IT 179
11
LinkedList Class

For Next Lecture

Read Chapter 3

Testing & Debugging

LinkedList Class

- Doubly-linked list implementation of the List interface.

LinkedList Example


```
// Import the LinkedList class
import java.util.LinkedList;

public class Main {
  public static void main(String[] args) {
    LinkedList<String> cars = new LinkedList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    System.out.println(cars);
  }
}
```

```
[Volvo, BMW, Ford, Mazda]
```

Table 2.6 Methods of Class `LinkedList<E>`

Method	Behavior
`public void add(int index, E obj)`	Inserts object `obj` into the list at position `index`.
`public void addFirst(E obj)`	Inserts object `obj` as the first element of the list.
`public void addLast(E obj)`	Adds object `obj` to the end of the list.
`public E get(int index)`	Returns the item at position `index`.
`public E getFirst()`	Gets the first element in the list. Throws `NoSuchElementException` if the list is empty.
`public E getLast()`	Gets the last element in the list. Throws `NoSuchElementException` if the list is empty.
`public boolean remove(E obj)`	Removes the first occurrence of object `obj` from the list. Returns `true` if the list contained object `obj`; otherwise, returns `false`.
`public int size()`	Returns the number of objects contained in the list.

The Iterator

- An iterator can be viewed as a moving place marker that keeps track of the current position in a particular linked list
- An `Iterator` object for a list starts at the list head
- The programmer can move the `Iterator` by calling its `next` method.
- The `Iterator` stays on its current list item until it is needed
- An `Iterator` traverses in O(n) while a list traverse using get() calls in a linked list is O(n^2)

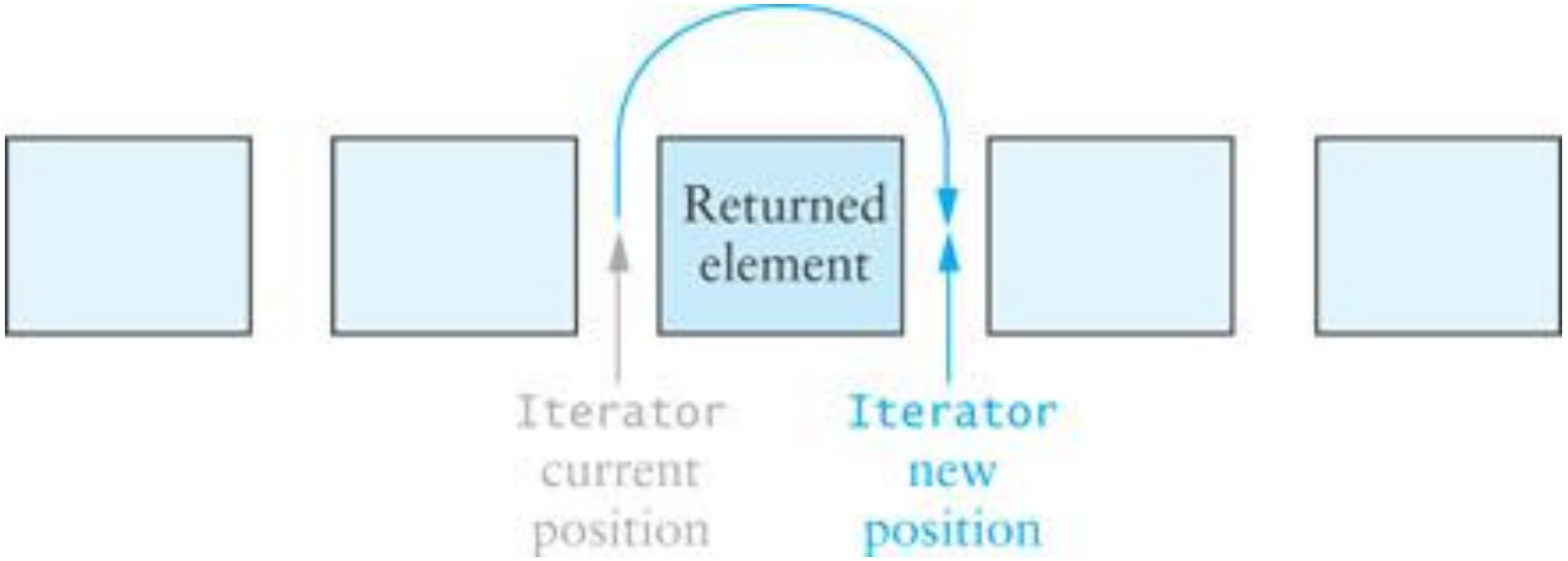
`Iterator` Interface

- The `Iterator` interface is defined in `java.util`
- The `List` interface declares the method `iterator` that returns an `Iterator` object that iterates over the elements of that list

Method	Behavior
`boolean hasNext()`	Returns `true` if the `next` method returns a value.
`E next()`	Returns the next element. If there are no more elements, throws the `NoSuchElementException`.
`void remove()`	Removes the last element returned by the `next` method.

`Iterator` **Interface** (cont.)

- An `Iterator` is conceptually *between* elements; it does not refer to a particular object at any given time



`Iterator` **Interface** (cont.)

- In the following loop, we process all items in `List<Integer>` through an `Iterator`

```
Iterator<Integer> iter = lst.iterator();
while (iter.hasNext()) {
  int value = itr.next();
  // Do something with value
  ...
}
```

Example (Demo)


```
public static void main(String[] args)
    {

        LinkedList<String> cars = new LinkedList<String>();
        cars.add("Volvo");
        cars.add("BMW");
        cars.add("Ford");
        cars.add("Mazda");
        System.out.println(cars);

        // Setting the ListIterator at a specified position
        Iterator list_Iter = cars.iterator();

        // Iterating through the created list from the position
        System.out.println("The list is as follows:");
        while (list_Iter.hasNext())
        {
            System.out.println(list_Iter.next());
        }

    }
```

Iterators and Removing Elements

- You can use the `Iterator remove()` method to remove items from a list as you access them
- `remove` deletes the most recent element returned
- You must call `next()` before each `remove()`; otherwise, an `IllegalStateException` will be thrown
- `LinkedList.remove` vs. `Iterator.remove`:
  - `LinkedList.remove` must walk down the list each time, then remove, so in general it is O(*n*)
  - `Iterator.remove` removes items without starting over at the beginning, so in general it is O(1)

Iterators and Removing Elements (cont.)

- To remove all elements from a list of type `Integer` that are divisible by a particular value:

```
public static void removeDivisibleBy(LinkedList<Integer>
                                                aList, int div) {
  Iterator<Integer> iter = aList.iterator();
  while (iter.hasNext()) {
    int nextInt = iterNext();
    if (nextInt % div) {
      iter.remove();
    }
  }
}
```

`ListIterator` Interface

- `Iterator` limitations
  - Traverses `List` only in the forward direction
  - Provides `remove` method, but no `add` method
  - You must advance the `Iterator` using your own loop if you do not start from the beginning of the list
- `ListIterator` extends `Iterator`, overcoming these limitations

```
public interface ListIterator<E> extends Iterator<E>
```

`ListIterator` **Interface** (cont.)

- As with `Iterator`, `ListIterator` is conceptually positioned between elements of the list
- `ListIterator` positions are assigned an index from 0 to `size`

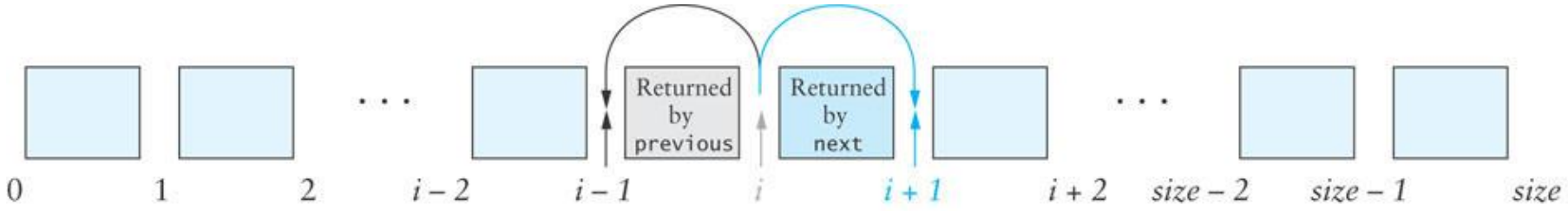


Table 2.8 `ListIterator<E>` Interface

Method	Behavior
`void add(E obj)`	Inserts object `obj` into the list just before the item that would be returned by the next call to method `next` and after the item that would have been returned by method `previous`. If method `previous` is called after `add`, the newly inserted object will be returned.
`boolean hasNext()`	Returns `true` if `next` will not throw an exception.
`boolean hasPrevious()`	Returns `true` if `previous` will not throw an exception.
`E next()`	Returns the next object and moves the iterator forward. If the iterator is at the end, the `NoSuchElementException` is thrown.
`int nextIndex()`	Returns the index of the item that will be returned by the next call to `next`. If the iterator is at the end, the list size is returned.
`E previous()`	Returns the previous object and moves the iterator backward. If the iterator is at the beginning of the list, the `NoSuchElementExcepton` is thrown.
`int previousIndex()`	Returns the index of the item that will be returned by the next call to `previous`. If the iterator is at the beginning of the list, −1 is returned.
`void remove()`	Removes the last item returned from a call to `next` or `previous`. If a call to `remove` is not preceded by a call to `next` or `previous`, the `IllegalStateException` is thrown.
`void set(E obj)`	Replaces the last item returned from a call to `next` or `previous` with `obj`. If a call to `set` is not preceded by a call to `next` or `previous`, the `IllegalStateException` is thrown.

Table 2.9 Methods that return ListIterators

Method	Behavior
`public ListIterator<E> listIterator()`	Returns a `ListIterator` that begins just before the first list element.
`public ListIterator<E> listIterator(int index)`	Returns a `ListIterator` that begins just before position `index`.

Example (Demo)


```
public static void main(String[] args)
    {

        LinkedList<String> cars = new LinkedList<String>();
        cars.add("Volvo");
        cars.add("BMW");
        cars.add("Ford");
        cars.add("Mazda");
        System.out.println(cars);

        // Setting the ListIterator at a specified position
        ListIterator<String> list_Iter = cars.listIterator(cars.size());

        // Iterating through the created list from the position
        System.out.println("The list is as follows (Using ListIterator):");
        while (list_Iter.hasPrevious())
        {
            System.out.println(list_Iter.previous());
        }

    }
```

Enhanced `for` Statement

- The for each statement is also called an enhanced `for` statement
- The enhanced `for` statement creates an `Iterator` object and implicitly calls its `hasNext` and `next` methods
- Other `Iterator` methods, such as `remove`, are not available

Enhanced `for` Statement (cont.)

- The following code counts the number of times `target` occurs in `myList` (type `LinkedList<String>`)

```
count = 0;
for (String nextStr : myList) {
  if (target.equals(nextStr)) {
    count++;
  }
}
```

Enhanced `for` Statement (cont.)

- In list `myList` of type `LinkedList<Integer>`, each `Integer` object is automatically unboxed:

```
sum = 0;
for (int nextInt : myList) {
  sum += nextInt;
}
```

Enhanced for Statement - Demo


```
public static void main(String[] args)
    {
        LinkedList<Integer> nums = new LinkedList<Integer>();
        for (int i = 1; i < 10; i++)
        {
            nums.add(i);
        }

        System.out.println("The list is now: " + nums);

        int sum = 0;
        for (int nextInt : nums)
        {
            sum += nextInt;
        }
        System.out.println("The sum of the first 10 integers is: " + sum);

    }
```

Enhanced `for` Statement (cont.)

- The enhanced `for` statement can also be used with arrays, in this case, `chars` or type `char[]`

```
for (char nextCh : chars) {
  System.out.println(nextCh);
}
```

Example


```
char[] chars = { 'a', 'b', 'c', 'd', 'e', 'f' };
        for (char nextCh : chars)
        {
            System.out.println(nextCh);
        }
```

Application of the `LinkedList` Class

Section 2.8

An Application: Ordered Lists

- We want to maintain a list of names in alphabetical order at all times
- **Approach**
  - Develop an `OrderedList` class (which can be used for other applications)
  - Use a `LinkedList` class as a component of the `OrderedList` (if `OrderedList` extended `LinkedList`, the user could use `LinkedList`'s `add` methods to add an element out of order)