Turiba University
Ekhlas Rayees Khan

HISTORY OF PROGRAMMING

PROFESSIONAL BACHELOR THESIS

Study Programme: Computer Systems

Author: **Ekhlas Rayees Khan**

***Thesis* Advisor:** **Janis Peksa**

Riga, 2025

The Pre-Electronic Concept (1800s)

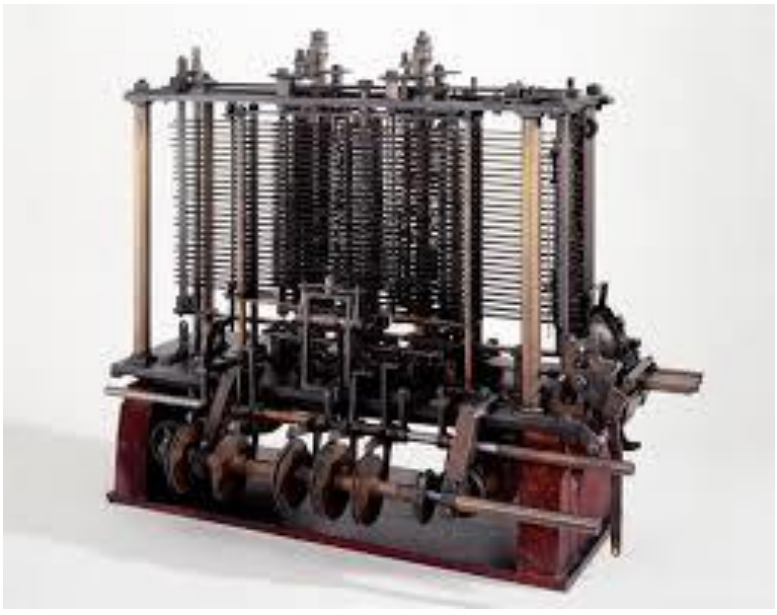
The idea of the stored program preceded electronic computation, defining the concept of instructions separate from the machine's task.

1. ### Programmable Mechanism:

   The Jacquard Loom used **punched cards** to store weaving patterns, demonstrating that complex mechanical actions could be controlled by external data.

2. ### Algorithmic Foundation:

   Ada Lovelace formalized an algorithm for Charles Babbage's conceptual Analytical Engine. She is recognized as the first programmer for writing instructions intended for a general-purpose machine.

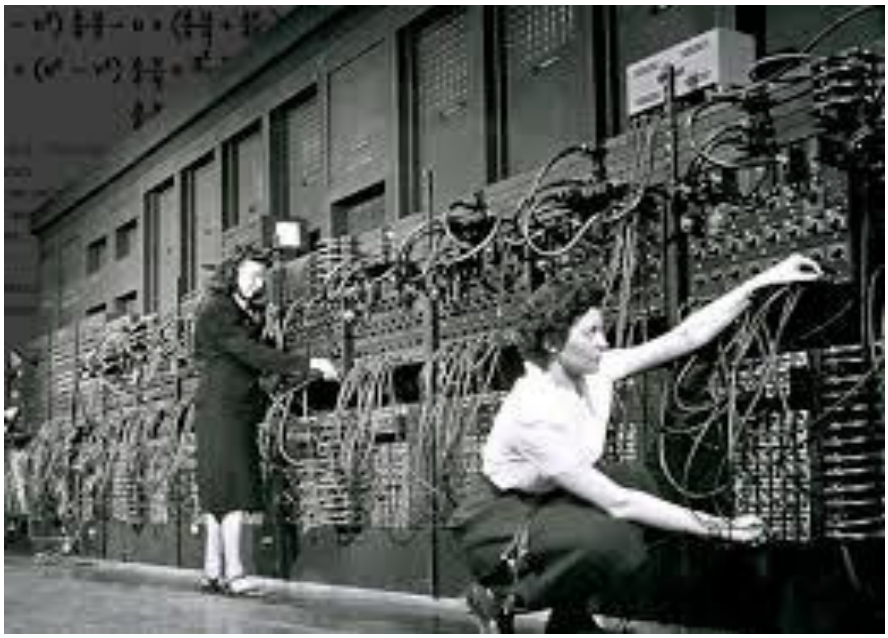


---

3. The Dawn of Electronic Code (1940s – Early 1950s)

Early, computers were difficult to program, necessitating the invention of the first translator.

1. **Direct Input:** Initial programming involved physically reconfiguring wires and switches, followed by direct binary input called **Machine Code**, which was prone to errors and very slow.
2. **Symbolic Relief:** The invention of the **compiler** (by Grace Hopper, among others) allowed for the creation of **Assembly Language**. Assembly used short, mnemonic codes (e.g., ADD) that were then automatically translated into machine code.



3. First High-Level Languages (Late 1950s)

These languages bridged the gap between human language and machine instructions, enabling widespread application.

1. **Scientific Computation: FORTRAN (1957)** was developed by IBM for scientific and engineering calculations. It prioritized efficiency for mathematical formulas.

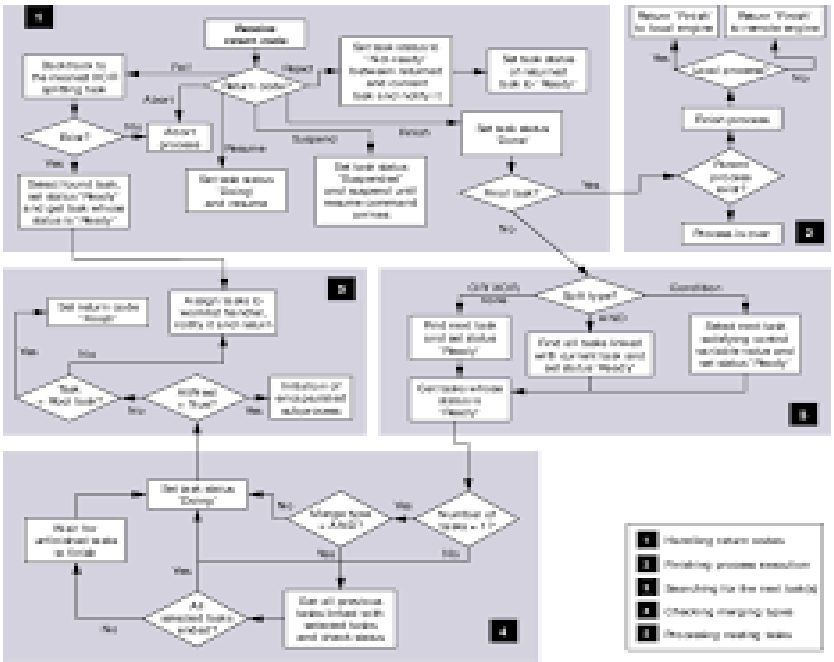
2. **Business Logic: COBOL (1959)** was designed for finance and administrative data processing. Its verbose, English-like syntax was intended to make programs readable by non-technical managers.

3. Structured Programming and System Languages (1970s)

This era focused on improving code organization and creating powerful language for operating systems.

1. **Control Flow Management:** The push for **Structured Programming** replaced unstructured jumps (like the GOTO statement) with logical structures (loops, if/else), significantly improving code reliability and maintenance.

2. **The OS Foundation:** The **C language (1972)** provided a powerful, efficient, medium-level interface. Its success was cemented when it was used to rewrite the **UNIX** operating system, establishing it as the standard for systems programming.

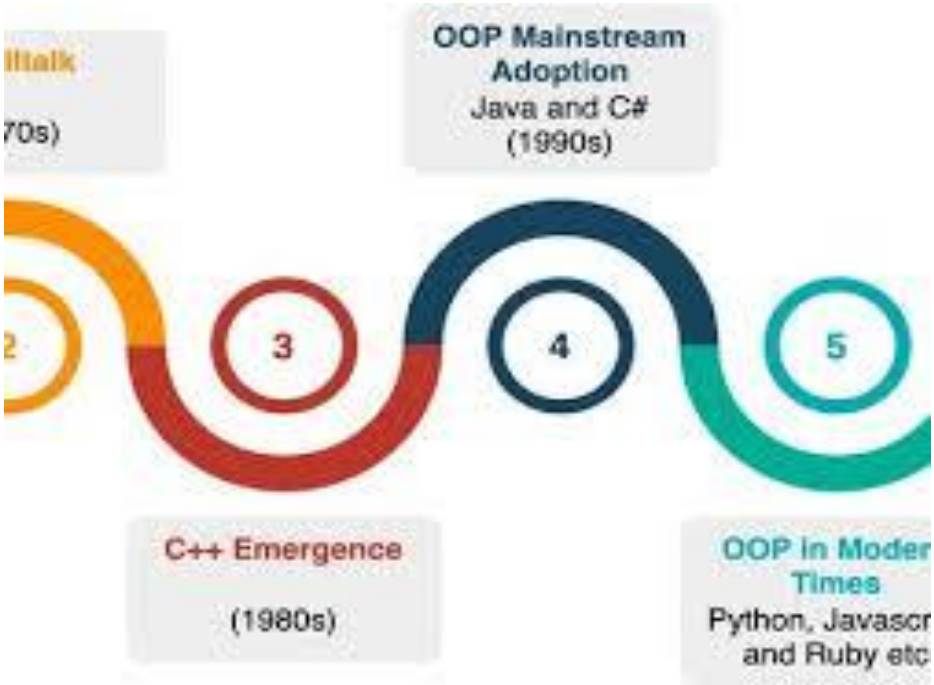


5. The Object-Oriented Paradigm (1980s)

To handle increasingly large and complex software, a new organizational model was required.

1. **Encapsulation: Object-Oriented Programming (OOP)** introduced the concept of "objects" that bundle data and the functions (methods) that operate on that data, improving modularity.

2. **Hybrid Dominance: C++ (1983)** added OOP capabilities (classes) to the C language base. Its combination of performance and structural complexity made it dominant for large applications, such as operating systems and game engines.



6. The Internet and Cross-Platform Needs (1990s)

The rise of the World Wide Web created demand for platform independence and client-side execution.

1. **Platform Independence: Java (1995)** used the Java Virtual Machine (JVM) to achieve its goal of "Write Once, Run Anywhere," becoming the standard for large-scale enterprise server applications.

2. **Client-Side Scripting: JavaScript (1995)** was initially designed to run simple scripts inside the web browser. It evolved rapidly to become the primary language for interactive front-end development and later, server-side (Node.js).

3. **Readability Focus: Python (1991)** gained momentum due to its clean syntax and readability, growing from a scripting tool into the primary language for scientific computing and data analysis.

7. Modern Specialization and Automation (Present)

The current trend involves creating languages with highly specific design goals and integrating AI into the coding process.

1. **Safety and Concurrency:** Languages like **Rust** and **Go** were designed to solve modern problems like memory management safety and efficient multi-core processing, offering modern alternatives to C++.

2. **Assisted Development:** The integration of AI tools, such as generative code assistants, represents the latest stage of abstraction, where the programmer guides an **AI** to write and debug significant portions of the code.