Module 6 (first part)
High Level Language Programming

6.1 Algorithms, Flowcharts and Pseudocode

An algorithm is a step by step description of how a problem should be solved. Algorithms can be expressed in many kinds of notation including:

- Natural languages (e.g. Maltese, English etc.)
- Pseudocode
- Flowcharts
- Programming languages

6.1.1 Flowcharts

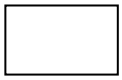
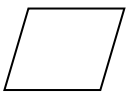
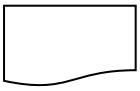
A flowchart describes an algorithm graphically by special symbols each having a particular meaning. There are basically two types of flowcharts:

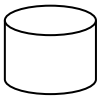
- System flowchart
- Program flowchart

A system flowchart shows how data flows in an information system. A program flowchart is a detailed description of the logic of a program.

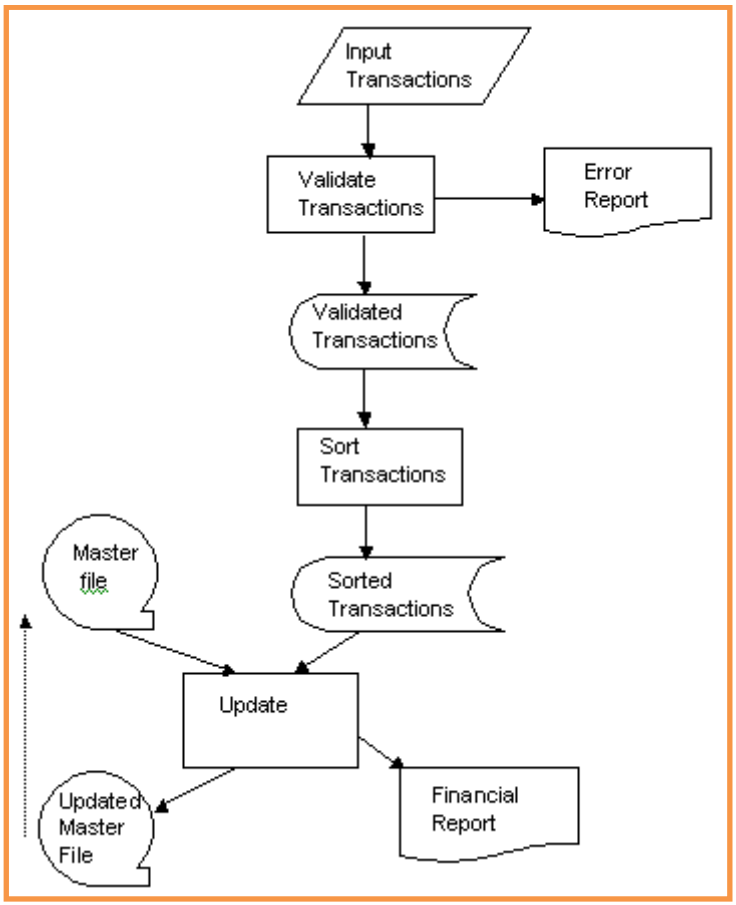
6.1.1.1 System Flowchart

The following table shows some symbols used in system flowcharts.

Symbol	*Symbol Name*	*Symbol Description*
	Process	Shows a statement or a number of statements. A statement directs the computer to perform a specified action.
	Input / Output	Indicates input to or output from a process.
	Document	Printed or written data.
	Sequential Access Storage	For example a file stored on tape.
	Stored Data	For example file stored on disk.

	Magnetic Disk	This symbol depicts a database.

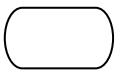
System Flowchart Symbols

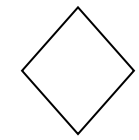
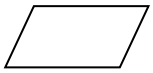


A System Flowchart

6.1.1.2 Program Flowchart

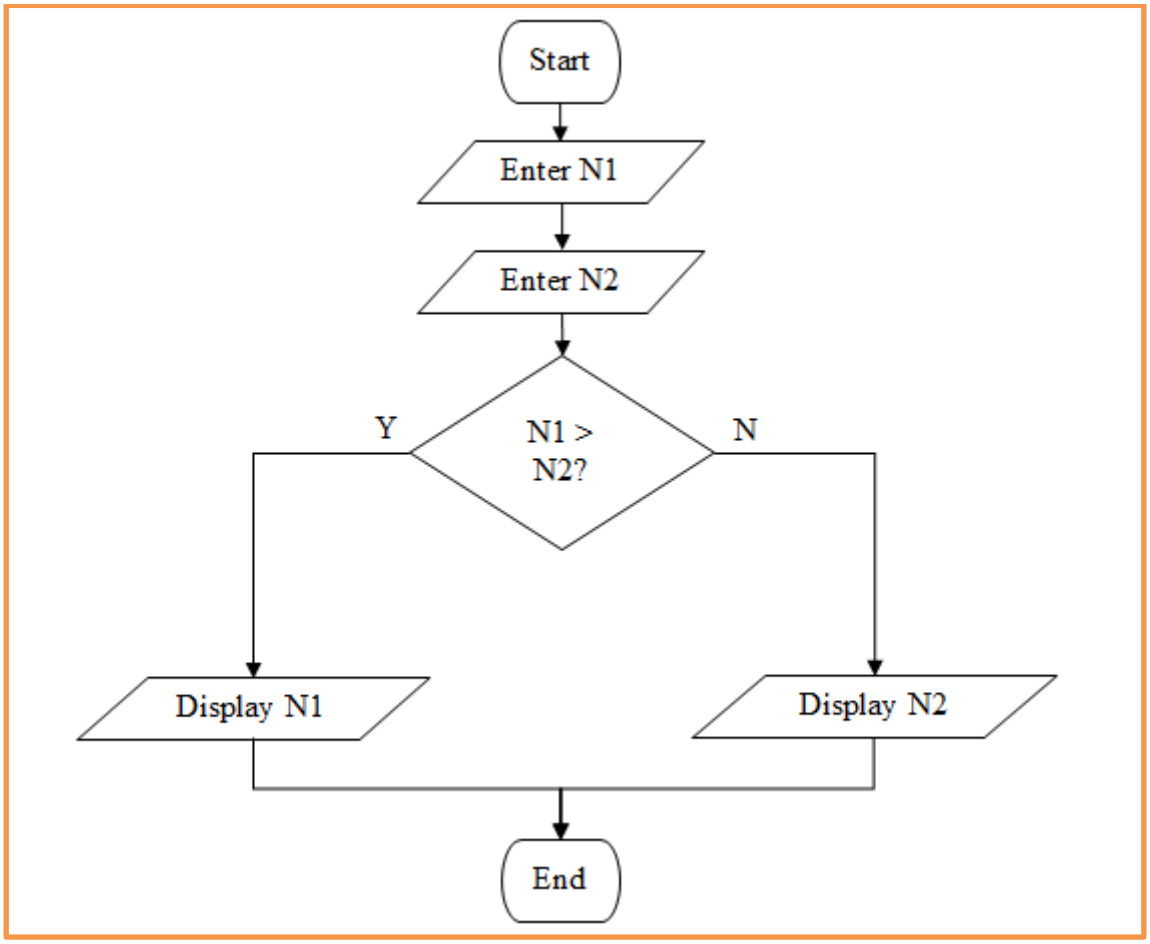
Below is a table showing the basic program flowchart symbols.

Symbol	Symbol Name	Symbol Description
	Terminator	Terminators show the start and stop points in a process
	Process	Shows a statement or a number of statements. A statement directs the computer to perform a specified action.
	Predefined Process (Subroutine)	A predefined process symbol is a marker for another process that is defined elsewhere.

	Decision	Indicates a question or branch in the process flow. Typically a decision shape is used when there are 2 options but can be used when there are more than two options.
	Input / Output	Indicates input to or output from a process.
	Connector	This symbol is used to connect two parts, on the same page, together.
	Off-page connector	This symbol is used to connect two parts of the same flowchart distributed on two pages.

Program Flowchart Symbols

The following diagram shows a simple example of a flowchart. In the algorithm two numbers N1 and N2 are inputted and the largest of them is displayed on the monitor.



A Program Flowchart

6.1.2 Pseudocode

Pseudocode (besides flowcharts and other techniques) is another way to express algorithms. When writing the algorithm in pseudocode the author builds the

algorithm on the writing style expressed by high-level languages. The pseudocode is actually a draft of a program. It is a program written without using the exact syntax. Later the pseudocode can be transformed into real code with relative ease. Pseudocode is a mixture of program constructs and a natural language (in our case English).

The following algorithm is the one expressed as flowchart in the preceding section. It finds the larger of two numbers.

```
begin
      input number N1
      input number N2
            if N1 is larger than N2
                  then display N1
                  else display N2
end
```

Pseudocode of a program that finds the larger of two numbers

6.1.3 Exercise

Express the following algorithms both as Flowcharts and as Pseudocode.

a) A program asks the user to give it the name of the user's father. Then it asks for the father's age. It then asks for the user's name and finally his or her age. The program calculates the difference between the ages and then displays the sentence "Your father is … years bigger than you".

b) A bus goes on a trip and makes one stop in the middle. In this stop a number of people leave the bus and other new passengers get on the bus. Write a program that asks how many passengers started the trip, how many passengers stopped in mid-trip and how many passengers went into the bus in mid-trip. The program will then display the number of passengers that arrived at the end of the trip.

c) A program asks a user to enter the number of sides of a polygon. If the number 4 is entered then the program displays the message "the shape is a quadrilateral. Otherwise it displays the message "the program is not a quadrilateral".

d) This program is about whether profit was made from the sale of an object. The program asks the user "how much was the object bought?" and the user enters the price. Then the program asks how much it was sold. The program then displays either the message "a profit was made" or "no profit was made".

e) A club is organising a buffet. When one books for the buffet one will pay €12 per person if the number of persons being booked is more than 4. Otherwise the price is €15 per person. Write a program that asks the user to enter the number of persons to be booked for the buffet and then the program will display the total amount to be paid.

f) A businessman organising a meal in a restaurant. Adults pay €10 and children €6. If the total price to be paid exceeds €30, a discount of 10% is

given. Write a program that calculates the amount to be paid after being given the number of adults and children.

6.2 High Level Language (Java) Structures

A high level language offers the programmer the following structures:

- Linear structure
- Selection structure
- Repetition structure
- Branch structure

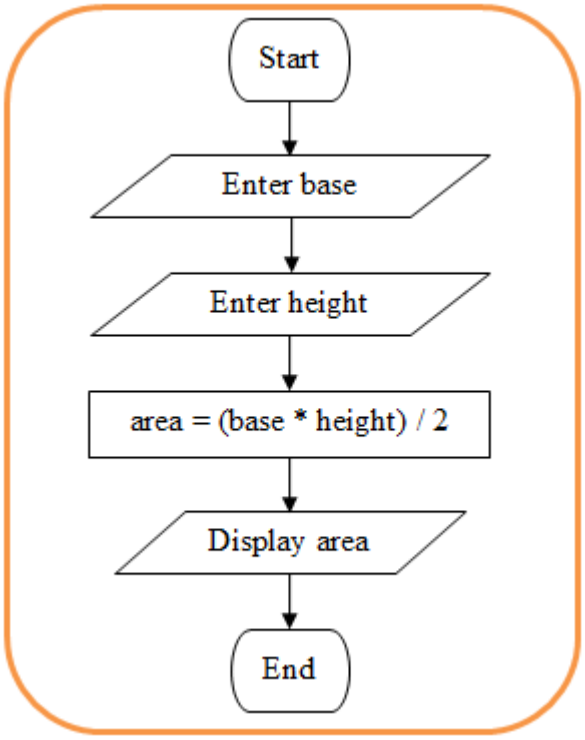
6.2.1 Linear structure

'Linear' means that instructions are executed sequentially from top to bottom. Consider a program that calculates the area of a triangle.

```
begin

    display "Area of a triangle"
    input base
    input height
    area = (base * height) / 2
    display area

end
```



Finding the Area of a Triangle

6.2.1.1 Exercise

Express the following programs by means of a flowchart and by pseudocode:

(a) A program is given a value for the temperature in Centigrade. The program calculates its Fahrenheit equivalent and displays it on the monitor. The formula for the conversion from Centigrade to Fahrenheit is F = C*9/5+32.

(b) A program is given the value of a radius. It first calculates the area of circle with that radius and then calculates the volume of a sphere with the same radius. It then displays the results on the monitor.

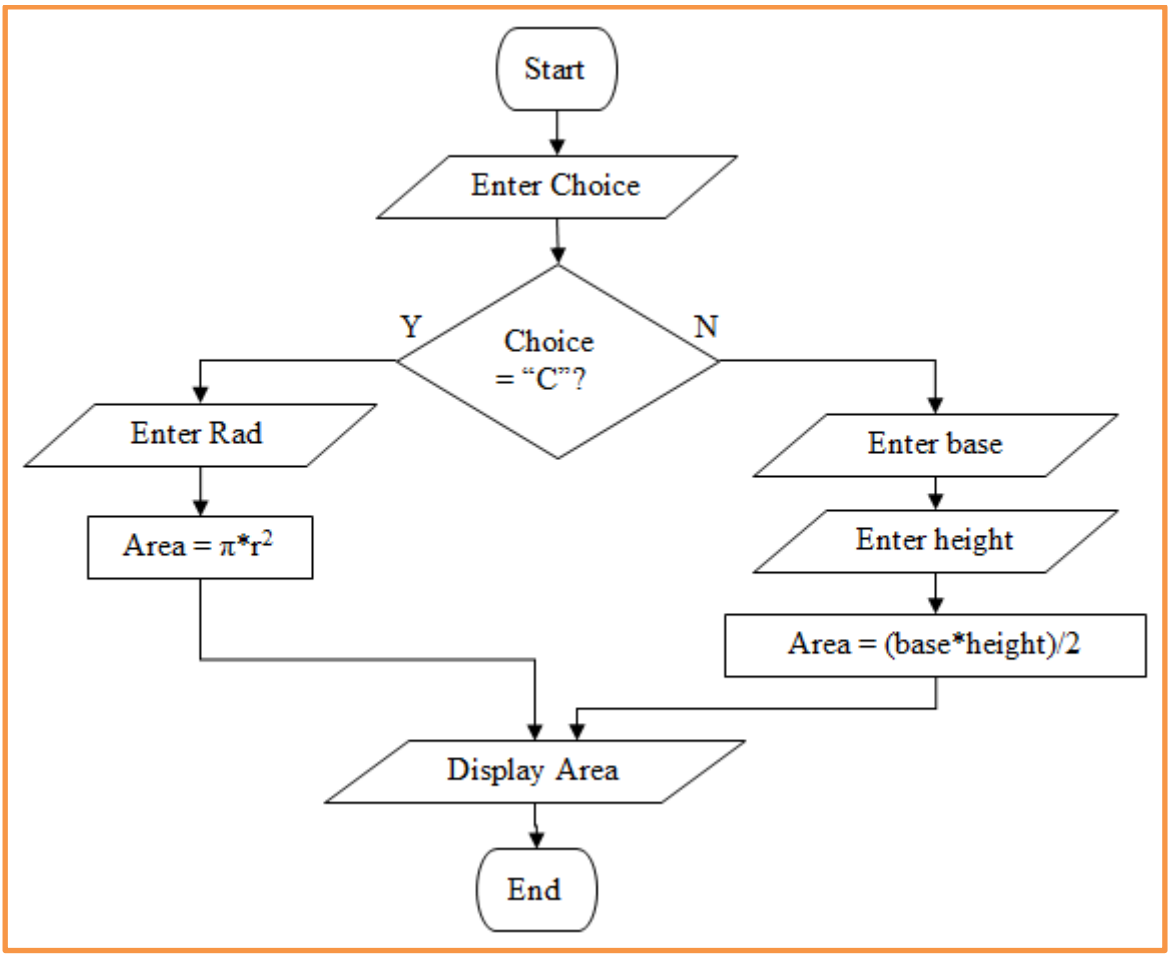
6.2.2 Selection structure

Selection structure is also known as selective or conditional structure. In these structures a condition indicates to the program which statements to execute. There are two selection structures. These are:

(a) if-then-else statement
(b) switch (or 'case') statement

6.2.2.1 If-then-else structure

The if-then-else structure makes the program choose one of two possible statements. An example in pseudocode and flowchart is shown hereunder.



A program making use of the if..then..else statement

6.2.2.1.1 Exercise

1 Express the flowchart in the diagram above in pseudocode.

2 Express the following algorithms in pseudocode and as a flowchart.

(a) A program receives in input a name of a person and his/her age. If the age is less than 18 the program displays the message "you cannot vote". Otherwise it displays the message "you can vote".

(b) A program receives in input a mark. If the mark is less than 50 the program displays the word “fail”. Otherwise it displays the word “pass”.

(c) A program receives in input a mark. If the mark is less than 50 it displays the word “fail”. Otherwise if the mark is less than 75 it displays the word “pass”. In the case the mark is 75 or over it displays the word “distinction”.

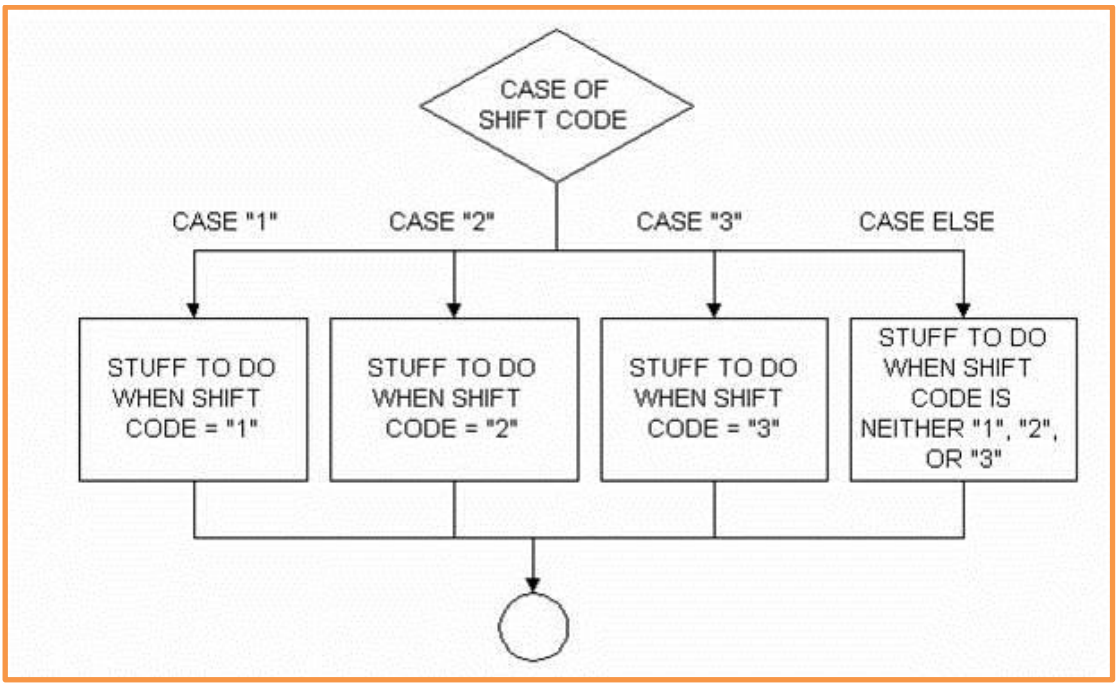
(d) A program that is given three numbers and it displays the largest one.

(e) A program receives three numbers and it decides whether they were entered in ascending order or not.

(f) A program receives three numbers in input and displays the numbers in ascending order.

6.2.2.2 Switch structure

The switch (or case) statement is used when the program has to follow one out of more than two options. The diagram below shows how this can be represented by means of a flowchart.



An example of the switch statement represented by a flowchart

6.2.3 Repetition structure

The repetition structure is also known as iterative or loop structure. There are three such structures. In these structures a sequence of statements is repeated if a condition is met. The three structures are the following:

- while statement
- do-while (or ‘repeat-until’) statement
- for statement

6.2.3.1 While statement

The following diagram shows a snippet of a program written in Java making use of the while loop.

```
int count = 1;
   while (count < 11)
   {
      System.out.println("Count is: " + count);
      count++;
   }
```

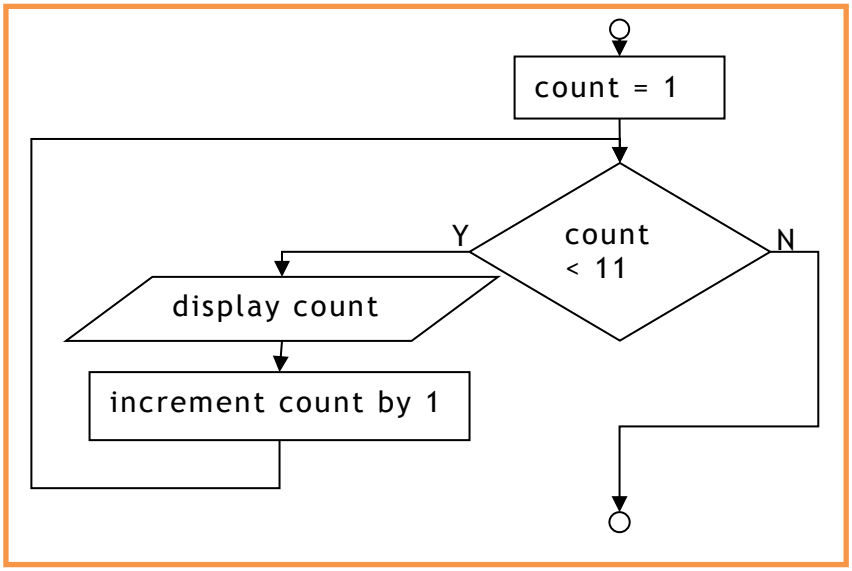
A snippet in Java making use of the while loop

The pseudocode of the above program can be expressed as:

```
give initial value of 1 to 'count'
while ('count' is less than 11)
      begin
            display count
            increment count by 1
      end
```

Pseudocode of the program in previous snippet

The flowchart is represented in the following diagram.



Flowchart of the same snippet

The 'while loop' is called Pre-Tested because the condition is tested at the start of the loop. The flowchart of this program is shown below.

6.2.3.1.1 Exercise

What does the snippet shown in the previous diagrams do?

6.2.3.2 Do-while statement

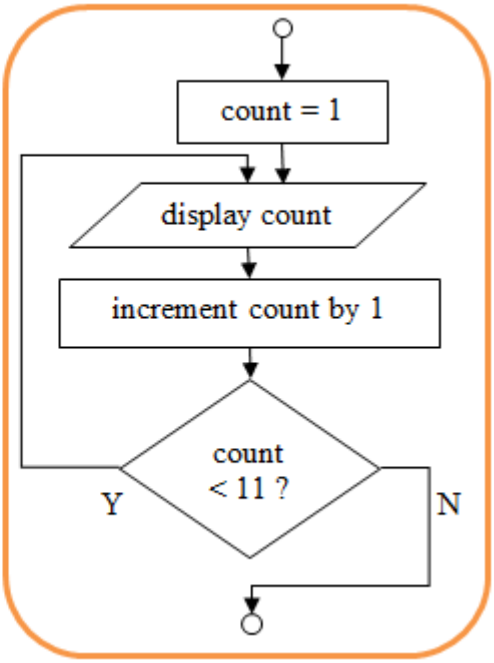
In some languages (e.g. in Pascal) the do-while statement is represented as a repeat-until statement with the same effect. The following code from a Java program makes use of the do-while statement.

```
int count = 1;
do
    {
        System.out.println("Count is: " + count);
        count++;
    } while (count < 11);
```

A snippet of a program in Java that makes use of the do…while statement

This kind of loop is called post-tested because the test that decides whether the loop should be repeated or not is found at the end of the loop. The pseudocode and flowchart of the above snippet are shown in the diagram below.

```
initiate 'count' with value 1
  do
     display 'count'
     increment the value of 'count' by 1
  while (the value of 'count' is less than 11)
```



Pseudocode and flowchart of the previous snippet

6.2.3.2.1 Exercise

a) What does the program shown in the last diagrams do?

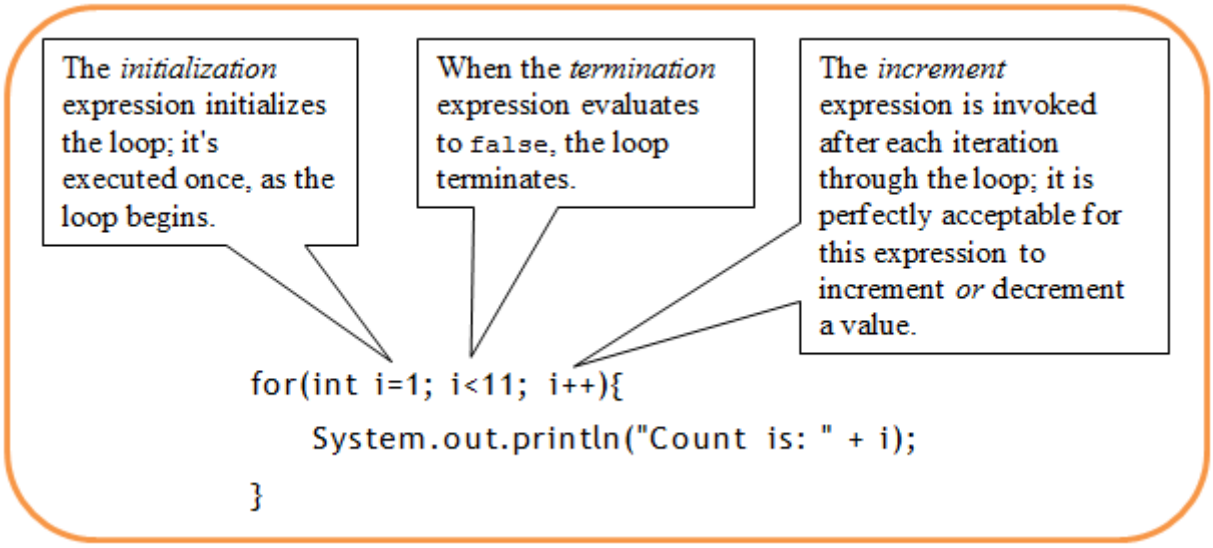
b) Which of the following statements are true?

(i) The code in the while statement is always executed at least once.

(ii) The code in the do..while statement is always executed at least once.

6.2.3.3 For statement

An example of the for-statement as used in Java is shown in the diagram below.

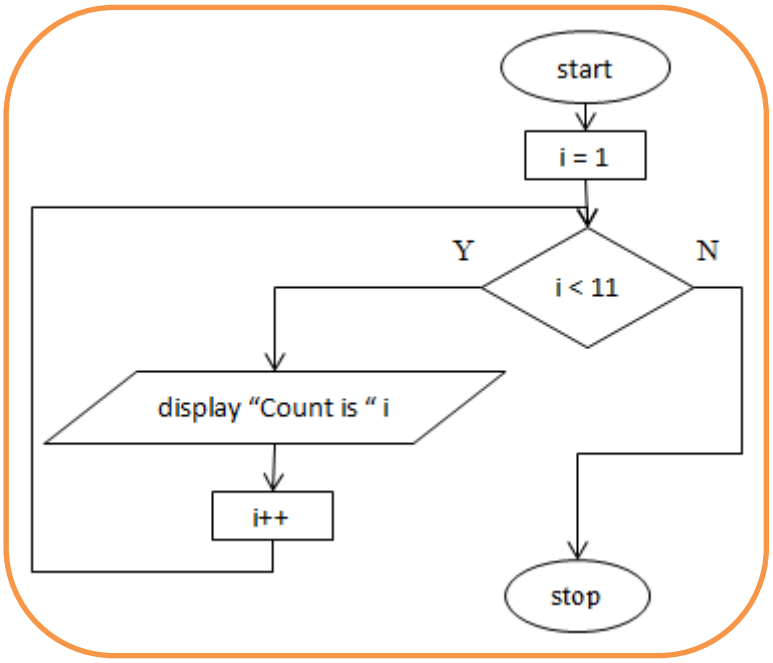


Use of the For loop

One way to express the above snippet in pseudocode is the following. Following pseudocode is a flowchart to express the logic of the loop.

```
for i = 1 to 10 do
    begin
        display ( i )
    end
```

Pseudocode of snippet shown above



Flowchart of snippet shown above

6.2.3.4 Nested Loops

Loops can be coded inside each other and the term for this is 'nested loops'.

6.2.3.5 Exercise

Represent the following algorithms as flowcharts and also as pseudocode. You can choose any loop you like.

(a) A program displays the values 3, 6, 9 … 36.

(b) A program displays the values 6, 11, 16 ... 66.

(c) A program is given a whole number n and then displays the first n multiples of 10 e.g. if it is given 3 it displays 10, 20, 30.

(d) A program receives in input a sequence of numbers. When 0 is entered it means that there are no more numbers to input. The program finds the sum of the inputted numbers.

(e) A program receives in input a sequence of numbers. When 0 is entered it means that there are no more numbers to input. The program counts how many numbers have been entered. The 0 is not counted.

(f) A program receives in input a sequence of numbers. When 0 is entered it means that there are no more numbers to input. The program finds the average of the inputted numbers. 0 is not counted as one of the numbers.

(g) A program receives in input a sequence of numbers. When 0 is entered it means that there are no more numbers to input. The program finds the total of the positive numbers.

6.2.4 Branch statements

This structure is also known as jump structure. The branch statements in Java are the following:

(a) break statement
(b) continue statement
(c) goto statement
(d) return statement

These allow the programmer to break off from the sequential order and, apart from the return statement, have to be used with care preferably not used at all.

6.3 Data Structures

A data structure is an organisation of data. There are many different ways how data can be organised. Each data structure has its own operations that can be performed on the data e.g. to add data, to delete data, to sort data etc. The purpose of a data structure is to keep data in relation with one another.

6.3.1 Static and Dynamic Data Structures

Some data structures after being created cannot be increased or decreased in size i.e. the maximum number of elements that the data structure can hold is defined once during the creation of the data structure. That number is fixed.

Such data structures are called ‘static’. An array as defined in Java (and other programming languages) is a static data structure.

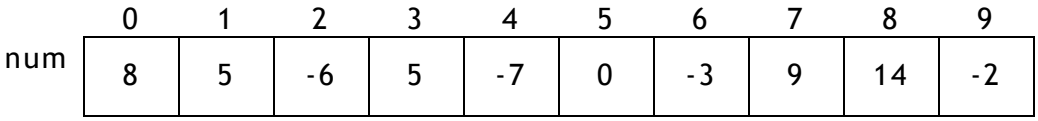
On the other hand ‘dynamic’ data structures when created can increase or decrease in size according to how many elements it contains. A file of records is one example of a dynamic data structure.

6.3.2 Arrays

An array is an indexed collection of data values of the same type.

6.3.2.1 One-Dimensional Arrays

A one-dimensional array is a sequence of elements of the same type. Each element of the one-dimensional array is referred to by an index.



Array num

A few expressions are the following:

(a)	num[1] – 3	(this expression evaluates to 1)
(b)	2*num[7] + num[0]	(this expression evaluates to 26)
(c)	num[2+4]	(this expression evaluates to -3)
(d)	num[12]	(this expression is invalid)

The following are a few assignment statements (a statement is a sentence in a HLL that tells the computer what to do):

a) num[4] = num[2] + 7; (this statement changes the value of num[4]; num[4] is now equal to 1)

b) i = 6;
num[i] = num[i+1]; (this statement changes the value of num[6]; num[6] is now equal to 9)

Applications of One-Dimensional Arrays

One-dimensional arrays can be used to sort elements. For example the elements from a file (on disk) can be copied in an array, then sorted, then put in a new file on disk. One-dimensional arrays can also be used to implement other data structures like queues and stacks.

6.3.2.1.1 Exercise on One-Dimensional Arrays

a) Considering the array ‘num’ in shown above and evaluate the following expressions:

i) num[8] + 12

ii) 3*num[9] + 2*num[2]

iii) 2*num[2 + 5]

b) Consider again the array 'num' and show how the array changes after the execution of the following assignment statements:

i) num[3] = 2*num[2] + 7;

ii) num[8] = num[1 + 5] + 2*num[4];

iii) i = 4;
num[i] = i*num[3] - 8*i;

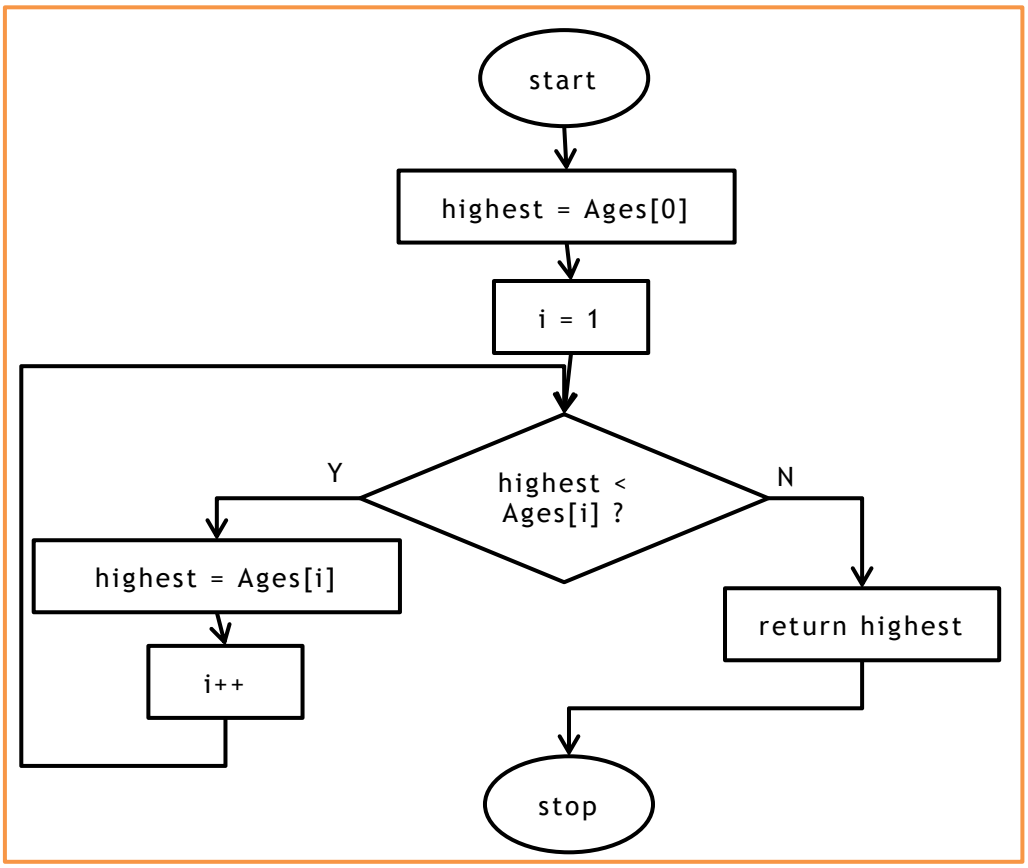
iv) i = 2;
j = 4;
num[i + j] = 3*num[3*i] + 2*num[j];

v) for t = 0 to 9 do
num[t] = num[t] + 3;

c) Write a program in Java declares a one-dimensional array called Ages having 8 elements. The program offers the following options:

(i) Fill the array with values
(ii) Display on the screen all the values of the array
(iii) Display the highest age
(iv) Display the lowest age
(v) Display the average age
(vi) Display the range of ages (i.e. the difference between highest and lowest ages)
(vii) Quit program

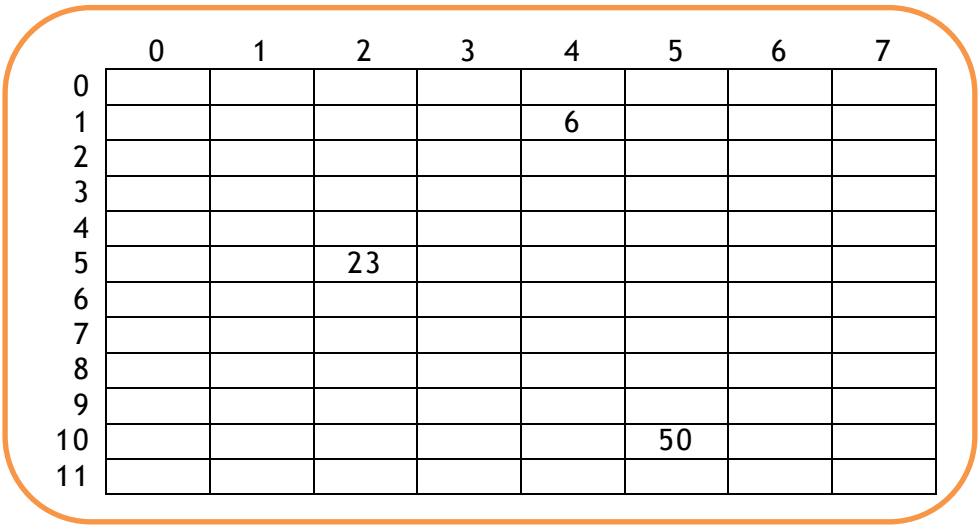
For each of the above options draw a flowchart or write it in pseudocode. The one for the highest age has been done for you as a flowchart.



Flowchart to find highest age in array

6.3.2.2 Multi-Dimensional Arrays

Let us take an example of a 2-dimensional array named Sales (see diagram below).



	0	1	2	3	4	5	6	7
0								
1					6			
2								
3								
4								
5			23					
6								
7								
8								
9								
10						50		
11								

Array Sales

In this example the horizontal indexes (0 to 7) represent Salespersons while the vertical indexes (0 to 11) represent codes of objects.

In Java the table would be created thus:

```
int Sales [] [] = new int [12] [8];
```

Using the Java notation, Sales [5] [2] is equal to 23. This means that salesperson 2 sold 23 items whose code is 5. The following are some expressions (referring to the array Sales):

(a) Sales [10] [5] + 2 * Sales [1] [4] (evaluates to 62)
(b) Sales [5] [4 div 2] – 15 (evaluates to 8)

6.3.2.2.1 Applications of Multi-Dimensional Arrays

The Sales array is an example of the use of a 2-dimensional array. In this case we had two indices that represented 'Salesperson' and 'Code of Item'. Each array element represents the number of items sold. A digital picture is another example of a two-dimensional array where each element (pixel) represents a colour. The mathematical branch of matrices can be programmed such that each matrix is a 2-dimensional array. There can be various applications for multi-dimensional arrays depending on the subject being programmed.

6.3.2.2.2 Exercise on Multi-Dimensional Arrays

(a) Draw an array similar to Sales above that contains 4 rows and 3 columns. Call this array 'ArNum'. This array will hold integers.

(b) Show how this array is created in Java.

(c) Fill some values in the array as shown here:

(i) ArNum [2] [1] = 7 (assume that the first index represents the row)

(ii) ArNum [0] [0] = -3

(iii) For i=1, 2 and 3 ArNum [i] [0] = 2 * i.

(iv) For the other elements ArNum [i] [j] = i + j.

(d) Evaluate the following expressions:

(i) ArNum [2] [1] + 2 * ArNum [3] [0].

(ii) Add all ArNum [1] [j] for j = 0, 1 and 2.

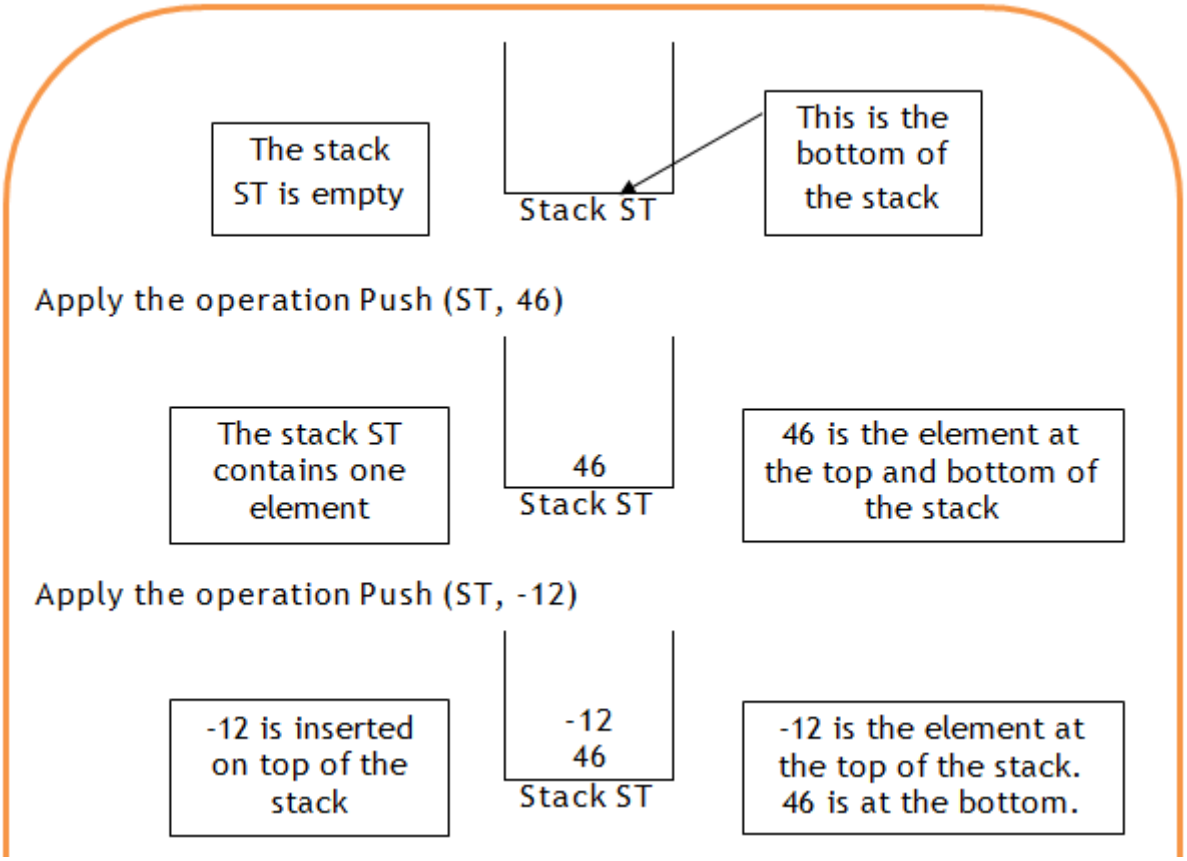
(iii) Add all ArNum [i] [2] for i = 0, 1, 2 and 3.

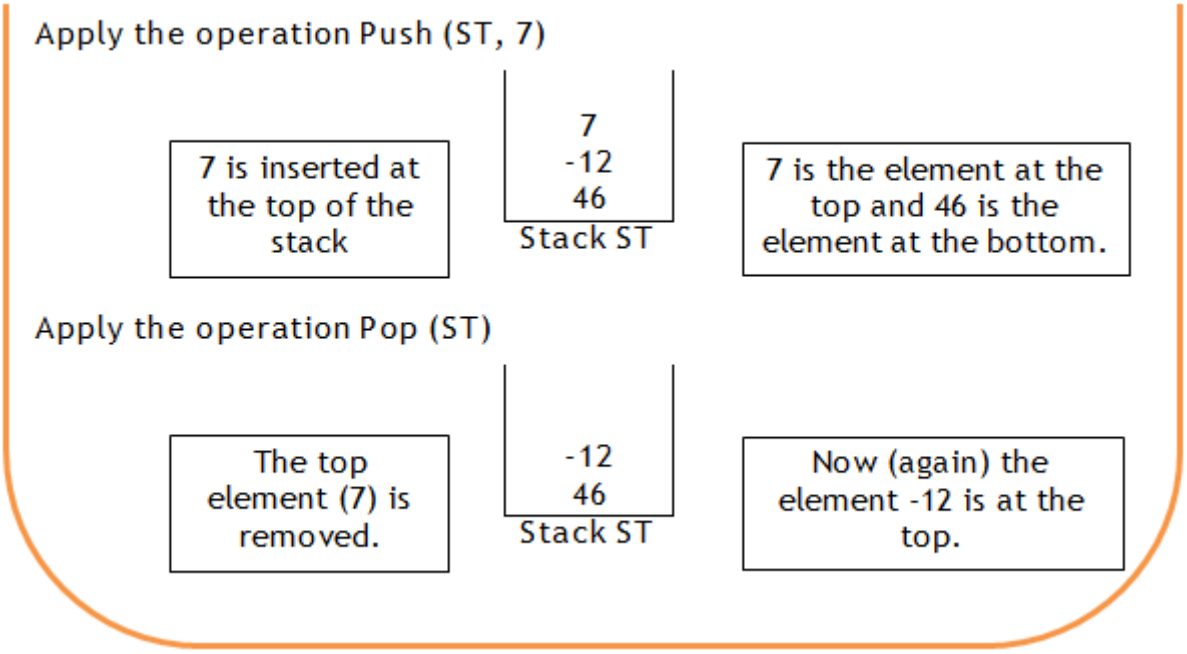
6.3.3 Stacks

A stack is a data structure that simulates a stack of items put one above another in such a way that when a new item is included in the stack it is put on top and when an item is removed it is the one at the top that is removed. This is called a LIFO (Last In First Out) discipline. When an item is put on top of the stack the operation is called 'push'. When the top item is removed the operation is called 'pop'.

6.3.3.1 An Example of a Stack

Let us imagine a stack of integers named ST. The following diagram shows a number of operations on stack ST.





Operations on a stack

6.3.3.2 Applications of Stacks

There are various applications of stacks in computer science. One of them is its use to keep track of procedure returns i.e. if a procedure A is called and inside A a procedure B is called and inside B a procedure C is called then when procedure C ends the computer has to continue executing procedure B and when this ends then procedure A has to continue its execution. To keep track where the computer must go to continue a procedure a stack is used.

6.3.3.3 Exercise on Stacks

A stack named STCK that holds numbers is initially empty. Show how STCK will appear after the following operations:

```
push (STCK, 5)
push (STCK, 8)
push (STCK, -3)
n = pop (STCK)
push (STCK, 7-n)
p = pop (STCK)
q = pop (STCK)
r = pop (STCK)
push (STCK, p - q + r)
```

6.3.4 Queues

A queue is a data structure that follows a FIFO (First In First Out) discipline. It follows the principles of a normal queue where elements arriving at the queue are placed at the end of the queue and the element that leaves the queue is the first element. The Queue has two basic operations which are 'insert' (also called 'enqueue') and 'remove' (also called 'dequeue').

6.3.4.1 An Example of a Queue

Let the queue name be 'Jobs' and let it hold integers.

Create queue Jobs	Jobs	(This is an empty queue)
Insert (Jobs, 7)	Jobs 7	(7 is the first and last element)
Insert (Jobs, 12)	Jobs 7 12	(7 is the first element, 12 is the last)
Insert (Jobs, 3)	Jobs 7 12 3	
Remove (Jobs)	Jobs 12 3	
Insert (Jobs, 22)	Jobs 12 3 22	
Remove (Jobs)	Jobs 3 22	

An example of a queue

6.3.4.2 Applications of Queues

Two applications of stacks are the Spooler, where a printer keeps a note of all the files to be printed. When a file is printed its name and location are removed from the queue and the next file in the queue is printed. A file that requires printing is put at the end of the queue if the printer is busy. Another example of the use of a queue is the keyboard buffer where characters pressed on the keyboard are kept in a queue before being passed to RAM.

6.3.4.3 Exercise on Queues

A queue named QU holds RAM addresses. Follow the steps below to show how QU is changing:

Insert (QU, AB6h)
Insert (QU, 24Fh)
Insert (QU, 109h)
Remove (QU)
Remove (QU)
Insert (QU, 10Ah)

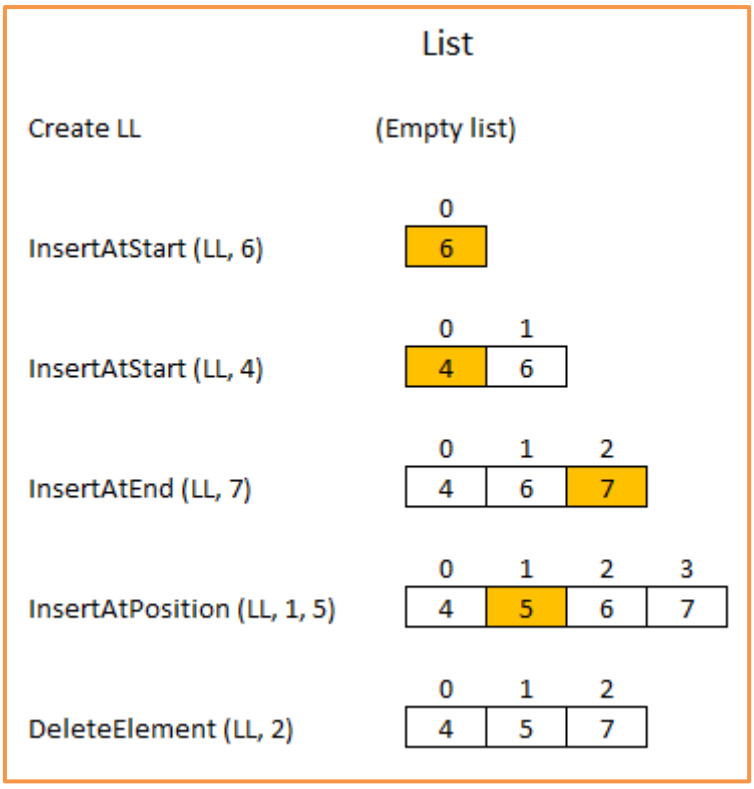
6.3.5 Linear Lists

Like an array, a linear list stores a collection of objects of a certain type, usually denoted as the elements of the list. The elements are ordered within the linear list in a linear sequence. Linear lists are usually simply denoted as lists.

Unlike an array, a list is a data structure allowing insertion and deletion of elements at an arbitrary position of the sequence. Lists are not limited to a certain maximum number of elements from the beginning on (like an array). So they are a dynamic data structure.

6.3.5.1 Example of a Linear List

Let the linear list be called LL. The following diagram shows a few operations on the list.



Operations on a linear list

6.3.5.2 Exercise on Linear Lists

Start with an empty list LST. How will LST look like after the following operations?

InsertAtStart (LST, 4)
InsertAtStart (LST, 7)
InsertAtEnd (LST, 3)
InsertAtStart (LST, 9)
DeleteElement (LST, 2)
InsertAtPosition (LST, 2, 8)
DeleteElement (LST, 1)

6.4 High Level Languages Features

A high-level computer language is one which is problem-oriented and machine-independent. Machine independent means that the programmer of a high-level language need not know the architecture of the computer (unlike for example the

assembly-language programmer). All one needs to know is the grammar of the language. Some languages, like Java, are called general purpose because by means of them one can program various programs in different fields. Others are written for particular areas for example FORTRAN is used for mathematical applications and COBOL is used for commercial applications.

6.4.1 Some Terms

Identifiers: Names we give to our variables, constants, classes, and methods.

Reserved words: Words that are part of the syntax of a high-level language. Each holds a specific meaning and cannot be used as an identifier e.g. while, double etc.

ASCII and Unicode: Two different codes that convert characters and control codes (like 'backspace', 'carriage return', etc.) into a number.

Escape sequence: A sequence of characters that have a special meaning e.g. \t which stands for 'insert a tab.

Literal: A fixed value. Examples are 4 (an integer literal), 56.7 (a floating point literal), "this is a string" (a string literal), True (a Boolean literal) etc.

Variable: A value that can change during the running of a program. A variable has (i) a name and (ii) a type. In Java (and in many other languages) all variables must first be declared before they can be used.

Constant: A value that cannot change during the running of a program.

6.4.2 Scope and Visibility of Variables

A variable's 'scope' refers to those parts of the program in which the variable can be used. For example if a variable is declared inside a method it cannot be used outside that method. We say that the scope of the variable is the method. We also say that the variable is visible within the method but is not visible outside the method.

```
class ScopeOfVariables
{
   static int a = 7;

   public static void main (String args[])
   {
      int b = 2;

      System.out.println ( a );

      System.out.println ( b );

      // System.out.println ( c );
      // This gives an error because c is defined
      // within meth1 and is not known within this method
```

```
        meth1();
        meth2();
    }

    static void meth1()
    {
        int c = 5;

        System.out.println ( a );
        // System.out.println ( b ); Error
        System.out.println ( c );
    }

    static void meth2()
    {
        int d = 4;

        for ( int i = 0; i<4; i++ )
            {
                System.out.println ( d*i*a );
            }

        System.out.println ( a );
        // System.out.println ( b ); Error
        // System.out.println ( c ); Error
        System.out.println ( d );
        // System.out.println ( i ); Error because i
        // is defined for use in the 'for' loop only.
    }
}
```

A program for studying the scope of variables

The following table gives the scopes of all the variables used in the above program.

Variable	*Scope*
a	The scope is the whole program because the variable is like a global variable.
b	The scope of ‘b’ is only the method ‘main’ since ‘b’ is defined within this method. Outside ‘main’ the variable ‘b’ cannot be accessed.
c	The variable ‘c’ is visible only inside method ‘meth1’ because it is inside this method that it is declared.
d	The variable ‘d’ is visible only inside method ‘meth2’ because it is inside this method that it is declared.
i	The scope of ‘i’ is only the ‘for’ loop within the method meth2.

A table that shows the scope of all the variables in the previous program

6.4.3 Data Types

A ‘type’ is a set of values and operations on these values. For example the type ‘int’ has values {…, -2, -1, 0, 1, 2, …} and operations {+, -, *, /, MOD, DIV,…}. The type ‘boolean’ has values {true, false} and operations {AND, OR, NOT, …}.

Some data types are referred to as ‘simple’ data types (or ‘atomic’ or ‘primitive’). A variable of a simple data type is made up of one value.

Some data types are called ‘structured’ because a variable declared of having such a type is made up of more than one value (i.e. it is a data structure). Examples are array, string or record.

Other types are called ‘reference’ types because a variable declared with such types will hold the ‘reference’ (‘address’ or ‘pointer’) to the (structured) value.

6.4.3.1 Java Primitive Types and Reference Types

The types of the Java programming language are divided into two categories:

- primitive types
- reference types

6.4.3.1.1 Java’s Primitive Types

The eight primitive data types supported by the Java programming language are:

- byte: The byte data type is an 8-bit signed two's complement integer. It has a minimum value of -128 and a maximum value of 127 (inclusive).
- short: The short data type is a 16-bit signed two's complement integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive).
- int: The int data type is a 32-bit signed two's complement integer. It has a minimum value of -2,147,483,648 and a maximum value of 2,147,483,647 (inclusive).
- long: The long data type is a 64-bit signed two's complement integer. It has a minimum value of -9,223,372,036,854,775,808 and a maximum value of 9,223,372,036,854,775,807 (inclusive).
- float: The float data type is a real number that ranges from approximately -3.40E+38 to 3.40E+38. It is a single-precision 32-bit floating point.
- double: The double data type is a double-precision 64-bit floating point. Its range of values is approximately from -1.80E+308 to 1.80E+308.
- boolean: The boolean data type has only two possible values: true and false. This data type represents one bit of information, but its "size" isn't something that's precisely defined.
- char: The char data type is a single 16-bit Unicode character. It has a minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65,535 inclusive).

6.4.3.1.2 Java’s Reference Type

The other types that are not primitive are reference types. That means that arrays are reference types, as are instances of classes i.e. if you declare a variable ‘xyz’ as an array the value contained in xyz is not the array but a reference (pointer, memory address) to where the array is. The following sections consider specific cases of reference types.

6.4.3.1.2.1 The Java String Type

Consider the following program in Java:

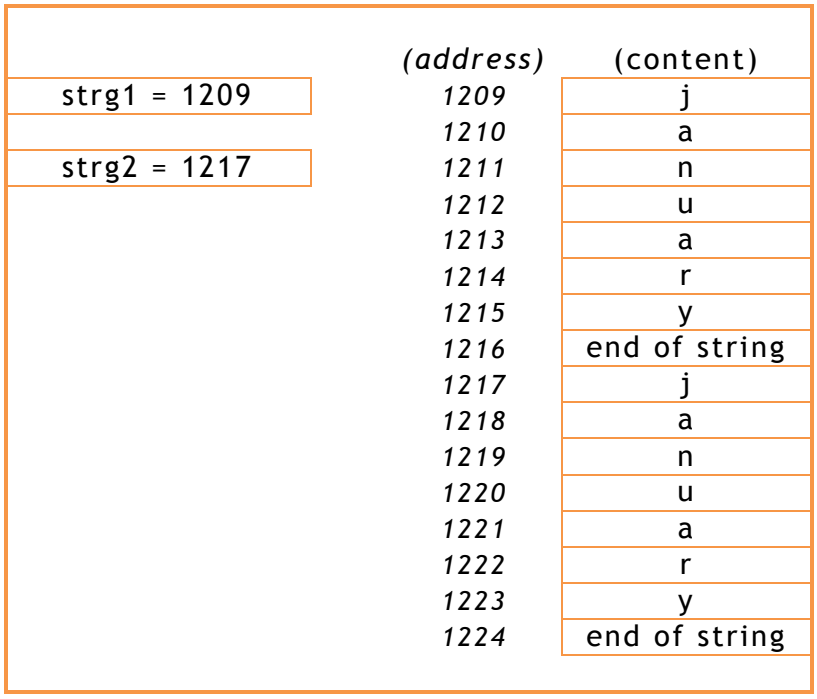
```
public class AboutStrings2
{
   public static void main (String args[])
   {
      String st1, st2;
      st1 = "november";
      st2 = "november";
      System.out.println (st1 == st2); //outputs 'true'
      System.out.println (st1.equals(st2)); //outputs 'true'

      String str1 = new String();
      String str2 = new String();
      str1 = "december";
      str2 = "december";
      System.out.println (str1 == str2); //outputs 'true'
      System.out.println (str1.equals(str2)); // outputs 'true'

      String strg1 = new String("january");
      String strg2 = new String("january");
      System.out.println (strg1 == strg2); //outputs 'false'
      System.out.println (strg1.equals(strg2)); // outputs 'true'
   }
}
```

A program about equating string variables

Note that when using the == operator st1, st2, str1 and str2 are treated like variables of primitive type while strg1 and strg2 are treated as variables of reference type. Consider the diagram below.



How two strings are memorised

The values of strg1 and strg2 are respectively 1209 and 1217. So strg1 is not equal to strg2. It is the content of their reference that is equal.

6.4.3.1.2.2 Classes as Types

Java itself is enriched with packages that hold classes that can be used by the user. There are hundreds of these classes. We mention only three. 'Scanner' is a class that can be used to enter information from the keyboard. 'LinkedList' is a class that implements the data structure Linked List. 'BigDecimal' is a class that is used when accurate arithmetic on very large numbers is required. A programmer can also define classes (called user-defined).

Most classes can be used as types where objects are created according to the definition of the class. This is analogous to creating a variable having a particular type. An object is a more complex comparison to a variable. The type of an object is a class.

Look at the following program that is made up of three classes.

```
class taxi
{
   //the taxi is paid at 1.2 euros per kilometre
   private final float RATE_PER_KM = 1.2f;

   private int kilom;

   //this is a constructor
   public taxi ()
   {
      kilom = 0;
   }

   //a second constructor
   public taxi (int k)
   {
      kilom = k;
   }

   //accessor
   public float getRate ()
   {
      return RATE_PER_KM;
   }

   //another accessor
   public int getKilom ()
   {
      return kilom;
   }

   public float payment ()
   {
      return kilom * RATE_PER_KM;
   }
}

class hiredCar
```

```
{
   //the hired car is paid 5c per kilometre
   private final float RATE_PER_KM = 0.05f;

   //the hired car is paid 5 euros per day
   private final float RATE_PER_DAY = 5.0f;

   private int kilom;

   private int days;

   //this is a constructor
   public hiredCar ()
   {
      kilom = 0;
      days = 0;
   }

   //a second constructor
   public hiredCar (int k, int d)
   {
      kilom = k;
      days = d;
   }

   //accessor
   public float getRatePerKm ()
   {
      return RATE_PER_KM;
   }

   //another accessor
   public float ratePerDay ()
   {
      return RATE_PER_DAY;
   }

   //another accessor
   public int getKilom ()
   {
      return kilom;
   }

   //another accessor
   public int getDays ()
   {
      return days;
   }

   public float payment ()
   {
      return (kilom * RATE_PER_KM + days * RATE_PER_DAY);
   }
}

import java.util.*;

class payments
```

```
{
   public static void main (String args[])
   {
      int choice;

      Scanner scan = new Scanner (System.in);
      scan.useDelimiter ("\n");

      do
      {
         System.out.println ();
         System.out.println ("PAYMENTS");
         System.out.println ();
         System.out.println ("1. Payment for Taxi");
         System.out.println ("2. Payment for Hired Car");
         System.out.println ("0. End Program");
         System.out.println ();
         System.out.print ("Enter your choice : ");
         choice = scan.nextInt();

         switch (choice)
         {
            case 1 : payForTaxi();
                  break;
            case 2 : payForHiredCar();
                  break;
         }
      }while (choice != 0);
   }

   static void payForTaxi ()
   {
      int km;
      float pay;
      taxi t;

      Scanner scan = new Scanner (System.in);
      scan.useDelimiter ("\n");

      System.out.println ();
      System.out.print ("How many kilometres? : ");
      km = scan.nextInt();

      t = new taxi (km);
      pay = t.payment();

      System.out.println ();
      System.out.println ("Payment = " + pay);
   }

   static void payForHiredCar ()
   {
      int km, dys;
      float pay;
      hiredCar hc;

      Scanner scan = new Scanner (System.in);
      scan.useDelimiter ("\n");
```

```
        System.out.println ();
        System.out.print ("How many kilometres? : ");
        km = scan.nextInt();

        System.out.println ();
        System.out.print ("How many days? : ");
        dys = scan.nextInt();

        hc = new hiredCar (km, dys);
        pay = hc.payment();

        System.out.println ();
        System.out.println ("Payment = " + pay);
    }
}
```

Classes as types

The three classes 'taxi', 'hiredCar' and 'payments' are all user defined. The class 'payments' defines 3 objects whose type is a class. For now look at a class as a complex data type and look at an object as a complex kind of variable whose type is a class. The following table shows which objects are used in the class 'payments'

class	*object*
Scanner (this class is defined by Java)	scan
Taxi (this class is user-defined)	t
hiredCar (this class is user-defined)	hc

The objects created in the previous program

6.4.4 Expressions and Operators

Expressions are phrases made up of operators and operands. An expression is evaluated to give an overall result. The most common expressions are the arithmetical ones. Look at the arithmetic expression 3.4 – 4.5 * (3 – 1.2). The operators in this expression are minus (used twice) and multiplication. The operands are 3.4, 4.5, 3 and 1.2. This expression evaluates to -4.7. The following table shows some different types of expressions, their operators, operands and evaluation.

Expression	*Type of Expression*	*Operators*	*Operands*	*Evaluation*
3 + 53 mod 4	integer	+, mod	3, 53, 4	6
4.2 + 3.1 * 4.0	float	+, *	4.2, 3.1, 4.0	16.6
c++ (where c='p')	character	++	c	'q'
"wind" + "mill"	string	+ (concatenation not plus)	+	"windmill"

True And False	Boolean	And	True, False	False
9>8	inequality	>	9, 8	True

Expressions

6.4.5 Statements

A statement is an instruction that tells the computer what to do. Some statements are simple like a++ but others are more complex like an if-then-else statement or a loop statement. Statements can get more complex if one is nested inside another.

6.4.6 Precedence of Operators

Consider the arithmetic expression 2 + 3 * 4. This works out to 14. The multiplication must be worked out before the addition. If we switch the order of the operators and work out the addition first we arrive at the erroneous result of 20. So we say that multiplication has precedence over addition. In our secondary-school years this was called the BODMAS rule. Let us consider another example: 4>5-3. The minus operator must be worked out first. So we say that "-" has a precedence over ">".

6.4.7 Associativity of Operators

Suppose we have the expression A AND B AND C. Which AND should be calculated first, the one on the left or the one on the right? Associativity tells us whether the one on the left should be worked out first (left-to-right) or the one on the right (right-to-left).

The following table gives the precedence and associativity rules of the Java operators.

Operator	*Description*	*Associativity*
()	Parentheses (grouping)	left-to-right
++ --	Unary postincrement/postdecrement	right-to-left
++ -- + - ! ~ (*type*)	Unary preincrement/predecrement Unary plus/minus Unary logical negation/bitwise complement Unary cast (change *type*)	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >= instanceof	Relational less than/less than or equal to Relational greater than/greater than or equal to Type comparison	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right

&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
?:	Ternary conditional	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left

All the Java operators ordered according to priority with their corresponding associativity rules

6.4.7.1 Exercise

Evaluate the following expressions:

(a) 4 + 9 / 3

(b) 5 - 12 MOD 5 + 17 DIV 3

(c) 3.4 + 12.4 / 4 * 2.2

(d) “Keep” + “ it ” + “simple.”

(e) kar++ (where kar = ‘B’)

(f) “after” > “before”

(g) (3 - 4 < 12) OR (15=22)

(h) (‘i’ = ‘I’) OR (12 < 27) AND (9 <= 14)

6.5 Standard Algorithms

The algorithms we will examine perform Searching and Sorting.

6.5.1 Searching Algorithms

Only one algorithm will be studied. This is called the Linear Search.

6.5.1.1 Linear Search

A linear search (sequential search) is an algorithm that looks for a particular value in a sequence of values. The search starts from the beginning of the sequence and elements are checked one by one until the required value is found or until the end of the sequence is reached without finding the element.

The linear search is usually used on an unsorted sequence of elements. It is a slow method and is only practical to use when the number of elements is not large. Consider the following example where an array Seq contains integers.

	0	1	2	3	4	5	6	7	8	9
Arr	26	31	9	12	10	-25	3	32	-9	15

The Integer-Array Arr

To look for the number 12 the algorithm first checks Arr[0], then Arr[1] etc. until it arrives at Arr[3] and finds the number 12. The following algorithm in pseudocode describes the Linear Search.

```
Given an array A
Indices of A start with 0 and end at N-1
To find the value k

begin
      found = false;
      index = 0;
      while (found = false) and (index <= N-1) do
            begin
                  if A[index] = k
                        then found = true
                        else increment index by 1
            end
      if found = true
            then return index
            else return -1
end
```

Pseudocode of the linear search

6.5.1.1.1 Exercise

Draw a flowchart of the Linear Search.

6.5.2 Sorting Algorithms

There are many sorting algorithms, some are slow in execution and some are fast. Here we will consider only two.

- Insertion Sort
- Bubble Sort

6.5.2.1 Insertion Sort

The Insertion Sort picks elements one by one and each time it sorts the picked elements until all the elements are sorted. Here is an example.

The following table shows the steps for sorting the sequence of numbers 5 7 0 3 4 2 6 1. On the left side (the sorted part of the sequence) the values are underlined.

5	7	0	3	4	2	6	1
5	7	0	3	4	2	6	1
0	5	7	3	4	2	6	1

0	3	5	7	4	2	6	1
0	3	4	5	7	2	6	1
0	2	3	4	5	7	6	1
0	2	3	4	5	6	7	1
0	1	2	3	4	5	6	7

An example of the insertion sort

6.5.2.1.1 Pseudocode of Insertion Sort

The following is a non-detailed algorithm of the Insertion Sort. Assume a sort that organises elements in ascending order.

```
begin
   1)   create an empty sequence called LEFT
   2)   call the sequence of elements to be sorted RIGHT
   3)   remove the first element (leftmost) from RIGHT (call it EL1)
   4)   place EL1 in LEFT
   5)   LEFT contains one element so it is sorted
   6)   remove the first element (leftmost) from RIGHT (call it EL2)
   7)   place EL2 in LEFT before or after EL1 in such a way that the two
        elements are sorted
   8)   remove the first element (leftmost) from RIGHT (call it EL3)
   9)   without removing the order of elements EL1 and EL2, place EL3 in
        the first, second or third position so that EL1, EL2 and EL3 are
        sorted
   10)  continue in this fashion until RIGHT is empty
   11)  at this point all the elements are in LEFT and are sorted
end
```

Non-Detailed Pseudocode of Insertion Sort

The above algorithm is very similar to a description in natural language.

A much more detailed pseudocode of the insertion sort is shown hereunder.

```
Assumptions
       (i)    The elements to be sorted are found in array A,
       (ii)   (ii) The indices of A are natural numbers starting from 0 and
              ending in N-1,
       (iii)  (iii) Sorting is done in ascending order

begin
      for i=1 to N-1 do
            begin
                  value = A[i]
                  j = i-1
```

```
                    while (A[j] > value) and (j > =0) do
                          begin
                                A[j+1] = A[j]
                                j = j-1
                          end
                    A[j+1] = value
              end
end
```

Detailed Pseudocode of Insertion Sort

6.5.2.1.2 Exercise

a) Draw the flowchart of the insertion sort.

b) Sort the following sequence of numbers using the insertion sort: 25, 10, 7, 36, 18, 12

6.5.2.2 Bubble Sort

The Bubble Sort is a simple sorting algorithm. It works by repeatedly going through the list to be sorted and compares each pair of adjacent elements. If the two values are in the wrong order then the elements are swapped. This means that if the algorithm is performing a sort in ascending order and A and B are compared then they are swapped if A>B.

After one “pass” (a “pass” is performed when each pair of adjacent elements is compared) other passes are repeated until the list of elements is sorted.

The algorithm gets its name from the way smaller elements "bubble" to the top of the list.

As an example let us take the array of numbers "5 1 4 2 8", and sort it in ascending order. In each step the underscored elements are the ones being compared.

First Pass:

(5 1 4 2 8) to (1 5 4 2 8). Here the algorithm compares the first two elements, and swaps them.

(1 5 4 2 8) to (1 4 5 2 8). Swap since 5 > 4.

(1 4 5 2 8) to (1 4 2 5 8). Swap since 5 > 2.

(1 4 2 5 8) to (1 4 2 5 8). Now, since these elements are already in order (8 > 5) the algorithm does not swap them.

Second Pass:

(1 4 2 5 8) to (1 4 2 5 8). No swap.

(1 4 2 5 8) to (1 2 4 5 8). Swap since 4 > 2.

(1 2 4 5 8) to (1 2 4 5 8). No swap.

(1 2 4 5 8) to (1 2 4 5 8). No swap

Now, the array is already sorted, but our algorithm does not know this yet. The algorithm needs one whole pass without any swap to know the array is sorted.

Third Pass:

(1 2 4 5 8) to (1 2 4 5 8). No swap.
(1 2 4 5 8) to (1 2 4 5 8). No swap.
(1 2 4 5 8) to (1 2 4 5 8). No swap.
(1 2 4 5 8) to (1 2 4 5 8). No swap.

Finally, the array is sorted, and the algorithm can terminate.

6.5.2.2.1 Pseudocode of Bubble Sort

The following pseudocode describes the Bubble Sort in a non-detailed way.

```
Assume the elements are stored in an array A.
Assume the indices of the array elements start from 0 till N-1.

begin
      sorted = “not yet”
      while sorted = “not yet” do
            begin
                  compare all adjacent elements A[i] and A[i+1] starting from i=0
                  till i=N-2.  If A[i] > A[i+1] then swap the elements. Also in this
                  case the variable ‘sorted’ is assigned the value “not yet”.  If
                  there are no swaps ‘sorted’ is assigned the value “yes”
            end
end
```

Non-Detailed Algorithm of Bubble Sort

A detailed pseudocode is shown hereunder.

```
Assume the elements are stored in an array A.
Assume the indices of the array elements start from 0 till N-1.

begin
   do
      swapped = false
      for i=0 to N-2 do
         begin
            if A[ i ] > A[ i + 1 ]
               then
                  begin
                     swap( A[ i ], A[ i + 1 ] )
                     swapped := true
                  end
         end
      N = N - 1
   while swapped
end
```

Detailed Algorithm of Bubble Sort

6.5.2.2.2 Exercise

(a) Draw the flowchart of the bubble sort.

(b) Perform the bubble sort on the following sequence of numbers: 25, 48, 23, 60, 33

(c) In the detailed pseudocode of the bubble sort we find the statement N = N – 1. Why is this statement present?

6.6 Programming Paradigms

A programming paradigm is a set of principles that describe the structure of a language. There are various programming paradigms and for each paradigm there are various languages that follow the model (paradigm). For example the languages BASIC, C, FORTRAN, COBOL and Pascal all follow the Imperative Paradigm. Java, C++ and Smalltalk are three languages built on the Object-Oriented Paradigm.

6.6.1 Natural Languages and Formal Languages

Natural languages are the ones spoken between people like Maltese, English and Italian while formal languages are the ones constructed by academics for a specific purpose. Programming languages are all formal languages.

6.6.1.1 Syntax and Semantics

'Syntax' refers to the way in which words may be combined to form phrases and sentences. We may think of syntax as a set of rules telling us how to form sentences.

'Semantics' is the study of the meaning of words, expressions, sentences etc.

Natural language is riddled with ambiguity. This is where a single word, phrase of sentence has multiple meanings independent of context. For example "Bob went to the bank" can mean (i) Bob visited a financial institution or (ii) Bob visited the side of a river.

6.6.1.2 Context-Free and Non-Context Free Grammars

A context-free grammar is one where the syntax of each constituent is independent of the symbols occurring before and after it in a sentence.

Non-context-free grammars cannot identify the meaning of a term without considering the terms before and/or after the term. Natural languages have non-context-free grammars. For example consider the following two sentences.

- He was hit in his shoulder.
- He was hit in Rome.

The first meaning of 'hit' refers to a part of the body while the second refers to a place where an accident happened.

Most of the syntactic structure of modern programming languages is context-free.

6.6.2 Imperative (Procedural) Paradigm

Imperative languages are also called 'procedural'. In these languages the programmer writes a sequence of statements (algorithm) and these express to the computer how a problem should be solved. Examples of procedural languages are BASIC, C/C++, FORTRAN, Java, COBOL and Pascal. Most computer languages are imperative.

The following dBASE examples show procedural and non-procedural ways to list a file. Procedural and non-procedural languages are also considered third and fourth-generation languages.

Procedural (3GL)	Non-Procedural (4GL)
USE FILEX DO WHILE .NOT. EOF ? NAME, AMOUNTDUE SKIP ENDDO	USE FILEX LIST NAME, AMOUNTDUE

Imperative and non-procedural paradigms

6.6.3 Declarative Paradigm

Declarative programming describes what the program should accomplish, rather than describing how to go about accomplishing it. This is in contrast with imperative programming, which requires an explicit algorithm that describes all the steps that a program should follow to solve a particular problem.

Declarative programming is often defined as any style of programming that is not imperative. Declarative paradigm is an umbrella term that includes a number of programming paradigms.

6.6.4 Object-Oriented Paradigm

This type of programming supports a model wherein the 'Data' and their associated 'Processing' (called 'Methods') are defined as self-contained entities called 'objects'. Object-oriented programming (OOP) languages, such as C++ and Java, provide a formal set of rules for creating and managing objects.

6.6.4.1 Objects

Objects are key to understanding object-oriented technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle.

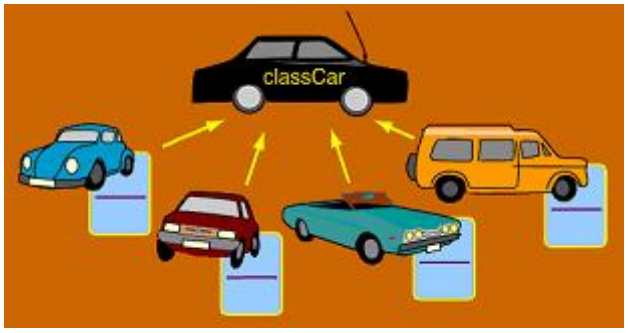
Real-world objects share two characteristics: They all have state and behaviour. Dogs have state (name, colour, breed, hungry) and behaviour (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence,

current speed) and behaviour (changing gear, changing pedal cadence, applying brakes). Identifying the state and behaviour for real-world objects is a great way to begin thinking in terms of object-oriented programming.

Software objects are conceptually similar to real-world objects: they too consist of state and related behaviour. An object stores its state in 'fields' ('variables' in some programming languages) and exposes its behaviour through 'methods' ('functions' in some programming languages). Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication. An 'object' is also called an 'instance of a class'. In the program shown in the next section bike1 and bike2 are two objects. They are both instances of the class Bicycle.

6.6.4.2 Classes

In the real world, you'll often find many individual objects all of the same kind. There may be thousands of other bicycles in existence, all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that your bicycle is an Instance of the Class of Objects known as Bicycles. A class is the blueprint from which individual objects are created.



Class and Objects

The following Bicycle class is one possible implementation of a bicycle:

```
class Bicycle {

      int cadence = 0;
      int speed = 0;
      int gear = 1;

      void changeCadence(int newValue) {
          cadence = newValue;
      }

      void changeGear(int newValue) {
          gear = newValue;
      }

      void speedUp(int increment) {
          speed = speed + increment;
      }
```

```
    void applyBrakes(int decrement) {
        speed = speed - decrement;
    }

    void printStates() {
        System.out.println("cadence:"+cadence+"
                speed:"+speed+" gear:"+gear);
    }
}
```

Class Bicycle

The fields (variables) 'cadence', 'speed', and 'gear' represent the object's state, and the methods (changeCadence, changeGear, speedUp etc.) define its interaction with the outside world. (Cadence is the speed at which you turn the cranks, measured in revolutions per minute (rpm)).

You may have noticed that the Bicycle class does not contain a main method. That's because it's not a complete application (i.e. program); it's just the blueprint for bicycles that might be used in an application. The responsibility of creating and using new Bicycle objects belongs to some other class in your application.

Here's a BicycleDemo class that creates two separate Bicycle objects and invokes their methods:

```
class BicycleDemo {
    public static void main(String[] args) {

        // Create two different Bicycle objects
        Bicycle bike1 = new Bicycle();
        Bicycle bike2 = new Bicycle();

        // Invoke methods on those objects
        bike1.changeCadence(50);
        bike1.speedUp(10);
        bike1.changeGear(2);
        bike1.printStates();

        bike2.changeCadence(50);
        bike2.speedUp(10);
        bike2.changeGear(2);
        bike2.changeCadence(40);
        bike2.speedUp(10);
        bike2.changeGear(3);
        bike2.printStates();
    }
}
```

Class BicycleDemo

The output of this program (i.e. the class BicycleDemo, that makes use of the class Bicycle) prints the final values of pedal cadence, speed, and gear for the two bicycles:

```
cadence:50 speed:10 gear:2
cadence:40 speed:20 gear:3
```

6.6.4.3 Methods

In other languages methods are called 'functions', 'subroutines' or 'procedures'. A method may take 0 or more parameters and may or may not produce a result. Java methods never return more than one result. All methods in Java are part of some class. The following is an example of a method.

```
boolean evenOrNot (int x)
{
        if ( (x%2) == 0) return true;
                else return false;
}
```

A Method

A value passed to a method is called an 'argument'. The variable that receives the argument is called a 'parameter'. So x is a parameter. Now if evenOrNot is prompted by an object called nu as in nu.evenOrNot(9) then the number 9 is an argument. However in many cases the terms 'parameter' and 'argument' are considered as synonymous.

6.6.4.4 Encapsulation

Encapsulation is one of the three major features in object-oriented programming. The other two are Inheritance and Polymorphism. Encapsulation refers to the creation of self-contained modules that bind processing functions to the data. These user-defined data types are called 'classes', and one instance of a class is an 'object'. For example, in a payroll system, a class could be Manager, and Pat and Jan could be two instances (two objects) of the Manager class. Encapsulation ensures good code modularity. The following diagram depicts a class called 'Shape'.

Data	Variable 'colour' of type 'string'
Methods	moveDown(...) moveUp(...) moveLeft(...) moveRight(...) makeInvisible() makeVisible()

Class 'Shape'

From the class 'Shape' one can create as many objects as one requires in one's program. Let us say that three objects are created (i.e. three instances of the

class ‘Shape’). Let these objects be called ‘Shp1’, ‘Shp2’ and ‘Shp3’. These are depicted in the following diagram.

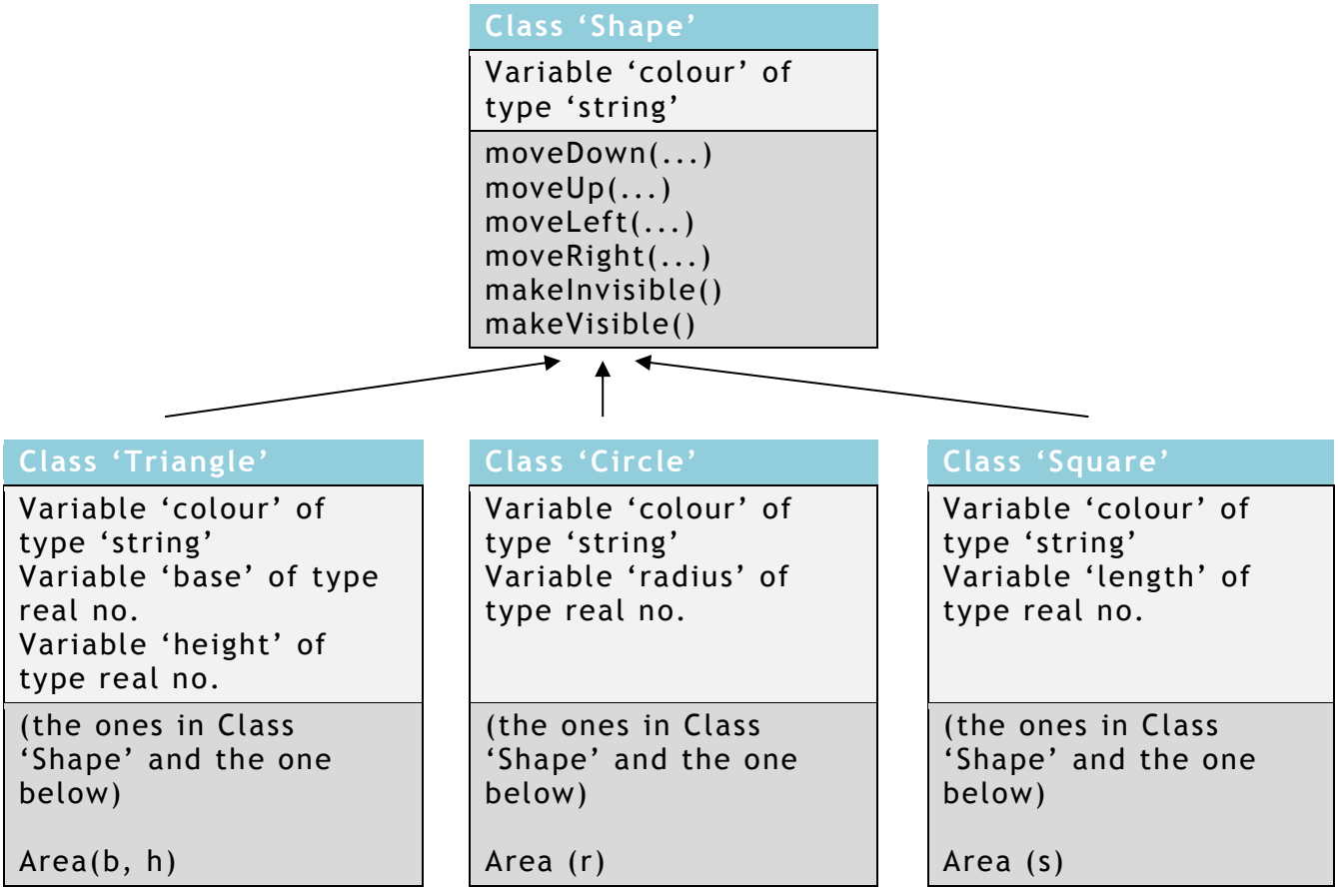
Object ‘Shp1’	Object ‘Shp2’	Object ‘Shp3’
DATA colour = “green”	DATA colour = “yellow”	DATA colour = “violet”
METHODS (as in class)	METHODS (as in class)	METHODS (as in class)

Objects from class ‘Shape’

6.6.4.5 Inheritance

Classes are created in hierarchies, and inheritance allows the structure and methods in one class to be passed down the hierarchy. That means less programming is required when adding functions to complex systems. If a step is added at the bottom of a hierarchy, then only the processing and data associated with that unique step needs to be added. Everything else about that step is inherited.

Example: Let us consider the class ‘Shape’ and from it we create three other classes called ‘Triangle’, ‘Circle’ and ‘Square’. The following diagram depicts this procedure.



Inheritance

The class ‘Shape’ in this case is called a ‘superclass’ while the classes ‘Triangle’, ‘Circle’ and ‘Square’ are all ‘subclasses’ of the class ‘Shape’.

6.6.4.6 Information Hiding

Encapsulation causes information hiding. When creating a class for use by other programmers the way it is implemented is hidden to these programmers. The creator of the class can change its implementation without causing any disturbance to the programmers that use the class.

6.6.4.7 Polymorphism

Object-oriented programming allows procedures about objects to be created whose exact type is not known until runtime. For example, a screen cursor may change its shape from an arrow to a line depending on the program mode. The routine to move the cursor on screen in response to mouse movement would be written for "cursor," and polymorphism allows that cursor to take on whatever shape is required at runtime. It also allows new shapes to be easily integrated.

In the above example one can apply the method Area to any object that is an instance of one of the classes Triangle, Circle and Square.

6.6.4.8 Abstraction

Abstraction is a term that refers to a simplified representation that contains only those features that are relevant for a particular task.

6.6.4.9 Benefits of Object-Oriented Programming

The concepts and rules used in object-oriented programming provide these important benefits:

- The concept of a data class makes it possible to define subclasses of data objects that share some or all of the main class characteristics. Inheritance forces a more thorough data analysis, reduces development time, and ensures more accurate coding.
- The definition of a class is re-useable not only by the program for which it is initially created but also by other object-oriented programs.
- The concept of data classes allows a programmer to create any new data type that is not already defined in the language itself.
- Information-hiding: By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.
- Modularity: The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.
- Code re-use: If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.

- Pluggability and debugging ease: If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace it, not the entire machine.

6.7 Features of Java

In the following part of the notes some important Java features are discussed. However this does not mean that such features are possessed by Java alone.

6.7.1 The Java API

Java API ("Application Programming Interface") is a huge collection of library routines. In the Java API, classes and interfaces are grouped in packages.

All these classes are written in Java and run on the JVM. Java classes are platform independent but the JVM is not platform independent. You will find different downloads for each OS.

6.7.2 Static Classes

The following Java program consists of two classes "StaticDemo" and "StaticDemoProg". In the class "StaticDemo" two variables are defined:

- normalVar is a normal instance variable. This means that every object of the class StaticDemo would have a particular value for its variable normalVal. Say object1 would have its variable normalVal equal to 7 and object2 would have its normalVal equal to 12.
- staticVar has different features. It is like a global variable and if one object changes its value this value is changed for all objects that are instances of the class StaticDemo.
- Note that the static variable staticVar can be called from any object (e.g. object1.staticVar = 25) or from the class itself (e.g. StaticDemo.staticVar = 9).

```
class StaticDemo
{
   int normalVar; //this is a normal instance variable
   static int staticVar; //this is a static variable
}
class StaticDemoProg
{
   public static void main (String args[])
   {
      StaticDemo object1 = new StaticDemo();
      StaticDemo object2 = new StaticDemo();

      object1.normalVar = 12;
      object2.normalVar = 15;

      System.out.println ( object1.normalVar + ", " +
                                                 object2.normalVar );
```

```
        object1.staticVar = 25;

        System.out.println ( object1.staticVar + ", " + object2.staticVar
);

        StaticDemo.staticVar = 9;

        System.out.println ( object1.staticVar + ", " + object2.staticVar
);
    }
}
```

Instance and Static Variables

Therefore the output of the above program is:

12, 15
25, 25
9, 9

Methods can also be "static" methods or "instance" methods. The difference between a static method and a normal (instance) method is that the static method can be called through its class name without any object of that class being created.

The following Java program is made up of two classes. There are no instance methods. To call the method "nicePlace" no object is created. The output, after executing "StatMethProg" is the string "Msida".

```
class StatMeth
{
    static String nicePlace()
    {
        return "Msida";
    }
}

class StatMethProg
{
    public static void main (String args[])
    {
        System.out.println ( StatMeth.nicePlace() );
    }
}
```

A Program with no Instance Methods

Static methods are restricted to call only other static methods and static data. A class that contains a static method or any other static property like a static variable is itself called static. However it can also contain other non-static methods and properties.