A static implementation:

- uses an **array of a specific maximum length**, and all storage is allocated **before** run-time.
- orders the items in the array with the index related to the position of the item in the list.

We need:

- a variable to keep track of array elements assigned to the list, i.e. the current number of items in the list, or current size of the list.
- a variable to record the maximum length of the array, and therefore of the list.

Data Structure:

```java
private final int MAX_LIST = 100; //max length of list
private int items[MAX_LIST]; //array of list items
private int numItems; //current size of list
```

We present here again the data structure definition for a static implementation of a list. We have already discussed this slide in the previous lecture. In this lecture we show how to use this data structure to implement the access procedures for a list in the case of an array-based (or static) implementation.

We’ll then show the dynamic implementation of lists and show different data structures for implementing lists.
This is an example implementation of the class ListArrayBased. This class is an example of static implementation of lists. The attributes are the data structure given in the previous slide, and the methods are implementations of the access procedures for lists defined in the interface ListInterface shown in the previous lecture.

An example of an array with maximum size equal to 50 is considered here, where each element of the array is of type Object. The constructor ListArrayBased() is the implementation of the list operation createList() seen before. The other methods here are quite straightforward. Note that they are declared to be public methods as they will be called and used by any external program that needs to use lists.

The implementation of the access procedures “add” and “remove” is given in the next two slides.

The implementation of “get” shown here uses a method call “translate(index)”. Translate is a private method of this class needed to cover (or translate the position number of an item in a list with its respective index value in the array. This is because, as we have already said in the previous lecture, the position of an items in a list starts from 1 whereas the array index values start from 0. For instance the item in the first position of a list is stored in the array with an array index value equal to 0. The private method “Translate” performs this conversion. Note also that this method is only an auxiliary method of our array-based list implementation. It’s not an access procedure of the ADT lists and as such it has to be declared “private”. The example implementation of “translate” is also given in the next slide.

The full implementation of this class ListArrayBased includes also the methods given in the next two slides.
This slide provides an example implementation of the ADT operation “add”.

As mentioned in the previous slide, the private method “translate” is used by the other methods to define the array index value that corresponds to a given position value in the list. Since array indexes start from 0, whereas a position in a list starts from 1, the translation is essentially given by decreasing by 1 the value of a given position.
This slide gives the implementation of the last method of our class ListArrayBased, which implements the ADT access procedure remove. Removing an item at a given position index, is performed by shifting all the items in the list at position > index one place left.

Note that if the item that we want to remove is the last item in the list, i.e. the index parameter is equal to numItems, the for loop is not executed, which means that no shift of items is performed and that the deletion is implemented by just decrementing of one the value of numItems.
A program segment that uses ListArrayBased

```java
Static public void main(String args[]){
    ........
    ListArrayBased aList = new ListArrayBased();
    String dataItem;
    aList.add(1,"Cathryn");
    ........
    dataItem = (String) aList.get(1);
    ........
}
```

References such as `aList.numItems`, `aList.items[4]`, `aList.translate(6)` are ILLEGAL.

**WHY?**

Note that reference within this program such as `aList.numItems`, `aList.items[4]`, `aList.translate(6)` are ILLEGAL, as the attributes of the class ListArrayBased are declared to be private.

In summary, to implement an ADT, given an implementation-independent specification of the ADT operations, we must first choose a data structure to contain the data (in this case we have chosen the array items and the variable numItems). Next we define and implement a class. The ADT operations are public methods within this class, and the ADT data are declared (private) data field of this class. We then implement the class’ methods.

The program that uses the class will be able to access the data only by using the ADT operations.

The main drawback of an array-based implementation is that it restricts the number of items that can be stored in a list. An alternative implementation of the ADT list is the dynamic implementation, which does not have this restriction and in general provides a more efficient memory allocation. We discuss in the remaining part of this lecture the dynamic implementation of lists.
Dynamic Implementation by Linked List

Basic ingredients:

- a “reference variable”
- a node

Linked list: Nodes linked to one another, the beginning of the list referenced by a “reference variable”:

The array implementation given in the previous slides is not always the best data structure to use to maintain a collection of data. An array has a fixed size, but the ADT list can have an arbitrary length. Moreover, the array implementation orders the items of a list using its physical order. So we need to shift data when we want to insert or delete an item at a specific position.

The dynamic implementation overcomes these limitations. The basic ingredients (or data structure) of a dynamic implementation are a “reference variable” (that acts like a pointer) and a “node”. The reference variable has the sole purpose of locating the first node in the list. A node is instead composed by a “data”, which is an item in the list and whose type is the same type of the items in the list, and a “link” to the next node in the list. Such link is itself a reference variable used to locate the next item in the list.

A linked list is then given by nodes that are linked to one another and whose first node is referenced by a reference variable, called head. The last item in the list is itself a node, but whose link has value “null” since there is no other node after it! To distinguish the reference variables used in the nodes to implement the links from the reference variable that refers to the first node in a list, we call the latter a pointer.

The pointer head is different from the other links in the diagram above in that it is not within one of the nodes. Rather, it is a simple reference variable that is external to the node data structure, whereas a link (or field “next” of a node) is an internal reference variables within the node data structure of the list. So, the pointer is simply a reference variable that enables us to access the list’s beginning. Also note that pointer always exists, even when there are no nodes in the linked list. In particular: when the value of the pointer is null, then it means that the pointer does not reference anything and therefore that the list is empty.
The Underlying Data Structure

A linked list is a dynamic list, implemented using reference variables. This means that the storage needed is allocated at run-time, according to the length of the list at any time.

Data structure:

```java
class Node{
    private Object item;
    private Node next;
    ...
}
```

```java
class ListReferenceBased{
    private Node head;
    private int numItems;
    ....
}
```

Two data fields in class Node:
- item, which may contain only values of the type Object
- next, the reference variable for the next node, which is of type Node.

A reference variable for the first node of a list, which is also of type Node, and the number of elements in a list.

The underlying data structure for a linked list includes a class called “Node”, which is used to implement each item of a list and link the items together, and the reference variable called “head”, which is used as reference to the first node in the list. The latter is an attribute of the class list itself, whereas the class Node is the implementation of the underlying data structure used to implement the list.

In this slide, the right box gives the Java type declaration of a linked list, whereas the left box gives the Java type declaration of the underlying data structure needed to implement a linked list.

As we will see later in this lecture, a reference-based implementation of a linked list will essentially be given by the implementation of the class ListReferenceBased, which include information like the number of items in the list and the reference variable “head” for pointing to the first element in the list. The class Node is really a “type”, and provide the underlying data structure (or type of the items in a list) used by the class ListReferenceBased to implement the list. For instance it is used to declare that the reference variable head is a variable of “type” Node, since it has to refer to the first element of a list, which is an instance of the class Node.
The data structure of linked list is more complex than the array used in the static implementation. First of all, its component node requires itself a proper implementation, which is shown in this slide. The implementation of the access procedures of the ADT list I gave in first slide of the previous lecture will use this class Node as underlying data structure.

Given that a node is composed of two pieces of information (an item, and a link), its type declaration could for instance be given by:

```java
public class Node{
    public Object item;
    public Node next;

    public void setItem(Object newItem){
        item = newItem;
    }

    public Object getItem( ){
        return item;
    }

    public void setNext(Node nextNode){
        next = nextNode;
    }

    public Object getNext( ){
        return next;
    }

} // end class Node
```

But this type declaration would violate the ADT principle that data fields of the underlying data structure of an ADT must be declared as private, since it has to be made accessible only by the methods of the ADT itself and not by an external program. The correct type declaration of the node is as given in this slide, where the two attributes “item” and “next” are declared as private.

As a consequence the class Node needs methods for accessing and changing the values of these two attributes. The methods setItem and setNext set the values of the two data fields of a node, whereas the methods getItem and getNext read the values of these two data fields.

Two possible constructors can be used for this class. These are given here:

```java
public Node(Object newItem){
    item = newItem;
    next = null;
}
//end constructor

public Node(Object newItem, Node nextNode) {
    item = newItem;
    next = nextNode;
}
//end constructor
```
Creating and Linking two nodes

Node n1 = new Node();
Node n2 = new Node();
n1.setItem(new Integer(5)); // set item in first node
n2.setItem(new Integer(9)); // set item in second node
n1.setNext(n2); // link the nodes

Additional information

- The “next” field of the last node in a list should be set to `null`.
- When the list is empty, the reference variable head should have value `null`.
- When last reference to a node is removed, the system marks the node for garbage collection.

In this slide we give an example of how to use the class Node defined before. In particular we show how we can create two nodes and link them together. Note that in this example we have used just the default constructor. These examples are important for and used in the implementation of the access methods of the linked list. As we will diagrammatically show in the next two slides, adding an element to a list means in effect first creating a node where the value of the element is stored and then insert this node in the linked list by appropriately setting (or changing) the links of the adjacent nodes.

The first two statements in the left of the slide create two instances of the class Node which are referred to by the reference variables n1 and n2 respectively. The third statement initialises the data file item in the first node, whereas the next statement initialises the data file item in the second node. The last statement, instead links the nodes by making the first node reference the second.

Using the two new constructors given in the previous slide, we could rewrite the above program in the following way:

```java
Node n = new Node(new Integer(9));
Node first = new Node(new Integer(5), n);
```
Deleting a Specified Node from a Linked List

Three main steps:
1. Locate the node to delete.
2. Disconnect this node from the linked list by changing references.
3. Return the node to the system.

E.g.: delete the second Node in the list

Let us assume that we already have a linked list and that we want to delete the second node in the list. The first task is to search for the node. This process is defined by the function “find” given in the next few slides. The deletion of an item can use two reference variables “curr” and “prev”. The first points to the current node that needs to be deleted, and the second to the previous node in the list. To delete the current node, we just need to alter the value of the field “next” in the node that precedes it, to make it reference to the node that follows the current node, thus bypassing it on the chain (the dashed line indicates the old reference value). Note that this reference change does not directly affect the current node. Since the reference “curr” still references the second node in the list, this remain in existence. Therefore, in order to return the node to the system, we need to either set “curr” equal to null, or change it to reference the node immediately after the one that has been deleted. The second diagram in this slide shows the second case. Note that it is also good practice to set the “next” field of the deleted node to null.

The reason for having the “prev” reference is to have a direct way to access the node that precedes the one that is going to be deleted, since the links in the list cannot be followed backwards! The following statement would be sufficient to delete the node that “curr” references:

`prev.setNext(curr.getNext());`

Deleting the first node in a list: It really wouldn’t make sense to say that “prev” references the node that precedes the first node in the list! Deleting the first node is therefore a special case, where “curr” references the first node and “prev” is null. NOTE: When we delete the first node in a list we must change the value of the pointer head, to reflects the fact that after the deletion the list has a new first node (i.e. the node that was second prior to the deletion is now first). We can make this change to “head” by using the following assignment statement:

`head = head.getNext();`

If the node to be deleted is the only node in the list, the above statement will assign the value null to the reference “head”, as it is supposed to be the case when the list is empty.
Inserting a Node into a Specified Position

Three main steps:
1. Determine the point of insertion
2. Create a new node and store the new data in it.
3. Connect the new node to the linked list by changing references.

E.g.: insert a new Node at position 3:

Let us assume that we already have a linked list and that we want to insert a new node into a specified position of a linked list. We assume that this new node is referenced by the variable newNode.

The first thing to do is to locate the given position within the linked list. This operation, implemented used the auxiliary method “find” given in slide 13, will give the appropriate values for the reference variables “pre” and “curr”. The insertion should then consist of adding a new node in the linked list between the nodes referenced by prev and curr, as illustrated in the second diagram. The following two assignment statements would be sufficient to perform this insertion:

\[
\text{newNode.setNext(curr);} \\
\text{prev.setNext(newNode);} \\
\]

The question now is “how did the variable newNode come to reference the appropriate new node?”.

This can be done by using the “new” operator in Java. We would need to declare before the actual insertion the following creation of the new Node:

\[
\text{newNode = new Node(item).} \\
\]

**Inserting a new node as first element:** The insertion of newNode as first element of the list is a special case. In this case, in fact, we have also to set the value of the pointer head to be equal to newNode, since head will have to point to the newNode as first element in the list. At the same time, the link in newNode has to be set equal to the value that was initially referenced by P. The following two assignments would accomplish this:

\[
\text{newNode.setNext(head);} \\
\text{head = newNode;} \\
\]

Note that if the list is empty before the insertion, the value of head is null; so the first assignment would assign to the link in the new Node the value null. This is correct, as in this case the new Node will be the first as well as the last node in the list!
Reference-based Implementation of the ADT List

- The interface `ListInterface` stays unchanged.
- The class `ListReferenceBased` implements the same interface:
  ```java
  public class ListReferenceBased implements ListInterface {
      private Node head;  // reference variable locating the first node in the list
      private int numItems; // number of items in the list
      // definitions of constructors and methods
      ........
  }
  ``
- Reference variables “prev” and “curr” are locale to the methods that need them; they are not data fields of the class `ListReferenceBased`.
- A method “find(i)” can be defined as private method that takes a position number i and returns a reference to the i-th node in the linked list.

How would we implement a linked list? We refer to this implementation with the term “reference-based” implementation of the ADT list.

The first thing to observe is that the interface `ListInterface` that we have defined previously is still valid, as it provides definition of the main operations allowed on the ADT list, which are independent on the type of data structure used to implement the list.

We can think of calling this reference based implementation “ListReferenceBased”. This will be a class that implements the same interface `ListInterface`. This class should therefore include the data field “head”, which is the reference variable used to locate the first node in the linked list, and the data field “numItems”, which gives the current number of items in the list. Both `head` and `numItems` are private fields of the ListReferenceBased class. These declarations provides the type declaration of a linked list. The remaining part of the class should include the implementation of the constructor, and of the methods that implements the access procedures of a list, as well as any private (auxiliary) method that may be needed. Among these private methods, an important one is the method for locating an item to delete or insert in the list. Such operations correspond to the list operations “remove” and “add” respectively. These take always an index value indicating a particular position in the list. The method `find(i)` should take this index value i and return a reference to the i-th node in the list. This reference can be used to determine the reference values of “prev” and “curr”.

**Note:** the two reference variables are not data field of the class `ListReferenceBased`, but rather reference variables local to the methods “add” and “delete” that need them.
In this slide I have given as example of the implementation of the (default) constructor for a linked list, and the implementation of two of its basic operations. I have also given the implementation of the auxiliary method “find(index)” with its pre and post conditions. Since this method is only auxiliary (i.e. meant to be used only by the access procedures of the ADT) it is declared to be private. This method is indeed needed by the ADT operations “get”, “add” and “remove”.

How would the implementations of “get”, “add”, and “remove” be? These are given to you as part of your second unassessed coursework.
List Variations

1) Doubly – Linked Lists

```
head prev item next
```

Each node has 2 reference variables to: previous node
next node

Reference variable to the previous node in the first node is null;
similarly, reference variable to the next node in the last node.

Data structure

```
public class Node{
    private Object item;
    private Node next;
    private Node previous;
    // definitions of constructors and methods
}
```

This is a particular variant of linked lists. A doubly-linked list has a different data structure in which
the class “Node” includes not only the information of the item in the list and the reference to the next
node in the list, but also a reference to the previous node in the list. This class Node will have also to
include a method “getPrecede()” to return the value of the variable referencing to the previous node.
Note that the reference to the previous node in the first element of a list is obviously equal to null in
the same way as the reference to the next node in the last element of a list is also equal to null.
The implementation of the access procedures “IsEmpty()”, “size()” are unchanged. The constructor
for the double linked list is the same as for the linked list: just assign head to be equal to null and the
numItems to be equal to zero.
The advantage of the doubly linked list is that it allows you, for instance, to delete a node without
traversing the list to establish the values of “curr” and “prev”. The value of “prev” is already included
in the node in question. In fact we could write “prev = curr.getPrecede();”.
Because there are more references to set, the mechanics of inserting into and deleting from a doubly
linked list are a bit more involved than for a single linked list. Also the special cases of inserting or
deleting a node at the beginning of such a list are more elaborated.
List Variations (continued)

2) Circular Lists

Is a single-linked list in which the last node references to the head of the list.

It allows searches for sequences of items to continue from a “current position”, rather than always starting at the head of the list.

When we traverse a linked list, if we reach the last node of the list, in order to access the first node, we must resort to the reference variable “head”. Suppose that we change the reference variable of the last node so that, instead of containing the value null, it references the first node. The result is a “circular linked list”.

The main property of this list is that every node in the list has a successor, so that we can start at any node and traverse the entire list.

Although we could think of a circular list as not having either a beginning or an end, it would still be useful to have an external reference variable “head” that references to one of the nodes in the list. If this “head” references to the first node, than to get to the last node, we would still traverse the entire list. If it references to the last node in the list, then we would be able to access to the first as well as to the last element of the list without traversing it.

A value “Null” for this external reference variable would still mean that the list is empty.

However, no reference variable of any node in such as list will have value Null. So, one question would be “how do we know that we have traversed the entire list?” We can compare the value of the next reference variable for a node with the value of “head” if “head” references the last element in the list.

**Note:** the operations insertion and deletion need to be changed, in this case as well.
Summary

Lists may be implemented:

- **Statically** using arrays which must be declared and are allocated fixed space before run-time. Index of items in a list are associated with index of the array.
- **Dynamically** using nodes which are allocated at run-time, and linked using reference variables.

Linked Lists are useful for:

- Maintaining sorted sequences of values.
- Representing sparse data.
- Building other more complex data structures.