Chapter 12
Iterators

The Iterator Concept

- Many array algorithms are designed around a sequential scan of the list using an index.
- The ArrayList and LinkedList classes implement the List interface and so have get() and set() methods which access and update an element by index.

The Iterator Concept (2)

- Even though a LinkedList collection has get() and set() methods, the methods have $O(n)$ efficiency and are not suitable for index-based applications.
- The Bag class does not provide any means for sequencing through its data.
- The design of a collection type must address the problem of scanning its elements. There are three guidelines that should apply.

The Iterator Concept (3)

- Guideline 1: A collection must define an efficient scanning mechanism consistent with good object-design principles. Requiring the implementation to make the data public is not a solution. The mechanism should follow a design pattern so that general scanning algorithms apply to all collection types.

The Iterator Concept (4)

- Guideline 2: A collection should provide an object, called an iterator, that has methods which allow a scan of a collection similar to the scan of an array with an index. Methods should allow for a loop to sequentially scan the elements.

The Iterator Concept (end)

- Guideline 3: Each collection class understands its underlying structures and is thus responsible for implementing iterator methods.
Collection Iterators

- An iterator is an object that accesses the elements in a collection.
- At any point in the scan, the iterator can access the value of the corresponding element.

Collection Iterators (2)

- The figure illustrates an iterator for a LinkedList object. In (a) the iterator, called iter, references the first element in the list with Integer value 8. In (b), iter has moved forward one position and references Integer value 17.

Collection Iterators (3)

- The generic Iterator interface defines the methods that are available to an iterator.
- Every data structure that implements the Collection interface also implements the Iterator interface.
- Creating an iterator is done by the method iterator() which is specified in the Collection interface. The method returns an Iterator object that references the first element in the collection.

Collection Iterators (end)

- Create an iterator by first declaring an Iterator reference variable.
  ```java
  // declare LinkedList for integer objects
  LinkedList<Integer> aList;
  // declare Iterator for Integer objects
  Iterator<Integer> iter;
  // create iterator and assign to iter
  iter = aList.iterator();
  ```

Factory Methods

- A factory method is a name for a non-constructor method that creates objects. The method iterator() of a Collection object is a factory method. You use it to create an iterator, not a constructor.
  ```java
  Iterator<String> iter = aStringList.iterator();
  ```

Iterator Methods

- The Iterator method hasNext() indicates whether more values remain in a collection traversal.
  ```java
  // continue while there are remaining elements
  while (iter.hasNext())
  {
  ...
  }
  ```
Actual movement through the collection is done by the method next(), which returns the value of the next collection element and moves forward one element.

Calling next() when hasNext() is false results in the NoSuchElementException.

By initializing an iterator and applying a while statement that uses hasNext() and next(), the following is a template for scanning a collection.

```java
// initialize iter to reference first element in a collection c
iter = c.iterator();
// loop accesses successive elements to the end of the collection
while (iter.hasNext())
{
    // obtain the next value and move forward
    value = iter.next();
    <act on value>
}
```

To remove an element from the underlying collection after a call to next(), use the remove() method.

```java
interface Iterator<T>  
Methods

boolean hasNext()  
Returns true if the iteration has more elements

T next()  
Returns the next element in the list and moves this iterator forward one position

void remove()  
Removes from the underlying collection the last element returned by a call to the iterator method next()
```

Any collection class that implements the Collection interface must include iterators.

An algorithm that relies on traversing elements in a collection and extracting their values can be implemented as a generic method using an iterator:

```java
public static <T extends Comparable<? super T>>
T max (Collection c)
{
    // create an iterator positioned at the first element
    Iterator<T> iter = c.iterator();
    // assign maxValue the value of the first
    // element and advance iter
    T maxValue = iter.next();, scanObj;
    // scan the rest of the elements in the collection
    while (iter.hasNext())
    {
        scanObj = iter.next();
        if (scanObj.compareTo(maxValue) > 0)
            maxValue = scanObj;
    }
    return maxValue;
}
```
Iteration using the Enhanced for Statement

- When a collection must be accessed without removing elements, a enhanced form of the for statement can be used. It takes care of sequencing through collection elements without having to declare and use an iterator.

Enhanced for Statement

```java
Time24[] appArr = {new Time24(8,30), new Time24(10.00), new Time24(12,30), new Time24(1,45), new Time24(3,15)};
LinkedList<Time24> appList = new LinkedList<Time24>();
// copy elements from array to LinkedList appList
for (int i = 0; i < appArr.length; i++)
appList.add(appArr[i]);
System.out.println("Original appointments: " + appList);
// enhanced for creates an iterator scan of appList
for (Time24 t : appList)
t.addTime(120);
System.out.println("Revised appointments: " + appList);
```

Output:

Original appointments: [8:30, 10:00, 12:30, 1:45, 3:15]
Revised appointments: [10:30, 12:00, 14:30, 3:45, 5:15]

List Iterators

- All List collections have a second type of iterator, called a list iterator that takes advantage of the linear ordering of elements in the collection.
- A list iterator can traverse the list in either direction and also modify the list.
- The methods associated with a list iterator are specified in the generic ListIterator interface which extends the Iterator interface.

List Iterators (2)

- A collection creates a ListIterator object using one of two methods.
- The List method listIterator() returns a ListIterator object that references the first element.
  ```java
  ListIterator<Integer> lIter = aList.listIterator();
  ```
- A second version of listIterator() takes a position as an argument and returns a ListIterator object that references the element at the specified position.
  ```java
  ListIterator<Integer> lIter = aList.listIterator(index);
  ```

List Iterators (3)

- If index == size(), the iterator points just past the end of the list. This is useful if we intend to scan the list in the reverse direction.

```
Position 0 1 2 3 4 5 6 7 8 9 10 11
```

```
Position 0 1 2 3 4 5 6 7 8 9 10 11
```
The ListIterator interface extends the iterator interface and so it defines the methods hasNext(), next(), and remove().

The methods hasPrevious() and previous() are available for a backward scan with actions that parallel hasNext() and next() in a forward scan.

An iterator can only remove an element from a collection. A ListIterator has update methods add() and set() that can add a new element and assign an element to have a new value.

### API for Methods Unique to the ListIterator Interface

<table>
<thead>
<tr>
<th>interface ListIterator&lt;T&gt; extends Iterator&lt;T&gt;</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>void add(T item)</td>
</tr>
<tr>
<td></td>
<td>Insert the specified item into the list at the</td>
</tr>
<tr>
<td></td>
<td>iterator position</td>
</tr>
<tr>
<td></td>
<td>boolean hasPrevious()</td>
</tr>
<tr>
<td></td>
<td>Returns true if this list iterator has more</td>
</tr>
<tr>
<td></td>
<td>elements when traversing the list in the</td>
</tr>
<tr>
<td></td>
<td>reverse (backward) direction.</td>
</tr>
<tr>
<td></td>
<td>T previous()</td>
</tr>
<tr>
<td></td>
<td>Returns the previous element in the list and</td>
</tr>
<tr>
<td></td>
<td>moves this list iterator back one position</td>
</tr>
<tr>
<td></td>
<td>void set(T item)</td>
</tr>
<tr>
<td></td>
<td>Replaces the last element returned by next or</td>
</tr>
<tr>
<td></td>
<td>previous with the specified item</td>
</tr>
</tbody>
</table>

- **ListIterator Set Method**
  - The set() method updates the value of an element during an iteration.
  - The method must be used in tandem with the methods next() (or previous()) since it assigns the new value to the element last returned by one of these extraction methods.
  - First call next() to extract a value, determine what the update should be, and then use set() to assign the new updated value.

### Backward Scan of a List

- Using the methods hasPrevious() and previous(), an iterator can move backward in a list.
- The method hasPrevious() returns true if there are list elements remaining when traversing the list in the reverse direction.
- The method previous() returns the previous element of the list and moves the iterator down one position in the list.
The method reverseOutput() takes a linked list as an argument and uses a list iterator to output the list in reverse order. The scan uses the index version of listIterator() and the methods hasPrevious() and previous().

```java
public static <T> void reverseOutput(LinkedList<T> aList)
{
  // create iterator that starts just past end of the list
  ListIterator<T> revIter = aList.listIterator(aList.size());
  // loop outputs the value of element accessed by previous()
  while (revIter.hasPrevious())
    System.out.print(revIter.previous() + " ");
  System.out.println();
}
```

Use reverseOutput() to output colors in the list colorList.
```java
reverseOutput(colorList);
```

Output:
```
blue  green  black  red
```

The list iterator add() method inserts a new element into the list immediately before the value that would be returned by next() and after the previous list value.

```java
listIter.add(5);
```

After inserting the element, the iterator still references the same element. However, a call to previous() will return the new value.

If the iterator references the first element (hasPrevious() == false), add() inserts a new element at the front of the list.

If the iterator points past the end of the list, then add() inserts a new element at the back of the list.

The method insertList() takes two LinkedList arguments aList and bList along with an integer argument pos specifying a position in aList. The method inserts elements from bList into aList starting at index pos. Two list iterators are used. The first uses listIterator() to create an iterator at the first element in bList.
ListIterator Add Method

```java
public static <T> void insertList(LinkedList<T> aList, LinkedList<T> bList, int pos) {
    // declare iterator and set to start of the list
    ListIterator<T> aIter = aList.listIterator(pos);
    ListIterator<T> bIter = bList.listIterator();
    // scan bList and insert (add) elements into aList;
    // iterator aIter continues to reference the same element
    while (bIter.hasNext())
        aIter.add(bIter.next());
}
```

Assume `stateListA` and `stateListB` are two LinkedList collections containing string abbreviations for US states. The statements insert `stateListB` into `stateListA` at position 2 and then display the updated list.

```java
Assume stateListA and stateListB are two LinkedList collections containing string abbreviations for US states. The statements insert stateListB into stateListA at position 2 and then display the updated list.
```

```
Output: [NY, AL, WI, TN, NV, MT, MA]
```

### The Iterator Design Pattern

- The iterator design pattern provides a means accessing the elements of a collection in order from first to last with no concern for the underlying implementation of the collection.
- This is precisely what `Iterator` provides in a general collection.
- A `ListIterator` provides additional functionality related to the sequential storage of data.

### Ordered Lists

- Many applications require an ordered list. Adding a new element requires scanning existing values and identifying the correct location at which to add the new element.
- Begin by initializing a list iterator to reference the start of the list.
- Scan the list, looking for the first element whose value is ≥ the new item.
- This identifies the insertion location. Use the `add()` method to place the new item in the list.

```java
Assume intList is a linked list containing the Integer values 60, 65, 74, and 82 and curr is the list iterator.
```

```
60  65  74  82
first  last
```

```java
Insert 50 in the list:
- A first call to `next()` extracts the value 60, which is ≥ 50, and the scan terminates with `curr` referencing the second element in the list.
- Value 50 should be added to the front at the element `curr` originally referenced before it advanced to 65. Move the iterator backward with `previous()` and use `add()`.
```
Ordered Lists (4)

- Insert 70 in the list:
  - The scan terminates when next() extracts the value 74 which is the first value that is ≥ 70.
  - As in the first case, use previous() to reset curr and then use add() to insert the element.

```
curr = orderedList.listIterator();
while (curr.hasNext())
    if (item.compareTo(curr.next()) <= 0)
        curr.previous();
        break;
```

orderedList.add(item);

Ordered Lists (end)

- Insert 90 in the list:
  - The scan of the list fails to find an element with greater than 90. The iterator curr has reached the end of the list (curr.hasNext() == false). Insert 90 at the back of the list referenced by the current value of curr.

```
// add item before curr; if curr is at the end of the list adds item
// as the last element of the list
curr.add(item);
```

insertOrder()

```
// insert item into the ordered list
public static <T extends Comparable<? super T>>
void insertOrder(LinkedList<T> orderedList, T item)
{
    ListIterator<T> curr = orderedList.listIterator();
    while (curr.hasNext())
        if (item.compareTo(curr.next()) <= 0)
            curr.previous();
            break;
    curr.add(item);
}
```

insertOrder() (2)

```
// add item before curr; if curr is at the end of the list adds item
// as the last element of the list
curr.add(item);
```

insertOrder() (end)

- If the list has n elements, the worst-case performance occurs when the insertion occurs at the end of the list. This case requires n comparisons and has running time O(n).
  - On the average, we expect to search half the list to find an insertion point. As a result, the average running time is O(n).
  - The best case is O(1), which occurs when the insertion takes place at the front of the list.

Removing Duplicates from Ordered List

- The process involves scanning the list with an iterator and comparing the current value with a target value. The algorithm is presented using an example.
  - The iterator curr initially references the first element in the list. A call to next() extracts the value 5 and moves the iterator. The initial value becomes the target value.
  - target = curr.next();
A second call to next() extracts a duplicate value which is removed from the list. Note that curr now references 7 and the deletion removes the previous element.

Two additional calls to next() extract values that are not duplicates. Each value updates the target (figure (a)-(b)).

A final call to next() extracts a duplicate value and moves curr past the end of the list. The duplicate element is removed and the scan results in an ordered list without duplicate values.

```
// remove duplicate values from the linked list
public static <T> void removeDuplicates(LinkedList<T> aList) {
    // current value and the target
    T currValue, target;
    // list iterator that scans the list
    Iterator<T> curr;
    // start at the front of the list
    curr = aList.iterator();
    // assign target the first list element
    target = curr.next();
    // cycle through the list and remove duplicates
    while(curr.hasNext()) {
        // record the current list value
        currValue = curr.next();
        // if currValue equals target, remove it;
        // otherwise reassign the target to the current value
        if (currValue.equals(target))
            curr.remove();
        else
            target = currValue;
    }
}
```

UML for the LinkedList Class
OrderedList Collection

- Use inheritance to create an ordered list class by extending the LinkedList class.
  - The subclass can use the Collection interface methods in the superclass and can also use the indexed-based remove() and get() methods in LinkedList. These methods do not affect the ordering of the elements.
- List update methods such as add() and set() and ListIterator methods add() and set() can destroy the ordering of the list.

OrderedList Collection (end)

- Let the LinkedList superclass store the elements.
  - Override the add() method that inserts an element at the back of the list using the implementation of the insertOrder() algorithm.
  - Invalidate the inappropriate methods add(index, element), addFirst(), addLast(), and set() by overriding them with code that throws an exception.

OrderedList Class Methods

- The OrderedList class has a constructor that creates an empty list. Its implementation simply calls the default constructor in the superclass.

```java
// constructor creates an empty ordered list
public OrderedList() { super(); }
```

OrderedList Class Methods (2)

- All of the LinkedList methods that could destroy the ordering of elements in a collection are overridden with an implementation that throws an UnsupportedOperationException.

```java
// insert element at an index:
public void add(int index, T item)
{
    throw new UnsupportedOperationException
        ("OrderedList add(index, element): Invalid operation");
}
```

OrderedList Class Methods (3)

- The implementation of add() uses the algorithm for insertOrder() with a final statement that returns the boolean value true indicating that a new element enters the list.

```java
public boolean add(T item)
{
    < code for insertOrder() >
    return true;
}
```

OrderedList Class Example

```java
// create an empty list and a list iterator reference
OrderedList<String> ordList = new OrderedList<String>();
Iterator<String> iter;
ordList.add("green");     // green
ordList.add("blue");      // blue green
ordList.add("red");      // blue green red
ordList.add("black");    // black blue green red
index = ordList.indexOf("red"); // index = 3
ordList.remove(2);        // black blue red
// Initialize iter to reference the first element
iter = ordList.iterator();
// output value from next()
System.out.println(iter.next()); // Output: black
// move the iterator, delete element and output list
iter.next();
iter.remove(); // delete blue
System.out.println(ordList);  // Output: [black, red]
```
Application: Word Frequencies

A document is input from a text file and output displays the distinct words and their frequencies in alphabetical order. The application uses the class WordFreq whose instances store a word and the number of times the word has occurred (the word frequency).

When a word is first encountered in the document, the WordFreq constructor creates an object with the word and a frequency of 1.

For each subsequent occurrence of the word, use the method increment() which increments the frequency field in the object.

Word Frequencies (2)

In order that its objects can be stored in an OrderedList collection, the WordFreq class implements equals() and compareTo(). These methods use the word field to compare objects.

When a word is first encountered in the document, the WordFreq constructor creates an object with the word and a frequency of 1.

For each subsequent occurrence of the word, use the method increment() which increments the frequency field in the object.

Word Frequencies (3)

Declare the OrderedList collection wordList and a Scanner that is linked to a user designated text file.

A while loop processes each word from the file.

- After reading a word, use the constructor to create a WordFreq object wf with the word as its argument.
- The method search() looks for a target in an OrderedList and returns an iterator referencing the value or null if the target is not in the list.

Word Frequencies (end)

In the figure, a search for “pickled” uses the WordFreq object wf = <“pickled”, 1>. The return value is an iterator denoted by iter. The frequency for the object <“pickled”, 3> is incremented so the frequency is 4.

search()

```java
public static <T extends Comparable<? super T>> ListIterator<T> search(OrderedList<T> ordList, T target) {
    // initialize a list iterator
    ListIterator<T> iter = ordList.listIterator();
    T curr;

    // move through the ordered list
    while (iter.hasNext()) {
        // get the current list value
        curr = iter.next();

        // see if current value matches target
        if (curr.equals(target)) {
            // match; move iterator back to the match
            // and return its value
            iter.previous();
            return iter;
        }

        // if the target is less than current
        // value, we won’t find the target in
        // the ordered list
        else if (target.compareTo(curr) < 0)
            return null;

        // the target is larger than any value in the list
        return null;
    }
}
```

search() (end)

// see if current value matches target
if (curr.equals(target)) {
    // match; move iterator back to the match
    // and return its value
    iter.previous();
    return iter;
}

// if the target is less than current
// value, we won’t find the target in
// the ordered list
else if (target.compareTo(curr) < 0)
    return null;

// the target is larger than any value in the list
return null;
import ds.util.OrderedList;
import ds.util.ListIterator;
import java.util.Scanner;   // for file input
import java.io.*;
public class Program12_1
{
public static void main(String[] args)
throws IOException
{
// words read from file and inserted
// into wordList
OrderedList<WordFreq> wordList =
    new OrderedList<WordFreq>();
// scanner to parse words in the file
Scanner fileIn;

// input words to end-of-file
while (fileIn.hasNext())
{
    word = fileIn.next();
    // create a wordFreq object with frequency 1
    wf = new WordFreq(word);
    // search to see if object is in the list
    iter = search(wordList, wf);
    if (iter != null)
    // yes; increment the word frequency
     iter.next().increment();
    else
    // word is new; insert obj into the list
    wordList.add(wf);
}
displayWords(wordList);
}

public static <T extends Comparable<? super T>>
ListIterator<T> search(OrderedList<T> ordList,
T target)
{
// initialize a list iterator
ListIterator<T> iter = ordList.listIterator();
    T curr;
// move through the ordered list
while (iter.hasNext())
{
    // get the current list value
    curr = iter.next();
    // see if current value matches target
    if (curr.equals(target))
    {
        // match; move iterator back to the match
        // and return its value
        iter.previous();
        return iter;
    }
    // if the target is less than current value,
    // target not found in the ordered list
    else if (target.compareTo(curr) < 0)
    return null;
    // the target is larger than any value
    // in the list
    return null;
}

// output the word and frequency in 15
// character positions; limit output to 4
// elements per line
public static void displayWords(OrderedList aList)
{
ListIterator iter = aList.listIterator();
int count = 0, i;
String blanks;

// strings for input line and parse
// of words on the line
String word = null;
// WordFreq object for elements in wordList
WordFreq wf = null;
// use to search for the current word
// in the ordered list
ListIterator<WordFreq> iter;
// scanner name is a command-line argument
fileIn = new Scanner (new FileHeader(args[0]));

// scanner to parse words in the file
Scanner fileIn;

// words read from file and inserted
// into wordList
OrderedList<WordFreq> wordList =
    new OrderedList<WordFreq>();
// scanner to parse words in the file
Scanner fileIn;

// input words to end-of-file
while (fileIn.hasNext())
{
    word = fileIn.next();
    // create a wordFreq object with frequency 1
    wf = new WordFreq(word);
    // search to see if object is in the list
    iter = search(wordList, wf);
    if (iter != null)
    // yes; increment the word frequency
     iter.next().increment();
    else
    // word is new; insert obj into the list
    wordList.add(wf);
}
displayWords(wordList);

// strings for input line and parse
// of words on the line
String word = null;
// WordFreq object for elements in wordList
WordFreq wf = null;
// use to search for the current word
// in the ordered list
ListIterator<WordFreq> iter;
// scanner name is a command-line argument
fileIn = new Scanner (new FileHeader(args[0]));

// scanner to parse words in the file
Scanner fileIn;

// input words to end-of-file
while (fileIn.hasNext())
{
    word = fileIn.next();
    // create a wordFreq object with frequency 1
    wf = new WordFreq(word);
    // search to see if object is in the list
    iter = search(wordList, wf);
    if (iter != null)
    // yes; increment the word frequency
     iter.next().increment();
    else
    // word is new; insert obj into the list
    wordList.add(wf);
}
displayWords(wordList);

// strings for input line and parse
// of words on the line
String word = null;
// WordFreq object for elements in wordList
WordFreq wf = null;
// use to search for the current word
// in the ordered list
ListIterator<WordFreq> iter;
// scanner name is a command-line argument
fileIn = new Scanner (new FileHeader(args[0]));

// scanner to parse words in the file
Scanner fileIn;

// input words to end-of-file
while (fileIn.hasNext())
{
    word = fileIn.next();
    // create a wordFreq object with frequency 1
    wf = new WordFreq(word);
    // search to see if object is in the list
    iter = search(wordList, wf);
    if (iter != null)
    // yes; increment the word frequency
     iter.next().increment();
    else
    // word is new; insert obj into the list
    wordList.add(wf);
}
displayWords(wordList);

// strings for input line and parse
// of words on the line
String word = null;
// WordFreq object for elements in wordList
WordFreq wf = null;
// use to search for the current word
// in the ordered list
ListIterator<WordFreq> iter;
// scanner name is a command-line argument
fileIn = new Scanner (new FileHeader(args[0]));

// scanner to parse words in the file
Scanner fileIn;
while (iter.hasNext())
{
    String str = iter.next().toString();
    blanks = " ";
    for (i=0;i < 15-str.length();i++)
        blanks += " ";
    System.out.print(blanks);
    System.out.println(str);
    count++;
    if (count % 4 == 0)
        System.out.println();
}
System.out.println;

The Adapter Design Pattern

- The adapter design pattern converts the public portion (interface) of a class into another interface.
- Use the adapter pattern when you want to use an existing class whose interface does not match the one you need.
  - The OrderedList subclass with its revised add() method is an application of the adapter pattern.
- In addition to inheritance, object composition can also be used to realize the adapter pattern. The stack and queue classes are built using this technique.

Selecting a Sequence Collection

- Use an ArrayList if the application needs direct access and the program performs all insertions and deletions at the end of the sequence. If the application requires infrequent modifications to the sequence at intermediate positions, an ArrayList is still acceptable. However, applications requiring frequent insert and delete operations at intermediate locations should use a LinkedList collection.

Selecting a Sequence Collection (end)

- Use a LinkedList when the application requires frequent insertions and deletions at arbitrary locations in the list and direct (index) access is not required. A LinkedList is not a good choice if the application requires many search operations. These applications should use a set or map. These collections provide very fast running times for searches.