# Types of Programming Language

**Paradigms**

- Paradigms are ways of thinking. Programming paradigms are ways of programming.

**Generations**

1. Machine Code

- In binary

- Low level

- Difficult to write

1. Assembler

- Makes programming easier

- Replaces machine code functions with mnemonics.

1. Procedural

- Most higher level languages.

1. Database

- Closer to human languages than typical high-level programming.

- Most used for databases and AI.

**Turing Complete:** A Programming language is referred to as Turing complete if it can solve all problems it has been proved computers can solve.

- Most languages are Turing complete but some are more suited to different problems and paradigms.

**Imperative Language:** You tell the computer what to do step by step.

**Declarative Language:** You tell the computer the qualities the solution should have and let the computer work out the solution.

The main classes of programming language:

1. Low Level
2. Procedural
3. Object-Orientated
4. Declarative
5. Functional

**Low Level**

- Programming statements are geared towards a particular CPU family.

- Mnemonics that have 1:1 relationship with machine code.

- Chip maker provides programmers with the language for each CPU.

**Advantages**

- Good for close control of CPU therefore good for efficiency.

**Disadvantages**

- Difficult to use as you have to memorise lots of codes and debugging is a pain.

- Non-portable.

**Procedural**

- Code specific instructions for the computer to carry out.

- Extremely common.

- 3rd generation, easy to read.

- Variables and functions partition code into manageable chunks.

- Imperative, Sequential and 3rd generation.

**Advantages**

- Excellent for general purpose programming.

- Lots of documentation and references for algorithms.

- Good level of control without knowing CPU details.

- Portable source code.

**Disadvantages**

- So many options that programmers tend to specialise in one.

- Needs to be precise and takes a while to debug.

- Not as efficient as lower level.

- Poor with situations where conditions are not clear.

**Object-Orientated**

- An OOP program makes use of the idea of classes and objects.

- A program consists of a number of objects and classes with their own internal methods and variables.

- Before OOP came along, procedural languages treated data and instructions that acted on data as different things which made it possible to write faulty code that overwrote data.

- OOP is imperative.

- OOP makes data hiding, code re-use and maintenance easier.

**Advantages**

- OOP makes it easier to provide working code. A class can be fully tested and released for other coders in a team to use.

- Classes can be treated as **black boxes.** “if it ain’t broke, don’t fix it”.

- Great for code reuse

- There are many design patterns available to solve common programming tasks.

- Portable code, providing you are using the right compiler.

**Disadvantages**

- “Steep Learning Curve” \*cough\* Albanozzo.

- More complex than procedural.

- Not as compact as a low level language.

**Declarative Logic**

- A declarative language is a non-procedural and very high level (4th generation) language.

- Not imperative.

- Programmer specifies what needs doing rather than how it should be done.

- The software will seek an answer to the question by interrogating a database containing facts and rules.

- Non-procedural, it does not matter what order the fact and rules are in the computer will always fine the best path towards the answer.

- There will be a matching answer or a ‘False’ returned where there was no answer to be found.

- Geared more towards AI and expert systems where inexact databases are handled or general decisions are made.

**Declarative Functional**

- Provide answers to problems purely by applying calculations to input data.

- They do not change anything else other than the input data provided at that point in time.

- No side effects.

- These calculations are called ‘functions’ hence functional programming.

- Highly mathematical uses / finance.

**Assembly Language (Little Man Computer)**

- Each line consists of opcode possibly followed by an operand.

- Executed during FDE cycle.

**Instruction Set**

Load - **LDA** - Load contents of address into accumulator.

Store - **STA** - Store contents of accumulator to address.

Add - **ADD** - Add contents of address XX to accumulator.

Sub - **SUB** - subtract contents of address XX from accumulator.

Input - **INP** - Copy value from input box to accumulator.

Output - **OUT** - Copy value from accumulator to outbox.

End - **HLT** - Stop the execution.

Branch if 0 - **BRZ** - Branch to address XX if accumulator value equal to 0.

Branch >= 0 - **BRP** - Branch to address XX if accumulator value greater than or equal to 0.

Branch Always - **BRA** - Branch to address XX.

Data Storage - **DAT** - Reserve memory address XX as data.

**Addressing Modes**

- Every location has its own unique address. We want to be able to retrieve or store data/instructions in memory.

- There are different ways for programs to refer to these addresses - **memory address modes.**

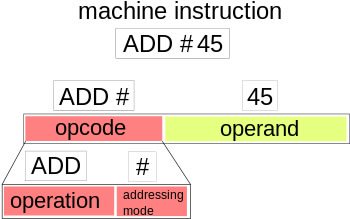
- The five memory address modes are:

1. Direct
2. Indirect
3. Immediate
4. Indexed

**Addressing Modes:** Memory address modes determine the method used by a running program to access its data or next instruction.

- Memory includes both RAM and the internal registers of the CPU.

- If you are coding using a low level language you will make direct use of the address modes whereas high level languages will make use of them but hide these details from the programmer.



**Immediate Addressing**

- Immediate addressing means the data to be used is hard-coded into the instruction: the operand **is** the data.

- Fastest method of addressing as it does not involve fetching from memory at all.

- Very useful for carrying out instructions involving constants.

- Example: ADD 2

**Direct (Absolute) Addressing**

- The operand contains the memory address telling the CPU where to look.

- Fast but not as fast as immediate addressing.

- Cons:

- Code depends on correct data always being in the same location.

- Space for addresses is limited by the size of the operand.

- Generally a good idea to avoid so you can have ‘relocatable data’.

- Good if your computer only ever runs a single program.

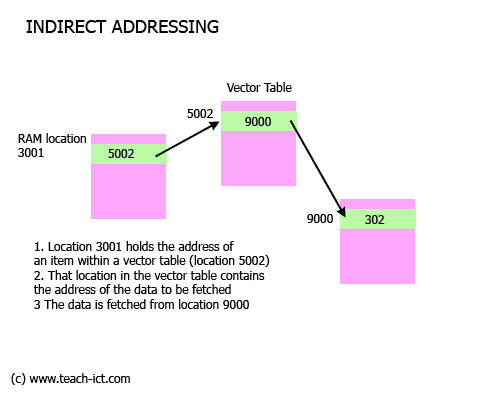
- Example: SUB (3001)

**Indirect Addressing**

- Address of the data is held in an intermediate location so the address is first ‘looked up’ then used to locate the data itself.

- For example, may programs make use of software libraries that get loaded into memory at run time by the loader. These libraries will be placed in a different memory location each time they are loaded.

- Indirect addressing allows the programmer to access these subroutines if they do not know the starting address of the code.



- Example: MOV A, @5002

- Slower than direct as you need two things to be looked up.

- Gives you more space for addressing than direct addressing.

**Indexed Addressing**

- Final address for data is obtained by adding an offset to a base address.

-Very often, a chunk of data is stored as a complete block in memory.

- For example, it makes sense to store arrays as contiguous blocks in memory.

- The array has a base address then an index is used to access specific items within the array.

- Fast and good for manipulating data structures such as arrays.

- If an array is relocated in memory then only the base address needs to be changed.

**Object Orientated Programming**

- In OOP, a solution to a problem is represented through objects. Each object has attributes (variables) that are properties of the object and methods which are actions that the object can carry out.

**Classes and Objects**

- Classes are templates, they define what variables and methods an object should have.

- One advantage of OOP is that once one class has been written it can be used in any program where its needed.

- A constructor is a method that defines what happens when an object that is an instance of that class is created.

**Inheritance**

- Inheritance allows us to have classes that have all the methods and variables of another class - its superclass - but also methods and variables of its own.

- The subclass can also override methods of its superclass if necessary.

**Polymorphism**

- Polymorphism is when methods behave differently depending on the context in which they are used.

- An example is when you have a polymorphic array that contains different classes but all inherit from a common superclass.

**Encapsulation**

- Encapsulation is the pattern of making attributes private within a class but having public methods to change and access them. This means other classes cannot directly change or access the attribute but must use the methods.

- The programmer can strictly control the way in which these attributes are changed and accessed meaning that the chance of unforeseen changes being made by other classes is reduced and the code is more easily debugged.