UNIT I

INTRODUCTION TO PROBLEM SOLVING

INTRODUCTION TO COMPUTER PROBLEM SOLVING

- ❖ Computer problem solving is a fundamental skill that individuals in the field of computing and technology need to possess.
- ❖ It involves the ability to analyze a problem, devise a solution & implement that solution using computational thinking and programming skills.
- ❖ Some of the key concepts related to computer problem solving are as:

1. Understanding the Problem

- ❖ ***Problem Definition*****:** Clearly articulate the problem at hand. What is the desired outcome, and what are the constraints or requirements?
- ❖ ***Data Analysis*****:** Understand the data involved and how it relates to the problem. Identify patterns and trends.

INTRODUCTION TO COMPUTER PROBLEM SOLVING

2. Decomposition

- ❖ Break down the problem into smaller, manageable sub-problems.
- ❖ This makes it easier to focus on specific aspects & solve them individually.

3. Algorithmic Thinking

- ❖ Develop step-by-step procedures or algorithms to solve each sub-problem.
- ❖ Algorithms are sets of instructions that, when followed, lead to a solution.

4. Abstraction

- ❖ Focus on the essential details while ignoring unnecessary complexities.
- ❖ Abstraction helps in simplifying the problem, making it easier to understand and solve.

INTRODUCTION TO COMPUTER PROBLEM SOLVING

5. Pattern Recognition

❖ Identify similarities between the current problem and the problems that you have solved before.

❖ Recognizing patterns can lead to the application of known solutions.

6. Coding/Programming

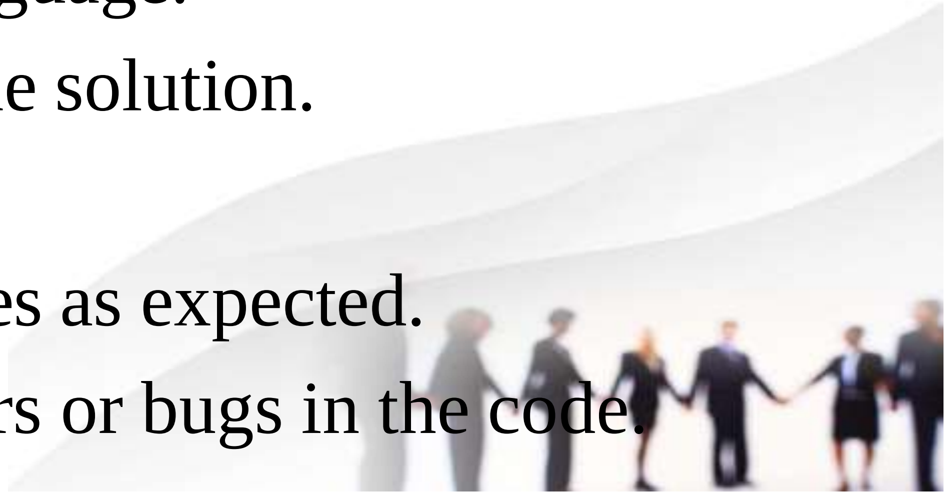
❖ Translate the algorithm into a programming language.

❖ This step involves writing code to implement the solution.

7. Testing and Debugging

❖ Thoroughly test the program to ensure it behaves as expected.

❖ Debugging involves identifying and fixing errors or bugs in the code.



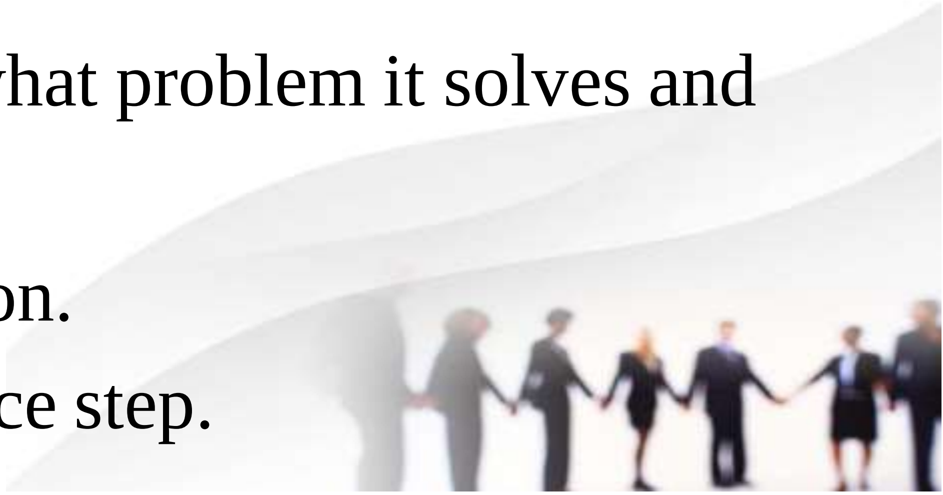
INTRODUCTION TO COMPUTER PROBLEM SOLVING

8. Optimization

- ❖ Code optimization has limitless benefits in the field of programming.
- ❖ It improves the efficiency and performance of the solution.
- ❖ This may involve refining the algorithm, optimizing code, or using more efficient data structures.

9. Documentation

- ❖ Document the code, explaining how it works, what problem it solves and how to use it.
- ❖ This aids in future maintenance and collaboration.
- ❖ Software documentation is of utmost significance step.



INTRODUCTION TO COMPUTER PROBLEM SOLVING

10. Iterative Process

- ❖ Problem solving is often an iterative process which repeats the same steps of processes again and again.
- ❖ After implementation, evaluate the solution, gather feedback and make improvements as necessary.

11. Collaboration

- ❖ Computer problem solving is not always a solitary activity.
- ❖ Collaboration with peers, feedback from others, and learning from shared experiences can enhance problem-solving skills.
- ❖ Thus, collaboration is a must in all circumstances.

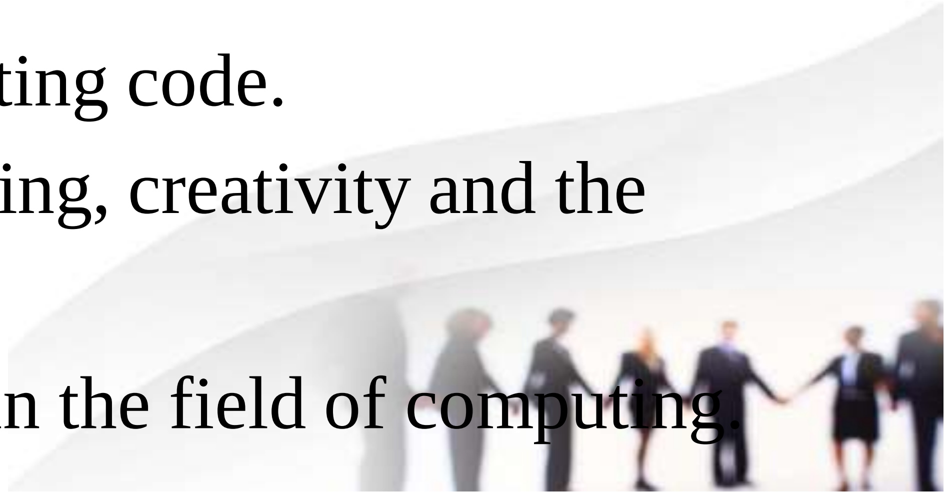
INTRODUCTION TO COMPUTER PROBLEM SOLVING

12. Adaptability

- ❖ The computing landscape is dynamic, with new technologies and challenges emerging regularly.
- ❖ Being adaptable and open to learning new approaches is crucial for effective problem solving.

❑ **Conclusion**

- ❖ Computer problem solving is not just about writing code.
- ❖ It's a holistic process that involves critical thinking, creativity and the ability to communicate solutions effectively.
- ❖ Developing these skills is essential for success in the field of computing.



THE PROBLEM SOLVING ASPECT

1. Definition of the Problem

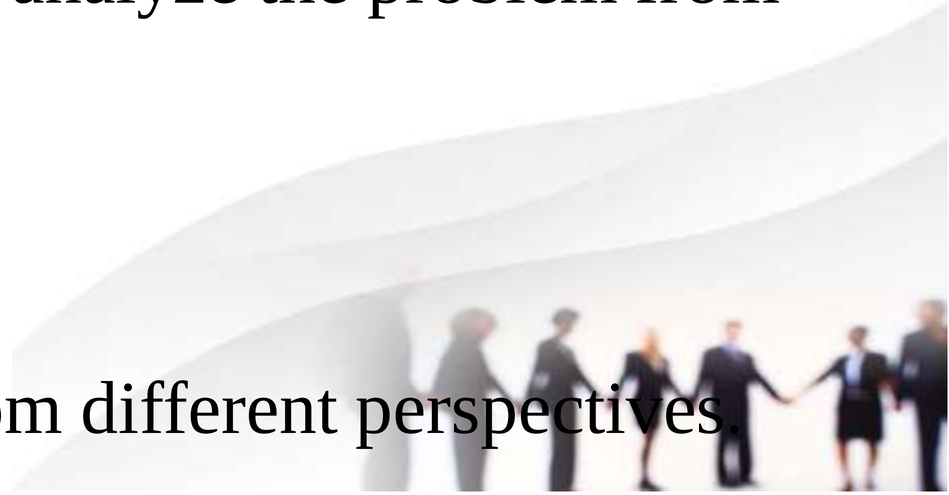
- ❖ Definition of a problem is the first and foremost step.
- ❖ Clearly define the problem to be solved.
- ❖ Understand the requirements, constraints, and expected outcomes.

2. Critical Thinking

- ❖ Apply logical reasoning and critical thinking to analyze the problem from various angles.

3. Creativity

- ❖ Seek innovative and efficient solutions.
- ❖ Think outside the box to approach problems from different perspectives.



THE PROBLEM SOLVING ASPECT

4. Analyzing root causes

- ❖ Investigate the underlying cause of the problem.
- ❖ Use tools like 5 Whys or Fishbone Diagram to analyze the root cause.
- ❖ Proper analysis of the root cause of a problem can save time, money, resources and efforts.

5. Brainstorming solutions

- ❖ Generate a variety of possible solutions, and encourage creative thinking.
- ❖ Choosing and implementing a solution
- ❖ Evaluate all the possible solutions, and select the most feasible solution.
- ❖ Then, execute it with a structured plan.

THE PROBLEM SOLVING ASPECT

6. Reflective practice

- Thinking about or reflecting on what you do is an important part of learning from experience.
- Regular reflection can help you identify areas to develop and accelerate your development.
- The correct problem-solving technique for a given situation depends on the individual, their experience, and their resourcefulness.

TOP DOWN DESIGN

❖ This approach usually starts from the top of the program and moves down in a step-by-step manner.

1. **Hierarchical Structure**

❖ The structure of these types of problems is typically hierarchical in nature.

❖ It involves breaking down the problem into smaller, more manageable modules or functions.

2. **Divide and Conquer**

❖ This is a very popular strategy in computing domain.

❖ The strategy is capable of tackling each module independently by simplifying the overall process of problem-solving.

TOP DOWN DESIGN

3. Abstraction

- ❖ Focus on high-level structure before diving into the details.
- ❖ The abstraction basically hides the unnecessary details of the system from all those users who don’t need them.
- ❖ This aids in understanding and designing complex systems.
- ❖ Abstraction is found everywhere in all kinds of applications.

IMPLEMENTATION OF ALGORITHMS

- An algorithm is a set of instructions that are followed in order to solve a problem or complete a task.
- A finite sequence of instructions that are used to solve a specific problem or perform a computation.
- Algorithms are mostly used to perform calculations, data processing and other operations.
- Implementing an algorithm takes following points into considerations:
    1. Algorithm Design
    2. Coding
    3. Programming Paradigms

IMPLEMENTATION OF ALGORITHMS

1. Algorithm Design

- ❖ Develop a step-by-step plan to solve the problem.
- ❖ Algorithms provide a systematic approach to reaching a solution.
- ❖ Algorithm can be written to solve any kind of problems - computational as well as non-computational.
- ❖ Designing an algorithm is an integral part of computing.

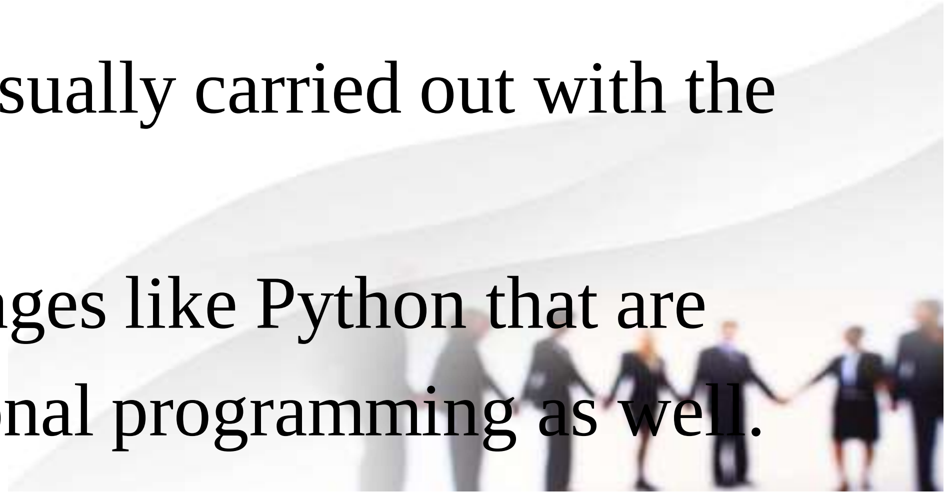
2. Coding

- ❖ Translate the algorithm into a specific programming language.
- ❖ Write code to implement the steps outlined in the algorithm.

IMPLEMENTATION OF ALGORITHMS

3. Programming Paradigms

- Choose appropriate programming paradigms based on the nature of the problem we want to solve.
- Prime examples of paradigms can be procedural, object-oriented, etc.
- The procedural programming is usually carried out with the help of languages like C.
- Similarly, the object-oriented programming is usually carried out with the help of languages like C++, Java.
- However, there are certain programming languages like Python that are object oriented in nature but support the functional programming as well.



PROGRAM VERIFICATION

- ❖ Maintaining correctness of program is very crucial for effective software development process.
- ❖ Program verification is carried out with the help of following techniques:

1. **Testing**

- ❖ Testing is the most preferred process in case of verification.
- ❖ One must try to execute programs with various inputs to ensure it produces a correct output.
- ❖ Testing can be classified into unit testing, integration testing, performance testing, white-box testing, black-box testing, etc.

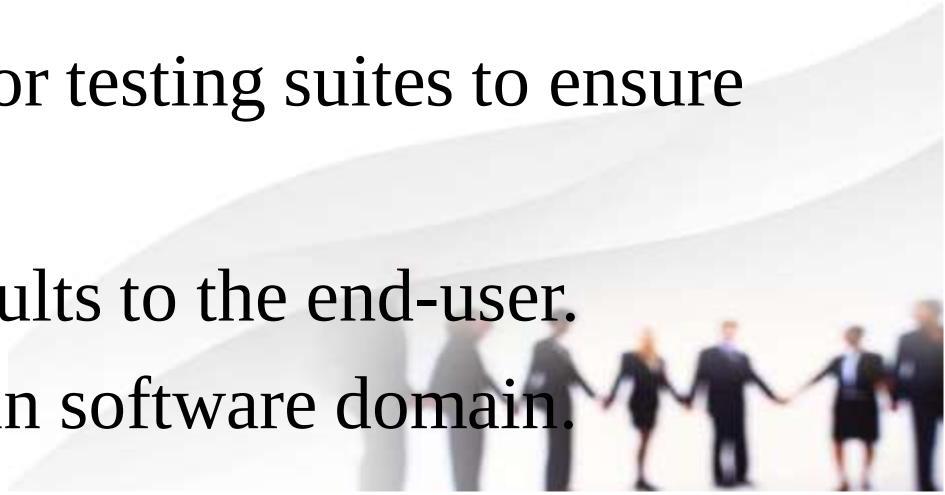
PROGRAM VERIFICATION

2. Debugging

- ❖ Identify and fix errors (bugs) in the code that may arise during testing.
- ❖ A bug is a common phenomenon in programming as it is a manual practice.
- ❖ However, it is expected to clear all the bugs before the software is handed over to the client.

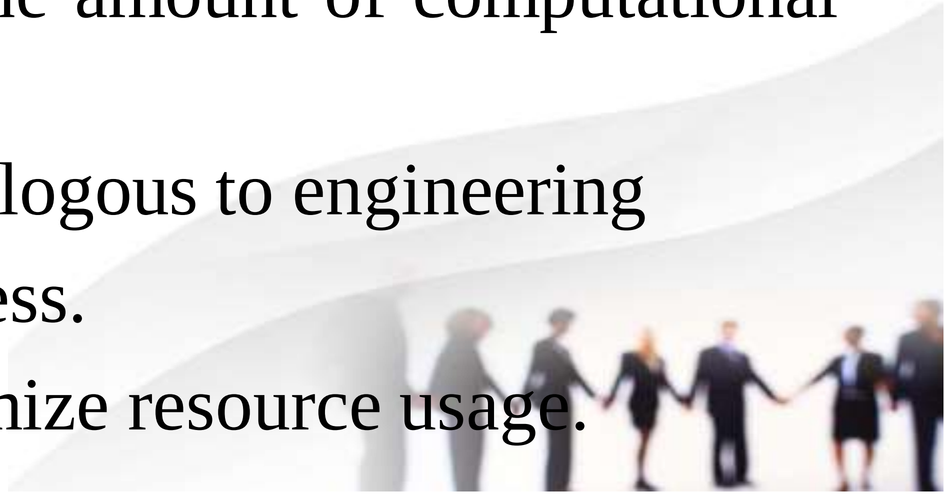
3. Verification Methods

- ❖ Employ techniques such as formal verification or testing suites to ensure the correctness of the program.
- ❖ A correct program can only give the desired results to the end-user.
- ❖ Hence, verification methods must be deployed in software domain.



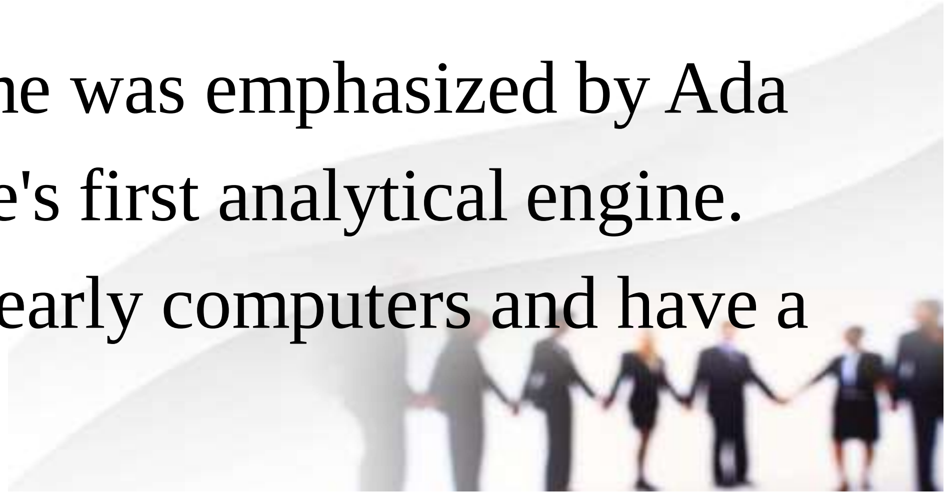
THE EFFICIENCY OF ALGORITHMS

- ❖ An algorithm's efficiency is how well it solves a problem in terms of time and memory usage.
- ❖ An algorithm is considered efficient if it uses resources at or below an acceptable level, such as running in a reasonable amount of time or space on a computer.
- ❖ Its a property of an algorithm that relates to the amount of computational resources used by the algorithm.
- ❖ Algorithmic efficiency can be thought of as analogous to engineering productivity for a repeating or continuous process.
- ❖ For maximum efficiency it is desirable to minimize resource usage.



THE EFFICIENCY OF ALGORITHMS

- The different resources such as time & space complexity cannot be compared directly, so which of two algorithms is considered to be more efficient often depends on which measure of efficiency is considered to be the most important one.
- For example, bubble sort and quick sort are both algorithms to sort a list of items from smallest to largest.
- The importance of efficiency with respect to time was emphasized by Ada Lovelace in 1843 as applied to Charles Babbage's first analytical engine.
- Modern computers are significantly faster than early computers and have a much larger amount of memory available.



THE EFFICIENCY OF ALGORITHMS

- ❖ Here are some ways to measure an algorithm's efficiency:

1. Computational complexity theory

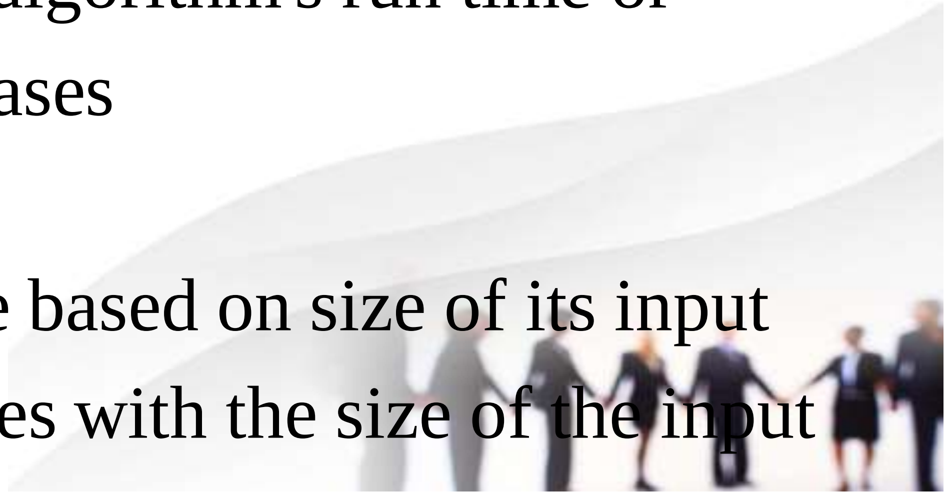
- ❖ Uses a mathematical function to estimate an algorithm's efficiency based on its input size

2. Big O notation

- ❖ A mathematical notation that describes how an algorithm's run time or space requirements grow as the input size increases

3. Time Complexity

- ❖ Measures how long algorithm takes to complete based on size of its input
- ❖ Analyze how running time of algorithm increases with the size of the input



THE EFFICIENCY OF ALGORITHMS

4. Space complexity

❖ Evaluate the amount of memory an algorithm requires for execution.

❖ Tracks how much memory the algorithm needs as the input size grows.

❖ Here are some tips for writing efficient algorithms:

5. Optimization:

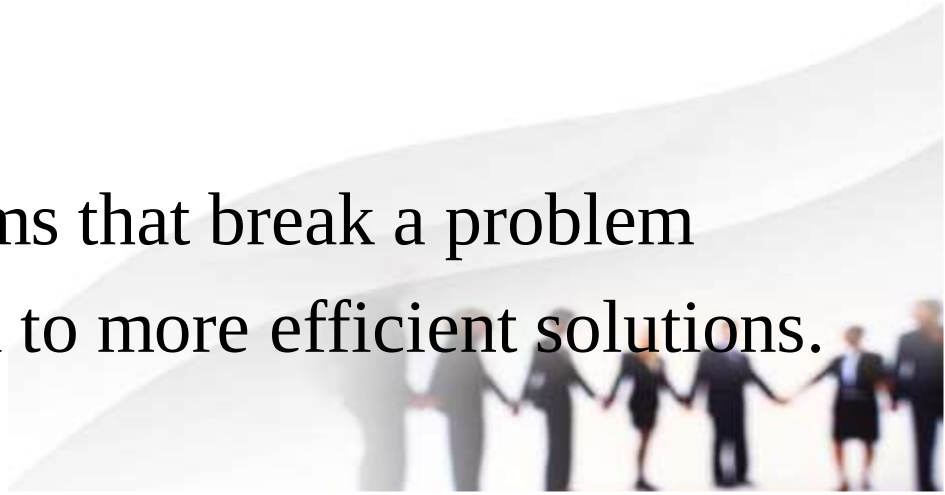
❖ Refine algorithms to improve its efficiency.

❖ Consider the factors like speed and resource utilization.

❖ That means, ideally an algorithm should solve the problems at the highest speed with lowest possible resources being utilized.

THE EFFICIENCY OF ALGORITHMS

- ❖ Here are some tips for writing efficient algorithms:
- ➢ Understand the problem
- ➢ Choose the right data structures and algorithms
- ➢ Write pseudocode or comments
- ➢ Implement and test your code
- ➢ Optimize and refactor your code
- ➢ Learn from others and practice
- ➢ Divide and conquer algorithms are the algorithms that break a problem down into smaller subproblems, which can lead to more efficient solutions.



THE ANALYSIS OF ALGORITHMS

- ❖ Algorithm analysis is the study of how algorithms perform in terms of time and space, and how much work they do to complete a task.
- ❖ It's a key part of computer science.
- ❖ Here are some concepts related to algorithm analysis:

1. **Algorithm complexity**

- ❖ It is a measure of how many resources an algorithm needs as the input size goes on increasing.

2. **Asymptotic analysis**

- ❖ A theoretical analysis that uses mathematical notations to understand how algorithms perform in terms of run-time.

THE ANALYSIS OF ALGORITHMS

3. Big O notation

❖ A mathematical notation used to describe an algorithm's time complexity.

❖ It can be used to estimate running time complexity in different scenarios, such as the best, average, or worst case.

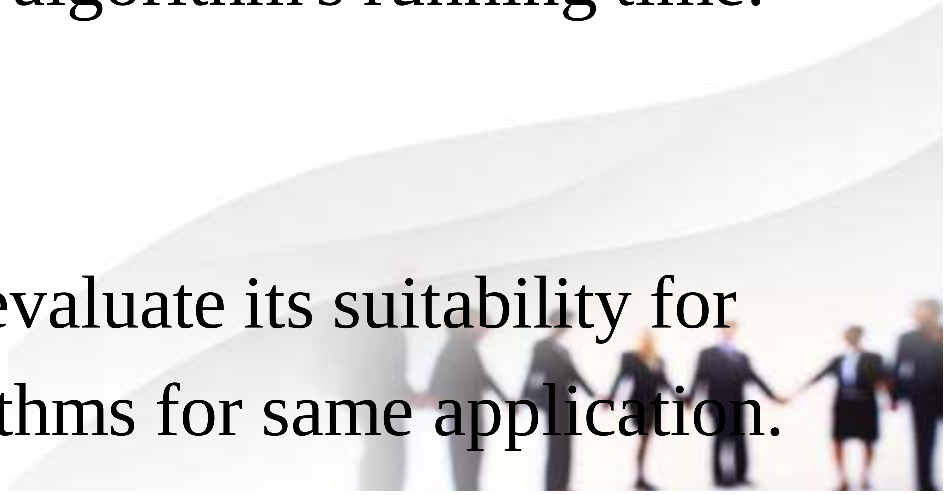
4. Theta notation

❖ It symbolizes the upper and lower bounds of an algorithm's running time.

5. Algorithm efficiency

❖ A measure of how well an algorithm performs.

❖ The main purpose of analyzing algorithm is to evaluate its suitability for different applications/compare it to other algorithms for same application.



THE ANALYSIS OF ALGORITHMS

6. Algorithmic Analysis

- ❖ Evaluate the performance of algorithms using mathematical models or empirical studies.

7. Big-O Notation

- ❖ Express the upper bound of an algorithm's time complexity to understand its scalability.

8. Trade-offs

- ❖ Consider the trade-offs between time complexity, space complexity, and other factors when choosing algorithms.

FUNDAMENTAL ALGORITHMS

❖ Fundamental algorithms are the basic, essential procedures and techniques used in CS and programming to solve common computational problems.

❖ They serve as foundational building blocks for more complex algorithms and are crucial for understanding algorithmic design and analysis.

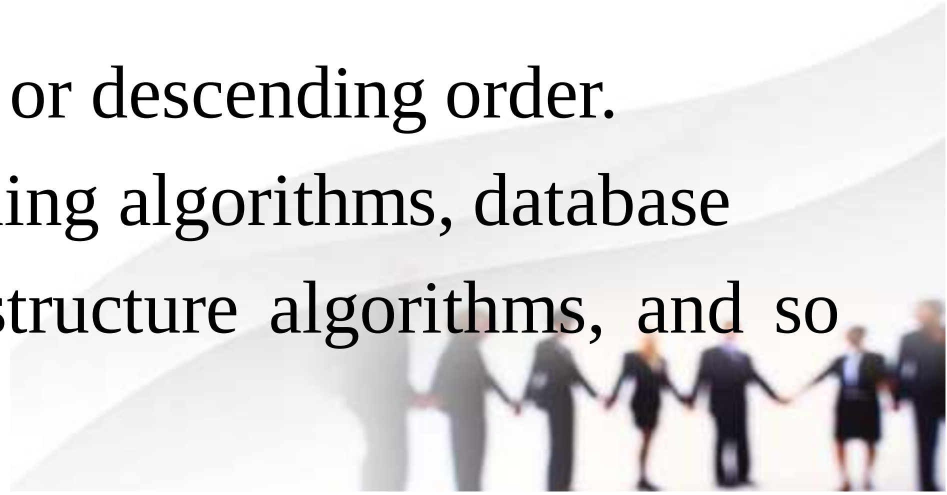
❖ Here's an explanation of the key aspects of fundamental algorithms:

1. Sorting Algorithms

❖ ***Purpose:*** Arrange the elements in an ascending or descending order.

❖ These algorithms are extensively used in searching algorithms, database algorithms, divide & conquer methods, data structure algorithms, and so on.

❖ ***Examples:*** Bubble Sort, Merge Sort, QuickSort



FUNDAMENTAL ALGORITHMS

2. Searching Algorithms

❖ ***Purpose*****:** Locate a specific item within a collection of elements.

❖ ***Examples*****:** Binary Search, Linear Search.

3. Graph Algorithms

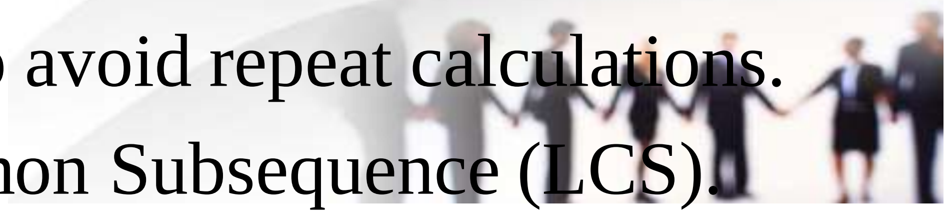
❖ ***Purpose*****:** Analyze relationships & connections between entities that are represented as vertices and edges.

❖ ***Examples*****:** BFS, DFS, Dijkstra's Algorithm, etc.

4. Dynamic Programming

❖ ***Purpose*****:** Solve problems by breaking down into subproblems and solving each subproblem only once, storing solutions to avoid repeat calculations.

❖ ***Examples***: Fibonacci Sequence, Longest Common Subsequence (LCS).



FUNDAMENTAL ALGORITHMS

5. Greedy Algorithms

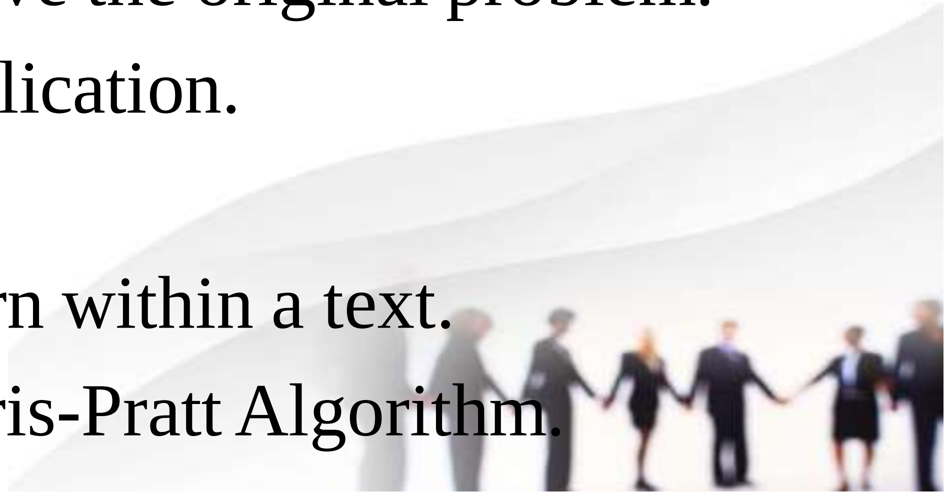
- ***Purpose***: Make the optimal choices at each stage to find a global optimum.
- ***Examples***: Dijkstra's Algorithm, Kruskal's Algorithm.

6. Divide and Conquer

- ***Purpose***: Break down a problem into smaller subproblems, solve them independently, and combine the solutions to solve the original problem.
- ***Examples***: QuickSort, Strassen's Matrix Multiplication.

7. String Matching

- ***Purpose***: Find occurrences of a particular pattern within a text.
- ***Examples***: Naive String Matching, Knuth-Morris-Pratt Algorithm.



FUNDAMENTAL ALGORITHMS

8. Tree Algorithms

❖ ***Purpose***: Manipulate and traverse tree structures efficiently.

❖ ***Examples***: Binary Tree Traversal, Binary Search Tree (BST) Operations.

9. Hashing

❖ ***Purpose***: Quickly locate a data record given its search key.

❖ ***Examples***: Hash Table.

10. Backtracking

❖ ***Purpose***: Systematically search through all possible solutions to find the correct one for a given problem.

❖ ***Examples***: N-Queens Problem.

FUNDAMENTAL ALGORITHMS

❑ **Importance of Fundamental Algorithms:**

❖ **Problem Solving**

➢ Fundamental algorithms provide proven methods for solving common computational problems.

❖ **Efficiency**

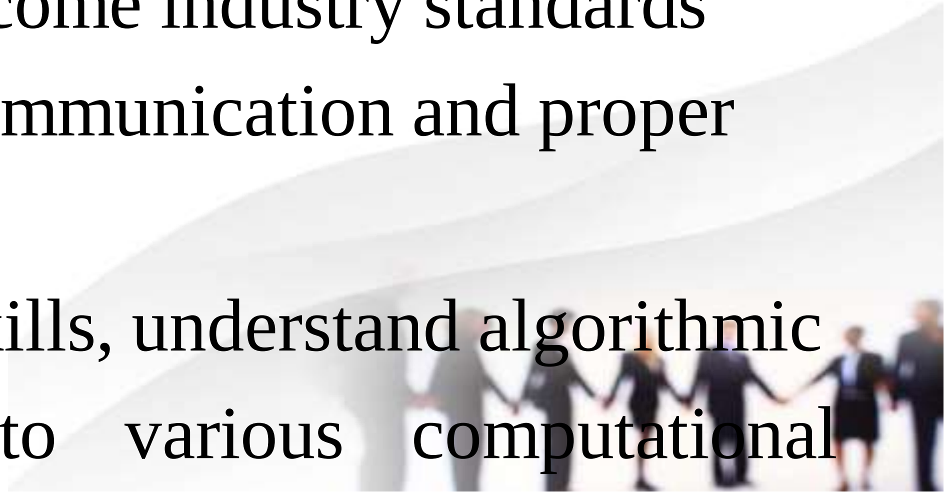
➢ They are designed to be efficient in terms of time and space complexity.

❖ **Understanding Complexity**

➢ Studying these algorithms helps in understanding fundamental concepts of time & space complexity which is crucial for analyzing their performance.

FUNDAMENTAL ALGORITHMS

- ❖ **Foundation for Advanced Algorithms**
  - ➢ More complex algorithms often build upon the fundamental ones.
  - ➢ The fundamental algorithms thus prove to be essential for developing advanced computational solutions.
- ❖ **Standardization**
  - ➢ Many of these fundamental algorithms have become industry standards
  - ➢ Understanding them is essential for effective communication and proper collaboration in the field of computer science.
- ❖ Individuals can enhance the problem-solving skills, understand algorithmic principles and apply efficient solutions to various computational



GENERAL PROBLEM STRATEGIES

- ❖ The general problem-solving strategies can be applied to a wide range of challenges and are not limited to specific domains.
- ❖ These strategies can be helpful in various aspects of life, including academics, work and personal development. They are as follows:

1. Define the Problem

- ❖ Clearly articulate the problem.
- ❖ Understand the scope, constraints, and desired outcome.

2. Understand the Context

- ❖ Consider the broader context and implications of the problem.
- ❖ How does it fit into the larger picture?

GENERAL PROBLEM STRATEGIES

3. Analyze the Causes

- ❖ Identify the root causes of the problem.
- ❖ Understanding why the problem exists is key to finding effective solutions.

4. Generate Possible Solutions

- ❖ Brainstorm multiple solutions without evaluating them initially.
- ❖ Encourage creativity and explore diverse options.

5. Evaluate Options

- ❖ Assess the pros and cons of each solution.
- ❖ Consider feasibility, potential risks, and the impact of each option.

GENERAL PROBLEM STRATEGIES

6. Prioritize Solutions

- ❖ Rank the potential solutions based on their effectiveness, feasibility, and alignment with goals.

7. Break Down the Problem

- ❖ Divide a problem into smaller, more manageable parts.
- ❖ Address each part systematically.

8. Iterate and Refine

- ❖ Be open to refining your approach based on feedback and new information.
- ❖ Problem-solving is often an iterative process.

GENERAL PROBLEM STRATEGIES

9. Seek Input from Others

❖ Collaborate with others to gain different perspectives.

❖ Fresh viewpoints can lead to innovative solutions.

10. Use Analogies

❖ Draw parallels between the current problem and similar problems you or others have faced in the past.

11. Trial and Error

❖ Experiment with different approaches.

❖ Learn from failures and use them to adjust your strategy.

GENERAL PROBLEM STRATEGIES

12. Apply Critical Thinking

- ❖ Analyze information objectively, question assumptions and consider some alternative viewpoints.

13. Mind Mapping

- ❖ Create visual representations of the problem and potential solutions.
- ❖ This can help organize thoughts and relationships.

14. Time Management

- ❖ Allocate time effectively for each stage of the problem-solving process.
- ❖ Avoid spending too much time on one aspect.

GENERAL PROBLEM STRATEGIES

15. Consider Constraints

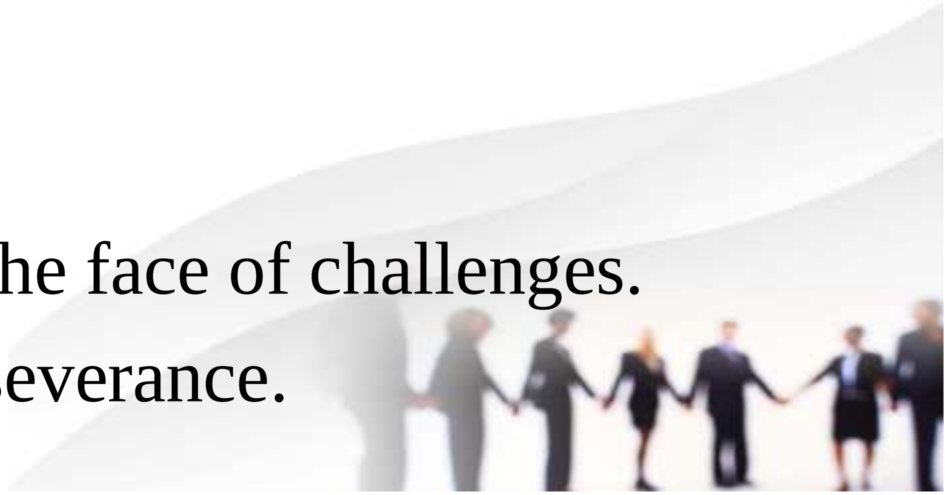
- ❖ Take into account any limitations or constraints that may affect the feasibility of certain solutions.

16. Use Technology

- ❖ Leverage tools and technology to gather information, model scenarios, and analyze data.

17. Stay Positive and Resilient

- ❖ Maintain a positive mindset and be resilient in the face of challenges.
- ❖ A positive attitude can foster creativity and perseverance.



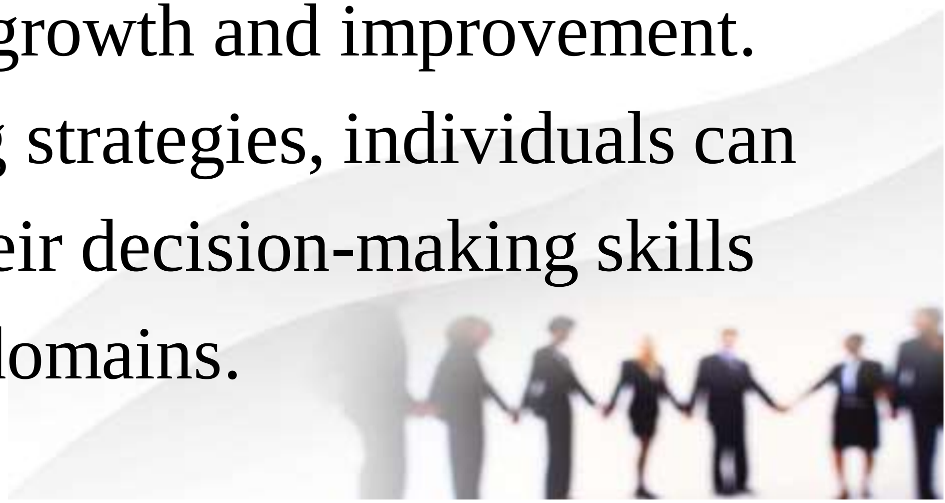
GENERAL PROBLEM STRATEGIES

18. Reflect on the Process

- ❖ After solving the problem, reflect on the process.
- ❖ What worked well & what to improve for future problem-solving efforts?

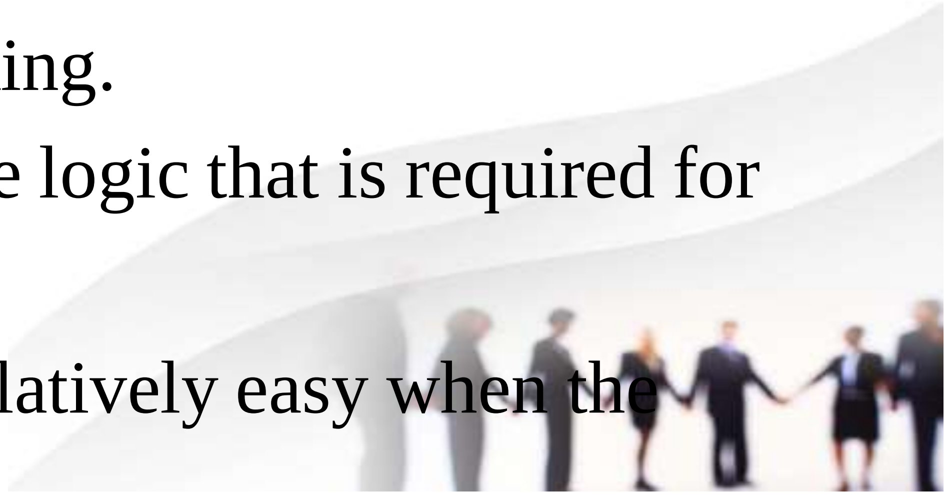
19. Continuous Learning

- ❖ Embrace a mindset of continuous learning.
- ❖ Each problem-solving experience provides the growth and improvement.
- ❖ By incorporating these general problem-solving strategies, individuals can approach challenges systematically, enhance their decision-making skills and develop effective solutions across various domains.



INTRODUCTION TO PROGRAM PLANNING TOOLS

- ❖ In the context of program planning, several tools are utilized to design, organize, and communicate the structure and logic of a computer program.
- ❖ Some tools for program planning are algorithm, flowchart and pseudo code.
- ❖ All these tools play a vital role in the field of computer programming.
- ❖ It is always expected to create an algorithm, flowchart as well as pseudo code before starting the actual job of programming.
- ❖ These tools are certainly helpful for building the logic that is required for writing a proper computer program.
- ❖ The task of computer programming becomes relatively easy when the programmers use all these tools



INTRODUCTION TO PROGRAM PLANNING TOOLS

- **ALGORITHMS**

❑ **Definition**

❖ An algorithm is a step-by-step procedure or a set of well-defined rules for solving a specific problem or accomplishing a particular task.

❑ **Purpose in Program Planning**

❖ Provides a clear and unambiguous description of the solution.

❖ Serves as the foundation for writing actual code.

❖ Offers a high-level understanding of the logic behind the program.

INTRODUCTION TO PROGRAM PLANNING TOOLS

- **FLOWCHARTS**

❑ **Definition**

❖ A flowchart is a graphical representation that depicts the flow of control and data in a program.

❖ It uses various symbols and arrows to illustrate the sequence of steps and decision points.

❑ **Purpose in Program Planning**

❖ Visualizes the logical structure and flow of the program.

❖ Helps identify and understand decision points and processes.

❖ Aids in communication and collaboration among team members.

INTRODUCTION TO PROGRAM PLANNING TOOLS

- **PSUEDO CODE**

❑ **Definition**

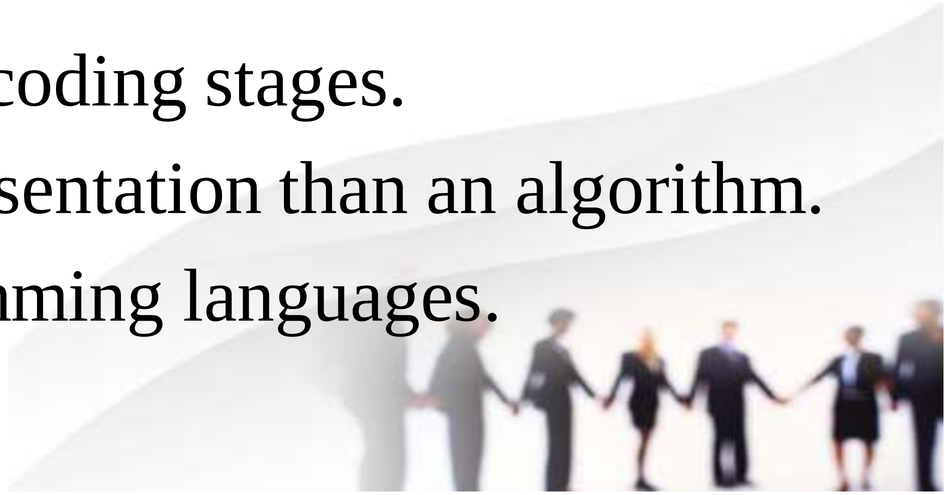
❖ Pseudo code is a simple, high-level description of a computer program or algorithm that uses the combination of natural language along with the basic programming constructs.

❑ **Purpose in Program Planning**

❖ Provides a bridge between the algorithmic and coding stages.

❖ Allows for a more detailed and structured representation than an algorithm.

❖ Facilitates easier translation into actual programming languages.



INTRODUCTION TO PROGRAM PLANNING TOOLS

❑ **How They Work Together?**

1.Algorithm to Flowchart

❖ Start with an algorithm, defining the logic of the program.

❖ One can translate the given algorithm into a flowchart in order to visualize the flow of control and decision points.

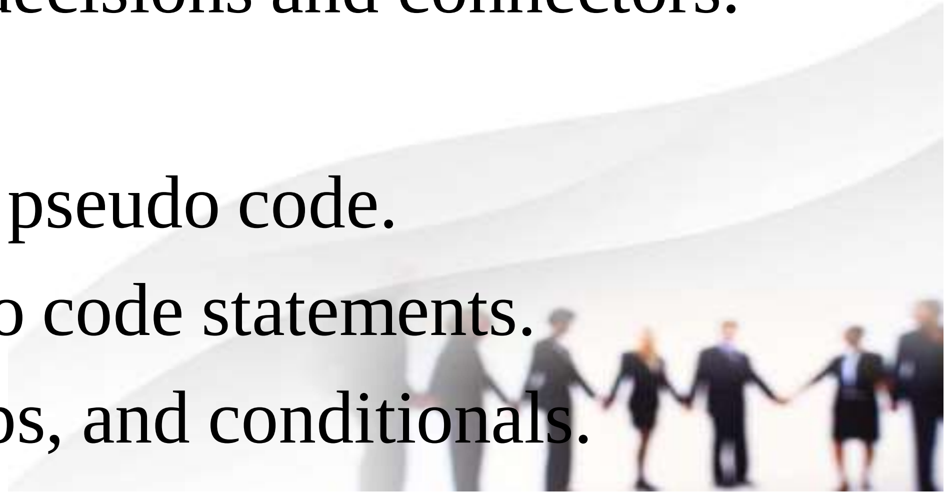
❖ Use flowchart symbols to represent processes, decisions and connectors.

2.Flowchart to Pseudo Code

❖ Use the flowchart as a guide to writing detailed pseudo code.

❖ Translate each flowchart symbol into the pseudo code statements.

❖ Include necessary details such as variables, loops, and conditionals.



INTRODUCTION TO PROGRAM PLANNING TOOLS

3.Pseudo Code to Program Code

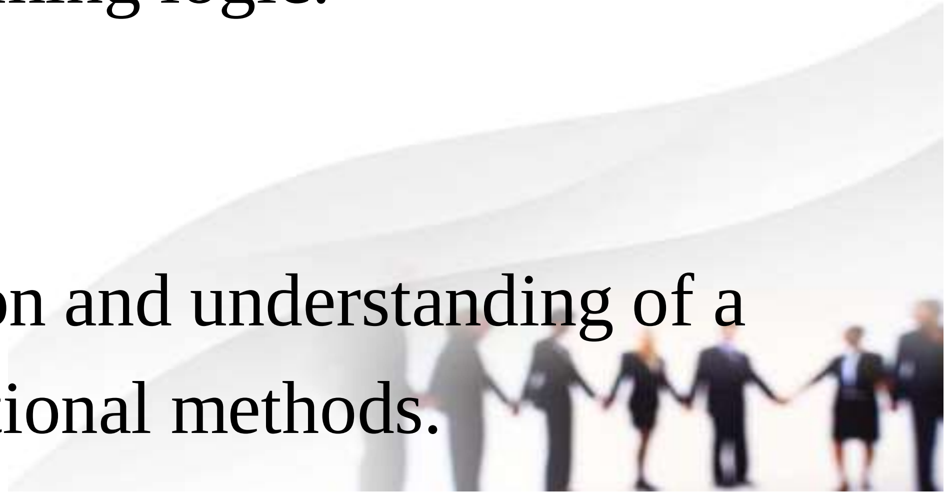
- ❖ Convert pseudo code into actual programming code in a specific language.
- ❖ Consider the syntax and conventions of the chosen programming language.
- ❖ Implement the logic defined by the algorithm, flowchart and pseudo code.

INTRODUCTION TO PROGRAMMING LOGIC

- Programming logic is a fundamental aspect of computer programming that focuses on designing algorithms and writing code to solve problems.
- It involves the application of mathematical and logical concepts to create efficient and effective solutions.
- Programming logic building involves a number of key elements, in general.
- Here's an overview of key elements in programming logic:

1. **Problem Solving**

- ***Definition***
- Programming logic begins with the identification and understanding of a problem that needs to be solved using computational methods.



INTRODUCTION TO PROGRAMMING LOGIC

- ***Approach***
- Break down complex problems into smaller, more manageable parts, and devise step-by-step solutions.

2. Algorithms

- ***Definition***
- Algorithms are precise, unambiguous sets of instructions that specify a sequence of operations to be executed to solve a particular problem.
- ***Importance***
- Algorithms serve as the blueprint for writing code, providing a structured and logical approach to problem-solving.

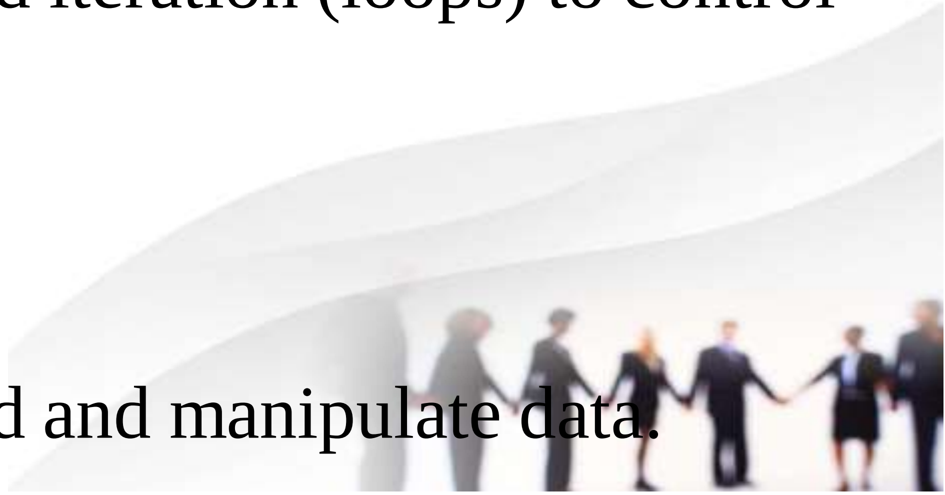
INTRODUCTION TO PROGRAMMING LOGIC

3. Sequence and Control Structures

- ***<u>Sequence</u>***
- The order in which statements are executed, representing the flow of control from one instruction to the next.
- ***<u>Control Structures</u>***
- Include decision-making (if-else statements) and iteration (loops) to control the flow of the program.

4. Variables and Data Types

- ***<u>Variables</u>***
- Represent storage locations in a program to hold and manipulate data.



INTRODUCTION TO PROGRAMMING LOGIC

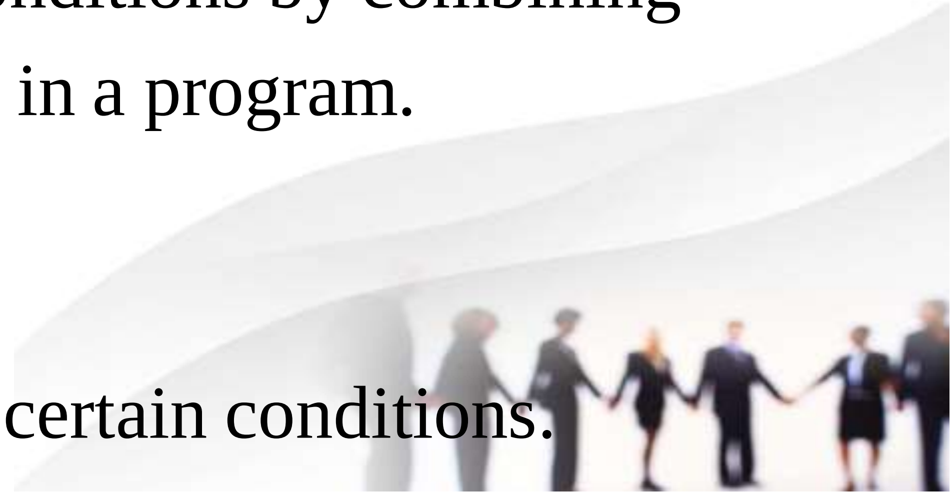
- ***Data Types***
- Specify the type of data a variable can store.
- *Examples*: Integers, floating-point numbers, strings, etc.

5. Logical Operators

- ***AND, OR, NOT***
- Logical operators are used to create complex conditions by combining simpler conditions, enhancing decision-making in a program.

6. Conditional Statements

- ***If-else Statements***
- Allow the program to make decisions based on certain conditions.



INTRODUCTION TO PROGRAMMING LOGIC

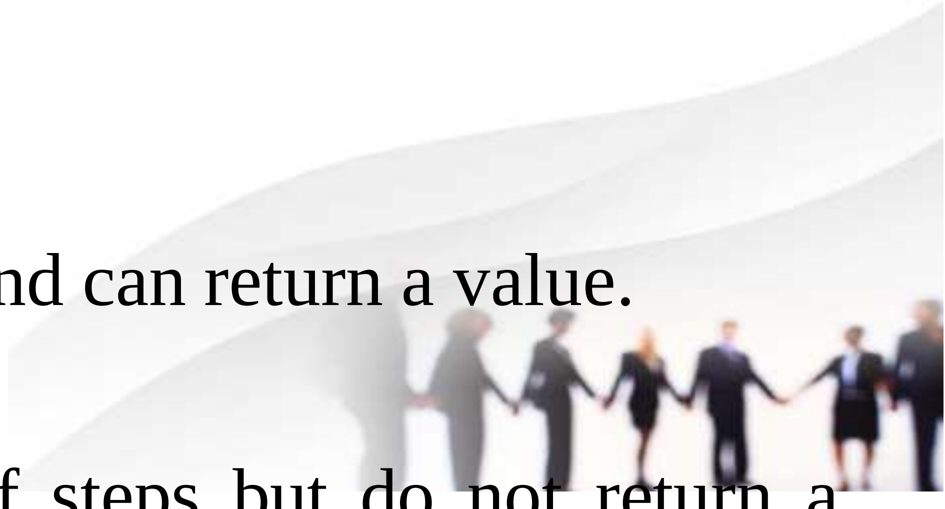
- ❖ ***<u>Switch Statements</u>***
- ❖ Provides a way to select from various options based on value of expression.

7. Loops (Iteration)

- ❖ ***<u>For, while, do-while Loops</u>***
- ❖ Enable the repetition of a set of statements allowing their efficient handling.

8. Functions and Procedures

- ❖ ***<u>Functions</u>***
- ❖ Segments of code that perform a specific task and can return a value.
- ❖ ***<u>Procedures</u>***
- ❖ Just like functions, they carry out a series of steps but do not return a



INTRODUCTION TO PROGRAMMING LOGIC

9. Arrays and Lists

❖ ***Arrays***

❖ Collections of elements of the same data type, accessed using an index.

❖ ***Lists***

❖ Dynamic arrays that can grow or shrink in size.

10. Debugging and Error Handling

❖ ***Debugging***

❖ Identifying and fixing errors in the code.

❖ ***Error Handling***

❖ Implementing strategies to manage errors and unexpected situations.

INTRODUCTION TO PROGRAMMING LOGIC

11. Modularity

- ❖ Breaking down a program into smaller, self-contained modules or functions for better organization and maintainability.

12. Documentation

- ❖ ***<u>Comments and Documentation</u>***
- ❖ Adding comments to code and maintaining documentation to enhance code readability and understanding.

13. Object-Oriented Programming (OOP) Concepts

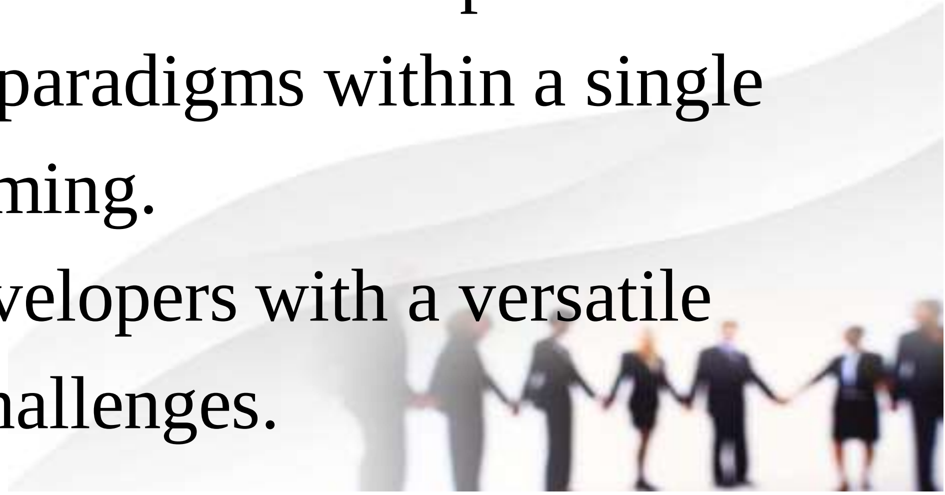
- ❖ ***<u>Encapsulation, Inheritance, Polymorphism, Abstraction</u>***
- ❖ Principles of OOP that enhance code organization and reusability.

UNIT II

PROGRAMMING PARADIGM

OVERVIEW OF PROGRAMMING PARADIGM

- ❖ A programming paradigm is fundamental style/approach to programming which defines the way a programmer organizes and structures code to solve the given problems.
- ❖ Each paradigm comes with its own concepts, principles, and techniques.
- ❖ The choice of a programming paradigm depends on the nature of problem, goals of the software and preferences of programmer or development team.
- ❖ Often, modern programming involves a mix of paradigms within a single application, known as multi-paradigm programming.
- ❖ Understanding different paradigms provides developers with a versatile toolkit for approaching diverse programming challenges.



OVERVIEW OF PROGRAMMING PARADIGM

1. Imperative Programming

❖ Programs consist of statements that change a program's state.

❖ Emphasizes the sequence of steps to achieve a specific goal.

❖ Uses variables to store and manipulate data.

❖ Example Languages: C, Fortran.

2. Procedural Programming

❖ □Organizes code into procedures or functions.

❖ □Focuses on procedures that perform tasks.

❖ □Uses a step-by-step approach to problem-solving.

❖ □Example Languages: Pascal, C.

OVERVIEW OF PROGRAMMING PARADIGM

3. Object-Oriented Programming (OOP)

- ❖ Organizes code into classes and objects.
- ❖ Encapsulation, inheritance, and polymorphism are core principles.
- ❖ Emphasizes the modeling of real-world entities.
- ❖ Example Languages: Java, C++, Python.

4. Functional Programming

- ❖ Treats computation as the evaluation of mathematical functions.
- ❖ Avoids changing state and mutable data.
- ❖ Focuses on pure functions and immutability.
- ❖ Example Languages: Haskell, Lisp, Scala.

OVERVIEW OF PROGRAMMING PARADIGM

5. Declarative Programming

- ❖ Describes what the program should accomplish without specifying how.
- ❖ Emphasizes expressing the logic and rules rather than procedures.
- ❖ SQL is a declarative language for database queries.
- ❖ Example Languages: SQL, HTML.

6. Event-Driven Programming

- ❖ Responds to events triggered by user actions or system events.
- ❖ Utilizes event handlers to manage responses.
- ❖ Common in graphical user interfaces (GUIs).
- ❖ Example Languages: JavaScript, Python (with libraries like Tkinter).

OVERVIEW OF PROGRAMMING PARADIGM

7. Logic Programming

- ❖ Represents a program as a set of logical statements.
- ❖ Utilizes rules and facts to express relationships and constraints.
- ❖ Predicates are used to make logical inferences.
- ❖ Example Languages: Prolog.

8. Parallel Programming:

- ❖ Focuses on concurrent execution of tasks.
- ❖ Utilizes parallelism for increased performance.
- ❖ Common in high-performance computing and distributed systems.
- ❖ Example Languages: CUDA, OpenMP.

OVERVIEW OF PROGRAMMING PARADIGM

9. Scripting Languages

- ❖ Designed for automation and quick development.
- ❖ Often interpreted and dynamically typed.
- ❖ Used for system administration, web development, and data analysis tasks.
- ❖ Example Languages: Python, Ruby, Shell scripting languages.

10. Aspect-Oriented Programming (AOP)

- ❖ Separates concerns by modularizing cross-cutting concerns.
- ❖ Allows the definition of aspects that cut across modules.
- ❖ Enhances modularity and maintainability.
- ❖ Example Languages: AspectJ.

BASIC ELEMENTS OF PROGRAMMING LANGUAGES

- ❖ The basic elements of programming languages are fundamental building blocks that programmers use to create instructions & express algorithms.
- ❖ These elements are common across various programming languages and they form the syntax and structure of the code.
- ❖ Understanding and mastering these basic elements is crucial for any programmer, as they form the foundation for expressing algorithms, building logic, and creating functional and efficient software.
- ❖ The combination and arrangement of these elements define the uniqueness of different programming languages.

BASIC ELEMENTS OF PROGRAMMING LANGUAGES

Here are the key basic elements:

1. Variables

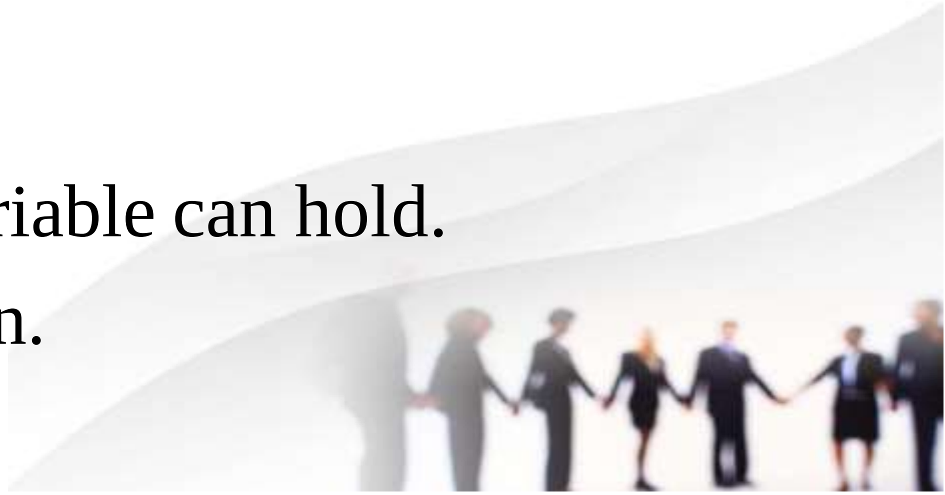
- ❖ ***Definition***: Containers that store data values.
- ❖ ***Role***: Used to represent and manipulate data within a program.
- ❖ ***Example***:

```
age = 25
name = "John"
```

2. Data Types:

- ❖ ***Definition***: Specifies the type of data that a variable can hold.
- ❖ ***Common Types***: Integer, Float, String, Boolean.

□



BASIC ELEMENTS OF PROGRAMMING LANGUAGES

❖ ***Example***:

```
int number = 42;
double pi = 3.14;
```

3. Operators

❖ ***Definition***: Symbols that perform operations on variables and values.

❖ ***Common Operators***: Arithmetic (+, -, *, /), Comparison (==, !=, <, >), Logical (&&, ||, !)

❖ ***Example***:

```
result = 10 + 5
is_equal = (a == b)
```

□

BASIC ELEMENTS OF PROGRAMMING LANGUAGES

4. Control Structures

- ***Definition***: Statements that control the flow of execution.
- ***Common Structures***: Conditional Statements (if, else, switch), Loops (for, while, do-while).
- ***Example***:

```
if (x > 0)
        {
        // do something
        }
Else
        { // do something else }
```

BASIC ELEMENTS OF PROGRAMMING LANGUAGES

5. Functions/Methods

❖ ***Definition***: Blocks of code that perform a specific task.

❖ ***Role***: Encapsulate and modularize code, promoting reusability.

❖ ***Example***:

```
def greet(name): return "Hello, " + name + "!"
```

6. Arrays and Lists

❖ ***Definition***: Ordered collections of elements.

❖ ***Role***: Store and manipulate multiple values under a single name.

❖ ***Example***:

```
var numbers = [1, 2, 3, 4, 5];
```

□

BASIC ELEMENTS OF PROGRAMMING LANGUAGES

7. Objects and Classes

❖ ***Definition***: Object is instance of a class representing real-world entities.

❖ ***Role***: Support Object-Oriented Programming (OOP) principles.

❖ ***Example***:

```
class Car { String model; int year; }
```

8. Comments

❖ ***Definition***: Annotation within code for documentation or clarification.

❖ ***Role***: Enhance code readability and provide context.

❖ ***Example***:

```
# This is a single-line comment
""" This is a multi-line comment """
```

BASIC ELEMENTS OF PROGRAMMING LANGUAGES

9. Input/Output (I/O)

- ***Definition***: These are the operations that involve taking input from users and displaying output.
- ***Role***: It is used to facilitate interaction between the program and the user or an external systems.
- ***Example***:

```
Scanner scanner = new Scanner(System.in);
int userInput = scanner.nextInt();
System.out.println("User input: " + userInput);
```

BASIC ELEMENTS OF PROGRAMMING LANGUAGES

10. Constants

- ***Definition***: They are the fixed values that do not change during the execution of the program.
- ***Role***: They are generally used for values that should remain constant throughout the program.
- ***Example***:

```
#define PI 3.14
#define g 9.81
#define R 8.314
```

BASIC ELEMENTS OF PROGRAMMING LANGUAGES

11. Variables Scope

- ***Definition***: The region of the program where a variable is accessible.
- ***Common Scopes***:

Local scope (within a function),

Global scope (accessible throughout the program).

- ***Example***:

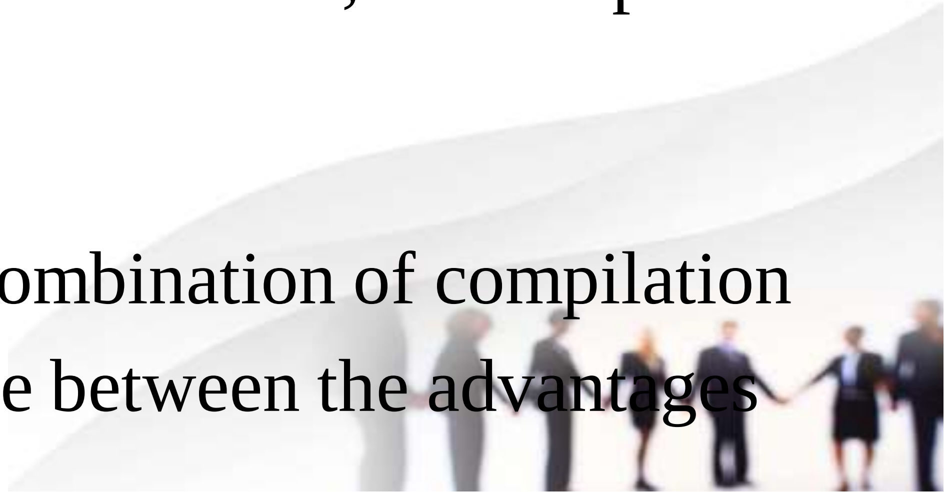
Python code

```
global_variable = 10
def my_function(): local_variable = 5
```

□

COMPILED VS. INTERPRETED LANGUAGES

- "Compiled" and "interpreted" refer to two different approaches to executing code in programming languages.
- These terms describe how a program written in a specific language is translated into machine code and executed by the computer.
- The choice between the two, however, depends on specific requirements of the project, including performance considerations, development speed,
  and platform independence.
- Many modern languages and platforms use a combination of compilation and interpretation techniques to strike a balance between the advantages of both approaches



COMPILED VS. INTERPRETED LANGUAGES

❑ **Compiled Languages**

1. Compilation Process

❖ The entire source code is translated into machine code or an intermediate code by a compiler before execution.

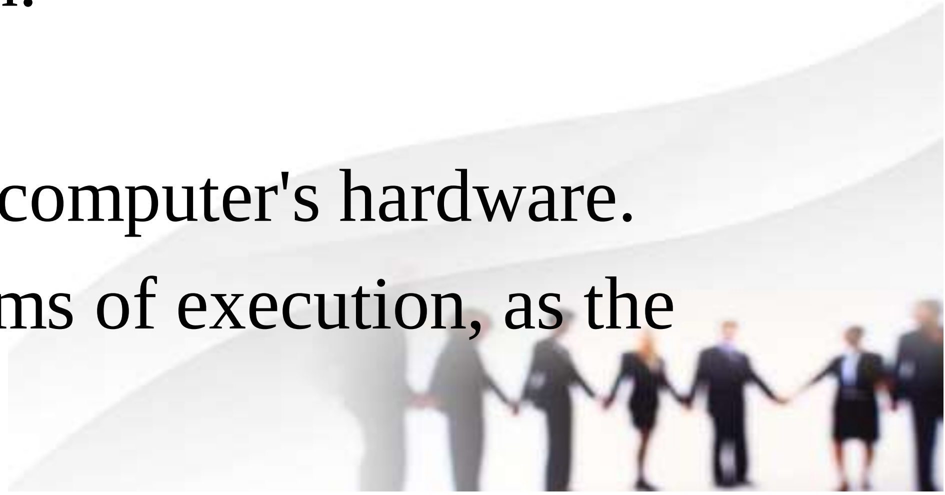
❖ This process results in the creation of an executable file.

❖ Compilation happens before the program is run.

2. Execution

❖ The compiled code is executed directly by the computer's hardware.

❖ The compiled program tends to be faster in terms of execution, as the translation has already occurred.



COMPILED VS. INTERPRETED LANGUAGES

3. Examples

❖ C, C++, Rust, Fortran.

4. Advantages

❖ Faster execution as the code is already translated into machine code.

❖ Can optimize code for specific hardware.

❖ The application is not performance-sensitive.

5. Disadvantages

❖ Longer development cycles due to the compilation step.

❖ Platform-dependent executables

❖ May need recompilation for different platforms.

COMPILED VS. INTERPRETED LANGUAGES

❑ **Interpreted Languages**

1. Interpretation Process

❖ The source code is read and executed line by line by an interpreter during the runtime.

❖ There is no separate compilation step in case of interpreted languages.

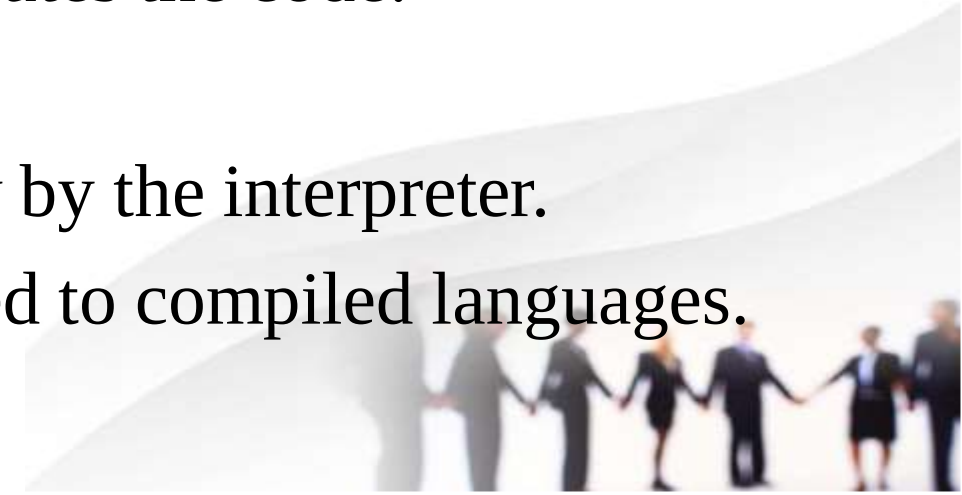
❖ Instead, the interpreter directly reads and executes the code.

2. Execution

❖ The code is translated and executed on-the-fly by the interpreter.

❖ The program's execution is slower as compared to compiled languages.

❖ Hence, the overall execution time increases.



COMPILED VS. INTERPRETED LANGUAGES

3. Examples

❖ Python, JavaScript, Ruby, PHP.

4. Advantages

❖ Shorter development cycles as there is no compilation step.

❖ Code can be platform-independent

❖ As long as an interpreter is available, one can easily implement the same code over different platforms.

5. Disadvantages

❖ Generally slower execution as interpretation happens at runtime.

❖ Code may be less optimized for specific hardware.

COMPILED VS. INTERPRETED LANGUAGES

❑ **Hybrid Approaches**

1. **Just-In-Time (JIT) Compilation**

❖ Combines aspects of both compilation and interpretation.

❖ Code is initially interpreted but is translated into machine code and cached for future executions.

2. **Intermediate Languages**

❖ Some languages, like Java, use an intermediate bytecode that is interpreted by the Java Virtual Machine (JVM).

❖ This allows for platform independence

❖ It means “write once, run anywhere” kind of system.

COMPILED VS. INTERPRETED LANGUAGES

❑ **Choosing Between Compiled and Interpreted Languages**

- **When to Use Compiled Languages?**

❖ Performance is a critical factor.

❖ There is a need for low-level memory control.

❖ The application is computationally intensive.

- **When to Use Interpreted Languages?**

❖ Rapid development is a priority.

❖ Platform independence is crucial.

❖ The application is not performance-sensitive.

SYNTAX

- Understanding the concepts of syntax, semantics and data types is fundamental in programming.
- These terms are crucial for writing correct and meaningful code.
- ***Definition***: Syntax refers to the set of rules that dictate how programs written in a specific programming language should be structured.
- ***Role***: Ensures that the code is grammatically correct.
- ***Example***:

Python code

```
# Correct syntax in Python print("Hello, World!")
```

- ***Importance***: Failure to adhere to proper syntax will result in syntax errors, and the code will not run.

SEMANTICS

- ❖ ***<u>Definition</u>***: Semantics deals with the meaning or interpretation of a program's code.
- ❖ ***<u>Role</u>***: It focuses on the logic and functionality of the code rather than its structure.
- ❖ ***<u>Example</u>***:

Python code

# Correct semantics in Python result = 5 + 3

- ❖ ***<u>Importance</u>***: Code with correct syntax may still produce unexpected or incorrect results if the semantics are flawed.

DATA TYPES

- ❖ ***Definition***: Data types specify the type of values that variables can hold.
- ❖ ***Common Data Types***:
- ❖ Integer, Float (floating-point number), String (text), Boolean (true/false), List (ordered collection), Dictionary (key-value pairs).
- ❖ ***Example***:

```
# Different data types in Python
        age = 25                # Integer
        height = 5.9            # Float
        name = "John"           # String
        is_adult = True         # Boolean
```

DATA TYPES

- ***Role***: Ensures that operations on variables are meaningful and that the correct kind of data is used.
- ***Importance***: Using the wrong data type can lead to errors or unexpected behavior in the program.
- The syntax ensures that code is written in correct & consistent structure.
- While the semantics focuses on the meaning and logic of the code.
- Data types define the nature of variables, ensuring meaningful and accurate operations.
- Proper understanding and application of these concepts are essential for writing reliable and effective programs.

IMPERATIVE AND NON-IMPERATIVE LANGUAGES

❖ Programming languages can be broadly categorized into two main paradigms: imperative and non-imperative (or declarative).

❖ These approaches help to express computation and programming logic.

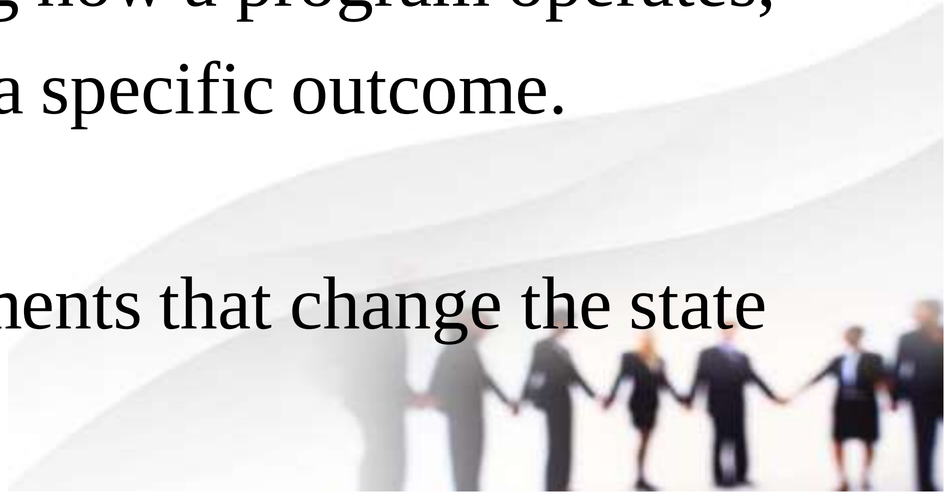
❖ **Imperative Languages**

1. Definition

❖ Imperative programming focuses on describing how a program operates, providing step-by-step instructions to achieve a specific outcome.

2. Key Characteristics

❖ ***State Modification***: Programs consist of statements that change the state of the program.



IMPERATIVE AND NON-IMPERATIVE LANGUAGES

- ***Mutable Variables***: Variables can be assigned new values, and their values can be changed during the program's execution.
- ***Control Flow***: Emphasizes the sequence of steps to perform a computation.
- ***Examples***: C, C++, Java, Python (to some extent).

3. Example (C Programming)

```
#include <stdio.h>
int main()
{
int x = 5; int y = 10;
```

IMPERATIVE AND NON-IMPERATIVE LANGUAGES

```
int sum = x + y;
printf("Sum: %d\n", sum);
return 0;
}
```

4. Advantages

- ❖ Efficient control over the computer's memory and hardware.
- ❖ Well-suited for low-level programming and system-level tasks.

5. Disadvantages

- ❖ Code can become complex and harder to understand.
- ❖ May require explicit management of resources.

IMPERATIVE AND NON-IMPERATIVE LANGUAGES

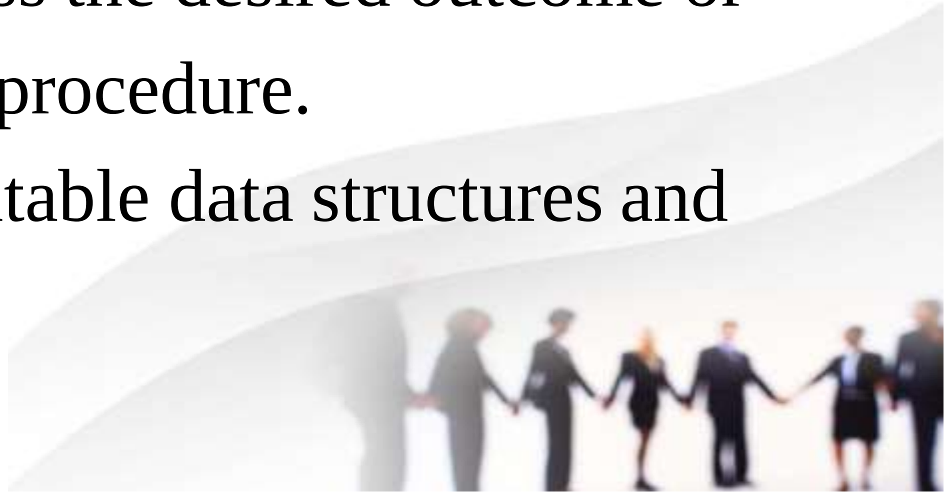
- ❖ **Non-Imperative (Declarative) Languages**

1. **Definition**

- ❖ Non-imperative programming focuses on what the program should accomplish without specifying how to achieve it.

2. **Key Characteristics**

- ❖ ***Declaration of Desired State***: Programs express the desired outcome or properties without specifying the step-by-step procedure.
- ❖ ***Immutable Data***: Emphasizes the use of immutable data structures and avoids modifying state.
- ❖ ***Examples***: Haskell, Lisp, SQL, HTML.



IMPERATIVE AND NON-IMPERATIVE LANGUAGES

3. Example (SQL)

```
SELECT column1, column2 FROM table WHERE condition;
```

4. Advantages

❖ Code tends to be more concise and expressive.

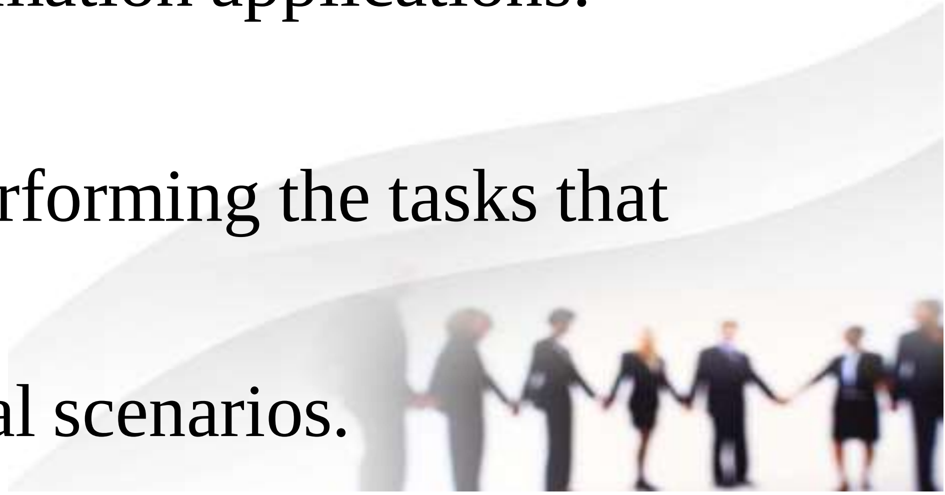
❖ Easier to reason about and parallelize.

❖ Well-suited for data manipulation and transformation applications.

5. Disadvantages

❖ These languages are not always suitable for performing the tasks that require explicit control over state changes.

❖ May not be as efficient in certain computational scenarios.



IMPERATIVE AND NON-IMPERATIVE LANGUAGES

- **Hybrid Approaches**

1. Functional Programming

- A subset of non-imperative programming.
- Emphasizes the use of pure functions, immutability and avoiding the side effects of the progamming constructs.
- Examples include Haskell, Lisp, Python and JavaScript.

2. Object-Oriented Programming (OOP)

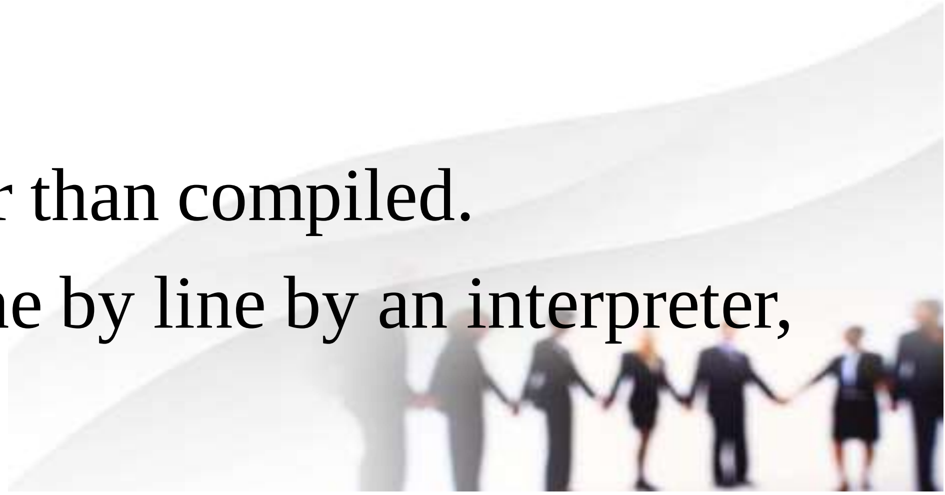
- While often considered imperative, OOP has declarative aspects, especially with encapsulation and abstraction.
- Focuses on modeling real-world entities using classes and objects.

SCRIPTING LANGUAGES

- ❖ Scripting languages are programming languages designed for specific tasks that involve automation, rapid prototyping, and ease of use.
- ❖ These languages are often interpreted and allow developers to write scripts to perform various tasks, ranging from system administration and automation to web development and data analysis.
- ❖ **Characteristics of Scripting Languages**

1. Interpreted

- ❖ Scripting languages are often interpreted rather than compiled.
- ❖ This means that the source code is executed line by line by an interpreter, which simplifies the development process.



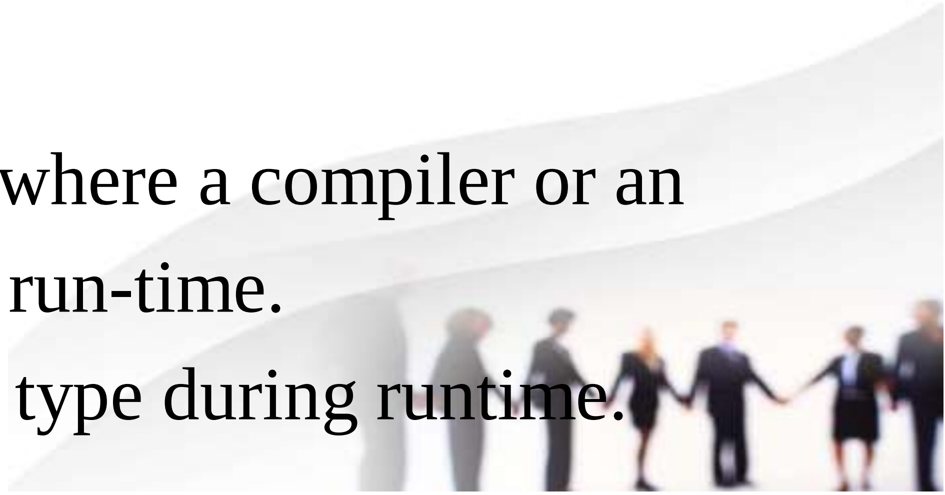
SCRIPTING LANGUAGES

2. High-Level

- ❖ Scripting languages are generally high-level languages.
- ❖ They provide abstractions that only keeps the essentials and removing unnecessary information.
- ❖ The abstractions can be used effectively to simplify complex operations and reduce the need for low-level details.

3. Dynamic Typing

- ❖ Many scripting languages use dynamic typing where a compiler or an interpreter assigns a type to all the variables at run-time.
- ❖ Thus, they allow variables to change their data type during runtime.



SCRIPTING LANGUAGES

4. Scripting for Automation

❖ Commonly used for automating repetitive tasks, system administration, and configuration tasks.

5. Rapid Prototyping

❖ Well-suited for rapid development and prototyping due to their concise syntax and ease of use.

6. Flexibility

❖ Often designed to be flexible and forgiving in nature.

❖ This allows developers to focus on achieving tasks without strict adherence to complex syntax.

SCRIPTING LANGUAGES

❖ **Examples of Scripting Languages:**

1. Python

❖ Versatile scripting language used for

- ➢ Web development
- ➢ Data analysis
- ➢ Machine learning
- ➢ Automation

❖ ***Example:***

Python code

```
print("Hello, World!")
```

SCRIPTING LANGUAGES

2. JavaScript

- Mainly used for scripting in web browsers to make web pages interactive.
- ***Example:***

Javascript code

```
alert("Hello, World!");
```

3. Bash (Shell Scripting)

- For system administration & automation in Unix/Linux environments.
- ***Example:***

Bash code

```
echo "Hello, World!"
```

SCRIPTING LANGUAGES

4. Ruby

- ❖ A dynamic, object-oriented scripting language
- ❖ Used for web development (Ruby on Rails) & general-purpose scripting.
- ❖ ***Example:***

Ruby code

```
puts "Hello, World!"
```

5. Perl

- ❖ Practical Extraction and Reporting Language,
- ❖ Widely used for text processing, system administration and web development.

SCRIPTING LANGUAGES

- ***Example:***

Perl code

```
print "Hello, World!\n";
```

6. PHP

- A premier server-side scripting language
- Prominently used for the web development and creating dynamic web pages as and when necessary.
- **Example:**

PHP code

```
<?php echo "Hello, World!"; ?>
```

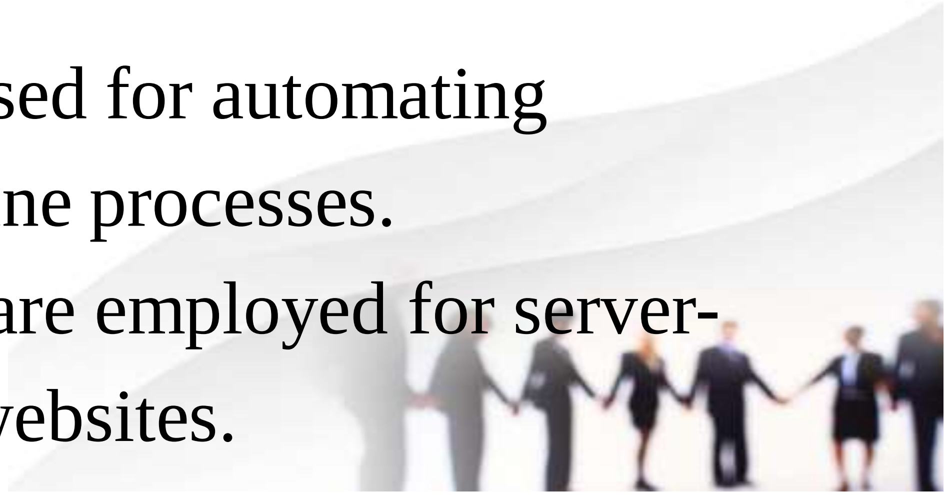
SCRIPTING LANGUAGES

7. Shell Scripting Languages

- ❖ Specific scripting languages for command-line environments, often used for system administration tasks.
- ❖ Powershell code

```
Write-Output "Hello, World!"
```

- ❖ **Shell Scripting Language Use Cases**
- ❖ ***Automation***: Scripting languages are widely used for automating repetitive tasks, system maintenance, and routine processes.
- ❖ ***Web Development***: Many scripting languages are employed for server-side scripting, enhancing the functionality of websites.



SCRIPTING LANGUAGES

- ***Data Analysis***: Python and other scripting languages are popular choices for data analysis and manipulation tasks.
- ***System Administration***: Bash and PowerShell are commonly used for system administration tasks on Unix/Linux and Windows systems, respectively.
- ***Prototyping***: Rapid development and concise syntax make scripting languages ideal for prototyping and quick development cycles.

DATA ORIENTED LANGUAGES

- ❖ Data-oriented languages are programming languages that prioritize efficient manipulation and management of data.
- ❖ These languages are designed to streamline operations related to data processing, storage, and retrieval.
- ❖ They often provide features and constructs that make it easy to work with large datasets and optimize data-centric tasks.
- ❖ Data-oriented languages play a vital role in the field of data science, data analytics and other computational tasks where efficient manipulation of data is crucial.

DATA ORIENTED LANGUAGES

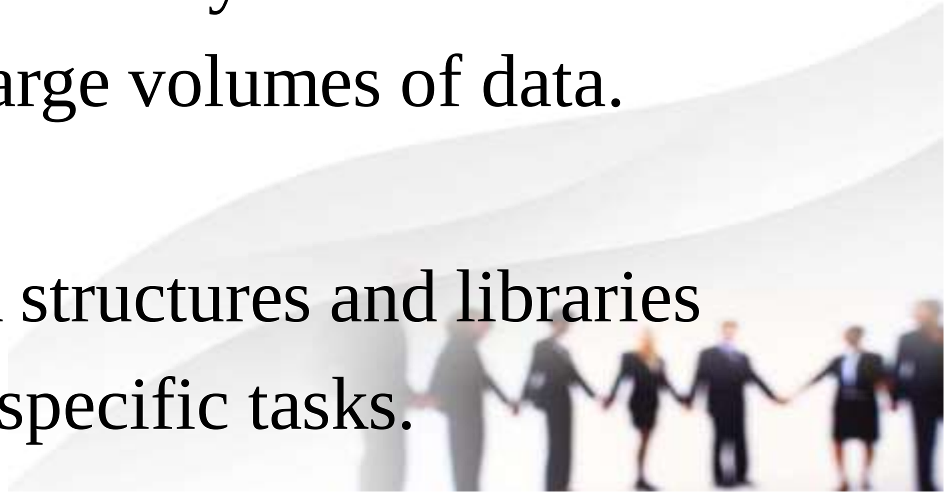
- ❖ The choice of a data-oriented language often depends on the specific requirements of the task at hand, ranging from database management to scientific computing and big data processing.
- ❖ **Characteristics of Data-Oriented Languages**

1. **Efficient Data Processing**

- ❖ The data-oriented languages are designed to efficiently handle the manipulation, transformation and analysis of large volumes of data.

2. **Optimized Data Structures**

- ❖ These languages often include specialized data structures and libraries for handling data in ways that are efficient for specific tasks.



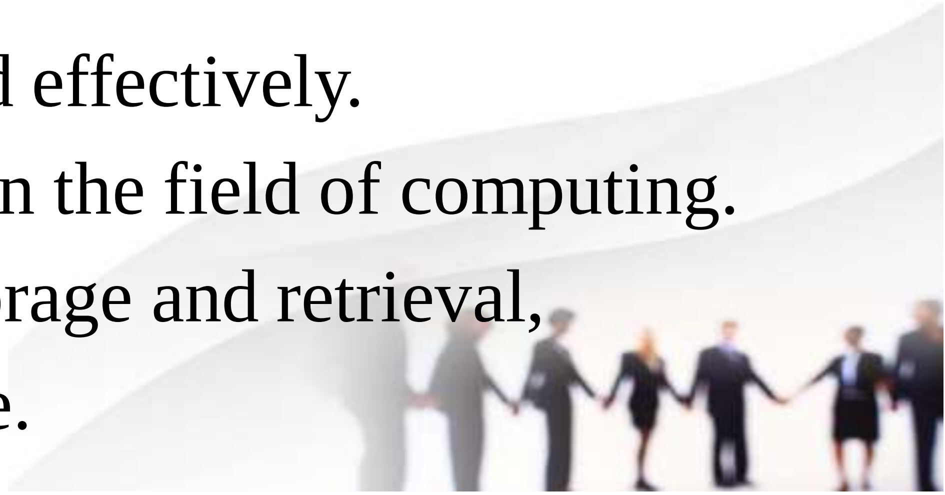
DATA ORIENTED LANGUAGES

3. Parallel and Concurrent Processing

- ❖ Many data-oriented languages are designed to take advantages of the parallelism and concurrency.
- ❖ This allows for faster data processing on modern multi-core systems.

4. Memory Management

- ❖ Memory management is one of the most crucial task as it allows to store and use the computer's memory efficiently and effectively.
- ❖ Efficient memory management is a key focus in the field of computing.
- ❖ It is very essential in order to optimize data storage and retrieval, reducing overhead and improving performance.



DATA ORIENTED LANGUAGES

5. Domain-Specific Features

❖ These languages may include domain-specific features and constructs tailored for tasks such as database operations, data analysis, and scientific computing.

❖ **Examples of Data-Oriented Languages**

1. SQL (Structured Query Language)

❖ Domain-specific language for managing & querying relational databases.

❖ ***Example*:**

SQL code

```
SELECT column1, column2 FROM table WHERE condition;
```

DATA ORIENTED LANGUAGES

2. R Programming

- ❖ A statistical programming language used for data analysis, visualization, and statistical modeling.
- ❖ ***<u>Example</u>***:

```
data <- read.csv("data.csv")
summary(data)
```

3. Julia

- ❖ A high-performance language for technical computing with a focus on data science, machine learning, and scientific computing.

DATA ORIENTED LANGUAGES

❖ ***Example***:

```
function square(x)
    return x^2
        end
result = square(5)
```

4. (I) MATLAB

❖ A numerical computing language used for matrix manipulation, data analysis and algorithm development.

❖ ***Example***:

```
A = [1 2; 3 4];
B = A * 2;
```

DATA ORIENTED LANGUAGES

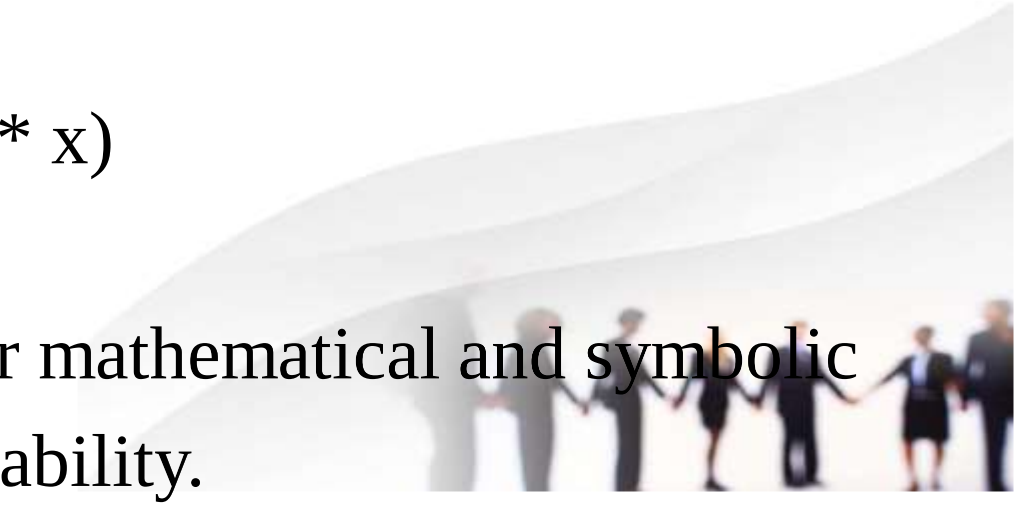
(II) SCALA

- A general-purpose programming language that integrates the features of object-oriented and functional programming.
- These features are primarily used for large-scale data processing with Apache Spark.
- ***<u>Example</u>***:

```
val data = list (1, 2, 3, 4, 5)
val square = data.map (x => x * x)
```

5. Haskell:

- A functional programming language used for mathematical and symbolic computation, emphasizing purity and immutability.



DATA ORIENTED LANGUAGES

- ***Example***:

```
square :: Int -> Int
square x = x * x
```

- ***Use Cases***:
- **Data Analysis and Statistics**
- These languages are commonly used for statistical analysis, data visualization, and processing large datasets.
- **Scientific Computing**
- Data-oriented languages are applied in scientific research and computational tasks, especially those involving simulations and numerical analysis.

DATA ORIENTED LANGUAGES

- ❖ **Database Management**
- ❖ SQL is a standard for managing and querying relational databases, playing a critical role in data storage and retrieval.
- ❖ **Machine Learning and AI**
- ❖ Some data-oriented languages, like R and Julia, are popular choices for developing machine learning models and algorithms.
- ❖ **Big Data Processing**
- ❖ Languages like Scala, used with Apache Spark, enable large-scale data processing and analysis in distributed computing environments.

OBJECT ORIENTED LANGUAGES

- ❖ Object oriented programming (OOP) languages are the languages that are based on the principles of object-oriented programming.
- ❖ OOP is a programming paradigm that uses objects (bundles of data) and associated methods to design and structure code.
- ❖ **Characteristics of Object-Oriented Languages**

1. Objects

- ❖ Objects are instances of classes
- ❖ They encapsulate data (attributes) and methods (functions) that operate on that data.

OBJECT ORIENTED LANGUAGES

2. Classes

❖ Classes are blueprints or templates for creating objects.

❖ They define the structure and behavior of objects.

3. Encapsulation

❖ Encapsulation involves bundling data (attributes) and the methods that operate on that data into a single unit (object).

4. Inheritance

❖ Inheritance allows a class to inherit properties and behaviors from another class, promoting code reuse and creating a hierarchy of classes.

OBJECT ORIENTED LANGUAGES

5. Polymorphism

❖ Polymorphism allows objects of different classes to be treated as objects of a common superclass.

❖ It enables flexibility and extensibility in code.

6. Abstraction

❖ Abstraction involves simplifying complex systems by modeling classes based on the essential properties and behaviors they share.

OBJECT ORIENTED LANGUAGES

- **Examples of Object Oriented Languages**

1. Java

- A general purpose, class based and object oriented programming language known for its platform independence.
- Platform independent language means “write once, run anywhere” kind of language.

2. C++

- An extension of the C programming language with features like classes and objects.
- It supports both procedural and object-oriented programming.

OBJECT ORIENTED LANGUAGES

3. C#

❖ It is developed by Microsoft.

❖ C# is a modern, object oriented programming language designed for building Windows applications and web services.

4. Python

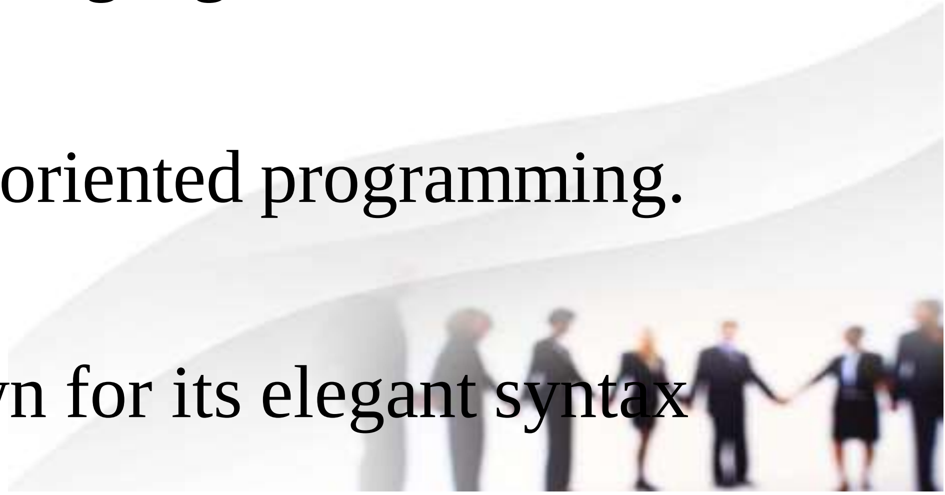
❖ Python is a versatile, high-level programming language

❖ It has a clear and concise syntax.

❖ It usually supports both procedural and object-oriented programming.

5.Ruby:

❖ Ruby is dynamic and reflective language known for its elegant syntax



OBJECT ORIENTED LANGUAGES

❑ **Use Cases of Object Oriented Languages**

1. Software Design and Modeling

❖ Object-oriented languages are particularly well-suited for modeling the real-world entities and their interactions.

2. Code Reusability

❖ Inheritance and encapsulation facilitate code reuse, leading to more maintainable and modular codebases.

3. Graphical User Interface (GUI) Development

❖ OOP languages are commonly used for building GUI applications, as they provide a natural way to model UI components.

OBJECT ORIENTED LANGUAGES

4. Large Scale Software Development

❖ OOP promotes modular and scalable design, making it suitable for developing complex and large-scale software systems.

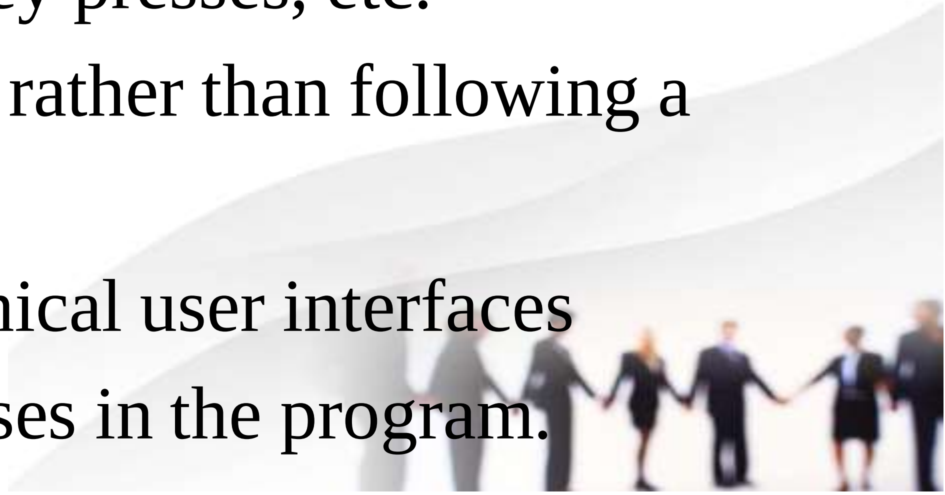
5. Game Development

❖ OOP is often used in game development to model various game objects, characters and their interactions.

□

EVENT DRIVEN PROGRAMMING

- ❖ Event driven programming is a programming paradigm in which the flow of the program is determined by the events.
- ❖ The events can be user actions, sensor outputs, or messages from other programs or threads.
- ❖ The user actions are generally initiated by the users.
- ❖ Some of the examples include mouse clicks, key presses, etc.
- ❖ The program responds to events as they occur, rather than following a predetermined sequence of steps.
- ❖ This paradigm is particularly common in graphical user interfaces (GUIs), where user interactions trigger responses in the program.



EVENT DRIVEN PROGRAMMING

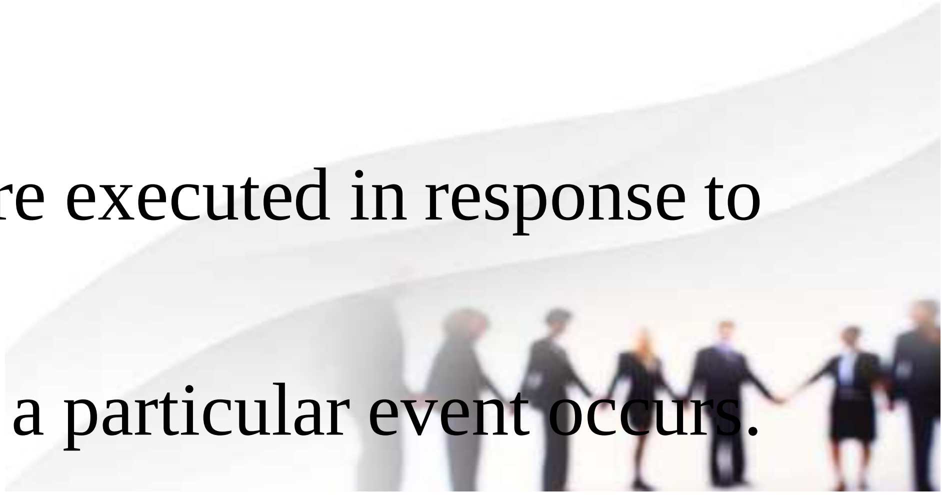
- ❖ **Key Concepts of Event Driven Programming**

1. Events

- ❖ Events are occurrences that happen during the execution of a program, often triggered by user actions or changes in the environment.
- ❖ Examples include mouse clicks, keyboard inputs, timer expirations and network activity.

2. Event Handlers

- ❖ Event handlers are functions or methods that are executed in response to specific events.
- ❖ They define how a program should react when a particular event occurs.



EVENT DRIVEN PROGRAMMING

3. Event Loop

❖ The event loop is a fundamental part of event driven programming.

❖ It continually checks for the occurrence of events and dispatches them to their corresponding event handlers.

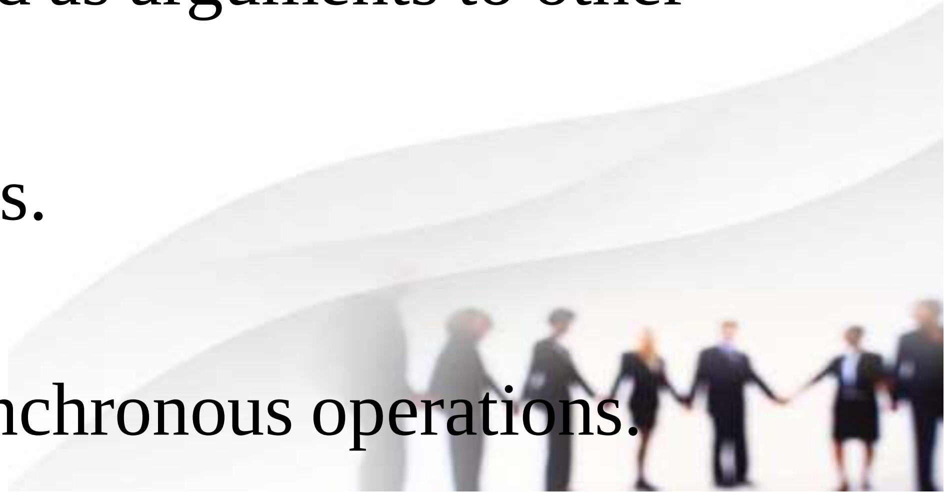
4. Callback Functions

❖ Callback functions are functions that are passed as arguments to other functions or methods.

❖ They are executed when a specific event occurs.

5. Asynchronous Programming

❖ Event-driven programming often involves asynchronous operations.



EVENT DRIVEN PROGRAMMING

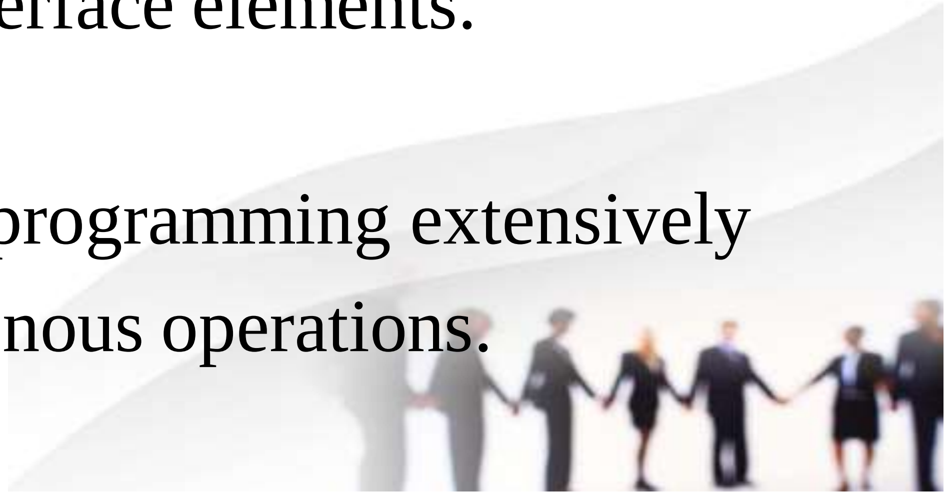
- ❖ Instead of waiting for a task to complete, the program continues its execution, and a callback is invoked once the task is finished.
- ❖ **Use Cases of Event Driven Programming**

1. Graphical User Interfaces (GUIs)

- ❖ Most GUI frameworks use event-driven programming to handle user interactions with buttons, menus, and other interface elements.

2. Web Development

- ❖ JavaScript in web browsers uses event-driven programming extensively to respond to user actions and handle asynchronous operations.



EVENT DRIVEN PROGRAMMING

3. Networking

❖ Event-driven programming is often used in networking applications to handle events such as incoming data or connection status changes.

4. Embedded Systems

❖ In embedded systems, event-driven programming can be employed to respond to sensor inputs or external triggers.

5. Real-time Systems

❖ Systems that require real-time responsiveness, such as simulations or games, often use event-driven architectures.

EVENT DRIVEN PROGRAMMING

❑ **Advantages of Event-Driven Programming:**

1. Responsive User Interfaces

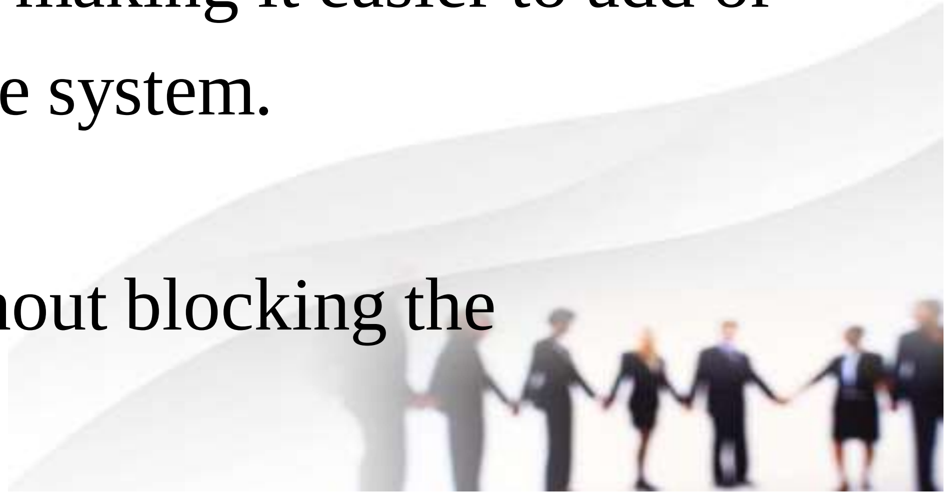
❖ Well-suited for building responsive and interactive user interfaces that react to user actions.

2.Modularity and Extensibility

❖ Event-driven architectures support modularity, making it easier to add or modify components without affecting the entire system.

3.Asynchronous Operations

❖ Effective for handling asynchronous tasks without blocking the program's execution.



EVENT DRIVEN PROGRAMMING

4. Event-Driven Systems Scale Well

❖ Event-driven systems can efficiently scale to handle a large number of events and interactions.

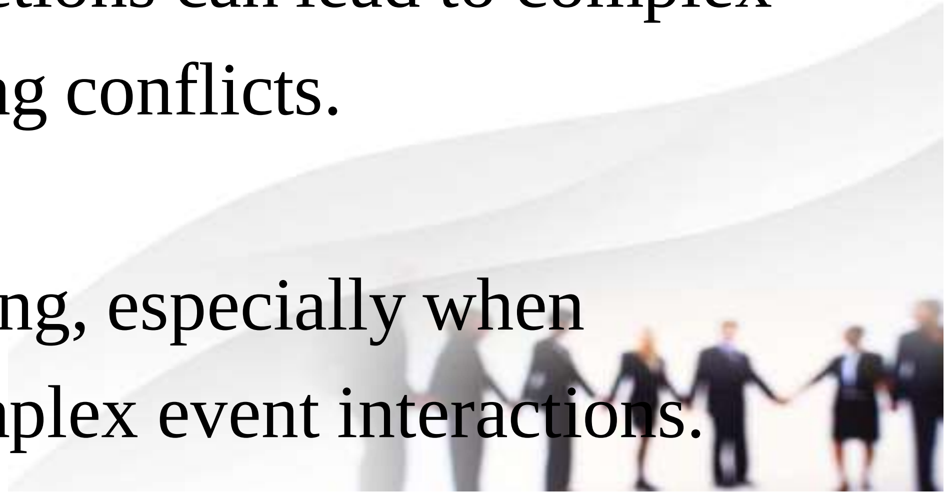
❖ **Challenges of Event-Driven Programming**

1. Complexity

❖ Managing a large number of events and interactions can lead to complex code and potential issues such as event handling conflicts.

2. Debugging

❖ Debugging event-driven code can be challenging, especially when dealing with asynchronous operations and complex event interactions.



EVENT DRIVEN PROGRAMMING

3. Order of Event Execution

❖ Understanding the order in which events are executed is crucial for developing predictable and reliable systems.

Unit III: Functional Programming and Logic Programming

Overview of Functional Programming:

- Definition of a function
- Domain and Range
- Total and Partial Functions
- Strict Functions
- Recursion
- Referential Transparency

Functional Programming -

Function: A mapping from a set of inputs (domain) to a set of outputs (range).

- **Domain:** The set of all possible inputs.
- **Range:** The set of all possible outputs.

Examples:

- Mathematical: f(x) = x^2
- Programming: def square(x): return x * x

Definition of a Function

- In functional programming, a function is a fundamental building block, similar to its mathematical counterpart.
- It maps one or more input values (the domain) to a single output value (the range).
- **Domain**: The set of all possible input values that a function can accept.
- **Range:** The set of all possible output values that a function can produce.
- **For example**, the function $f(x) = x + 2$ has a domain of all real numbers and a range of all real numbers.

Total and Partial Functions

Total Function: A function that is defined for every input value in its domain.

- Defined for all elements in the domain.
- Example: f(x) = x + 2 for all x in ℤ.

Partial Function: A function that is not defined for some input values in its domain.

- Defined only for a subset of the domain.
- Example: f(x) = 1/x (undefined for x = 0).

Strict Functions

- A function is strict if it evaluates its argument before execution.
- A strict function is a function that evaluates all of its arguments before returning a result. In contrast, a non-strict function (also known as a lazy function) may not evaluate all of its arguments if the result can be determined without them.
- Example:
- Strict: f(x) = x + 1 (needs x before proceeding).

- Non-strict: Lazy evaluation, common in languages like Haskell.

Recursion

Recursion: A function that calls itself to solve smaller sub-problems.

Recursion is a powerful technique in functional programming where a function calls itself to solve a smaller version of the same problem. This allows for elegant and concise solutions to problems that can be broken down into smaller, self-similar subproblems.

Components:

- Base Case
- Recursive Case

Recursion

- **Base Condition**

The base condition is the condition that is used to terminate the recursion. The recursive function will keep calling itself till the base condition is satisfied.

- **Recursive Case**

Recursive case is the way in which the recursive call is present in the function. Recursive case can contain multiple recursive calls, or different parameters such that at the end, the base condition is satisfied and the recursion is terminated.

Syntax Structure of Recursion

- return_type recursive_func {
-     ....
-         // Base Condition
-         // Recursive Case
-         ....
- }

Recursion

Example: Factorial

```
• def factorial(n):
•     if n == 0:
•         return 1
•     else:
•         return n * factorial(n - 1)
```

Types of Recursion in C++

- There are two different types of recursion which are as follows:

1. Direct Recursion
2. Indirect Recursion

In direct recursion, the function contains one or more recursive calls to itself. The function directly calls itself in the direct recursion and there is no intermediate function.

In indirect recursion, the function does not call itself directly but instead, it calls another function which then eventually calls the first function creating a cycle of function calls.

Applications of Recursion

- **Solving:** Fibonacci sequences, Factorial Function, Reversing an array, Tower of Hanoi.
- **Backtracking**: It is a technique for solving problems by trying out different solutions and undoing them if they do not work. Recursive algorithms are often used in backtracking.
- **Searching and Sorting Algorithms:** Many searching and sorting algorithms, such as binary search and quicksort, use recursion to divide the problem into smaller sub-problems.
- **Tree and Graph Traversal:** Recursive algorithms are often used to traverse trees and graphs, such as depth-first search and breadth-first search.
- **Mathematical Computations:** Recursion is also used in many mathematical computations, such as the factorial function and the Fibonacci sequence.
- **Dynamic Programming:** It is a technique for solving optimization problems by breaking them down into smaller sub-problems. Recursive algorithms are often used in dynamic programming.

Drawbacks of Recursion

- **Performance:** Recursive algorithms can be less efficient than iterative algorithms in some cases, particularly if the data structure is large or if the recursion goes too deep.
- **Memory usage:** Recursive algorithms can use a lot of memory, particularly if the recursion goes too deep or if the data structure is large. Each recursive call creates a new stack frame on the call stack, which can quickly add up to a significant amount of memory usage.
- **Code complexity:** Recursive algorithms can be more complex than iterative algorithms.
- Debugging: Recursive algorithms can be more difficult to debug than iterative algorithms, particularly if the recursion goes too deep or if the program is using multiple recursive calls.
- **Stack Overflow:** If the recursion goes too deep, it can cause a stack overflow error, which can crash the program.

Referential transparency

- Referential transparency is a property of expressions that guarantees that an expression can be replaced with its value without changing the meaning of the program. This is a key property that makes functional programs easier to reason about and optimize.
- For example, in the expression 2 + 2, the subexpression 2 + 2 can be replaced with its value 4 without changing the meaning of the expression.

Referential Transparency

- A function is referentially transparent if it produces the same output for the same input every time.
- Ensures:
- No side effects
- Easy reasoning about code
- Example:

```
def add(x, y):
return x + y  # Always produces the same output for the same inputs
```

Overview of Logic Programming:

- **Basic Constructs**

1. Facts, Rules, Queries
2. Goals and Predicates
3. Variables
4. Existential and Conjunctive Queries

- **Definition and Semantics of Logic Programs**

- **Recursive Programming**

- **Computational Model of Logic Programming**

Logic Programming - Introduction

- Logic Programming: A paradigm based on formal logic.
- Programs consist of:

1. Facts: Basic assertions about objects and relationships.
2. Rules: Logical implications.
3. Queries: Questions about the facts and rules.

Example:

```
parent(john, mary).
parent(mary, alice).
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
```

Logic Programming-Core Concepts

- **Declarative Paradigm:** Logic programming focuses on what needs to be computed rather than how to compute it. You describe the relationships and facts, and the system figures out the solution.
- **Knowledge Representation:** Logic programs represent knowledge as a collection of logical statements:
- **Facts:** Simple statements representing basic truths.

Example: father(john, david).

- **Rules:** Express relationships between facts.

Example: ancestor(X, Y) :- parent(X, Y). (X is an ancestor of Y if X is a parent of Y)

- **Inference:** The system uses logical inference (like deduction) to derive new facts from the existing knowledge base.
- **Queries:** You ask questions to the program, and it uses inference to find the answers.
- Example: ancestor(john, ?X). (Find all X such that John is an ancestor of X)

Basic Constructs in Logic Programming

- **Facts:**
- Atomic statements.
- Example: likes(john, pizza).
- **Rules:**
- Logical relationships.
- Example: loves(X, Y) :- likes(X, Y), kind(Y).
- **Queries:**
- Questions asked to the system.
- Example: ?- likes(john, X).

Key Components of Logic Programming

1) Predicates: Represent relationships between objects.

Example: father(X, Y), parent(X, Y), likes(Person, Food)

2)Variables: Represent unknown values.

- Often denoted by uppercase letters (e.g., X, Y, Z)

3)Unification: The process of finding substitutions for variables to make two terms equal.

4)Resolution: A fundamental inference rule used to derive new facts.

Example of Logic Programming

- Let's say we have the following facts and rules:
- parent(john, mary).
- parent(mary, david).
- ancestor(X, Y) :- parent(X, Y).
- ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
- The query ancestor(john, ?X). would result in the answer X = mary and X = david.

Applications of Logic Programming

1. Artificial Intelligence:

Expert systems

Natural language processing

Planning and robotics

2. Databases:

Declarative query languages (like Datalog)

3. Software Engineering:

1. Formal verification
2. Software specification

Popular Logic Programming Language: Prolog

- Prolog is a widely used logic programming language that exemplifies these concepts.

Variables and Queries

Variables:

- Represent unknown values.
- Start with uppercase letters (e.g., X, Y).

Existential Queries:

- Example: ?- parent(X, mary). (Finds any X such that X is Mary’s parent).

Conjunctive Queries:

- Example: ?- parent(X, Y), parent(Y, Z). (Finds grandparents).

Semantics of Logic Programs

- **Definition:** Logical programs describe relationships using facts and rules.
- **Execution:**
- Query resolution via unification and backtracking.
- Example:

```
likes(john, pizza).
likes(mary, pizza).
?- likes(X, pizza).  % Resolves X = john, mary.
```

Recursive Programming in Logic

Recursion in Logic Programming:

- Logical rules can be recursive.
- Example:

```
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

Computational Model:

Uses resolution and unification to deduce results.

Functional Programming	Logical Programming
It is totally based on functions.	It is totally based on formal logic.
In this programming paradigm, programs are constructed by applying and composing functions.	In this programming paradigm, program statements usually express or represent facts and rules related to problems within a system of formal logic.
These are specially designed to manage and handle symbolic computation and list processing applications.	These are specially designed for fault diagnosis, natural language processing, planning, and machine learning.
Its main aim is to reduce side effects that are accomplished by isolating them from the rest of the software code.	Its main aim is to allow machines to reason because it is very useful for representing knowledge.
Some languages used in functional programming include Clojure, Wolfram Language, Erland, OCaml, etc.	Some languages used for logic programming include Absys, Cycl, Alice, ALF (Algebraic logic functional programming language), etc.
It reduces code redundancy, improves modularity, solves complex problems, increases maintainability, etc.	It is data-driven, array-oriented, used to express knowledge, etc.
It usually supports the functional programming paradigm.	It usually supports the logic programming paradigm.
Testing is much easier as compared to logical programming.	Testing is comparatively more difficult as compared to functional programming.
It simply uses functions.	It simply uses predicates. Here, the predicate is not a function i.e., it does not have a return value.

Conclusion

Functional Programming:

- Emphasizes immutability, pure functions, and recursion.

Logic Programming:

- Centers around facts, rules, and queries.
- Recursive definitions enable complex relationships.

References

- Programming in Haskell - Graham Hutton

- Learn Prolog Now! - Patrick Blackburn, Johan Bos, and Kristina Striegnitz

- Foundations of Logic Programming - J. W. Lloyd

Unit IV
Object Oriented Programming

Object Oriented Programming Concept

- Object-Oriented Programming or OOPs refers to languages that use objects in programming.

- Object-oriented programming aims to implement real-world entities like data hiding, inheritance, polymorphism, etc. in programming.

- The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

Need for object-oriented programming

- To make the development and maintenance of projects more effortless.
- To provide the feature of data hiding that is good for security concerns.
- We can solve real-world problems if we are using object-oriented programming.
- It ensures code reusability.
- It lets us write generic code which will work with a range of data, so we don’t have to write basic stuff over and over again.

Benefits of object-oriented programming

- Code Reusability
- Improved Scalability
- Enhanced Maintainability

Features of OOP

- Class
- Object
- Inheritance
- Polymorphism
- Dynamic Binding
- Abstraction

Basic Constructs in OOP

- **Classes:**

- Blueprint for creating objects

- Contains attributes and methods

- **Objects:**

- Instances of classes

- Represent real-world entities

Basic Constructs in OOP

- **Methods:**

Define behaviors of objects

Types: Instance methods, Static methods, and Class methods

- **Attributes:**

Define object properties

Types: Instance attributes and Class attributes

Class

- A class is a user-defined data type. A class is like a blueprint for object.
- It consists of data members and member functions, which can be accessed and used by creating an instance of that class.
- It represents the set of properties or methods that are common to all objects of one type.
- For Example: Consider the Class of Cars with properties such as the no. of wheels, speed limit, mileage, range, etc.
- So here car is the class while the no. of wheels, speed limit, mileage are their properties.

Object

- It is a basic unit of OOP and represents the real-life entities.
- An Object is an instance of a Class.
- When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.
- An object has an identity, state, and behavior.
- Each object contains data and code to manipulate the data.
- Objects can interact without having to know details of each other’s data or code.
- It is sufficient to know the type of message accepted and type of response returned by the objects.
- E.g. “Dog” is a real-life Object, which has some characteristics like color, Breed, Bark, Sleep, and Eats.

Data Abstraction

- Data abstraction is one of the most essential and important features of object-oriented programming.
- Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.
- Consider a real-life example of a man driving a car.
- The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car, but he does not know about how on pressing the accelerator the speed is increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes in the car.
- This is what abstraction is.

Encapsulation

- Encapsulation is defined as the wrapping up of data under a single unit named Class
- It is the mechanism that binds together code and the data it manipulates.
- In encapsulation, the variables or data of a class are hidden from any other class and can be accessed only through any member function of their class in which they are declared.
- As in encapsulation, the data in a class is hidden from other classes, so it is also known as data-hiding.
- This concept helps in protecting the internal state of an object and hiding the implementation details from the outside world.

Inheritance

- Inheritance is an important pillar of Object-Oriented Programming.
- The capability of a class to derive properties and characteristics from another class is called Inheritance.
- When we write a class, we inherit properties from other classes.
- So when we create a class, we do not need to write all the properties and functions again and again, as these can be inherited from another class that possesses it.
- Inheritance allows the user to reuse the code whenever possible and reduce its redundancy.

Example of Inheritance

```
class Animal
{
    public:
        void eat()
        {
            cout << "Eating..." << endl;
        }
};
class Dog : public Animal
{
    public:
        void bark()
        {
            cout << "Barking..." << endl;
        }
};
```

Polymorphism

- The word polymorphism means having many forms.
- In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.
- For example, A person at the same time can have different characteristics.
- Like a man at the same time is a father, a husband, an employee.
- So the same person posses different behavior in different situations.
- This is called polymorphism.

Example of Polymorphism

```
class Shape
{
        public:
                virtual void draw()
                {
                                cout << "Drawing shape..." << endl;
                }
};
class Circle : public Shape
{
public:
        void draw()
        {
                cout << "Drawing Circle..." << endl;
      }
};
```

Types of Polymorphism

• Polymorphism is a core concept of Object-Oriented Programming (OOP) that allows functions or methods to have multiple forms.

1. Compile-time (Static) Polymorphism

- Function Overloading
- Operator Overloading
- Templates (Parameterized Polymorphism)

2. Run-time (Dynamic) Polymorphism

- Function Overriding (via Virtual Functions)

I. Compile-time (Static) Polymorphism

(a) Function Overloading

- Function overloading allows multiple functions with the same name but different parameters.

- **Syntax:**

```
class ClassName
{
    public:
        void functionName();                    // No parameters
        void functionName(int x);               // One parameter
        void functionName(double x);            // Different parameter type
};
```

Example of Function Overloading

```
#include <iostream.h>
using namespace std;
class Math
{
        public:
                void add(int a, int b)
                {
                    cout << "Sum (int): " << a + b << endl;
                }
                void add(double a, double b)
                {
                   cout << "Sum (double): " << a + b << endl;
                }
                void add(int a, int b, int c)
                {
                        cout << "Sum (3 ints):" << a + b + c << endl;
                }
};
int main()
{
    Math obj;
    obj.add(5, 10);             // Calls first function
    obj.add(2.5, 3.5);          // Calls second function
    obj.add(1, 2, 3);           // Calls third function
    return 0;
    getch();
}
```

(b) Operator Overloading

- Operator overloading allows custom behavior for C++ operators when used with user-defined data types.
- Any operator has its own meaning.
- The feature of operator overloading enables the programmer to provide a new meaning to the operator.
- **Syntax:**

```
class ClassName
{
    public:
            ReturnType operator symbol(Arguments);
};
```

II. Run-time (Dynamic) Polymorphism

(a) Function Overriding

- Function overriding allows a derived class to redefine a function inherited from the base class.
- **Syntax:**

```
class Base {
public:
    virtual void functionName(); // Virtual function in base class
};
class Derived : public Base {
public:
    void functionName() override; // Overridden function in derived class
};
```

(b) Virtual Functions and Base Class Pointers

Example:

```
class Base
{
public:
    virtual void show() { // Virtual function
        cout << "Base class function" << endl;
    }
};
class Derived : public Base
{
public:
    void show() override { // Overriding function
        cout << "Derived class function" << endl;
    }
};
```

```
int main()
{
        Base* ptr;
        Derived obj;
        ptr = &obj;
        ptr->show();
        return 0;
}
```

Conclusion

Type	Method	Description
Compile-time	Function Overloading	Multiple functions with the same name but different parameters
Compile-time	Operator Overloading	Custom behavior for operators (+, *, ==)
Run-time	Function Overriding	Redefining a function in the derived class
Run-time	Virtual Functions	Enables dynamic binding with base class pointers

Dynamic Binding

- Dynamic binding (also called **late binding** or **runtime polymorphism**) occurs when the function call is resolved at runtime rather than at compile time.
- Typically achieved using **virtual functions** and **base class pointers or references**.
- In dynamic binding, the code to be executed in response to the function call is decided at runtime.
- Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run time.
- Dynamic Method Binding One of the main advantages of inheritance is that some derived class D has all the members of its base class B.
- Once D is not hiding any of the public members of B, then an object of D can represent B in any context where a B could be used.
- This feature is known as subtype polymorphism.

Example of Dynamic Binding

```
class Base {
public:
     virtual void show() {
        cout << "Base class function" << endl;
    }
};
class Derived : public Base
{
public:
    void show()
{
        cout << "Derived class function" << endl;
    }
};
```

```
int main()
{
    Base* ptr;
    Derived obj;
    ptr = &obj;
    ptr->show();
    return 0;
}
```

Key Takeaways

1. **Virtual function:** Declaring a function as Virtual in the base class ensures dynamic Binding.

2. **Base Class Pointers/References**: Used to call overridden functions in derived classes.

3. **Function Overriding**: The derived class provides a new implementation of a function that exists in the base class.

4. **Late Binding**: The function call is resolved at runtime instead of compile time.

Interface in C++

- In C++, the concept of an interface is implemented using abstract classes that contain pure virtual functions.
- An abstract class serves as a blueprint.
- It specifies a set of methods that derived classes must implement, without providing the method implementations themselves.
- This approach ensures that any class inheriting from the abstract class adheres to a specific contract, promoting consistent behavior across different parts of a program.

Definiton of an Interface in C++

- To create an interface in C++, you can define an abstract class with pure virtual functions.
- A pure virtual function is declared by assigning 0 to the function declaration within the abstract class.
- Classes that inherit from this abstract class are required to provide concrete implementations for these pure virtual functions.
- Otherwise, they too become abstract and cannot be instantiated.

Definiton of an Interface in C++

```
#include <iostream>
class IShape
{
public:
        virtual void draw() const = 0;
};
class Circle : public IShape
{
public:
    void draw() const override
   {
         std::cout << "Drawing a circle." << std::endl;
    }
};
```

```
class Square : public IShape
{
public:
    void draw() const override
    {
        std::cout << "Drawing a square." << std::endl;
    }
};
int main() {
    IShape* shape1 = new Circle();
    IShape* shape2 = new Square();

   shape1->draw(); // Output: Drawing a circle.
    shape2->draw(); // Output: Drawing a square.

 delete shape1;
  delete shape2;
 return 0;
}
```

Benefits of Using Interfaces in C++

❑ Polymorphism

- Interfaces allow for the creation of flexible and reusable code, enabling different classes to be treated uniformly based on the shared interface.

❑ Decoupling

- By programming to an interface rather than a concrete implementation, code dependencies are reduced, enhancing modularity and maintainability.

❑ Multiple Inheritance

- C++ supports multiple inheritance, allowing a class to implement multiple interfaces, thereby combining different behaviors as needed.

Constructors

- A constructor is a member function of a class that has the same name as the class name.
- It helps to initialize the object of a class.
- It can either accept the arguments or not.
- It is used to allocate the memory to an object of the class.
- It is called whenever an instance of the class is created.
- It can be defined manually with arguments or without arguments.
- There can be many constructors in a class.
- It can be overloaded but it can not be inherited or virtual.
- There is a concept of copy constructor which is used to initialize an object from another object.

Constructors

- Though a constructor is a member function of the class, the syntax differs as compared to any other member function.
- The constructor usually do not have a return type as in case of a normal member function.
- The body of the constructor usually but not necessarily contains the initialization of the data members of the class.
- Syntax:

```
ClassName()
{
    //Constructor's Body
}
```

Destructors

- Like a constructor, destructor is also a member function of a class that has the same name as the class name preceded by a tilde(~) operator.
- It helps to deallocate the memory of an object.
- It is called while the object of the class is freed or deleted.
- In a class, there is always a single destructor without any parameters so it can’t be overloaded.
- It is always called in the reverse order of the constructor.
- If a class is inherited by another class and both the classes have a destructor then the destructor of the child class is called first, followed by the destructor of the parent or base class.

Destructors

- Though a destructor is quite similar to the constructor, the syntax differs a little as compared to a constructor.
- The destructor also do not have a return type as in case of a constructor.
- The syntax for the constructor can be given as follows:
- Syntax:

```
ClassName()
{
    //Constructor's Body
}
```

Example

```
#include <iostream>
using namespace std;
class Z
{
public:
     Z()
    {
        cout<<"Constructor called"<<endl;
    }
    ~Z()
    {
        cout<<"Destructor called"<<endl;
    }
};
```

Example

S. No.	Constructor	Destructor
1.	Constructor helps to initialize the object of a class.	Whereas destructor is used to destroy the instances.
2.	It is declared as className(arguments if any){Constructor’s Body }.	Whereas it is declared as ~ className(no arguments){ }.
3.	Constructor can either accept arguments or not.	While it can’t have any arguments.
4.	A constructor is called when an instance or object of a class is created.	It is called while object of the class is freed or deleted.
5.	Constructor is used to allocate the memory to an instance or object.	While it is used to deallocate the memory of an object of a class.
6.	Constructor can be overloaded.	While it can’t be overloaded.
7.	The constructor’s name is same as the class name.	Here, its name is also same as the class name preceded by the tiled (~) operator.
8.	In a class, there can be multiple constructors.	While in a class, there is always a single destructor.
9.	There is a concept of copy constructor which is used to initialize an object from another object.	While here, there is no copy destructor concept.
10.	They are often called in successive order.	They are often called in reverse order of constructor.

Unit No. 5

Overview of Programming Languages

Overview of Programming Languages: Ruby

- A dynamic,interpreted,object-oriented, general-purpose programming language.
- Designed for programmer happiness and productivity.
- Created by Yukihiro "Matz" Matsumoto in the mid-1990s.
- Known for its elegant syntax and focus on human readability.
- Used in web development, automation, and scripting.
- It is known for its simplicity, productivity, and readability.

Key Features of Ruby

- Dynamic Typing: Type checking is done at runtime, offering flexibility.
- Object-Oriented: Everything is an object in Ruby.
- Interpreted: Code is executed line by line, no compilation needed.
- Garbage Collection: Automatic memory management.
- Large Standard Library: Wide range of built-in tools and functionalities.
- Vibrant Community: Active and supportive community.
- Ruby on Rails: Powerful web development framework.

Setting up Ruby (Interpreter)

- How to run Ruby Code
- Installation: Download and install Ruby from ruby-lang.org.
- Interactive Ruby (irb): A command-line interpreter for experimenting with Ruby code. Great for learning and quick testing.
- Running Scripts: Save Ruby code in a .rb file and execute it from the command line: ruby my_script.rb
- Demo: Show a live demonstration of opening irb and running a simple command like puts "Hello, World!". Also show running a simple script.
- Image: Screenshots of installing Ruby and using irb.

Basic Syntax

- Comments: # for single-line comments, =begin and =end for multi-line comments.
- Variables: Dynamically typed, no explicit type declaration. variable_name = value.
- Data Types: Numbers (integers, floats), Strings, Booleans, Symbols, Arrays, Hashes.

Example Code:

- # This is a comment
- name = "Alice" # String
- age = 30   # Integer
- pi = 3.14159   # Float
- is_student = true # Boolean

Strings in Ruby

- Title: Working with Strings
- Content:
- String Literals: Single quotes (') or double quotes (")
- String Interpolation: Using #{} inside double quotes to embed variables.
- String Methods: length, upcase, downcase, strip, split, concat, etc.

Example Code:

- name = "Bob"
- puts "Hello, #{name}!" # String interpolation
- puts name.length   # Output: 3
- puts name.upcase   # Output: BOB
- puts " Ruby ".strip # Output: Ruby

String Manipulation

- Concatenation: Using + or <<.
- Substring Extraction: Using [].
- Regular Expressions: Powerful pattern matching.
- Example Code:

```
"Hello" + " " + "World"  # Output: Hello World
"Ruby"[0..2]             # Output: Rub
"Ruby".gsub("R", "J")    # Output: Juby (using regular expression)
```

Input and Output

1. Title: Talking to the User
2. Content:

- puts: Prints a string to the console followed by a newline.
- print: Prints a string to the console without a newline.
- gets: Reads a line from the console.
- chomp: Removes the trailing newline from a string read by gets.

3. Example Code:

```
print "Enter your name: "
name = gets.chomp
puts "Hello, #{name}!"
```

Control Flow (Brief Overview)

1. if, elsif, else: Conditional statements.
2. while, until: Loops.
3. for: Iterating over collections.

- Example Code (if statement):

```
age = 20
if age >= 18
  puts "You are an adult."
else
  puts "You are a minor."
end
```

Control Structures in Ruby

- Control structures manage the flow of execution.
- Types:

1. Conditional statements
2. Looping constructs

- Help execute code based on conditions and repetitions.

Conditional Statements

- **if-else Statement:**

```
x = 10
if x > 5
  puts "x is greater than 5"
else
  puts "x is 5 or less"
end
```

elsif Statement

```
age = 18
if age < 13
  puts "Child"
elsif age < 20
  puts "Teenager"
else
  puts "Adult"
end
```

Case Statement (Alternative to if-elsif-else)

```
fruit = "apple"
case fruit
when "banana"
  puts "It is a banana"
when "apple"
  puts "It is an apple"
else
  puts "Unknown fruit"
end
```

Loops in Ruby

- Used to execute a block of code multiple times.
- **While Loop:**

```
x = 1
while x <= 5
  puts x
  x += 1
end
```

Until Loop

- y = 1
- until y > 5
-   puts y
-   y += 1
- end

For Loop

- for i in 1..5 do
-   puts i
- end

Each Loop (Iterators)

- [1, 2, 3, 4, 5].each do |num|
-   puts num
- end

Duck Typing in Ruby

- Ruby is dynamically typed; type is determined at runtime.
- Duck Typing: "If it quacks like a duck, it's a duck."
- Focuses on behavior rather than explicit types.

```
def make_sound(animal)
  animal.sound
end

class Dog
  def sound
    puts "Woof!"
  end
end

class Cat
  def sound
    puts "Meow!"
  end
end

d = Dog.new
c = Cat.new
make_sound(d) # Woof!
make_sound(c) # Meow!
```

Arrays in Ruby

- Ordered collections of elements.
- Can store multiple types of data.
- Creating an Array
- arr = [1, "hello", 3.5, true]

Array Operations

- arr.push(10)   # Add element
- arr.pop   # Remove last element
- arr.shift   # Remove first element
- arr << "new"   # Append element
- arr.each { |x| puts x } # Iterate

Hashes in Ruby

- Key-value pairs, similar to dictionaries.
- **Creating a Hash**
- person = {"name" => "John", "age" => 25, "city" => "New York"}
- **Accessing Values**
- puts person["name"] # Output: John
- **Modern Hash Syntax**
- person = {name: "John", age: 25, city: "New York"}
- puts person[:name] # Output: John

Hash Operations

- person[:gender] = "Male" # Add new key-value pair
- person.delete(:age) # Remove key-value pair

Symbols in Ruby

- Lightweight, immutable string-like objects.
- Used as keys in hashes for efficiency.

Creating Symbols

- :my_symbol
- **Symbols vs. Strings**
- "hello".object_id != "hello".object_id # Different objects
- :hello.object_id == :hello.object_id # Same object

Introduction to Prolog

- A logic programming language.
- Based on formal logic and automated theorem proving.
- Used for AI, natural language processing, knowledge representation, etc.
- Programs consist of facts and rules.

Structures

- Title: Structures: Building Blocks of Knowledge
- Complex terms representing objects and their relationships.
- Constructed from functors (name) and arguments.
- Examples: point(10, 20), person(john, 30, city(london)), tree(node(left, a), node(right, b)).
- Example: date(2024, 10, 26) represents a date.
- Image: A diagram illustrating a structured term.

Matching Structures

- Unification: The Heart of Prolog
- The process of determining if two terms can be made identical by substituting variables.
- Used for pattern matching and rule application.
- X = Y attempts to unify X and Y.

Example:

- ?- point(X, Y) = point(10, 20). % X = 10, Y = 20
- ?- point(X, 20) = point(10, Y). % X = 10, Y = 20
- ?- point(X, X) = point(10, 20). % Fails (X cannot be both 10 and 20)

Equality and Comparison Operators

- Title: Comparing Terms
- Content:
- =: Unification (as discussed).
- ==: Identity (terms are identical without variable substitution).
- \==: Non-identity.
- @<, @>, @=<, @>=: Term comparison (alphabetical order).
- Example:

Example of Equality and Comparison Operators:

- ?- X = 10, X == 10. % Succeeds
- ?- 10 = 10.   % Succeeds
- ?- 10 == 10.   % Succeeds
- ?- a @< b.   % Succeeds

Arithmetic

- Title: Doing Math in Prolog
- is: Evaluates an arithmetic expression and unifies the result with a variable.
- +, -, *, /, mod: Arithmetic operators.
- Note: = is not for arithmetic evaluation; use is.
- Example:
- ?- X is 2 + 3.   % X = 5
- ?- Y is 10 mod 3.   % Y = 1
- ?- 5 = 2 + 3.   % Fails (unification, not evaluation)

Lists

- Title: Lists: Ordered Collections
- Content:
- Ordered collections of terms.
- Enclosed in square brackets: [ ].
- Can contain any type of term, even other lists.
- [a, b, c], [1, 2, 3], [a, [b, c], d].

List Notation

- Title: List Construction
- Content:
- []: Empty list.
- [Head | Tail]: Head is the first element, Tail is the rest of the list.
- [a, b, c] is equivalent to [a | [b | [c | []]]].
- Example:
- ?- [H | T] = [a, b, c]. % H = a, T = [b, c]
- ?- [a, b, c] = [a | [b, c]]. % Succeeds

Splitting Lists

- Title: Deconstructing Lists
- Content:
- Using unification and the [Head | Tail] notation to access list elements.
- Recursive predicates are often used for list processing.
- Example (member/2 - checks if an element is in a list):
- member(X, [X | _]).   % X is the head of the list
- member(X, [_ | Tail]) :- % X is in the tail of the list
-     member(X, Tail).

Enumerating Lists

- Title: Working with All List Elements
- Content:
- Recursive predicates are the primary way to process all elements in a list.
- Example (length/2 - calculates the length of a list):
- length([], 0).
- length([_ | Tail], N) :-
-     length(Tail, N1),
-     N is N1 + 1.

Example: append/3

- Title: Appending Lists
- Content:
- A classic example of list manipulation in Prolog.
- Concatenates two lists.
- Code:
- append([], L, L).
- append([H | T1], L2, [H | T3]) :-
-     append(T1, L2, T3).

Example Usage:

- ?- append([a, b], [c, d], L). % L = [a, b, c, d]

Conclusion

- Prolog: Powerful for Logical Reasoning
- Content:
- Structures and lists are fundamental data structures in Prolog.
- Unification is the key operation.
- Recursive predicates are essential for list processing.

UNIT VI

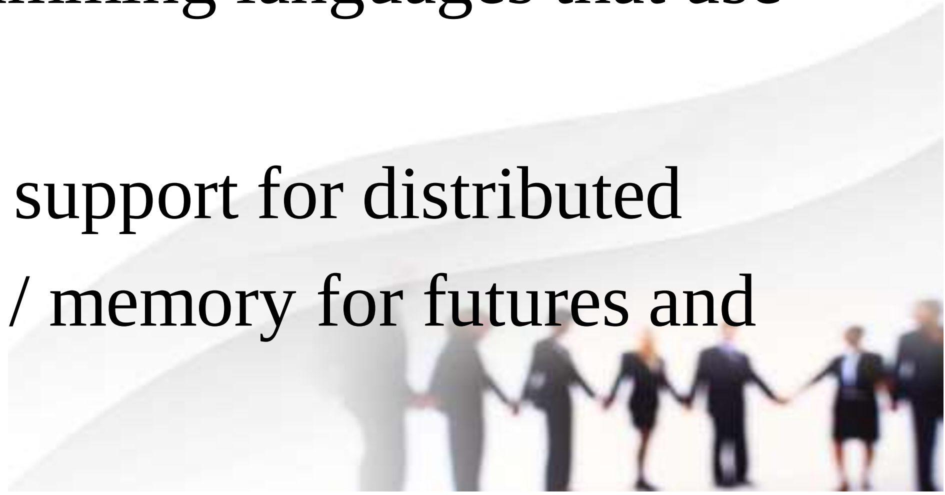
ADVANCED PROGRAMMING

CONCURRENT PROGRAMMING

- ❖ Advanced programming language definition associates in input/output variables, the domain and the computable function with each program.
- ❖ It is based on a computer automaton when inputs from In, states from St are viewed as instructions, memory states, respectively.
- ❖ **Concurrent programming**
- ❖ Concurrent programming is a technique in computer science where multiple tasks are executed simultaneously but not necessarily in parallel.
- ❖ It allows efficient management of resources and improved responsiveness within software applications.

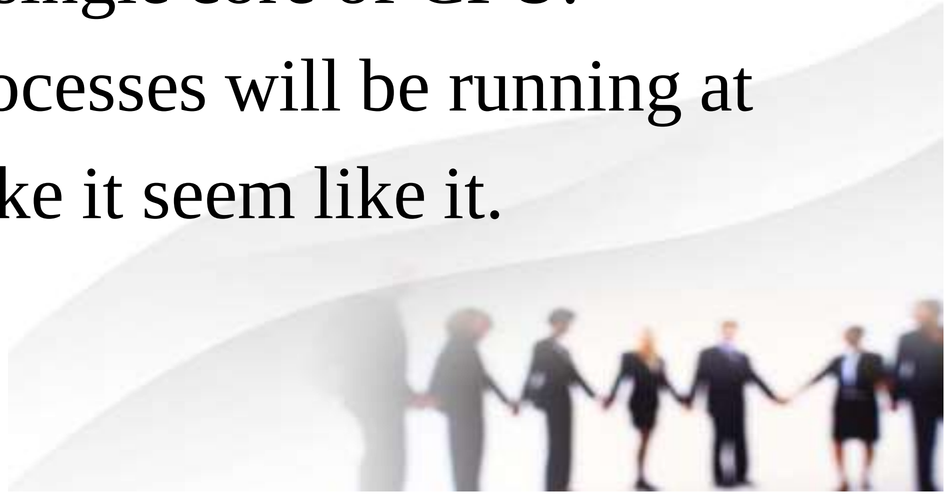
CONCURRENT PROGRAMMING

- ❖ Simply described, it's when you are doing more than one thing at the same time.
- ❖ It should not be confused with the parallelism, concurrent or concurrency programming when multiple sequences of operations run in overlapping periods of time.
- ❖ Concurrent programming languages are programming languages that use language constructs for concurrency.
- ❖ These constructs may involve multi-threading, support for distributed computing, message passing, shared resources / memory for futures and promises.



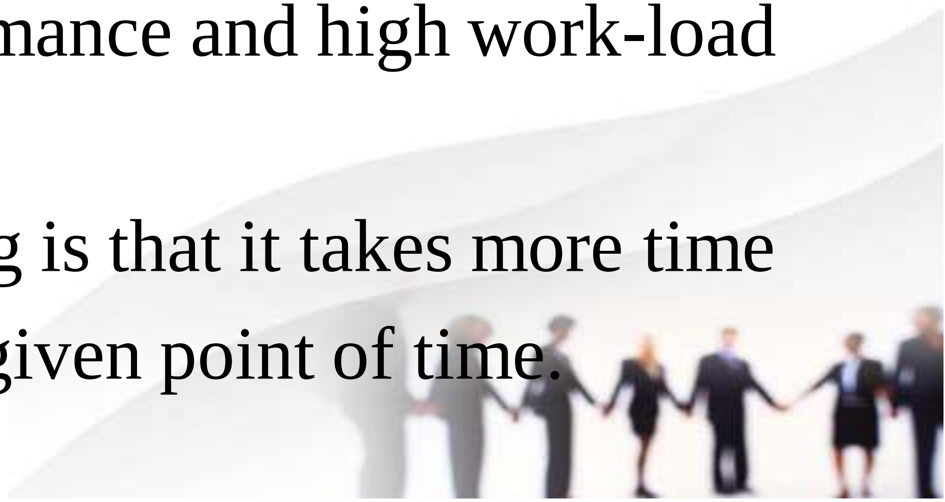
CONCURRENT PROGRAMMING

- ❖ Concurrency generally refers to the events or circumstances that are happening or existing at the same time.
- ❖ In programming terms, concurrent programming is a technique in which two/more processes start, run in an interleaved fashion through context switching and complete in an overlapping time period by managing access to shared resources, for example - on a single core of CPU.
- ❖ This doesn’t necessarily mean that multiple processes will be running at the same instant – even if the results might make it seem like it.



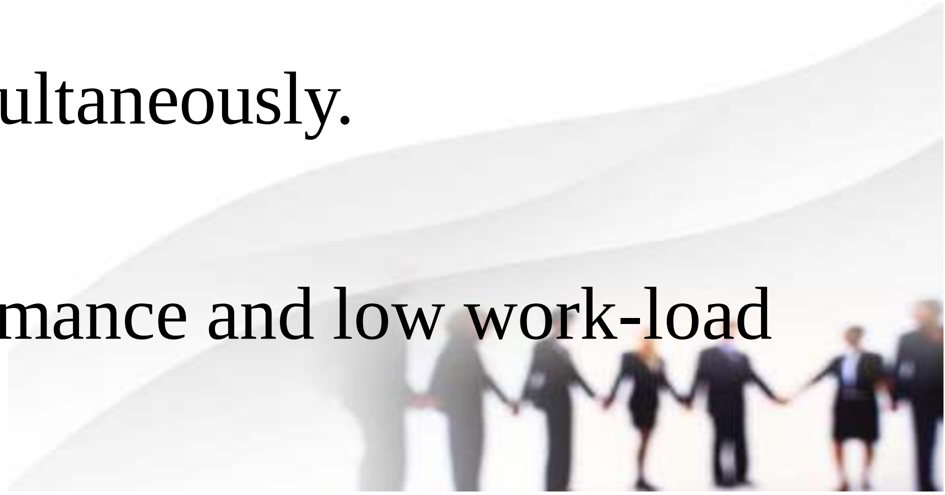
SERIAL VS. PARALLEL PROGRAMMING

- ❖ Sequential Computing is the type of computing where one instruction is given at a particular time and the next instruction has to wait for the first instruction to execute.
- ❖ It is also known as a traditional computing method because all the instructions are executed in a sequence.
- ❖ It is having a single processor with low performance and high work-load of the processor.
- ❖ The main disadvantage of using this computing is that it takes more time as a single instruction is getting executed at a given point of time.



SERIAL VS. PARALLEL PROGRAMMING

- ❖ To remove this disadvantage of sequential computing, parallel computing was introduced.
- ❖ Parallel Computing is a type of computing where many calculations or process executions are carried out parallelly or simultaneously.
- ❖ The type of computing where processes can execute simultaneously at a time is known as parallel computing.
- ❖ It saves time as the processes are executed simultaneously.
- ❖ It solves larger problems.
- ❖ There are multiple processors with high performance and low work-load per processor.



PROCESS COMMUNICATION

- ❖ Process communication is the exchange of information between processes in a system.
- ❖ It can also refer to the study of how communication behavior interacts with process scheduling behavior.

❑ **Essential steps in the communication process**

1. Idea formation

- ❖ Develop an idea to transmit.
- ❖ Every communication begins with initial sender.
- ❖ The sender develops an idea which needs to be transmitted.

PROCESS COMMUNICATION

2. Encoding

- ❖ Encode the message.
- ❖ The message is put into a specialized format for transmission or storage.

3. Channel selection

- ❖ Choose the right channel for delivery.
- ❖ The sender chooses the right channel to deliver the message.

4. Message delivery

- ❖ Deliver the message.
- ❖ The message is delivered to the receiver through a proper channel.

PROCESS COMMUNICATION

5. Message reception

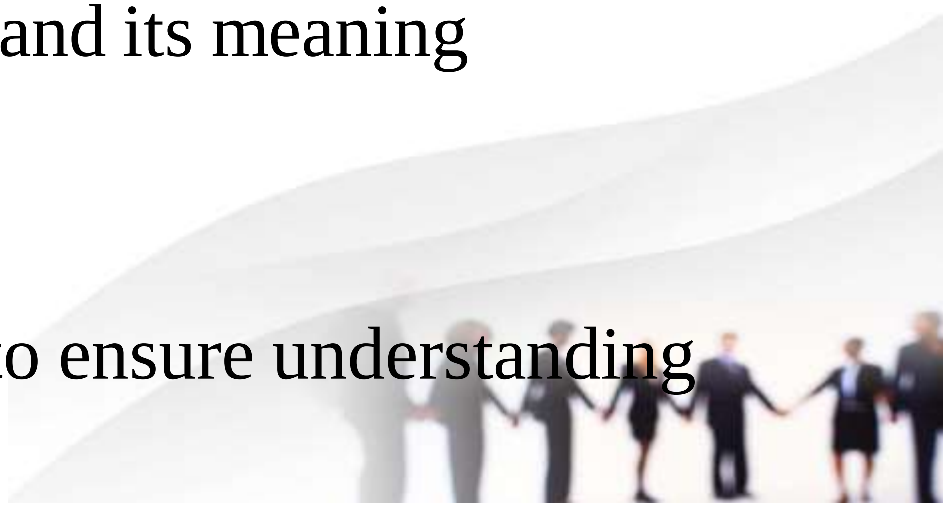
- ❖ The recipient receives the message.
- ❖ The recipient receives the message

6. Decoding

- ❖ The message is decoded.
- ❖ The recipient interprets the message to understand its meaning

7. Feedback

- ❖ The receiver provides feedback.
- ❖ The recipient provides feedback to the sender to ensure understanding

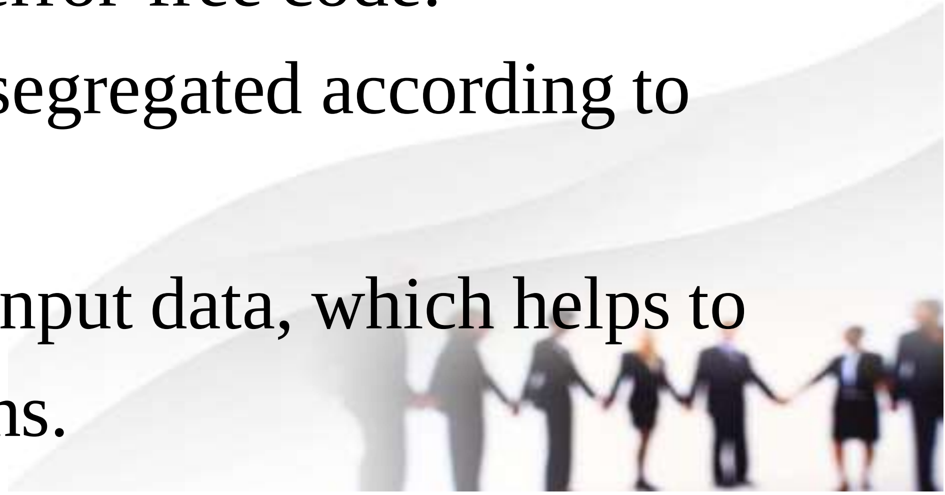


BASIC CONCEPTS OF PROGRAMMING

- ❖ The fundamental concepts of programming form the foundation upon which all the software development is built.
- ❖ These concepts are universal across various programming languages as well as different frameworks.
- ❖ Irrespective of the programming language we choose to learn these basic concepts for use.
- ❖ The basic concepts of programming are similar across the languages.
- ❖ Some of these concepts include atoms, variables and data types.
- ❖ Let us explain each of these one by one.

DATA TYPES

- ❖ Data types in programming are classifications that determine the type of value a variable can hold.
- ❖ Understanding data types is important for programming because it helps ensure that various operations are performed correctly and resources are used efficiently.
- ❖ It also helps programmers write effective and error-free code.
- ❖ When coding, variables need to be stored and segregated according to their data types.
- ❖ This informs the compiler about the expected input data, which helps to classify the type of data that the system contains.



DATA TYPES

❖ Some common data types can be explained as follows:

❑ **Integer**

❖ A numeric data type that stores numbers without a fractional component.

❖ Used to store all the numeric values in the field of programmig.

❑ **Float**

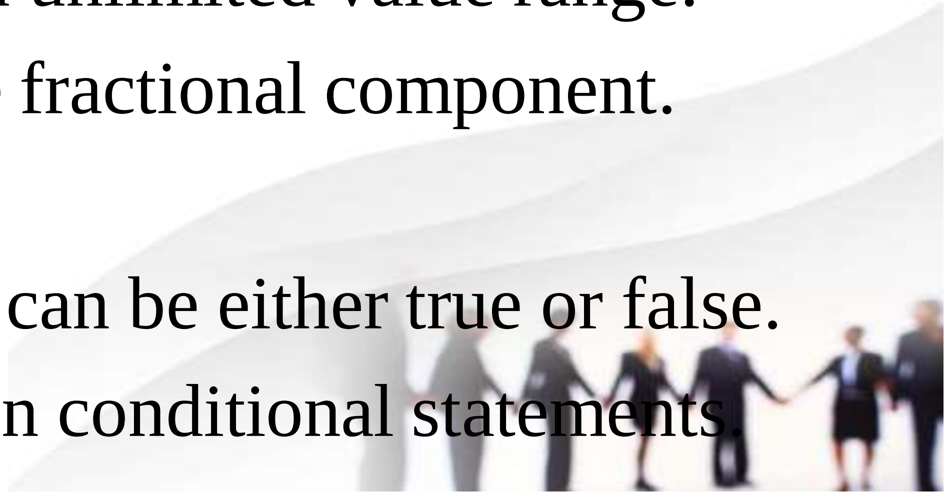
❖ A single-precision 32-bit floating point with an unlimited value range.

❖ They are used to store numbers with a variable fractional component.

❑ **Boolean**

❖ A data type that represents a logical entity that can be either true or false.

❖ Boolean values are often used to control flow in conditional statements.

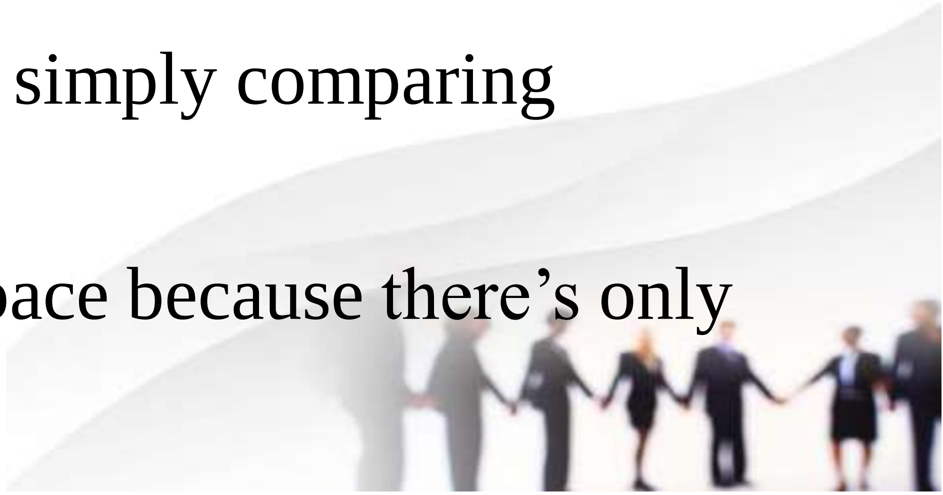


DATA TYPES

- **String**
- It usually contains a sequence of characters, digits, or symbols - always treated as text
- A data type that is often implemented as an array data structure of bytes or words.
- **Char**
- A data type that stores characters.
- The characters stored in a char variable have a value equivalent to their integer code
- This integer code is also known as an ASCII code.

ATOMS

- An atom is a pointer to a unique, immutable sequence of zero or more arbitrary bytes.
- Most atoms are pointers to null-terminated strings, but a pointer to any sequence of bytes can be an atom.
- There is single occurrence of any atom, which is why it's called an atom.
- Two atoms are identical if they point to the same location.
- Comparing two byte sequences for equality by simply comparing pointers is one of the advantages of atoms.
- Another advantage is that using atoms saves space because there’s only one occurrence of each sequence.



ATOMS

❖ In programming, atoms can refer to a few different things depending upon the application where it is used.

1) Atomic actions

❖ Actions that happen all at once, without stopping in the middle.

❖ They either happen completely or not at all.

❖ For example, reads and writes are atomic for reference variables and most primitive variables.

2) Atoms in C

❖ A pointer to a unique, immutable sequence of bytes.

❖ Atoms can save space, compare equality & act as keys in data structures.

ATOMS

3) Atoms in Closure

- ❖ A reference type that manages shared, synchronous, independent state.
- ❖ Atoms can be created using the keyword “atom”.
- ❖ They are accessed with the help of deref/@.

4) Atom, a programming language

- ❖ A concurrent programming language for embedded applications.
- ❖ Atom's concurrency model uses guarded atomic actions.
- ❖ These actions eliminate the need for mutex locks.
- ❖ Mutex or mutual exclusion referes to the

ATOMS

5) Atom, a text editor

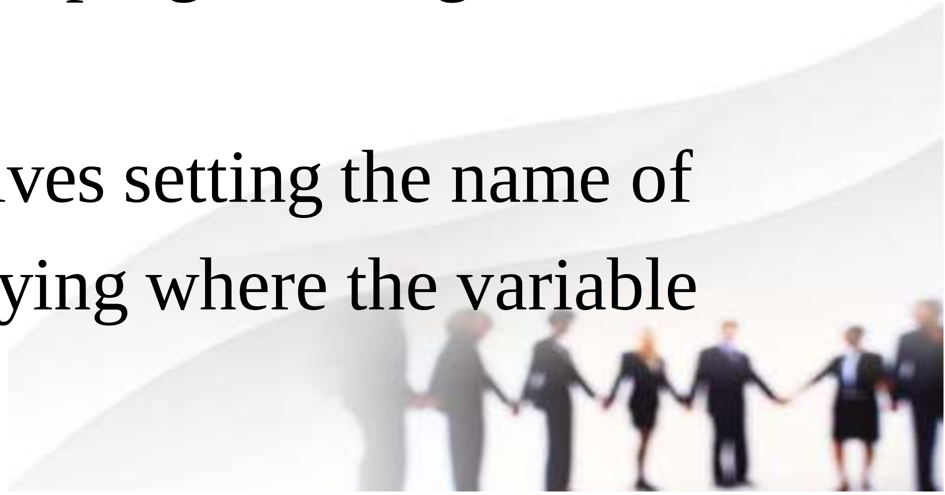
- ❖ A desktop application that can be customized with HTML, CSS and JavaScript.
- ❖ Atom uses tree structure to provide syntax highlighting for multiple programming languages and file formats.

6) Advanced Tools for Organization Management (ATOM)

- ❖ A school intervention program that uses digital tools to improve students' organizational and time management skills.

VARIABLES

- ❖ In programming, a variable is a named container for a set of data or object that can change based on the information or conditions passed to the program.
- ❖ The value of a variable is usually initialized to a default value, like "0", and the user supplies the actual values.
- ❖ Here are some things to know about variables in programming.
- ❑ **Variable declaration**
- ❖ The process of creating a variable, which involves setting the name of the variable and its data type along with specifying where the variable will be stored in memory.



VARIABLES

- ❑ **Examples of variables**
- ❖ Some common examples of variables include integers, floats, booleans, strings, lists and arrays.
- ❑ **Types of variables**

1. Local variables

- ❖ A local variable is declared inside a function or block of code and its scope is limited to that function or block.
- ❖ Local variables are found in each and every program.
- ❖ They are present in RAM only during the execution of the function where they are located.

VARIABLES

2. Global variables

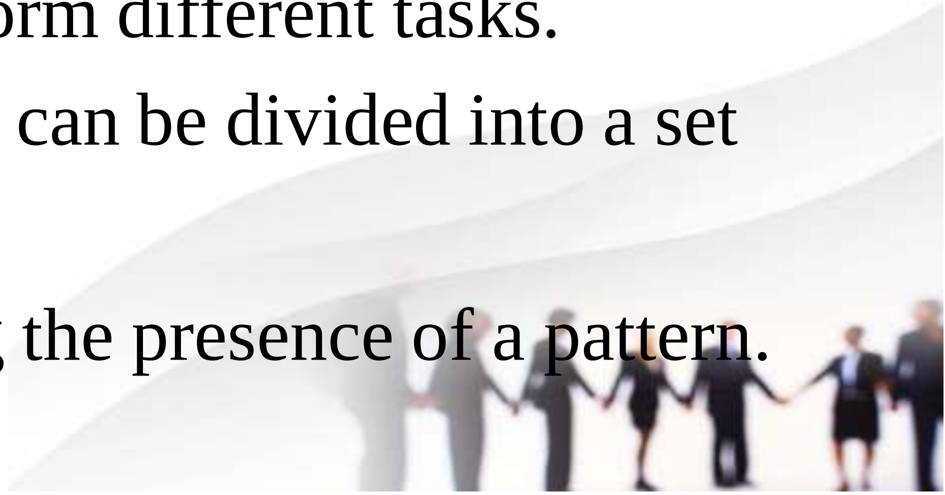
- ❖ Global variables are declared outside of any function or block and can be accessed by all routines in the program.
- ❖ They are always present in RAM during execution.

❑ **Reusing the variables**

- ❖ Variables can be reused multiple times whenever necessary.
- ❖ In general, it’s a bad idea to reuse/recycle variables.
- ❖ It makes code less readable, debugging more difficult and can lead to bad designs.

PATTERN MATCHING

- ❖ Pattern matching is a technique that involves testing an expression or value to determine if it has certain characteristics.
- ❖ Pattern matching is a fundamental approach for testing and validating code and data.
- ❖ It can improve the readability and correctness of the code.
- ❖ Patten matching technique can be used to perform different tasks.
- ❖ In case of patten matching, the given sequence can be divided into a set of tokens.
- ❖ These tokens prove to be helpful in identifying the presence of a pattern.

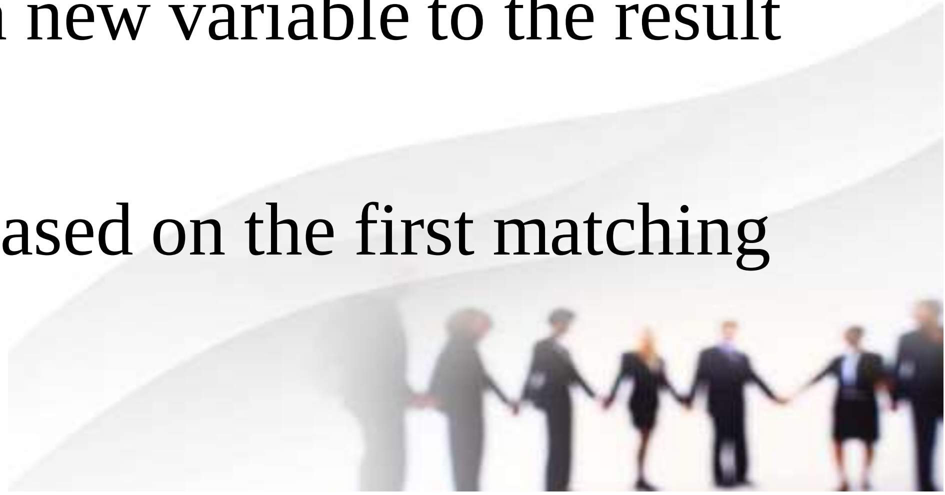


PATTERN MATCHING

- ❖ Pattern matching can be used for various tasks such as:
- ▪ **Find occurrences**
- ❖ Find all occurrences of a pattern in a text
- ▪ **Test structure**
- ❖ Test if an object has a particular structure
- ▪ **Extract data**
- ❖ Extract data from an object if there's a match
- ▪ **Deconstruct value**
- ❖ Deconstruct a value into its constituent parts

PATTERN MATCHING

- ❖ Pattern matching can be used in a variety of programming languages, including C#, Java and Python.
- ❖ Here are some examples of how pattern matching can be used.
- ❑ **C#**
- ❖ The "is expression" and "switch expression" support pattern matching.
- ❖ The "is expression" can conditionally declare a new variable to the result of an expression.
- ❖ The "switch expression" can perform actions based on the first matching pattern for an expression.



PATTERN MATCHING

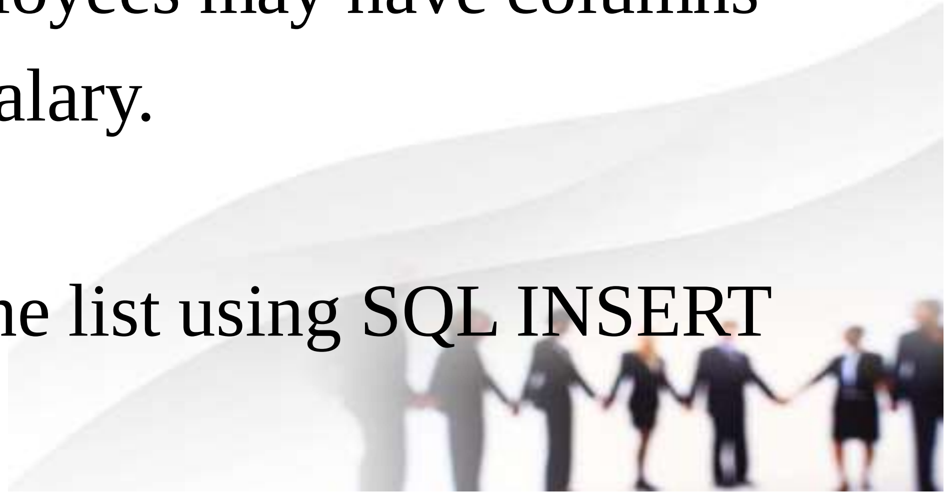
- **Java**
  - Pattern matching can be used to conditionally extract data from objects with more concise and robust code.
- **Python**
  - Pattern matching can be used as a switch statement to match an input against a number of possible cases.
  - Say for example, you can match an HTTP status code to print "okay" if it is 200, "not found" if it is 404, or "server error" if it is 500.

LISTS

- ❖ In database programming, lists are typically represented as sets of related records within database tables.
- ❖ These records are organized in rows and columns in the form of tables.
- ❖ Here each row represents an individual item or entity and each column represents a specific attribute or property of that item.
- ❖ Lists can be manipulated and queried using Structured Query Language (SQL) or through procedural extensions provided by the database management system (DBMS).
- ❖ There are various ways of using lists effectively in the field of database programming.

LISTS

- ❖ Here's how lists are commonly utilized in database programming:
- ❑ **Table Structure**
- ❖ Lists are often represented using tables with multiple rows.
- ❖ Each row in the table corresponds to an item in the list, and each column represents a different attribute or property of that item.
- ❖ For example, a table representing a list of employees may have columns such as employee_id, name, department, and salary.
- ❑ **Inserting Data**
- ❖ Data is inserted into the database to populate the list using SQL INSERT statements.



LISTS

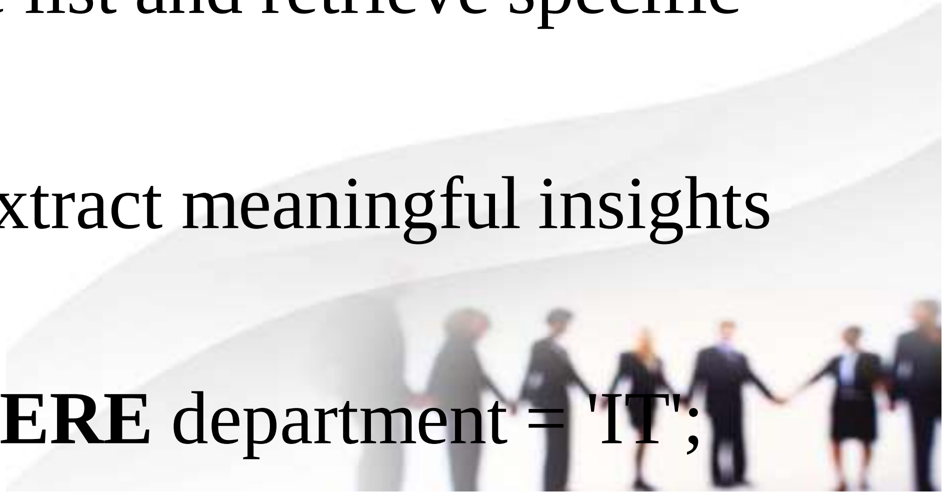
- ❖ Each INSERT statement adds a new row to the table, representing a new item in the list.

  INSERT INTO employees (employee_id, name, department, salary) **VALUES** (1, 'John Doe', 'IT', 60000);

❑ **Querying Data**

- ❖ SQL SELECT statements are used to query the list and retrieve specific information based on specified criteria.
- ❖ Queries can filter, sort, and aggregate data to extract meaningful insights from the list.

  SELECT * FROM employees **WHERE** department = 'IT';



LISTS

❑ **Updating Data**

❖ Data in the list can be updated using SQL UPDATE statements.

❖ These statements modify existing rows in the table to reflect changes in the data.

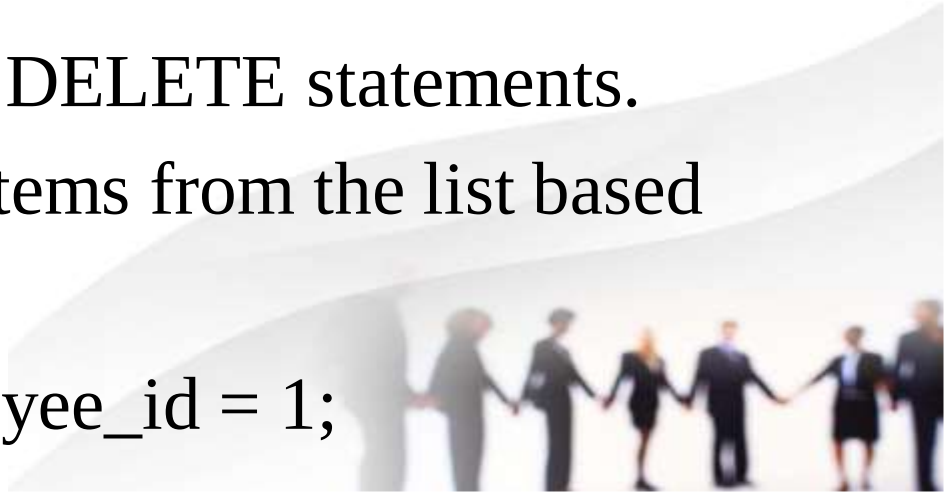
UPDATE employees **SET** salary = 65000 **WHERE** employee_id = 1;

❑ **Deleting Data**

❖ Rows can be removed from the list using SQL DELETE statements.

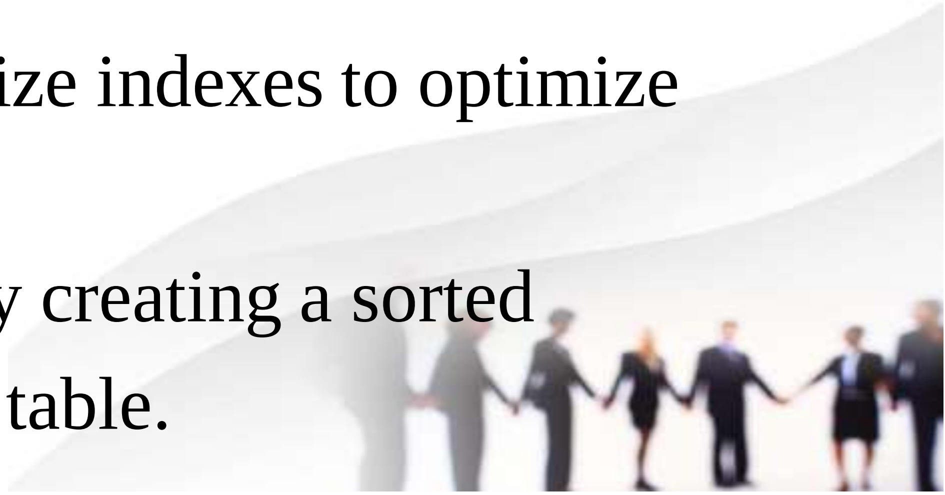
❖ These statements are used to remove specific items from the list based on the specified criteria.

❖ **DELETE FROM** employees **WHERE** employee_id = 1;



LISTS

- ❑ **Relational Operations**
- ❖ Lists in relational database management systems (RDBMS) can be related to each other through foreign key constraints.
- ❖ This allows for the creation of relationships between lists, enabling more complex data modeling and querying capabilities.
- ❑ **Indexes and Optimization**
- ❖ For large lists, database programmers may utilize indexes to optimize query performance.
- ❖ Indexes provide a faster way to retrieve data by creating a sorted structure based on one or more columns in the table.

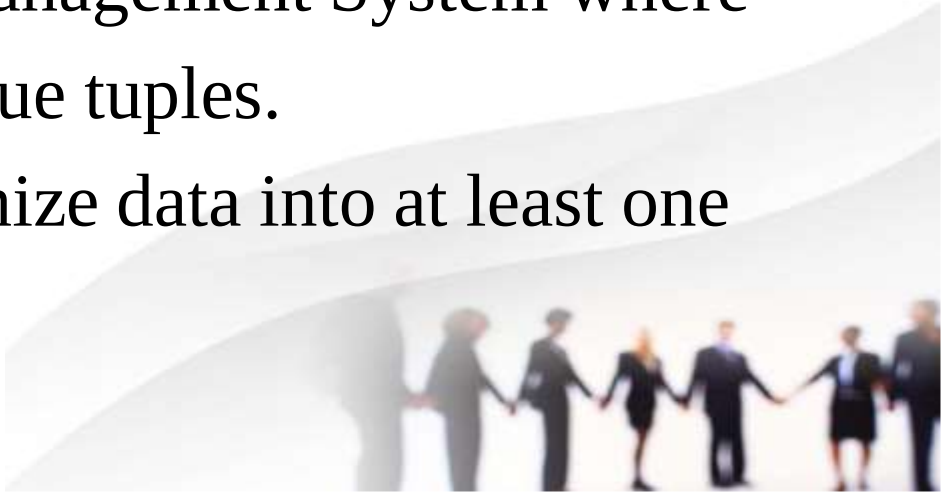


LISTS

- Overall, the lists in database programming are represented as the tables of related records.
- They are manipulated and queried using SQL statements or procedural code provided by the DBMS.
- They serve as a fundamental mechanism for storing and managing structured data in a database system.

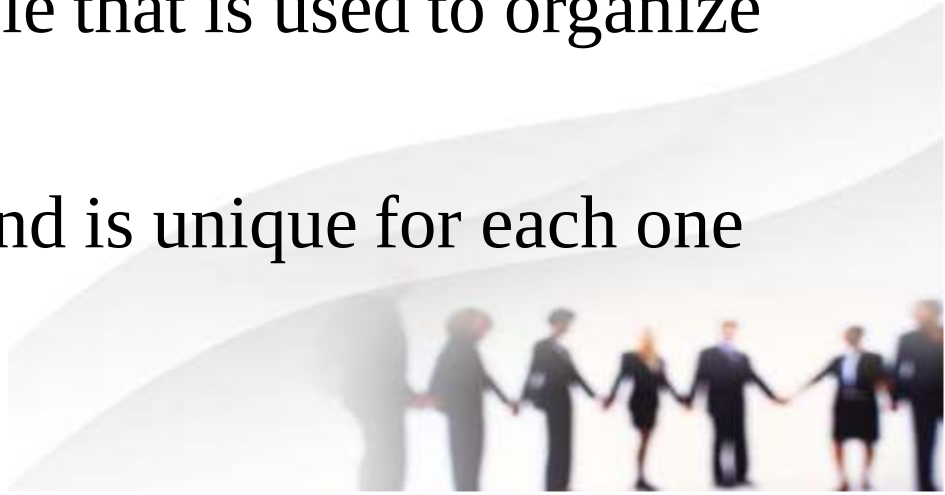
TUPLES

- A tuple, also known as a record or row, is a basic unit of data in a relational database management system (DBMS).
- A tuple represents a single instance of a relation, or table, in the database.
- Each tuple contains a set of values, or attributes, that correspond to the columns, or fields, of the relation.
- E.F. Codd invented the Relational Database Management System where he defined relationships as a collection of unique tuples.
- The relational model uses unique keys to organize data into at least one table of rows and columns.
- These rows can be depicted as Tables.



TUPLES

- ❖ A tuple in a database management system is one record in the context of relational databases (one row).
- ❖ You can compare the data present in the database with a spreadsheet, with rows (known as tuples) and columns (known as fields or attributes) representing various data types.
- ❖ In DBMS, a unique key is assigned to each table that is used to organize and identify the elements.
- ❖ This key is known as the table’s primary key and is unique for each one of the record present in the table.



TUPLES

- ❖ In DBMS, the user can add a column containing the value from another table’s column.
- ❖ This enables the user to link the tuple of different tables.
- ❖ The rows in the tables represent the records in the database, and the columns represent the attributes associated with the entity.

❑ **Types of Tuples**

- ❖ There are two types of tuples in a database management system:

1. Physical Tuples
2. Logical Tuples

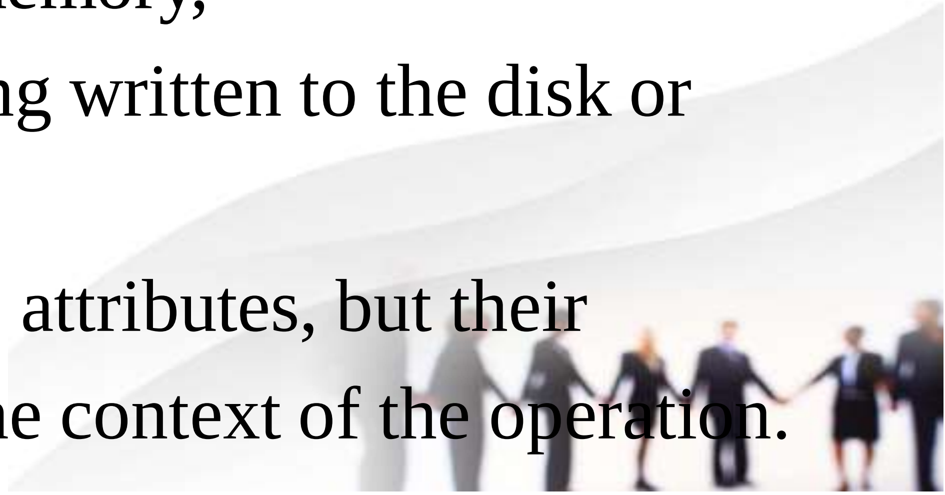
TUPLES

1. Physical Tuples

- ❖ Physical Tuples are nothing but the actual data stored in the storage media of a database.
- ❖ It is also known as a record or row.

2. Logical Tuples

- ❖ Logical Tuples are the data representation in memory,
- ❖ Here, the data is temporarily stored before being written to the disk or during a query operation.
- ❖ Both physical and logical tuples have the same attributes, but their representation and usage can differ based on the context of the operation.



DATABASE PROGRAMMING

- ❖ This programming information includes various resources for database programming, distributed database programming, SQL and embedded SQL programming.
- ❑ **Database programming**
- ❖ DB2 for IBM I provides a wide range of support for setting, processing, and managing database files.
- ❑ **Distributed database programming**
- ❖ Distributed database programming describes the distributed relational database management portion of the IBM I licensed program.

DATABASE PROGRAMMING

❖ Distributed relational database management provides applications with access to data that is external to the applications and typically located across a network of computers.

❑ **Embedded SQL programming**

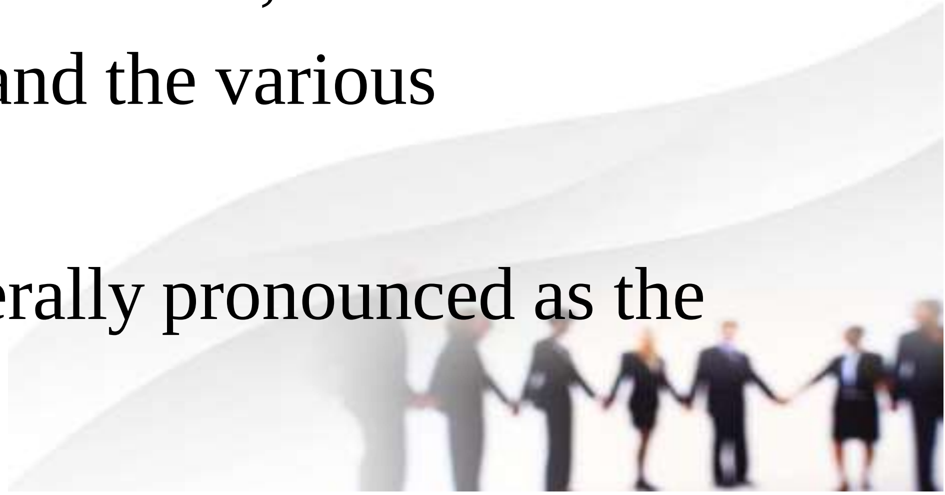
❖ This topic collection explains how to create database applications in host languages that use DB2 for IBM i SQL statements and functions.

❑ **SQL programming**

❖ The DB2 for IBM i database provides a wide range of support for the Structured Query Language (SQL).

DATABASE PROGRAMMING

- **SQL XML programming**
- The DB2 for IBM I database provides a wide range of support for using the SQL portion of the XML programming language.
- Structured query language (SQL) is a programming language for storing and processing information in a relational database.
- A relational database stores information in tabular form, with rows and columns representing different data attributes and the various relationships between the data values.
- The Structured Query Language (SQL) is generally pronounced as the “sequel” or “ess queue ell”.

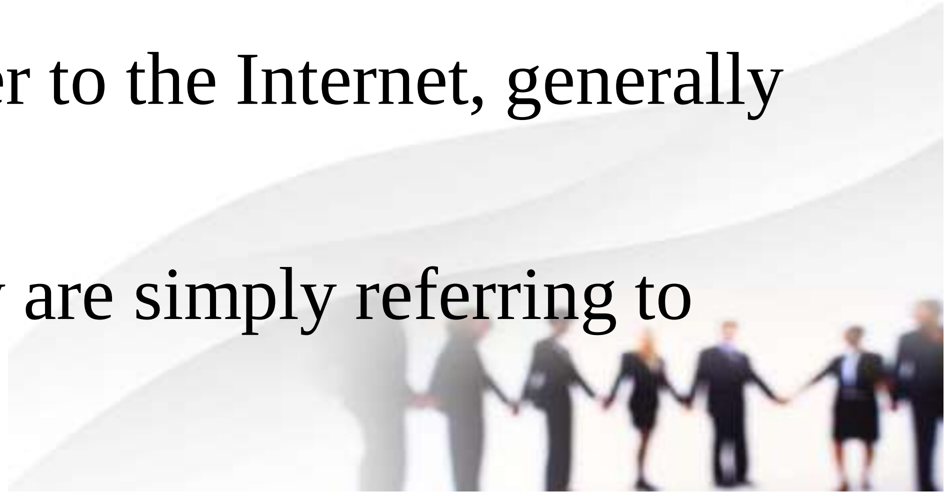


DATABASE PROGRAMMING

- SQL is a specialized computer programming language.
- It is tailored for interacting with data stored in relational databases.
- SQL provides all the necessary tools to create, read, update and delete (CRUD) the data in a database.

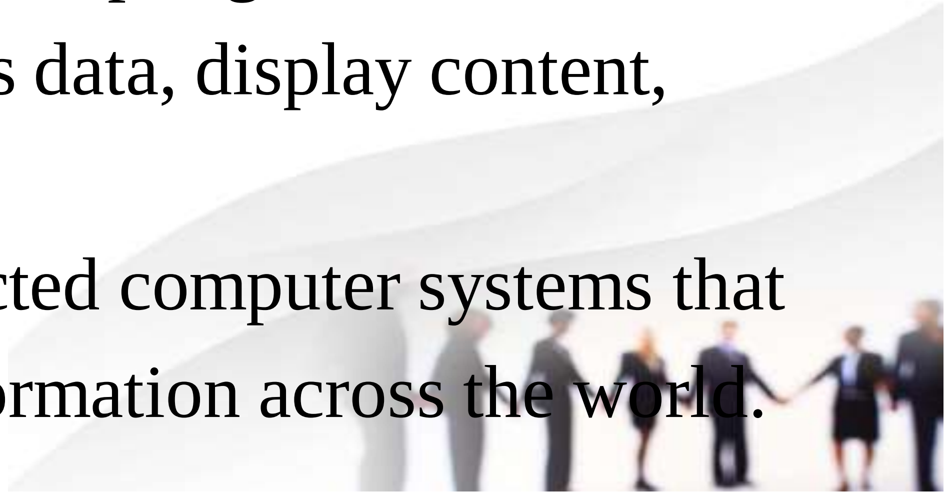
INTERNET PROGRAMMING

- ❖ The Internet is a vast network of computers, and servers, which communicate with each other.
- ❖ The Internet is a vast network that connects billions of computers and other electronic devices all around the world.
- ❖ You can get nearly any information, communicate with anyone on the globe, and do a lot more with the Internet.
- ❖ All of this is possible by connecting a computer to the Internet, generally known as going online.
- ❖ When someone says a computer is online, they are simply referring to the fact that it is linked to the Internet.



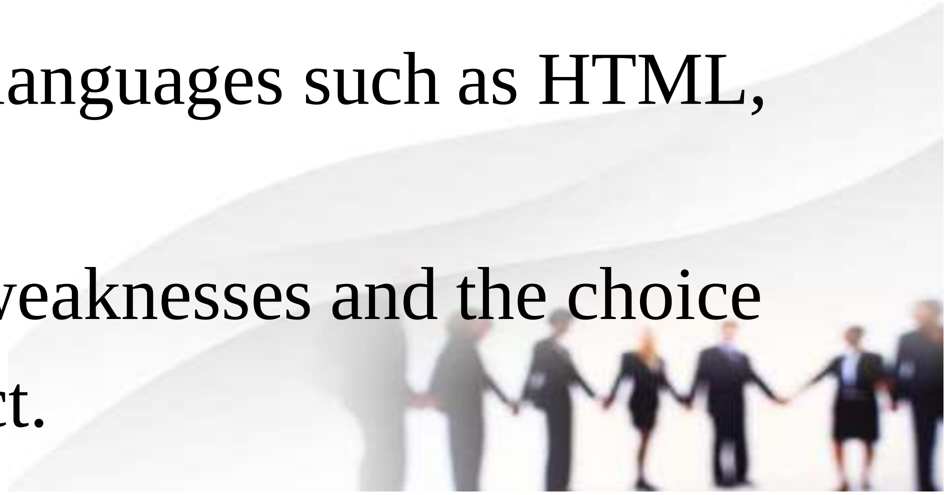
INTERNET PROGRAMMING

- ❖ The Internet is a global network of interconnected computers and servers that allows people to communicate, share information, and access resources from anywhere in the world.
- ❖ Web programming involves creating dynamic websites that are interactive and user-friendly.
- ❖ This includes the use of databases, server-side scripting, and client-side scripting to create applications that can process data, display content, and interact with users.
- ❖ The Internet is a global network of interconnected computer systems that enables communication and the sharing of information across the world.



INTERNET PROGRAMMING

- ❖ The Internet has revolutionized the way people communicate, learn, and conduct business.
- ❖ Web programming refers to the development of web applications and websites that are accessed over the Internet.
- ❖ Web programming involves creating web pages, web applications and other online content that can be displayed in a web browser.
- ❖ It is achieved using a variety of programming languages such as HTML, CSS, JavaScript, PHP, Python, Ruby and Java.
- ❖ Each of these languages has its strengths and weaknesses and the choice of language depends on the needs of the project.



INTERNET PROGRAMMING

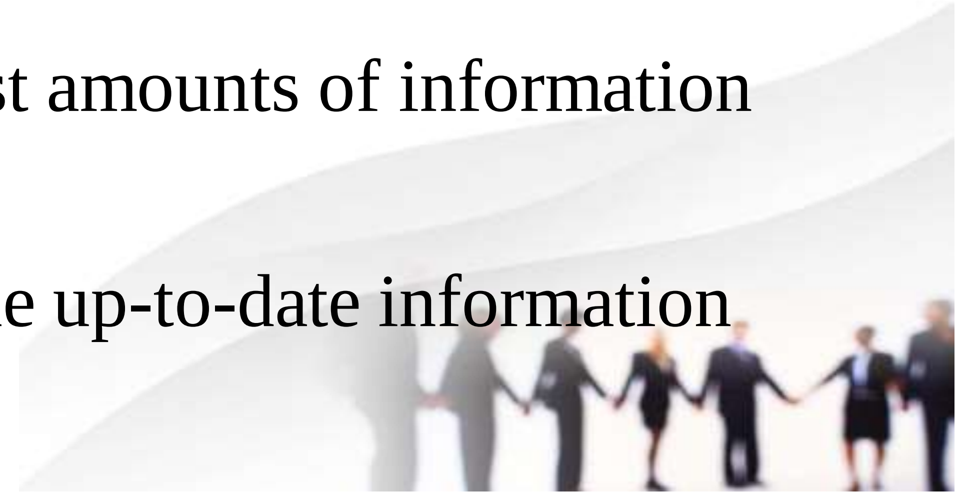
- ❑ **Uses of Internet and Web programming**

1. Communication

- ❖ The Internet has revolutionized communication, allowing people to connect with each other through email, social media, video conferencing, and instant messaging.

2. Information sharing

- ❖ The Internet has made it possible to access vast amounts of information quickly and easily.
- ❖ Websites like Wikipedia and news sites provide up-to-date information on a wide range of topics.



INTERNET PROGRAMMING

3. E-commerce

❖ The Internet has enabled businesses to sell products and services online, creating new opportunities for entrepreneurs and small businesses.

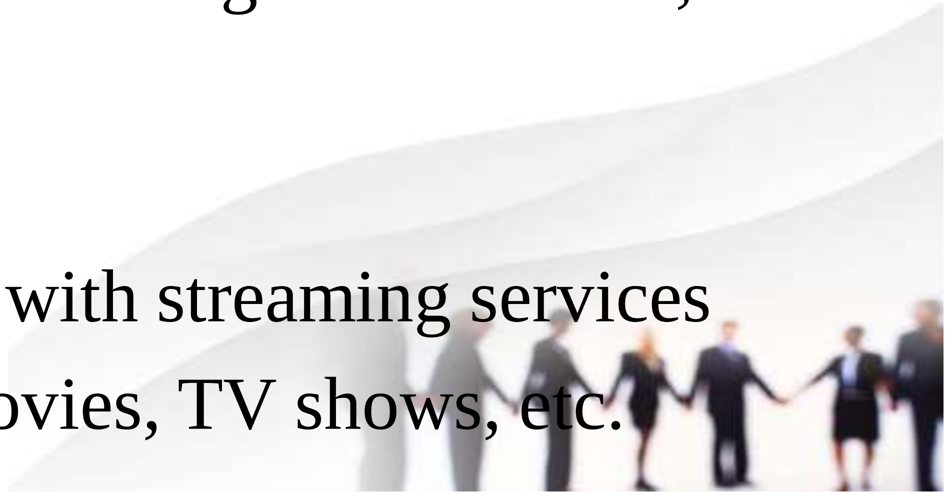
4. Education

❖ The Internet has opened up new opportunities for education

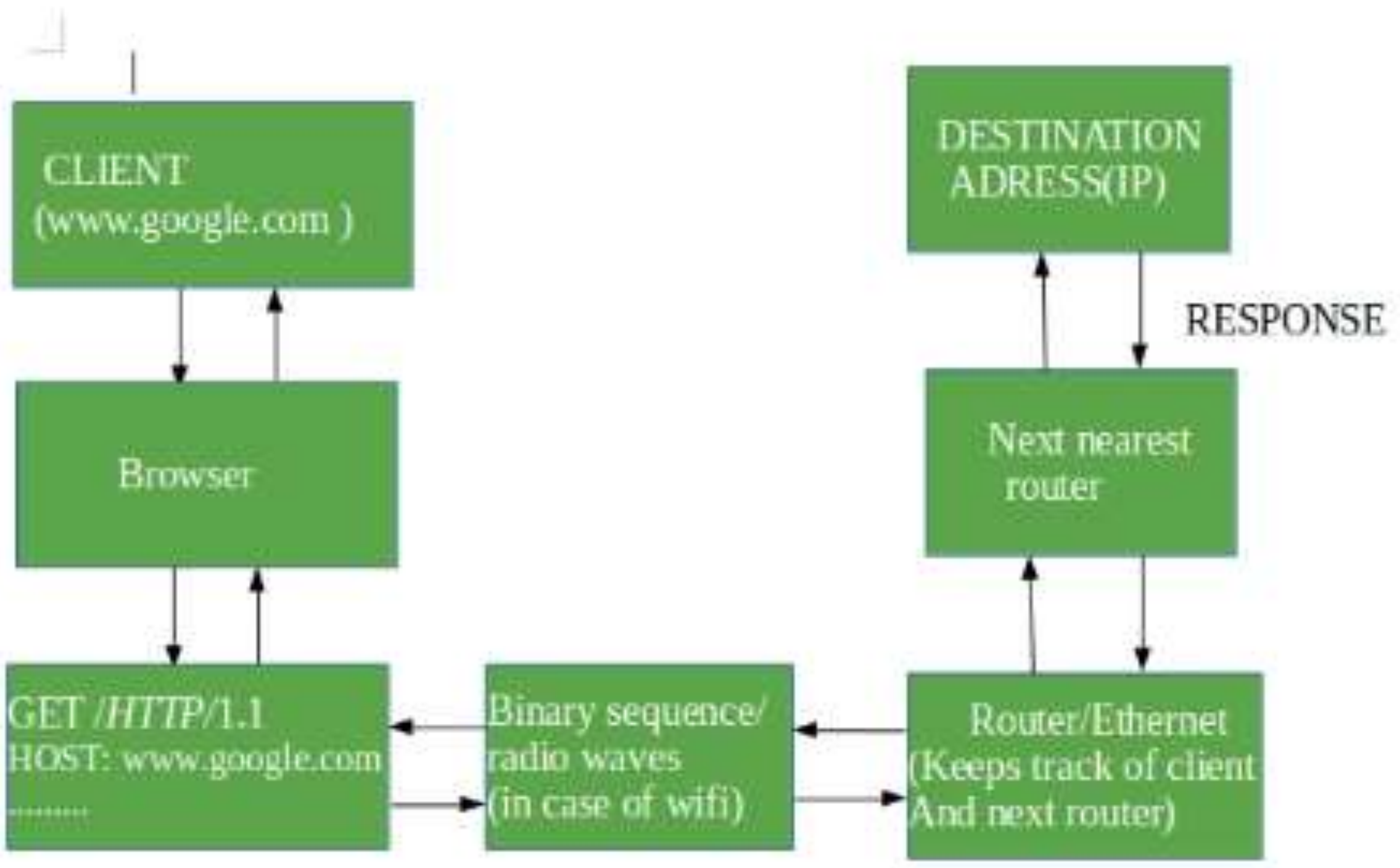
❖ This makes it possible for people to learn online through the MOOCs, webinars and other online courses.

5. Entertainment

❖ Internet has transformed way of entertainment with streaming services like Netflix & YouTube providing access to movies, TV shows, etc.



DESIGN PRINCIPLES OF INTERNET PROGRAMMING



Client-Server

DESIGN PRINCIPLES OF INTERNET PROGRAMMING

1. Client-side

- ❖ First, when we type a URL like www.google.com, the browser converts it into a file
- ❖ GET /HTTP/1.1 (where GET means we are requesting some data from the server and HTTP refers to a protocol that we are using, 1.1 refers to the version of the HTTP request)
- ❖ Host: www.google.com
- ❖ And some other information

DESIGN PRINCIPLES OF INTERNET PROGRAMMING

2. Server-side

- ❖ Now the server receives the binary code and decodes it and sends the response in the following manner:
- ❖ HTTP/1.1 200 ok (where 200 ok is the status)
- ❖ Content-type:type/HTML

❑ **Body of page**

- ❖ Now, this is converted back to binary by the server and sent to the IP address that is requesting it.
- ❖ Once the codes are received by the client, the browser again decodes the information in the following way.

DESIGN PRINCIPLES OF INTERNET PROGRAMMING

- First, it checks the status.
- Then it starts reading the document from the HTML tag and constructs a tree-like structure.
- The HTML tree is then converted to corresponding binary code and rendered on the screen.
- In the end, we see the website front-end.

DESIGN PRINCIPLES OF INTERNET PROGRAMMING

3. Protocols

- ❖ HTTP (HyperText Transfer Protocol) is the primary protocol used for communication on the web.
- ❖ Clients send HTTP requests to servers, which respond with the respective HTTP responses.
- ❖ HTTPS (HTTP Secure) is a secure version of HTTP that encrypts the data transferred between the client and server.

WINDOWS PROGRAMMING

- ❖ Windows programming involves using Windows GUI to create various applications that access the Microsoft Windows operating system.

❑ **Windows API**

- ❖ The application programming interface (API) that allows programs to access Windows features.
- ❖ Uses dynamic link library (DLL) technology to access API functionality.

❑ **Windows SDK**

- ❖ The Windows Software Development Kit (SDK) includes header files that allow access to Windows operating system components.
- ❖ Visual Studio installs the Windows SDK by default for C++ Desktop workload development.

WINDOWS PROGRAMMING

- **Forms**
- The main component of a Windows application, a form is a container for the application's control objects.
- A Windows application can have one or more forms.
- **Control objects**
- Objects that have properties that can be manipulated to change their behaviors or attributes.
- All objects have the Name and Text properties, with Name for the programmer and Text for the user.
- Many resources are available for learning Windows programming.

WINDOWS PROGRAMMING

❖ Some of the peculiar resources for learning Windows programming can be explained as follows:

❖ **Windows Programming for Everyone**

❖ A Udemy course that teaches how to program on a Windows computer using the Rexx programming language.

❖ **Programming Windows**

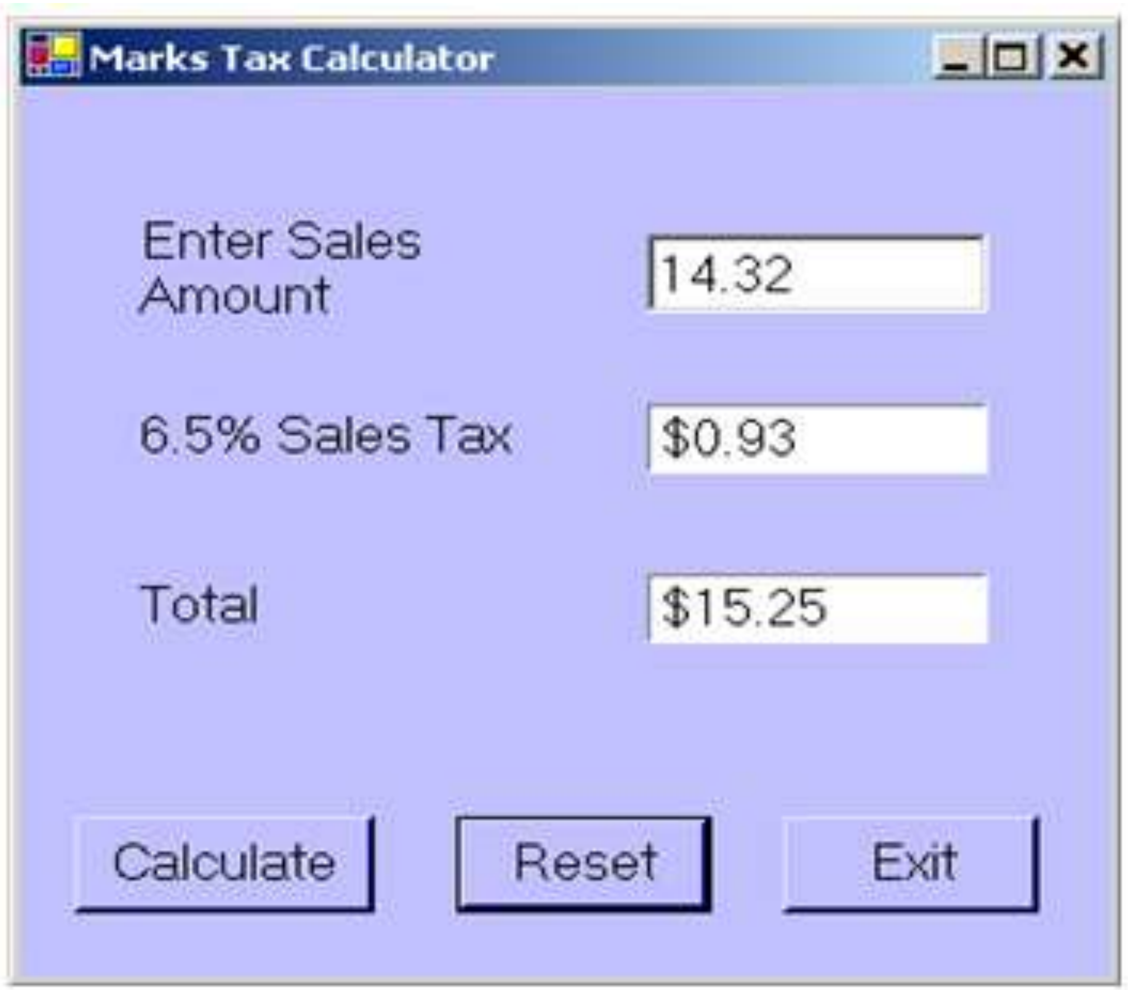
❖ A book by Charles Petzold that is considered a classic introductory text on Windows programming.

❖ **Introduction to Windows Programming**

❖ A YouTube video that discusses Windows programming.

WINDOWS PROGRAMMING

- ❖ A Windows application is a program which uses the standard Windows Graphical User Interface (GUI).
- ❖ An example of a Windows based application is shown here:
- ❖ The main component of a Windows application is the form.
- ❖ A Windows application may consist of a single form, or many forms.
- ❖ The form is the organizational "container" for the controls.


Marks Tax Calculator

Enter Sales Amount 14.32

6.5% Sales Tax $0.93

Total $15.25

WINDOWS PROGRAMMING

- ❖ An event in a programming context is a user initiated action that causes some resulting program actions to occur.
- ❖ For example, when you are in a text editing program, and you press the s key, the letter s appears in your document.
- ❖ The key-press is the user generated event, and the appearance of the letter s is the resulting action.
- ❖ Another example is when you move your mouse, the mouse pointer on the screen moves.
- ❖ The mouse movement is the event, and the pointer moving is the resulting action.