

## TOPIC: SPREADSHEET APPLICATION

### Definition of a Spreadsheet Application

A **spreadsheet application** is a software program used to store, organize, analyze, and manipulate data in a tabular format. It consists of rows and columns that form cells where users can input text, numbers, and formulas. Spreadsheet applications are commonly used for tasks such as financial analysis, budgeting, data visualization, and record-keeping.

Some common examples of spreadsheet applications include:

- Microsoft Excel
- Google Sheets
- Lotus 1-2-3
- LibreOffice Calc
- Apple Numbers
- Zoho Sheet

### Terminologies

- **Cell(s):** The intersection of row and column forms a cell. It is the smallest unit in a spreadsheet where data is entered, identified by a column letter and row number (e.g., A1, B5).
- **Rows:** Horizontal arrangements of cells in a spreadsheet, labelled with numbers (1, 2, 3...). Rows begin with 1 and end at 1048576.
- **Columns:** Vertical arrangements of cells, labelled with letters (A, B, C...). Columns begin with A up to XFD.
- **Worksheet:** This is a single sheet within a spreadsheet application containing rows and columns for data entry and analysis.
- **Workbook:** This is a collection of worksheets within a single spreadsheet file.
- **Active Cell:** The currently selected cell in a worksheet, usually highlighted with a border.
- **Cell Address (Reference):** The unique identifier of a cell, based on its column letter and row number (e.g., A1, C5).
- **Formula:** This is a mathematical expression entered in a cell to perform calculations or operations (e.g., = A1 + B2 + B3).
- **Function:** A predefined formula in a spreadsheet that performs calculations (e.g., SUM(), AVERAGE()).
- **Arguments:** refers to the values or references passed into a function to perform a specific calculation or operation. Arguments are enclosed in parentheses and can be numbers, text, cell references, ranges, or other expressions. Text must be in double quotes (“ ”) when used as an argument.
- **Name Box:** The area that displays the address of the active cell.
- **Sheet Tab:** The tab at the bottom of the workbook that allows switching between different worksheets.
- **Quick Access Toolbar:** A customizable toolbar that provides quick access to frequently used commands like Save, Undo, and Redo.
- **Ribbon (Menu Bar):** A collection of tabs and command groups that provide access to various spreadsheet tools and functions.
- **Scroll Bars:** Used to navigate horizontally or vertically through a large worksheet.
- **Row and Column Headers:** Numbered rows (1, 2, 3...) and lettered columns (A, B, C...) that help identify cell locations.

# FUNCTIONS AND FORMULAS

## Function

A **function** is a predefined formula in a spreadsheet application that performs calculations or operations on data. Functions take one or more inputs (called arguments) and return a result.

### Example:

- =SUM(A1:A5) → Adds the values in cells A1 to A5.
- =AVERAGE(B1:B10) → Calculates the average of the values in B1 to B10.

## Formula

A **formula** is an equation manually created by the user in a spreadsheet to perform calculations. A formula can use *\*cell references, numbers, operators (e.g., +, -, \*, /), and functions*. Every formula starts with an **equal sign (=)**.

### Example:

- =A1 + A2 + A3 → Adds values in A1, A2, and A3.
- =B1 \* 10 → Multiplies the value in B1 by 10.
- =(C1 + C2) / 2 → Finds the average of C1 and C2 without using a function.

## 1. Date and Time Functions

These are functions that help retrieve and manipulate date and time values.

- **TODAY()** – Returns the current date.
- **DAY(A1)** – Extracts the day from a given date in cell A1.
- **MONTH(A1)** – Extracts the month from a given date.
- **YEAR(A1)** – Extracts the year from a given date.

### Example:

If A1 contains "2024-02-11",

- =DAY(A1) returns **11**
- =MONTH(A1) returns **2**
- =YEAR(A1) returns **2024**
- =TODAY() returns **27-02-2025**
- =NOW() returns **10:22**

## 2. Mathematical Functions

Mathematical functions used for numerical calculations.

- **SUM()** – Adds values in specified cells.
- **PRODUCT()** – Multiplies values in specified cells.
- **SUMIF(range, condition, [sum\_range])** – Adds numbers in a range that meet a condition.
- **ROUND(value, decimals)** – Rounds a number to a specified number of decimal places.

### Example:

- =SUMIF(A1:A10, ">50") – Adds all numbers greater than 50 in the range A1:A10.
- =ROUND(12.5678, 2) – Returns **12.57**.

### 3. Statistical Functions

Used for counting and analyzing numerical and non-numerical data.

- **COUNT(range)** – Counts numeric values in a range.
- **COUNTA(range)** – Counts all non-empty cells (both numbers and text).
- **COUNTIF(range, condition)** – Counts cells that meet a condition.

**Example:**

- =COUNT(A1:A10) – Counts only cells that contain numeric values.
- =COUNTA(A1:A10) – Stands for COUNT ALL. It counts all non-empty cells.
- =COUNTIF(A1:A10, ">50") – Counts numbers greater than 50.

### 4. Text Functions

These are functions used to manipulate text strings. Text functions include the following;

- **PROPER(text)** – Capitalizes the first letter of each word.
- **UPPER(text)** – Converts text to uppercase.
- **LOWER(text)** – Converts text to lowercase.
- **CONCATENATE(text1, text2, ...)** – Joins multiple text strings together.
- **RIGHT(text, [num\_chars])**: Returns the specified number of characters from the end of a string.
- Example: RIGHT("Hello", 2) returns "lo".
- **LEFT(text, [num\_chars])**: Returns the specified number of characters from the start of a string.
- Example: LEFT("Hello", 2) returns "He".
- **MID(text, start\_num, num\_chars)**: Returns the specified number of characters from the middle of a string, starting at a position you define.
- Example: MID("Hello", 2, 3) returns "ell".

**Example:**

- =PROPER("excel functions") → **"Excel Functions"**
- =UPPER("hello") → **"HELLO"**
- =LOWER("WORLD") → **"world"**
- =CONCATENATE("Hello", " ", "World") → **"Hello World"**

### 5. Lookup and Reference Functions

These are functions used to retrieve values from a dataset. They include the following;

- **LOOKUP(lookup\_value, lookup\_vector, result\_vector)** – Finds a value in a range.
- **VLOOKUP(lookup\_value, table\_array, col\_index, [range\_lookup])** – Searches for a value in a table and returns a corresponding value from a specified column.

**Example:**

Student	Score	Grade
John	85	B+
Alice	80	B
Harun	88	B+
Ahmed	74	C+
Salia	92	A

To find the **Score** of John using the LOOKUP function,

=LOOKUP("John", A2:A6, B2:B6). This returns **85**

Using the VLOOKUP to find the **Grade** of Salia will be;

=VLOOKUP("John", A2:B10, 2, FALSE) → Returns John's score from column B.

## 6. Logical Functions

These are functions used for decision-making based on conditions.

- **IF(condition, value\_if\_true, value\_if\_false)** – Checks a condition and returns different values based on the result.
- **AND(condition1, condition2, ...)** – Returns TRUE if all conditions are met.
- **OR(condition1, condition2, ...)** – Returns TRUE if at least one condition is met.
- **IFERROR(expression, value\_if\_error)** – Returns an alternate value if an error occurs.
- =IF(AND(A1>=50, B1>=50), "Pass", "Fail")
- =IF(OR(A1>=50, B1>=50), "Pass", "Fail")

**Example:**

- =IF(A1>50, "Pass", "Fail") – Returns "Pass" if A1 is greater than 50, otherwise "Fail".
- =IFERROR(1/0, "Error occurred") → Returns "Error occurred" instead of showing a division error.

## 7. Database Functions

Used to perform calculations on database-like structures in spreadsheets.

- **DSUM(database, field, criteria)** – Adds values that match specified conditions.

- **DMIN(database, field, criteria)** – Finds the minimum value that meets criteria.
- **DMAX(database, field, criteria)** – Finds the maximum value that meets criteria.
- **DCOUNT(database, field, criteria)** – Counts numeric values that meet criteria.

### Example:

If you have a dataset with "Sales" in column B, and you want to sum only the sales that are greater than 1000, you can use:

- `=DSUM(A1:B10, "Sales", D1:D2)` (where D1:D2 contains the filtering criteria).

## Data Types in a Spreadsheet and Their Uses

- **Number:** Used for performing calculations, such as quantities, scores, or statistical values (e.g., 100, 45.6).
- **Text (Label):** Non-numeric data used for names, descriptions, or headings (e.g., "John Doe", "Product Name").
- **Date/Time:** Represents calendar dates and time values, useful for scheduling and tracking (e.g., 12/02/2024, 10:30 AM).
- **Currency:** Displays monetary values with currency symbols, used for financial calculations (e.g., \$500.00, €75.50).
- **Percentage:** Expresses values as percentages, commonly used in statistical analysis (e.g., 25%, 0.75 formatted as 75%).
- **Boolean (Logical):** Represents TRUE or FALSE values, used in decision-making and logical conditions.
- **Custom Data Types:** Includes user-defined formats like phone numbers, product codes, or specialized numerical formats.

## Editing Data

- To modify data in a cell, double-click the cell or select it and edit in the **Formula Bar**.
- Press **Enter** to save changes or **Esc** to cancel.
- Use **Backspace** to delete characters one by one or **Delete** to remove entire content.
- Copy (**Ctrl + C**) and Paste (**Ctrl + V**) allow moving or duplicating data.

## Conditional Formatting

- Allows automatic formatting of cells based on specified conditions.
- Access via **Home > Conditional Formatting** in Excel.
- Examples:
  - Highlighting cells above a certain value.
  - Formatting duplicate values.
  - Applying color scales or data bars based on value ranges.

## Custom Number Format

- Enables formatting numbers, dates, and text according to custom rules.
- Access via **Format Cells > Number > Custom** in Excel.
- Examples:
  - `#,###` → Adds thousand separators (e.g., 1,000).
  - `"Grade: "0` → Displays numbers as "Grade: 90".
  - `dd-mmm-yyyy` → Formats date as "12-Feb-2025".

## Importing Text Files

- Used to bring external data from **.txt** or **.csv** files into a spreadsheet.
- Access via **Data > Get External Data > From Text/CSV** in Excel.
- Import Wizard allows selecting delimiters (comma, tab, space) to properly structure data.

## Paste Special Option

- Provides advanced pasting options beyond simple copy-paste.
- Access via **Right-click > Paste Special** or **Ctrl + Alt + V**.
- Options include:
  - **Values:** Pastes only the raw data without formulas.
  - **Formats:** Copies only the formatting.
  - **Transpose:** Swaps rows and columns.
  - **Multiply/Divide:** Applies mathematical operations during pasting.

## Displaying Data

Effectively displaying data in a spreadsheet makes it easier to read, analyze, and interpret. Some common ways to enhance data presentation include:

- **Adjusting Column Width and Row Height:** Ensures that data is fully visible by resizing columns and rows to fit their content.
- **Text Wrapping:** Keeps all text visible within a cell without expanding the column width by wrapping long text onto multiple lines.
- **Merging Cells:** Combines multiple cells into one, useful for creating headers and formatting structured tables.
- **Sorting and Filtering:**
  - **Sorting** arranges data in ascending or descending order based on selected criteria (e.g., alphabetical order or numerical order).
  - **Filtering** allows users to display only specific data that meets certain conditions.
- **Conditional Formatting:** Highlights key information by changing cell colors, font styles, or icons based on conditions (e.g., marking negative values in red or highlighting the top 10 values).

## Freeze Row, Column, and Title

Freezing specific parts of a spreadsheet ensures that important data (such as headers or labels) remains visible while scrolling.

- **Freeze Top Row:** Keeps the first row fixed, which is useful for keeping column headers visible when working with large datasets.
  - To freeze the top row: Go to **View > Freeze Panes > Freeze Top Row**.
- **Freeze First Column:** Keeps the first column in place, useful when working with large spreadsheets that require constant reference to the first column.
  - To freeze the first column: Go to **View > Freeze Panes > Freeze First Column**.
- **Freeze Multiple Rows or Columns:**
  - Select a cell below the row and to the right of the column you want to freeze.
  - Go to **View > Freeze Panes > Freeze Panes** to lock everything above and to the left of the selected cell.

## What-If Tables (Data Tables)

What-If Analysis is used to explore different outcomes based on changes in input values. It helps in decision-making and forecasting.

### Types of What-If Tables:

#### 1. One-Variable Data Table:

- Used to see how changing a single input affects an outcome.
- Example: Changing the interest rate in a loan calculation to see its effect on monthly payments.

#### 2. Two-Variable Data Table:

- Used to analyze the impact of two different inputs on a result.
- Example: Examining how changes in both loan amount and interest rate affect monthly payments.

To create a data table, go to **Data > What-If Analysis > Data Table**, then input row and column variables.

## Data Protection

Data Protection is the act of limiting unauthorized or accidental modification of data for consistency. Protecting data ensures that important information is not accidentally changed or deleted.

### Ways to Protect Data in a Spreadsheet:

- **Protecting a Sheet:** Restricts editing or modifications to the entire sheet or specific areas.
  - Go to **Review > Protect Sheet**, set a password, and choose the actions users can perform.
- **Protecting a Workbook:** Prevents users from adding, deleting, or renaming sheets within the workbook.
  - Go to **Review > Protect Workbook** and set a password.
- **Locking Specific Cells:** Prevents modifications to specific cells while allowing other areas to remain editable.
  - Select the cells, right-click, choose **Format Cells > Protection**, check **Locked**, then protect the sheet.

### Adding and Removing Passwords

#### Adding a Password to a File:

- **To prevent unauthorized access:**
  1. Open the file.
  2. Go to **File > Info > Protect Workbook > Encrypt with Password**.
  3. Enter and confirm a password.
  4. Click **OK** and save the file.
- **To restrict editing but allow opening:**
  1. Go to **File > Save As > Tools > General Options**.
  2. Enter a password for modification.

3. Save the file.

### Removing a Password from a File:

- Open the file and enter the password.
- Go to **File > Info > Protect Workbook > Encrypt with Password**.
- Delete the password and save the file.

Password protection is useful for securing sensitive data but should be used carefully to avoid losing access to important files.

### Protecting Worksheets

By default, when a worksheet is protected, all cells are locked. If you want to allow certain cells to remain editable:

1. Select the cells you want to keep editable.
2. Right-click and choose **Format Cells**.
3. Go to the **Protection** tab.
4. Uncheck **Locked**, then click **OK**.

### How to Protect a Worksheet

1. Click on the **Review** tab in the Excel ribbon.
2. Select **Protect Sheet**.
3. A dialog box will appear. Choose the actions users can perform (e.g., selecting cells, inserting rows, deleting columns, etc.).
4. Enter a password (optional) to prevent unauthorized changes.
5. Click **OK** and confirm the password.

### Procedure to Unprotect a Worksheet

1. Go to the **Review** tab.
2. Click **Unprotect Sheet**.
3. Enter the password (if required) and press **OK**.

### What is Conditional Formatting?

Conditional Formatting is a feature in spreadsheet applications (such as Microsoft Excel or Google Sheets) that allows you to apply different formatting styles (e.g., colors, bold text, font changes) to cells based on specific conditions. This helps in visually identifying important data trends, errors, or key values.

### Steps to Apply Conditional Formatting in Excel

1. **Select the range of cells** you want to apply the formatting to.
2. Go to the **Home** tab and click on **Conditional Formatting** in the toolbar.
3. Choose the type of rule you want to apply, such as:
  - **Highlight Cells Rules** (e.g., values greater than a specific number).
  - **Top/Bottom Rules** (e.g., top 10 highest values).
  - **Data Bars** (graphical bars inside cells).
  - **Color Scales** (gradient colors based on value ranges).
  - **Icon Sets** (symbols like arrows or checkmarks).



4. Enter the condition (e.g., "Format cells greater than 100") and choose the formatting style (e.g., red fill color).
5. Click **OK** to apply the formatting.

## **Merging and Splitting Cells, Columns, and Rows**

### **Merging Cells**

Merging cells combines multiple adjacent cells into a single larger cell, often used for titles or headers.

### **How to Merge Cells in Excel**

1. Select the cells you want to merge.
2. Go to the **Home** tab.
3. Click **Merge & Center** (or use **Merge Across** or **Merge Cells** options).
4. The selected cells will be combined into one.

**Note:** When merging, only the content in the top-left cell is retained; other cell contents are deleted.

### **Splitting Merged Cells**

If you want to undo merging and return to separate cells:

1. Select the merged cell.
2. Go to the **Home** tab.
3. Click **Merge & Center** again (this will unmerge the cells).

## **Merging and Splitting Columns & Rows**

Unlike cells, columns and rows cannot be merged but can be adjusted in width and height for better formatting.

## **Sorting and Querying for Information in Spreadsheets**

### **Sorting Data**

Sorting helps organize data in a meaningful order, making it easier to analyze and interpret. In spreadsheet applications like Microsoft Excel or Google Sheets, you can sort data by a single column, multiple columns, or even perform custom sorts.

### **Sorting Data by Multiple Columns**

1. Select the range of data you want to sort (including column headers).
2. Go to the **Data** tab and click **Sort**.
3. Choose the first column to sort by and select **Ascending** or **Descending** order.
4. Click **Add Level** to sort by a second column.
5. Repeat for additional columns if needed.
6. Click **OK** to apply sorting.

Example: If sorting student records, you might first sort by **Class** and then by **Name** within each class.

## Performing Custom Sorts

Custom sorts allow sorting based on specific conditions, such as sorting by months (January, February, etc.) instead of alphabetically.

1. Select the data range.
2. Click **Sort** under the **Data** tab.
3. Choose a column, then select **Custom List**.
4. Enter the preferred order (e.g., "Low, Medium, High" or days of the week).
5. Click **OK** to apply.

## Querying for Information

Querying helps filter and extract specific data based on defined criteria.

### Creating a Single or Multiple Criteria Query

1. Select the dataset.
2. Go to **Data > Filter** to enable filter options.
3. Click the filter dropdown on the desired column.
4. Enter a condition (e.g., "Show only values greater than 50").
5. To apply multiple criteria, use filters on different columns.

Example: If managing sales data, you could filter products sold in **January** with sales **above \$1000**.

### Using Advanced Query/Filter (Advanced Filtering in Excel)

Advanced filtering allows more complex queries using **AND/OR** conditions.

1. Select your dataset.
2. Go to **Data > Advanced Filter**.
3. Choose whether to filter in place or copy results to another location.
4. Define criteria in a separate range (e.g., "Price > 500" AND "Category = Electronics").
5. Click **OK** to apply.

## Using Graphs and Charts to Represent Data

When we use Graphs and charts, it helps us visually represent data trends and comparisons.

### Steps to Create a Chart in Excel

1. Select the dataset to visualize.
2. Go to the **Insert** tab and choose a chart type (e.g., **Bar, Line, Pie, Column**).
3. Adjust chart elements such as title, labels, and colors.
4. Use the **Chart Tools** menu to customize further.

### Common Chart Types and Their Uses

- **Column Chart** – A column chart represents data using vertical bars. Each bar's height corresponds to a value, making it easy to compare different categories. It is useful when displaying distinct groups or categories with numerical values.
- **Line Chart** – A line chart connects data points with a continuous line, showing changes over time. It is effective for illustrating trends, patterns, and fluctuations in data across periods.
- **Pie Chart** – A pie chart is a circular chart divided into slices, where each slice represents a portion of the whole. The size of each slice is proportional to the value it represents, making it ideal for showing percentage distributions.
- **Bar Chart** – A bar chart is similar to a column chart but uses horizontal bars instead of vertical ones. It is particularly useful for comparing values across categories, especially when category names are long or there are many categories.

## TOPIC: INTRODUCTION TO PROGRAMMING

### Computer Program

This is a set of instructions that a computer can follow to perform a specific task. These instructions are written in a programming language and can be simple, like performing a basic calculation, or complex, like running a web browser or a video game. In its human-readable form, a program is also called source code. This source code is then translated into a form the computer can directly execute, known as machine code.

**Programming**, on the other hand, is the process of creating those programs. Thus, it's the act of writing the step-by-step instructions (code) that tell a computer what to do or how to perform a task. A professional who does this is called a **programmer**.

### What is a Programming Language?

A programming language is a formal, artificial language used to create computer programs. It consists of a specific set of rules, syntax, and vocabulary that programmers use to write instructions. These instructions, known as source code, define algorithms and data structures that a computer can interpret and execute. In essence, a programming language serves as a crucial bridge, translating human logic and intentions into machine-readable commands that direct a computer's behavior.

Eg.

### Analogy

*A programming language is like the language of a science textbook.*

- Scientists (programmers) use it to write experiment procedures (programs).
- A student (computer) conducting the experiment must understand the language to follow the instructions.
- If the textbook is in a language the student doesn't understand, a translator (compiler or interpreter) is needed to translate it into a language the student knows (machine code).
- Some translators (compilers) translate the entire book before the student starts the experiment, while others (interpreters) translate step by step as the student reads.

### History of Programming languages

In **1843**, **Ada Lovelace** wrote the first algorithm for a mechanical computer, making her the world's first programmer.

In **1945**, **John von Neumann** introduced the **von Neumann architecture**, which allowed computers to store both programs and data in memory. This breakthrough laid the foundation for modern programming. Around the same 1945, computers used **machine language** (binary 0s and 1s), but **Assembly Language** was later introduced to simplify coding.

In **1957**, **John Backus** developed **FORTRAN** (Formula Translator) for scientific computing. FORTRAN was considered the first high-level language. This was followed by **COBOL (Common Business Oriented Language)** in **1959** by **Grace Hopper** for business applications. COBOL was similar to English, hence was a High-level language.

The **1970s** saw the birth of **C (1972)** by **Dennis Ritchie**, which became the foundation for modern programming. **Bjarne Stroustrup** later enhanced it into **C++ (1983)** with object-oriented features.

With the rise of the internet in the **1990s**, **Python (1991)** by **Guido van Rossum**, **Java (1995)** by **James Gosling**, and **JavaScript (1995)** by **Brendan Eich** revolutionized software development.

In the 2000s, **C# (2000)** by **Microsoft**, **Swift (2014)** by **Apple**, and **Kotlin (2016)** emerged for modern app development.

Today, programming continues to evolve with AI and cloud computing shaping the future.

## Categories of Programming Languages

Programming languages are broadly categorized into two main types:

1. Low-level languages
2. High-level languages

### Low-Level Programming Languages

Low-level programming languages are those that are **closer to the machine language** and interact directly with the computer's processor. They are efficient in terms of speed and memory usage but are more difficult for humans to read and write compared to high-level languages. Low-level languages are primarily used in system programming, embedded systems, etc.

Examples of Low-level language are Assembly language and Machine language.

#### Machine Language (First Generation Language - 1GL)

Machine language is the most basic form of programming language, consisting entirely of **binary digits (0s and 1s)**. Since computers can only understand binary instructions, machine language allows programs to be executed without requiring any translation. However, writing machine code is extremely difficult for humans because it lacks readability and is prone to errors.

For example, an instruction to add two numbers in machine language might look something like this:  
**11011010 10101100**

Since machine language is directly understood by the computer's CPU, it executes very quickly. However, it is highly specific to the processor it is written for, meaning that machine code written for one type of processor will not work on another.

#### Advantages of Machine Language

- ✓ **Fast Execution** – Directly understood by the CPU, making it the fastest language.
- ✓ **No Translation Needed** – Does not require a compiler or interpreter, reducing processing time.
- ✓ **Full Hardware Control** – Allows direct interaction with the computer's hardware and memory.
- ✓ **Efficient Memory Usage** – Uses minimal system resources, making it ideal for embedded systems.
- ✓ **Works on Any System with the Same CPU** – No need for additional software to run the code.

#### Disadvantages of Machine Language

1. **Difficult to Learn and Use** – Written in binary (0s and 1s), making it hard to read, write, and debug.
2. **Hardware Dependent** – Code written for one type of processor does not work on another.
3. **Error-Prone** – Since it consists of long sequences of binary digits, even a small mistake can cause major issues.
4. **Time-Consuming** – Writing programs in machine language takes a lot of time and effort.
5. **Lack of Portability** – Cannot be transferred easily between different computer systems.

## Assembly Language

Assembly language is a **low-level programming language** that uses symbolic names (called **mnemonics**) instead of binary code (0s and 1s) to represent machine instructions. It is easier to read and write than machine language but still requires knowledge of the computer's architecture.

Unlike machine language, which consists entirely of binary numbers, assembly language uses short, readable codes (mnemonics) like MOV, ADD, and SUB to represent operations. However, since computers cannot directly understand mnemonics, a special program called an **assembler** is used to convert assembly code into machine code for execution.

### Opcode

Opcode (Operation Code) is the **part of an instruction** that specifies the operation to be performed by the CPU. It tells the computer **what to do** (e.g., add, move, or store data).

**Example:** In ADD AX, BX, the opcode is ADD (addition operation).

### Operand

An operand is the **data or memory location** on which the opcode operates. It specifies **what to process** in an instruction.

**Example:** In ADD AX, BX, AX and BX are operands (the values being added).

### Mnemonics

Mnemonics are **symbolic short words** used in assembly language to represent machine instructions, making the code more readable. They replace binary opcodes with human-friendly commands.

**Example:** Instead of 10110000 00000001 (binary machine code), assembly language uses MOV AL, 1, where MOV is the mnemonic.

Instruction	Meaning
MOV	Transfers data from one location to another.
ADD	Adds two values and stores the result.
SUB	Subtracts one value from another.
STA	Stores the contents of the accumulator into memory.
LDA	Loads data from memory into the accumulator.
OUT	Sends data from the accumulator to an output device.
INP	Receives data from an input device into the accumulator.
	Jumps to a specified memory address in the program.

<b>JMP</b>	
<b>CMP</b>	Compares two values and sets flags accordingly.
<b>HLT</b>	Stops program execution.

## Assemblers

An **assembler** is a software program that **translates assembly language code** into **machine language (binary code)** so that the computer's processor can execute it.

### How Assemblers Work

1. **Takes Assembly Code** – The programmer writes code using mnemonics like MOV, ADD, STA.
2. **Converts to Machine Code** – The assembler translates these mnemonics into binary instructions.
3. **Generates Executable Code** – The output is a machine-readable file that the CPU can run.

### Advantages of Assembly Language

1. **Faster Execution** – Since it is closer to machine code, it runs much faster than high-level languages.
2. **Efficient Memory Usage** – Allows direct control over hardware, leading to optimized memory and CPU usage.
3. **More Readable than Machine Code** – Uses mnemonics instead of binary, making it easier to write and understand.
4. **Direct Hardware Control** – Provides access to system components like CPU registers and memory.
5. **Useful for System Programming** – Ideal for writing operating systems, embedded systems, and device drivers.
6. **Compact and Optimized Code** – Produces smaller and more efficient executable files compared to high-level languages.

### Disadvantages of Assembly Language

1. **Difficult to Learn and Write** – Requires knowledge of hardware architecture and low-level coding.
2. **Time-Consuming Development** – Writing programs in assembly takes longer compared to high-level languages.
3. **Hardware Dependent** – Code written for one processor may not work on another without modification.
4. **Error-Prone** – Since it requires precise coding, debugging can be challenging.
5. **Limited Portability** – Assembly programs must be rewritten for different computer architectures.
6. **Less Readable** – Though better than machine code, assembly language is still harder to understand than high-level languages.

<b>Assembly Language</b>	<b>Machine Language</b>
Uses symbolic names (mnemonics) like MOV, ADD.	Uses binary code (0s and 1s).
More human-readable and easier to write.	Not human-readable and difficult to write.
Needs an <b>assembler</b> to convert to machine code.	Directly executed by the CPU without translation.
Easier to modify and debug than machine code.	Difficult to modify and debug.
Slightly slower due to translation	Fastest execution speed.
Example: <code>MOV AL, 1h</code>	Example: <code>10110000 01100001</code>

## Data Types

This is the type of data a variable can hold that tells the compiler or interpreter how the data should be handled and what operations can be performed on it. Data type include the following:

**Integer (int):** Represents whole numbers, positive or negative, without decimal points.

Eg. 14, -65, 342, etc.

**Float:** Represents real numbers with decimal points. Eg. 5.21, 9.013, 0.02, etc.

**String:** Represents sequence of characters, enclosed in quotes. Eg. "Hello World!", "Hamza", "Muslima", etc.

**Character (char):** Represents a single character. Eg. 'A', '5', 'y', 'n', 't', 'f', etc.

**Boolean (bool):** A value that can be one of two values. Eg. True or False.  
Used in logical operations.



**Double:** Represents numbers with decimal points up to about 16 decimal places.

## Variables and Constants

A variable is a named storage location used to hold a value or data during the execution of a program.

It acts as a placeholder for different types of information, such as numbers, text, or complex data structures.

Variables allow programmers to manipulate and store data.

## Declaration of Variables

Before a variable is used, it must be initially declared

To declare a variable, state the data type and the name of the variable

For example, to declare a variable that is intended to hold the name of a student, the declaration could be;

- `String StudName;`
- `String studentName;`
- `String stud_Name;` Etc.

A variable to store the age of a student could be;

- `Int age;`
- `Int Stud_age;`
- `Int studentAge;`
- Etc.

**NOTE:** In most programming languages, declaration and assignment go together.

For example; `String studentName = "kofi";`

## Constants

Constants are variables that hold values that cannot be changed or modified during the execution of a program.

Once a constant is defined and assigned a value, it remains fixed throughout the program's execution, and any attempt to alter its value results in an error or is simply ignored.

Constants are used to represent fixed or unchanging data that is critical to the program's logic, configuration settings, or any value that should not be accidentally or intentionally modified during the program's runtime.

## Declaration of Constants

In declaring a constant, a keyword is used to indicate that it is a constant, not a variable, depending on the programming language used.

For example; in JavaScript, the keyword **const** is used to precede the constant's name.

Eg. **const** Rate = 15.3

In Java, the keyword for constant is **final**, followed by datatype and name of constant.

Eg. **final** double Pi = 3.147;

## Rules for Variable Declaration

1. Variables cannot contain special characters. Eg. @, #, %, &, ?, etc. except the underscore (\_).
2. Variables must not begin with a number. Eg. 4age
3. Keywords cannot be used as variable names.
4. Limitation of number of characters in the name of the variable.
5. Repetition of variables names in the same scope is not allowed.
6. Some languages require assignment at the time of variable declaration.

NOTE: These rules are for most programming languages, not all. Languages such as Qbasic.

## Exercise

1. Differentiate between variables and constants.
2. Write one example of declaration of each in Q1.

## Expressions and Assignments

An **expression** is a combination of values, variables, and operators that resolves to a single value. It's a fundamental building block in programming used to calculate or retrieve a value.

- **Example:** (5 + 3) \* 2 is an expression. The operations are carried out, resulting in the single value 16.
- Another example is x > 5, which evaluates to a boolean value, either `true` or `false`.

An **assignment** is the process of storing a value in a variable. The **assignment operator**, typically =, is used to copy the value on the right side of the operator into the variable on the left side.

- **Example:** x = 10 is an assignment statement. The value 10 is assigned to the variable x. Similarly, result = (5 + 3) \* 2 assigns the result of the expression on the right (16) to the variable result.

## Operators and Precedence

**Operators** are special symbols that perform specific operations on one or more operands (values or variables).

- **Arithmetic Operators:** Used for mathematical calculations.
    - **+** : Addition (e.g., `5 + 3` results in 8)
    - **-** : Subtraction (e.g., `10 - 4` results in 6)
    - **\*** : Multiplication (e.g., `2 * 6` results in 12)
    - **/** : Division (e.g., `15 / 3` results in 5)
  - **Relational Operators:** Used to compare two operands and produce a boolean result (`true` or `false`).
    - **==** : Equal to (e.g., `5 == 5` is `true`)
    - **!=** : Not equal to (e.g., `5 != 10` is `true`)
    - **<** : Less than (e.g., `3 < 7` is `true`)
    - **>** : Greater than (e.g., `12 > 9` is `true`)
  - **Logical Operators:** Used to combine multiple boolean expressions.
    - **AND**: Returns `true` if both expressions are `true`.
    - **OR**: Returns `true` if at least one expression is `true`.
    - **NOT**: Inverts the boolean value of an expression.
- 

## Operator Precedence

**Operator precedence** is a rule that dictates the order in which operators in an expression are evaluated. Operators with higher precedence are performed before those with lower precedence.

- **Example:** In the expression `5 + 3 * 2`, multiplication (`*`) has higher precedence than addition (`+`). Therefore, the expression is evaluated as `5 + (3 * 2)`, which is `5 + 6`, resulting in 11. Parentheses (`()`) can be used to override precedence, forcing an operation to be evaluated first, such as in `(5 + 3) * 2`, which evaluates to 16.
- 

## Input/Output Statements

- **Input:** Statements that allow a program to receive data from an external source, like a user typing on a keyboard. Common keywords include `read`, `input`, or `cin`.
  - **Output:** Statements that allow a program to display data to an external destination, like a screen. Common keywords include `print`, `output`, or `cout`.
-

## Built-in Functions

**Built-in functions** are pre-written blocks of code that perform specific tasks. They save time and simplify programming. Examples include functions for mathematical calculations (`sqrt`, `pow`), string manipulation (`len`, `upper`), or type conversion (`int`, `str`).

---

## Sequential and Conditional Execution

- **Sequential Execution:** The default flow of a program, where instructions are executed one after another, in the order they are written.
  - **Conditional Execution:** Allows a program to make decisions and execute different code blocks based on whether a condition is `true` or `false`. The primary construct is the `if-else` statement. For example, `if (age >= 18) {...} else {...}`.
- 

## Looping Constructs

**Loops** are used to repeat a block of code multiple times. This is essential for tasks that require repetition.

- **for loop:** Repeats a set number of times. Used when you know the number of iterations in advance.
  - **while loop:** Repeats as long as a condition is `true`. Used when the number of iterations is unknown and depends on the condition.
  - **do-while loop:** Similar to a `while` loop, but the code block is executed at least once before the condition is checked.
- 

## Single-Dimensional Arrays

A **single-dimensional array** is a data structure that stores a collection of elements of the same data type in contiguous memory locations. It's indexed, allowing you to access any element directly using its position (index). For example, `scores = [90, 85, 78, 92]` is an array of integers.

---

## Nested Loops

A **nested loop** is a loop placed inside another loop. The inner loop executes all its iterations for each single iteration of the outer loop. This is useful for tasks that involve processing multi-dimensional data structures, like a 2D array or grid. For example, you can use nested loops to print a multiplication table or iterate through the cells of a spreadsheet.

# Terminologies in Programming

---

## Boolean Expression

A **Boolean expression** is a statement that evaluates to either `true` or `false`. These are fundamental to decision-making in programming. For example, `x > 5` is a Boolean expression.

---

## Class

A **class** is a blueprint or a template for creating objects in object-oriented programming (OOP). It defines the properties (data) and methods (functions) that an object of that class will have.

---

## Comment

A **comment** is a line of text within a program's source code that the compiler or interpreter ignores. Programmers use comments to explain their code, making it easier for others (or themselves in the future) to understand.

---

## Compiler

A **compiler** is a program that translates source code written in a high-level programming language (like C++ or Java) into machine code that a computer's processor can execute. It checks for errors and produces an executable file.

---

## Debugging

**Debugging** is the process of finding and fixing errors or bugs in a program's code. This can involve using special tools to step through the code and inspect the values of variables to identify where the problem lies.

---

## Event Procedure

An **event procedure** (also called an **event handler**) is a block of code that runs in response to a specific event, such as a user clicking a button, typing in a text box, or a timer running out.

---

## Syntax

**Syntax** refers to the set of rules that define the correct structure and grammar of a programming language. It's like the grammar rules of a spoken language.

---

## Variable

A **variable** is a named storage location in a computer's memory that holds a value. The value of a variable can change during the program's execution.

---

## Compile-Time Error

A **compile-time error** is an error detected by the compiler before the program is executed. This category includes syntax errors. The program will not compile until these errors are fixed.

---

## Syntax Errors

**Syntax errors** are mistakes in the code's structure, violating the rules of the programming language. Examples include forgetting a semicolon at the end of a line or misspelling a keyword. The compiler or interpreter will catch these.

---

## Runtime Errors

A **runtime error** is an error that occurs while the program is running. The program may compile correctly but crash or produce unexpected results when it tries to perform an invalid operation, such as dividing by zero or accessing a non-existent file.

---

## Coding

**Coding** is the process of writing instructions for a computer in a programming language. It involves translating logic and algorithms into source code.

---

## OOP

**OOP** stands for **Object-Oriented Programming**. It's a programming paradigm based on the concept of "objects," which can contain both data and methods. The goal is to structure programs in a way that models real-world objects and their interactions.

---

## Program Development Life Cycle

The **program development life cycle** is a systematic process for creating a new program. It ensures the final product is efficient, reliable, and meets the user's needs. The key steps are:

---

## 1. Problem Definition

This is the initial stage where you clearly **identify and understand the problem** you want to solve with the program. It involves defining the program's purpose, what it should accomplish, and its target audience. A well-defined problem is the foundation of a successful program.

---

## 2. Problem Analysis

Once the problem is defined, you **analyze the requirements**. This step involves figuring out what data the program will need (**inputs**), what calculations or processes it will perform, and what information it should produce (**outputs**). You also consider any constraints or limitations.

---

## 3. Algorithm Design and Representation

An **algorithm** is a step-by-step procedure for solving the problem. In this stage, you design the logic of the program. The algorithm can be represented using tools like **flowcharts** (diagrams that show the steps and decisions) or **pseudocode** (a plain-language description of the code).

---

## 4. Actual Coding

This is the process of translating the algorithm into a **specific programming language**. You write the source code, following the syntax and rules of the chosen language. This is where the program takes its concrete form.

---

## 5. Testing and Debugging

After writing the code, you must **test it thoroughly** to find and fix errors. **Testing** involves running the program with various inputs to ensure it produces the correct output. **Debugging** is the process of locating and correcting any errors (**bugs**) found during testing.

---

## 6. Complete Documentation and Implementation

The final step is to create **documentation** that explains how the program works and how to use it. This includes user manuals and technical details for other programmers. Once everything is documented and tested, the program is **implemented**, meaning it's installed and put into use.

# What is an Algorithm?

An **algorithm** is a precise, step-by-step procedure for solving a particular problem or completing a task. It's a set of unambiguous instructions that, when followed, will reliably produce a solution. Think of an algorithm as a recipe for a computer. Just like a recipe lists the ingredients and steps to bake a cake, an algorithm lists the inputs and instructions for a computer to get a specific output.

---

## Key Characteristics of an Algorithm

- **Sequence of Steps:** An algorithm is a series of well-defined steps. These steps must be executed in a specific order to achieve the desired result.
- **Set of Instructions:** The instructions are a clear method for what needs to be done. There's no room for interpretation; each step is explicit.
- **Problem-Solving:** The entire purpose of an algorithm is to solve a given problem. Whether it's sorting a list of names or calculating a complex equation, the algorithm provides the method to do it.

For example, to find the largest number in a list, an algorithm would be:

1. Start with the first number in the list and assume it's the largest.
2. Go through the rest of the numbers one by one.
3. If you find a number that's larger than the one you currently have, replace it.
4. After checking all the numbers, the one you have is the largest.

## Techniques for Representing Algorithms

Algorithms can be represented in various ways to make them easier to understand and communicate before they're written in a specific programming language. The main techniques are pseudocode, flowcharts, and actual code.

---

## Pseudocode

**Pseudocode** is a simplified, informal language used to describe the steps of an algorithm. It's not a real programming language, so it doesn't have a strict syntax. It combines natural language with some programming-like keywords (like `IF`, `THEN`, `ELSE`, `FOR`, `WHILE`) to outline the logic of the algorithm. Pseudocode is helpful because it's language-independent and focuses on the logic rather than the syntax.

### Example:

```
BEGIN
    DECLARE number AS INTEGER
    READ number
    IF number > 0 THEN
        PRINT "Positive"
    ELSE
        PRINT "Not positive"
    END IF
END
```


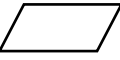
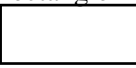
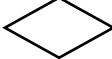

---



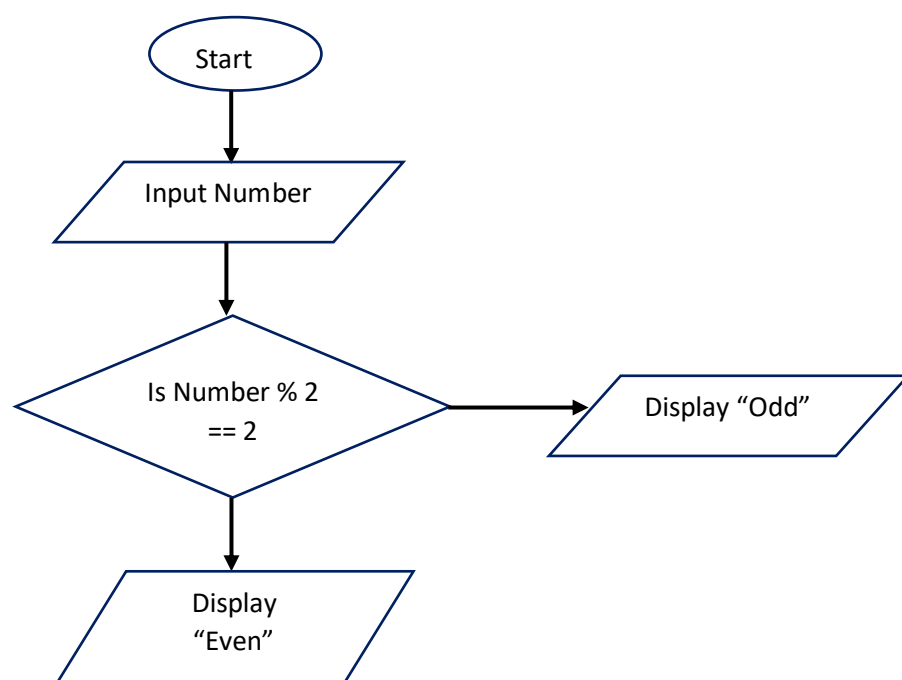
## Flowcharts

A **flowchart** is a graphical representation of an algorithm. It uses standard symbols to illustrate the sequence of steps and decisions. Flowcharts are great for visualizing the flow of a program and making the logic easy to follow.

- **Common Symbols:**
  - **Oval:** Represents the start or end of the program.
  - **Rectangle:** Represents a process or action.
  - **Parallelogram:** Represents input or output.
  - **Diamond:** Represents a decision point (e.g., a condition like `IF . . . THEN`).
  - **Arrows:** Show the direction of the flow.
- Each representation has its own advantages, depending on the level of clarity and detail required.

Symbol	Name	Meaning
<b>Oval</b> 	Terminator (Start)	Represents the <b>Start</b> or <b>End</b> of a flowchart.
Parallelogram 	Input/Output	Used for <b>input</b> (e.g., user data) and <b>output</b> (e.g., displaying results).
Rectangle 	Process	Represents a <b>step</b> in the process, such as calculations or assignments.
Diamond 	Decision	Used for <b>decision-making</b> (Yes/No, True/False) conditions.
Arrow 	Flowline	Shows the <b>direction</b> of the flow from one step to another.

### Flowchart to Determine Whether a Number is Even or Odd



---

## Actual Code

The ultimate way to represent an algorithm is by writing it in a **specific programming language**. This is the final step where the logical steps of the algorithm are translated into instructions that a computer can execute. This representation requires adhering to the strict syntax rules of the chosen language.

- **Example (Python):**

Python

```
number = int(input("Enter a number: "))

if number > 0:
    print("Positive")
else:
    print("Not positive")
```

## Conditionals

**Conditionals** allow a program to make decisions and choose between different paths of execution. They check if a specific condition is `true` or `false` and then run a particular block of code based on the result. The most common type is the `if-else` statement.

### Structure:

```
If (condition = true)
    statement block 1
Else
    statement block 2
End if
```

### How it Works:

The program first evaluates the `condition`. If the condition is `true`, it executes the code in `statement block 1`. If the condition is `false`, it skips to the `Else` part and executes the code in `statement block 2`.

---

## Loops

**Loops** are used to repeat a block of code multiple times. They are essential when you need to perform the same action over and over without writing the same lines of code repeatedly.

### Structure:

```
Loop while (condition = true)
    statement block
End Loop
```

### How it Works:

The program checks the `condition` at the beginning of each cycle. If the condition is `true`, it executes the `statement block`. After the block is finished, it goes back and checks the condition again. The loop

continues to run as long as the condition remains `true`. When the condition becomes `false`, the loop stops, and the program moves on to the next instruction.

# Topic: INTRODUCTION TO DATA PROCESSING SYSTEMS

## Microsoft Access

Microsoft Access is a **Database Management System (DBMS)** designed for small to medium-sized projects. It's known for its user-friendly graphical interface that makes it easy to create and manage databases without extensive programming knowledge. As part of the Microsoft 365 suite, it combines a relational database engine with tools for creating **tables, forms, reports, and queries**.

Access is often used for standalone applications or for small team collaborations, but it has limitations on file size and concurrent users, which makes it less suitable for large-scale enterprise solutions.

---

## Examples of Other Database Softwares

Database software can be categorized into two main types:

1. Relational database
2. Non-relational database (NoSQL).

### Relational Database

These systems store data in tables with predefined schemas and enforce relationships between them.

- **MySQL:** A popular open-source relational database widely used for web applications. It is known for its speed and reliability.
- **PostgreSQL:** An advanced open-source relational database known for its stability and extensive features.
- **SQL Server:** Microsoft's enterprise-level relational database, designed for large volumes of data and complex transactions.
- **Oracle Database:** A powerful, commercial relational database used by large companies for its high performance, scalability, and security.
- 

### Non-Relational (NoSQL) Database

These systems are more flexible and are designed to handle unstructured or semi-structured data, often at a large scale.

- **MongoDB:** A popular document-oriented database that stores data in flexible, JSON-like documents.
- **Cassandra:** An open-source, distributed database designed to handle massive amounts of data across many servers, ensuring high availability.
- **Redis:** An open-source, in-memory data store used as a database, cache, and message broker, known for its extreme speed.

## Core Terminologies

- **Data:** Raw facts and figures.
  - *Example:* The number "25", the name "John Smith", and the address "123 Main St." are all pieces of data.

- 
- **Tables:** The main structures for storing data, organized into rows and columns.
  - *Example:* A "**Students**" table with columns for **StudentID**, **Name**, and **Major**
  -
- **Fields:** The columns in a table, representing a specific attribute.
  - *Example:* In the "**Students**" table, **StudentID**, **Name**, and **Major** are the fields.
  -
- **Records (or Tuples):** The rows in a table, representing a single instance of an entity.
  - *Example:* A single row in the "**Students**" table containing data for one student, such as (**101**, "**Alice**", "**Computer Science**").
  -
- **Views:** A virtual table created from an SQL query, used to simplify complex data retrieval.
  - *Example:* A view named "**RecentStudents**" that shows only students who enrolled in the last year.
  -
- **Forms:** A graphical interface for entering, editing, and viewing data.
  - *Example:* A form with text boxes to input a new student's name and ID.
  -
- **Queries:** A request for data from a database.
- - *Example:* The SQL query `SELECT Name, Major FROM Students WHERE StudentID = 101;` to retrieve a student's information.
  -
- **Schema:** The logical structure of the database.
  - *Example:* A blueprint of a database showing the "**Students**", "**Courses**", and "**Enrollment**" tables and how they are related.
- **Forms:** Used to enter, edit, and view data in a user-friendly interface. They can be customized to make data entry easier and more efficient.
  - *Example:* A data entry form for a library could have fields for a book's title, author, and publication date. This form simplifies the process of adding new books to the database without needing to directly interact with the table.
  -
- **Reports:** Used to present data in a formatted, professional way for printing or viewing. They can summarize and analyze data from your tables and queries.
  - *Example:* A report could be generated to show a list of all books checked out in the last month, organized by due date and formatted for printing.
  -
- **Auto Wizard:** This is a feature that guides you step-by-step through a process, such as creating a form or report, to simplify complex tasks.
  - *Example:* The Form Wizard can walk you through selecting the fields you want to include in a new form and automatically arrange them into a layout.
  -
- **Design View:** This is a workspace that allows you to create or modify the structure of database objects, such as tables, forms, queries, and reports. It gives you full control over the design and layout.

- *Example:* In **Design View** for a table, you can manually add, remove, or change fields and set their data types (e.g., Number, Text, Date/Time). For a form, you can drag and drop fields, add buttons, and change the colors and fonts to create a custom layout.
- 

## Keys

- **Primary Key:** A field that uniquely identifies each record and cannot be empty.
    - *Example:* **StudentID** in the "**Students**" table.
    -
  - **Foreign Key:** A field in one table that links to the **primary key** of another table.
    - *Example:* In an "**Enrollment**" table, the **StudentID** field would be a **foreign key** referencing the "**Students**" table's **primary key**.
    -
  - **Candidate Key:** An attribute or set of attributes that could serve as a **primary key**.
    - *Example:* In a "**Students**" table, both **StudentID** and **SocialSecurityNumber** could be candidate keys.
    -
  - **Composite Key:** A key made up of two or more attributes.
    - *Example:* In a concert ticket table, a **composite key** might be (**ConcertID**, **SeatNumber**).
- 

## Relationships

- **Relationship:** A connection between tables.
    - *Example:* A logical connection between a "**Students**" table and a "**Courses**" table.
    -
  - **One-to-One Relationship:** One record is linked to only one other record.
    - *Example:* A "**Users**" table and a "**UserProfiles**" table. Each user has one profile.
  - **One-to-Many Relationship:** One record is linked to many records in another table.
    - *Example:* A "**Departments**" table and an "**Employees**" table. One department can have many employees.
    -
  - **Many-to-Many Relationship:** Multiple records in one table can be linked to multiple records in another, typically using a **linking table**.
    - *Example:* A "**Students**" table and a "**Courses**" table. One student can take many courses, and a course can have many students. This is resolved by an "**Enrollment**" **linking table**.
- 

## Integrity and Constraints

- **Entity Integrity:** A rule that the **primary key** cannot contain empty values.
  - *Example:* The **StudentID** field must always have a value and cannot be empty.
  -
- **Referential Integrity:** A rule that a **foreign key** value must match an existing **primary key** value in the referenced table.
  - *Example:* In an "**Enrollment**" table, you cannot add a record with **StudentID = 999** if that ID does not exist in the "**Students**" table.
  -
- **Data Integrity:** The overall accuracy and consistency of data.

- *Example:* Ensuring all phone numbers in a "**Contacts**" table follow a consistent format.
- 

## Additional Terminologies

- **Report:** A formatted summary of data from tables.
  - *Example:* A weekly sales report showing sales by product and region.
  -
- **Index:** A data structure that speeds up data retrieval.
  - *Example:* An **index** on the **LastName** field to quickly find all contacts with a last name of "Jones".
  -
- **Normalization:** The process of organizing data to reduce redundancy.
  - *Example:* Breaking a single table with **EmployeeName**, **DepartmentName**, and **DepartmentManager** into two separate tables: one for "**Employees**" and one for "**Departments**".
- **SQL** (*Structured Query Language*): The standard language for managing relational databases.
  - *Example:* `UPDATE Employees SET Salary = 55000 WHERE EmployeeID = 123;` to change a specific employee's salary.