DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CSL 101 ; Introduction to Computers and Programming
Semester II 2007-08

Handout : From Caml to Java Author **Sandeep Sen**

1 Moving to an imperative style

Most popular programming languages like Fortran, C, Java follow a different style than the one we have been using so far. Broadly speaking, the programs are written as a sequence of instructions rather than a single function. One or more of these instructions can be a function call. Let us first look at the way functions are written in Java[1]

The following code that implements the power function in Java

```
1 //computing x to power n recursively
2
3  class recpower {
4
5    public static void main ( String argv [] )
6
7      throws java.io.IOException
8
9      {
10          int x , k ;
11
12   System.out.println( "Type in number and exponent ");
13       x = Keyboard.readInt() ;
14       k = Keyboard.readInt() ;
15
16    int ans = power(x ,k) ;
17      System.out.println( ans) ;
18
19      }
20
21  static int power(int x, int k) //The actual function
22
23    {
24
25        if ( k==0 ) return 1 ;
26
27        else {
28
29                int t = power( x, k/2 ) ;
30                     if (( k% 2 ) == 0)
```

[1]Java has become popular because of the way it supports object oriented programming and graphics user interface (gui) and a platform independent support which will not be our focus in the beginning.

```
31                              return t*t ;

32                      else return x*t*t ;
33              }
34          }
35      }
```

If you examine lines 21 – 35, it looks nearly identical to an OCAML function barring the obvious syntax difference (i.e. there is no *then* word in the conditional statement). The previous part of the code may appear very different but we will not elaborate too much except for pointing out that lines 13,14, 17 deal with reading and writing the input from the keyboard and terminal respectively.

To run a java program you have to first *compile* using the command $javac < filename >$ (the filename has a .java suffix) and then subsequently run the compiled program (which has a .class suffix) with $java < filename >$. The file storing the java program must have a name that is the same as the name of the **class** in line 1 with a suffix .java , i.e. this program must be stored in a file named recpower.java.

Here are some hints about the syntax of a java function - in line 21 the header of the function starts with a keyword **static** followed by the type of the value returned by the function and then the types of the inputs within the parenthesis.

To use the mathematical functions you have to *import* a library (set of predefined functions) called *java.lang.Math* and invoke these functions using Math.$< function >$. The function Keyboard.readInt in line 13 reads inputs of type integer from the Keyboard[2]. The following variation of the power function is a generalization and throws light on many aspects of Java.

```
//computing x to power n recursively where x can be float (double in Java)

  import java.lang.Math ;

 class recpower {

   public static void main ( String argv [] )

     throws java.io.IOException

     {
          double x ; int k ;

  System.out.println( "Type in number (double)  and exponent (integer) ");
      x = Keyboard.readDouble() ;
      k = Keyboard.readInt() ;

   double ans = power(x ,k) ;
```

[2]This is not a predefined type and you can copy it from the course page

```
 System.out.println("The value of " + x + " to the power " + k + " is "+ ans) ;

    }

static double square( double x) //squares input

 { return Math.pow( x , 2) ;} //predefined math function

static double power(double x, int k)

  {

      if ( k==0 ) return 1.0 ;

      else {

             if (( k% 2 ) == 0) {return square( power( x, k/2 )); }
                   else { return x*square( power( x, k/2 )) ;}
             }
        }
   }
```

2 Scope

This section can be skipped initially and the reader can come back to it after he/she attains more familiarity with Java programming.

```
//illustrating scope rules

  import java.lang.Math ;

class scopetest {

   final static double pi = 3.14 ; //like a constant
          static double x = 10.0 ;
   public static void main ( String argv [] )

     throws java.io.IOException

     {
          double x ;
          int k ;

    System.out.println(argv[0]);
```

```
   System.out.println(argv[1]) ;
     System.out.println(pi) ;
     //x = 5.0 ;
       System.out.println(x) ;
     testlocal();
     { double y = 1.0;  System.out.println(y); double x = 2.0;}
       //y = x;

     //pi = 3.0 ;

   }

     public static void testlocal() {
         double x = 15.0 ;
         System.out.println(x) ; }

}
```

2.1 Nesting classes

```
//computing x to power n recursively

  import java.lang.Math ;

 class recpower2 {

   final double pi = 3.14 ;
   public static void main ( String argv [] )

     throws java.io.IOException

     {
          double x ; int k ;

  System.out.println( "Type in number (double)  and exponent (integer) ");
      x = Keyboard.readDouble() ;
      k = Keyboard.readInt() ;

   double ans = power(x ,k) ;
  System.out.println("The value of " + x + " to the power " + k + " is "+ ans) ;

     }
```

```
  static double power(double x, int k)

 {
class nested {

  double value;

   nested(double y)

    { value = y ;}

  double square( ) //squares input

  { double x = value; return x*x ;} //squaring value

            }//nested


       if ( k==0 ) return 1.0 ;

       else { double t = power(x, k/2) ; nested z = new nested(t);
                           t = z.square() ;

              if (( k% 2 ) == 0) {
                      return t; }
                    else { return x*t ;}
              }
         }
    }
```

3 From recursion to iteration

Let us review the fibnew function described in section 4.1. This calls a recursive function ifib with four parameters - (prev, curr, i, n). As the recursion unfolds, more and more calls are generated with different values of the parameter. Namely after k calls there are k sets of parameter values. The way the function is evaluated only the last call is significant since the same value is returned at all the levels. (You can verify that by using the directive #trace). Therefore it is a waste of resources - both in terms of time and space the way the function is evaluated. A desirable way will be to only keep track of the *latest* values of the function parameters and stop evaluation when we have the last call. This is precisely where we can make use of *variables* that stores the last value assigned (written) to it.

3.1 The notion of a variable

A variable is associated with a name and a value (the latest) that is stored in a memory location. A variable has an associated type which determines what values it can be assigned. The standard

predefined types int Java are

- **integer** byte (8 bits) , short (16 bits) , int (32 bits) and long (64 bits)
- **float** float (32 bits) and double (64 bits)
- **boolean**
- **char** 16 bits unicode. There is also a special type **Strings** to handle a sequence of characters and there are many useful library functions supporting it.

One can access (read) the variable without changing its value or assign a new value to it. Initially when the program begins execution, the value of a variable is undefined until you explicitly assign a value (don't assume that it is zero).

You can think about it like a piece of paper where you can write a value to memorize something like your car's license plate number and you can change it every time you buy a car. If x is a variable (name) then a typical assignment instruction has syntax

```
x = 5 ;

x = y*z //x is assigned the product of the values of y and z

x = x + 1 ; // the new value of x is the previous value of x plus 1

x = < expression > in general
```

So we can associate a variable with each of the four parameters and keep updating them as necessary. We have to be careful that the updates are done properly. If *prev , curr , i , n* are the variables then

```
curr = curr + prev ;// new curr is old curr + old prev
i = i +1 ;
```

Note that n doesn't change - however we still have to update prev to the value of old curr. But we have lost the value of old curr ! If $prev_1$ and $curr_i$ refer to the values during the i-th call then what we need are $curr_{i+1} = curr_i + prev_i$ and $prev_{i+1} = curr_i$. However we have only two variables for all these versions - see if you can achieve the desired effect. A simple solution would be

```
temp = curr ; //temp remembers the previous value of curr
curr = curr + prev ;// new curr is old curr + old prev
prev = temp ;
```

3.2 Writing a loop in Java

With variables we can update the parameters as required - however we still need a mechanism to start and end. You don't want to write a long sequence of **if ..then .. else** statements. The following control structure in Java (there are equivalent in other programming languages) that does the job.

```
 1 //computing nth Fibonacci number recursively
 2
 3  class ifib {
 4
 5    public static void main ( String argv [] )
 6
 7      throws java.io.IOException
 8
 9      {
10           int n , i, prev, curr , temp ;
11
12   System.out.println( "Type in the index larger than 2");
13       n = Keyboard.readInt() ;
14
15      prev = 0; curr = 1 ; i = 2 ;
16  while ( i < n) {
17       temp = curr ;
18       curr = curr + prev ;
19       prev = temp;
20       i = i+1 ;
21          }
22   System.out.println( curr);
23
24   }
25  }
```

The control structure of a loop has the following characteristics - (i) initial condition (ii) terminating condition (iii) body of the loop. The body of the loop is executed as long as the terminating condition is not satisfied. Other similar constructs are **for .. do** . Before the loop is encountered you must ensure that all the variables (used inside the loop) have been initialized correctly.

3.3 Ensuring correctness

Since this program has been directly derived from a recursive function, it should be possible to prove that the program computes the n-th Fibonacci number as desired. In the context of loops, the induction proofs are called *loop invariants*. For every variable that undergoes changes in the loop we try to compute a general expression in terms of the i-th execution of the loop. In this particular case, we will prove the following.

Claim Let $prev_j, curr_j$ denote the values of the variables $prev$ and $curr$ when the value of $i = j$. Then $curr_j = Fib_j$ and $prev_j = Fib_{j-1}$ immediately following line 15.

Proof (by induction on i). Base case $i = 2$, $curr_2 = 1 = Fib_2$ and $prev_2 = 0 = Fib_1$.
Suppose the assertion is true for $i < j$, then for $i = j < n$. Then

$$curr_j = curr_{j-1} + prev_{j-1} = Fib_{j-1} + Fib_{j-2} = Fib_j$$

$$prev_j = curr_{j-1} = Fib_{j-1}$$

To complete our correctness proof, we further observe that line 22 is executed when $i = n$ and therefore what is written out is $curr = curr_n = Fib_n$. Note that the loop invariant can be written for any stage of the loop but it is most useful when it is written at the beginning or at the end since, once the control enters the loop all the instructions are executed (even if the condition fails in the middle).

Loop invariants also let us argue about the continuation condition of the loop. For instance if we write the condition
while (i ¡= n)
then what we have at line 22 is $curr = curr_{n+1} = Fib_{n+1}$ instead of Fib_n.

4 More controls

4.1 For

```
for ( i = 2 ; i < n ; i++ ) //i is incremented implicitly

{
            temp = curr ;
            curr = curr + prev ;
            prev = temp;
}
```

Will this give you the right output or you have to change the upper limit of i ?

4.2 switch

The "if ..else " statments may become very complex when there are multiple choices involved. The **switch** staement is an useful alternative in such a situation. Different actions are initiated based on different values of a variable (of integer type). The program below gives a simple illustration of the syntax and the typical use of this control statement.

```
//validity of a date

  import java.lang.Math ;

 class calendar {

    public  enum Months {January , February, March, April} ;
// enumerative types are declared globally

   public static void main ( String [] argv )

     throws java.io.IOException
 {
          int k , m , year ;
          boolean valid ;
```

```
        Months mon;

 System.out.println( "Type in date ");
    k = Keyboard.readInt() ;
 System.out.println( "Type in month ");
    m = Keyboard.readInt() ;
    mon = Months.values()[m-1] ; //returns the month corresp to an integer
                      //starting with 0
  System.out.println( "Type in year in 4 digits");
    year = Keyboard.readInt() ;

      valid = true ;

   switch(mon) {
       case January: if (k > 31){valid = false ;}  ; break ;
       case February : if ((year % 4 != 0) &&  (k > 28))
                         {valid = false ;}
                       if ((year % 100 ==  0) && (year % 400 != 0) &&
                        (k > 28 )) {valid = false;} ;
                       if (k > 29) {valid = false;} ;
                          break;
       case  March : if (k > 31){valid = false  ;} ; break;
        case  April : if (k >30) { valid = false ;} ;
      default : valid = false   ;

          } //end switch
       if (valid == false) {System.out.println ("not valid") ;}
            else {System.out.println ("valid");}
   }


}
```

5 Arrays

Lists allow only sequential access to the elements, namely, we cannot access the $i+1$st element without accessing the i-th element. Sometimes, programs can be made to run faster if we can access any of the elements directly (without having to access the all the previous elements). For instance, if we carry out a binary search, then we must access the *median* element very quickly to gain advantage over sequential searching. This kind of random access is achieved by an array that provides us indices to all the elements. Usually the indices are numbered $0, 1, \ldots$, i.e. an array A has elements $A_0, A_1 \ldots$ where an element A_i can be accessed directly and behaves like a variable. An array has a bounded length depending on our requirements. Like a list an array consists of elements of the **same** type - we are not allowed to mix types.

In Java an array has to be announced in the beginning as

```
<type>[] <name>
\\For example int [] count ; double [] marks ; etc
```

Arrays can be allocated dynamically using

```
count = new int[10] \\there are 10 elements count[0] , count[1] .. count[9]
marks = new double[100]
```

Therefore the size is fixed at the time of allocation. They can be combined as

```
int[] count = new int[10] ;
```

The following JAVA program achieves some very common functions of arrays, like reading and writing as well as sorting (using insertion sort). You need not write these common functions again and again - rather use it like predefined functions. It is a good practice to build a library of commonly used functions.

```
 1 //Array.java defines read, write and sorting on arrays
 2
 3   import java.lang.Math ;
 4
 5    class Array {
 6
 7
 8
 9   static void readarray (double [] A) {
10
11    int i ;
12
13       for (i = 0 ; i <= A.length-1 ; i++)
14
15         { A[i] = Keyboard.readDouble(); }
16
17     }
18
19
20    static void printarray(double [] A) {
21
22      int i ;
23
24       for (i = 0 ; i <= A.length-1 ; i++)
25
26         { System.out.print (A[i]+ " ");}System.out.println(" ");
27
28    }
29
30    static void sortarray( double [] A ) { //insertion sort
```

```
31
32       int i,j, k ; double x ;
33
34     for (i=1 ; i < A.length ; i++) {
35
36          x = A[i] ;
37
38          j = i-1 ;
39
40         while ( j>= 0 && A[j] > x)
41
42           {
43             A[j+1] = A[j] ;
44             j-- ; //j = j-1
45
46           } //end of while
47        A[j+1] = x ;
48                 } //end of for
49
50    }
51  }
```

The sorting program is based on a technique called insertion sort where starting with the first element we increase the sortedness by considering one more element till we have exhausted all elements.

The *loop invariant* for the for loop (lines 34-48) is A[0] ..A[i-1] is sorted at line 34.

The proof of this will be based on the following loop invariants inside the while-loop.

A[j], A[j+1] ...A[i-1] are in locations j+1, j+2 ..i and A[i] = A[i+1] is smaller than A[j]..A[i-1] after line 43.

After line 44, this becomes A[j], A[j+1] ...A[i-1] are in locations j+2, j+3 ..i and A[i] = A[i+1] is smaller than A[j+1]..A[i-1]

Also in both invariants, A[0], A[1] ..A[j] are in locations 0, 1, ..A[j] If we exit the inner loop, x=A[i] is larger than A[j], i.e. A[0], A[1] .. A[j], x= A[i] , A[j+1], ..A[i-1] is sorted and are in locations 0, 1, ..i.

Note that all these assertions must be proved formally (by induction on j starting from i-1).

From this the correctness of the for-loop invariant follows.

We can now write a short program for Binary Search using the functions defined in Array.java, especially the sorting function. This is not a very efficient sorting program but you can plug in a more efficient sorting algorithm if required. For this you do not have to modify the Binary Search program !

```
1 //binsearch.java
  2
  3   import java.lang.Math ;
  4
  5  class binsearch {
  6
  7    public static void main ( String argv [] )
```

```
 8
 9      throws java.io.IOException
10
11      {
12           double  key ; int k ;
13
14   System.out.println( "Type the size of array  ");
15       k = Keyboard.readInt() ;
16
17      double [] A = new double [k] ;
18
19 System.out.println("Type the values of array - location 0,1 .."+ (k-1)) ;
20
21         Array.readarray(A);
22          System.out.println ("Initial array") ;
23        Array.printarray(A) ;
24            System.out.println("Type the value of key (double)");
25
26
27           key =  Keyboard.readDouble();
28
29         Array.sortarray(A) ; //first sorts the array
30
31
32
33          bsearch(A, key, 0, k-1) ; // initial call
34
35        }
36   static void bsearch(double [] A , double k, int low, int high )
37 // the recursive version of binary search
38 {
39
40               int mid ;
41
42 if (high == low ) { if (A[low] == k)  System.out.println("found "+ low);
43                         else System.out.println(" not found ");}
44               else { mid = (low + high)/2 ;
45            if (A[mid] == k)  System.out.println("found "+ mid);
46            else { if (A[mid] > k) {high =mid -1 }
47                                    else low = mid+1;}
48 if (high >= low){bsearch(A,k, low, high);}
49          else {System.out.println(" not found ");} }
50
51                        }
52                 }
```

```
53
54  }
```

Enumerating Permutations

Given n distinct objects, we would like to enumerate all the permutations. Since there are $n!$ permutations we would like to print them out immediately rather than storing them. The following program generates the permutations based on the following recurrence - let $N = [1, 2 \ldots n]$ denote the n distinct objects and $\Pi(N)$ are the permutations.

$$\Pi[N] = \sum_{i=1}^{n} [i]\Pi[N - \{i\}]$$

```
 1 // permute.java
 2
 3
 4  import java.io.* ;
 5  import java.lang.Math;
 6
 7  class permute {
 8
 9   public static void main (String argv [] )
10
11    throws java.io.IOException
12
13        {  int i, k;
14
15
16  System.out.println( "Type the size of array  ");
17       k = Keyboard.readInt() ;
18
19      int [] A = new int [k] ;
20
21     for (i =0 ; i <= k-1;i++ )
22      {  A[i] = i ;}
23
24  System.out.println ("Initial array") ;
25          Array.printarray(A) ;
26
27   System.out.println ("The permutations are");
28
29          recpermute(A , 0 , k );
30
31         }// end main
32
```

```
33   static void recpermute (int [] A , int i, int n) // recursive function
34
35     {    int j, temp ;
36
37            if (i == n) Array.printarray(A);
38             else
39
40                  {
41                    for ( j =i ; j <= (n-1) ; j++ )
42
43                        {
44                          temp = A[j] ; A[j] = A[i] ; A[i] = temp;
45                                     // swap A[i} with A[j]
46
47                           recpermute (A , i+1 , n);
48
49                         temp = A[j] ; A[j] = A[i] ; A[i] = temp;
50                                   //restore original permutation
51
52                          } //end for
53
54
55                   }//end else
56
57
58
59        } // end recpermute
60
61
62    } // end class
```

Multidimensional arrays are natural extensions of one dimensional arrays where (in Java) a two dimensional array is an array of one dimensional arrays. The following are example of two dimensional array declarations.

```
double[][] matrix = new double [20][20] \\creates a 20 by 20 matrix

double[][] upper = new double [20][]; \\each array has different size
                                    \\useful for say lower triang matrix
 for (int i = 0; i < 20; i++)
    A[i] = new float[i+1] ;
```

This is possible (unlike many other languages) since it is an array of arrays and the internal representation is like that. We can access the dimensions of an array A by A.length - say when we pass it as a parameter (so we don't have to pass the dimensions separately as parameter.

The statement A = B where A and B are arrays has the effect that both A and B **refer** to the same array. To create a copy, we must copy the elements explicitly (element by element). Similarly passing A as a parameter to a function creates another reference and whenever we change the array elements inside the function, the actual elements also change.

The following JAVA program multiplies a matrix by a vector. Notice the liberal use of functions - this makes code modular and easy to read.

```
1 //multiply matrix by a vector
2
3   import java.lang.Math ;
4
5  class matvec {
6
7    public static void main ( String argv [] )
8
9      throws java.io.IOException
10
11     {
12           int m,n, i, j ;
13
14   System.out.println( "Type number of rows of matrix ");
15       m = Keyboard.readInt() ;
16    System.out.println( "Type number of columns of matrix ");
17       n = Keyboard.readInt() ;
18
19   int [][] matrix = new int [m][n] ; // declaring an integer matrix
20
21       readmatrix(matrix) ;
22         printmatrix(matrix) ;
23
24          System.out.println( "Type vector ");
25
26          int [] vector = new int[n] ; //read vector
27
28        for ( j = 0 ; j <= n-1 ; j++ )
29
30              { vector[j] =  Keyboard.readInt() ;}
31
32          Array.printarray(vector) ;
33
34          Array.printarray(mult(matrix, vector)) ;
35
36     } //end of main
37
38  static void readmatrix ( int [][] matrix )
39
```

```
40     {
41        int i, j ;
42
43   for ( i = 0 ; i <= matrix.length -1 ; i++ ) //read matrix
44    {
45      System.out.println( "Type row "+ i);
46      for ( j = 0 ; j <= matrix[i].length-1 ; j++ )
47        {
48          matrix[i][j] = Keyboard.readInt() ;
49
50        }
51       System.out.println( " " );
52    }
53
54  } //end of readmatrix
55
56    static void printmatrix ( int [][] matrix )
57
58     {
59        int i, j ;
60
61   for ( i = 0 ; i <= matrix.length -1 ; i++ ) //print matrix
62    {
63      for ( j = 0 ; j <= matrix[i].length -1 ; j++ )
64        {
65          System.out.print(matrix[i][j] + "  ") ;
66
67        }
68          System.out.println(" ");
69
70    }
71
72  } // end of print matrix
73
74  static int[] mult ( int[][] matrix , int[] vector )
75
76           // returns matrix vector product
77
78  {
79        int i,j ;
80
81     int[] C = new int[matrix.length] ;
82
83   for ( i = 0 ; i <= matrix.length -1 ; i++ ) //compute product
84    {
```

```
85      C[i] = 0 ;
86      for ( j = 0 ; j <= matrix[i].length -1 ; j++ )
87        {
88          C[i] = C[i] + matrix[i][j]*vector[j];
89
90        }
91          //System.out.println(C[i]);
92
93    }
94     return C ;
95  }  //end of mult
96
97  } // end of class
```

6 Lists, references and objects

Java doesn't have predefined lists like OCAML but it has all the mechanisms that enable us to define simple structures like lists and later we will see more complicated examples.[3]

Like the OCAML lists we can think of list as a structure that may be empty or it has a first element followed by a (recursively defined) list. The following is a JAVA fragment describing list

```
class Intlist {
private int value ; //the first element
private Intlist tail ; //recursively defined remaining part

 public Intlist (int v , Intlist next) {
   // the procedure to create a list - must  have the same name as the class
    value = v ; tail = next ;
  }

  public int getValue () {return value ; }

  public intlist getTail () {return tail ;}

 }  // end of class
```

We can create a list of integers 54, 21, 3 , -8 using the following fragment. Notice the use of **new** instruction that creates a new element of the list.

```
Intlist cell4 = new Intlist (-8, null) ;
Intlist cell3 = new Intlist (3, cell4) ;
Intlist cell2 = new Intlist (21, cell3) ;
Intlist cell1 = new Intlist (54, cell2) ;
```

[3]Incidentally similar mechanisms are provided in OCAML that we didn't pursue

```
//or equivalently

Intlist list = new Intlist (54, new Intlist (21 , new Intlist (3 ,
                    new Intlist (-8 , null)))) ;
```

We describe below an entire JAVA class for the definition and some basic function of integer list.

```
 1 //Intlist.java defines an integer list recursively
 2 //Also builds a list from input from the keyboard
 3
 4   import java.lang.Math ;
 5
 6    class Intlist {
 7
 8      private int value ; //the first element
 9      private Intlist tail ;//recursively defined remaining part
10
11     public Intlist (int v , Intlist next) {
12    // the procedure to create a list - must  have the same name as the
class
13     value = v ; tail = next ;
14         }
15
16
17     public Intlist getTail () {return this.tail ;} /* "this" refers to
          the caller of the procedure - return tail also works */
18
19     public int getValue () {return this.value ; }
20
21
22
23   static Intlist readreverseList () {
24
25 // first entry is the tail and the last the head
26
27    int inputval ;
28    Intlist front = null ;
29
30    System.out.print("Enter number:  ") ;
31    inputval = Keyboard.readInt() ;
32
33    while (!Keyboard.eof()) {
34
35      front = new Intlist(inputval, front) ;
36      System.out.print("Enter number:  ") ;
37      inputval = Keyboard.readInt() ;
```

```
38          }
39
40    System.out.println() ;
41    return front ;
42
43    }
44     }
```

We can add for functions to the class IntList, say counting the number of elements using a recursive procedure identical to OCAML.

```
public int length () {
  if (tail == null) return 1 ;
    else return 1 + tail.length() ; \\ we use the . (dot) to refer to
   //procedure length. In JAVA the tail is referred to as a client.

  } // Does this procedure work on a null list ?
```

We can also define a *class* method (prefixed with the word **static**) that accomplishes the same task.

```
static int countlist(Intlist l) //class method
        {
        if (l == null) {return 0 ;}
           else { return(1 + countlist(l.tail)) ;} //
           }
```

The list can be printed (front-to-back) using the following code

```
while ( ! ( list == null))
             {
              System.out.println(list.getValue()) ;//list.value doesn't
                                 //work because value was declared private
              list = list.getTail() ;
             }
```

Instance and Class variables

Objects are inherently *dynamic* by nature - they are created using a constructor and continue to exist until they can be referenced. The *grabage collector* implicitly destroys unreferenced objects. All variables that are associated with an object are called *instance* variables. If there is a variable that is common to all objects in a class then it is prefixed by **static**.

A method that is prefixed by **static** doesn't use any instance variables -all its variables are static. A variable prefixed by **final** is similar to a constant, i.e., its value cannot be reassigned.

7 Essence of Object Oriented Programming

The way we described and dealt with a new type in this section is the essence of object oriented programming. Not only we defined a new data-type, viz. integer-lists, we also clubbed together some

basic functions (also known as methods) relevant for the new type. For the predefined types like INTEGERS and String, the same methodology is followed.

Programs/Algorithms manage and manipulate data and efficient organisation of data can lead to non-trivial savings in running time and space. It has been observed that individual algorithms have specific patterns of accessing and manipulating data and more often than not they bear siimilarities. These have been recognised by algorithm designers who have classified them under some well known categories of *data structure*. A study and understanding of fundamental data structures can be very useful that saves the programmers from *ad hoc* implementation. Some of the fundamental data structures include *stacks, queues, binary serach trees and heaps*. Very often a data structure is first conceived as a functional entity that supports some specific operations. This is called an *abstract data type*. For example a *dictionary* data type must support searching (and for the dynamic case, insertion and deletion of new objects). There can be various implementation of dictionary as in linked list, sorted arrays, binary search trees and all these data structures support the above operations with varying requirements in time and space. We choose the one most appropriate for our application.

Here we study two elementary data structures - stacks and queues and give Java implementation using circular linked list. A linked list is an ordered set of elements, starting with a *first* element followed by a linked list of the remaining elements. A *circular* linked list has the elements in a cyclic order where the last element is followed by the first element. Although, strictly speaking the cycle is not a total ordering we maintain two specific consecutive elements called *last* and *first*. These are the same elements if the list has only one element and otherwise the list is *empty*. Some common operations supported on this are
(i) insert a new element at the front or back
(ii) delete the last or the first element (if the list is non-empty).
(iii) Initialise a circular linked list.

7.1 Circular Lists

The basic building block is the same as a Intlist - a record with a data field and a reference to another record.

```
class Inode {


    public  int value ;

     public Inode next ;

    public Inode ( int x , Inode y ) {

          value = x ; next = y ;
       }

}
```

We will first describe the circular list and subsequently use it to implement Queues and Stacks.

```
//Circlist.java defines an integer circular list recursively

  import java.lang.Math ;

   class Circlist {

      private Inode last ;//recursively defined remaining part

    public Circlist (int v ) {
   // the procedure to create a list - must  have the same name as the class

    Inode x  = new Inode ( v , null) ; x.next = x ;
         last = x ;

        }



    public int lastnode () {return last.value ; } //returns the last value

    public int first () {

                      // return thefirst element of the circ list
           return last.next.value ;}

    public void Insert_hd (int e) {

    // inserts the element before the head so it becomes new head

     Inode x = new Inode (e , last.next) ; last.next = x ;

        }

     public void Insert_tail(int e) {

 //inserts after last, i.e. the new element becomes the last element

      Inode x = new Inode (e, last.next) ; last.next = x ;

          last = last.next ;
         }

     public void Change_tail (int e){
 // changes the value of tail
```

```
           last.value = e ;
        }

     public void Elim_hd () {

//Remove the head - note that removing tail will be very expensive


       last.next = (last.next).next ;

          }

    }
```

7.2 Stacks and Queues

A queue supports the following operations -
(i) Enqueue (insert) at the back (ii) Dequeue (delete) from the front

```
//Queue.java defines a Stack built on top of Circular queue

  import java.lang.Math ;

   class Queue {

      private Circlist q ;//recursively defined remaining part


    public Queue () {

        q = new Circlist (-1) ;

        }


    public int dequeue () { int a ;
                   if (!(Empty())) { a = q.first(); q.Elim_hd() ; return a;}
                             else

// removes the first element queue is not empty

            { System.out.println("Empty Queue") ; return -1 ;}
```

```
                } //


    public void enqueue (int e) {

    // inserts the element after the tail so it becomes new tail

       q.Change_tail(e);
       q.Insert_tail (-1) ;

        }

     public boolean Empty () {

 //Is the stack empty ?

          return ( q.first() == -1 );
         }

    }
```

A stack supports the following operations
(i) Push (insert) at the top (front) (ii) Delete the top (iii) Report the top element

```
//Stacks.java defines a Stack built on top of Circular queue

  import java.lang.Math ;

   class Stacks {

      private Circlist st ;//recursively defined remaining part


    public Stacks ( ) {

        st = new Circlist (-1) ;

        }


    public void pop () {if (!(Empty())) { st.Elim_hd() ;} else
            { System.out.println("Empty Stack") ;}
                  } //

    public int top () {
```

```
                    // return thefirst element of the circ list
          return st.first() ;}

   public void push (int e) {

   // inserts the element before the head so it becomes new head

      st.Insert_hd(e) ;

       }

    public boolean Empty () {

//Is the stack empty ?

         return ( st.first() == -1 );
        }

   }
```

8 File input and output

Data that is simply read from the keyboard or viewed on the screen is ephemeral, namely, it exists only for the time the program is running. Most useful real-life problems that we deal with data that reside on files, namely that exist whthere or not a program is running. Moreover the same data can be used by more than one program.

Java views files as a stream of characters or a stream of bytes (integers). A character file is viewed as a sequence of characters interspersed with *end-of-line* characters (
n) and ends with *eof* character (ctrl-D). A special Java package called java.io provides convenient interface for reading and writing to a file. The *Filereader* class provides us with method *read* for reading a single character. A *wrapper class*[4] called *Buffered Reader* provides us methods for reading end-of-line using *readLine()*.

Output is supported by *Filewriter* for writing a single character using write(). *BufferedWriter* allows us to write strings and *PrintWriter* lets us write predefined types like integers, doubles, and even boolean. System.out (that prints on the screen) is an object of the class PrintWriter. Filewriter also uses a method close() to close a file (and actually completes the printing).

The corresponding classes for integer-files are *InputstreamReader* that reads a single byte while the wrapper-class is still BufferedReader. The folowing two programs illustrates using files for I/Os.

```
// copies a program from one file to another adding line numbers

 import java.io.* ;
```

[4] A superclass that enhances the basic class

```
class filecat {

 public static void main (String[] args)
     throws IOException
  {

  FileReader fr = null;
  PrintWriter pr = new PrintWriter(new BufferedWriter(new FileWriter
                          (args[1]))) ;
 try {
   fr = new FileReader(args[0]) ;
   }

 catch (FileNotFoundException e) {
  System.out.println("File not Found : " + args[0]) ;
    System.exit(-1) ;

   }

BufferedReader br = new BufferedReader(fr) ;

 String s = "" ;
 int line = 1;

 while ( s != null) {

try { s = br.readLine() ; }

catch (IOException ioe) {

 System.out.println("Error Reading File") ;
  System.exit(-1) ;
 }

   if (s != null) { pr.println(line + "\t" + s) ; }
  line++ ;
}

        pr.close(); //for writing closing is necessary

 } //main

 } //class
```

The following program counts the number of words, lines and characters in a ASCII file. A word is

defined to a maximal sequence of contiguous non-blank characters.

```
//counting in files - words, lines, characters

import java.io.*;

class filecount {
  public static void main(String[] args)
       throws java.io.IOException, java.io.FileNotFoundException
  {
    int count = 0,lc = 0, wc = 0;
    boolean inw = false;
    int cc;
    InputStream is;
    String filename;

    if (args.length >= 1) {
      is = new FileInputStream(args[0]);
      filename = args[0];
    } else {
      is = System.in;
      filename = "Input";
    }

    while ((cc = is.read()) != -1) {
      count++;
      if (cc == '\n')
lc++;
      if (cc == '\n' || cc == '\r' || cc == ' ' || cc == '\t') {
inw = false;
      } else {
if (! inw)
  wc++;
inw = true;
      }
    }
    System.out.println(filename + " has " + count + " chars, " + wc + " words and " + lc + " lines
  }
}
```

8.1 Reading input from Keyboard

Consider the following program for reading input from the keyboard. JAVA views input and output as *streams* of bytes. The `InputStreamReader` class reads one character at a time from the input. Notice that in line 12 we create an instance of this class. Since we want to read input from the keyboard (i.e., the standard input), the argument to the constructor is `System.in`. If we want to read from a file, we would give the filename as the argument. Often we use a wrapper around the `InputStreamReader` class. This is done because often we would like to do things more complicated than reading a single character at a time. For example, we might want to read an entire line, or a large set of characters. Of course this can be done by calling the `read` method in the `InputStreamReader` class iteratively. But this method tends to be very slow because reading from a file takes significant amount of time.

To make the process more efficient, we use the wrapper class `BufferedReader`. The constructor of this class takes an instance of the `InputStreamReader` class – this is done in line 11–12. The `BufferedReader` class defines a buffer of default length, and fills the buffer each time it reads from the input (so, it reads a large chunk of input in one go).

The `BufferedReader` class has the following methods – (i) `readline()` : it reads an entire line of the input, and returns a `String`. It returns `null` if it has reached the end of the file. (ii) `read()` : it reads the next character in the input. It returns -1 if it has reached the end of file.

The `Keyboard.java` class reads input from the Keyboard. The input could be an integer, double, string or character. It also assumes that each line contains exactly one such input. The `iseof` defined on line 5 is a boolean variable which tells whether we have reached the end of the file or not. We illustrate the `readInt()` method, the rest are similar. The method assumes that the next line contains an integer. Line 16 checks whether we have already reached the end of the file, i.e., if the keyboard input is over (by typing ctrl-D). If so it just returns 0. Otherwise it reads the next line in the input (line 20). The next line gets stored in the variable `s`. If `s` is null, we know that we have reached the end of the file, and so we execute lines 27–28. Otherwise we first remove leading and trailing blank spaces from `s` by calling `s.trim()`. We create an instance of the `Integer` class on line 31. This class stores the string `s` (after it has been trimmed) as an integer. The `intValue()` method returns the integer value corresponding to this string.

```
1  import java.io.*;
2
3  class Keyboard {
4
5     static boolean iseof = false ;
6     static char c ;
7     static int i ;
8     static double d ;
9     static String s ;
10
11 static BufferedReader input
12           = new BufferedReader (new InputStreamReader (System.in)) ;
13
14 public static int readInt () {
15
16   if (iseof) return 0;
```

```
17   System.out.flush ();
18
19   try {
20       s= input.readLine () ;
21   }
22
23   catch (IOException e) {
24     System.exit(-1) ;
25   }
26   if (s ==null) {
27       iseof = true ;
28       return 0 ;
29   }
30
31   i = new Integer(s.trim()).intValue();
32   return i ;
33  }
34 public static double readDouble () {
35
36   if (iseof) return 0.0;
37   System.out.flush ();
38
39   try {
40       s= input.readLine () ;
41    }
42
43   catch (IOException e) {
44     System.exit(-1) ;
45    }
46    if (s ==null) {
47       iseof = true ;
48       return 0.0 ;
49    }
50
51    d = new Double(s.trim()).doubleValue();
52    return d ;
53  }
54 public static String readString () {
55
56   if (iseof) return null;
57   System.out.flush ();
58
59   try {
60       s= input.readLine () ;
61      }
```

```
62
63   catch (IOException e) {
64     System.exit(-1) ;
65   }
66   if (s ==null) {
67      iseof = true ;
68      return null ;
69   }
70
71   return s ;
72  }
73public static char readChar () {
74
75  if (iseof) return (char)0;
76  System.out.flush ();
77
78  try {
79       i = input.read () ;
80      }
81
82  catch (IOException e) {
83     System.exit(-1) ;
84   }
85    if (i  == -1) {
86       iseof = true ;
87       return (char) 0 ;
88     }
89
90   return (char) i ;
91  }
92
93 public static boolean eof () {
94
95  return iseof ;
96
97 }
98
99 }
100
```

Exercise 1 *Study the Keyboard.java program carefully and see what happens after a ctrl-D character terminates the current read. Can you read anything following this (from the keyboard) ? Canyou modify it such that two separate Reads can have a ctrl-D in between, i.e. you may want to read two polynomials.*