



Cambridge (CIE) A Level Computer Science



Your notes

Programming Paradigms

Contents

- * Paradigms
- * Low-Level Programming
- * Imperative Programming
- * Declarative Programming



Paradigms

What is a programming paradigm?

- A programming paradigm is a **style or approach to programming** that influences:
 - How programs are **written**
 - How problems are **broken down**
 - How solutions are **structured**
- **Different** paradigms are suited to **different** types of problems and systems

Programming paradigms comparison table

Paradigm	Description	Key characteristics	Examples
Low-level	Closest to machine code, using mnemonics to directly control hardware	- Direct memory access - Register manipulation - Hardware-specific instructions	x86 Assembly, ARM Assembly
Imperative (Procedural)	Tells the computer how to perform tasks using sequences of commands	- Step-by-step instructions - Use of loops, conditions - Procedures/functions	C, Pascal, Python (procedural)
Object-Oriented	Models real-world entities using objects that combine data and behaviour	- Classes and objects - Encapsulation, inheritance, polymorphism - Modularity	Java, C++, Python (OOP)
Declarative	Describes what should be done, not how to do it	- Rule-based or logic-based - No explicit control flow - Focus on outcomes/results	SQL, Prolog, Haskell

Strengths and weaknesses of programming paradigms

Paradigm	Strengths	Weaknesses
Procedural	- Clear flow of control (top to bottom) - Efficient for simple tasks	- Becomes hard to manage in large programs - Poor modularity can lead to



Your notes

	- Easy to implement algorithms - Strong step-by-step logic	- redundancy - Not ideal for complex state-based systems
Object-Oriented	- Enhances modularity with encapsulation - Real-world modelling via objects - Code reuse through inheritance - Polymorphism for flexible interfaces	- Can become overly complex - Slower due to object overhead - Misuse leads to bloated hierarchies - Not ideal for every problem
Low-Level (Assembly)	- Complete control over hardware - Highly optimised for performance - Transparent view of machine operations	- Steep learning curve - Hardware-specific, not portable - Manual memory management is error-prone - Difficult to debug/scale
Declarative	- Focuses on the result rather than the process - Concise and expressive - Suitable for complex logic or rule-based problems (e.g. AI, SQL queries)	- Harder to learn for beginners - Less control over program flow - Not suited for all types of problems - Debugging can be challenging due to abstraction



Low-level programming

How do I write low-level code using addressing modes?

- Machine code and assembly language are examples of low-level languages and were introduced in [section 4 - Assembly Language Basics](#)
- Before continuing, make sure you understand the different [addressing modes](#)

Immediate addressing

- Syntax:** `LOAD #value`
- Meaning:** Load a constant value directly into the accumulator

`LOAD #5` ; Load the constant value 5 into the accumulator
`ADD #3` ; Add 3 to the accumulator
`STORE 200` ; Store result at memory address 200

- Use when you want to work with **literal values**, not data in memory

Direct addressing

- Syntax:** `LOAD address`
- Meaning:** Load the value stored **at** the specified memory address

`LOAD 100` ; Load the value stored at memory address 100
`ADD 101` ; Add the value at address 101
`STORE 102` ; Store result in address 102

- Use when the data is stored at **known memory addresses**

Indirect addressing

- Syntax:** `LOAD @address`
- Meaning:** The address given **points to another address**, where the data is stored

`LOAD @150` ; Load value from the address stored in location 150
`STORE 200` ; Store it in address 200

- Example:
 - `Memory[150] = 300`
 - `Memory[300] = 42`
 - `LOAD @150` loads 42 into the accumulator.
- Use when data is stored in **dynamically referenced locations**

Indexed addressing



Your notes

- **Syntax:** `LOAD base[X]`
- **Meaning:** Load from **base address + index register** (e.g. `X`)

`LOAD 500[X]` ; Load value from address (500 + contents of index register X)
`ADD 501[X]` ; Add value from another offset

- Assume:
 - `X = 2`
 - This will access address 502 and 503
- Use for working with **arrays or tables**, where the index changes

Relative addressing

- **Syntax:** `BRANCH offset`
- **Meaning:** Jump to a new instruction **relative** to the current instruction

`BRZ +3` ; If accumulator is zero, skip 3 lines ahead
`BRP -2` ; If positive, jump 2 lines back

- Used in control flow:

```
LOOP: LOAD #1
      SUB 200
      BRZ END
      BR -3 ; Loop back to LOOP
END:  HALT
```

- Use for **loops and conditional branches** without needing absolute addresses



Imperative programming (Procedural)

How do I write imperative (procedural) code?

- High-level languages such as **Python, Java, and Visual Basic** support the imperative style
- Imperative code can be made **more organised and readable** by using **structured programming principles**, such as dividing code into **functions and procedures**
 - Before continuing, ensure you're confident with the [basics of programming](#) logic and have a good understanding of [structured programming](#)

Imperative programming – pseudocode (no procedures or functions)

- **Task:** Convert a temperature to:
 - Fahrenheit from Celsius
 - Celsius from Fahrenheit
 - Kelvin from Celsius
- **Rules:**
 - No procedures or functions
 - Use only **global variables**
 - Use a top-down, sequential flow (basic imperative structure)

Imperative pseudocode

```
// Global variables
DECLARE inputTemp : REAL
DECLARE convertedTemp : REAL
DECLARE choice : INTEGER

OUTPUT "Select conversion type:"
OUTPUT "1. Celsius to Fahrenheit"
OUTPUT "2. Fahrenheit to Celsius"
OUTPUT "3. Celsius to Kelvin"

INPUT choice
OUTPUT "Enter the temperature:"
INPUT inputTemp

IF choice = 1 THEN
    SET convertedTemp = (inputTemp * 9 / 5) + 32
    OUTPUT "Temperature in Fahrenheit: ", convertedTemp
ELSE IF choice = 2 THEN
    SET convertedTemp = (inputTemp - 32) * 5 / 9
    OUTPUT "Temperature in Celsius: ", convertedTemp
```

```

ELSE IF choice = 3 THEN
    SET convertedTemp = inputTemp + 273.15
    OUTPUT "Temperature in Kelvin: ", convertedTemp
ELSE
    OUTPUT "Invalid option selected."
ENDIF

```



Your notes

Walkthrough – What’s happening?

- All variables are **declared globally**
- The entire program is written as **one long block**
- There is **no reusability** (e.g. if you want to convert again, you'd have to rerun the whole thing)
- Changes to the program will likely lead to **code duplication** and harder maintenance

Structured (procedural) programming – pseudocode

- **Goal:** Rewrite the above code using **procedures or functions**, **local variables**, and **modular logic**

```

// Global variable
DECLARE choice : INTEGER

PROCEDURE main()
    OUTPUT "Select conversion type:"
    OUTPUT "1. Celsius to Fahrenheit"
    OUTPUT "2. Fahrenheit to Celsius"
    OUTPUT "3. Celsius to Kelvin"

    INPUT choice

    IF choice = 1 THEN
        CALL convertCtoF()
    ELSE IF choice = 2 THEN
        CALL convertFtoC()
    ELSE IF choice = 3 THEN
        CALL convertCtoK()
    ELSE
        OUTPUT "Invalid option selected."
    ENDIF
ENDPROCEDURE

PROCEDURE convertCtoF()
    DECLARE inputTemp : REAL
    DECLARE result : REAL
    OUTPUT "Enter temperature in Celsius:"
    INPUT inputTemp
    SET result = (inputTemp * 9 / 5) + 32
    OUTPUT "Temperature in Fahrenheit: ", result
ENDPROCEDURE

PROCEDURE convertFtoC()
    DECLARE inputTemp : REAL
    DECLARE result : REAL
    OUTPUT "Enter temperature in Fahrenheit:"
    INPUT inputTemp

```

```

SET result = (inputTemp - 32) * 5 / 9
OUTPUT "Temperature in Celsius: ", result
ENDPROCEDURE

```

```

PROCEDURE convertCtoK()
  DECLARE inputTemp : REAL
  DECLARE result : REAL
  OUTPUT "Enter temperature in Celsius:"
  INPUT inputTemp
  SET result = inputTemp + 273.15
  OUTPUT "Temperature in Kelvin: ", result
ENDPROCEDURE

```

```

// Start program
CALL main()

```

Walkthrough – what’s improved?

- Logic is **divided into reusable blocks** (procedures)
- Each procedure has **local variables**
- The program is easier to **read, maintain, and extend**
- Only one global variable (**choice**) is used for communication
- Additional features (like repeating conversions or logging) can be added more easily

Summary

Feature	Imperative	Structured (Procedural)
Reusability	No reuse	Easy to reuse procedures
Modularity	All logic in one block	Code divided into named units
Readability	Can become unclear in longer code	Easier to understand each part
Variables	All global	Mix of global and local
Maintenance	Changes affect entire block	Easier to update specific parts
Ideal for real-world apps?	Not ideal	Industry standard approach



Your notes



Declarative programming

How do I write declarative code?

- Declarative programming is a paradigm where you describe **what** you want the program to accomplish, not **how** to do it
- **SQL** is a common example of declarative programming
- It allows you to **extract and manipulate data using queries**, without needing to specify the control flow
- SQL was covered in more detail in [Section 8 – SQL](#)
- Declarative code is made up of:
 - **Facts** – things that are known or assumed to be true
 - **Rules** – logical relationships between facts
- Unlike imperative programming, where order and control flow are essential, declarative code allows you to define **facts and rules in any order**, and then write a **query** that uses them
- In the following examples, a logic-based declarative language (like **Prolog**) is used

Example: Facts about dog breeds

type(beagle, hound).
type(labrador, retriever).
type(poodle, companion).
type(golden_retriever, retriever).
type(bulldog, companion).

size(beagle, medium).
size(labrador, large).
size(poodle, medium).
size(golden_retriever, large).
size(bulldog, medium).

shedding(beagle, moderate).
shedding(labrador, heavy).
shedding(poodle, low).
shedding(golden_retriever, heavy).
shedding(bulldog, low).

Interpreting clauses

- Each fact uses the format:

predicate(object, property).

Clause	Meaning
--------	---------



Your notes

<code>type(poodle, companion).</code>	A poodle is a companion dog
<code>size(labrador, large).</code>	A Labrador is a large-sized dog
<code>shedding(bulldog, low).</code>	A bulldog has low shedding

Sample queries

- Use queries to retrieve information from the facts

?- size(poodle, medium).
true.

?- type(X, retriever).
X = labrador ;
X = golden_retriever.

?- shedding(X, low), size(X, medium).
X = poodle ;
X = bulldog.

Key points

Concept	Description
Fact	A statement like <code>type(beagle, hound).</code> that is always true
Rule	A logical relationship built from facts (not shown here)
Query	A question to the knowledge base, e.g. <code>size(X, medium).</code>
Any order	Facts and rules can appear in any order
Result	The interpreter matches patterns in queries against known facts