Programming Fundamentals

A program is a set of instructions executed by a computer to perform a specific task. These instructions are written in binary (1s and 0s) because it's the simplest and most efficient way for computer hardware to process data and instructions. However, binary programs created for one type of computer may not work on another due to differences in hardware architecture and binary compatibility.

Computers use 1s and 0s (binary) because it's the fastest and simplest way for hardware to process instructions and data. Binary aligns with how computer chips physically operate, making it efficient for both computation and control.

HIGH-LEVEL VS LOW-LEVEL PROGRAMMING LANGUAGES

- **Low-Level Languages** (e.g., Assembly):
  - Close to machine code
  - Platform-specific
  - Fast and efficient, but hard to read and write
  - Used when hardware-level control is critical (e.g., device drivers, embedded systems)
- **High-Level Languages** (e.g., Python, Java, C#):
  - Closer to human language
  - Easier to read, write, and debug
  - Portable across different platforms
  - Focuses on *what* needs to be done rather than *how* to do it

SPECIALIZED VS GENERAL-PURPOSE LANGUAGES

- **Specialized Languages** (e.g., HTML, SQL):
  - Designed for a narrow domain/task
  - Optimized for specific purposes
  - Not suitable for broader use
- **General-Purpose Languages** (e.g., Java, C++, Python):
  - Can be used to build a wide variety of applications
  - More flexible, but not as optimized for any one specific task

COMPILER VS INTERPRETER

- **Compiler:**
  - Translates entire code at once
  - Produces optimized machine code
  - Faster execution, but platform-dependent
  - Example: C, C++, Java (compiled to bytecode)
- **Interpreter:**
  - Translates and executes code line-by-line

- Easier for testing, debugging, and learning
- Slower and less optimized
- Example: Python, JavaScript

◆ **REAL-WORLD APPLICATION**

- Most applications today are built using **multiple languages**:
  - Frontend: HTML (markup), CSS (style), JavaScript (logic)
  - Backend: Java/Python/C# for logic, SQL for database queries
  - Native code: C/C++ for performance-intensive modules (e.g., browser engines)

◆ **Final Thoughts:**

- The **choice of language** depends on:
  - The **problem you're solving**
  - The **performance requirements**
  - The **platforms you're targeting**
  - Your **team's expertise**
- Learning both **low- and high-level**, as well as **general-purpose and specialized languages**, makes you a more **versatile and effective** programmer.

◆ **ELEMENTS OF A COMPUTER PROGRAM:**

- A program consists of:
  - **Data**: Information the program uses.
  - **Logic**: Instructions that operate on the data.
  - **Flow**: Sequence in which instructions and data are processed.

◆ **ANALOGY: CONVEYOR BELT WITH BOXES**

- Think of the program as a **conveyor belt** with **boxes**:
  - Some boxes contain **data**, some contain **instructions**, some contain **both**.
  - The **CPU** picks up each box one at a time, reads what's inside, and acts accordingly.

◆ **STORAGE (MEMORY) AND ACCESS:**

- Data and instructions need to be **stored in memory** (like shelves).
- The CPU retrieves items from memory to process, similar to gathering ingredients while following a **recipe**.

◆ **RECIPE ANALOGY:**

- **Ingredients** = Data
- **Instructions** = Logic
- **Steps** = Flow of execution
- Everything must be stored and accessed when needed.

- **MEMORY ADDRESSES VS. VARIABLES:**
  - Low-level languages require **manual memory address management** (complex).
  - High-level languages use **variables**:
    - A **variable** is a **human-readable name** that points to a memory location.
    - Example: price = 1299, instead of remembering the exact address where 1299 is stored.
- **CUPBOARD ANALOGY FOR VARIABLES:**
  - Like **labelling boxes in a kitchen**, variables help us quickly find and use data.
  - Variable is a named label through which you can get access to where that label is pointing to. What location and memory you're talking about.
- **FUNDAMENTALS OF PROGRAMMING LANGUAGE SYNTAX AND STRUCTURE**
  - **Tokens**: The basic building blocks of a programming language.
    - **Identifiers**: Custom names created by the programmer (e.g., price, title).
    - **Keywords**: Reserved words defined by the language (e.g., if, let) — not customizable.
    - **Separators**: Punctuation that provides structure (e.g., ;, {}, ()) — similar to punctuation in natural language.
    - **Operators**: Symbols that perform operations (e.g., =, +, >, *) — vary across languages.
    - **Literals**: Fixed values like numbers, strings, or boolean values (42, "hello", true).
    - **Comments**: Non-executable text for human readers, explaining the code.
  - **Syntax Importance**:
    - Syntax ensures **unambiguous instructions** for both **computers and humans**.
    - Computers convert human-readable code into machine instructions — thus, **precise syntax** is essential.
    - Like legal contracts in human language, programming languages require strict structure and rules.
  - **Human and Machine Understanding**:
    - Programming languages are designed to be readable by humans but interpretable by machines.
    - The structure evolved to make code both **verifiable** and **repeatable**.
  - **Purpose of Programming**:
    - All programs fundamentally involve:
      - **Calculations**
      - **Testing conditions**
      - **Making decisions**
      - **Executing steps based on logic**
  - **Programs Vary Widely**:

    - Programs can perform various tasks — from business logic to scientific computations.
    - Complexity varies (e.g., simple calculators to sophisticated simulations), but all use the same basic elements.
- **Code Maintenance**:
    - Comments are crucial for future reference, especially when revisiting code after a long time.
- **Programming as Construction**:
    - Writing a program is like **building with bricks** — whether it's a simple shed or a Taj Mahal, the foundational components remain the same.

🔹 SEQUENTIAL EXECUTION

- Programs execute instructions **step-by-step** by default, in a **sequential** manner.
- Instructions are often terminated using a **semicolon (;)**, but some languages use **new lines** or **indentation**.

🔹 INSTRUCTION TERMINATION

- Syntax for ending instructions varies:
    - **Semicolon (;)**: Languages like C, Java.
    - **New Line**: Languages like Python.
    - Purpose: Helps the computer identify where one instruction ends and another begins.

🔹 HISTORICAL CONTEXT

- Earlier, **machine time was expensive**, so programmers optimized code heavily.
- Today, **developer time is more valuable**, so readability and maintainability are prioritized over compactness.

🔹 CODE STRUCTURE & BLOCKS

- As programs grow, they need to be structured like books: chapters (functions), paragraphs (blocks), etc.
- Helps with:
    - **Readability**
    - **Error detection**
    - **Code reuse**
    - **Team collaboration**

🔹 BLOCKS IN PROGRAMMING

- A **block** is a set of grouped instructions.
- Different languages define blocks differently:
    - **begin / end** (Pascal)
    - **Curly braces {}** (C, Java)
    - **Indentation** (Python)

- Indentation-based syntax improves readability but can lead to **bugs** if spaces are misaligned.

CONTROL FLOW

- **Conditional branching**:
  - **if-else**: Executes code based on condition.
  - **switch-case**: Jumps to a specific code block based on a variable's value.
- **Loops**:
  - Repeat a block of code until a condition is met.
  - Types: while, do-while, for.

MODERN PROGRAMMING BENEFITS

- Code can be modularized and **divided among team members**.
- Structured code allows for **collaboration**, **reuse**, and **maintainability**.
- Emphasis today is on writing **readable** and **manageable** code rather than ultra-efficient code.

KEY CONCEPT: ORGANIZING CODE WITH FUNCTIONS

- **Beyond Basic Structures**: Beyond if-else statements and loops, code organization involves defining reusable operations.
- **Terminology**: Terms like *function*, *procedure*, *method*, *routine*, *subroutine*, *subprogram* are essentially synonymous—each refers to a named, reusable unit of code that performs a task.

FUNCTIONS AND THEIR PURPOSE

- **Named Block**: A function is a block of code given a name (e.g., CalculateDiscount) to encapsulate a task.
- **Parameters**: Functions take inputs (parameters or arguments) which they use for internal computations.
- **Reusability**: The same function can be called multiple times with different arguments to perform the same logic.
- **Return Values**: Functions often return a result (e.g., the calculated discount).

BENEFITS OF USING FUNCTIONS

- **Consistency**: Helps ensure consistent logic across the program, especially important for business rules.
- **Readability**: Improves code clarity by encapsulating logic under descriptive names.
- **Collaboration**: Prevents different developers from implementing the same logic differently.

PROGRAMMING LANGUAGES AND SYNTAX

- **Hello World Comparison**: The same simple task ("Hello World" output) looks different in various programming languages (e.g., Ada, C, Java, Basic, COBOL, Fortran, JavaScript).
- **Syntax Variations**: Different languages have different syntax and keywords but share common logical structures.
- **Core Idea**: Once the syntax of a language is understood, understanding code logic becomes easier across languages.

ORGANIZING PROGRAM COMPONENTS

- **Data and Instructions**: Can be stored separately or together. The choice depends on the desired program design.
- **Example**: Instructions might say "add the next two values and store the result." This assumes data is flowing separately (e.g., on a conveyor).
- **Flexibility**: There's no single correct way—depends on context and programming goals.

DATA HANDLING APPROACHES

- **Modify Original Data**: Instructions can change the actual stored data.
- **Return Modified Copy**: Alternatively, instructions can generate a modified copy and leave the original unchanged.
- **Trade-Offs**: Choice impacts repeatability, side effects, and clarity of program behaviour.

INSTRUCTION TYPES

- **Process-Oriented Instructions**: Define *how* to do something (step-by-step).
- **Result-Oriented Instructions**: Define *what* you want as the outcome, leaving the "how" to the system.
- **Example**: Assembling a chair—either describe the assembly steps (process) or just describe the final product (result).

PROCEDURAL PROGRAMMING

- **Structure**: Contains procedures (named code units) that take input data (parameters), perform actions, and may modify global or passed-in data.
- **Issue**: Procedures that change input data lead to *non-repeatable behaviour* if called multiple times.
- **Context Dependency**: Outputs depend on the program state at the time of execution.

FUNCTIONAL PROGRAMMING

- **Structure**: Uses functions that do not modify input data but return new values.
- **Advantages**:
  - Predictable and repeatable results.
  - Easier testing and debugging.
  - Enables higher-order functions (functions as parameters or return values).
- **Best Practice**: Avoid side effects in functions to keep them pure.

COMBINING APPROACHES

- Real-world programs often mix **procedures** and **functions** based on suitability.

OBJECT-ORIENTED PROGRAMMING (OOP)

- **Core Idea**: Combines data and operations in a single entity called an object.
- **Encapsulation**: Objects manage their own internal state and expose behaviour via methods.
- **Example**: A Car object has a drive() method that encapsulates the complex internal operations.

- **Benefits**:
    - Code reusability and maintenance.
    - Logical grouping of related data and functions.
    - Simplifies program design by modelling real-world entities.

ENCAPSULATION AND ABSTRACTION

- Developers using an object don't need to understand its internal implementation—just its interface.
- The **developer who writes the object** needs to know the internals, while the **user of the object** treats it as a black box.

MEMORY & DATA TYPES METAPHORS

- **Shelf & boxes metaphor**: Memory is like a shelf, and each box (memory block) must fit the correct shape of data (e.g., text, number).
- **Shape sorter toy metaphor**: Different data types (text, number, date, etc.) must match the "shape" of the memory allocated.

ALL DATA IS BINARY

- Ultimately, all data becomes 1s and 0s at the machine level after compilation.
- High-level languages allow humans to work with more meaningful representations (like numbers and text), which are then translated into binary.

IMPORTANCE OF DATA TYPES

- Data types define how data is stored and what operations can be performed.
- For example, you can't perform arithmetic on the text "3" but can with the number 3.

STRONGLY VS WEAKLY TYPED LANGUAGES

- **Strongly Typed Languages:**
    - Enforce data type rules.
    - Prevent type errors (e.g., mixing text and numbers).
    - Compiler checks help catch mistakes early.
    - Promote code safety, readability, and maintainability.
- **Weakly Typed Languages:**
    - Allow changing a variable's type dynamically.
    - Favor convenience and speed for quick scripts.
    - Higher risk of bugs and harder to debug or maintain.
    - Often used for "throwaway code."

TRADE-OFFS

- **Weak typing**: Faster and more flexible, but riskier and harder to maintain.
- **Strong typing**: More disciplined and safer, especially in team or long-term projects.

SCRIPTING LANGUAGES & TYPING

- Many scripting languages (like JavaScript) are weakly typed.
- **TypeScript** is an example of adding strong typing to a weakly typed language.
- Recent trend: efforts to make scripting languages more structured.

FINAL ADVICE

- Don't blindly follow trends in language choice.
- Understand the needs of your project and choose the appropriate level of typing.

Persistence in Programming

- **Persistence** means saving program data to retain state across sessions.
- Two main methods:
    1. **Files**
    2. **Databases**

USING FILES

- Files are simple, named units of data stored on disk.
- Programmer is responsible for:
    - Reading/writing data to files.
    - Choosing and handling file formats (e.g., XML, JSON, CSV, JPEG).
- Each application must handle its own file I/O and format logic.

USING DATABASES

- **Databases** abstract file storage using a **Database Management System (DBMS)**.
- DBMS handles all read/write operations for the application.
- Applications communicate with the DBMS via **queries**.

RELATIONAL DATABASES (RDBMS)

- Store data in **tables** (structured in **rows and columns**).
- **Relational** = Table-based storage.
- Use **SQL (Structured Query Language)** for data operations:
    - **SELECT**, **INSERT**, **UPDATE**, **DELETE**
- SQL is human-readable and accessible to non-programmers (e.g., analysts).

BENEFITS OF DATABASES

- Abstract away low-level storage handling.
- Enable complex queries and data retrieval using simple, English-like SQL.
- Useful for structured and well-defined data models.
- Popular RDBMS examples: Oracle (one of the earliest and most performant).

🔹 DATA MODELING

- Relational DBs require **predefined schema** (tables and columns must be planned).
- Involves data analysis and structure design before storage.
- A separate skill set from programming — often handled by **data analysts or designers**.

🔹 REQUIREMENTS GATHERING

- Understand what kind of application you're building (e.g., restaurant POS vs. missile control).
- Collect and document specific functional and non-functional requirements.
- Get sign-off from stakeholders to ensure mutual understanding.

🔹 APPLICATION DESIGN & DEVELOPMENT

- Decide on data structures, logic, and features based on requirements.
- Start coding only after requirements and designs are clear.
- Package, compile, and deploy the application to the target environment (e.g., mobile, embedded system, POS, etc.).

🔹 MAINTENANCE AND SUPPORT

- Work doesn’t end after deployment.
- Applications need ongoing support for:
    - **Bug fixing**
    - **Feature enhancements**
    - **Performance improvements**

🔹 DOCUMENTATION

- Crucial at every stage:
    - Requirements documentation
    - Source code comments and technical documentation
    - Support and deployment guides
    - End-user manuals and training materials
- Good documentation improves collaboration and maintainability.

🔹 VERSION CONTROL

- Tracks changes in code, documents, and configurations.
- Supports:
    - Rollbacks to previous versions
    - Parallel development (branching and merging)
    - Release/version management
- Enables collaboration and experimentation safely.

- **TESTING**
  - Two key types:
    - **Unit Testing** – Testing individual functions or modules.
    - **System Testing** – Testing the application as a whole.
  - Ensures performance, correctness, usability, and expected behaviour.
  - Helps detect errors, crashes, and UI/UX issues.
- **DEBUGGING**
  - Analysing and fixing bugs after they are discovered in testing.
  - Requires understanding program flow and memory/data states.
  - Often bugs emerge only when functions interact, not in isolation.