INTRODUCTION

Meaning of Programming

Programming means to convert problem solutions into instructions for the computer. It also refers to the process of developing and implementing various sets of instructions to enable a computer todo a certain task.

A **program** is a finite set of sequenced instructions or commands given to a computer in order to carry out a particular task

A programming language is a vocabulary and set of grammatical rules designed for instructing a computer to perform specific tasks.

History of Programming Languages

First-Generation Programming Languages – Machine Language

A first-generation of programming languages includes machine-level programming languages. These languages were introduced in the 1940s and had the following characteristics:

- Instructions were entered directly in binary format (1s and 0s) and therefore they were tedious and error prone. Programmers had to design their code by hand then transfer it to a computer using a punch card, punch tape or flicking switches.
- Instructions were executed directly by a computer's central processing unit (CPU) i.e. they were executed very fast.
- Memory management was done manually.
- Programs were very difficult to edit and debug.
- Used to code simple programs only.

Second-Generation Programming Languages (2GL)- Low Level Programming Languages/Assembly Languages

They were introduced to mitigate the error prone and excessively difficult nature of binary programming.

- Introduced in the 1950s
- Improved on first generation by providing human readable source code which must be compiled/assembled into machine code (binary instructions) before it can be executed by a CPU
- Specific to platform architecture i.e. 2GL source code is not portable across processors or processing environments.
- Designed to support logical structure and debugging.

By using codes resembling English, programming becomes much easier. The use of these ***mnemonic codes*** such as ***LDA*** for load and ***STA*** for store means the code is easier to read and write. To convert an assembly code program into object code to run on a computer requires an

Assembler and each line of assembly can be replaced by the equivalent one line of object (machine) code:

Assembly Code	Machine Code
LDA A	000100110100
ADD #5	001000000101
STA A	001100110100
JMP #3	010000000011

Such languages are sometimes still used for kernels and device drivers, i.e. the core of the operating system and for specific machine parts.

Third-Generation Languages (3GL) – High-Level Languages

Third generation languages are the primary languages used in general purpose programming today. They each vary quite widely in terms of their particular abstractions and syntax. However, they all share great enhancements in logical structure over assembly languages.

- Introduced in the 1950s
- Designed around ease of use for the programmer (Programmer friendly)
- Driven by desire for reduction in bugs, increases in code reuse
- Based on natural language
- Often designed with structured programming in mind
- The languages are architecture independent e.g. C, Java etc.

Examples: Most Modern General Purpose Languages such as C, C++, C#, Java, Basic, COBOL, Lisp and ML.

Fourth Generation Languages

Fourth-generation programming languages are high-level languages built around database systems. They are generally used in commercial environments.

- Improves on 3GL and their development methods with higher abstraction and statement power, to reduce errors and increase development speed by reducing programming effort. They result in a reduction in the cost of software development.
- A 4GL is designed with a specific purpose in mind. For example languages to query databases (SQL), languages to make reports (Oracle Reports) etc.
- 4GL are more oriented towards problem solving and systems engineering. Examples: Progress 4GL, PL/SQL, Oracle Reports, Revolution language, SAS, SPSS, SQ

Fifth Generation Languages

Improves on the previous generations by skipping algorithm writing and instead provide constraints/conditions.

While 4GL are designed to build specific programs, 5GL are designed to make the computer solve a given problem without the programmer. The programmer only needs to worry about what problems needed to be solved and only inputs a set of logical constraints, with no specified algorithm, and the Artificial Intelligence (AI)-based compiler builds the program based on these constraints. Examples: Prolog, OPS5, Mercury

Types of Computer Programming Languages

There are two main types of computer programming languages; these are: Low-level language and High-level language.

i. **Low-Level Languages:** Low-level languages also known as machine language, are machine dependent and makes fast and efficient use of the computer. ***Low-level languages*** such as ***machine language*** and **assembly language** are closer to the hardware than are the high-level programming languages, which are closer to human languages. Low-level languages are converted to machine code without using a compiler or interpreter, and the resulting code runs directly on the processor. A program written in a low-level language runs ***very quickly***, and with a ***very small memory footprint***; an equivalent program in a high-level language will be more heavyweight. Low-level languages are ***simple***, but are considered ***difficult to use,*** due to the numerous technical details which must be remembered.

ii. ***High-level languages***: A high-level language is a problem-orientated programming language and are machine independent. ***High-level languages*** are closer to human languages and further from machine languages. The main advantage of high-level languages over low-level languages is that they are ***easier to read, write, and maintain***. Ultimately, programs written in a high-level language must be translated into machine language by a compiler or interpreter. The first high-level programming languages were designed in the 1950s. Now there are dozens of different languages, including Ada, Algol, BASIC, COBOL, C, C++, FORTRAN, LISP, Pascal, and Prolog.

Advantages of High-Level Language

- No knowledge of the computer in which the program will be run is required
- The programs are portable
- Very easy to learn and write

Disadvantages of High-Level Language

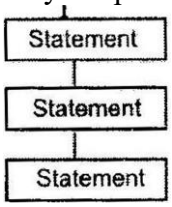
- It takes additional translation times to translate the source to machine code.

- High level programs are comparatively slower than low level programs.
- Compared to low level programs, they are generally less memory efficient.
- Cannot communicate directly with the hardware

1. STRUCTURED PROGRAMMING ELEMENTS

- Structured Programming is a **programming paradigm** introduced in the late 1960s (by **Edsger Dijkstra** and others) to overcome the difficulties of unstructured, "spaghetti" code caused by excessive use of ***goto*** statements.
- Structured programming (sometimes known as modular programming), is a programming paradigm that facilitates the creation of programs with readable code and reusable components. All modern programming languages support structured programming, but the mechanisms of support -- like the syntax of the programming languages -- vary.
- Structured programming encourages dividing an application program into a hierarchy of modules or autonomous elements, which, in turn, may contain other such elements. Within each element, code may be further structured using blocks of related logic designed to improve readability and maintainability. These may include `case`, which tests a variable against a set of values, and ***`repeat`***, ***`while`*** and `for`, which construct loops that continue until a condition is met. In all structured programming languages, an unconditional transfer of control, or ***`goto`*** statement, is deprecated and sometimes not even available.
- It aimed at improving ***clarity, quality***, and **development time** by using well-structured control flow constructs. Structured programming restricts itself to only **three fundamental control structures**:

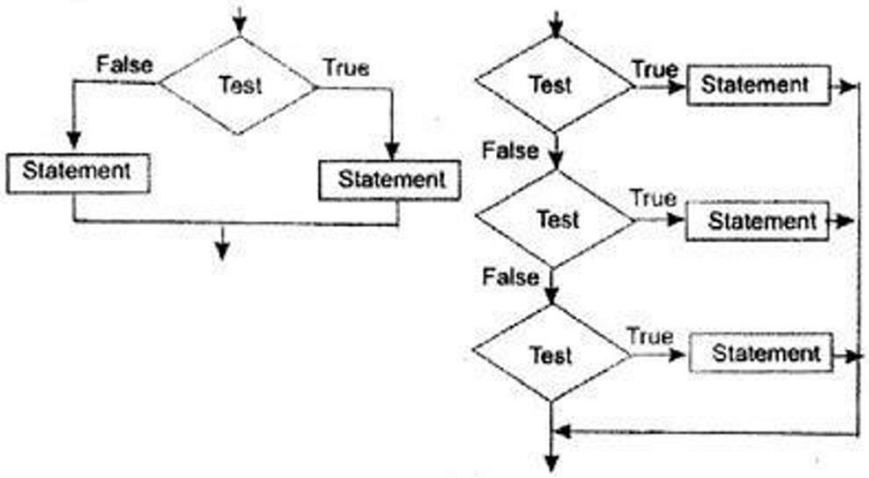
1. **Sequence Structure**–This is the execution of statements one after the other. This is a very simple module of Structured Programming.



C-like Structured Program:

```
int a = 10;
int b = 20;
int sum = a + b; // executed in sequence
```

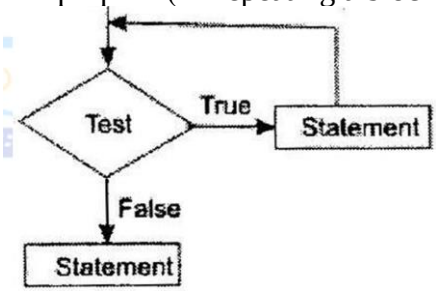
2. **Selection or Conditional Structure (Decision)** – The Program has many conditions from which correct condition is selected to solve problems (Choosing between alternatives.). These are (a***) if-else*** (b) ***else-if***, and (c) ***switch-case***



C-like Structured Program**:**

```
if (score >= 50) {
    printf("Pass");
} else {
   printf("Fail");
}
```

3. **Iteration (Repetition/Looping) Structure**– The process of repetition or iteration repeats statements blocks several times when condition is matched, if condition is not matched, looping process is terminated. In C, (a) goto, (b) for (), (c) do, (d) do – while are used for this purpose (i.e repeating a block until a condition is met).



C-like Structured Program**:**

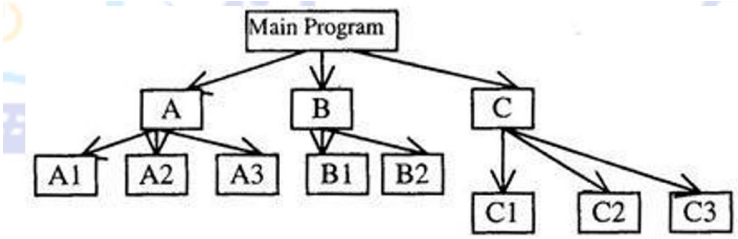
```
for ( init; condition; increment )
    { statement(s);
}
```

Example:

```
for (int i = 0; i < 5; i++) {
    printf("%d\n", i);  // prints numbers 0–4
}
```

Features of Structured Programming

1. **Top-Down Program Design**: Refers to breaking down a large problem into smaller ones and solving each small problem independently.



Example: Designing a payroll system → break into modules like *input employee details*, *calculate salary*, *generate payslip*.

2. **Use of Subroutines and Functions**: Code is organized into functions/procedures for reusability. Example in C:

```
int add(int x, int y)
  {
    return x + y;
  }
```

3. **Block Structure**: Programs are divided into logical blocks using { } in C/Java or BEGIN ... END in Pascal and it improves readability.
4. **Single Entry, Single Exit Principle**: Each control structure has **one entry point** and **one exit point**, avoiding untraceable jumps (goto).
5. **Clarity and Readability**: Code is designed to be easily read and understood by humans, not just machines.
6. **Structured Data Types**: Use of arrays, records/structs, and later abstract data types to organize data logically.

Advantages of Structured Programming

- ***Ease of Use***: This approach allows simplicity, as lines of program code can be accessed in the form of modules, rather than focusing on the entire thousands and millions of lines code. This allows ease in debugging the code and prone to less error.
- ***Reusability:*** It allows the user to reuse the functionality with a different interface without typing the whole program again. Ease of
- ***Maintenance:*** It helps in less collision at the time of working on modules, helping a team to work with proper collaboration while working on a large application.

Disadvantages / Limitations

- ***More memory space is required***- When the numbers of modules are out of certain range, performance of program is not satisfactory.
- ***Not ideal for modeling*** *real-world* **entities** (e.g., students, cars, employees) → led to OOP.
- ***Excessive reliance on functions*** may lead to difficulty in managing shared data.
- ***Complex*** relationships (like inheritance or polymorphism) are not easily represented.

2. STRUCTURED DESIGN PRINCIPLES

Structured design principles are **methodologies and practices** that guide programmers in building software that is **clear, reliable, efficient, and easy to maintain**. They form the bridge between **problem analysis** and **program implementation**.

The major principles are: **Abstraction, Modularity, Stepwise Refinement, and Structured Design Techniques.**

a) Abstraction

- Abstraction is the process of **hiding unnecessary details** and focusing only on the **essential features** of a concept or problem.
- It allows the programmer to think at a higher level without worrying about implementation details.

Example: The computer operators know only to operate computer, but they are unaware to internal organization of computer.

-

Types of Abstraction

i. **Data Abstraction**
   - Concerned with the **representation of data**.
   - The actual details of how data is stored are hidden.
   - Example: In C, a struct or in Java, a class can represent a Student. Users of the class don't need to know how the data is stored internally.

```
class Student {
   private String name;  // hidden data
   private int age;

   // abstraction through methods
   public void setName(String n) { name = n; }
   public String getName() { return name; }
}
```

   Here, the user interacts with ***setName()*** and ***getName()*** instead of directly accessing the name variable.

ii. **Control Abstraction**
   - Hides the **details of control flow** and allows the programmer to use a simplified construct.
   - Example: Instead of writing low-level machine instructions for looping, we use a for loop in C/Java.

```
for (int i = 0; i < 10; i++) {
   printf("%d ", i);
}
```

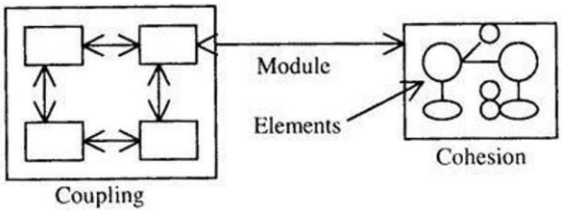
Advantages

- Simplifies complex systems.
- Encourages reusability.
- Increases readability and maintainability.

b) Modularity: Modularity means dividing a program into **independent modules** (components) where each module performs a specific function.

Characteristics of Good Modules

The evaluation of modeling is called ***coupling*** and ***cohesion***. The module coupling denotes number of interconnections between modules and module cohesion shows relationship among data or elements within a module.



- **High Cohesion** – Each module performs **one well-defined task**.
- **Low Coupling** – Modules are as **independent** as possible; minimal interdependence.

Benefits

- Easier debugging: errors can be traced to specific modules.
- Parallel development: different programmers can work on separate modules simultaneously.
- Reusability: modules can be reused in other programs.
- Better organization: programs become more manageable.

Example (C-style payroll program split into modules)

float computeGross(float basic, float allowance);
float computeNet(float gross, float taxRate);
void printPayslip(float gross, float net);
*Each function represents a **module** with a specific task.*

c) Stepwise Refinement: A top-down approach **to problem-solving where a complex problem is** repeatedly broken down **into smaller, simpler subproblems until each can be directly implemented.** It was proposed by **Niklaus Wirth (creator of Pascal).**

Process

1. Define the problem at a high level.
2. Break it into smaller tasks.
3. Refine each task into sub-tasks.
4. Stop when each sub-task is directly translatable into code.

Example: Find the average of N numbers

1. **High level**: Compute average of numbers.
2. **Stepwise refinement**:
   - Read N numbers
   - Find sum of numbers
   - Divide sum by N
   - Display result
3. **Pseudo-code**:

   Mathematica

   Start
   Input N
   For i = 1 to N
   Input number
   Add number to sum
   Average = sum / N
   Print Average
   End

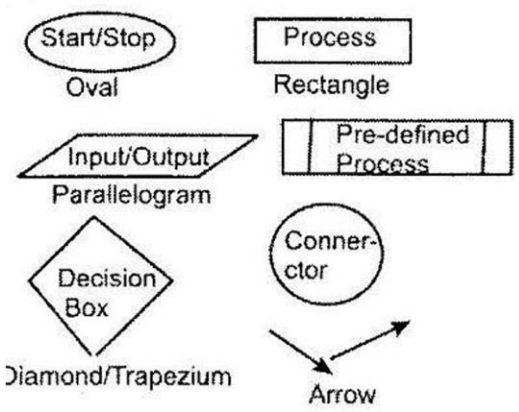
This method ensures **clarity and logical flow**.

d) Structured Design Techniques

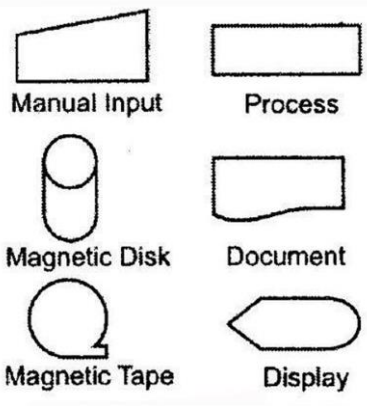
Structured design uses **graphical and textual tools** to plan and visualize programs before coding.

Key Techniques

1. **Flowcharts**
   - Graphical/ pictorial representation of program or an algorithm.
   - It is a tool and technique to find out solution of programming problems through some special symbols. It is a way to represent program using geometrical patterns. Prior to 1964, all manufactures use different types of symbols, there was no uniformity and standards of flowcharting. The Standard symbols were developed by American Standard National Institute (ANSI).
   - There are two types of Flow Chart: ***(a) Program Flow Chart and (b) System Flow Chart***

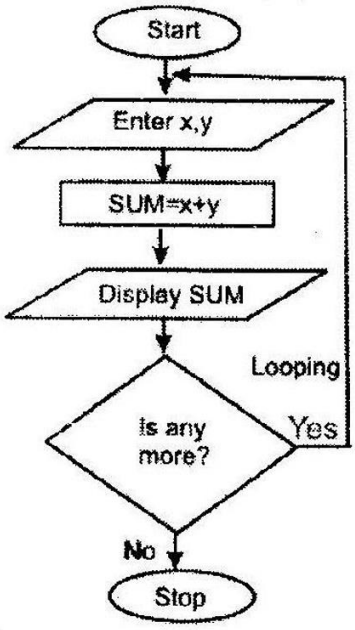


Program:Flowchart



System Flowchart Symbols

Example: A flow chart to input two numbers and display sum.



Exercise

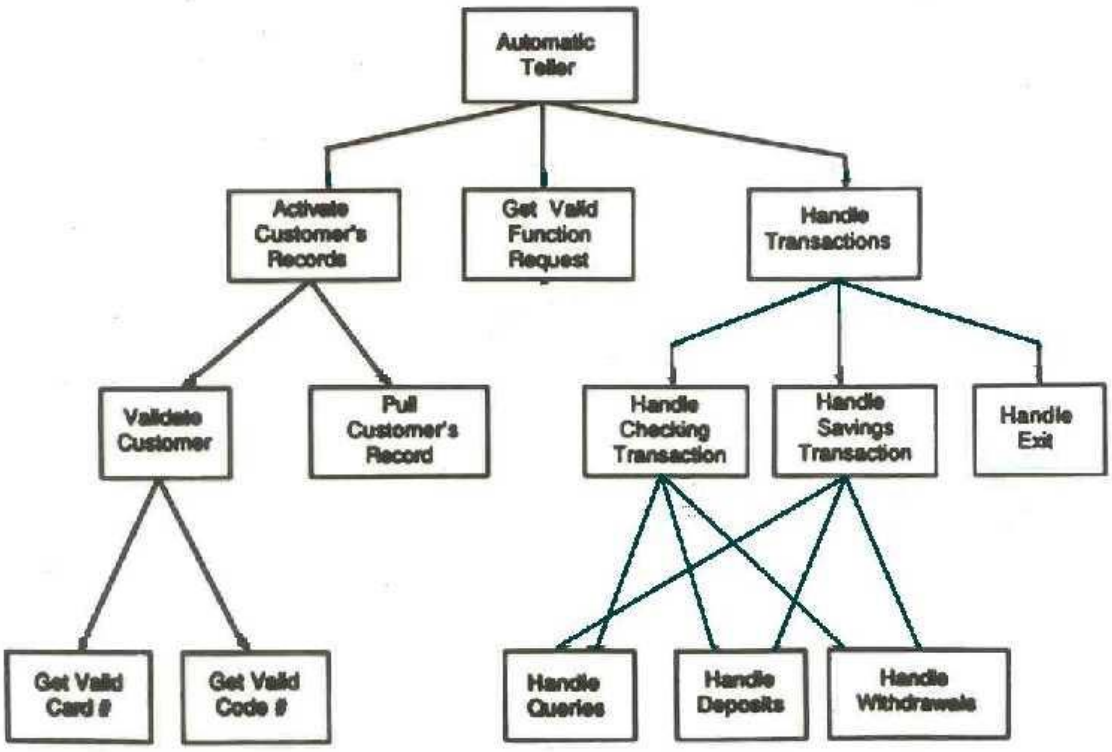
Draw a flowchart to enter principal, rate and time and display simple interest.
Draw a flowchart to find the sum of the given series.

2. **Pseudo-code**
   - Pseudocode: is a simplified version of code written in plain language, helping to outline a program's logic before implementation.
   - Focuses on logic rather than syntax.

Example

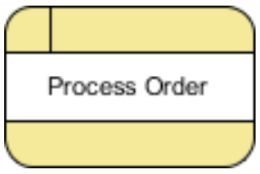
```
for i←1 to 100 do
  if i is divisible by 3 then
    output "Yes"
  else
output i
```

3. **Structure Charts**
   - Visual tool showing **hierarchical relationships** among modules. It is NOT a flowchart. It has nothing to do with the logical sequence of tasks to be perform. It does not show the order in which tasks are performed. Example in ATM Machine

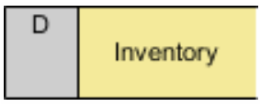


.

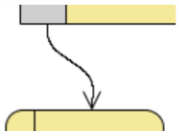
4. **Data Flow Diagrams (DFD)**
   - Data flow diagram is graphical representation of the flow of data in any system. Thus, DFD describes the processes that are involved in a system to transfer data from the input to the file storage and reports generation. DFD focus on **flow of data** through the system.
   - **DFD Components and Symbols**
     - ***Processes***− A process represents an activity or transformation within the system, converting incoming data into outgoing data. Represented by rectangular with *rounded corners, or oval*. It describes how each process transforms inputs into outputs.



     - ***Data Stores (Warehouses)***− These are repositories that serve as both sources and destinations for data within the system. they are represented by *open-ended rectangles*, symbolizing where data is stored within the system.

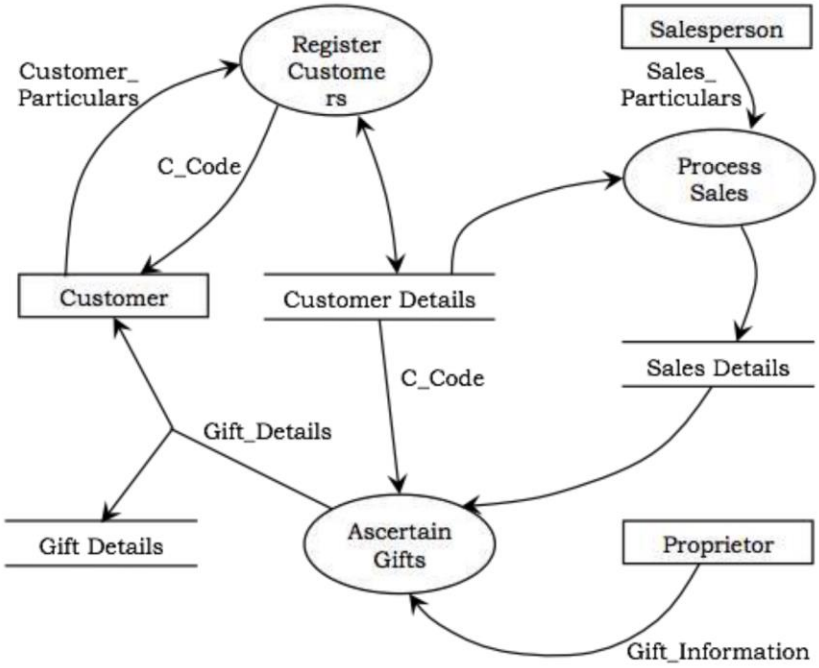


- ***Data Flow***– They illustrate how data travels from external entities to processes, between processes, from processes to data stores, and vice versa. A *directed arc* or *arrow* symbol is used to indicate data movement between different parts of the systems, labelled with the type of data being transferred.



- ***External Entities*** (Sources/Sinks/actors)– External entities are individuals, groups, departments, or other systems that interact with the modeled system but exist outside its defined boundaries. Represented by squares, or rectangles drawn outside of the system boundary, containing the entity name and an identifier..

Example: DFD with simple use case

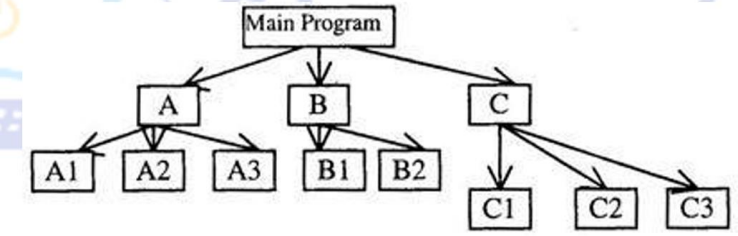


DFD of Wholesale Software

Approach

Structured Design Techniques focus on *top-down design* and *bottom-up implementation*

i) **Top-Down Design**: Start from the problem → break into smaller tasks.



Example: The main program is divided into sub-program A, B, and C. The A is divided into subprogram A1, A2 and A3.The B is into B1, and B2. Just like these subprograms, C is also divided into three subprogram C1, C2 and C3. The solution of Main program is obtained from sub program A, B and C.

ii) **Bottom-Up Implementation**: Write and test lowest-level modules first, then integrate into higher-level modules.

3. OBJECT-ORIENTED PROGRAMMING (OOP)

This is a programming paradigm that represents concepts as "*objects*" that have data fields (attributes that describe the object) and associated procedures known as *methods*. Objects, which are usually *instances of classes*, are used to interact with one another to design applications and computer programs.

Why OOP?

- **Problem with Structured Programming:**
  - Structured programming (like C) organizes code into functions and procedures.
  - It becomes difficult to model complex, real-world entities because data and functions are separated.
  - Example: A "Car" has attributes (color, speed) and actions (drive, brake). In structured programming, attributes are separate from functions, making management harder.
- **OOP Advantage:**
  - Introduces **objects** that **encapsulate both data and behavior**.
  - Helps model real-world entities more naturally.
  - Improves code **reusability**, **scalability**, and **maintainability**.

a) Classes & Objects

- **Class:** A class is a collection of similar objects. It is a **blueprint** or **template** for creating objects.
  - Defines the properties (fields/variables) and behaviors (methods/functions).

Example (Java):

```
class Car
{
  String color;
   int speed;
```

```
    void drive()
      {
         System.out.println("Car is driving");
      }
}
```

- **Object:** Objects are members of class. It is a **real instance** of a class containing **state** and **behavior**.

  Example:

  ```
  Car myCar = new Car(); // Object creation
  myCar.color = "Red";   // Assigning state
  myCar.drive();         // Calling behavior
  ```

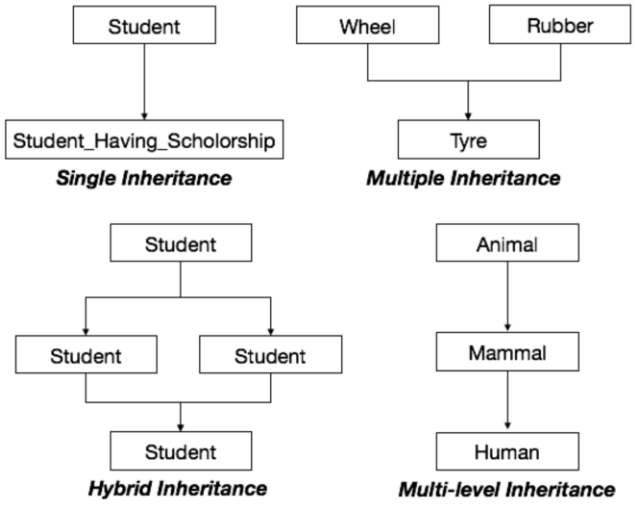
- **Flowchart (Class → Object):**

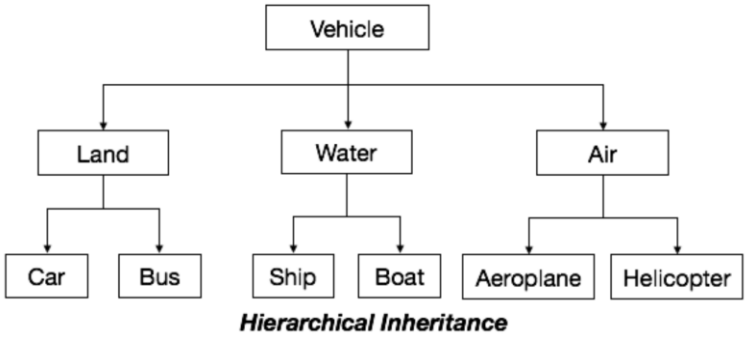
  Class (Car) ---> *Object (myCar, yourCar, etc.)*
  ↑ blueprint ↑ instance with actual values

b) Inheritance: Mechanism of **deriving new classes (child/subclass)** from **existing classes (parent/superclass)**. This promotes **code reuse** and **hierarchical classification**.
The main types of inheritance are:
Types of Inheritance

- ***Single Inheritance***: A subclass derives from a single super-class.
- ***Multiple Inheritance***: A subclass derives from more than one super-classes.
- ***Multilevel Inheritance***: A subclass derives from a super-class which in turn is derived from another class and so on.
- ***Hierarchical Inheritance***: A class has a number of subclasses each of which may have subsequent subclasses, continuing for a number of levels, so as to form a tree structure.
- ***Hybrid Inheritance***: A combination of multiple and multilevel inheritance so as to form a lattice structure

The following figure depicts the examples of different types of inheritance.





Hierarchical Inheritance

Example (Java):

```
class Vehicle {
   void start() {
      System.out.println("Vehicle starts");
   }
}

class Car extends Vehicle {
   void drive() {
      System.out.println("Car is driving");
   }
}
```

c) Polymorphism: The ability of the **same method name** to perform **different actions** depending on context. Polymorphism allows ***objects*** with different internal structures to have a common external interface

Example

Let us consider two classes, Circle and Square, each with a method *findArea()*. Though the name and purpose of the methods in the classes are the same, the internal implementation, i.e., the procedure of calculating area is different for each class. When an object of class Circle invokes its *findArea()* method, the operation finds the area of the circle without any conflict with the *findArea()* method of the Square class.

Types in Java:

Compile-time Polymorphism (Method Overloading):

- Same method name but different parameter lists.
- Resolved at compile time.

Runtime Polymorphism (Method Overriding):

- Subclass provides a specific implementation of a method already defined in the parent class.
- Resolved at runtime (dynamic binding).

d) Encapsulation: The process of **binding data (fields)** and **methods (functions)** into a single unit (class). It provides **data hiding** and controlled access using **access modifiers**.

Access Modifiers in Java:

- **private** → accessible only within the same class.
- **protected** → accessible within package and subclasses.
- **public** → accessible from anywhere.

4. Tools for Development

- **Compiler**: A compiler translates source code written in a high-level language into **machine code (binary instructions)** or **bytecode** (for Java).
- **Interpreter**: Executes source code **line by line** instead of compiling the whole program first. (Java uses JVM interpreter + JIT compiler).
- **Debugger**: A tool to test and find **logical/runtime errors** in code. Example: Debugger in Eclipse, IntelliJ IDEA.
- **IDE (Integrated Development Environment)**: A software suite that provides tools for development in one place. e.g., Eclipse, NetBeans, IntelliJ IDEA.

5. Java Programming Basics

Syntax

- Java is **case-sensitive** → **Main** and **main** are different.
  Strictly **object-oriented** → everything inside a class.

Data Types

- **Primitive types**: int (4 bytes), float (4 bytes), double (8 bytes), char (2 bytes), boolean (true/false), byte, short, long.etc.
- **Reference types**: Objects created from classes: *string, arrays, user-defined classes*.

Operators

- Arithmetic (+, -, *, /, %)
- Relational (==, !=, <, >)
- Logical (&&, ||, !)
- Assignment (=, +=, -=)

6. Control Flow Constructs

- **Sequence** – statements executed in order.

```
int x = 10;
int y = 20;
int z = x + y;   // executed in sequence
```

- **Selection** – if, if-else, switch.

```
if (x > y)
{
   System.out.println("x is greater");
}
else
{
   System.out.println("y is greater");
```

```
}
```

- **Iteration** – for, while, do-while.

```
for (int i = 0; i < 5; i++)
{
   System.out.println(i);
}
```

- **Branching** – break (exit loop), continue (skip iteration), return (exit method)..

7. Advanced Java Programming

a) Arrays

- Collection of homogeneous data elements stored in contiguous memory.
- One-dimensional, multi-dimensional arrays.

Examples:

1D Array

Java:

```
int arr[] = {1, 2, 3};
```

2D Array

Java:

```
int matrix[][] = new int[3][3];
```

b) Methods

- Function-like blocks inside classes.
- Support parameter passing (by value).

Java:

```
int add(int a, int b)
{
   return a + b;
}
```

c) Exceptions

- Mechanism to handle runtime errors using *try, catch, finally, throw, throws*.

Java:

```
try
{
   int x = 10 / 0;   // error
}
catch (ArithmeticException)
 {
   System.out.println("Cannot divide by zero!");
 }
finally
  {
     System.out.println("Done.");
```

```
    }
```

d) Applets

- Java programs embedded in web pages (deprecated in modern Java).
  Example (historical, not used today):

```
import java.applet.Applet;
import java.awt.Graphics;

public class HelloApplet extends Applet {
   public void paint(Graphics g) {
      g.drawString("Hello Applet", 50, 50);
   }
}
```

e) Abstract Classes & Interfaces

- ***Abstract class***: cannot be instantiated, may contain abstract methods.

```
abstract class Shape
{
   abstract void draw();
}
```

- ***Interface:*** defines a contract (methods without implementation).

```
interface Drawable
{
   void draw();
}
```

f) Persistence

- Mechanism to save object states (e.g., file handling *(FileWriter, FileReader),* serialization (*Saving an object to file*), database: *using JDBC (Java Database Connectivity*)).

8. Java Window Toolkit

- AWT (Abstract Window Toolkit) and **Swing** used for GUI programming.
- Components: Buttons, Labels, TextFields, Frames.
- Event-driven programming model.

9. OLE (Object Linking and Embedding)

- Microsoft technology for embedding and linking documents/objects (more common in Windows applications than Java).
- Java equivalent: *JavaBeans* and *component reuse*.