*Answer **all** the questions.*

1. An *array* is used to represent the status of a board game. In the representation of the board, below left:

- The board is 8 squares by 8 squares
- The game is played only on the white squares, the dark squares cannot be moved to
- The ‘x’ symbol represents player 1’s pieces
- The ‘o’ symbol represents player 2’s pieces

The Board

▨	x	▨	x	▨	x	▨	x
x	▨	x	▨	x	▨	x	▨
▨	x	▨	x	▨	x	▨	x
	▨		▨		▨		▨
▨		▨		▨		▨	
o	▨	o	▨	o	▨	o	▨
▨	o	▨	o	▨	o	▨	o
o	▨	o	▨	o	▨	o	▨

The Array

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
[0]	-1	1	-1	1	-1	1	-1	1
[1]	1	-1	1	-1	1	-1	1	-1
[2]	-1	1	-1	1	-1	1	-1	1
[3]	0	-1	0	-1	0	-1	0	-1
[4]	-1	0	-1	0	-1	0	-1	0
[5]	2	-1	2	-1	2	-1	2	-1
[6]	-1	2	-1	2	-1	2	-1	2
[7]	2	-1	2	-1	2	-1	2	-1

In the array representation of the game:

- –1 represents a square that cannot be moved to by either player
- 0 represents a white square that is currently empty
- 1 represents a square occupied by player 1
- 2 represents a square occupied by player 2

(a) State the number of empty white squares shown in the above diagram. *[1 mark]*

(b) State a suitable data type for an element of the array. *[1 mark]*

(c) Construct the *algorithm* that calculates and returns the number of empty squares on the board. *[4 marks]*

(This question continues on the following page)

(Question 1 continued)

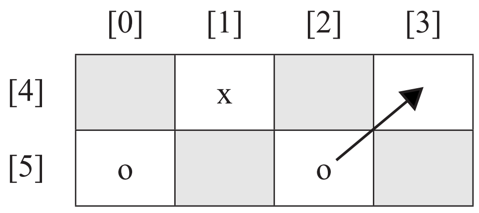
When the program begins the board array needs to be initialized with the values shown in the array diagram on the previous page. For example, all of the elements corresponding to the squares that cannot be moved to should be set to the value –1.

(d) Construct a method that sets to –1 all of the array elements that cannot be moved to. *[4 marks]*

Consider the player playing the o pieces. He may only move a piece following these rules:

1. The piece cannot move to a row or column that is off the edge of the board.
2. The piece must move forward 1 row and to the side 1 column only (i.e. diagonally, 1 square).
3. The piece cannot move to a position occupied by another piece (of either player).
4. The piece cannot move to a square represented by a –1.

In the diagram given below (showing a small part of the board), the arrow represents the only available move that follows the 4 rules given above.



(e) Construct the method `testMove` that accepts 4 parameters as follows:

- the row and column subscripts/indexes of the piece to be moved
- the row and column subscripts/indexes that the player wishes to move to

and returns a numeric code from 1 to 4 if one of the 4 rules, stated above, is broken.

For example:

```
int theMove = testMove(5, 2, 4, 3)
```

and a 0 would be returned to indicate that the move should succeed. *[8 marks]*

(f) Outline a method of storing the status of the board, which uses less memory. *[2 marks]*

Turn over

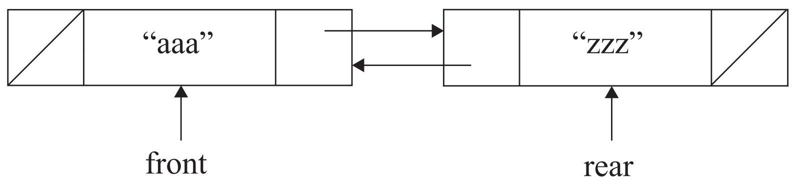
2. The following code represents a *class* that can be used to create a *doubly-linked list*:

```
public class DoubleNode
{
  private String name;
  private DoubleNode next;
  private DoubleNode prev;

  public DoubleNode()
  {
    name = null;
    next = null;
    prev = null;
  }
  public DoubleNode(String nm, DoubleNode nx, DoubleNode pv)
  {
    setName(nm);
    setNext(nx);
    setPrev(pv);
  }
  public void setName(String n){ name = n; }
  public void setNext(DoubleNode n){ next = n; }
  public void setPrev(DoubleNode p){ prev = p; }
  public String getName(){ return name; }
  public DoubleNode getNext(){ return next; }
  public DoubleNode getPrev(){ return prev; }
}
```

(a) Explain the concept of *encapsulation* using the above code as an example. *[2 marks]*

The DoubleNode Class is used to create a list of names with front and rear pointers to special nodes:



(This question continues on the following page)

(Question 2 continued)

The following statements are then executed:

```
DoubleNode d = new DoubleNode();
d.setName("Katsuyoshi");
DoubleNode x = new DoubleNode("Geraldine", d, front);
d.setPrev(x);
front.setNext(x);
rear.setPrev(d);
d.setNext(rear);
x.setPrev(front);
```

(b) Draw and label a diagram of the resulting doubly-linked list. *[5 marks]*

(c) Construct the code that does the following:

- inputs a name
- traverses the list
- if the list has no data nodes outputs an error message
- if the name is not in the list outputs an error message
- otherwise deletes the node containing the name from the list

There are no duplicate names in the list, the "aaa" and "zzz" nodes are always present. *[10 marks]*

(d) State the *BigO* efficiency of the algorithm outlined in part (c). *[1 mark]*

(e) Outline an application that could use a *queue*. *[2 marks]*

1. (a) 8; ***[1 mark]***

(b) int/long; ***[1 mark]***

(c) An example solution is:

```
public int countEmpty()
{
// braces do not have to be used...
  int count = 0;
  for (int r = 0; r < 8; r++)
  {
    for (int c = 0; c < 8; c++)
    {
      if (board[r][c] == 0)
      {
        count = count + 1;
      }
    }
  }
  return count;
}
```

*Award marks as follows up to **[4 marks max]**.*

a nested loop structure of any kind;
loop with correct initial condition;
loop with correct final condition;
test of array element = 0;
return of count or equivalent; ***[4 marks max]***

(d) An example solution is:

```
public void setBoard()
{
  // do even rows
  for (int r = 0; r < 7; r = r + 2)
    for (int c = 0; c < 7; c = c + 2)
      board[r][c] = -1;
  // do odd rows
  for (int r = 1; r < 8; r = r + 2)
    for (int c = 1; c < 8; c = c + 2)
      board[r][c] = -1;
}
```

*Award marks as follows up to **[4 marks max]**.*

recognizing that two passes are necessary or an internal test of odd/even rows is needed (or some tricky mod/div combo is attempted) without being completely correct in implementation;
correctly setting even rows, even columns;
correctly setting odd rows, odd columns;
setting board[r][c] to –1 for 16 distinct elements of the array (wherever!); ***[4 marks max]***

(e) An example solution is:

```
public int testMove(int sr, int sc, int fr, int fc)
{
  // check it is not off the board
  if ( (fr < 0) || (fr > 7) || (fc < 0) || (fc > 7) )
    return 1;
  // check the move is one row, one column
  // (Math.abs could be used here)
  if (((fr - sr) != -1) && ( ((fc - sc) != 1) || ((fc - sc) != -1)))
    return 2;
  // check target square is empty
  if (board[fr][fc] > 0)
    return 3;
  // check it is not a -1 square
  if (board[fr][fc] < 0)
    return 4;
  // ok
    return 0;
}
```

Award marks as follows up to ***[8 marks max]****.*

correct method signature;
each correct test ***[4 max]***
each correct return code ***[4 max]***
return of 0 if move succeeds; ***[8 marks max]***

Note that a (complete) solution like

`if (board[fr][fc] == 0) return 0;` (or `return board[fr][fc];`)

Does not follow the instructions given but may be awarded ***[1 mark]*** according to the above scheme.

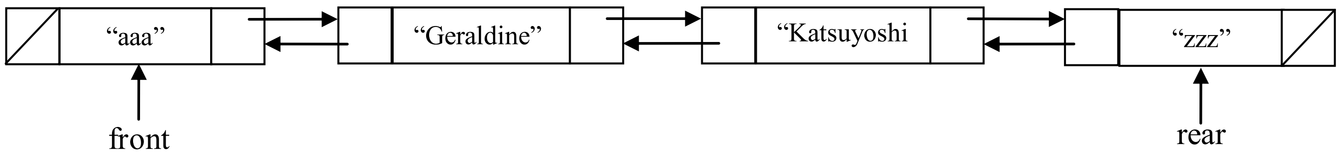
(f) *Award marks as follows up to* ***[2 marks max]****.*

The –1 squares/half the board need not be stored;
Thus the (32 valid) positions could be stored in a linear structure (array/linked list);
Requiring some additional processing to convert to board positions;
Making it more complex to decide whether a move is valid; ***[2 marks max]***

2. (a) *Award marks for the following up to* ***[2 marks max]****.*

The Class contains both data members and methods;
The data members cannot be accessed directly/they are private;
The (external) behaviour of the class is defined by the (public) methods; ***[2 marks max]***

(b) An example diagram.



Award marks for the following up to ***[5 marks max]****.*

both names in the nodes in the correct order;
the next pointer of "Geraldine" pointing to "Katsuyoshi" **and** the previous pointer pointing to "aaa";
the previous pointer of "Katsuyoshi" pointing to "Geraldine" **and** the next pointer pointing to "zzz";
front's next pointer linked to "Geraldine";
rear's previous pointer pointing to "Katsuyoshi"; ***[5 marks]***

(c) An example solution is:

```
public void removeName()
{
  DoubleNode temp = front;
  boolean done = false;
  String name = input("Please input a name: ");
  // Move to first data node, check if list is empty
  temp = temp.getNext();
  if (temp.getName().equals("zzz"))
  {
    output("No names in this list");
  }
  else
  {
   // traverse to end
   do
   {
     if(temp.getName().equals(name))
     {
       // found it; remove is eased because there
       // will always be a previous and next node
       // link from  previous to next then next to previous
       temp.getPrev().setNext(temp.getNext());
       temp.getNext().setPrev(temp.getPrev());
      // tidy up unlinked node references (not needed in answer)
      temp.setNext(null);
      temp.setPrev(null);
      // set exit loop flag
      done = true;
     }
     else
     {
       temp = temp.getNext();
     }
   } while ( !done && !(temp.getName().equals("zzz")) );
  // or !( done || (temp.getName().equals("zzz")) ) - de Morgan
  if (!done)
     output("Name does not exist in list");
}
}
```

(Do not penalise if String comparisons use == rather that equals method)

Award marks as follows up to ***[10 marks max]****.*

All local variables used are declared and initialized;
A name is input;
A good attempt is made to deal with the empty list case;
A correct attempt is made to deal with the empty list case ***[2 marks].***
An error message is output for an empty list;
The list is traversed in some way (move to next);
The traverse is stopped at the required node;
The previous node is linked to the next;
The next node is linked to the previous;
The traverse is terminated if the end of the list is found;
An error message is output when no name is found in the list; ***[10 marks max]***

(d) O (1); ***[1 mark]***

(e) *Award marks as follows up to* ***[2 marks max]***.

storing input characters;
for subsequent/sequential processing;
simulating a physical queue;
e.g. customers in a post office; ***[2 marks max]***
Accept any feasible application example.