Murdoch
UNIVERSITY

Topic 10

Data Structures

ICT167 Principles of
Computer Science

Murdoch UNIVERSITY

OBJECTIVES

- Classify common Data Structures according to whether they are:
  - **linear** or allow **direct access**
  - **homogeneous** or **heterogeneous**
  - **static** or **dynamic**
- Describe **Lists, Queues**, **Stacks**, **Sets**
- List and describe the **methods** which you would expect to find in the following classes: **List**, **Queue**, **Stack**

Murdoch UNIVERSITY

OBJECTIVES

- Describe the **difference between an array and an ArrayList object** in Java
- Understand generics (parameters for types) in Java
- Be able to use **an ArrayList** in a simple program
- Be able to use the **for-each** loop in Java
- Be able to use **a Stack** in a simple program

Murdoch UNIVERSITY

OBJECTIVES

- Be able to use **a Queue** in a simple program
- List some of the concerns of an implementer of a Data Structure class

Reading:

Savitch Chapter 12.1 plus extra material

DATA STRUCTURES

- Data structures are ways of collecting and organizing (a lot of) data into structures
- It is common to use a standard abstract data type (ADT) to manage a structured collection of data
  - Choose the ADT for the purpose
- There are many standard ADTs for data structures

DATA STRUCTURES

- They vary according to:
  - Whether the data is arranged in a **linear** way (eg: an array) or a **non-linear** way (eg: a tree shape)
  - How access to data items is allowed (one at a time, first one first – **sequential access**, or some sort of indexing – **direct access**)
  - Whether the data items have to be of the same type (**homogeneous**) or can be mixed (**heterogeneous**)

DATA STRUCTURES

- They vary according to:
  - Whether the data structure is **static** (of fixed size, known at compile time, eg: an array) or **dynamic** (can grow and shrink while program is running, eg: an ArrayList, a vector or a linked list)

DATA STRUCTURES

- Programmers need to be familiar with many data structures in order to:
  - Choose the right one for the job
  - Find it in a library
  - Use it correctly
  - Implement it themselves

DATA STRUCTURES

- In this unit we just get a basic idea of some common data structures
  - In later units you'll get to know many quite well
- We have already met a homogeneous, linear, direct access data structure of fixed length: the *array*
- Let us look at some others

THE LIST ADT

- There is no precise agreement in the literature about what this is exactly
- General idea – it is a linear structure of varying length
- It is a collection of data stored sequentially
  - For example, a list of students, a list of courses, a list of books, a list of companies, etc can be stored using a list

THE LIST ADT

- Often called linked list – a dynamic data structure commonly used in many programming languages
- Generally a list is homogeneous (i.e. each item in it is of the same type) but, as we will see, this is not very important in Java
- Some definitions allow only sequential access perhaps with the help of a cursor or list pointer

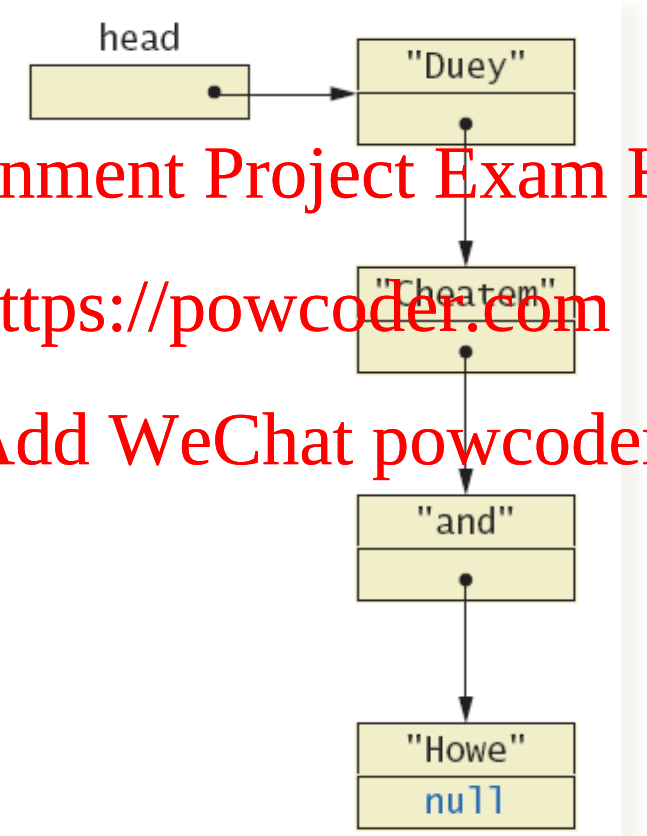
THE LIST ADT

- So you can look at (or remove) the current item only and have to move the cursor forwards or backwards through the list to access other elements
- Other definitions allow direct access; i.e. you can do things with the *i*th element, for any (meaningful) value of *i*
  - Eg: the pre-defined class ArrayList available in java.util package

THE LIST ADT

- A list (or linked list) consists of nodes
- Each node has a place for an element of data and a link (pointer) to another node
- In Java, each node is an object of a class that has two instance variables:
  - One for the data and one for the link
  - A pre-defined LinkedList is available in the java.util package

Figure from textbook



THE LIST ADT

- In giving a definition of an ADT (eg: *list)*, you need to just specify what operations are available on it and say what they do
  - These are the methods which you would expect to find if someone sold you a library with a *list* class in it
  - These are methods which you would have to provide if you wanted to sell your own *list* class
  - These are the only operations which you would be allowed to use if an exam question asked you to accomplish a task using a *list*

LIST OPERATIONS

- Here are some operations which you might expect to be available for a list of objects of type T (plus or minus a few)
- Eg: a list of **int**s, a list of **boolean**s, a list of **String**s, a list of Books

LIST OPERATIONS

- Some constructors plus the following methods:


```
public void makeEmpty()

public int size() // returns size of list

public T elementAt(int index)

- returns the element at position index

public void setElementAt(int index,
                         T newValue)

- changes element at position index to newValue
```

LIST OPERATIONS

```
public void removeElementAt(int index)
```

- removes element at position index and moves all the rest forward


```
public void insertElementAt(int index,
T newValue)
```


- puts newValue in the list at index position and moves the rest along


```
public void addElement(T newValue)
```

- puts newValue at the end of list

THE QUEUE ADT

- The Queue ADT is a homogeneous, linear structure but with restricted access
  - First-In-First-Out (FIFO) access order
- In a queue, insertions take place at the back (the tail) of a queue and deletions take place from the front (the head) of a queue

THE QUEUE ADT

- Operations include constructors (to create a new empty queue with a certain capacity), plus the following:


```
public boolean isEmpty()
public boolean isFull()
public void enqueue(T newValue)
```

  - puts newValue at the end of queue, also called append

THE QUEUE ADT

```
public T dequeue()
```

- returns the front value and removes it from the queue, also called remove

```
public int size()
```

- returns the size of the queue

You may also find:

```
public T peek()
```

- returns top value without removing it from queue

THE STACK ADT

- The Stack ADT is also a homogeneous, linear structure but with a different restricted access
  - Last-In-First-Out (LIFO) access order
- In a stack insertions and deletions take place only at the one end, referred to as the top of a stack

THE STACK ADT

- Operations include constructors (to create a new empty stack with a certain capacity), plus the following:


```
public boolean isEmpty()
public boolean isFull()
public void push(T newValue)
```

- puts newValue at the top

```
public T pop()
```

- returns the top value and removes it from the stack

THE STACK ADT

- Note that instead (or as well) you may find:

```
public void pop()
```

- removes top value from stack

and

```
public T peek()
```

- returns top value without removing it from the stack

THE SET ADT

- The Set ADT is a non-linear data structure
- Only **one** copy of any element is allowed in the set
- Operations include constructors and the following:


```
public void makeEmpty()

public void add(T newValue)
```

- adds newValue if it is not there already
  (does nothing if it is there already)

THE SET ADT

```
public void remove(T value)
- removes the only copy of value if it is there
public boolean isIn(T value)
- membership testing
```


- Other methods include: set union (+) and set intersection (*) which return a new set

- **Note:** check Java API and documentation for any data structure class which you use
- There are many variations on the above mentioned general ideas

DATA STRUCTURES IN JAVA

- Many data structures are homogenous (often for efficiency of memory usage reasons)
- This creates problems for writers of library classes. Do they supply code for:
  - a class of queues of **int**s
  - another class of queues of **double**s
  - another class of queues of **String**s
  - another class of queues of Books, etc etc?

DATA STRUCTURES IN JAVA

- In C++ the idea of parameterized classes is used
- Java also introduced parameterized classes – called ‘generics’ – in Java 5.0
- In Java the simple idea is to allow data structures to contain **Object**s

DATA STRUCTURES IN JAVA

- Each data structure is homogeneous as every one of its elements is an Object
- But anything (almost) can go into any data structure as (almost) everything is a (type of) **Object**

DATA STRUCTURES IN JAVA

- Until Java 5.0, there were two problems:
  - Primitive values (which are not Objects) have to be wrapped to allow them to be stored, and
  - Everything comes out of a data structure as an **Object** and you need to cast it back into its more specific type in order to call specific methods on it

DATA STRUCTURES IN JAVA

- Since Java 5.0 (jdk1.5) and later versions, which allow **automatic boxing** and **unboxing** of primitive types, the above problems are of much less concern now

Murdoch UNIVERSITY

THE CLASS ARRAYLIST

- In Java, arrays (alone) are set up as special built-in data structures, and they are static (fixed size)
- We know that we can have arrays of specific types including primitives
- However, once an array is created its size cannot be changed
  - Although it is possible to create a new larger array to replace the current array and copy its elements – it is awkward

THE CLASS ARRAYLIST

- A more elegant solution is to use an instance of the Java library class `ArrayList`
- `ArrayList` instances can be thought of as arrays that grow and shrink while a program is running
- However, the base type of an `ArrayList` instance must be a class type

THE CLASS ARRAYLIST

- `ArrayList` is not automatically part of Java
- It is available as part of the `java.util` package and must be imported by your program


```
import java.util.ArrayList;
```

- An instance of `ArrayList` is created in the same way as any other object except that its base type is specified using a new notation (called **generics**), as follows:

Murdoch UNIVERSITY

THE CLASS ARRAYLIST

```
ArrayList<String> list = new
                         ArrayList<String>(20);
```


- The above creates and names an `ArrayList` object list which can store instances of class `String` and has initial capacity of 20 elements

- To create an `ArrayList` instance of default capacity (default capacity = 10):

```
ArrayList<String> list = new
                         ArrayList<String>();
```

Murdoch UNIVERSITY

THE CLASS ARRAYLIST

- The `ArrayList` class includes constructors:

`ArrayList<Base_Type>()`

- constructs an empty list with initial capacity 10. The Base_Type must be a class type – i.e. it can not be a primitive type such as int or double

`ArrayList<Base_Type>(int initialCapacity)`

- constructs an empty list with the specified initial capacity. When the list needs to increase its capacity, the capacity doubles

THE CLASS ARRAYLIST

- Methods include:

```
void add(Base_Type obj)
```


- adds obj to the end of this list


```
void add(int index, Base_Type obj)
```


- inserts obj at the specified index position of this list. Shifts elements at subsequent positions to make room for the new entry by increasing their indices by 1

```
Base_Type get(int index)
```

- returns the element at the specified index position

Murdoch UNIVERSITY

THE CLASS ARRAYLIST

```
void set(int index, Base_Type obj)
```

- replaces element at the position specified by index with the given obj in this list

```
Base_Type remove(int index)
```

- removes and returns the element at the specified index

```
boolean remove(Object obj)
```

- removes the first occurrence of obj in this list

```
boolean contains(Object obj)
```

- returns true if obj is in this list, otherwise returns false

THE CLASS ARRAYLIST

```
int indexOf(Object obj)
```

- returns the index of the first occurrence of obj. Returns -1 if obj is not in the list


```
void clear()
```

- removes all of the elements from this list


```
int size()
```

- returns the size (number of elements) of the list

```
void ensureCapacity(int n)
```

- increases the capacity of this list, if necessary, to ensure that it can hold **n** elements

THE CLASS ARRAYLIST

```
void trimToSize()
```

- trims the capacity of this list to be the list's current size

```
boolean isEmpty()
```

- determines whether the list is empty or not

EXAMPLE

```
ArrayList<Integer> list = new
                          ArrayList<Integer>();
System.out.println("The initial size
      of list is:" + list.size());
for (int i=0;i < 15;i++)
   list.add(i*2+1);
   // autoboxing of int to wrapper Integer
System.out.println("\nThe numbers in
                    the list are:");
```

EXAMPLE

```
int temp;
for (int i=0; i < list.size(); i++){
   temp = list.get(i);
   // auto-unboxing of wrapper Integer to its
   // int value
   System.out.println(temp);
}
```


Murdoch UNIVERSITY

FOR-EACH LOOP

- Java (jdk1.5) introduced another form of the `for` loop
- You can use this with a collection of data such as an array or an `ArrayList`
- It is called the `for-each` loop or `enhanced for` loop
- It enables the traversing of a complete array without using an index variable

FOR-EACH LOOP

- For example:

```
int[] myList =
 {10,20,30,40,50,60,70,80,90,100};
for (int index : myList)
   System.out.println(index);
```


- This will display all the values from the array `myList` above

FOR-EACH LOOP

- Similarly, the second `for` loop code from the previous example can be written using the `for-each` loop as follows:


```
for (int i : list){
   int temp = i;
   System.out.println(temp);
}
```

- Or even:

```
for (int i : list){
   System.out.println(i);
}
```

ARRAYLIST VS ARRAY

- Arrays are fixed in size once they are created
  - Once you start putting values/objects in an array, you can not make it larger
- An `ArrayList` instance keeps increasing in size and capacity as you add more elements
- Size = actual number of elements in the list at the moment

ARRAYLIST VS ARRAY

- Capacity = the number of elements the list can currently hold (i.e. amount of memory currently reserved for the list)
  - Capacity can be explicitly increased and/or increases automatically anyway
- The base type of an array is specified when the array is declared
  - All elements of the array must be of the same type (i.e. arrays contain a fixed homogenous type of values including primitives)

ARRAYLIST VS ARRAY

- The base type of an `ArrayList` instance is a class
- Arrays have convenient traditional [square bracket] notation
- `ArrayList` instances are objects with constructors and methods
- Arrays are stored more efficiently

ARRAYLIST VS ARRAY

- `ArrayLists` are also implemented using arrays anyway (but that need not concern the client)
- Elements of an `ArrayList` move left or right during removal or insertion

ARRAYLISTDEMO CLASS

```
import java.util.ArrayList;
import java.util.Scanner;
public class ArrayListDemo {
   public static void main(String[] args){
      ArrayList<String> toDoList = new
                        ArrayList<String>();
      System.out.println("Enter items
         for the list, when prompted.");
      boolean done = false;
      Scanner kbd = new Scanner(System.in);
```

ARRAYLISTDEMO CLASS

```
while (!done) {
   System.out.println("Type entry");
   String entry = kbd.nextLine( );
   toDoList.add(entry);
   System.out.print("More items for
                        the list?");
   String ans = kbd.nextLine( );
   if (!ans.equalsIgnoreCase("yes"))
      done = true;
} // end while
```


Murdoch UNIVERSITY

ARRAYLISTDEMO CLASS

```
        System.out.println("List contains");
        int listSize = toDoList.size( );
        for (int position=0;
             position<listSize;position++)
             System.out.println(
                 toDoList.get(position));
        /* Alternate code for displaying the list:
        System.out.println("The list contains:");
        for (String element : toDoList)
            System.out.println(element); */
    }// end main
}// end class
```

ARRAYLISTDEMO CLASS

- Note that the above program will not compile under jdk 1.4 or earlier versions of Java because these versions did not allow generics

STACKS IN JAVA

- There is a **Stack** class in the **java.util** package
- It is derived from the **Vector** class
- Its methods include:
  - empty()
  - peek()
  - pop()
  - push(E item)

STACKS IN JAVA


```
// File: TestStackADT
/* Program to read in list of names and display the
list in reverse order */
import java.util.*;
class TestStackADT {
   public static void main(String[] args){
      Stack myStack <String> = new
                        Stack <String> ();
      String nameStr;
      boolean done = false;
```

STACKS IN JAVA

```
System.out.println("Please enter the
                          list of names.");
System.out.println("'quit' to finish.");
System.out.print("Name:");
Scanner kbd = new Scanner(System.in);
nameStr = kbd.nextLine();
while(
!nameStr.equalsIgnoreCase("quit")){
   myStack.push(nameStr);
   System.out.print("Name:");
   nameStr = kbd.nextLine();
} // end while
```


Murdoch UNIVERSITY

STACKS IN JAVA

```
        System.out.println("\nThe list in
                           reverse order is:");

        while (!myStack.empty()) {
           nameStr = myStack.pop();
           // typecast back to String required above
           System.out.println(nameStr);
        } // end while

        System.out.println("\nEnd of
                              Program - Bye.");
    }// end of main
}// end of class
```


Murdoch UNIVERSITY

QUEUES AND OTHER ADTs IN JAVA

- Other data structure classes can be found in Java API, Java class libraries purchased from software developers or found free on the Internet
- Also check out software provided with text books for undergraduate data structures courses
- Once you have downloaded such code, compile it and javadoc it and it is ready to use

QUEUES AND OTHER ADTs IN JAVA

- Note that you will have to obtain copyright clearance if you sell software which uses classes downloaded from other sources
- Java API has a `Queue` interface and a `LinkedList` class (in `java.util` package) which can be used to implement a simple queue
- However, it is easy to write your own `Queue` class based on the `ArrayList` class (covered in this topic), as follows:

GENERICQUEUE CLASS

```
/**
 * File: GenericQueue.java
 * Implements a Generic queue class using the
   java.util.ArrayList class
 * Usage: To create a queue of strings, use:
 * GenericQueue<String> myQueue = new
                          GenericQueue<String>();
 * @author P S Dhillon
 */
```


Murdoch UNIVERSITY

GENERICQUEUE CLASS

```
public class GenericQueue<E> {
   private java.util.ArrayList<E> list
      = new java.util.ArrayList<E>();
   public int size() {
      return list.size();
   }
   public E peek() {
      return list.get(0);
      // returns element at the head of
      // the queue without removing it
   }
```

GENERICQUEUE CLASS

```
public boolean isEmpty() {
   return list.isEmpty();
}
public void append (E obj) {
   list.add(obj);
   // adds element to the end of the queue
}
```


Murdoch UNIVERSITY

GENERICQUEUE CLASS

```
    public E remove() {
        E obj = list.get(0);
        // returns element at the head of the
        list.remove(0);
        // queue and removes it
        return obj;
    }
} // end GenericQueue class
```

EXAMPLE USE OF QUEUE CLASS

- **How bad is the queue at the cinema?**
- **Simulation using the Queue ADT**
- Purpose:
  - Simulate waiting in the queue for 30 minutes and produce a report consisting of:
    - Number of paid customers for movie A
    - Number of paid customers for movie B
    - Number of customers turned away because the queue was too long

Murdoch UNIVERSITY

EXAMPLE USE OF QUEUE CLASS

- Given:
  - One ticket office with one queue
  - Average time to serve one customer is 10 seconds
  - Probability of a new customer arriving during the 10 second interval is 0.7 (70%)
  - Maximum length of queue is 10

SIMULATION

- A QUEUE OF BOX OFFICE CUSTOMERS
- Pseudocode for the main loop:


```
Loop 180 times
   Customer Service:
      if the queue is not empty
      serve the person at the front of
      the queue
      increment count of the film chosen
```

SIMULATION

```
Customer Arrival:
   generate random probability for
another arrival
   if the probability is > 70%
      there is no arrival
   else
      if the queue has length 10
         the customer turns away
         increment turnedAway counter
```

SIMULATION

```
        else
            assemble a film for customer
            and place them in the queue
endloop
```

CINEMA QUEUE CLASS

```
/** CinemaQueue.java
 * This program simulates movie cinema queue,
 * where customers attend movie A or movie B
 * Uses the GenericQueue class
 * ------------------------------------- */
public class CinemaQueue {
   public static void main(String[] args){
      GenericQueue<Character> cineQ =
     new GenericQueue<Character>();
      int  timeUnit;
      int  turnedAway = 0;
```

CINEMA QUEUE CLASS

```
int  numA = 0; int numB = 0;
char movieChoice; // 'A' or 'B'
for (timeUnit=1;timeUnit <= 180;
                              timeUnit++){
   if ( !(cineQ.size( ) == 0)) {
      Character c = (Character)
                    (cineQ.remove());
      // removes customer from queue
      // casts Object to wrapper Character
```


Murdoch UNIVERSITY

CINEMA QUEUE CLASS

```
movieChoice = c.charValue();
// unwraps to char
if ( movieChoice == 'A')
   numA++;
else
   numB++;
} // end if
```


Murdoch UNIVERSITY

CINEMA QUEUE CLASS

```
if ( Math.random( ) <= 0.7 ){
   if (mineQ.size( ) >= 10)
      turnedAway++;
   else {
   // select choice & place it in queue
      if (Math.random() <= 0.5)
         movieChoice = 'A';
      else
         movieChoice = 'B';
```


Murdoch UNIVERSITY

CINEMA QUEUE CLASS

```
            cineQ.append( new
                Character(movieChoice));
                // enqueue the wrapped char
            } // end else
        } // end if
    } // end for loop
```


Murdoch UNIVERSITY

CINEMA QUEUE CLASS

```
        System.out.println("Movie A
                customers: " + numA);
        System.out.println("Movie B
                customers: " + numB);
        System.out.println("Number turned
                away: " + turnedAway);
    } // end main()
} // end class
```


Murdoch UNIVERSITY

IMPLEMENTING DATA STRUCTURE ADTs

- There is a lot of work done on implementing these classes
  - Many tricky methods are known to allow data structures to be implemented efficiently and you may study this in future units
- Often the classes have instance variables including an array to actually store all the elements plus perhaps some extra variables to record the state of the structure

IMPLEMENTING DATA STRUCTURE ADTs

- In implementing such classes the programmer needs to consider:
  - Efficiency issues, size of memory used, speed of operations
  - The actual way that the structure will be used (eg: some implementations of lists are better if we often want to remove items from the middle while other implementations are better if we never want to do that)

IMPLEMENTING DATA STRUCTURE ADTs

- In implementing such classes the programmer needs to consider:
  - Multi-threading (you must protect a data structure if there may be several parallel attempts to use it)
  - Making the style of usage and the method names fit in with standard usage

END OF TOPIC 10

Murdoch UNIVERSITY