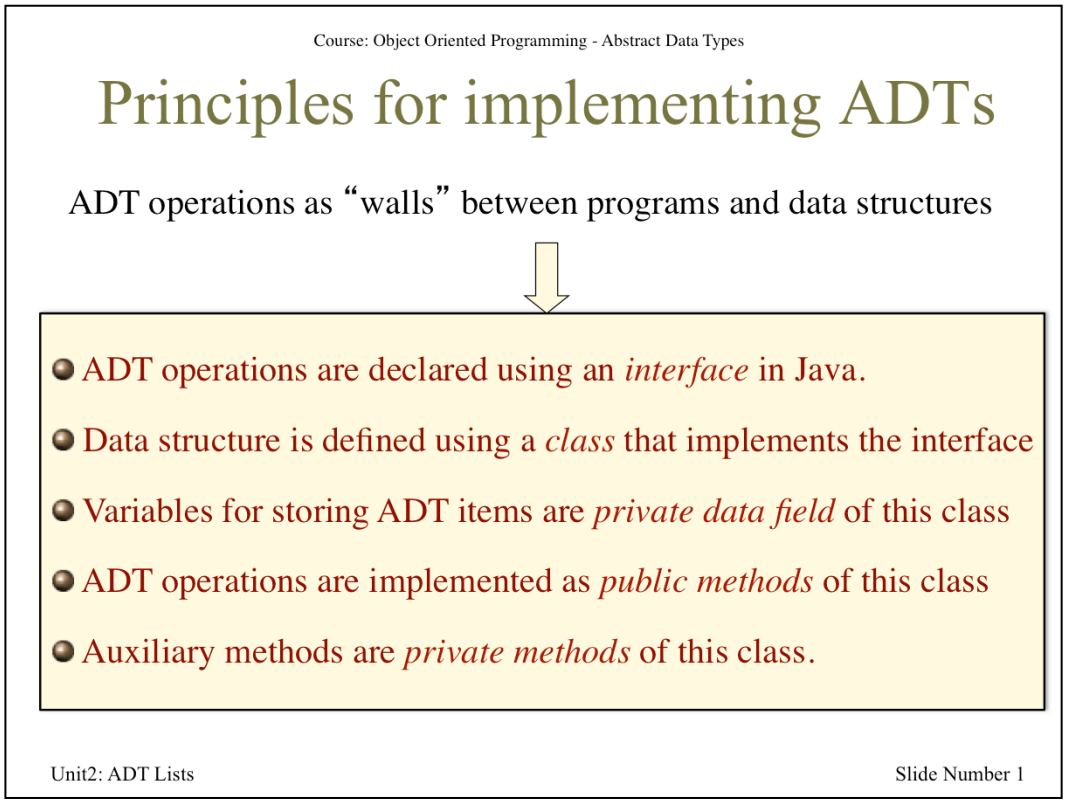


Let's remind ourselves what we discussed in the Introduction previous lecture to Abstract Data Types. We have said that ADT access procedures are like "walls" between any client program that uses an ADT and the actual data structure that implements the ADT. To implement an ADT, the most natural way is to define a Java interface that includes the declaration of the access procedures for the ADT that we want to create. Then device a class that implements this interface. This class will **encapsulates** the **data structure** necessary to store the data of our ADT. Note that such class may use one or more other classes, as data fields according to how "complex" the data structure needs to be for representing the elements of our ADT.

Each ADT operation is implemented as a *public method* in this ADT class. For instance, let us assume for the sake of argument, that we are using an array as data structure of our ADT list. So the array can be declared as data field of the ADT class List. To support the principle of encapsulation of the data structure, we need to keep the array data structure hidden from the client program. This is done by declaring the array to be a *private* field in the class List, and the operations that access the elements in the array as *public methods.*

The implementation of the access procedures may require *auxiliary* methods. These are normally declared as *private methods* of the ADT class. An example is the private method "translate(position)" given in the next few slides.

List Interface for the ADT List

```
public interface List<T>{
    public boolean isEmpty();
    //Pre: none
    //Post: Returns true if the list is empty, otherwise returns false.

    public int size();
    //Pre:none
    //Post: Returns the number of items currently in the list.

    public T get(int givenPos)
         throws ListIndexOutOfBoundsException
    //Pre: givenPos is the position in the list of the element to be retrieved
    //Post: If 1<= givenPos <= size(), the item at givenPos is returned.
    //Throws: ListIndexOutOfBoundsException  if givenPos < 1 or givenPos >= size()+1
```

Continued ….

This is a possible specification of the ADT list (i.e. as we have mentioned in the previous unit a possible contract between the client/application program and the ADT implementation). The ADT operations can be declared using a Java interface.

In this slide we have defined a generic interface for the ADT list, called `List<T>`. The notion of a list and its operations have to be independent from the type of elements that are stored in the list. The use of a generic interface facilitates this independence, as the type of the elements included in the list will be defined dynamically at run-time. An alternative definition of this interface could be given, where the type of the items in the list is Object. In this second case the use of access procedures would require castings to the specific types of items that are stored in the list. The use of generic interface helps avoid casting. Any class created in Java can be used as a type of the elements in the list. The type of the elements in a list has to be declared at the moment of creation of a list. This will appear clear when we see the class that implements the array-based data structure for our ADT list.

Note: In this interface we haven't included the ADT access procedure createList(). This is because this access procedure is normally given by the constructor of the class that implements the ADT list.

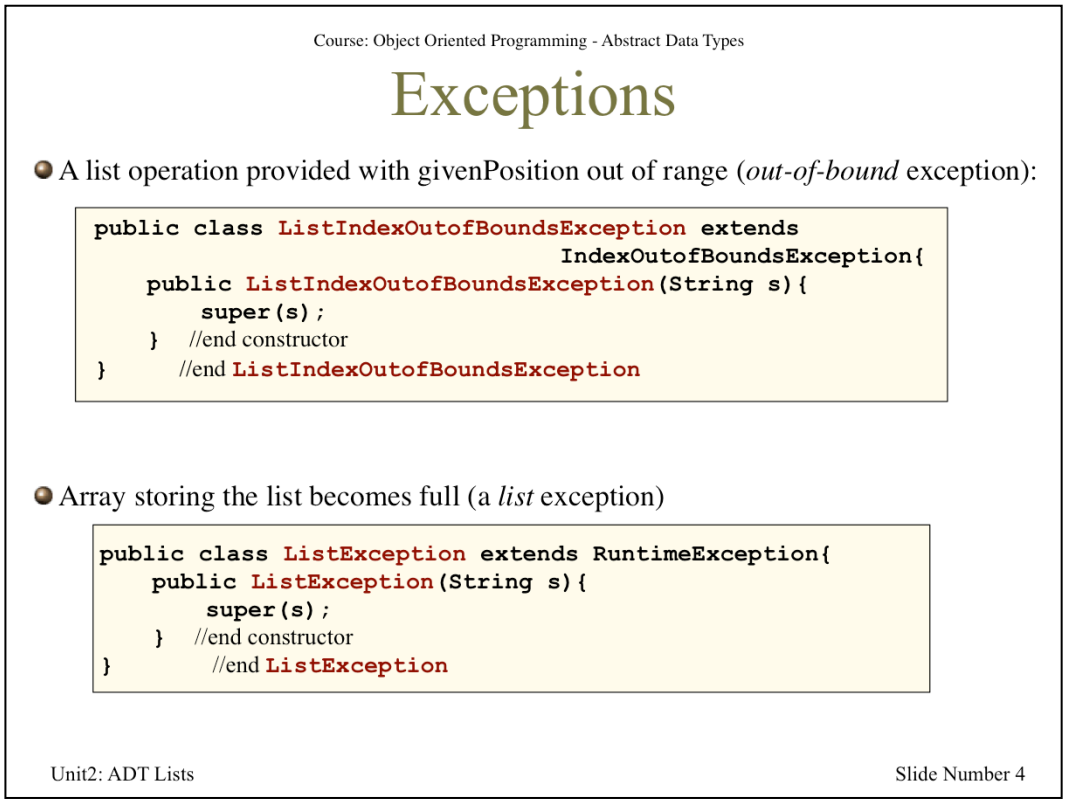
Note: The interface `List<T>` includes also the two methods given in the next slide, which define the list operations "add" and "remove" respectively .

…. *ListInterface* for the ADT List

```
public void add(int givenPos, T newItem)
      throws ListIndexOutOfBoundsException, ListException;
//Pre: givenPos indicates the position at which newItem should be inserted
//Post: If 1<= givenPos <= size() + 1, newItem is at givenPos, and elements
//       at givenPos and onwards are shifted one position to the right.
//Throws: ListIndexOutOfBoundsException when givenPos <1 or givenPos > size()+1
//Throws: ListException if newItem cannot be placed in the list.


public void remove(int givenPos)
      throws ListIndexOutOfBoundsException;
//Pre: givenPos indicates the position in the list of the element to be removed
//Post: If 1<= givenPos <= size(), the element at position givenPos is deleted, and
//       items at position greater than givenPos are shifted one position to the left.
//Throws: ListIndexOutOfBoundsException if givenPos < 1 or givenPos> size().

}  //end of List<T>
```



Some of the list operations have as parameter a position value that refers to a particular data item in the list. The `givenPos` parameter can, however, be out of "range". Each of the three operations `add`, `remove` and `get` might be provided a position value that is out of range. In this case the operation has to throw an exception. You have seen in the first part of the course that in case of exceptions you have a choice from the point of view of the client program: either deal with the possible raise of an exception, or propagate the exception upwards. In the case of ADTs the latter is a better choice.

We have defined in this slide a class that implements the out-of-range index exception for lists extending the more general `IndexOutOfBoundException` class given by the Java API.

The second exception here, is useful mainly in the case of static implementation of a list. As we will see in the following slides, a static implementation of a list uses an array as underlying data structure. Therefore, in this case, we would need to guess a priori a maximum size of the list for the Java runtime environment to reserve a specific number of memory references for the elements of the array. So the exception `ListException` would occur whenever the array storing the list becomes full and we are trying to add new items to it. We have defined this exception with the class `ListException`, which extends the general `RuntimeException` of Java.

Note: Remember that `RuntimeException` is the Java superclass of all those exceptions that can be thrown and left unchecked during the normal operation of the Java Virtual machine. `IndexOutOfBoundsException` is instead an exception thrown to indicate that an index of some sort is out of range. It extends the `RuntimeException`, but applications normally use the subclass `IndexOutOfBoundsException` to indicate exceptions similar to an index out of range. This is why `ListIndexOutOfBoundsException` has been here defined as a subclass of `IndexOutBoundsException` instead of `RuntimeException`.

Static Implementation of ADT List

A static implementation:

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

We need:

- a variable to keep track of the current number of elements 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:

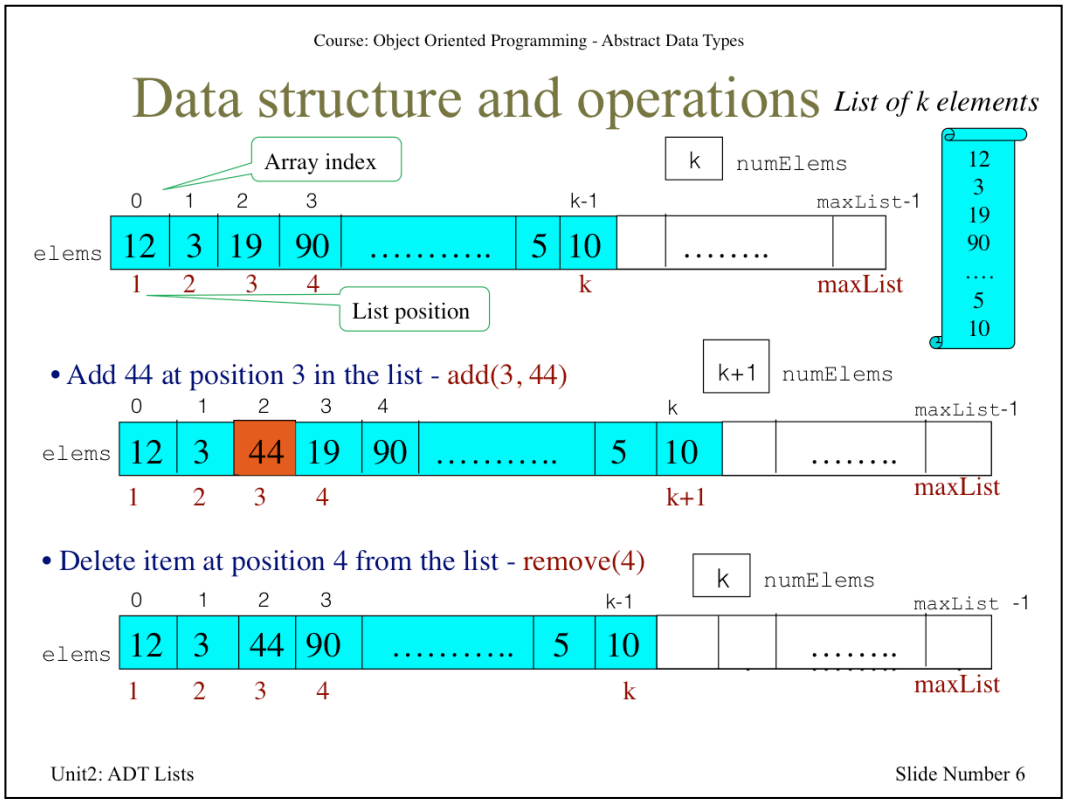
```
private final int maxList = 100;      //max length of the list
private T elems[maxList];             //array of elements in the list
private int numElems;                 //current size of the list
```


A static implementation of a list is an array-based implementation. This is a natural choice since both an array and a list identify their items by position. We call this array "elems". But, how much of the array will the list occupy? Possibly all of it, but probably not. Therefore, we need to keep track of the current length of the list (i.e. how many elements are currently included in the list) and how many spaces are unoccupied. We can define maxList as a pre-fixed value representing the physical size of the array. Note that in this slide we have assumed it to be a final attribute, but it can also be possible to define maxList as a parameter for the construct of the class that implements the List interface. In either cases we do need to store, as part of the data structure, the information of maximum length of the list.

In addition, we need to keep track of the current number of elements in the list, that is the list's "logical" size. We do this by using a variable called numElems. The value of numElems gives not only the current number of elements in the list but, at the same time, the array index of the next available space in the array, since array index starts from 0.

We said above that both arrays and lists identify their elements by position (i.e. the index in the array and the logical position in the list, respectively). Note, however, that whereas the positions of elements in a list starts from 1 (the first elements in a list is at position 1), the array index of the first element is 0. Hence, the list's k^{th} element will be stored in elems[k-1].

A private method "translate(index)" can be defined in the class that implements a list, as auxiliary method, to translate the logical position of an element in the list into its corresponding array index value.



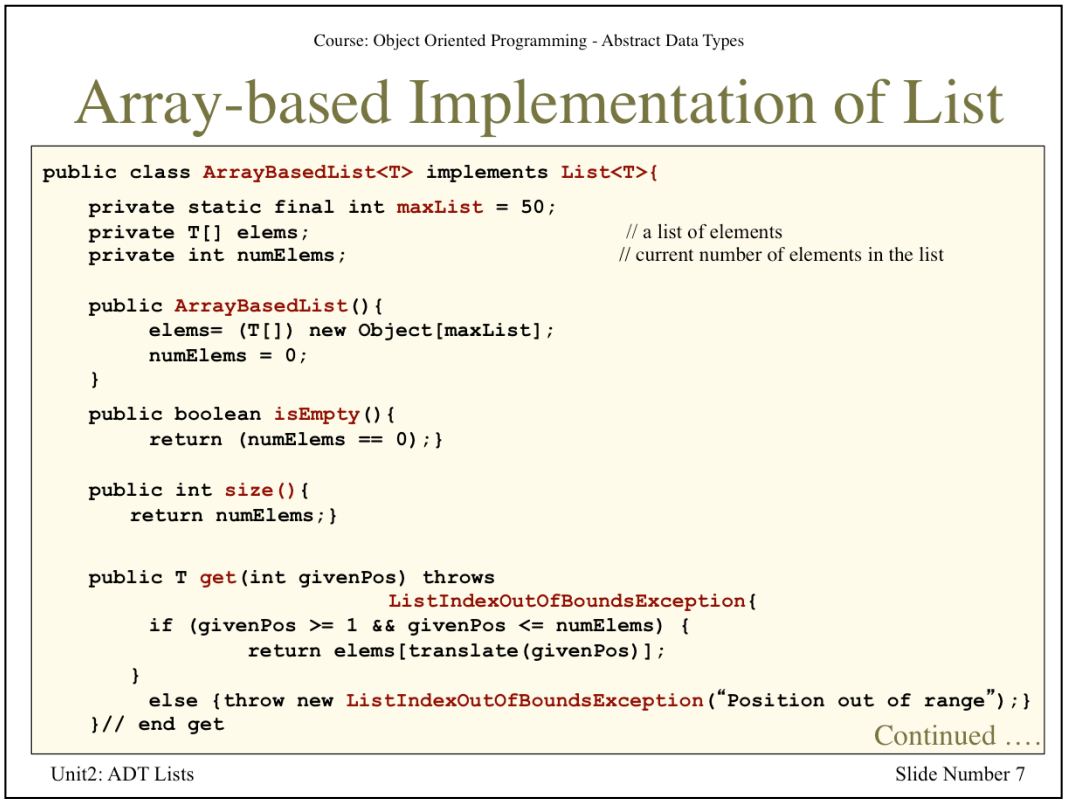
This slide gives a diagrammatical representation of the data structure used by an array-based implementation of the ADT list and the effect that the two operations of addition and deletion have on the data structure. We are considering here an example of a list of integers. But in general the type of the element of a list can be any object, a basic type of Java or a generic type.

To insert a new item at a given position in the list, we must "**make space**" in the array at the array index that corresponds to the given position in the list, i.e. shift to the right the elements in the list that are at the given position onwards, and insert the new element in the newly created opening. This is illustrated diagrammatically in the second array picture above.

Let's see instead how to delete an item at a particular position in the list. We could in principle just blank it out, but this strategy leads to gaps in the array. An array that is full of gaps will give the following problems:

1. `numElems`–1 is no longer the array index of the last element in the array, since `numElems` will appropriately be decreased because of the delete operation but the array index of the last element used for the list will not necessarily be decreased. We would therefore need to use an additional variable to keep track of the last position used in the array.
2. Because the items are spread out, the method **get** might have to look at every cell of the array.
3. When `elems[maxList-1]` is occupied, the list could appear full, even when fewer than `maxList` elements are present in the list.

Thus, what we need to do is to shift the elements of the array to fill the gap left by the deleted item, as shown in the third diagram above.



This is an example implementation of the class `ArrayBasedList<T>`. This class is an example of static implementation of an ADT list. The data fields provide the underlying data structure, and the methods are implementations of the access procedures declared in the interface `List<T>` presented in slides 2 and 3.

An example of an array with maximum size equal to 50 is considered here, where each element of the array is of generic type `T`. The constructor `ArrayBasedList()` is the implementation of the list operation createList() given in Unit 1. The other methods here are quite straightforward. Note that they are declared to be public methods as they will be called by a client/application program that needs to use a list (e.g. text editor class given in Unit 1).

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

The implementation of "get" uses a method call "`translate(index)`". This is a private method used to translate the position that an element has in the list into its corresponding array index value in the array. This is because, as we have already mentioned in the previous slide, the position of elements 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 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 list and as such it has to be declared "private". The example implementation of "`translate`" is also given in the next slide.

A full implementation of the class `ArrayBasedList<T>` includes also the methods given in the next two slides.

Array-based Implementation of List

```
public void add(int givenPos, T newItem) throws
                    ListIndexOutOfBoundsException, ListException {
    if (numElems == maxList) {
        throw new ListException("List is full"); }
    if (givenPos >=1 && givenPos <= numElems + 1){
        makeRoom(givenPos);
        elems[translate(givenPos)]= newItem;        //insert newItem
        numElems++;
    }
    else throw new ListIndexOutOfBoundsException("Position out of range");
}// end add


private void makeRoom(int position) {
//pre: 1 <= position <= numElems + 1
    for (int pos=numElems; pos>=position; pos--) {
      elems[translate(pos+1)] = elems[translate(pos)]; }
}

private int translate(int position){
     return position -1;
}//end translate
```

Continued ....

This slide provides an example implementation of the operation "`add`".

The private method `makeRoom`, instead, shifts the elements from `givenPos` onwards one position to the right in order to make room in the array at `givenPos` for the new item.

Array-based Implementation of List

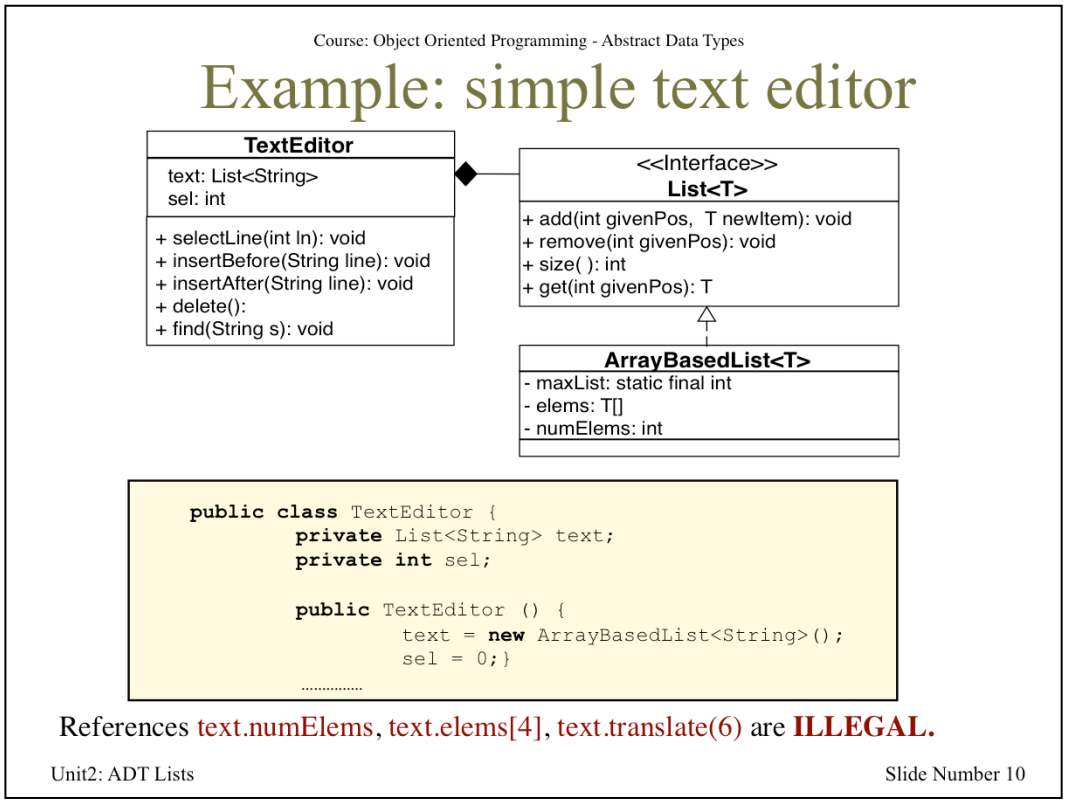
```
    public void remove(int givenPos) throws ListIndexOutOfBoundsException{
        if (givenPos >=1 && givenPos <= numElems){
            if (givenPos < numElems){
             // delete item by shifting left all item at position > givenPos
             removeGap(givenPos);
            }
            numElems--;
        }
        else throw new ListIndexOutOfBoundsException("Position out of range");
    }// end remove


    private void removeGap(int position){
    //pre: 1=< position < numElems
        for (int pos=position+1; pos<=size(); pos++){
            elems[translate(pos-1)] = elems[translate(pos)];}
    }

}//end ListArrayBased
```


This slide gives the implementation of the operation `remove` of our class `ArrayBasedList<T>`. Removing an element at a `givenPos`, is performed by shifting all the elements in the list at position > `givenPos` one place to the left. This is done by the auxiliary method `removeGap`.

Note that if the element that we want to remove is the last element in the list, i.e. the `givenPos` is equal to `numElems`, then the deletion is done by just decreasing the value of total number of elements in the list.



Now that we have defined the ADT List using a generic interface `List<T>` and implemented it by means of the class `ArrayBasedList<T>` we can refine our example of a simple Text editor. The class diagram shows the application program (i.e. class `TextEditor`), the generic interface `List<T>` that acts as "wall" between the application program and the class `ArrayBasedList<T>` that encapsulates the data structure used for implementing the List.

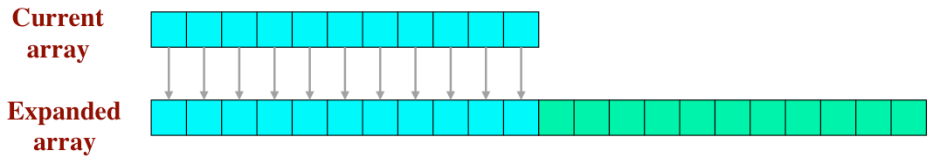
Different list implementations could be provided by the ADT implementer, but the application environment can still define its own methods (e.g. `insertAfter`, `insertBefore`, etc…) only in terms of the access procedures declared in the interface `List<T>` as shown in Slide 7, independently from the implementation of the List chosen. The only point in the application program where reference to the implementation of the List is made, is at creation time of an object `List`, as shown above in the constructor of the `TextEditor` class.

Note that in this application program direct reference to the data structures of the list are **ILLEGAL.** So statements of the form `text.numElems`, `text.elems[4]`, `text.translate(6)` are **ILLEGAL.**

One main feature of this type of static-implementation is that the fixed side of the underlying data structure allows only the implementation of lists that have limited length. This could be appropriate for applications where the limited length of a list is required. For instance, a list of passengers of a flight, a list of ticket holders to a movie should in general not exceed a known maximum capacity. For other applications, however, where the list could grow without bound, a static implementation with a fixed array data structure would not be ideal. You could think of using a very large maximum length to avoid cases of list full, but of course this would cause memory waste whenever we don't use all of the locations in the array. The task of the ADT implementer is therefore to provide the best choice of data structure and corresponding efficient ways of implementing the ADT access procedures for that data structure, where efficiency is in terms of time and/or memory. In the next slide, we suggest an alternative static implementations, that would allow a more efficient use of the memory in static-implementations of lists, without affecting too much the execution time.

Alternative static-implementation

Making use of dynamic expansion of an array



- Requires copying all existing elements of a list from the current array data structures to a new larger array
- Larger array has to be at least the double of the original one
- Extra computational cost but controllable memory waste.

A possible way of overcoming the problem of fixed size list is to use dynamic expansion of the underlying array data structure. Dynamic expansion means copying the content of the current (full) array (in our case the "`elems`" data field of the class `ArrayBasedList`) into a new array with double length than that of `elems`. Of course when doing this dynamic expansion you have to make sure that **the name of this new array is the same as that of the original array**.

What are then the advantages and drawbacks of using a dynamic expansion? You could avoid situations of list full whenever the array reaches its maximum pre-defined length. This would give the impression that you are using an underlying data structure that is dynamically allocated to store in principle unbound lists. However each expansion requires computational costs for copying the elements across from the original array to the larger one. You could reduce this cost by using larger expanded arrays that have at least double length of that of the original array so to spread out the cost of copying n elements throughout each group of n addition operations to the list (where n is the length of the original array).

Lists as part of Java Collection

Java provides an interface List similar to the one defined in this lecture. It is part of `java.util.List`

```
public interface List<E> extends Collection<E>
```

Java provides various implement the list interface, each using specific data structures.

`java.util.ArrayList<E>` represents each list by an array.

`java.util.LinkedList<E>` represents each list by a doubly-linked list.

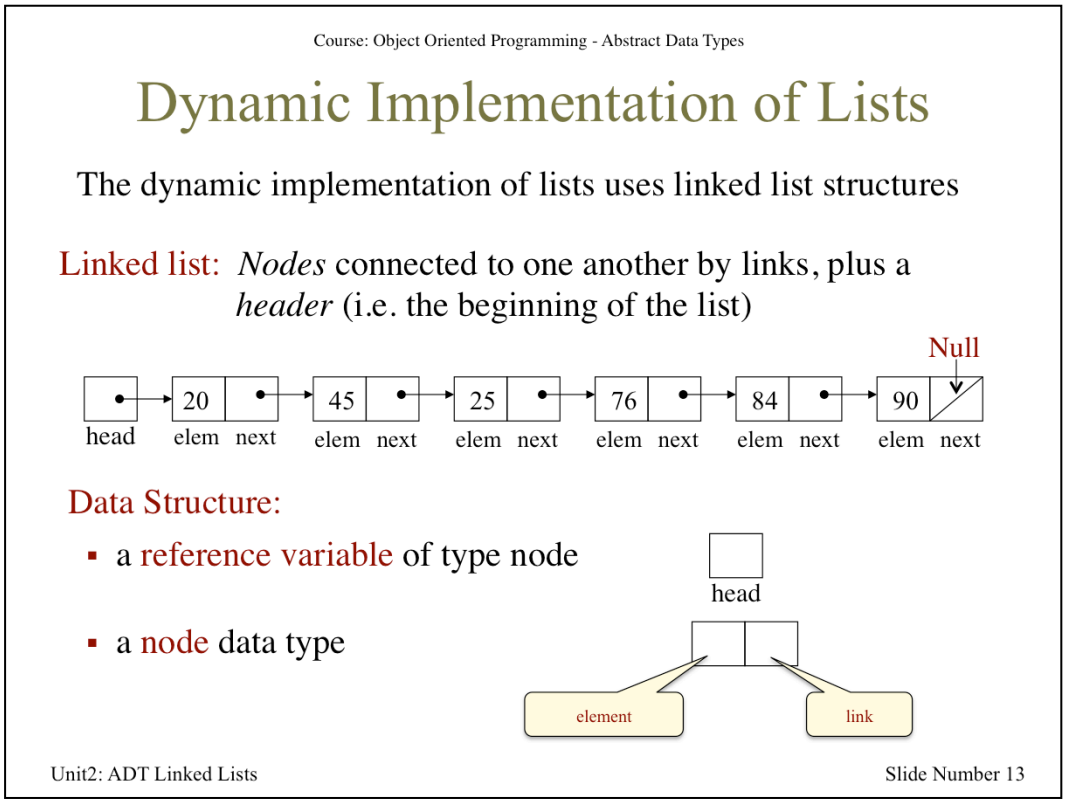
The above implementations are not synchronized. It requires external mechanisms for synchronizing access by multiple threads to the same list.

The ADT we have just presented includes the core operations of the Java `List<E>` interface included in the collection framework. The Java `List<E>` extends Collection and Iterable. We will see in the next lecture how to define an iterator over a list.

The access procedures provided here are subset of the methods provided by the Java `List<E>` interface. Note that the method `remove` in the latter returns also the element that is deleted from the list.

Different classes are provided in Java as implementations of the `List<E>` interface. The main ones are the `ArrayList<E>` and `LinkedList<E>`. They correspond to the two types of implementations of Lists that we are covering in this course, namely the static and the dynamic implementation. The example code given in this lecture corresponds to the implementation of the corresponding methods in the `ArrayList<E>` class. The dynamic expansion is also supported in the `ArrayList<E>`.

We will see in the next lecture the main features of a LinkedList implementation. One thing to remember is that the above JAVA classes as well as the example provided in this course are not synchronised. This means that in principle if we have multiple threads manipulating the same List, their access to the list is free and structure modifications made by one thread might after the execution of the other threads. Their access needs to be synchronised through some external mechanisms in order to control the changes they make on the ADT.



The static implementation of a list is not always the best way of implementing this ADT as it either leads to waste of memory or to more computationally expensive implementation of the access procedures. An additional disadvantage of static approach is that it requires the shifting of data whenever we want to add or remove an element in order to avoid gaps in the data structure.

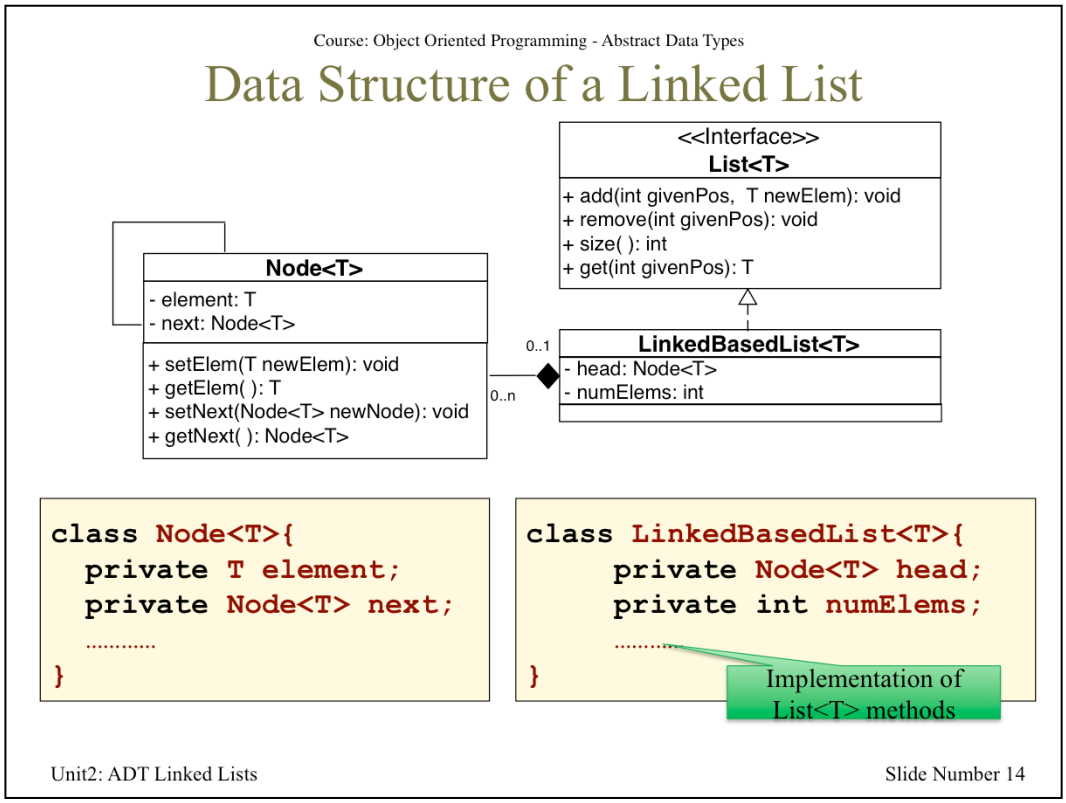
The dynamic implementation overcomes these limitations. The data structure of a dynamic implementation is that of a linked list, which consists of nodes connected to one another by links (i.e. reference variables) and a header variable that references the first node in the linked structure (i.e. the first element in the list). The data structure of a linked list includes therefore a "reference variable" (the head) and a "node" data type that stores an element of the list and creates the link with the next node (i.e. element) in the list. The **reference variable** `head` has the sole purpose of locating the first node in the list. A **node** data type includes an attribute `elem`, whose type defines the type of the elements in the list, and a `next` reference variable of type `node` that references the next element in the list.

In a linked list we can manipulate the individual elements and the links. We can therefore change the very structure of the list, which is only possible in an array-based implementation by means of deletion and additions operations.

The last element in the list is itself a node, but whose link reference variable has value "**null**" since there is no other node after it!

Note *head* is a reference variable of type Node, but it is not itself an Object of type Node. It differs from the other links in the diagram above in that it is external to the data structure Node. The reference variable `next` is instead an attribute of a Node object, and it is therefore declared within a class Node that implements the node data structure of the list.

Note that the head reference variable **always exists**, even when there are no nodes in the linked list. In particular: **when the value of the head is null, it means that it does not reference any Node and therefore that the list is empty.**

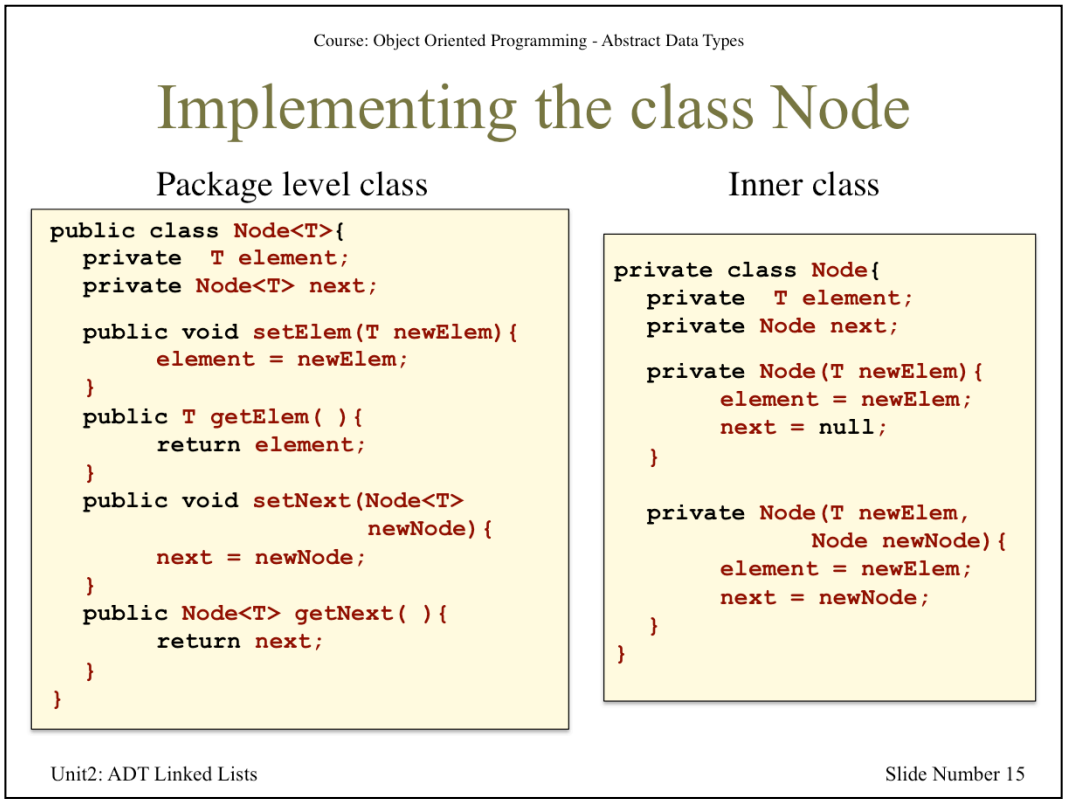


The underlying data structure for a linked list includes a class `Node<T>`, whose object stores an element of the list and the link to the next element in the list, a reference variable "`head`", which references the first node in the list and the integer `numElems` that keeps track of the number of nodes in the linked list. The latter is really superfluous but we include it here for uniformity with the array-based implementation that we saw in the previous lecture. The `head` and `numElems` are attributes of the class `LinkedBasedList` that implements the interface `List<T>`, whereas the class `Node<T>` is the implementation of the underlying data structure used by the `LinkedBasedList<T>` class.

A dynamic implementation of a list is essentially given by the implementation of a class `LinkedBasedList<T>`, which includes information about the number of elements in the list and the reference variable "*head*" for pointing to the first element in the list. The class `Node<T>` is really a data type that provides the data structure used by the class `LinkedBasedList<T>` for storing the elements in a linked manner.

The class `Node` is here declared as a generic class, where the generic type T is the type of its element. In this case we are assuming that this class has a package access (e.g. in the sense that it can be used by other ADTs implementation such as queues and stacks). But you could also opt to define Node to be a **inner class** of the class `LinkedBasedList<T>`. In this case the class `Node` would be private and so will be all its constructors. As inner class it does not need to be generic and the type T can be still the type of element as it will be dynamically linked by the binding that T has in the container class (i.e. `LinkedBasedList<T>`).

What is important to notice is that the data field `next` is not a reference variable of type T but a reference variable for the entire next node.



The data structure of linked list is more complex than the array used in the static implementation. First of all, its component node is itself a data structure and not just an element type as it is in the case for the static implementation.

This slide shows two different approaches for implementing the `Node` data structure. The one on the left is the implementation for a class Node which can be used by any other class within the same package. The one on the right is an example implementation of the class Node as a inner class of the `LinkedBasedList` class.

In both cases the attributes of the class `Node` have to be declared as private in order to guarantee access to the elements of the List only through the access procedures and not directly through some other classes.

As a consequence the class `Node` on the left hand side has to include methods for accessing (accessors) and changing (mutators) the values of its attributes. The *mutator* methods `setElem` and `setNext` set respectively the values of the two attributes, whereas the *accessor* methods `getElem` and `getNext` read the values of these two data fields.

In the case of the inner class `Node,` given in the right hand side of the slide, the attributes (although declared private) are directly accessible by the enclosing class. So no additional methods are needed. This type of implementation allows full encapsulation of the ADT and of its own underlying data structure, since the inner class `Node` would only be accessible by the class `LinkedBasedList`.

The examples of constructors given for the inner class, can be included in either of these classes. Note that, as also stated in the lecture on Generics, the constructor of a generic class does not take the type specification <T>. Therefore, the constructors of our class `Node<T>` are very similar to the constructors of a non-generic class Node.

Implementing LinkedBasedList(1)

```
public class LinkedBasedList<T> implements List<T>{

   private Node head;
   private int numElems;

   public LinkedBasedList(){
       head = null;
       numElems = 0;
   }

   <<Implementation of public access procedures isEmpty, size,
    get, add, remove go here>>

   private Node getNodeAt(int givenPos)
   <implementation deferred>

   private class Node{
       << given in slide 15 >>
   }
}
```


This slide gives an example of partial implementation of a class `LinkedBasedList<T>` that provides the dynamic implementation of an ADT list. In the tutorial you are asked to complete this implementation. Note that it makes use of the inner class `Node`.

The method `getNodeAt` is an auxiliary method needed to locate the node at a particular position. It normally returns the reference of a node object at position givenPos, and it assumes that the list is not empty and that the given position is within the range [1..numElems].

In the next set of slides we will see how to implement a linked list in full, assuming the data structure Node to be a public class instead of a inner class. This is because we will be using later on in the course the same class Node for the dynamic implementation of other ADTs like queues and stacks.

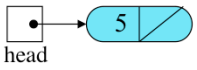
Course: Object Oriented Programming - Abstract Data Types

Implementing LinkedBasedList(2)

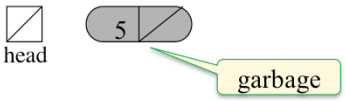
Main invariants

- The `next` attribute of the last node object 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.

```
head = new Node(new Integer(5));
```

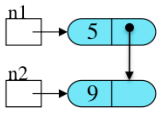



```
head = null;
```



Creating and Linking two nodes

```
Node<Integer> n1 = new Node( );
Node<Integer> n2 = new Node( );
n1.setElem(new Integer(5));
n2.setElem(new Integer(9));
n1.setNext(n2);
```



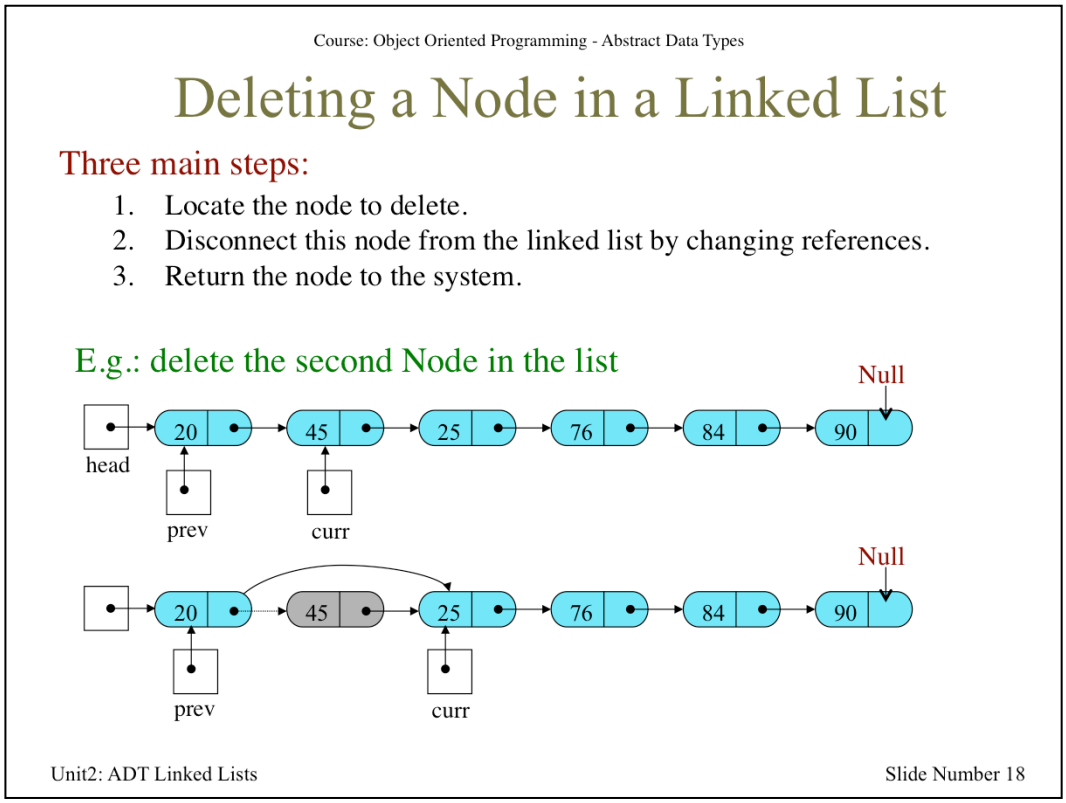
Unit2: ADT Linked Lists Slide Number 17

In the next few slides we show how to implement a linked list, dynamic data structure for a list. This implementation is different from the one given in the previous slide as it assume a public generic class `Node<T>` with its own accessor and mutator methods for its attributes.

Above, we give an example of how to create two nodes and link them together, i.e. how to form a chain of nodes. The first two Java statements create two objects of the class `Node` referred to by the reference variables `n1` and `n2` respectively. The third and forth Java statements set the value of the data field `element` in the objects `n1` and `n2` respectively. The last statement, instead, links the two nodes by making the first node reference the second. Note that in this example we have used just the default constructor for the class Node. Using the other `Node` constructors given in Slide 15 we could rewrite the above five lines code in the following more compact code:

```
Node<Integer>  n2 = new Node(new Integer(9));
Node<Integer> n1 = new Node(new Integer(5), n2);
```

In the dynamic implementation of ADT lists, adding an element to a list means, in effect, creating a node object that stores the element, and linking this node object in the correct position in the linked list by appropriately setting (or changing) the "links" of the nodes adjacent to its position in the list. Similarly deleting an element from a list means changing the links in the chain so that the previous node does not longer reference the node that needs to be removed. These operations are diagrammatically illustrated in the next slide.



Let us assume that we already have a linked list and that we want to delete a node in the list. The first task is to search for the node. This process is defined by the auxiliary function "`getNodeAt`" 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 change the value of the field "`next`" in the node that precedes it (i.e. in the node referenced by "`prev`") so to reference the node that follows the current node, thus bypassing the current node in the chain. The dashed line indicates the old value of next in the previous node. Note that this reference change does not directly affect the current node.

The diagram in this slide shows the deletion of the second node. 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())
```

Does this process allows the deletion of nodes at the end of a list, or the only singleton node in the list and nodes in the middle of a list in a way that the invariants given before are still satisfies? This is discussed in the lecture.

Deleting the first node in a list: The above process does not make much sense in the case of deleting the first node in the list because such a node does not have a precedent nodes. This case needs to be dealt separately. Deleting the first node is therefore a special case, where "`curr`" references the first node and "`prev`" is null. When we delete the first node in a list we must change the value of the `head`, to reference to the second node in the list (which after the deletion becomes the first node). We can make this change to "`head`" by using the following assignment statement:

```
head = head.getNext( );
```

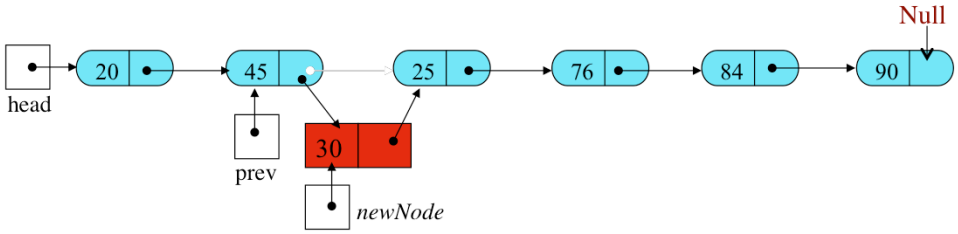
If the node to be deleted is the only node in the list, the above statement will assign the value *null* to the reference "head", as it is supposed to be always equal to null when the list is empty.

Adding a Node in a Linked List

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 given position. 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 using the same auxiliary method "`getNodeAt`", used for the deletion of a node, will give the reference to the node that just precedes the position where the new node needs to be added. The insertion should then consist of "`chaining`" a new node in the linked list between this precedent node and the next one. The following two assignment statements would be sufficient to perform this insertion:

```
newNode.setNext(pre.getNext());
prev.setNext(newNode);
```

Note that in the addition operation we don't need to create two reference variables "`curr`" and "`prev`". The reference "`prev`" would be sufficient. We could use the method `getNodeAt` as in the delete operation and get the reference value for `curr` from the previous node.

Inserting a new node as fist 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 data field `next` in `newNode` has to be set equal to the value that was initially referenced by `head` . The following two assignments would accomplish this:

```
newNode.setNext(head);
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 attribute `next` of `newNode` 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!

Would the above cases be sufficient for inserting a node at the end of the linked list? This is discussed during the lecture.

Course: Object Oriented Programming - Abstract Data Types

Implementing LinkedBasedList (2)

- The interface `List<T>` is the same as for `ArrayBasedList<T>`
- The class `LinkedBasedList<T>` implements `List<T>`:

```
public class LinkedBasedList<T> implements List<T>{
   private Node<T> head;
   private int numElems;
   <constructors and methods go here>
}
```

- Reference variables "prev" and "curr" are local to the methods that need them; they are not data fields of the class LinkedBasedList<T>.
- A method "`getNodeAt(i)`" is an auxiliary method that takes a position *i* and returns a reference to the i^{th} node in the linked list.

Unit2: ADT Linked Lists Slide Number 20

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

The interface `List<T>`, defined in Unit2 for static implementation of lists, is still valid since the access procedures are still those for an ADT list. It's only the underlying data structure and implementation of these procedures that change.

We can think of calling this reference-based implementation "`LinkedBasedList<T>`". This is again a generic class where the generic type T is assumed to be the type of the elements in the list. This will be a class that implements the generic interface `List<T>`.

This class should include the data field "`head`", which is the reference variable used to locate the first node in the linked list, and the data field "`numElems`", which keeps track of the current number of elements in the list. Both `head` and `numElems` are private fields of the `LinkedBasedList<T>` class.

The remaining part of the class includes the implementation of the constructor, and of the access procedures of an ADT List, as well as any private (auxiliary) method that may be needed. Among these auxiliary methods, an important one is the method "`getNodeAt`" for locating a Node at a given position, in order to either delete it or locate it as the precedent node for a new node to be added to the list. This method is used in particular by the access procedures "add", "delete" and "get" in order to determine the value of the reference variable "`prev`" or "`curr`".

Note that the two reference variables "prev" and "curr" are not data field of the class LinkedBasedList<T>, but are reference variables local to the methods "add" and "delete".

Course: Object Oriented Programming - Abstract Data Types

Implementing LinkedBasedList (2)

- The interface `List<T>` is the same as for `ListArrayBased<T>`
- The class `LinkedBasedList<T>` implements `List<T>`:

```
public class LinkedBasedList<T> implements List<T>{
   private Node<T> head;
   private int numElems;
   <constructors and methods go here>
}
```

- Reference variables "prev" and "curr" are local to the methods that need them; they are not data fields of the class LinkedBasedList<T>.
- A method "`getNodeAt(i)`" is an auxiliary method that takes a position *i* and returns a reference to the i^{th} node in the linked list.

Unit2: ADT Linked Lists Slide Number 21

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

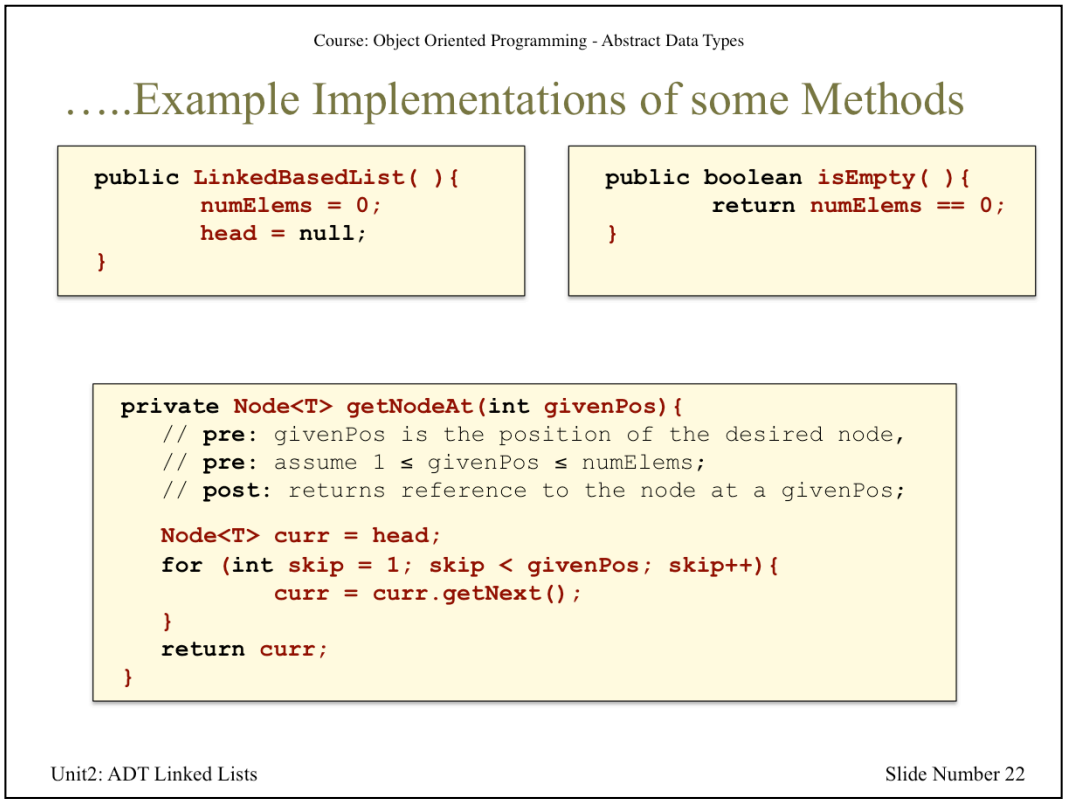
The interface `List<T>`, defined in Unit2 for static implementation of lists, is still valid since the access procedures are still those for an ADT list. It's only the underlying data structure and implementation of these procedures that change.

We can think of calling this reference-based implementation "`LinkedBasedList<T>`". This is again a generic class where the generic type T is assumed to be the type of the elements in the list. This will be a class that implements the generic interface `List<T>`.

This class should include the data field "`head`", which is the reference variable used to locate the first node in the linked list, and the data field "`numElems`", which keeps track of the current number of elements in the list. Both `head` and `numElems` are private fields of the `LinkedBasedList<T>` class.

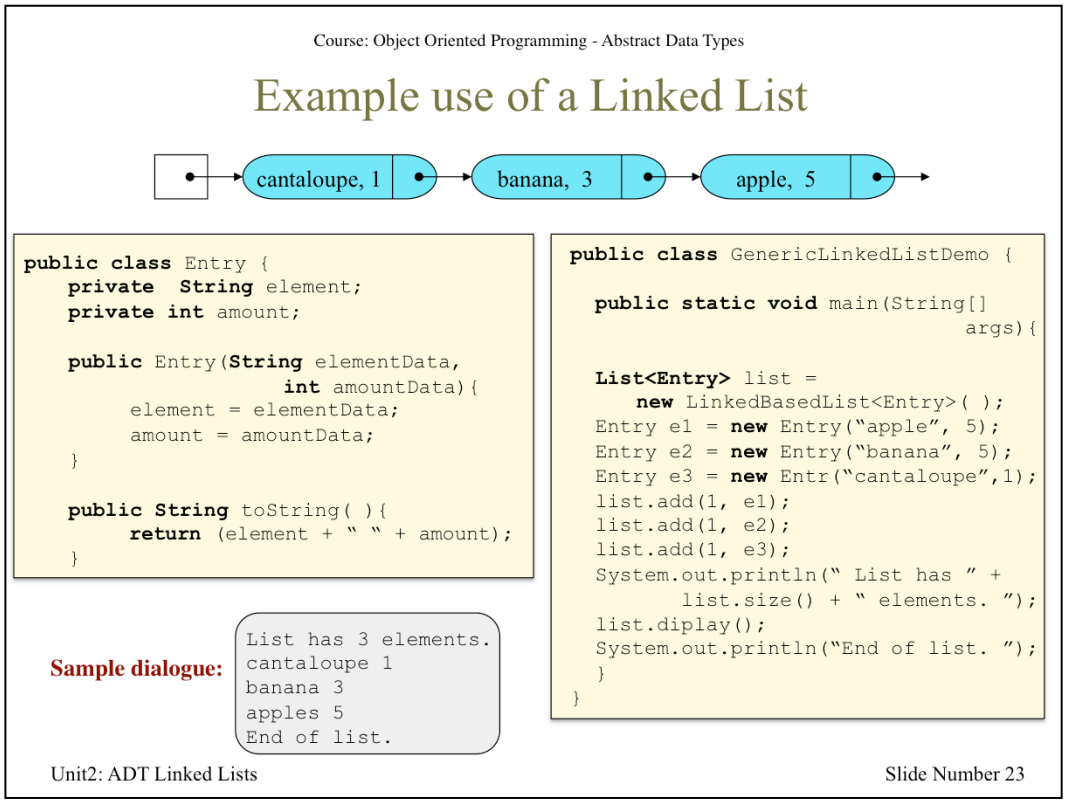
The remaining part of the class includes the implementation of the constructor, and of the access procedures of an ADT List, as well as any private (auxiliary) method that may be needed. Among these auxiliary methods, an important one is the method "`getNodeAt`" for locating a Node at a given position, in order to either delete it or locate it as the precedent node for a new node to be added to the list. This method is used in particular by the access procedures "add", "delete" and "get" in order to determine the value of the reference variable "`prev`" or "`curr`".

Note that the two reference variables "prev" and "curr" are not data field of the class LinkedBasedList<T>, but are reference variables local to the methods "add" and "delete".



In this slide I have given example implementation of the (default) constructor for a linked list, and one of its basic access procedures. I have also given the implementation of the auxiliary method "`getNodeAt(int givenPos)`" 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 unassessed Tutorial 1.



This slide shows an example use of Linked Lists. The element is of type `Entry` and the class `GenericLinkedListDemo` constructs a generic Linked List, adds three elements to the list and prints the elements in the list. The method `display()` is assumed here to be another access procedure of the class `LinkedBasedList.`

An example of how this access procedure might be implemented is given here.

```
public void display( ){
      Node<T> element = head;
      while (element != null){
         System.out.println((element.getElem( )));
         element = element.getNext( );
     }
}
```

You can define a more efficient method for printing the elements in a linked list that uses an iterator. This is shown at the end of this lecture.

Note also that for the command "System.out.println(element.getElem( ))" to work, the class Entry has to have a toString method.

Course: Object Oriented Programming - Abstract Data Types

List Variations

Doubly – Linked Lists

Node has 2 reference variables for previous node and next node

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

Node<T>

```
public class Node<T>{
        private  T item;
        private  Node<T> next;
        private  Node<T> previous;
        < constructors, accessor and
        mutator methods here>
}
```

Unit2: ADT Linked Lists Slide Number 24

This is a particular variant of linked lists. A doubly-linked list is a different data structure in which the class "Node" includes not only the information of the element in the list and the reference to the next node in the list, but also a reference to the previous node in the list. In this case the class Node has to include a method "`getPrevious()`" to return the reference 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 `LinkedBasedList` implementation may well include also a reference to the tail of the list. In this case more efficient access procedures could be implemented for accessing the last element of the list, appending an element at the end of list, or deleting the last element of a list, which would not require scanning the list from its head to its tail. The tail reference would allow direct access to the end of a list.

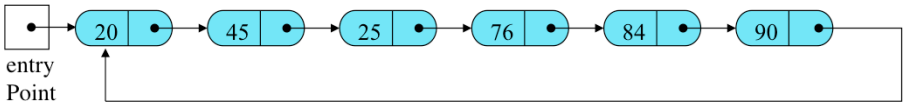
The implementation of the access procedures "`isEmpty()`", "`size()`" would remain unchanged. The constructor for the double linked list is the same as for the linked list: just assign head and tail to be equal to null and the `numElems` to be equal to zero.

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 "prev" and then "curr". We can directly locate the item at the given position and do the linkage needed for deleting or adding an element. The value of "prev" is already included in the node in question. In fact we could write "prev = curr.getPrevious();". Because there are more references to set, the mechanics of inserting and deleting from a doubly linked list are a bit more involved than that used for a single linked list.

You could also consider including additional access procedures to remove the last element in the list and remove the front element in the list.

List Variations (continued)

Circular Lists



A single-linked list in which the last node references the head of the list.

It allows the search for sequences of elements 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 "entryPoint". 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 "`entryPoint`" that references one of the nodes in the list. If this "`entryPoint`" references the first node, than to get to the last node, we would still traverse the entire list. If it references the last node in the list, then we would be able to access the first as well as 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 `'next" reference variable in any node of such a 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 "`entryPoint`", if "`entryPoint`" references the last element in the list.

Note: the operations of addition and deletion need to be changed accordingly.

Ordered Linked List

An ordered linked list is a linked list where elements have a special attribute, called key, and are totally ordered with respect to their key.

Access procedures:

createOrderedList( )
// post: Create an empty ordered list

put(searchKey, givenValue)
//post: Insert in the list an element with searchKey
//and givenValue in the correct position.
//If a node with searchKey exists, set its
//value to be equal to givenValue.

remove(searchKey)
//post: Removes the element whose key is equal
//to searchKey. Throws exception if such an
//element does not exist.

isEmpty( )
// post: Determine if an ordered list is
//empty

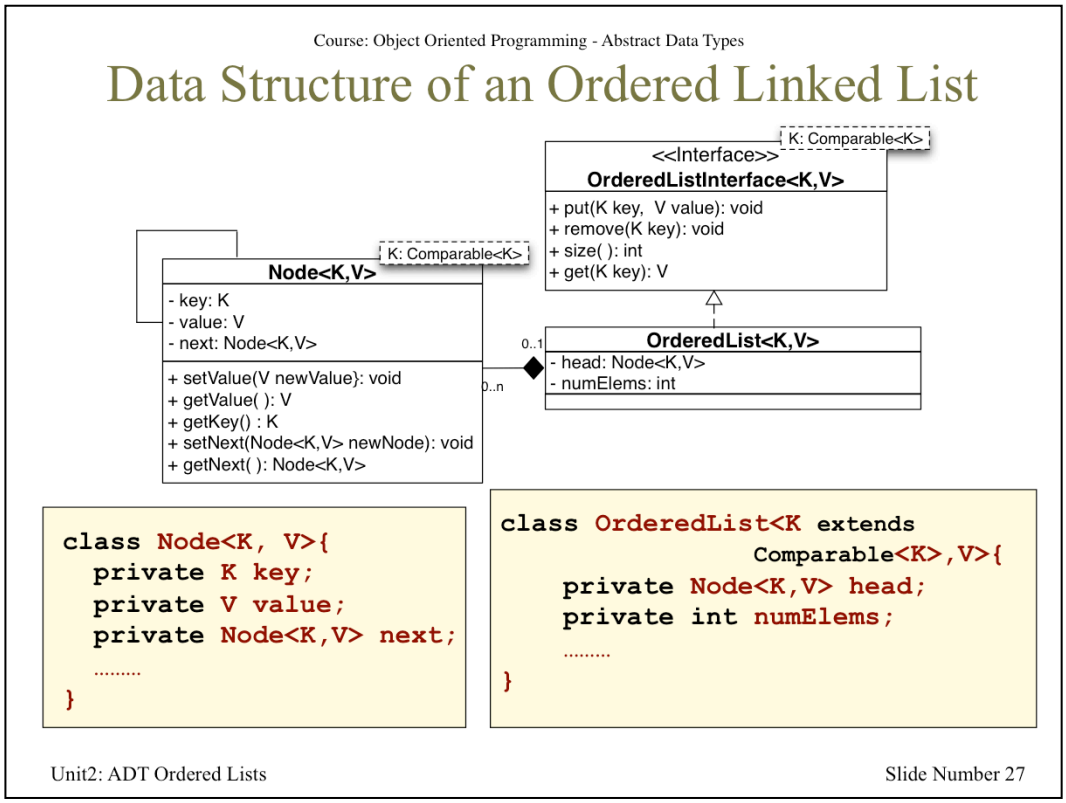
get(searchKey)
//post: Returns the element with key equal
//to searchKey. Returns null if the searchKey
//is not in the list

size( )
// post: Returns the number of elements in
// an ordered list

So far we have seen that access to elements in a list is done by their position. Ordered lists are special types of lists where the access to elements in the list is done via a search key. Elements in an ordered list include then a key and a value. We assume here that the key is unique for each element in the list.

An ordered list is therefore a liner sequence of elements sorted in ascending or dissenting order with respect to their key values. The main access procedures of getting, adding and removing elements from a list are different for an ordered list, whereas the access procedures size() and isEmpty( ) are the same as that of an (not sorted) list.

This slides gives the specification of the access procedures for an ordered linked list. Note that the operation for getting an element from an ordered list takes as input the key value of the element. Similarly for the access procedure “remove”. The “add” access procedure, instead,
takes both the key and the value of an element to insert. In alternative implementations, the remove access procedure may return null instead of throwing an exception.

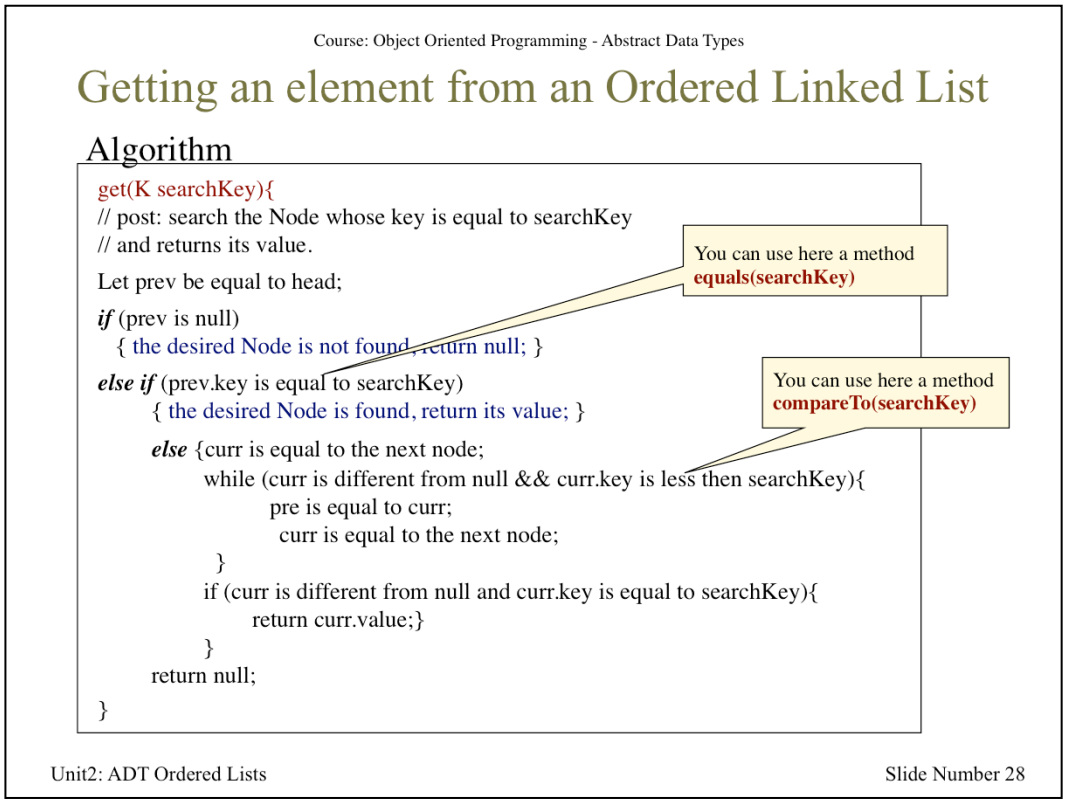


The implementation of an ordered linked list needs a more complex Node data structure. This has to include two generic type parameters. The first type parameter is for the key of an element in the list and the second type parameter is for the value of the element in the list. Note that the data type of V can itself be another complex data structure, for instance another linked list as well as any other user-defined type.

Note that the data type K, for the search key, has to be a class that implements the interface Comparable. In the declaration of the class OrderedList, `Comparable<K>` is the Java.lang generic interface that includes just one single method:

```
package java.lang;
public interface Comparable<T>{
    public int compareTo(T other);
}//end Comparable
```

So in the declaration of the class OrderedList, the generic type "K extends Comparable<K>" means that the generic type K can be bound to any data type that implements the interface Comparable. For instance, if K is a String, this would be OK because the Java type String implements Comparable. If K is any user-defined class C, this would have to be defined as to implement Comparable, i.e. it would have to include a method compareTo(C other).



This slide shows a pseudo code algorithm for the method `get(K searchKey)` of an ordered linked list. Note that to implement this method, and the other access procedures `put(K searchKey, V givenValue)` and `remove(K searchKey)`, it is important to be able to compare the key of a Node with the `searchKey`.

The method `compareTo(E other)`, defined in the Java.lang interface `Comparable<E>`, returns an integer. Consider the method call **`s.compareTo(t)`**. This returns a negative integer if s comes before t, it returns 0 if s and t are equal and it returns a positive integer if s comes after t. If you implement your own `compareTo(E other)` or a K data type that you have defined, you must make sure to define the ordering according to this specification.

Inserting an item in an Ordered Linked List

Algorithm

```
put(K searchKey, V givenValue){
// post: add a new Node with searchKey and givenValue, to the ordered linked list
// so that the order is satisfied. If a Node with givenKey already exists, its value is
// replaced with givenValue.
<< Find node prev>>
if (prev is null)
  { Add node at the head of the list; }
else if (prev.key is equal to searchKey)
      { set prev.value to be the givenValue; }
      else if (prev.key is less then searchKey)
            { add a new node with givenKey and givenValue after prev;}
        else {add a new node with givenKey and givenValue as first node;}
}
```


This slide shows the basic algorithm for the method `put(searchKey, givenValue)` of an ordered linked list. This method uses the auxiliary procedure "`findPrev`", defined in the next slide, for locating the Node in the list with key just before the `searchKey`.

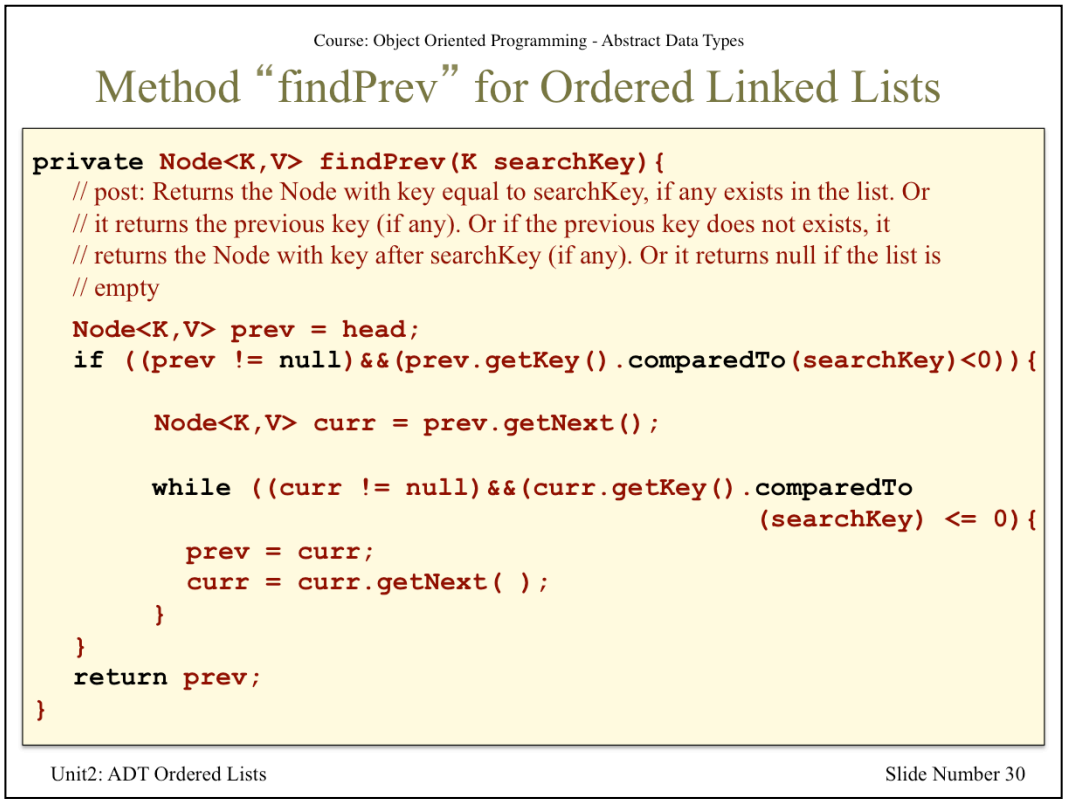
There are different cases to consider, which depend upon the value of the variable "`prev`" returned by the `findPrev` method call.

The variable "`prev`" can be null. This is when the list is empty, in which case the given element can be added at the head of the list.

The variable "`prev`" is not null but references a node with key equal to `searchKey`. In this case we just change the value to be equal to `givenValue`.

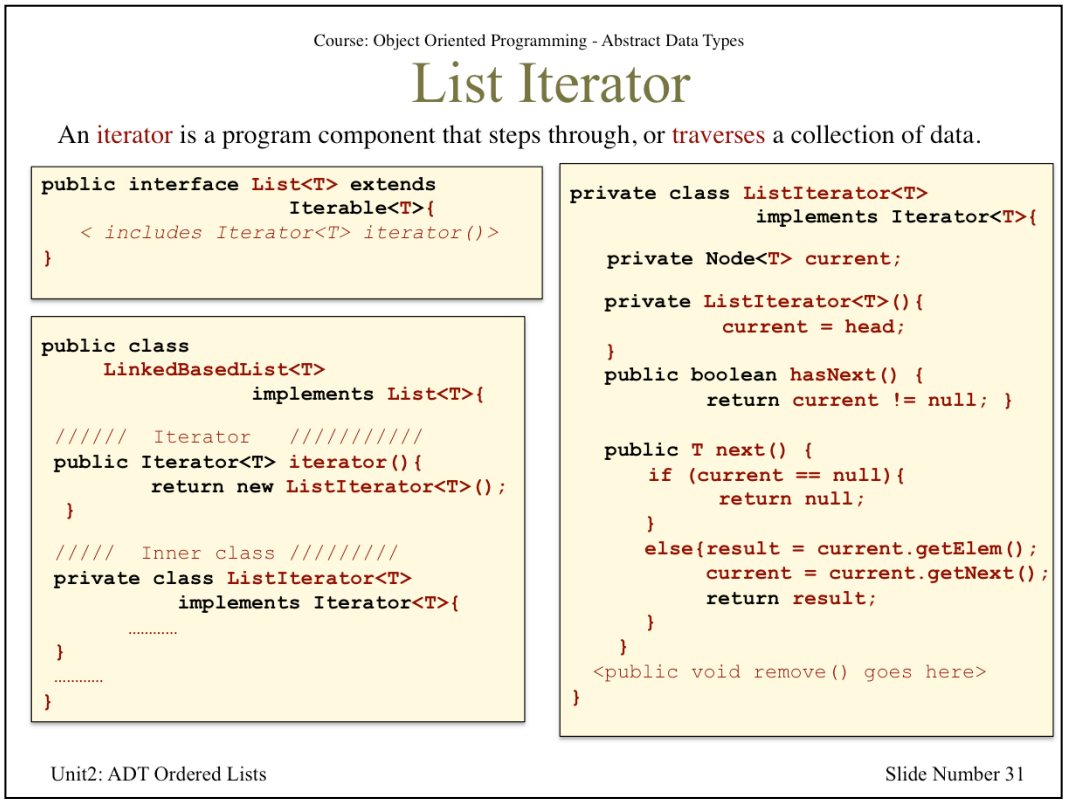
The variable "`prev`" is not null but references a node with key smaller than `searchKey`. In this case, `prev` is referring to the last Node in the list. The given element should then be added just after the node referred by `prev`, i.e. at the end of the list.

The variable "`prev`" is not null and reference a Node with key bigger than `searchKey`. In this case all the elements in the list have key bigger than `searchKey` and therefore the element should be added at the front of the list.



In this slide I have given the implementation of the auxiliary method `findPrev`. The cases covered by this method are the following:

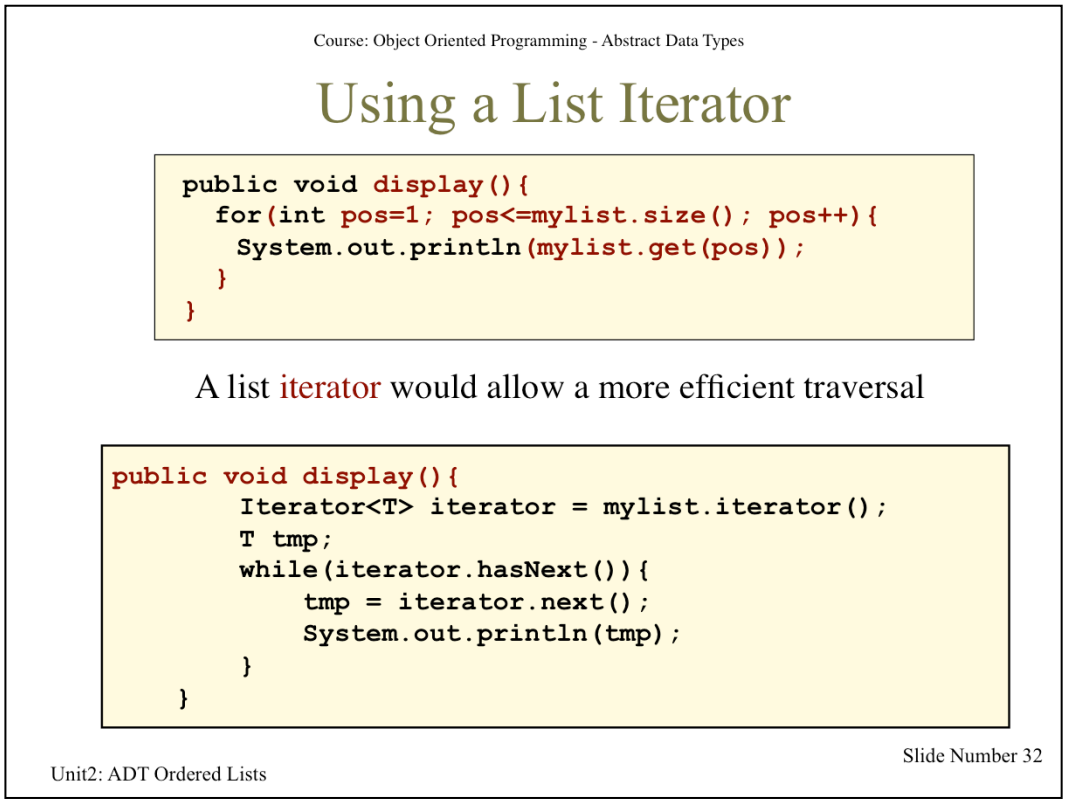
1) The linked list is empty. Then the method returns null.
2) The `searchKey` is smaller then all the keys in the linked list. In this case `prev` refers to the first element in the linked list.
3) The `searchKey` is bigger then all the keys in the linked list. In this case `prev` refers to the last element in the linked list.
4) The `searchKey` is within the range of keys stored in the linked list. In this case, the method returns the reference to the Node whose key field is the immediate predecessor of the searchKey in the ordered list, if the `searchKey` does not belong to the list; otherwise it returns the node with key equal to the `searchKey` .



This slide provides an example implementation of a list Iterator (for linked lists). A list iterator facilitates a more efficient traversal of a linked list.

There are different ways of implementing a list iterator. Here we have made use of a inner class iterator, called `ListIterator`. We have first defined the list interface to extend the standard Java.lang interface "`Iterable`". This is a generic interface. The class `LinkedBasedList` has then to include the implementation of the method `iterator()`, which basically creates and returns a new object `Iterator`. This is the object that is used by the client program to scan the linked list, as shown in the next slide. The methods `hasNext()` of the iterator object checks that there is a next element in the list and the method `next()` returns the next element. Note that in this case we have declared the return type of `next()` to be `T`.

To complete this example implementation, you have just to make sure that the iterator methods `hasNext()` and `next()` are implemented in the inner class `ListIterator` (as shown in the right-hand-side of this slide.



Let's consider, for example, a client method that displays the elements of a linked list of String. The method can only use access procedures for getting the elements in the list. The top box shows an example implementation of such method that makes use of a standard loop traversal. At each iteration, the call to the access procedure `mylist.get(pos)` will scan the list from the beginning up to the current value of `pos`. This is not always a very efficient way of traversing a list.

A list iterator would be more efficient. The bottom box shows how you would implement the display method using a list iterator.

Summary

Lists may be implemented statically or dynamically:

How do they compare?

Time efficiency expressed in Big Oh notation

Operations	Fixed-size Array	Linked List
add(pos, elem)	O(n) to O(1)	O(1) to O(n)
remove(pos)	O(n) to O(1)	O(1) to O(n)
get(pos)	O(1)	O(1) to O(n)
display()	O(n)	O(n)
size(), isEmpty()	O(1)	O(1)

We have seen in the last two lectures two different approaches for implementing an ADT list, the first is static and uses (fixed-size) array, the second is dynamic and uses linked lists. The natural question that would come into mind is which of these two approaches we should use. The general answer is that it depends really on the type of use that the client program makes of the list. To understand the pro and cons a summary table is given here of the time efficiency of the access procedures that we have seen so far, for each of the two types of implementation. This evaluation is given using the Big Oh notation, that you can read as "order of at most…."

Let's consider the static implementation. The time efficiency of the operation "add" and "remove" depends on the position. They take less time as the position increases. This is because we have to make less operations for shifting elements to the right or to the left respectively. On the other hand, the operation "get" has a constant time efficiency since you can access to an element in the list by using directly the array index.

If we consider the dynamic implementation, the operations "add" and "remove" take more time as the position increases, this is because you need to scan more elements to get to entries towards the end of the list. The operation "get" also takes more time as the position increases, for the same reason.

So, if the application uses more frequently the "get" operation on a "stable" list of elements, the static implementation would be better. If, on the other hand, the list of elements tends to grow often and maybe uses addition and deletion at or near the beginning of the list then the dynamic implementation would be better.

In summary, the operation that your application program uses most should influence your choice of implementation of the ADT list.