FORM TWO COMPUTER SCIENCE CHAPTER THREE COMPUTER PROGRAMMING

Meaning of computer programming

Programming **is the process of creating a set of instructions that a computer can follow to perform specific tasks or solve problems.**

It involves writing code in a language the computer can understand and execute. In simple terms, programming is how humans communicate with computers to make them perform desired actions.

Computer programming **is the process of writing instructions that a computer can understand and execute.**

These instructions are written in a programming language and tell the computer what to do, step by step, to complete a task or solve a problem.

Real life example:

Imagine you are making ugali. You need to follow a series of steps in a specific order to make it successful.

The instructions for making ugali would be like programming:

Step 1: Boil water in a pot.

Step 2: Gradually add maize flour (cornmeal) while stirring.

Step 3: Continue stirring until the mixture thickens.

Step 4: Cook until the mixture is firm and no longer sticky.

These steps must be followed precisely, just like writing code in a program. If you skip a step or get the order wrong, you might end up with a different result (such as, watery ugali or burnt flour). Similarly, in programming, if you do not follow the correct syntax or sequence of instructions, the program will not run as expected, and you will get errors or incorrect results.

Reasons for learning programming

(i) From smartphones and smart home devices to websites and business systems, **programming is the foundation of most modern technologies.** Learning to program helps you understand and shape the digital world around you.

(ii) **Creativity and problem-solving:** Coding allows you to create apps, games, and software solutions.

(iii) **High-demand careers:** Programming skills are in high demands across almost every industry or organization, open job opportunities in many industries, including technology, healthcare, education, entertainment, finance and many others.

(iv) **Tasks automation:** Programming can help in automating repetitive tasks to improve efficiency and accuracy. For example, programming can save time of manual effort in various activities such as data processing, file management, and many others.

(v) **Cross-industry Skills:** A person that learn how to program, can work in a variety of fields, including web development, mobile app development, artificial intelligence, machine learning, game development, data science, and more.

Importance of programming

(i) Development of Daily Technology: Programming helps in developing technologies used every day. *Example:* Mobile money systems like **M-Pesa, Tigo Pesa, and Airtel Money** in Tanzania rely on programming to allow people to send and receive money easily.

(ii) Solving Real-Life Problems: Programming is useful for solving real-life challenges in society. *Example:* Systems used by **TANESCO** for billing and tracking electricity usage are built using programming to reduce errors and improve service delivery.

(iii) Opportunities: Learning programming opens doors to good careers and self-employment. *Example:* Many Tanzanian youths work as **software developers, website designers, orappdevelopers**, creating apps for local businesses and schools.

(iv) Understanding How Computers Work: Programming helps people understand how computers and technology operate. *Example:* Students in Tanzanian colleges like **DIT and UDSM** learn programming to understand how systems such as student registration portals function.

(v) Increasing Efficiency in Industries: Programming makes work faster and easier in different industries. *Example:* **Banks in Tanzania**, such as NMB and CRDB, use programmed systems for ATM services and online banking.

(vi) Reducing Errors and Improving Accuracy: Programming reduces human errors by automating tasks. *Example:* **NECTA examination results processing** uses computer programs to ensure accurate marking and results management.

(vii) Supporting Development in Advanced Fields: Programming supports progress in areas like artificial intelligence, healthcare, and data analysis. *Example:* Hospitals in Tanzania use **electronic health record systems** to store and analyze patient information.

(viii) Enabling Global Communication: Programming makes global communication possible through the Internet and social media. *Example:* Tanzanians use **WhatsApp, Facebook, and Zoom** to communicate locally and internationally for business and education.

(ix) Creating and Customizing Software: Programming allows people to create and adjust software for specific needs. *Example:* Local developers create **school management systems** for Tanzanian primary and secondary schools.

(x) Supporting Science and Education: Programming is important in scientific research and education. *Example:* Tanzanian universities use **online learning platforms** and research software developed through programming.

Role of programming languages

Programming languages are essential tools in software development, enabling developers to create, modify, and maintain software applications.

The roles of programming languages in software development include:

(i) Allowing developers to write step-by-step commands that instruct computers to perform specific tasks efficiently.
(ii) Enabling the creation of various types of software, including web applications, mobile apps, desktop programs, operating systems, and enterprise solutions.
(iii) Automating repetitive processes increases productivity and efficiency in healthcare, finance, and manufacturing industries.
(iv) Aiding in software security by preventing vulnerabilities, mitigating cyber threats, and ensuring system stability and data privacy.
(v) Enabling developers to write high-performance code that optimises software efficiency, reduces processing time, and improves responsiveness.

Understanding Computers and Programming: From Hardware to Code

A computer functions because of a combination of hardware and software. Hardware carries out the tasks, while software provides instructions to follow. Together, they allow the computer to perform different functions and respond to user commands.

Hardware (the physical part)

Hardware is the part of a computer you can see and touch. The main parts of the hardware are:

(i) **CPU (Central Processing Unit)** – This is the computer's brain that follows instructions in programs.

(ii) **RAM (Main Memory)** – This stores data that the computer uses currently. It is fast but temporary.

(iii) **Storage (Hard Drive or SSD)** – This saves data for later use, such as photos or schoolwork.

(iv) **Input devices** – This helps you give commands to the computer, like a keyboard or mouse.

(v) **Output devices** – This shows results, like a monitor or printer.

Software (the invisible part)

Software is a collection of instructions or programs that direct a computer's hardware to perform specific tasks. It includes the programs and apps that run on the computer. There are two main types:

(i) **System software** – Helps the computer run, like Windows or macOS.

(ii) **Application software** – Lets you do tasks like typing, drawing, or browsing the Internet.

There are also tools like virus scanners and backup programs that help keep the computer safe and fast.

How computers store data

Computers use binary, which is just 0s and 1s, to store everything—words, pictures, music, and more.

(i) A "bit" is a single 0 or 1.

(ii) A "byte" is 8 bits, and can store one small piece of information, like a letter.

For example:

(i) The letter "A" is stored as 01000001

(ii) The number "157" is stored as 10011101

Bigger numbers need more bytes. Computers can combine bytes to store large values like 65,535 or even more.

Storing text, numbers, and media

(i) Texts use systems like ASCII and Unicode to convert letters into binary.

(ii) Images are made of tiny dots called pixels, each stored as numbers (iii) Sound is saved in small pieces (samples), also stored as numbers.

How a program works

When you run a program, it consists of many small instructions that the computer must follow. These instructions are executed by the CPU (Central Processing Unit), which is often referred to as the 'brain' of the computer.

How the CPU follows instructions

The CPU does simple things like:

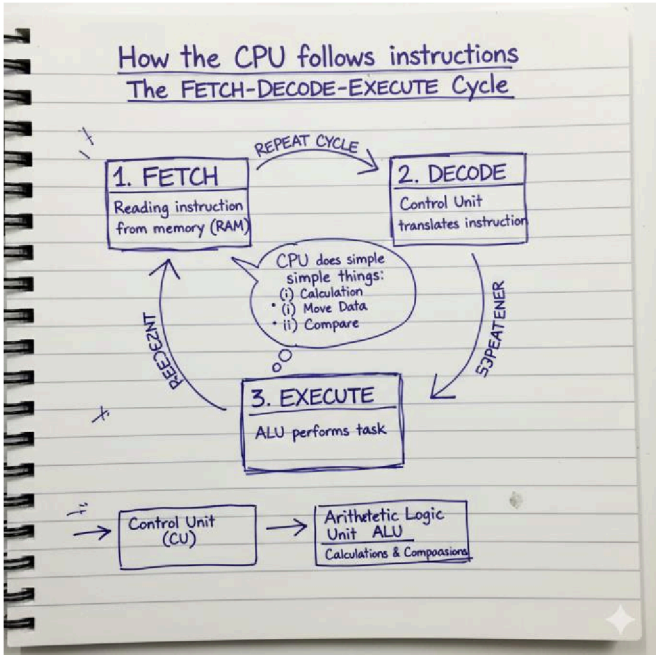
(i) Reading from memory

(ii) Doing calculation

(iii) Moving data

(iv) Comparing numbers

By doing many small steps very fast, it can complete large tasks.



How programs are written

(i) Computers only understand machine language, which is a long list of 0s and 1s.

(ii) Writing in machine language is hard, so we use assembly language, which uses easy words like:

ADD (add numbers)

MOV (move data)

(iii)A program called an assembler changes assembly language into machine language so the CPU can understand it.

Where programs are stored

Programs are saved on hard drives. When you open them, they are copied to RAM. The CPU then fetches and executes instructions directly from RAM during processing.

Generally, programming is done by following a few simple steps. The basic process of programming includes:

(i) Writing a list of instructions (this is called a program).

(ii) Saving the program on the computer.

(iii) Running the program, which loads it into the computer's memory (RAM).

(iv) The CPU reads and follows the instructions to perform the task.

Important terms in programming

Understanding the basic terms used in programming is essential for anyone learning to code. These terms form the foundation of how programs are written, understood, and executed. Some of the most important programming terms include:

Compiler

A compiler is a special software program that translates code written in a high-level programming language (such as C, Java, or C++) into machine language (binary code) that a computer's CPU can understand and execute.

It processes the entire source code at once, converting it into object code or an executable file. This allows the program to run directly on the computer hardware without needing the source code during execution.

Compared to interpreters, compilers typically require less time during execution, as the code is already fully compiled.

The compilation process generally involves several stages, including compiling, linking with system libraries, and executing the program. Common examples of compilers include Dev-C++, Borland C++, and Visual C++.

To make a computer execute your program, you need a compiler.

The importance of using a compiler includes:

(i) Translating source code into machine language – Computers cannot understand English-like C code, so the compiler converts it into binary (machine code).

(ii) Checking for errors – The compiler detects and reports mistakes in your code, helping you correct them before running the program.

(iii) Creating an executable file – After compilation, the compiler generates a file that the computer can run directly.

below is the table of Example of compilers for different programming languages

Programming Language	Compilers
C	GCC (GNU Compiler Collection), Clang, Microsoft Visual C++ (MSVC), Turbo C
C++	G++ (part of GCC), Clang++, Microsoft Visual C++ (MSVC), Borland C++

Python	Cython, PyInstaller, Nuitka
JavaScript	Babel, Google Closure Compiler, TypeScript Compiler (tsc)
Java	Java Compiler (javac), Eclipse Compiler for Java (ECJ), GraalVM Compiler
PHP	HipHop Virtual Machine (HHVM), PHP Zend Engine, RoadRunner
Swift	Swift Compiler (swiftc)
Kotlin	Kotlin Compiler (kotlinc), IntelliJ IDEA Kotlin Compiler
Go	Go Compiler (gc), GCC Go
Rust	Rust Compiler (rustc)
Pascal	Free Pascal Compiler (FPC), Delphi Compiler
Turbo Pascal	Turbo Pascal Compiler
Fortran	GNU Fortran (gfortran), Intel Fortran Compiler, IBM XL Fortran
R	R Compiler (Bytecode Compiler), GCC for R
COBOL	GnuCOBOL, IBM COBOL, Micro Focus COBOL
Haskell	Glasgow Haskell Compiler (GHC), Haskell Platform

Interpreter An interpreter is a type of translator that converts source code into machine (object) code one statement

at a time.

Unlike a compiler, it does not translate the entire high-level program at once. Instead, it processes each line sequentially during execution that takes a bit longer.

Interpreters are often used for programs that require less memory, as translating larger programs line-by-line can be time-consuming.

Common programming languages that use interpreters include Perl, Ruby, Python, and PHP.

Source code

Source code also referred to simply as code, **is a collection of computer instructions written by a programmer using a high-level or intermediate programming language.**

It may include comments to enhance understanding and aid in the maintenance of the program.

Source code can be written in various languages such as C++, Java, Python, PHP, and others. It is typically machine-dependent, meaning it may require modification to run on different hardware or operating systems.

An example of source code written in C++ is:

```
#include
<iostream> int main()
{
    std::cout << "What is your name?";
return 0;
}
```

Object code

Object code **is a machine-readable code, in a binary format (0s and 1s) that is obtained after converting source code.**

The object code is also known as machine code because it can only be read and understood by the processor.

Translators

Translators are programming language processors used to convert programs into machine-readable language.

They are classified into three main categories: **assemblers, compilers, and interpreters.**

Assemblers Assemblers are language processors used to convert programs written in assembly language into

machine code that a computer can execute directly.

The programming environment: where coding happens

A programming environment **is the space where developers write, test, and run their code. It includes tools that make coding easier and more efficient.**

There are different types of programming environments, depending on how and where coding is done. Below are **two** main types:

(a) **Integrated Development Environment (IDE):** is a software that helps programmers write, test, and fix their code more easily.

It combines several tools into one platform to make coding faster and more organised.

It usually includes a code editor, a compiler or interpreter, debugging tools, and other features to help programmers write and test their code more efficiently.

Examples include Visual Studio, PyCharm, and Eclipse. An IDE usually includes:

(i) Code editor: This is where you write your code. It makes coding easier by highlighting important parts, suggesting possible completions, and making your code look neat.

(ii) Compiler or interpreter: These tools turn the code you write into instructions the computer can understand and run. A compiler does this all at once, while an interpreter does it one line at a time.

(iii) Debugger: This helps you find and fix mistakes in your code. It lets you go through your code step by step to see what is going wrong.

(iv) Build tools: These help you automate common tasks like checking for errors, testing your code, or running your program, making everything faster and easier.

(v) Other helpful things: Some IDEs have tools to help you organise your files, manage projects, and even work with other programmers.

Online code editors

An online code editor is a web-based tool that allows programmers to write and run code directly in a web browser, without installing any software. These editors are great for beginners, students, and people who want to code on the go.

The following are some features of online code editors:

(i) No installation is required. You just open the website and start coding.

(ii) Accessible from any device with an Internet connection.

(iii) Some provide built-in collaboration features for group coding projects.

The following are some examples of online code editors:

(i) **Replit:** Supports multiple programming languages, allows

(ii) **Scratch:** A visual coding platform designed for beginners, especially children, to create animations and games using drag-and-drop blocks.

(iii) **JSFiddle:** A web-based tool for experimenting with JavaScript, HTML, and CSS

PRACTICAL 01

Title: Exploring Online Code Editors

Aim

To introduce and familiarize learners with online programming editors by exploring their interfaces, identifying key areas, and running simple programs.

Materials / Requirements

- Computer or tablet
- Internet connection
- Web browser (Chrome, Firefox, etc.)

- Online code editors: Replit (https://replit.com)

  Programiz (https://programiz.com/python-programming/online-compiler)

  JSFiddle (https://jsfiddle.net)

  Scratch (https://scratch.mit.edu)

Procedure / Steps

1. Open a web browser and connect to the Internet.
2. Visit one online code editor at a time (Replit, Programiz, JSFiddle, or Scratch).
3. Perform the following activities depending on the editor chosen:

a. Replit / Programiz:

i. Type the code:

```
print("Jambo,                                                    Africa!")
```

ii. Click the **Run** button to execute the program.

b. JSFiddle:

i. Select the **HTML** window.

ii. Type the code:

```
<h1>Hello World</h1>
```

iii. Click the **Run** button to display the output.

c. Scratch:

i. Open https://scratch.mit.edu and click **Create**.

ii. From the **Events** category, drag the block **"when green flag clicked"** into the coding area.

iii. From the **Looks** category, drag the block **"say [Hello!] for [2] seconds"** and attach it below the event block.

iv. Click the **green flag** to run the program.

4. Observe the editor and identify the following key areas:

a. Code or blocks area

b. Output or result area

c. Run or start button

Output / Results

- A simple program was successfully created and executed using an online editor.

- The program displayed text output such as **“Jambo, Africa!”** or **“Hello World”**, or made a Scratch sprite speak.
- Key parts of the editor were identified, including the code area, output area, and run/start button.

PRACTICAL 2:

Writing and Running a Simple Program in C

Title: Writing and Running a Simple C Program

Aim: To write, compile, and run a simple program in C and observe its output.

Materials / Requirements

- Computer with C compiler installed (Code::Blocks, Dev-C++, or online C compiler)
- Text editor or IDE

Procedure / Steps

1. Open the C programming editor (Code::Blocks, Dev-C++, or an online C compiler).
2. Type the following C code: #include <stdio.h> int main() {

   printf("Tanzania, my beautiful country!\n"); return 0;

   }

3. Save the file with a **.c** extension, for example: **tanzania.c**
4. Click **Run** or press **F9** to compile and execute the program.
5. Observe the output in the **console area**.

Output / Results

The console displayed:

Tanzania, my beautiful country!

Conclusion

The program successfully printed a message. Writing C programs involves using proper syntax, compiling the code, and running it to see results.

Deliverables

- Screenshot of code area, output, and run/start button
- A simple text-based C program

PRACTICAL 3:

Writing and Running a Simple Program in Python

Title: Writing and Running a Simple Python Program

Aim: To write, save, and run a simple Python program and observe its output.

Materials / Requirements

- Computer with Python installed or online editor (Replit, Programiz, etc.)
- Internet connection (if using an online editor)

Procedure / Steps

1. Open the Python programming editor (online or offline).
2. Type the following Python code:

```
print("Tanzania, my beautiful country!")
```

3. Save the file with a **.py** extension, for example: **tanzania.py**
4. Click **Run** or press **F5** to execute the program.
5. Observe the output in the **terminal/output area**.

Output / Results

The terminal displayed:

Tanzania, my beautiful country!

Text editors

A text editor is where you type and modify your programs. Modern editors provide syntax highlighting, auto-completion, and error detection to improve coding efficiency. Some commonly used text editors include:

(a) Basic editors: Notepad (Windows), Nano (Linux), vi/vim (Linux/macOS).

(b) Advanced code editors: Visual Studio Code (VS Code), Sublime Text, Atom.

(c) Integrated editors in IDEs: Code::Blocks, Dev-C++, CLion.

Libraries and frameworks

Libraries are collections of pre-written code that programmers can use to perform common tasks like connecting to a database or creating a user interface. Frameworks provide a more structured way to build applications, offering a foundation and set of rules to follow.

Debugging tools

These are used to find and fix errors (bugs) in code. Debuggers help programmers step through their code to identify the source of problems.

Introduction to programming languages

A programming language is a formal language comprising a set of instructions that can be used to produce various kinds of output.

It enables humans to communicate commands to a computer, allowing the creation of software programs, control of machine behavior, and expression of algorithms.

Programming languages are essential for creating software, mobile apps, websites, and other digital technologies.

The programming languages are classified into two types: low-level languages and high-level languages as shown in Figure below

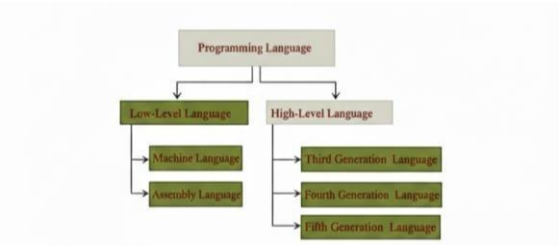


Figure *Programming languages classification*

Low-level languages

Low-level programming languages are characterised by little effort to translate programs into machine- readable form. They are not portable; hence, they are not easily transferred from one computer to another. These languages are machine-dependent and do not require any compiler to translate them into machine code. Examples of low-level programming languages include machine languages and assembly languages.

Types of low-level languages

There are two types of low-level languages: machine and assembly.

Machine language

Computers operate using the binary number system, which means they process and represent data using only two symbols: 0 and 1. Machine language, a low-level language, consists of binary code that is directly processed by the Central Processing Unit (CPU). Since every instruction is written in 0s and 1s, it becomes complicated for humans to understand and use as a primary programming language.

Assembly language

Assembly language is a step above machine language and is closer to how computers operate. It uses symbols and code to represent basic operations such as adding, moving, or comparing numbers. Since each computer has its own architecture, each one has its own set of instructions, making assembly language unique to that specific machine. While assembly language is more advanced than machine language and allows for faster programming, it remains difficult to write and read. To execute a program written in assembly language, it must first be converted into machine language (binary) using a program called an assembler.

Differences between machine language and assembly language

Machine language and assembly language are both low-level programming languages, but machine language is below assembly language in the hierarchy of computer languages. Assembly language includes human-readable commands, such as ADD, SUB, MUL, and DIV, while machine language does not contain any words or even letters.

Table below shows the Differences between Machine Language and Assembly Language

Machine Language	Assembly Language
It is only understood by the computers	It is only understood by human beings, not by computers
Data is represented in binary format (0's and 1's)	Data is represented in binary or hexadecimal
Modifications and error fixing cannot be done in machine language	Modifications and error fixing can be done in assembly language
It is very difficult to use	It is easy to use because some alphabets and mnemonics are used
Execution is faster in machine language because data is already present in binary format	Execution is slower compared to machine language

Advantages of low-level programming languages

(a) Programs designed using low-level memory require less memory to be executed, which means they run faster. (b) Low-level languages do not need compilers or interpreters to change into machine-readable form.

(c) Since a low-level language is closely related to a machine language, it provides direct interaction with the computer's internal memory.

(d) Low-level programming languages provide a conducive environment for utilising the processor.

(e) There is direct communication between hardware parts and low-level programming languages.

Disadvantages of low-level programming languages

(a) Low-level languages require high expertise in programming to design a program.

(b) Programs created with low-level languages are not portable simply because they cannot be transferred from one piece of hardware or software to another.

(c) Low-level programming languages have challenging environments for developing and executing programs.

(d) It is difficult to debug errors in low-level languages

High-level languages

High-level languages are machine-independent, meaning programs written in these languages can run on different computer systems without modification.

They do not interfere with the normal operation of the system. Unlike low-level languages, high-level languages are closer to human language than machine language, making them easier to read, write, and understand. Examples include Pascal, Fortran, COBOL, C, C++, Java, JavaScript, and Visual Basic. High-level languages simplify coding by abstracting hardware complexities.

They are categorized based on their approach, such **as procedural, object-oriented, functional, scripting, and logic-based languages,** each serving different programming needs.

Procedural languages

This type of programming language executes instructions sequentially, step-by-step. It is ideal for structured programming, where programs are organised into procedures or functions to enhance readability and maintainability. Examples include C, Pascal, and Fortran.

Object-Oriented Languages

Object-Oriented programming (OOP) is based on the concept of objects, which encapsulate both data and behaviour. It supports key principles such as encapsulation, inheritance, and polymorphism. These principles help create modular, reusable, and maintainable code in languages like Java, C++, and Python.

Functional languages

Functional languages are programming languages that think of programs like math functions. Instead of changing things step by step, they focus on calculating results by using functions without changing any data along the way. It encourages writing pure functions, enhancing program reliability and minimising side effects. Examples of functional languages include Haskell, Lisp, and F#.

Scripting languages

Scripting languages are typically interpreted rather than compiled and are designed for automating tasks and web development. They are widely used for rapid prototyping and scripting in system administration, data processing, and web applications. Examples of scripting languages are Python, JavaScript, and Perl.

Logic-based languages

Programs in this category are written as sets of logical statements that define relationships and rules rather than sequential instructions. They are commonly used in artificial intelligence and expert systems, where reasoning and inference are essential. An example of a logic-based language is Prolog.

Domain-Specific Languages (DSLs)

Specialised languages are designed for specific tasks rather than general-purpose programming. Markup languages, such as HTML, define the structure and presentation of documents, while query languages, like SQL, interact with databases and retrieve information efficiently. Notable examples of domain-specific languages include HTML (markup) and SQL (database queries).

Advantages of high-level programming languages

(a) High-level languages provide an easy way to recognise programs prone to errors.

(b) High-level languages are machine-independent and portable programming languages that are easily transferred from one device to another.

(c) High-level languages provide a conducive environment for developing, executing, and maintaining a program.

(d) High-level languages provide an easy way to learn and use them.

(e) Since high-level languages run on different platforms, they are widely used for programming.

Disadvantages of high-level programming languages

(a) The program, which is developed in a high-level language, must be interpreted or compiled to be changed into a machine-readable form.

(b) They are not fast, as changing a program from source code to object code requires time.

(c) They overload the processor with several typed statements, slowing the computer's processing speed. (d) They need enough memory to be maintained and executed. Since high-level languages are not written in machine code, they have no direct interaction with the hardware device.

Table below explains the differences between high-level and low-level languages.

Feature	High-Level Language	Low-Level Language
Level of abstraction	It provides a high level of abstraction from the hardware	It provides less or no abstraction from the hardware
Execution speed	It is slower than a low-level language	It is faster than a high-level language
Memory efficiency	It consumes more memory	It consumes less memory
Translation	It requires a compiler or an interpreter to convert the program into machine code or execute it	It requires an assembler to convert the program to machine code
Comprehensibility	It is easily understandable by the programmer	It is easily understandable by the computer
Machine dependency	It is machine-independent	It is machine-dependent
Portability	It is portable	It is not portable
Debugging and maintenance	Easy to debug and maintain	t is hard to debug and maintain
Examples	Python, Java, C++, Ruby	Assembly language, machine code

Generations of programming languages

Programming languages are classified into five generations based on their level of abstraction and capabilities. Low-level languages belong to the first and second generations, while high-level languages are categorised into the third, fourth, and fifth generations. Together, these make up the five generations of programming languages.

First Generation (machine language)

Instructions in machine language are written using binary digits 0s and 1s. These instructions represent elementary operations, such as adding or subtracting numbers. Since computers natively understand binary code, programs written in machine language do not require further translation. However, machine language is difficult to read and understand because it consists entirely of numeric code.

Writing and interpreting these programs requires a deep understanding of machine-level instructions. An example of a program written in machine language is shown in Table 3.4. This program adds two numbers stored in memory locations 0 and 1 and stores the result in location 2.

Table below *A program written in machine language*

Location of instruction	Function code	Operand address
0	0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 1 1 0
1	0 0 0	0 0 0 0 0 0 0 0 0 0 1 0 1 0
2	0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0

Second generation (assembly language)

Assembly language involves a set of symbolic operations called mnemonics. Mnemonics code is made up of two or three letters, such as LDN, STA, and JPU. Typical features of the assembly language differ from one computer to another.

A program written in assembly language should be translated from the source program to machine code, known as object code.

The program required to translate assembly language into machine code is known as an assembler. An example of a program written in assembly language is represented in Table below

Program	Meaning of code
1. MOVESTACK INDEX	Save the current base address of the stack in the index register
2. PUSH FIRST	Push the first number on the stack
3. PUSH SECON	Push the second number on the stack
4. INC STACK	Make space on the stack for the third number
5. JSR OAA	Jump to a subroutine, which orders and adds
6. POP TOTAL	Remove the total from the stack
7. POP SECON	Remove the second number from the stack
8. POP FIRST	Remove the first number from the stack

Third generation (high-level languages)

Third-generation languages (3GLs) refer to programming languages that allow programs to be divided into subprograms, commonly known as procedures. These languages are often called structured or procedural languages. Structured programming languages are designed to be simple, readable, and easily understood by

humans. They use a limited number of straightforward control structures compared to other high-level languages.

Common examples of third-generation languages include Pascal, FORTRAN, COBOL, BASIC, Ada, and C.

Fourth generation (very high-level languages)

Fourth-generation languages are designed to be closer to human language, making them easier to learn and use. They often incorporate graphical user interfaces (GUIs) and support event-driven programming, where the flow of the program is determined by user actions such as mouse clicks and keyboard input. These languages frequently use drop-down menus and forms to facilitate user interaction. Fourth-generation languages are commonly used for designing websites, developing applications, and managing databases. Examples include Python, Ruby, SQL, PHP, JavaScript, and Visual Basic.

Fifth generation languages

Fifth-generation languages (5GLs) represent the most advanced category of programming languages. They are designed to enable computers to solve problems with minimal or no human intervention. Unlike traditional programming languages that require developers to write detailed step-by-step algorithms, 5GLs focus on defining problems using constraints and logical relationships. The system then determines the solution automatically. These languages often include advanced tools such as application generators and inference engines, which significantly reduce the need for manual coding. As a result, 5GLs are especially valuable in fields like artificial intelligence and expert systems. Figure 3.6 shows the hierarchy of the generation of languages.

Table below is Hierarchy of generation of languages

The summary of the information on generations of programming languages is shown in Table below

Language generation	**Properties**
First generation languages	These are low-level programming language, such as machine language
Second generation languages	These are low-level programming languages such as assembly language

Third generation languages	These are high-level programming languages such as C, C++, Java, Visual Basic, and JavaScript
Fourth generation languages	These are languages that consist of statements that are like statements in the human language, such as CSS, Python, and SQL
Fifth generation languages	These are programming languages that have visual tools to develop a program like Mercury, OPS5, and Prolog

Examples of popular programming languages

1. **Scratch:** It is a beginner-friendly visual programming language, specially designed for children. It uses a drag-and-drop interface where users can combine blocks of code to create animations, games, and stories. It is a great way to learn the basics of programming, such as loops, conditions, and variables, without having to write any code. Scratch is widely used in schools to introduce young students to programming.

2. **Python:** It is a flexible, high-level programming language known for being easy to read and write. It is used across many fields, such as artificial intelligence (AI), data science, web development, and automation. With its large collection of libraries and frameworks, Python simplifies tasks like web scraping, machine learning, and data analysis. Due to its simple syntax and broad usage, it is a favourite among both beginners and experts.
3. **JavaScript:** It is a dynamic programming language mainly used to create interactive websites. It lets developers add features like animations, forms, and live updates to web pages. Alongside HTML and CSS, JavaScript is a core technology of the web. It is used for both front-end (client-side) and back-end (serverside) development, especially with frameworks like Node.js.
4. **Java:** This is a robust, object-oriented programming language used in many applications, particularly mobile development (Android apps) and large-scale enterprise systems. With its "write once, run anywhere" approach, Java programs can run on any system with a Java Virtual Machine (JVM). This makes it ideal for cross-platform applications and large backend systems.
5. **C++:** It is a high-performance language based on C that includes object-oriented features. It is commonly used in industries requiring fast and efficient software, such as video games, system software, and embedded systems. C++ is also crucial in fields like robotics and high-performance computing because it

provides fine control over system resources, making it perfect for tasks where speed and memory management are critical.

Algorithms in programming

An algorithm in programming is a step-by-step procedure or formula used to solve a specific problem. It acts as a set of instructions that guide how to process input and generate output. Algorithms are central to all computer programs and applications, helping to solve problems in a structured and efficient way. The key characteristics of an algorithm include having a defined input, producing a defined output, being finite (it must eventually stop), and being definite (each step is clearly specified).

There are many types of algorithms designed for different tasks. For example, sorting algorithms (like bubble sort or merge sort) arrange data in a specific order, while searching algorithms (such as linear search or binary search) find an element within a dataset. Other algorithms, like dynamic programming, break down problems into simpler subproblems that overlap, helping to solve complex issues more efficiently. Designing an algorithm involves understanding the problem, breaking it down into smaller tasks, and outlining the steps needed to solve it. After writing the steps, testing is crucial to ensure the algorithm works as intended. Computers follow these instructions exactly as written. Efficiency is a key reason why algorithms are essential in programming.

Examples of algorithms in real-life

1. Making a sandwich: Step 1: Get bread, Step 2: Add filling, Step 3: Close the sandwich.
2. Finding the largest number in a list: Compare numbers one by one until the largest is found.
3. Giving directions: Follow a series of steps to reach a destination.

Data types and data structures

In programming, data types and data structures are fundamental concepts that determine how data is stored, organised, and accessed. Understanding these concepts is essential for writing efficient, optimised, and maintainable code.

Data type

Data types define the kind of data a variable can store and determine the operations that can be performed on that data. Understanding data types is essential for efficient and error-free programming. Common data types include:

1. **String:** Represents a sequence of characters, such as words, phrases, or sentences.

   Example: "Hello, World!"
2. **Boolean:** Represents a logical value with only two possible outcomes: true or false. Commonly used in decisionmaking, such as yes/no or on/off scenarios.
3. **Numeric types:** Used to represent numbers and typically include:

   (a) **Integer:** Whole numbers without decimal points. Example: 1, -3, 42

   (b) **Floating-point (float):** Numbers that contain decimal points. Example: 3.14, -0.001
4. **Character:** Represents a single character, such as a letter, digit, or symbol. Example: 'a', '7', '$'
5. **Array:** A collection of multiple values of the same data type stored in an indexed sequence. Example: [1, 2, 3, 4] (an array of integers), ["apple", "banana", "cherry"] (an array of strings)

Data structures

Data structures are methods of organising and storing data in a way that allows for efficient access and modification. The choice of data structure depends on the specific needs of a program, such as how fast it needs to search for data, add new data, or remove data. Some common types of data structures include stacks, heaps, trees, linked lists, queues, tables, and graphs. Each of these structures offers different advantages for storing and organising data, depending on the task at hand.

To interact with and manipulate data stored in these structures, "control flow structures" are used. Control flow structures, such as *loops* and *conditional statements*, help manage how the program accesses, processes, and modifies the data within a given structure. A loop can be used to go through the elements of an array or to traverse a linked list. Similarly, a conditional statement can check if a queue is empty before performing an operation. By combining suitable data structures with control flow statements, programmers can build efficient programs that store, update, and process data effectively.

Control flow structures

In programming, control structures define how a program’s instructions are executed. They determine the order of execution, allow decisions between different paths based on conditions, and enable the repetition of

code blocks. These structures form the foundation for building logical and flexible programs. There are three types of control structures, namely sequential, selection, and iteration.

1. **Sequence**

The sequence control structure is the most basic form of control flow. In this structure, program instructions are executed one after another, from top to bottom, in the exact order they appear. Each step runs exactly once. A real-world analogy is following a step-by-step cooking recipe, as shown in Figure 3.8.

Sequence

(Flowchart showing three rectangular process boxes in a vertical line)

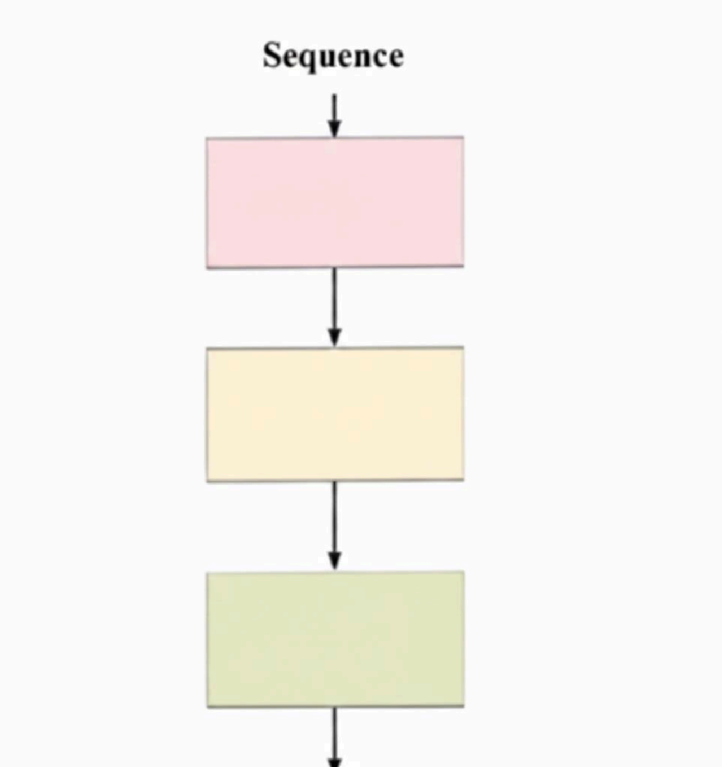


Figure above is the Flow chart for sequential control structure

For example:

Step 1: Gather ingredients

Step 2: Prepare utensils

Step 3: Make a "chapati"

2. **Selection** **(if-else)**

The selection control structure allows a program to make decisions based on certain conditions. It directs the program to follow different paths depending on whether a condition is true or false.

Selection 3. **Loops (repetition or iterations)**

A loop is a control structure in programming that allows a statement or block of code to execute repeatedly as long as a specified condition is true (evaluates to a Boolean value: true or false). Loops are fundamental and powerful because they help automate repetitive tasks efficiently.

Loops are used to repeat a set of instructions multiple times. This is especially useful when a program needs to perform the same operation either for a fixed number of times or until a certain event or condition occurs.

The iteration flowchart, begins with a decision node, checking whether the loop condition is met. Two paths extend from this node:

(i) **True (T) path** – If the condition holds (that is, true), the process moves to an action node, executes the required operation, and loops back to re-evaluate the condition.

(ii) **False (F) path** – If the condition is no longer valid, the iteration stops, and the flowchart exits the loop.

Iteration

(Flowchart showing a diamond decision node with a T path looping back and an F path exiting) **Two common types of loops:**

1. **For loop:** Repeats a block of code for a specific number of times.
2. **While loop:** Continues to execute a block of code as long as a specified condition is true.

For example:

(a) For Loop

For i = 1 to 5:

Print "Hello"

This will print "Hello" five times.

(b) While Loop

While the door is locked:

Try to open the door

Control structures help programmers define how their code should behave in different situations, enabling the creation of dynamic and responsive programs.

Problem-solving skills: Thinking like a programmer Breaking down complex problems

One of the most important skills for any programmer is approaching a complex problem methodically and breaking it down into smaller, more manageable pieces. This process allows programmers to focus on solving one small part of the problem at a time, which leads to more efficient problem-solving and easier debugging.

When faced with a complex task, it is common to start with a big-picture view and then, step-by-step, reduce it to smaller, easier-to-handle components. This approach is applicable in programming and useful in everyday life when tackling large projects or challenges.

(c) Write code to display graphics and handle player actions

Once the game's rules and actions are defined, the next step is programming the game's visuals and interactions. This involves rendering the game world, creating characters and objects, and ensuring the game responds to player inputs.

Key tasks:

(i) **Initialise the game environment** – Set up the game window and define the world settings.

(ii) **Create graphics** – Design and display the player's character, obstacles, background, and other visual elements.

(iii) **Implement player controls** – Write code that moves the character based on input (for example, pressing the arrow keys).

(iv) **Handle physics and interactions** – Implement mechanics like jumping, collisions, and gravity if needed.

Would you like me to extract specific sections or create a summary based on this text?

Based on the images provided, here is the transcribed text, organized by page number as shown in the documents.

(v) **Enhance the experience** – Add animations, sound effects, and other visual feedback.

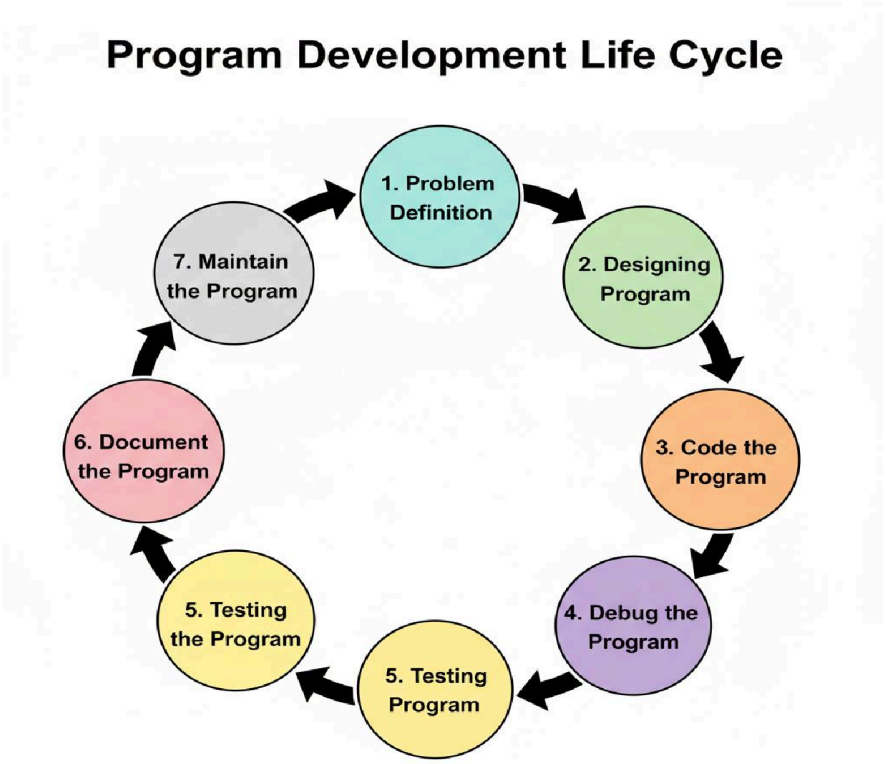
(vi) **Track game progress** – Include systems for scoring, level progression, or player health.

For example, if the player presses the right arrow key, the character should move right. If the character collides with an obstacle, the game might trigger an event such as reducing health or restarting the level.

Program Development Life Cycle (PDLC)

Program Development Life Cycle (PDLC), also known as Software Development Life Cycle (SDLC), is a structured process that outlines the key phases involved in developing a program. It includes steps such as planning, designing, building, testing, deploying, and maintaining the software as shown in Figure below The

PDLC helps programmers improve efficiency and quality by analyzing and optimizing each stage of development.



1. Problem definition
2. Designing program
3. Code the program
4. Debug the program
5. Testing the program
6. Document the program
7. Maintain the program

1. Problem definition

Problem Definition, also known as problem recognition, involves clearly understanding and defining the problem to be solved. At this stage, the software developer identifies the issue, gathers user requirements, and determines what the program is expected to achieve. This step ensures a clear understanding of the problem before development begins. The following questions provide answers to aspects that the programmer should understand at this stage.

(a) What issue needs to be addressed?

(b) What input data is needed to help design the program?

(c) What should the program produce as output?

(d) What methods are currently used to solve this problem?

(e) Can a software-based solution provide a more effective way to solve the problem?

Example of problem definition:

Problem: To calculate average of students

Input: Student score and total number of students

Output: Average score

Write a program that can be used to calculate the average score of students.

The formula used is:

Average = Sum of values / Number of students

During problem definition, the programmer should clearly describe what the program will do, including identifying the input and expected output. A thorough problem definition is essential to ensure user satisfaction and support the success of later development stages. Poor planning at this stage can lead to a system that fails to function as intended.

2. Designing a program

Program design is the second stage of the Program Development Life Cycle (PDLC), focused on creating a system that meets the requirements defined during problem definition. Using the approved requirements, the developer creates detailed design elements often through discussions, interviews, or prototypes. These design tools may include flowcharts, hierarchy diagrams, pseudocode, or entity-relationship diagrams to outline the program's structure and features.

Design elements provide enough detail for skilled programmers to develop the program.

Algorithm

An algorithm is a step-by-step method for solving a specific problem and is executed the same way each time when programmed.

Pseudocode

Tools like pseudocode and flowcharts are used to represent algorithms. Pseudocode uses structured natural language (such as, English or Kiswahili) to describe the logic of a program in a simple, readable format. While easier to follow than flowcharts, pseudocode cannot be directly executed by a computer.

Flowchart

A flowchart is a visual diagram that represents an algorithm by showing the sequence of steps needed to solve a problem. It uses standardized symbols connected by arrows to illustrate the flow of operations.

3. Coding the program

Coding is the third stage of the PDLC, where the programmer begins writing the actual program using a chosen programming language. In this context, you can decide to use any language, including the C programming language, due to its simplicity, structured nature, and suitability for learning basic programming concepts.

4. Debugging a program

Debugging is the process of identifying and fixing errors (bugs) in a program to ensure it runs correctly. These errors can be compile-time (like syntax mistakes) or run-time (such as division by zero), and each type affects the program differently. Effective debugging involves capturing the program's state when the error occurs, analyzing it to find the cause, and fixing the issue without introducing new problems.

5. Testing a program

Testing is the fifth stage of the PDLC and follows debugging, where the program is checked in a testing environment to ensure it functions as intended. Different testing types include functionality testing (covering user interface, security, and database), usability testing (involving target users to assess ease of use), compatibility testing (checking performance across devices), and performance testing (evaluating system response and behavior under heavy loads). These tests help confirm the program meets requirements and performs reliably before deployment.

6. Documentation of a program

Program documentation involves creating clear and simple written or visual materials to explain the software for end users, who often lack programming knowledge. It is essential throughout development to track program details, improve quality, support understanding by others, and aid user training. Common documentation types include user manuals, operational manuals, design documents, requirements documents, testing reports, and lists of known bugs.

7. Program maintenance

Program maintenance involves updating and modifying software after release to fix errors, enhance performance, remove outdated parts, and add new features. Its goal is to ensure the system continues to meet user needs and functions as intended. Maintenance types include corrective, preventive, perfective, and adaptive.

Good coding habits

Writing good code makes programs easier to read, fix, and improve. Here are some simple habits every programmer should follow when coding:

1. Keep your code neat and organised

(a) Arrange blocks in a logical order.

(b) Group related blocks together to make the program easier to understand.

(c) Use meaningful names for sprites, variables, and broadcasts (for example, score instead of x).

2. Reuse blocks and avoid repetition

(a) Create custom blocks (functions) for repeated actions instead of copying the same set of blocks many times.