Topic: 4.3.1 Programming paradigms

Programming paradigms are simply methods of programming. Initially, computers were programmed using binary language. This was difficult and led to many errors that were difficult to find. Programs written in binary are said to be written in machine code, this is a very low-level programming paradigm.

To make programming easier, assembly languages were developed. These replaced machine code functions with mnemonics and addresses with labels. Assembly language programming is also a low-level paradigm although it is a second generation paradigm. The figure below shows an assembly language program that adds together 2 numbers and stores the result.

Label	Function	Address	Comments
	LDA	X	Load the accumulator with the value of X
	ADD	Y	Add the value of Y to the accumulator
	STA	Z	Store the result in Z
	STOP		Stop the program
X:	20		Value of X = 20
Y:	35		Value of Y = 35
Z:			Location for result

Although this assembly language is an improvement over machine code, it is still prone to errors and code is difficult to debug, correct and maintain.

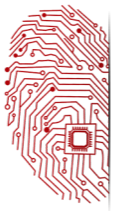
The next advance was the development of procedural languages. These are third generation languages and are also known as high-level languages. These languages are problem oriented as they use terms appropriate to the type of problem being solved. For example, COBOL (**C**ommon **B**usiness **O**riented **L**anguage) uses the language for business. It uses terms like, file, move, and copy.

FORTRAN (**FOR**mula **TRAN**slation) and ALGOL (**ALGO**rithmic **L**anguage) were developed mainly for scientific and engineering problems. Although one of the ideas behind the development of ALGOL was that it was an appropriate language to define algorithms. BASIC (**B**eginners **A**ll-Purpose **S**ymbolic **I**nstruction **C**ode) was developed to enable more people to write programs. All these languages follow the procedural paradigm. That is, they describe, step by step, exactly the procedure that should be followed to solve a problem.

The problem with procedural languages is that it can be difficult to reuse code and to modify solutions when better methods of solution are developed.

In order to address these problems, object-oriented languages (like Eiffel,Smalltalk, and Java) were developed.

In these languages, data and methods of manipulating the data, are kept as a single unit called an "object". The only way that a user can access the data is via the object's methods. This means that, once



Topic: 4.3.1 Programming paradigms

an object is fully working, it cannot be corrupted by the user. It also means that the internal workings of an object may be changed without affecting any code that uses the object.

Take the real world for example, it is full of **objects** not just individual values like in procedural languages. For instance, my car registration number W123ARB, is an object. Ahmed's car registration number S123AHM is another object. Both of these objects are cars and all cars have similar attributes. All cars should have a registration number, a certain engine capacity, certain color, and so on. So my car and Ahmed's car are instances of a class called "car". In order to model the real world, the Object-oriented programming (OOP) paradigm was developed. Unfortunately, OOP requires a large amount of memory and in the 1970s, memory was expensive and CPUs still lacked power. This slowed the development of OOP. However, as memory became cheaper and CPUs more powerful, OOP became more popular. By the 1980s Smalltalk, and later Eiffel, had become well established. These were true object-oriented languages. C++ also includes classes although the programmer does not have to use them. This means that C++ can be used either as a standard procedural language or an object-oriented language. Although OOP languages are procedural in nature, OOP is considered to be a newer programming paradigm.

The following is an example, using Java, of a class that specifies a rectangle and the methods that can be used to access and manipulate the data.

```
class Shapes {

   /* Shapes Version 1 by Roger Blackford July 2001
    ------------------------------------------------------
       This illustrates the basic ideas of OOP
   */

   // Declare three object variables of type Rectangle
   Rectangle small, medium, large;


       // Create a constructor where the initial work is done
   Shapes ( ) {
      // Create the three rectangles
      small = new Rectangle(2, 5);
      medium = new Rectangle(10, 25);
      large = new Rectangle(50, 100);

      //Print out a header
      System.out.println("The areas of the rectangles are:\n");

      //Print the details of the rectangles
      small.write( );
      medium.write( );
      large.write( );
   }//end of constructor Shapes.

   //All programs have to have a main method
      public static void main(String [ ] args) {
      //Start the programm from its constructor
      new Shapes ( );
```

Topic: 4.3.1 Programming paradigms

```
   }//end of main method.

}//end of class Shapes.

class Rectangle {
   //Declare the variables related to a rectangle
   int length;
   int width;
   int area;

   //Create a constructor that copies the initial values into the object's variables
   Rectangle (int w, int l) {
      width = w;
      length = l;

      //Calculate the area
      area = width * length;
   }//end of constructor Rectangle

   //Create a method to output the details of the rectangle
   void write ( ) {
      System.out.println("The area of a rectangle " + width + " by " + length + " is "
+ area);
   }//end of write method.
}//end of constructor
```

This example contains two classes. The first is called Shapes and is the main part of the program. It is from here that the program will run. The second class is called Rectangle and it is a template for the description of a rectangle.

The class Shapes has a constructor called Shapes, which declares two objects of type Rectangle. This is a declaration and does not assign any values to these objects. In fact, Java simply says that, at this stage, they have null values. Later, the new statement creates actual rectangles.

Here small is given a width of 2 and a length of 5, medium is given a width of 10 and a length of 25 and large is given a width of 50 and a length of 100.

When a new object is to becreated from a class, the class constructor, which has the same name as the class, is called by invoking the new operator.

The class Rectangle has a constructor that assigns values to width and length and then calculates the area of the rectangle.

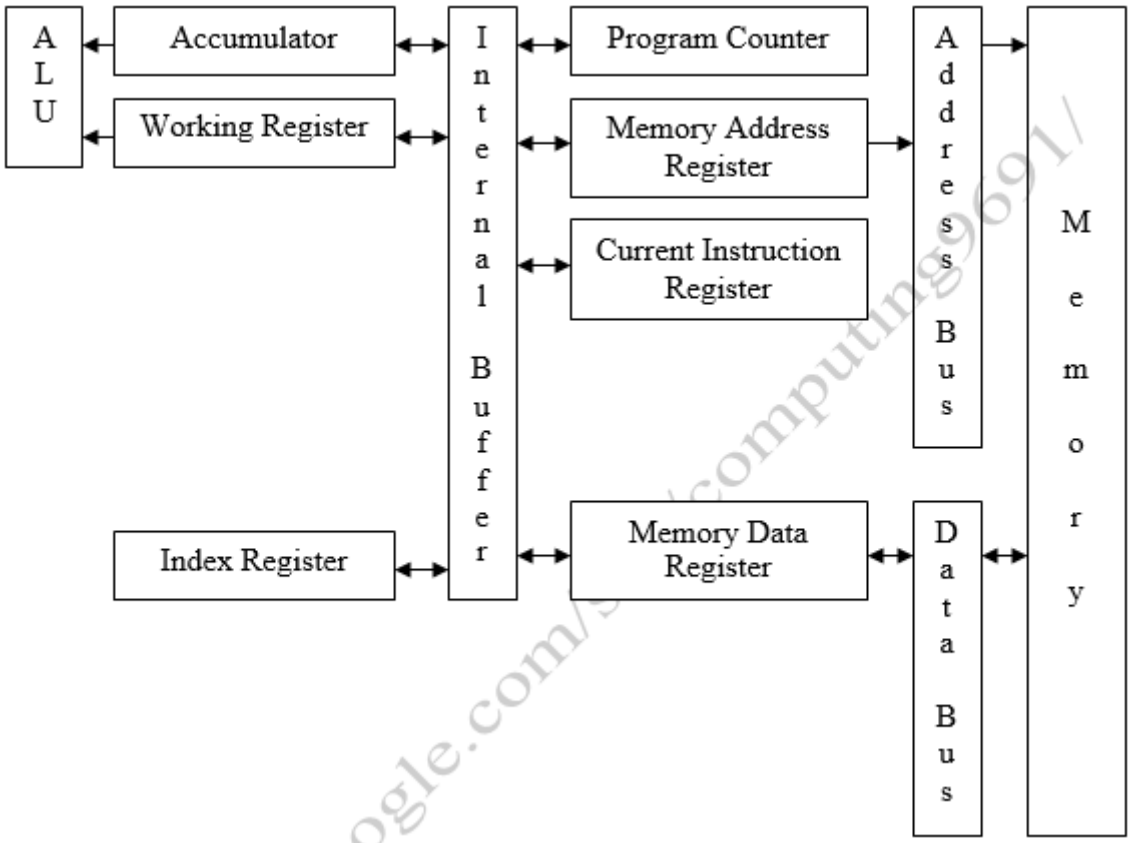
The class Rectangle also has a method called write( ). This method has to be used to output the details of the rectangles.

In the class Shapes, its constructor then prints a heading and the details of the rectangles. The latter is achieved by calling the write method. Remember, small, medium and large are objects of the Rectangle

class. This means that, for example, small.write( ) will cause Java to look in the class called Rectangle for a write method and will then use it. More details of Object-oriented programming will be given later.

Low-level programming

The figure below shows the minimum number of registers needed to execute instructions. Remember that these are used to execute *machine code* instructions not high-level language instructions.



- The ***program counter*** (PC) is used to keep track of the location of the next instruction to be executed. This register is also known as the Sequence Control Register (SCR).
- The ***memory address register*** (MAR) holds the address of the instruction or data that is to be fetched from memory.
- The ***current instruction register*** (CIR) holds the instruction that is to be executed, ready for decoding.
- The ***memory data register*** (MDR) holds data to be transferred to memory and data that is being transferred from memory, including instructions on their way to the CIR. Remember that the computer cannot distinguish between data and instructions. Both are held as binary numbers. How these binary numbers are interpreted depends on the registers in which they end up. The

Topic: 4.3.1 Programming paradigms

MDR is the only route between the other registers and the main memory of the computer.

- The ***accumulator*** is where results are temporarily held and is used in conjunction with a *working register* to do calculations.
- The ***index register*** is a special register used to adjust the address part of an instruction.

The question to ponder upon is how to these registers actually execute instructions. In order to do this, we shall assume that a memory location can hold both the instructions code and the address part of the instruction. For example, a 32-bit memory location may use 12 bits for the instruction code and 20 bits for the address part. This will allow us to use up 2^{12} = 4096 instruction codes and 2^{20} = 1,048,576 memory addresses.

To further simplify things, we shall use mnemonics for instructions such as

Code	Meaning
LDA	load the accumulator
STA	store the accumulator
ADD	add the contents of memory to the accumulator
STOP	Stop

We shall also use decimal numbers rather than binary for the address part of an instruction.

Suppose four instructions are stored in locations 300, 301, 302, and 303 as shown in the following table and that the PC contains the number 300.

Address	Contents	Notes
. . .	. . .	
300	LDA 400	Load accumulator with contents of location 400
301	ADD 401	Add contents of location 401 to accumulator
302	STA 402	Store contents of accumulator in location 402
303	STOP	Stop
304		
. . .	. . .	
400	5	Location 400 contains the number 5
401	7	Location 401 contains the number 7
402	?	Not known what is in location 402
. . .	. . .	

Topic: 4.3.1 Programming paradigms

The fetch part of the instruction is:

Action	PC	MAR	CIR	MDR
Copy contents of PC to MAR	300	300	?	?
Add 1 to PC	301	300	?	?
Copy contents of location pointed to by MAR into MDR	301	300	?	LDA 400
Copy instruction in MDR to CIR	301	300	LDA 400	LDA 400

The instruction is now decoded (not shown in the table) and is interpreted as 'load the contents of the location whose address is given into the accumulator'

Next is the execution phase. As the contents of an address are needed, the address part of the instruction is copied into the MAR, in this case 400.

Action	PC	MAR	CIR	MDR
Copy address part of instruction to MAR	301	400	LDA 400	LDA 400

Now use the MAR to find the value required and copy it into the MDR

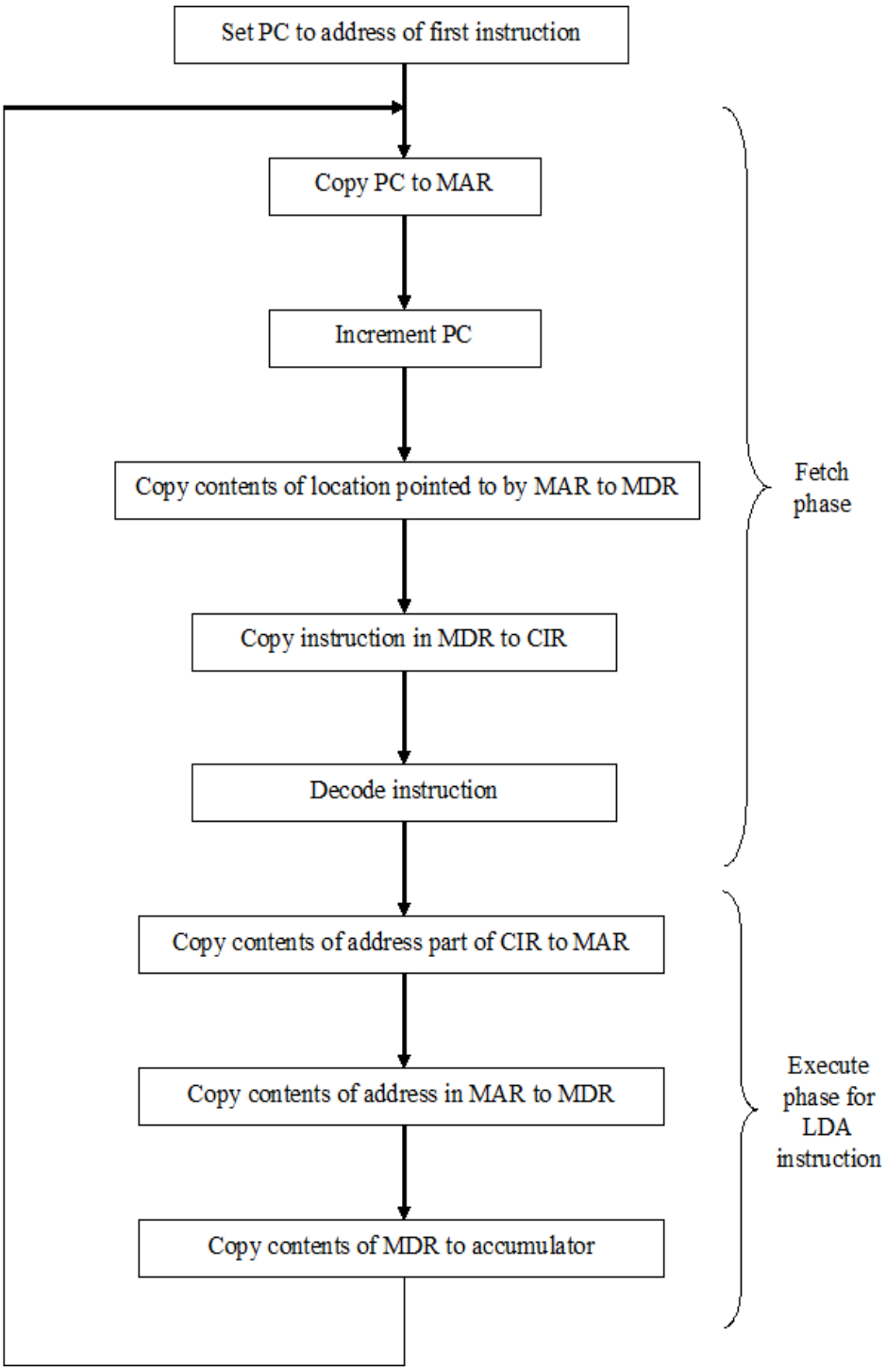
Action	PC	MAR	CIR	MDR
Copy contents of address given in MAR to MDR	301	400	LDA 400	5

Finally copy the contents of the MDR to the accumulator.

Now use the same steps to fetch and execute the next instruction. Note that the PC already contains the address of the next instruction.

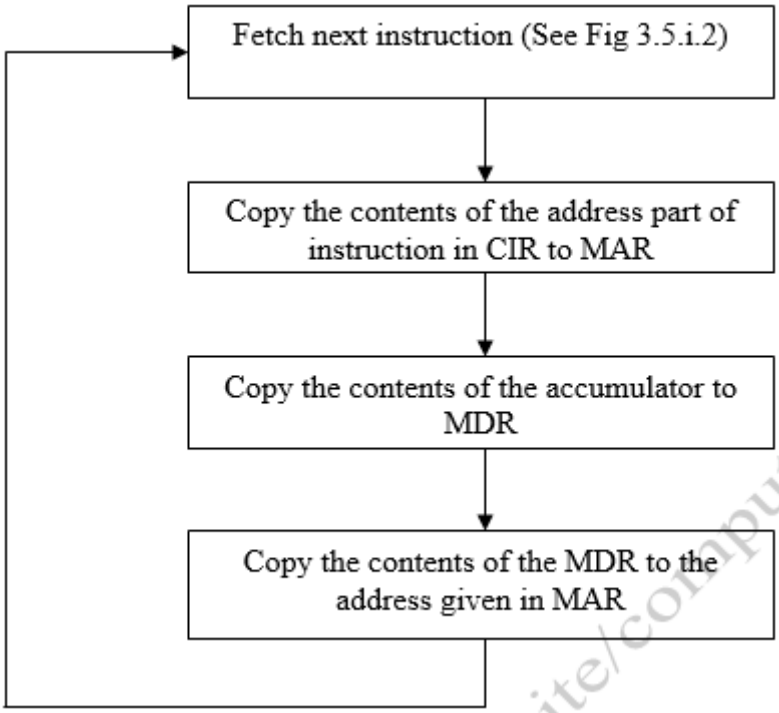
Note that all the data moves between memory and the MDR via the data bus. All the addresses use the address bus.

Topic: 4.3.1 Programming paradigms



Topic: 4.3.1 Programming paradigms

The figure below shows that the outcome of the execute instruction may vary depending on the instruction. For example, STA n (store the contents of the accumulator in the location with address n).

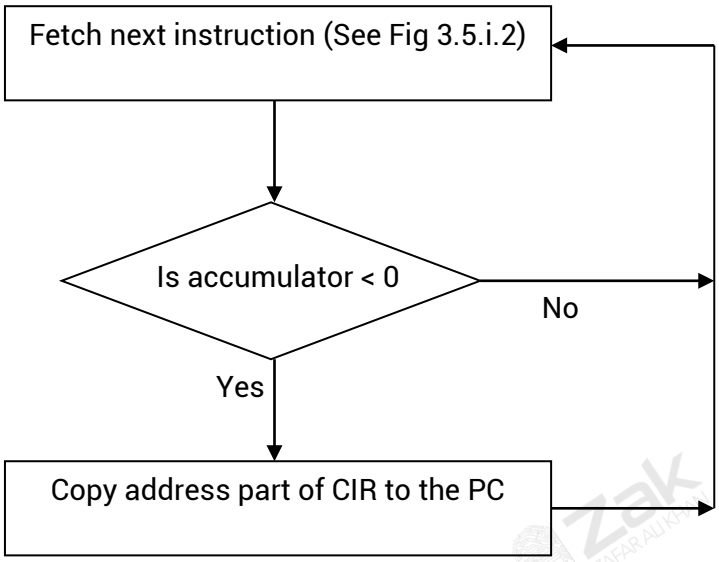


This process works fine but only allows for the sequential execution of instructions. This is because the PC is only changed by successively adding 1 to it. How can we arrange to change the order in which instructions are fetched? Consider these instructions.

Address	Contents	Notes
. . .	. . .	
300	ADD 500	Add the contents of location 500 to the accumulator
301	JLZ 300	If accumulator < 0 go back to the instruction in location 300
. . .	. . .	

Topic: 4.3.1 Programming paradigms

Supposed the PC contains number 300, after the instruction ADD 500 has been fetched and executed, the PC will hold the number 301. Now the instruction JLZ 300 will be fetched in the usual way and the PC incremented to 302. The next step is to execute this instruction. The steps are shown below.



Memory Addressing modes

There are many ways to locate data and instructions in memory and these methods are called 'memory address modes'

Memory address modes determine the method used within the program to access data either from within the CPU or from the external RAM. Some memory addressing modes can control the program flow.

The five memory address modes are:

- Direct
- Indirect
- Immediate
- Indexed
- Relative

Topic: 4.3.1 Programming paradigms

Immediate Addressing

Immediate addressing means that the data to be used is hard-coded into the instruction itself.

This is the fastest method of addressing as it does not involve main memory at all.

For example, you want to add '2' to the content of the accumulator.

The instruction is:

ADC 2

Nothing has been fetched from memory; the instruction simply adds 2 to the accumulator immediately.

Immediate addressing is very useful to carry out instructions involving constants (as opposed to variables). For example, you might want to use the 'PI' as a constant 3.14 within your code.

Direct Addressing

This is a very simple way of addressing memory – direct addressing means the code refers directly to a location in memory.

For example:

SUB (3001)

In this instance, the value held at absolute location 3001 in RAM is subtracted from the accumulator.

The good thing about direct addressing is that it is fast (but not as fast as immediate addressing) the bad thing about direct addressing is that the code depends on the correct data always being present at the same location.

It is generally a good idea to avoid referring to absolute memory addresses in order to have **'re-locatable code'** i.e. code that does not depend on specific locations in memory.

You could use direct addressing on computers that are only running a single program.

For example, an engine management computer only ever runs the code which the car engineers programmed into it, and so direct memory addressing is excellent for fast memory access.

Topic: 4.3.1 Programming paradigms

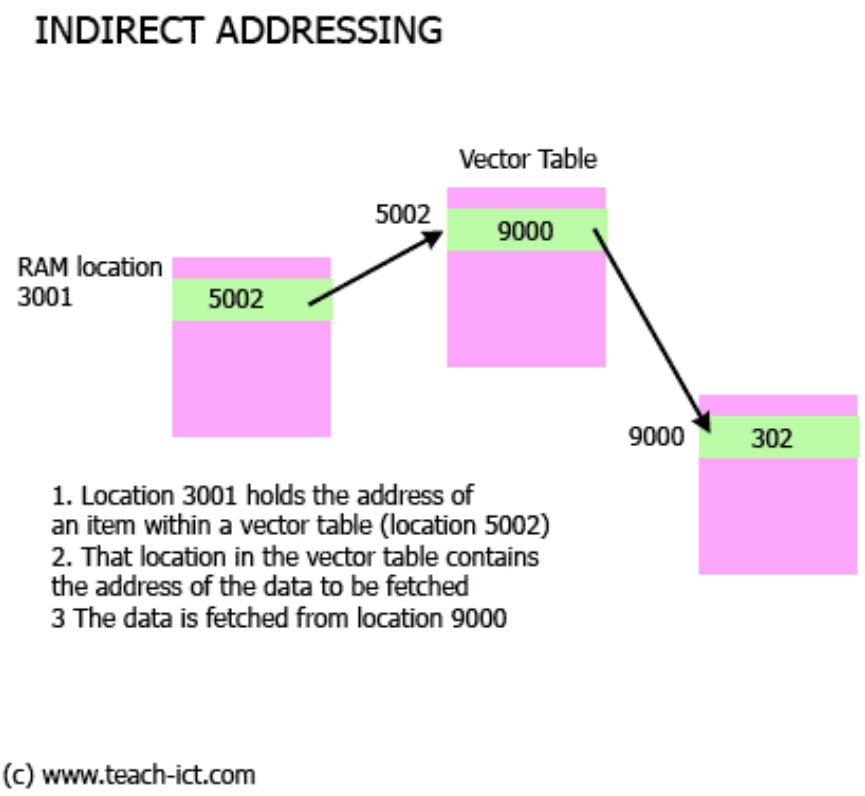
Indirect Addressing

Indirect addressing means that the address of the data is held in an intermediate location so that the address is first 'looked up' and then used to locate the data itself.

Many programs make use of software libraries that get loaded into memory at run time by the loader. The loader will most likely place the library in a different memory location each time.

So how does a programmer access the subroutines within the library if he does not know the starting address of each routine?

By indirect addressing!



Topic: 4.3.1 Programming paradigms

1. A specific block of memory will be used by the loader to store the starting address of every subroutine within the library. This block of memory is called **'vector table'.** A vector table holds addresses rather than data. The application is informed by the loader of the location of the vector table itself.

2. In order for the CPU to get to the data, the code first of all fetches the content at RAM location 5002 which is part of the vector table.

3. The data it contains is then used as the address of the data to be fetched, in this case, the data is at location 9000

A typical assembly language instruction would look like:

MOV A, @5002

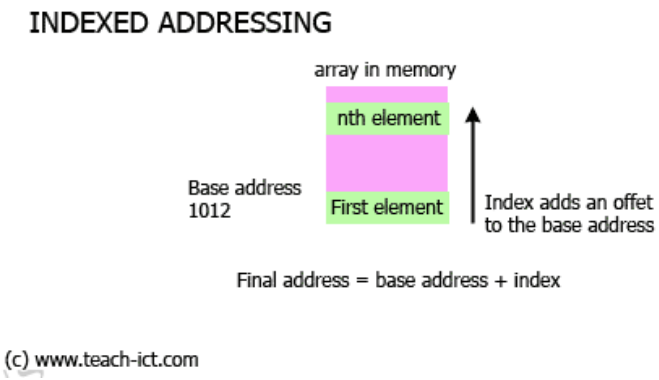
This looks to location 5002 for an address. That address is then used to fetch data and load it into the accumulator. In this instance, it is 302.

Indexed Addressing

Indexed addressing means that the final address for the data is determined by adding an offset to a base address.

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

For example, it makes sense to store arrays as contiguous blocks in memory (contiguous means being next to something without a gap). This array has a **'base address'** which is the location of the first element, then an **'index'** is used that adds an offset to the base address in order to fetch any other element within the array.



Topic: 4.3.1 Programming paradigms

Indexed addressing is fast an excellent for manipulating data structures such as arrays as all you need to do is set up a base address then use the index in your code to access individual elements.

Another advantage of indexed addressing is that if the array is re-located in memory at any point, then only the base address needs to be changed. The code making use of the index can remain exactly the same.

Relative Addressing

Quite often a program only needs to jump a little bit in order to jump to the next instruction. Maybe just a few memory locations away from the current instruction.

A very efficient way of doing this is to add a small offset to the current address in the program counter. (Remember that the program counter always points to the next instruction to be executed). This is called **'relative addressing'**

DEFINITION:

Relative addressing means that the next instruction to be carried out is an offset number of locations away, relative to the address of the current instruction.

Consider this bit of pseudo-code:

```
jump +3 if accumulator == 2
code executed if accumulator is NOT = 2
jmp +5 (unconditional relative jump to avoid the next line of code)

acc:
  code executed if accumulator is = 2)

carryon:
```

In the code above, the first line of code is checking to see if the accumulator has the value of 2 in it. If it does, then the next relevant instruction is 3 lines away. This is called a **conditional jump** and it is making use of relative addressing.

Another example of relative addressing can be seen in the jmp +5 instruction. This is telling the CPU to effectively avoid the next instruction and go straight to the 'carryon' point.

Topic: 4.3.1 Programming paradigms

Summary of memory modes

Memory modes	
Type	**Comment**
Immediate	Apply a constant to the accumulator. No need to access main memory
Direct or Absolute addressing	This is a very simple way of addressing memory - the code refers directly to a location in memory. Disadvantage is that it makes relocatable code more difficult.
Indirect Addressing	Looks to another location in memory for an address and then fetches the data that is located at that address. Very handy of accessing in-memory libraries whose starting address is not known before being loaded into memory
Indexed Addressing	Takes a base address and applies an offset to it and fetches the data at that address. Excellent for handling data arrays
Relative Addressing	Tells the CPU to jump to an instruction that is a relative number of locations away from the current one. Very efficient way of handling program jumps and branching.

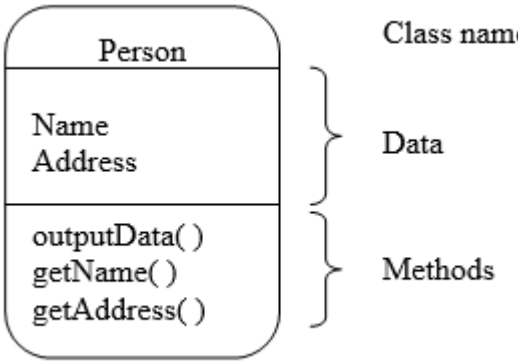
Topic: 4.3.1 Programming paradigms

Object-Oriented Programming (OOP)

Earlier in the chapter, an example of OOP was discussed and now you know some languages that support OOP. Java is an OOP language whose syntax is based on C++. All these languages have classes and derived classes and use the concepts of *inheritance, polymorphism, containment (aggregation).*

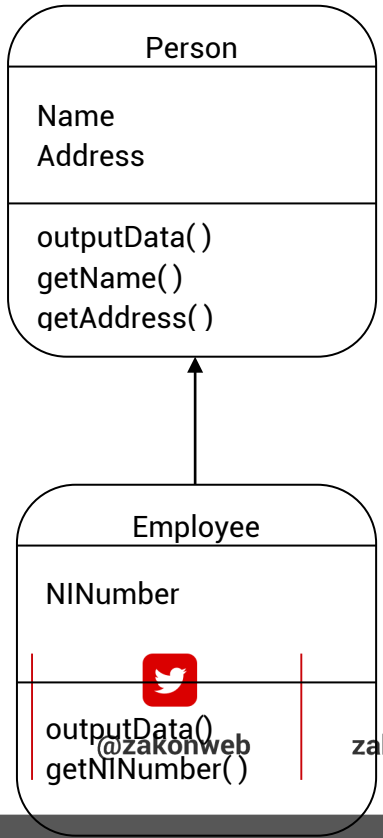
Inheritance allows the re-use of code and the facility to extend the data and methods without affecting the original code. In the following diagrams, we shall use a rounded rectangle to represent a class. The name of the class will appear at the top of the rectangle, followed by the data in the middle, and finally, the methods at the bottom.

Consider the class Person that has data about a person's name and address plus a method called outputData( ) that outputs the name and address, getName( ) and getAddress( ) that return the name and address respectively shown in the figure below.



Now suppose we want a class data and methods as Person and employee's National Insurance rewrite the contents of the class class called Employee that inherits and adds on the extra data and

This is shown in the figure below Employee inherits the data and Person. Person is called the *super-* the *derived class* from Person. An methods provided by Employee and



Employee that requires the same also needs to store and output an number. Clearly, we do not wish to person. We can do this by creating a all the details of the class Person methods needed

where the arrow signifies that methods provided by the class *class* of Employee and Employee is object of type Employee can use the those provided by Person.

Topic: 4.3.1 Programming paradigms

Notice that we now have 2 methods with the same name. How does the program determine which one the use? If myPerson is an instantiation of the Person class, then:

myPerson.outputData( );

will use the outputData( ) from the Person class. The statement:

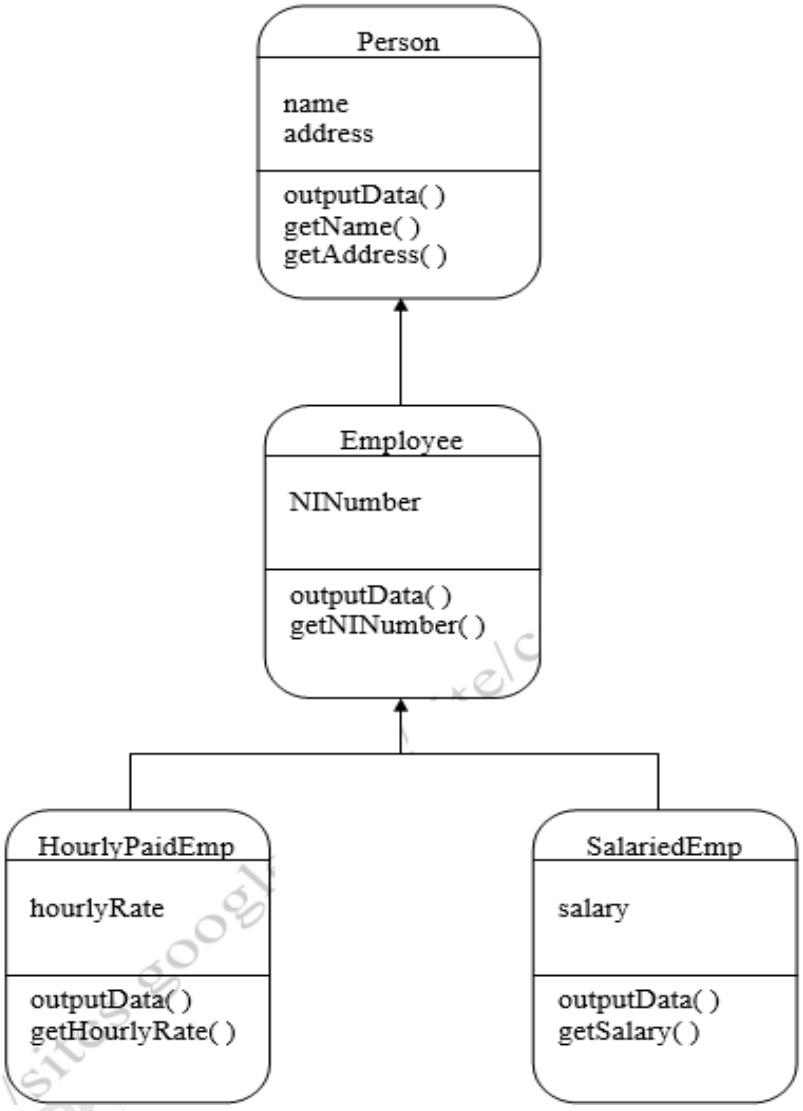
myEmp.outputData( );

will use the method outputData( ) from the Employee class if myEmp is an instantiation of the Employee class.

The concept of a method having two different meanings is called **polymorphism.**

Now suppose we have two types of employee; one is paid hourly and the other is paid a salary. Both of these require the data and methods of the classes Person and Employee but they also need to be different from one another.

Topic: 4.3.1 Programming paradigms



How can an object of type Employee output the name and address as well as the N.I number? The outputData( ) method in class Employee can refer to the outputData( ) method of its superclass. This is done by writing a method, in the class Employee, of the form

```
void outputData( ) {
      super.outputData( );
     System.out.println("The N.I. number is " + NINumber);
  }//end of outputData method.
```

Topic: 4.3.1 Programming paradigms

Here, super.outputData( ) calls the outputData( ) method of the super-class and then outputs the N.I number. Similarly, the other derived classes can call the methods of their super classes.

Association is a relationship between two objects. In other words, association defines the multiplicity between 2 objects. You may be aware of one-to-one, one-to-many, many-to-one, many-to-many all these words define an association between objects. Aggregation is a special form of association.

Aggregation is a special case of association. A directional association between objects. When an object 'has-a' another object, then you've got an aggregation between them. Direction between them specified which object contains the other object.

For example, consider two classes **Student class** and **Address class**. Each student must have an address so the relationship between student and address is a Has-A relationship. But if your consider its vice versa, then it would not make sense as an Address does not necessarily need to have a Student. Below example shows this theoretical explanation in a sample Java program.

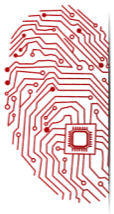
Topic: 4.3.1 Programming paradigms

Student Has-A Address

```
class Address
{
   int streetNum;
   String city;
   String state;
   String country;
   Address(int street, String c, String st, String coun)
   {
        this.streetNum=street;
        this.city =c;
        this.state = st;
        this.country = coun;
   }
}
class StudentClass
{
   int rollNum;
   String studentName;
   Address studentAddr;
   StudentClass(int roll, String name, Address addr){
        this.rollNum=roll;
        this.studentName=name;
        this.studentAddr = addr;
   }
   public static void main(String args[]){
        Address ad = new Address(55, "Agra", "UP", "India");
        StudentClass obj = new StudentClass(123, "Chaitanya", ad);
        System.out.println(obj.rollNum);
        System.out.println(obj.studentName);
        System.out.println(obj.studentAddr.streetNum);
        System.out.println(obj.studentAddr.city);
        System.out.println(obj.studentAddr.state);
        System.out.println(obj.studentAddr.country);
   }
}
```

CODE OUTPUT :

123
Chaitanya
55
Agra
UP
India



Topic: 4.3.1 Programming paradigms

The above example shows the **Aggregation** between Student and Address classes. You can see that in Student class, we have used the Address class to obtain the Student class.

Why do we need Aggregation?

To maintain code re-usability. To understand this, let's consider the above example again. Suppose there are two other classes **College** and **Staff** along with two classes **Student** and **Address.** In order to maintain Student's address, College Address and Staff's address, we don't need to use the same code over and over again. We just have to use the reference of Address class while defining each of these classes like:

Student **Has-A** Address (Has-a relationship between student and address)
College **Has-A** Address (Has-a relationship between college and address)
Staff **Has-A** Address (Has-a relationship between staff and address)

Hence, we can improve code re-usability by using the Aggregation relationship.

Definitions

A **class** describes the variables and methods appropriate to some real-world entity.

An **object** is an instance of a class and is an actual real-world entity.

Inheritance is the ability of a class to use the variables and methods of a class from which the new class is derived.

Aggregation is a directional association between two classes to help improve code re-usability.

Topic: 4.3.1 Programming paradigms

Declarative Programming

In declarative programming, the programmer can simply state what is wanted having declared a set of facts and rules. We now look at how this works using examples of Prolog scripts. In order to do this, we shall use the following facts.

```
female(jane).
female(anne).
female(sandip).
male(charnjit).
male(jaz).
male(tom).
parent(jane,mary).
parent(jane, rajinder).
parent(charnjit, mary).
parent(charnjit, rajinder).
parent(sandip, atif).
parent(jaz, atif).
```

Remember that variables must start with an uppercase letter; constants start with a lowercase letter.

Suppose we ask

```
male(X)
```

Prolog starts searching the database and finds male(charnjit) matches male(X) if X is given the value charnjit. We say that X is *instantiated* to charnjit. Prolog now outputs:

```
X = charnjit
```

Prolog then goes back to the database and continues its search. It finds male(jaz) so outputs

```
X = jaz
```

And again continues its search. It continues in this way until the whole database has been searched. The complete output is

```
X = charjit
X = jaz
X = tom
No
```

The last line means that there are no more solutions.
The query male(X) is known as a goal to be tested. That is, the goal is to find all X that satisfy male(X). If Prolog finds a match, we say that the search has succeeded and the goal is true. When the goal is true, Prolog outputs the corresponding values of the variables.
Now we shall add the rule:

```
father(X,Y) :- parent(X,Y), male(X).
```

This rule states that X is the father of Y if (the :- symbol) X is a parent of Y AND (the comma) X is a male.

Topic: 4.3.1 Programming paradigms

The database now looks like this:

```
female(jane).
female(anne).
female(sandip).
male(charnjit).
male(jaz).
male(tom).
parent(jane,mary).
parent(jane, rajinder).
parent(charnjit, mary).
parent(charnjit, rajinder).
parent(sandip, atif).
parent(jaz, atif).
father(X, Y) :- parent(X, Y), male(X).
```

Suppose our goal is to find the father of rajinder. That is, our goal is to find all X that satisfy:

```
father(X, rajinder)
```

In the database and the rule the components female, male, parent and father are called predicates and the values inside the parentheses are called arguments. Prolog now looks for the predicate father and finds the rule:

```
father(X,Y) :- parent(X,Y), male(X)
```

In this rule, Y is instantiated to rajinder and Prolog starts to search the database for:

```
parent(X,rajinder)
```

This is the new goal, Prolog finds the match:

```
parent(jane, rajinder)
```

If X is instantiated to jane, Prolog now uses the second part of the rule:

```
male(X)
```

with X = jane. That is, Prolog's new goal is male(jane) which fails. Prolog does not give up at this stage but *backtracks* to the match

```
parent(jane, rajinder)
```

and starts again, from this point in the database, to try to match the goal

```
parent(X, rajinder)
```

This time, Prolog finds the match:

```
parent(charnjit, rajinder)
```

with X instantiated to charnjit. The next step is to try to satisfy the goal

```
male(charjit)
```

Topic: 4.3.1 Programming paradigms

This is successful so

```
Parent(charnjitm rajinder) and male(charjit)
```

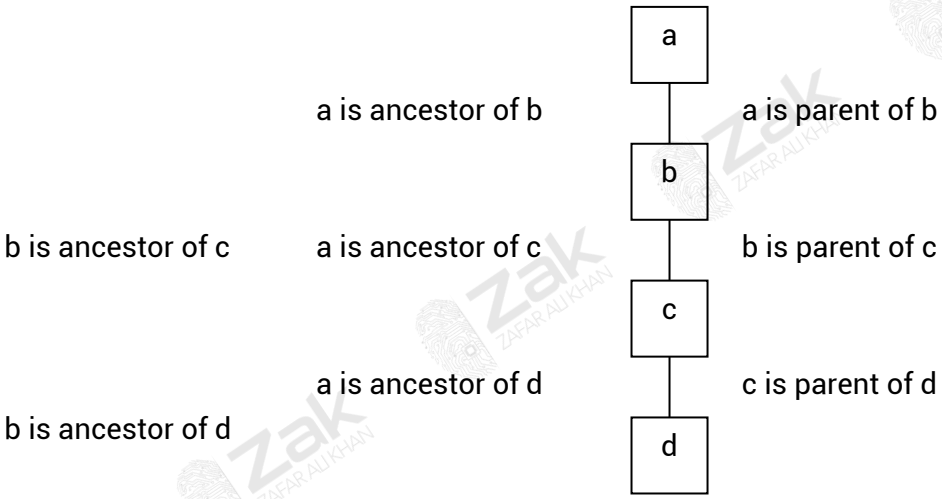
Are true. Thus father(charnjitm rajinder) is true and Prolog returns

```
X = charnjit
```

Prolog continues to see if there are any more matches. Since there aren't any more matches, Prolog outputs:

```
No
```

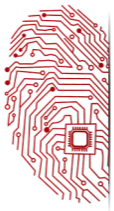
A powerful tool is Prolog is recursion. This can be used to create alternative versions for a rule. This is demonstrated in the figure below and shows how ancestor is related to parent.



This shows that X is an ancestor of Y if X is a parent of Y. But it also shows that X is an ancestor of Y if X is a parent of Z and Z is a parent of Y. It also shows that X is an ancestor of Y if X is a parent of Z , and Z is a parent of W and W is a parent of Y. This can continue forever. Thus the rule is recursive. In Prolog we require two rules that are written as

```
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z),
                  ancestor(Z, Y).
```

The first rule states that X is an ancestor of Y if X is a parent of Y. This is saying that a is an ancestor of b, b is an ancestor of c and c is an ancestor of d.



Topic: 4.3.1 Programming paradigms

The second rule is in two parts. Let us see how it works which represents the database

```
parent(a, b).
parent(b, c).
parent(c, d).
```

by setting the goal

```
ancestor (a, c).
```

Prolog finds the first rule and tries to match parent(a, c) with each predicate in the database. Prolog fails but does not give up. It backtracks and looks for another rule for ancestor. Prolog finds the second rule and tries to match

```
parent(a, Z).
```

It finds

```
parent(a, b)
```

so instantiates Z to b.

This is now put into the second part of the rule to produce

```
ancestor(b, c).
```

This means that Prolog has to look for a rule for ancestor. It finds the first rule

```
ancestor(X, Y) :- parent(X, Y).
```

Thus Prolog looks for

```
parent(b, c)
```

and succeeds.

This means that with X = a, Y = c we have Z = b and the second rule succeeds. Therefore Prolog returns

```
Yes.
```

Now try to trace what happens if the goals are

```
ancestor(a,d)
```

and

```
ancestor(c, b).
```

Topic: 4.3.1 Programming paradigms

You should find that the first goal succeeds and the second fails. This form of programming is based on the mathematics of predicate calculus. Predicate calculus is a branch of mathematics that deals with logic. All we have done in this Section is based on the rules of predicate calculus. Prolog stands for Programming in LOGic and its notation is based on that of predicate calculus. In predicate calculus the simplest structures are atomic sentences such as

Mary *loves* Harry

Philip *likes* Zak

The words in italics are part of the names of relationships. We also have atomic conclusions such as

Mary *loves* Harry if Harry *loves* Mary

John *likes* dogs if Jane *likes* dogs

Frank *likes* x if x *likes* computing

In the last atomic conclusion, x is a variable. The meaning of the conclusion is that Frank likes anything (or anybody) that likes computing.

Joint conditions use the logical operators OR and AND, examples of which are

John likes x if x is female AND x is blonde
Mary loves Bill OR Mary loves Colin if Mary loves Don

The atomic formulae that serve as conditions and conclusions may be written in a simplified form. In this form the name of the relation is written in front of the atomic formula. The names of the relations are called predicate symbols. Examples are

```
loves(Mary, Harry)
likes(Philip, Zak)
```

We use the symbol ← to represent if and have

```
loves(Mary, Harry) ← loves(Harry, Mary)
likes(John, dogs) ← likes(Jane, dogs)
```

The AND is represented by a comma in the condition part of the atomic conclusion. For example

```
likes(John, x) ← female(x), blonde(x)
```

Topic: 4.3.1 Programming paradigms

The OR is represented by a comma in the conclusion. For example

```
female(x), male(x) ← human(x)
```

These examples show the connection between Prolog and predicate calculus. You do not need to understand how to manipulate the examples you have seen any further than has been shown in this Section.

AS with the previous Section we include here the definitions of terms used in this Section. Remember, they can be used when a question says "State the meaning of the term …

Definitions

Backtracking is going back to a previously found successful match in order to continue a search.

Instantiation is giving a variable in a statement a value.

Predicate logic is a branch of mathematics that manipulates logical statements that can be either True or False.

A *goal* is a statement that we are trying to prove whether or not it is True or False.

Imperative programming

This topic has been covered in detail in chapter 2.3