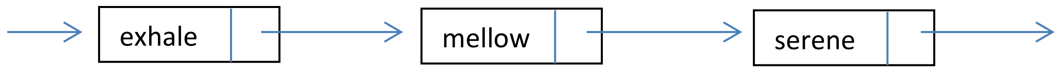
Chapter 7: Linked Lists

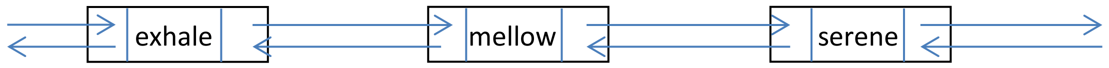
- In this chapter we will continue our study of the collection classes by introducing LinkedList class, part of Java collection Framework.
- LinkedList class also implements List interface, which means that most of the method headings should be already be familiar to you. Heading of these methods are the same but the definitions are quite different.
- There are some significant performance differences between ArrayList and LinkedList class.
- LinkedList class lacks the random access feature of ArrayList object.
- To access a LinkedList's element from an index it requires a loop. However, LinkedList objects allow constant time for insertion and deletion where ArrayList worstTime for insertion and deletion is linear in n.

7.1 **What is a LinkedList**

- SinglyLinkedList class is toy class which servers mainly to prepare you for the more powerful LinkedList class.
- A linked list is List object (that is, an object in a class that implements the List interface) in which the following property is satisfied:
  - Each element is contained in an object, called **Entry** object, which also includes a reference, called a **link**, to the **Entry** object that contains the next element in the list.

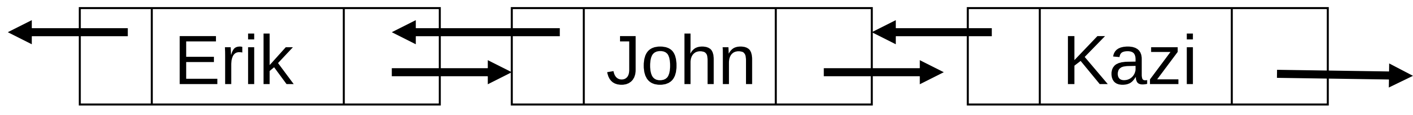


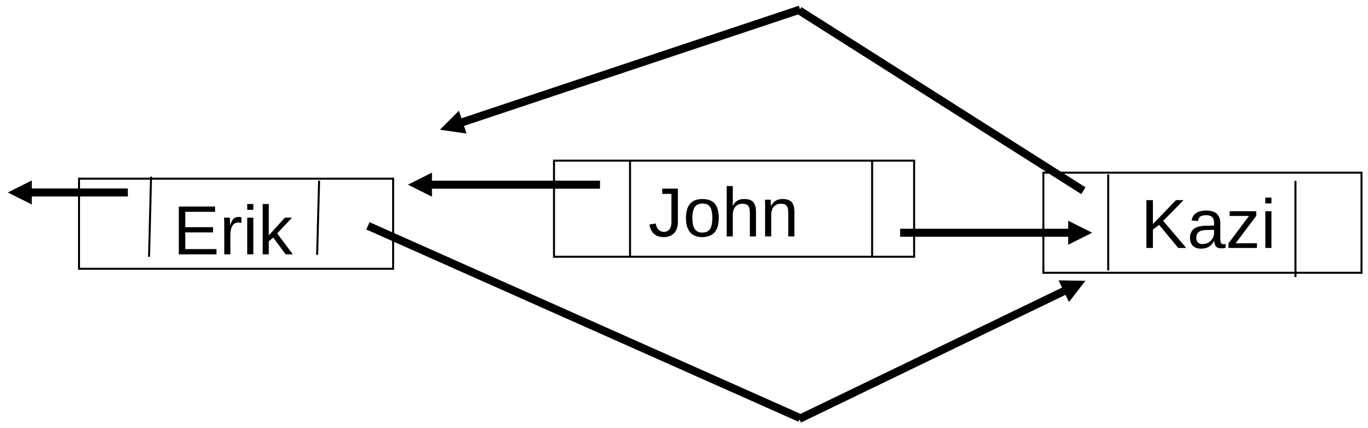
- For entry object that holds last element, there is no next element.
- Some Entry objects also satisfy the following property:
  - Each **Entry** object includes a link to the **Entry** object that contains the previous element in the list.



- Lined list that satisfies second property is called **doubly-linked list**.
- The beauty of a linked list is that insertions and removals can be made without moving any elements: Only the links are altered.

For example, to remove the Entry **with "John":**



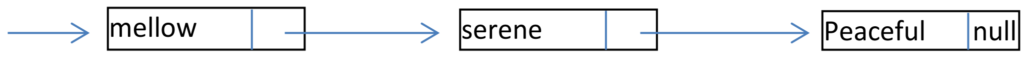


7.2 **The SinglyLinkedList Class - A singly – linked, toy Class**

- SinglyLinkedList class has very little functionality, and is not part of the Java Collection Framework. You will never use it for application programs.
- However, this class should be viewed as the toy class that highlights the concepts of links and iterators, two essential features of the Java Collection Framework.
- LinkedList class, doubly-linked, is quite powerful but also somewhat more complex.
- The elements in a SinglyLinkedList object are not stored contiguously, so with each element we must provide information on how to get to the next element in the collection.
- First we create class to hold a reference to an element and a **next** reference.

```
protected class Entry<E>
{
        protected E element;
        protected Entry<E> next;
}
```

- In this Entry class there are no specified methods and two fields, with parameter E the type parameter.
- The **next** field in an **Entry** holds the reference to another Entry object. A reference to an Entry object is called a link.
- We use arrow to indicate that the next field at the base of the arrow contains a reference to the Entry object pointed to by the tip of the arrow. Type of the element is String rather than reference to String.



- In the last Entry object the next field has the value null, which indicates that there is no subsequent Entry object.

public class SinglyLinkedList <E> extends AbstractCollection<E> implements List<E>

Method specifications page 269.

7.2.1 **Fields and Method Definitions in the SinglyLinkedList Class**

- In the previous example, we are missing a link to a first Entry object.

  protected Entry <E> head;

  Suppose we construct a SinglyLinkedList object as follows:

  SinglyLinkedList<Integer> scoreList = new   SinglyLinkedList<Integer>();

```
public SinglyLinkedList()
{
        head = null;
}
```

- Reference field is automatically initialized to null.

- To test whether the collection is empty we use isEmpty() method. This method tests the head field to check if there is element in the collection.

```
public boolean isEmpty()
{
        return head == null;
}
```

One important point of the SinglyLinkedList class is that a linked structure for storing a collection of elements is different from an array or ArrayList object in two key respects:

1. The sizes of the collection need not to be known in advance. We simply add elements at will. Therefore, we do not have to worry about, as we do with an array, about allocation too much space or too little space. But it should be noted that in each Entry object, the next field consumes extra space; it contains program information rather than problem information.
2. Random access is not available. To access some elements, we would have to start by accessing the head element, and then accessing the next element after the head element and so on.

- addToFront() method

```
public void addToFront(E element)
{
    Entry <E> newEntry = new Entry<E>();
    newEntry.element = element;
    newEntry.next = head;
    head = newEntry;
}
```

7.2.2 **Iterating through a SinglyLinkedList Object**

- An iterator is an object that enables a user to loop through a collection without accessing the collection's fields.
- We have not yet established a way to loop through the elements in a singlylinkedlist object.
- We can develop an Iterator class for the SingliyLinkedList object that would implement the Iterator interface. That is our class will provide a definition for the methods heading listed in the Iterator interface.
- Our class will provide full definition of default constructor, next(), and hasNext(). The remove method will throw an exception.

Page 277.

- In defining the three methods, we will deal with three next
    - A next field in the SinglyLinkedListIterator class;
    - A next() method in the SinglyLinkedListIterator class;
    - A next field in the Entry class.

1. SinglyLinkedListIterator class has to have a constructor.

```
protected SinglyLinkedListIterator()
{
    next = head;
}
```

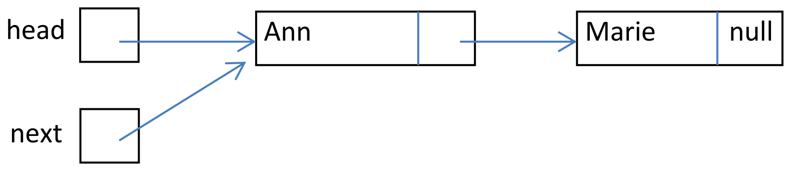
- If we do not have constructor, compiler will generate a default constructor and JVM will initialize next field to null.
  Note: construct of the SinglyLinkedListIterator class can access head because this class will be embedded in the SinglyLinkedList class where head is field.

2. hasNext() method

```
public boolean hasNext()
{
    return next!=null;
}
```

3. remove() method will simply throw an exception

4. next() method



```
public E next()
{
        E theElement = next.element;
        next = next.next;
        return theElement;
}
```

- The **next** field in the SinglyLinkedListIterator object is a reference to an Entry object that has a **next** field.
- From the previous example element returned should be Ann.
- In order to iterate through the SinglyLinkedList object, we will have to associate a SinglyLinkedListIterator object with a SinglyLinkedList object. To accomplish we will use Iterator() method that creates necessary connection.

```
public Iterator<E> iterator()
{
        return new SinglyLinkedListIterator();
}
```

- Value returned is reference to a SinglyLinkedListIterator.
- Now we can create the appropriate iterator. Such as,

```
SinglyLinkedList<Boolean> myLinked = new SinglyLinkedList<Boolean>();
Iterator<Boolean> itr = myLinked.iterator();
```

  Now to printout each element we can just simply,

```
while (itr.hasNext())
        System.out.println(itr.next());
```

7.3 **Doubly – Linked Lists**

- For the most part, the LinkedList class has the same method headings as the ArrayList class, but those classes have different time estimates for some methods.
  For example,

  public E get (int index)

  public E set (int index, E element)

  worstTime(n) is linear in n

  versus constant for an ArrayList.

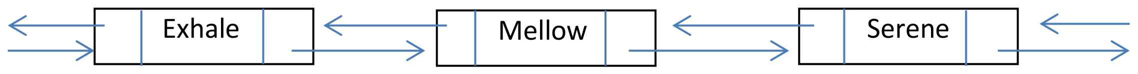
- Sometimes LinkedList versions are faster:

  public boolean add (E element)

  The worstTime(n) is constant, versus linear in n for an ArrayList object because of the possibility of re-sizing.

- Basically, to get to a position in a LinkedList takes linear-in-n time, but once you get there, you can remove or insert in constant time.
- That emphasizes the importance of iterators, because once an iterator is positioned somewhere in the collection, you can insert or remove in constant time.

Suppose we have,

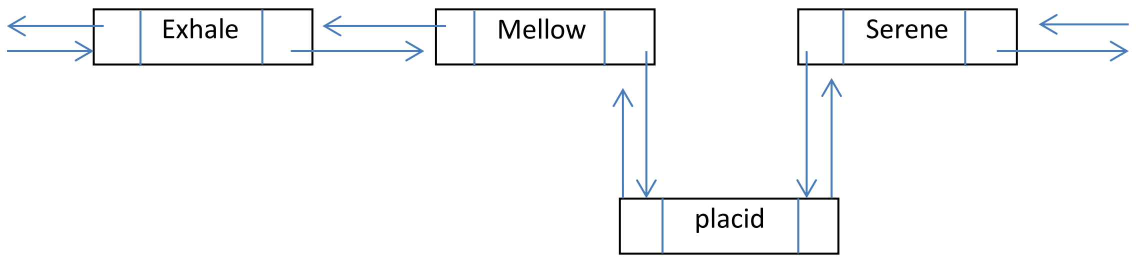


If we wish to insert "placid" in front "serene".

We will first have to construct a new Entry object, which would store "placid" as its element, and then we have to four links.

- The previous and next link for "placid"
- The next link for of "mellow"
- And previous link of "serene"

Note: Insertion of a new element will be in a constant time. That is, worstTime(n) is constant.



7.3.1 A User's view of the LinkedList Class

- Since LinkedList class implement the List interface, LinkedList objects support a variety of index-based methods such as get and indexOf.
- The indexes always start at 0, thus a LinkedList object with three elements has its first element at index 0 its second element at index 1, and its third element at index 2.

7.3.2 The LinkedList Class versus The ArrayList Class

- LinkedList class does not have a constructor with initial capacity parameter because LinkedList object grow and shrink as needed.
- And the related methods ensureCapacity and trimToSize are not needed for the LinkedList class.
- However, LinkedList class has some methods, such as removeFirst() and getLast(), that are not in the ArrayList class.
- worstTime(n) is constant for these methods, where n represents the number of elements in the calling object.

Page 283 – method headings for the public methods in the LinkedList class.

7.3.3 LinkedList Iterators

- In the LinkedList class, the iterators are bi-directional: they can move either way (to the next element) or backwards (to the previous element)
- The name of the class that defines the iterators is ListItr. The ListItr class – which implements the ListIterator interface is embedded as a private class in the LinkedList class.

Methods in the embedded ListItr class, which implements ListIterator:

IT1. public void add (E element)
IT2. public boolean hasNext( )
IT3. public boolean hasPrevious( )
IT4. public E next( )
IT5. public int nextIndex( )
IT6. public E previous( )
IT7. public int previousIndex( )
IT8. public void remove( )
IT9. public void set (E element)

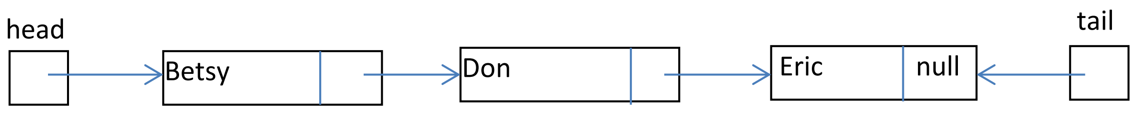
For each of these methods, worstTime(n) is constant!

7.3.5 Fields and Heading of the LinkedList Class

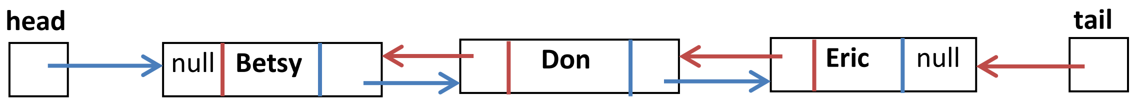
- For the implementation of the LinkedList class, the primary decision is to what the fields will be.
- First, we can consider using the SinglyLinkedList class. However, the problem with implementation of this class is that the upper boundary (Big-O) of worstTime(n) for some methods may not satisfy performance specifications.

- SinglyLinkedList class has one field:
  protected Entry<E> head;
- And embedded Entry class has two fields:
  protected E element;
  protected Entry<E> next;

- Adding the element at the end of the list will require linear in n time.
- To get around this problem we can add another field **tail**, which would hold a reference to the last entry in the SinglyLinkedList object. This will help us defining the addLast() method.
- However, implementing the removeLast() method would be much more challenging.



- For this we will have to change the reference stored in **next** field of the Entry object preceding the Entry object referenced by **tail**. For this task we need a loop which would give us a worstTime(n) linear in n.
- Idea of having head and tail fields can be used in the DoublyLinkedList class.
- The nested Entry class will support a doubly-linked list by having three fields:
  protected E element;
  protected Entry<E> previous, next;



- Java Collection Framwork's implementation of the LinkedList class is doubly-linked, but does not have head and tail fields. Instead, there is a header field, which contains reference to a special Entry object, called a "dummy entry" or "dummy node".

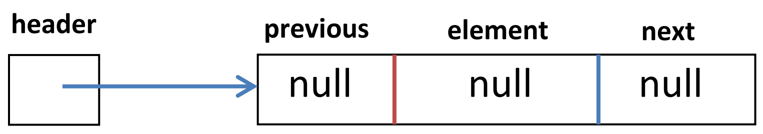
```
public class LinkedList<E> extends AbstractSeqentialList <E> implements List<E>, Queue <E>,
java.lang.Cloneable, java.io.Seralizable
{
        private transient int size = 0;
        private transient Entry <E> header = new Entry<E>(null, null, null);
}
```

- Size field keeps track of the number of elements in the calling LinkedList object.
- Transient modifier merely indicates that field is not saved if the elements in a LinkedList object are serialized, that is saved to an output stream.
- Entry class has three fields, one for an element and two for links.
- Only method in Entry class is constructor that initializes the three fields.

```
private static class Entry<E>
{
        E element;
        Entry<E> next;
        Entry<E> previous;

        Entry (E element, Entry<E> next, Entry<E> previous)
        {
                this.element = element;
                this.next = next;
                this.previous=previous;
        }
}
```

- **Element** field will hold a reference to the Entry object's element
- **Next** will contain a reference to the Entry one position further in the LinkedList object
- **Previous** will contain a reference to the Entry one position earlier in the LinkedList object.



- Header field always points to the same dummy entry, and entry's dummy element field always contains null.
- Next field will point to the Entry object that houses the first element in the LinkedList object, and the previous field will point to the Entry object that houses the last element in the LinkedList object.
- Since LinkedList object uses dummy entry and not head and tail fields, this ensures that every Entry object in a linked list will have both a previous Entry object and a next Entry object.

User’s guide for choosing ArrayList or LinkedList:

- If the application entails a lot of accessing and/or modifying elements at widely varying indexes, an ArrayList will be much faster than a LinkedList.

- If a large part of the application consists of iterating through a list and making insertions and/or removals during the iterations, a LinkedList will be much faster than an ArrayList.

7.3.6 Creating and Maintaning a LinkedList object

- The significance of the header entry is that there is always an entry in back of and in front of any entry.
- That simplifies insertions and removals, as well as iterations from the back of the list to the front.

7.3.7 Definition of the two-parameter add Method

```
private void add (int index, E element)
{
        if(index == size)
                addBefore(element, header);
        else
                addBefore(element, entry(index));
}

private Entry<E> addBefore (E element, Entry<E> e)
{

}
```