What is programming? **(Here there are some explanations are explained two times that is different from first explain so read carefully for full understanding.)**

**Programming** is the process of writing instructions (code) for computers to perform specific tasks.

It involves creating algorithms, solving problems, and implementing solutions using programming languages like Python, Java, C++, and more.

1. **Code**: A set of instructions written in a programming language.
2. **Algorithm**: A step-by-step procedure to solve a problem.
3. **Syntax**: Rules defining how code must be written in a language.
4. **Debugging**: Fixing errors in code.
5. **Program/Software**: The final product that runs on a computer or device.

These are the concepts in programming.

Why Learn Programming?

* Automate repetitive tasks.
* Build websites, apps, games, and AI systems.
* Enhance problem-solving and logical thinking.
* High demand in tech jobs.

**--------------------------------------------------------------------------------------------------------------------------**

**Programming paradigms**

A **programming paradigm** is a fundamental style or approach to writing code, based on a set of principles and techniques.

List of programming paradigms

**1. Imperative**

**2. Declarative**

**3. Object-Oriented (OOP)**

**4. Event-Driven**

**5. Concurrent & Parallel**

**6. Generic**

**7. Aspect-Oriented (AOP)**

**8. Metaprogramming**

**9. Dataflow**

**10. Reactive**

**11. Array-Oriented** (e.g., APL)

**12. Symbolic** (e.g., Lisp)

**13. Automata-based**

**14. Quantum**

**--------------------------------------------------------------------------------------------------------------------------**

The most fundamental programming approaches (or paradigms) define how programmers’ structure and design their code. These approaches influence problem-solving strategies, code organization, and software architecture.

The **four main fundamental approaches** are:

1. **Imperative Programming**

**Focus:** "How" to perform tasks step-by-step.

**Characteristics:**

* Uses statements that modify program state.
* Follows a sequence of commands (like a recipe).
* Relies on variables, loops, and conditionals.

Subtypes:

* **Procedural Programming** (C, Pascal, BASIC)
  + Organizes code into reusable procedures/functions.
* **Structured Programming** (Avoids goto, uses loops & conditionals)

2. **Declarative Programming**

* Focuses on **what** needs to be achieved rather than how
* Describes the desired result without specifying the control flow
* Includes:
  + **Functional programming** (uses pure functions, avoids state changes)
  + **Logic programming** (uses formal logic and rules)

Example: SQL (declarative), Haskell (functional), Prolog (logic)

3. **Object-Oriented Programming (OOP)**

**Focus:** Organizing code into "objects" with data and behaviour.  
**Characteristics:**

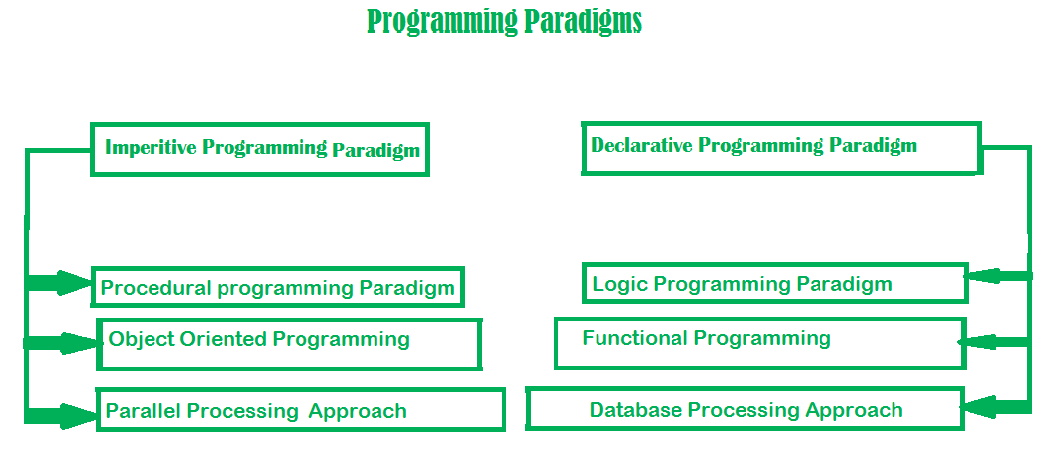
* Uses **classes** (blueprints) and **objects** (instances).
* Follows **4 key principles**:
  1. **Encapsulation** (Hiding internal details)
  2. **Inheritance** (Reusing code via parent-child classes)
  3. **Polymorphism** (Same method, different behaviours)
  4. **Abstraction** (Simplifying complex systems)

**Example (Java)**

4. **Event-Driven Programming**

**Focus:** Responding to events (user actions, system events).  
**Characteristics:**

* Uses **event loops** and **callback functions**.
* Common in GUIs, web apps, and IoT systems.
* **Example (JavaScript - Button Click Event):**



Here there are two **Main types that are imperative and declarative programming.**

**Imperative programming paradigms**

It is one of the oldest programming paradigms.

It features close relation to machine architecture. It is based on Von Neumann architecture.

It works by changing the program state through assignment statements. It performs step by step task by changing state.

The main focus is on how to achieve the goal. The paradigm consists of several statements and after execution of all the result is stored.

**Advantages:**

1. Very simple to implement
2. It contains loops, variables etc.

**Disadvantage:**

1. Complex problem cannot be solved
2. Less efficient and less productive
3. Parallel programming is not possible

**Example of imperative paradigms**

C developed by Dennis Ritchie and Ken Thompson

Fortan developed by John Backus for IBM

Basics developed by John G Kemeny and Thomas E Kurtz

**Declarative programming paradigms**

It is divided as Logic, Functional, Database.

In computer science the *declarative programming* is a style of building programs that expresses logic of computation without talking about its control flow. It often considers programs as theories of some logic. It may simplify writing parallel programs.

The focus is on what needs to be done rather how it should be done basically emphasize on what code is actually doing. It just declares the result we want rather how it has been produced.

This is the only difference between imperative (how to do) and declarative (what to do) programming paradigms. Getting into deeper we would see logic, functional and database.

**How Imperative & Declarative Programming Are Implemented**

**1. Imperative Implementation**

* **Direct Execution**: Code maps closely to CPU instructions (loops → JMP, variables → registers).
* **Manual Control**: Programmer explicitly manages memory, loops, and state changes.
* **Example**:

// C (Imperative)

int sum = 0;

for (int i = 0; i < 10; i++) { sum += i; }

 Compiles to **assembly** with explicit jumps and register operations.

**2. Declarative Implementation**

* **Abstraction Layer**: Runtime/compiler translates high-level logic into low-level operations.
* **Automatic Optimization**: The system decides *how* to execute (e.g., SQL query planner).
* **Example**:

-- SQL (Declarative)

SELECT name FROM users WHERE age > 18;

Database converts this to **imperative steps** (index scans, memory allocations).

**3. Key Differences in Implementation**

| **Aspect** | **Imperative** | **Declarative** |
| --- | --- | --- |
| **Control** | Programmer writes CPU-like steps | Runtime/compiler generates steps |
| **Hardware** | Direct access (memory/CPU) | Indirect (managed by runtime) |
| **Debugging** | Easier (visible steps) | Harder (hidden optimizations) |
| **Performance** | Tunable (manual optimizations) | Auto-optimized (sometimes slower) |

**4. Where They Meet**

* **Compilers**: Turn declarative code (Haskell, SQL) into imperative machine code.
* **Web Browsers**: Convert declarative HTML/CSS into GPU-accelerated rendering.
* **Modern Apps**: Mix both (e.g., React’s declarative UI + imperative JavaScript logic).

**5. Analogy**

* **Imperative** = Cooking a recipe yourself (control every step).
* **Declarative** = Ordering food (you specify the dish, not how it’s made).