeated multiplications require increasingly long scaled integers, often dramatically slowing calculations as they soon exceed the limits of any available binary or decimal integer hardware.

Further, even financial calculations need to deal with an increasingly wide range of values. For example, telephone calls are now often costed in seconds rather than minutes, with rates and taxes specified to six or more fractional digits and applied to prices quoted in cents. Interest rates are now commonly compounded daily, rather than quarterly, with a similar requirement for values which are both small and exact. And, at the other end of the range, the Gross National Product of a country such as the USA (in cents) or Japan (in Yen) needs 15 digits to the left of the decimal point. The manual tracking of scale over such wide ranges is difficult, tedious, and very error-prone. The obvious solution to this is to use a floating-point arithmetic.

The use of floating-point may seem to contradict the requirements for exact results and preservation of scales in commercial arithmetic; floating-point is perceived as being approximate, and normalization loses scale information.
Numbers are stored on the computer in binary form. In other words, information is encoded as a sequence of 1's and 0's. On most computers, the memory is organized into 8-bit bytes. This means each 8-bit byte stored in memory will have a separate address. This extensive use of decimal data suggested that it would be worthwhile to study how the data are used and how decimal arithmetic should be defined.

### Self-Assessment Question(s)

### Self-Assessment Answer

Please insert relevant answers

## 3.2 Binary Number System

Any real number can be represented in any base, humans use base 10, but computer hardware generally uses base 2 representation.
Base 10 digits 0 1 2 3 4 5 6 7 8 9
Base 2 digits 01
Recall that in base 10, the digits of a number are just coefficients of power of the base (10) example
$417 = 4* 10^2 + 1 * 10 + 7 * 10^0$
Similarly in base 2, the digits of a number are just coefficients of powers of the base( 2) example
$1011 = 1*2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$
The base 10 system is also known as decimal system while base 2 is referred to as binary.
How can we convert an integer from decimal to binary? Here is a simple algorithm
WhileN> 0 Do
Write N% 2 // remainder when N is divided by 2,
N ←N/2 // divide N by 2,
Note that the remainder will always be either 0 or 1, a binary digit or bit. The resulting sequence of bits is the binary representation of the integer N.
Example: Find the binary representation of the decimal integer N = 23

| Integer | Remainder |
|---|---|

| | |
|---|---|
| 23 | |
| 11 | 1 |
| 5 | 1 |
| 2 | 1 |
| 1 | 0 |
| 0 | 1 |

So the decimal integer N = 23 is represented as 10111
Lets check
$10111 = = 1*2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0$
$= 16 + 4 + 2 + 1 = 23$
Note: we just shifted from base 2 to base 10

## Self-Assessment Question(s)

## Self-Assessment Answer

Please insert relevant answers

# 3.3 Converting Fractions

How can we convert a fraction from decimal to binary? Here is a simple algorithm
While F! = 0 Do
Multiply F by 2
Record the "Carry" across the decimal point
F < -- the fraction part
End While
Note that the carry will always be either 0 or 1, a binary digit or bit. The resulting sequence of bits is the binary representation of the fraction F.

~~Example: find the binary representation of the decimal fraction~~

F = 0.3125

| Carry | Fraction |
|---|---|
| | .3125 |

| | |
|---|---|
| 0 | 6250 |
| 1 | 2500 |
| 0 | 5000 |
| 1 | 0000 |

So the decimal fraction F = 0.3125 is represented as .0101
Let's check
$.0101 = 0*2^{-1} + 0 * 2^{-2} + 0 * 2^{-3} + 1 * 2^{-4}$
= .25 + .0625 + .3125: we just shifted from base 2 to base 10

## 3.4 Converting a General Decimal Number

A general decimal number like 43.375 would be converted to binary by converting the integer and fractional parts separately (as shown earlier) and combine the result.

The decimal integer 43 would be represented as 101011. The fractional part .375 would be represented in binary as 011. So 43.375 would be represented in binary as 101011.011.

## 3.5 Binary Representation and Precision

Many decimal fractions have infinite decimal expansions. The same is true of binary representation, but there are some surprising differences. Example, consider the decimal fraction 0.1, how would this be represented in binary?

| Carry | Fraction |
|---|---|
| | .1 |
| 0 | .2 |
| 0 | .4 |
| 0 | .8 |
| 1 | .6 |
| 1 | .2 |
| 0 | .4 |
| 0 | .8 |
| 1 | .6 |
| 1 | .2 |
| | --- |

Clearly this pattern will now repeat forever. So 0.1 would be represented in binary as 0.00011001100110011---

### Self-Assessment Question(s)

### Self-Assessment Answer

Please insert relevant answers

## 3.6 Float Representation

A float is stored in scientific form (but in binary). Supposed that we have a float variable:

Float X = 0.1, what would actually be stored in hardware?

So 0.1 would be represented exactly in binary as 0.00011001100110011---

A float is stored in scientific form (but in binary)

0.00011001100110011--- = (1.1001100110011---) * $2^{-100}$

The exponent is stored using 7 bits and the fractional part is stored using 23 bits( with two bits used for the signs). We don't store the first '1', so 0.1 would be stored as

-0000100 + .10011001100110011001100

Exponent Mantissa (fractional part)

## 3.7 Storage Error

So 0.1 would be stored in hardware as: -0000100 + .10011001100110011001100

Converting that to decimal, you have the value 0.999999940395355 that's fairly close, but not quite equal to 0.1. This called storage error (or conversion error). This is typical most real numbers cannot be stored exactly as floats or even as doubles. Using a double will improve accuracy since a double store 53 bits for the mantissa, but there will still be inevitable storage error.

### Self-Assessment Question(s)

### Self-Assessment Answer

Please insert relevant answers

## 4.0 Conclusion

In this unit we took a look at the extensive use of decimal data in programming So that the students can study how the decimal data are used and how decimal arithmetic are stored in the hardware of computer.

## 5.0 Summary

In introducing this module, it was stated that Decimal data accept two optional parameters, *precision* and *scale*. Precision is the number of digits in a number. Scale is the number of digits to the right of the decimal point in a number. For example, the number 123.45 has a precision of 5 and a scale of 2. And the following were discussed:

1. Decimal Numbers
2. Storage/Implementation Of Decimal Numbers
3. Binary Number System
4. Converting Fractions
5. Converting A General Decimal Number
6. Binary Representation And Precision
7. Float Representation
8. Storage Error

## 6.0 TUTOR-MARKED ASSIGNMENT

1. Write an algorithm that converts a decimal number to binary coded decimal representation?
2. How can 0.1 be stored in hardware?
3. Explain the term storage error

## 7.0 REFERENCES/FURTHER READING

Principles of Programming Languages by Macclennan, 3rd edition, 1999, Oxford
Concept of Programming Languages, 8th edition by Robert W. Sebesta, Addison Wesley, ISBN -13:978-0321-49362-0, ISBN-10:0-321-49362-1

# Module 4

## Language Design

# Unit 1

# Programming Language Properties

CONTENTS

# 1.0 INTRODUCTION

The goal of a programming language is to make it easier to build software. A programming language can help make software flexible, correct, and efficient. How good are today's languages and what hope is there for better languages in the future?

Many programming languages are in use today, and new languages are designed every year; however, the quest for a language that fits all software projects has so far come up short. Each language has its strengths and weaknesses, and offers ways of dealing with some of the issues that confront software development today.

**Which Language is the best?**

There is no general agreement in language design. According to Hebert Meyer:
"*No programming language is perfect. There is not even a single best language; there are only languages well suited or perhaps poorly suited for particular purposes*."
Moreover, he also stated that: *"A useful language needs arrays, pointers and a generic mechanism for building data structures"*

# 2.0 Learning Outcome

By the end of this unit, you should be able to:
1. list four properties of programming language and describe them;
2. evaluate the tradeoffs between the different programming principles considering such issues as efficiency and regularity.

# 3.0 Learning Content

## 3.1 Programming Language Properties

Programming languages have four properties:
– Syntax
– Names
– Types
– Semantics
For any language:
– Its designers must define these properties
– Its programmers must master these properties

### A. Syntax

The *syntax* of a programming language is a precise description of all its grammatically correct programs. When studying syntax, we ask questions like:

– What is the grammar for the language?
– What is the basic vocabulary?
– How is syntax errors detected?

### B. Names

Various kinds of entities in a program have names: *variables, types, functions, parameters, classes, objects, …*
Named entities are bound in a running program to:
– *Scope*
– *Visibility*
– *Type*
– *Lifetime*

### C. Types

A *type* is a collection of values and a collection of operations on those values.
• Simple types
– *numbers, characters, booleans, …*
• Structured types
– *Strings, lists, trees, hash tables, …*
• A language's *type system* can help to:
– *Determine legal operations*
– *Detect type errors*
– *Optimize certain operations*

### D. Semantics

The meaning of a program is called its *semantics*. In studying semantics, we ask questions like:
– *When a program is running, what happens to the values of the variables?*
– *What does each statement mean?*
– *What underlying model governs run-time behavior, such as function call?*
– *How are objects allocated to memory at run-time?*

## Self-Assessment Question(s)

## Self-Assessment Answer

Please insert relevant answers

## 3.2 History and Design Criteria

1. At the beginning: Effeciency of execution FORTRAN

2. Next step: Readability COBOL, Algol60
3. Next step: General and orthogonal Simula67, Algol68
4. Difficult to understand, less predictable, inefficient implementations.Wrong way?
5. Next step: Simplicity and abstraction Pascal, C, Euclid, CLU, Modula-2, Ada
proof of correctness of programs - limited success
6. Next step: Reliability ML, Haskel

7. FORTRAN: Efficiency in execution
8. COBOL: English-like nontechnical readability
9. Algol60: Block-structured language
10. Pascal: simple instructional language to promote top-down design
11. C++: better abstraction while preserving efficiency and compatibility with C

What then makes a successful language? The Key characteristics include:

– *Simplicity and readability*
– *Clarity about binding*
– *Reliability*
– *Support*
– *Abstraction*
– *Orthogonality*
– *Efficient implementation*

## Self-Assessment Question(s)

## Self-Assessment Answer

Please insert relevant answers

## 3.3 Language Design principles

**A. Efficiency**

This was "first" goal in FORTRAN: *execution efficiency*.

It is still an important goal in some settings (C++, C).Many other criteria can be interpreted from the point of view of efficiency:

1. programming efficiency: writability or expressiveness (ability to express complex processes and structures)
2. reliability (security).
3. maintenance efficiency: readability- This was seen as a goal for the first time in Cobol

**Other Kinds of Efficiency**

1. Efficiency of execution (optimizable)
2. Efficiency of translation. Are there features which are extremely difficult to check at compile time (or even run time)? e.g. Alogol – prohibits assignment to dangling pointers
3. Implementability (cost of writing translator)

**B. Regularity:** How well the features of a language are integrated

1. Regularity is a measure of how well a language integrates its features, so that there are no unusual restrictions, interactions, or behavior. Easy to remember.
2. Regularity issues can often be placed in subcategories:3.
3. Generality: are constructs general enough? (Or too general?)
4. Orthogonality: are there strange interactions?
5. Uniformity: Do similar things look the same, and do different things look different?

**Features that Lack Generality**

1. procedures and functions in Pascal: that can be passed as parameters but there are no procedure variable
2. Fixed-length array in Pascal
3. Operator "==" in C: can't compare structured data
4. No named constants in FORTRAN

**Features that Lack Orthogonality**

1. Pascal: functions can only return scalar or pointer
2. C: local variables must be defined a the begining of a block
3. C: Pass all parameters by value except arrays

**Examples of non Orthogonality in C++**

1. We can convert from integer to float by simply assigning a float to an integer, but not vice versa. (not a question of ability to do – generality, but of the way it is done)
2. Arrays are pass by reference while integers are pass by value.
3. A switch statement works with integers, characters, or enumerated types, but not doubles or Strings.

**Features that Lack Uniformity**

In C++:

1. class A {....}; // Semicolon required
2. int f () {} // Semicolon forbiden
3. Pascal: return values from function look like assignment
4. C: operator "&" and "&&"

**Examples of lack of Uniformity is C++**

1. functions are not general: there are no local functions (simplicity of environment).

2. declarations are not uniform: data declarations must be followed by a semicolon, function declarations must not.

3. lots of ways to increment – lack of uniformity (++i, i++, i = i+1)

4. i=j and i==j look the same, but are different. Lack of uniformity

**Other Language Design Principles**

**1.** Simplicity: Pascal
2. Expressiveness (conciseness): C
3. Extensibility: ML
4. Restrictability
5. Consistency with Accepted Notations and Conventions:
6. FORTRAN: DO 99 I = 1.10
7. Preciseness: existence of a precise definition for the language
8. Machine independence
9. Security

### Self-Assessment Question(s)

### Self-Assessment Answer

Please insert relevant answers

## 4.0 Conclusion

In this module we took a look at the properties and principles of programming language to give students a better understanding of programming language features..

## 5.0 Summary

In introducing this module, it was stated that programming language is characterized by it properties such as:

- Efficiency
- Regularity
- Simplicity: Pascal
- Expressiveness (conciseness): C
- Extensibility: ML
- Restrictability
- Consistency with Accepted Notations and Conventions: FORTRAN: DO 99 I = 1.10
- Preciseness: existence of a precise definition for the language
- Machine independence
- Security

## 6.0 TUTOR-MARKED ASSIGNMENT

Discuss regularity as one the programming principles in terms of its generality, orthorgonality and uniformity.

## 7.0 REFERENCES/FURTHER READING

Principles of Programming Languages by Macclennan, 3rd edition, 1999, Oxford
Concept of Programming Languages, 8th edition by Robert W. Sebesta, Addison Wesley, ISBN -13:978-0321-49362-0, ISBN-10:0-321-49362-1

# Module 5

## Data Types

# Unit 1

## Data Types

Contents

## 1.0 Introduction

In computer science, **primitive data type** is either of the following:

1. a *basic type* is a data type provided by a programming language as a basic building block. Most languages allow more complicated composite types to be recursively constructed starting from basic types.
2. a *built-in type* is a data type for which the programming language provides built-in support.

In most programming languages, all basic data types are built-in. In addition, many languages also provide a set of composite data types. Depending on the language and its implementation, primitive data types may or may not have a one-to-one correspondence with objects in the computer's memory. However, one usually expects operations on basic primitive data types to be the fastest language constructs there are. Integer addition, for example, can be performed as a single machine instruction, and some processors offer specific instructions to process sequences of characters with a single instruction. In particular, the **C** standard mentions that "a 'plain' int object has the natural size suggested by the architecture of the execution environment". This means that int is likely to be 32 bits long on a 32-bit architecture. Basic primitive types are almost always value types.

Most languages do not allow the behavior or capabilities of primitive (either built-in or basic) data types to be modified by programs. Exceptions include Smalltalk, which permits all data types to be extended within a program, adding to the operations that can be performed on them or even redefining the built-in operations.

## 2.0 Learning Outcome

At the end of this unit, you should be able to:
1. be able to list the basic data types;
2. be able to state the difference between Primitive data types and Abstract data types; and
3. Describe the importance of Data Abstraction in programming.

## 3.0 Learning content

## 3.1 Overview of Primitive Data Types

The actual range of primitive data types that is available is dependent upon the specific programming language that is being used. For example, in C, strings are a composite but built-in

data type, whereas in modern dialects of BASIC and in JavaScript, they are assimilated to a primitive data type that is both basic and built-in.

**Classifications of basic primitive types may include:**

1. character (character, char);
2. integer (integer, int, short, long, byte) with a variety of precisions;
3. floating-point number (float, double, real, double precision);
4. fixed-point number (fixed) with a variety of precisions and a programmer-selected scale.
5. boolean, logical values **true** and **false**; and
6. reference (also called a *pointer* or *handle*), a small value referring to another object's address in memory, possibly a much larger one.

More sophisticated types which can be built-in include:

1. Tuples in ML, Python;
2. Linked lists in Lisp;
3. Complex numbers in Fortran, C (C99), Lisp, Python;
4. Rational numbers in Lisp, Perl 6;
5. Hash tables in various guises, in Lisp, Perl, Python;
6. First-class functions, closures, continuations in languages that support functional programming such as Lisp, ML, Perl 6, D.

**Specific Primitive Data Types**

**Integer Numbers**

An integer data type can hold a whole number, but no fraction. Integers may be either signed (allowing negative values) or unsigned (nonnegative values only). Typical sizes of integers are:

| Size | Names | Signed Range | Unsigned Range |
|---|---|---|---|
| 8 bits | Byte | −128 to +127 | 0 to 255 |
| 16 bits | Word, short int | −32,768 to +32,767 | 0 to 65,535 |
| 32 bits | Double Word, long int (win32, win64, 32-bit Linux) | −2,147,483,648 to +2,147,483,647 | 0 to 4,294,967,295 |
| 64 bits | long int (C in 64-bit linux), long long (C), long (Java, the signed integer | −9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 | 0 to 18,446,744,073,709,551,615 |

| | | | |
|---|---|---|---|
| | variant only)) | | |
| unlimited | Bignum | | |

Literals for integers consist of a sequence of digits. Most programming languages disallow use of commas for digit grouping, although Fortran (77, 90, and above, fixed form source but not free form source) allows embedded spaces, and Perl, Ruby, and D allow embedded underscores. Negation is indicated by a minus sign (−) before the value. Examples of integer literals are:

- 42
- 10000
- −233000

**Booleans**

A Boolean type, typically denoted "bool" or "boolean", is typically a *logical type* that can be either "true" or "false". Although only one bit is necessary to accommodate the value set "true" and "false", programming languages typically implement boolean types as one or more bytes.

Most languages (Java, Pascal and Ada, e.g.) implement booleans adhering to the concept of *boolean* as a distinct logical type. Languages, though, may implicitly convert booleans to *numeric types* at times to give extended semantics to booleans and boolean expressions or to achieve backwards compatibility with earlier versions of the language. In C++, e,g., boolean values may be implicitly converted to integers, according to the mapping false → 0 and true → 1 (for example, true + true would be a valid expression evaluating to 2). The boolean type bool in C++ is considered an integral type and is a cross between numeric type and a logical type.

**Floating-Point Numbers**

A floating-point number represents a limited-precision rational number that may have a fractional part. These numbers are stored internally in a format equivalent to scientific notation, typically in binary but sometimes in decimal. Because floating-point numbers have limited precision, only a subset of real or rational numbers are exactly representable; other numbers can be represented only approximately.

Many languages have both a single precision (often called "float") and a double precision type.

Literals for floating point numbers include a decimal point, and typically use "e" or "E" to denote scientific notation. Examples of floating-point literals are:

- 20.0005
- 99.9

- −5000.12
- 6.02e23

Some languages (e.g., FORTRAN, Python, D) also have a complex number type comprising two floating-point numbers: a real part and an imaginary part.

**Fixed-Point Numbers**

A fixed-point number represents a limited-precision rational number that may have a fractional part. These numbers are stored internally in a scaled-integer form, typically in binary but sometimes in decimal. Because fixed-point numbers have limited precision, only a subset of real or rational numbers are exactly representable; other numbers can be represented only approximately. Fixed-point numbers also tend to have a more limited range of values than floating point, and so the programmer must be careful to avoid overflow in intermediate calculations as well as the final results.

**Characters and Strings**

A character type (typically called "char") may contain a single letter, digit, punctuation mark, symbol, formatting code, contro lcode, or some other specialized code (e.g., a by the order mark). Some languages have two or more character types, for example a single-byte type for ASCII characters and a multi-byte type for Unicode characters. The term "character type" is normally used even for types whose values more precisely represent code units, for example a UTF-16 code unit as in Java and JavaScript.

Characters may be combined into strings. The string data can include numbers and other numerical symbols but will be treated as text.

In most languages, a string is equivalent to an array of characters or code units, but Java treats them as distinct types (java.lang. String and char[]). Other languages (such as Python, and many dialects of BASIC) have no separate character type; strings with a length of one are normally used to represent (single code unit) characters.

Literals for characters and strings are usually surrounded by quotation marks: sometimes, single quotes (') are used for characters and double quotes (") are used for strings.

Examples of character literals in C syntax are:

- 'A'
- '4'
- '$'
- '\t' (tab character)

Examples of string literals in C syntax are:

- "A"
- "Hello World"

**Numeric Data Type Ranges**

Each numeric data type has its maximum and minimum value known as the range. Attempting to store a number outside the range may lead to compiler/runtime errors, or to incorrect calculations (due to truncation) depending on the language being used.

The range of a variable is based on the number of bytes used to save the value, and an integer data type is usually able to store $2^n$values (where $n$is the number of bits that contribute to the value). For other data types (e.g. floatingpoint values) the range is more complicated and will vary depending on the method used to store it.

There are also some types that do not use entire bytes, e.g. a boolean that requires a single bit, and represents a binary value (although in practice a byte is often used, with the remaining 7 bits being redundant). Some programming languages (such as Ada and Pascal) also allow the opposite direction, that is, the programmer defines the range and precision needed to solve a given problem and the compiler chooses the most appropriate integer or floating point type automatically.

### 3.1.1 Abstract Data Type

In a modern computer, data consists of a binary bits, but meaningful data is organized into primitive data types such as integer, real, Boolean and into more complex data structures such as arrays and binary trees.

These data types and data structures always come along with associated operations that can be done on the data. For example, a 32 bit integer data type is defined both by the fact that a value of type integer consists of 32 binary bits but also by the fact that two integer values can be added, subtracted, multiplied, compared and so on.

An array is defined both by the fact that it is a sequence of data items of the same basic type, but also by the fact that it is possible to directly access each of the positions in the list based on its numeric index. So the idea of a data type include a specification of the possible value of that type together with the operations that can be performed on those values.

An algorithm is an abstract idea and a program is an implementation of an algorithm. Similarly, it is useful to be able to work with abstract idea behind data type or data structure without getting

bogged down in the implementation details. The abstraction in this case is called an “abstract data type”.

An abstract data type specifies the values of the type, but not how those values are represented as collections of bits, and it specifies operations on those values in terms of their inputs, outputs and effects rather than as particular algorithms or program code.

An Abstract Data Type or ADT, consists of:

- a specification of the possible values of the data type
- a specification of the operations that can be performed on those values in terms of the operations, input, output and effects.

We are all used to dealing with primitive data types as abstract data types. It is quite likely that you don’t know the details of how values of type double are represented as sequences of bits. The details are infact rather complicated. However, you know how to work with double values by adding them, multiplying them, truncating them to values of integer, inputing and outputting them and so on.

When you create your own data types like stacks,queues, trees and even more complicated structures you are necessarily deeply involved in the data representation and the implementation of the operations. However, when you use those data types you don’t have to worry about the implementation. All you want is to be able to use them the way you use the built in primitive types as abstract data types. This is in conformance with one of the central principles of software engineering: encapsulation, also known as information hiding, modularity and separation of interface from implementation.

## Self-Assessment Question(s)

## Self-Assessment Answer

Please insert relevant answers

## 3.2 Stacks and Queues

Let's consider two familiar data structures, stacks and queues. Generally, a stack or queue is designed to contain items of some specified type. For example, we can have stacks of integers or queues of strings. The type of the items contained in a stack or queue is called its base type.

A possible value of a stack-the data that the stack contains at a given time-is a sequence of items belonging to its base type. A possible value of a stack of integers is a sequence of integers. Exactly the same thing could be said about a queue of integers. So, what's the difference between a stack and a queue? The difference is the set of operations defined for the data type. In particular, the difference is in the meaning of the operation that removes one item from the data structure: A stack's pop operation removes the item that has been on the stack for the shortest amount of time, while a queue's dequeue operation removes the item that has been on the queue the longest.

A stack is a last in, first out (LIFO) data structure. The item that is in line to be the next one removed from the stack is the one that was added last, that is most recently. The operations on a stack include adding and removing items. Typically, we also specify operations for making a stack empty and for testing whether it is empty. Since we consider a stack to be a stack of items of some specified base type, the base type is also part of the definition. Formally, we get a different stack ADT for each different base type. So, we define the ADT “stack of Base Type" where Base Type could be integer, string, or any other data type.

For any data type, Base Type, we define an abstract data type Stack of Base- Type. The possible values of this type are sequences of items of type BaseType (including the sequence of length zero). The operations of the ADT are:

**push(x):** Adds an item, x, of type Base Type to the stack; that is, modifies the value of the stack by appending x to the sequence of items that is already on the stack. No return value.

**pop:** Removes an item from the stack and returns that item. The item removed is the one that was added most recently by the push operation. It is an error to apply this operation to an empty stack.

**make Empty:** Removes all items from the stack. No return value.
**is Empty:** Returns a value of type boolean, which is true if the stack contains no items and is false if there is at least one item on the stack.

Note that the definition of the ADT does not just list the operations. It specifies their meaning, including their effects on the value of the stack. It also specifies the possible errors that can occur

when operations are applied. Errors are a particularly difficult thing to deal with in an abstract way. Ideally, we should know what will happen if the error occurs, but in practice this often depends on a particular implementation of the ADT. So, in the specification of an ADT, we generally won't specify the effect of using an operation incorrectly. If we are imagining an implementation in Java or C++, we should specify that the error will throw an exception of some specified type. This would allow users of the ADT to design error-handling strategies for their programs.

We will use this definition of the stack ADT, but it is not the only possible definition. In some variations, for example, the pop operation removes an item from the stack but does not return it, and there is a separate top operation that tells you the item on the top of the stack without removing it. Some implementations of stacks have a fixed maximum size. These stacks can become “full," and it is an error to try to push an item onto a full stack. Such stacks should have an isFull operation to test whether the stack is full. However, this operation would be useless (it would always return the value true) for stack implementations that don't impose a maximum size. On the whole, it seems better to leave it out of the general definition.The definition of the queue ADT is very similar,with operations named enqueue and dequeue replacing push and pop.
For any data type, BaseType, we define an abstract data type Queue of Base-Type. The possible values of this type are sequences of items of type BaseType (including the sequence of length zero). The operations of the ADT are:

**enqueue(x):** Adds an item, x, of type BaseType to the queue; that is, modifies the value of the queue by appending x to the sequence of items that is already on the queue. No return value.

**dequeue:** Removes an item from the queue and returns that item. The item removed isthe one that has been in the queue the longest. It is an error to apply this operation to an empty queue.

**makeEmpty:** Removes all items from the queue. No return value.

**isEmpty:** Returns a value of type boolean, which is true if the queue contains no items and is false if there is at least one item in the queue.

A queue is a first in, first out (FIFO) data structure. The item that is next in line is the one that arrived first in the queue and that has been waiting in the queue the longest.

An abstract data type is not something that can be use directly in a program, for that, you need an actual implementation of the ADT, a concrete data type that says exactly how the data of the ADT are represented and the operations are coded.

In Java or C++, an ADT can be implemented as a class. Here for example, is an implementation of a stack of integer ADT in C++. Typically in C++, a class is defined in two files: a header (or 'h') files and implementation (or 'cc) file.

```
// The header file, Instack.h

Class Instack {

    Public:

        Instack; // A constructor for this class

        Void Push (Int x); // The operations of the stack ADT

        Int Pop ( );

        Void makeEmpty ( );

        Bool isEmpty ( );

        Private:

        Int data[100]; // contains the items on the stack

        Int top          // number of items on the stack

};

// The file Instack.cc

# include "Intstack.h"

Instack: : Instack ( ) {

        Top = 0; // a stack is empty when it is first created

}

Void Intstack : : push ( int x ) {

        If (top == 100 ) // ERROR, throw an exception

                Throw " Attempt to push onto a full stack"

                Data [top] = x;

}

Int Instack : : Pop ( ) {
```

```
        If  (top == 0) // ERROR, throw an exception
                Throw " Attempt to pop from an empty stack
                Top = - -
                Return data [top];
}
Void Intstack : : makeEmpty ( ) {
        Top = 0;
}
Boo Intstack : : isEmpty ( ) {
Return (top == 0 );
}
```

Note that the data for the stack is stored in the private part of the class. This prevents code that uses the class from making any use of the specific representation of the data. The data is simply inaccessible. This ensures that if the data representation is changed, code that uses the class will not have to be modified.

### Self-Assessment Question(s)

### Self-Assessment Answer

Please insert relevant answers

## 3.3 Data Abstraction

In computer science, abstraction is the process by which data and programs are defined with a representation similar in form to its meaning (semantics), while hiding away the implementation details. Abstraction tries to reduce and factor out details so that the programmer can focus on a

few concepts at a time. A system can have several abstraction layers whereby different meanings and amounts of detail are exposed to the programmer. For example, low-level abstraction layers expose details of the computer hardware where the program is run, while high-level layers deal with the business logic of the program.

In computer programming, abstraction can apply to control or to data: **Control abstraction** is the abstraction of actions while **data abstraction** is that of data structures.

- Control abstraction involves the use of subprograms and related concepts controlflows
- Data abstraction allows handling data bits in meaningful ways. For example, it is the basic motivation behind data type.

One can regard the notion of an object (from object-oriented programming) as an attempt to combine abstractions of data and code.

The recommendation that programmers use abstractions whenever suitable in order to avoid duplication (usually of code) is known as the abstraction principle. The requirement that a programming language provide suitable abstractions is also called the abstraction principle.

Data abstraction enforces a clear separation between the *abstract* properties of a datatype and the *concrete* details of its implementation. The abstract properties are those that are visible to client code that makes use of the data type—the *interface* to the data type—while the concrete implementation is kept entirely private, and indeed can change, for example to incorporate efficiency improvements over time. The idea is that such changes are not supposed to have any impact on client code, since they involve no difference in the abstract behaviour.

## 3.4 Distinguishing Abstract Data Types (Adts) and Procedural Data Abstraction (Pda)

Abstract data types are often called user-defined data types, because they allow programmers to define new types that resemble primitive data types. Just like a primitive type INTEGER with operations +, −, _, etc., an abstract data type has a type domain, whose representation is unknown to clients, and a set of operations defined on the domain. In this context the phrase "abstract type" can be taken to mean that there is a type that is "conceived apart from concrete realities".

Object-oriented programming involves the construction of objects which have a collection of methods, or procedures, that share access to private local state. Objects resemble machines or other things in the real world more than any well-known mathematical concept. The term "object" is not very descriptive of the use of collections of procedures to implement a data abstraction. Thus we adopt the term procedural

data abstraction as a more precise name for a technique that uses procedures as abstract data. In the remainder of this course, procedural data abstraction (PDA) will be used instead of "object-oriented programming". By extension, the term "object" is synonymous with procedural data value.

It is argued that abstract data types and procedural data abstraction are two distinct techniques for implementing abstract data. The basic difference is in the mechanism used to achieve the abstraction barrier between a client and the data. In abstract data types, the primary mechanism is type abstraction,while in procedural data abstraction it is procedural abstraction. This means, roughly, that in an ADT the data is abstract by virtue of an opaque type: one that can be used by a client to declare variables but whose representation cannot be inspected directly. In PDA, the data is abstract because it is accessed through a procedural interface – although all of the types involved may be known to the user. This characterization is not completely strict, in that the type of a procedural data value can be viewed as being partially abstract, because not all of the interface may be known; in addition, abstract data types rely upon procedural abstraction for the definition of their operations.

Despite the very different approaches taken by ADT and PDA, they can be understood as orthogonal ways to implement a specification of a data abstraction. A data abstraction can be characterized in a very general way by defining abstract constructors together with abstract observations of the constructed values. Using these notions, a data abstraction may be defined by listing the value of each observation on each constructor. The difference between PDA and ADT concerns how they organize and protect the implementation of a data abstraction. The choice of organization has a tremendous effect on the flexibility and extensibility of the implementation.

ADTs are organized around the observations. Each observation is implemented as an operation upon a concrete representation derived from the constructors. The constructors are also implemented as operations that create values in the representation type. The representation is shared among the operations, but hidden from clients of the ADT.

PDA is organized around the constructors of the data abstraction. The observations become the attributes, or methods, of the procedural data values. Thus a procedural data value is simply defined by the combination of all possible observations upon it.

Viewing ADTs and PDA as orthogonal ways to organize a data abstraction implementation provides a useful starting point for comparing the advantages and disadvantages of the two paradigms. The decomposition used, either by constructor or observer, determines how easy it is to add a new constructor or observer. Adding a new feature that clashes with the organization is difficult. In addition, the tight coupling and security in an ADTs makes it less extensible and flexible, but supports verification and optimization. The independence of PDA implementations

has the opposite effect. However, in several cases where PDA might be expected to have difficulty, the problems are lessened by the use of inheritance and subtyping.

Most existing programming languages that support PDA also make use of ADTs. C++ for example supports both ADTs and PDA in the same framework; the effect is more of interweaving than unification, since the trade-offs between the two styles are still operative. So a good programming style seems to call for implementing ADTs using objects. While objects and ADTs are fundamentally different, they are both forms of data abstraction. The general concept of data abstraction refers to any mechanism for hiding the implementation details of data.

### Self-Assessment Question(s)

### Self-Assessment Answer

Please insert relevant answers

## 4.0 Conclusion

In this module we discussed that In a modern computer, data consists of a binary bits, but meaningful data is organized into primitive data types such as integer, real, Boolean and into more complex data structures such as arrays and binary trees. These data types and data structures always come along with associated operations that can be done on the data. So, An Abstract Data Type or ADT, consists of:

- a specification of the possible values of the data type
- a specification of the operations that can be performed on those values in terms of the operations, input, output and effects.

## 5.0 Summary

In introducing this module, we discussed generally on data types as it applies to it abstraction under the following units:

1. Overview Of Primitive Data Types
2. Abstract Data Types

3. Stacks And Queues
4. Data Abstraction
5. Distinguishing Between Abstract Data Type (Adt) And Procedural Data Type (Pda)

## 6.0 Tutor-Marked Assignment

Discuss the differences and similarities between the primitive data types and abstract data type.

## 7.0 References/Further Reading

1. Principles of Programming Languages by Macclennan, 3rd edition, 1999, Oxford
2. Concept of Programming Languages, 8th edition by Robert W. Sebesta, Addison Wesley, ISBN -13:978-0321-49362-0, ISBN-10:0-321-49362-1