# Lists and the `Collection` Interface

Based on Koffmann and Wolfgang

Chapter 4

# Chapter Outline

- The **`List`** interface
- Writing an array-based implementation of **`List`**
- Linked list data structures:
  - Singly-linked
  - Doubly-linked
  - Circular
- Implementing **`List`** with a linked list

# Chapter Outline (2)

- The **`Iterator`** interface
- Implementing **`Iterator`** for a linked list
- The Java **`Collection`** framework (hierarchy)

# The **`List`** Interface

- An *array* is an indexed structure:
  - Select elements *in any order*, using subscript values
  - That is: such selection is *efficient*
- Access elements in *sequence*:
  - Using a *loop that increments* the subscript
- You *cannot*
  - Increase/decrease the *length*
  - Add an element in the middle ...
    - Without shifting elements to make room
  - Remove an element ...
    - Without shifting elements to fill in the gap

# The `List` Interface (2)

The **`List`** interface supports:

- Obtaining the element at a specified position
- Replacing the element at a specified position
- Determining if a specific object is in the list, and where
- Adding or removing an element at either end
- Inserting or removing an element at any position
- Removing a specific object, regardless of position
- Obtaining the number of elements in the list
- Traversing the list structure without a subscript
- ... etc.

# The **List** Interface (3)

Some **`List<E>`** operations:

```
E get (int index);  // checks bounds
E set (int index, E element);
boolean contains (Object obj);
int indexOf (Object obj);
int size ();
boolean isEmpty ();
boolean add (E element);
void add (int index, E element);
boolean remove (Object obj);
E remove (int index);
```

# The **`List`** Interface (4)

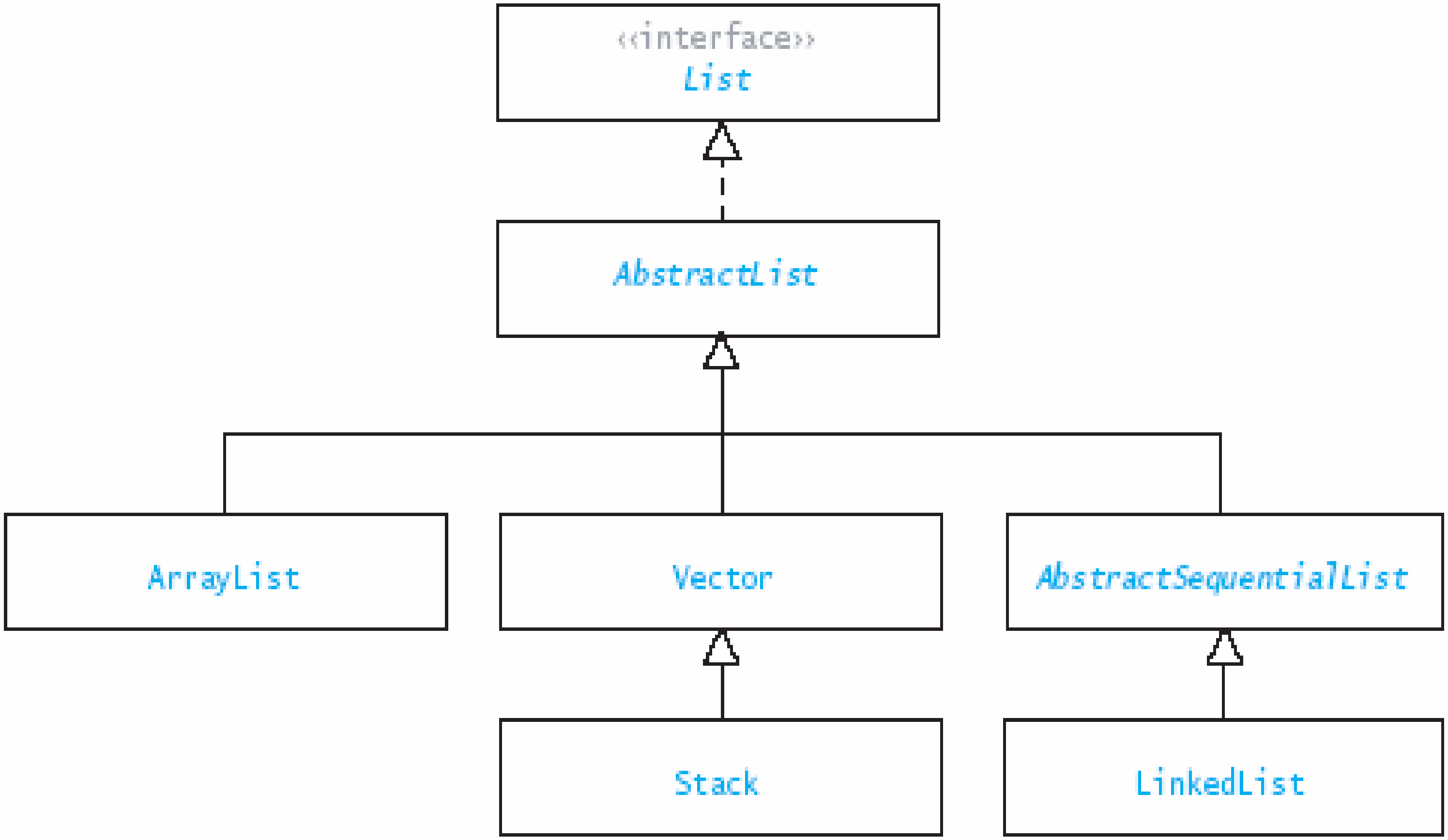
- Implementations vary in the efficiency of operations
  - Array-based class efficient for access by position
  - Linked-list class efficient for insertion/deletion

- Arrays can store primitive-type data
- The **`List`** classes all store references to objects

# The **`List`** Hierarchy

**FIGURE 4.1**

The `java.util.List` Interface and Its Implementers



# The **`ArrayList`** Class

- Simplest class that implements the **`List`** interface
- Improvement over an array object
- Use when:
  - Wants to add new elements to the end of a list
  - Still need to access elements quickly in any order

```
List<String> lst = new ArrayList<String>();
lst.add(“Bashful”);
lst.add(“Awful”);
lst.add(“Jumpy”);
lst.add(“Happy”);
```

# The **`ArrayList`** Class (2)

Original List

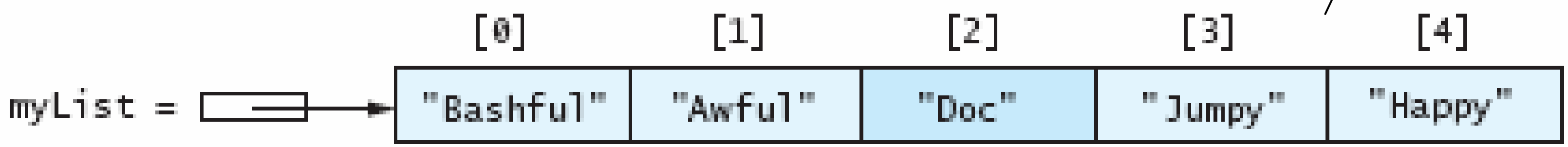
**`lst.add(2, “Doc”);`**

# The `ArrayList` Class (3)

**FIGURE 4.2**

Insertion in the Middle and at the End of an `ArrayList` Object

Original List



After insertion of "Doc" before the third element

**Had to slide [2],[3] to [3],[4]**

# The `ArrayList` Class (4)

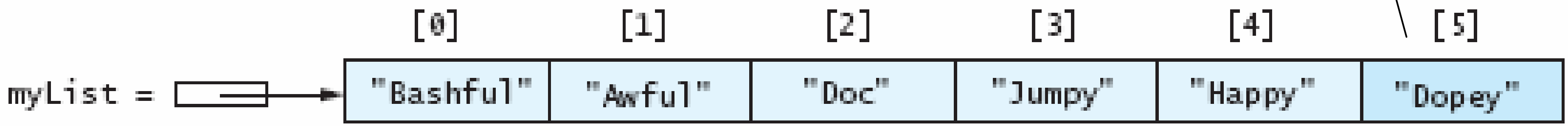
**FIGURE 4.2**

Insertion in the Middle and at the End of an `ArrayList` Object

Original List

`lst.add(“Dopey”);  // add at end`

**Cheap to add at end**



After insertion of "Dopey" at the end

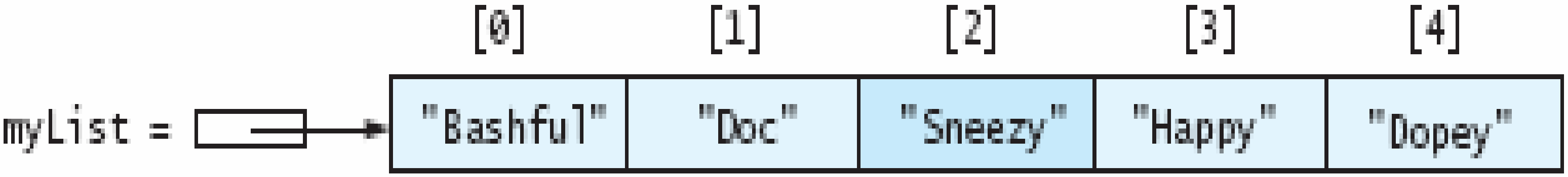
# The **`ArrayList`** Class (5)

`lst.remove(1);`

After removal of "Awful"

**Had to slide [2..5] to [1..4]**

`lst.set(1, “Sneezy”);`



After replacing "Jumpy" with "Sneezy"

**Replacement is cheap**

# Using **`ArrayList`**

- Additional capability beyond what an array provides
  - Insertion, deletion, built-in search, etc.

- Stores items of any Object class

- Cannot store values of primitive type:
  - Must use wrapper classes

# Generic Collection **`ArrayList<E>`**

- The **`<E>`** indicates a *type parameter*
- Here **`E`** can be any Object type
- Every element of **`ArrayList<E>`** must obey **`E`**
- This is a Java 5.0 innovation
- Previously all you had was **`ArrayList`**
    - That is equivalent to 5.0 **`ArrayList<Object>`**
    - So **`ArrayList<E>`** is more restrictive:
        - Catches more errors at compile time!

# Generic Collection **`ArrayList<E>`** (2)

```
List<String> lst = new ArrayList<String>();
ArrayList<Integer> numList =
    new ArrayList<Integer>();

lst.add(35);    // will not type check

numList.add(“xyz”);  // will not type check
numList.add(new Integer(35));  // ok
numList.add(35);  // also ok: auto-boxes
```

# Why Use Generic Collections?

- Better type-checking: catch more errors, earlier
- Documents intent
- Avoids downcast from **`Object`**

# How Did They Maintain Compatibility?

- Generics are strictly a compiler thing
- They do *not* appear in bytecode
- It is as if the <...> stuff is *erased*
  - Called *erasure semantics*
- We tell you because there are places where it affects what you write, etc.

# Example Applications of **`ArrayList`**

```
ArrayList<Integer> someInts =
    new ArrayList<Integer>();
int[] nums = {5, 7, 2, 15};
for (int i = 0; i < nums.length; ++i) {
  someInts.add(nums[i]);
}

int sum = 0;
for (int i = 0; i < someInts.size(); i++) {
  sum += someInts.get(i);
}
System.out.println(“sum is “ + sum);
```

# Example Applications of **ArrayList** (2)

```
ArrayList<Integer> someInts =
    new ArrayList<Integer>();
int[] nums = {5, 7, 2, 15};
for (int n : nums) {
  someInts.add(n);
}

int sum = 0;
for (int n : someInts) {
  sum += n;
}
System.out.println(“sum is “ + sum);
```

# Using **`ArrayList`** in **`PhoneDirectory`**

```
private ArrayList<DirectoryEntry> dir =
    new ArrayList<DirectoryEntry>();

int index = dir.indexOf(
    new DirectoryEntry(name, “”));

DirectoryEntry ent = dir.get(index);

dir.add(new DirectoryEntry(name, newNumber));
```

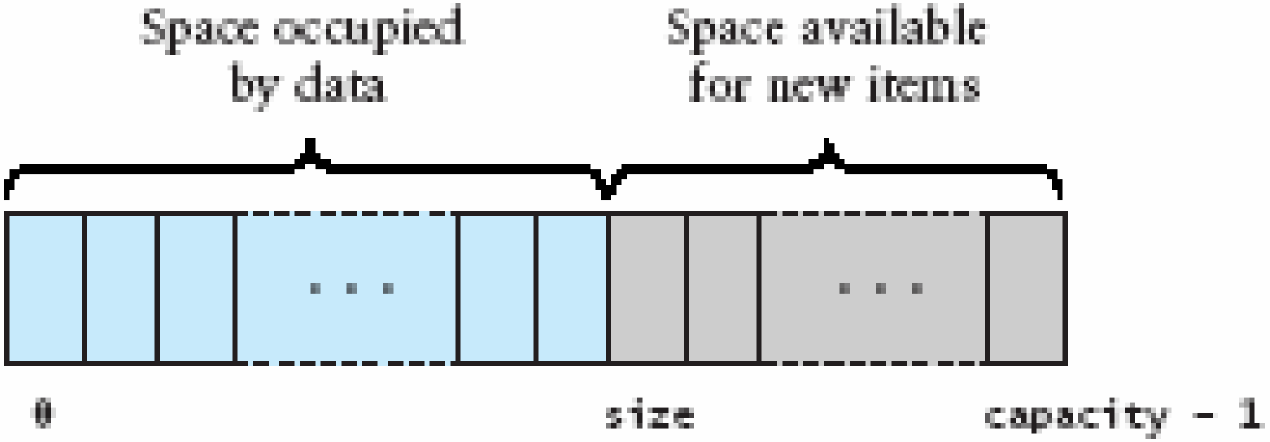
# Using **ArrayList** in **PhoneDirectory** (2)

```
public String addOrChangeEntry (String name,
                                String number) {
  int index = dir.indexOf(
    new DirectoryEntry(name, “”));
  String oldNumber = null;
  if (index != -1) {
    DirectoryEntry ent = dir.get(index);
    oldNumber = ent.getNumber();
    ent.setNumber(number);
  } else {
    dir.add(new DirectoryEntry(name, newNumber));
  }
  modified = true;
  return oldNumber;
}
```

# Implementing an **`ArrayList`** Class

- **`KWArrayList`**: simple implementation of **`ArrayList`**
  - Physical size of array indicated by data field **`capacity`**
  - Number of data items indicated by the data field **`size`**

FIGURE 4.3
Internal Structure of `ArrayList`



# `KWArrayList` Fields, Constructor

```
public class KWArrayList<E> {
  private static final int INITIAL_CAPACITY = 10;
  private E[] theData;
  private int size = 0;
  private int capacity = 0;

  public KWArrayList () {
    capacity = INITIAL_CAPACITY;
    theData = (E[]) new Object[capacity];
    // Cast above needed because of erasure
    // semantics; cannot do new E[capacity].
    // Cast will generate a compiler warning;
    // it’s ok!
  }
}
```

# Implementing **`ArrayList.add(E)`**

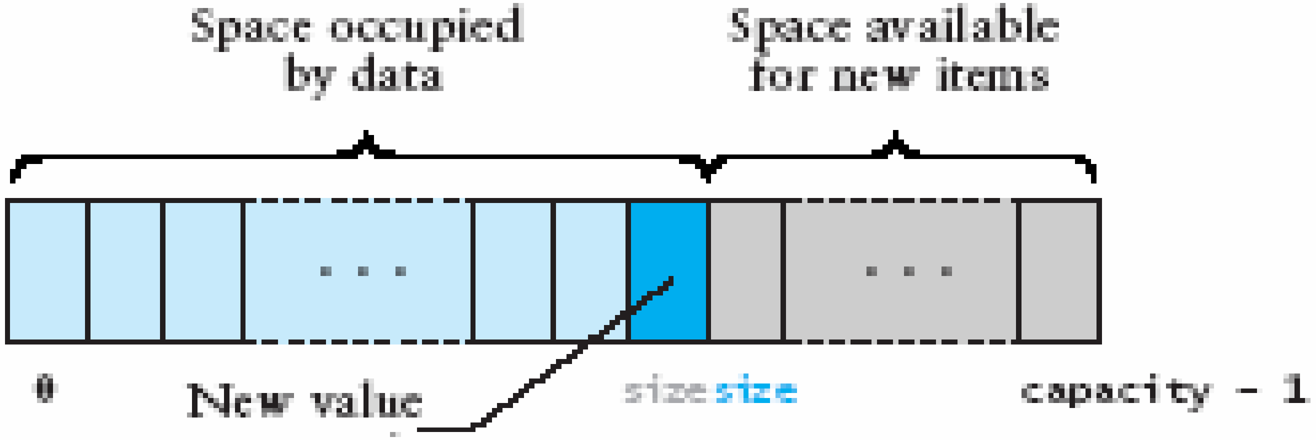


**FIGURE 4.4**
Adding an Element to the End of an `ArrayList`

# Implementing **`ArrayList.add(E)`** (2)

```
public boolean add (E anEntry) {
  if (size == capacity) {
    reallocate();
  }
  theData[size] = anEntry;
  size++;
  return true;
}
```

# Implementing **`ArrayList.add(E)`** (3)

```
public boolean add (E anEntry) {
  if (size >= capacity) {
    reallocate();
  }
  theData[size++] = anEntry;
  return true;
}
```

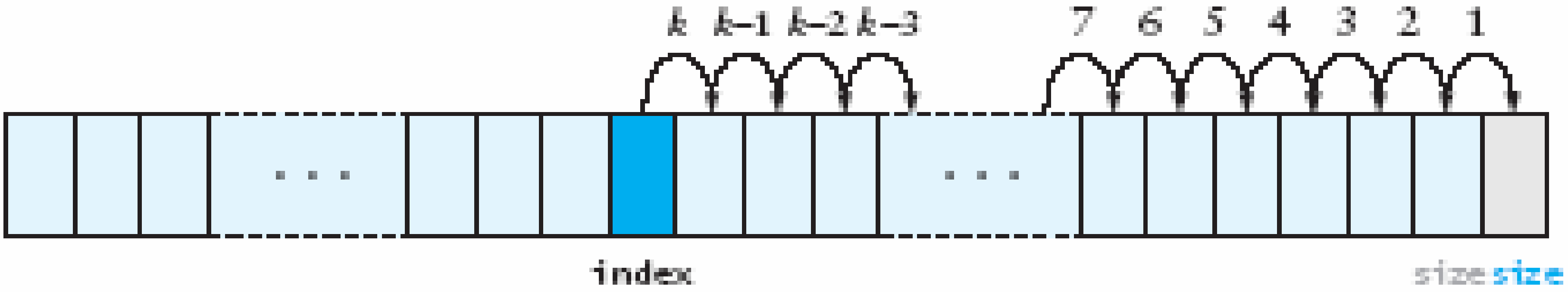
# Implementing **`ArrayList.reallocate()`**

```
private void reallocate () {
  capacity *= 2;  // or some other policy
  E[] newData = (E[])new Object[capacity];
  System.arraycopy(theData, 0,
                   newData, 0, size);
  theData = newData;
}
```

# Implementing **`ArrayList.add(int,E)`**

**FIGURE 4.5**

Making Room to Insert an Item into an Array



# Implementing `ArrayList.add(int,E)` (2)

```
public void add (int index, E anEntry) {
  // check bounds
  if (index < 0 || index > size) {
    throw new
      IndexOutOfBoundsException(index); }
  // insure there is room
  if (size == capacity) { reallocate(); }
  // shift data
  for (int i = size; i > index; i--) {
    theData[i] = theData[i-1];
  }
  // insert item
  theData[index] = anEntry;
  size++;
}
```

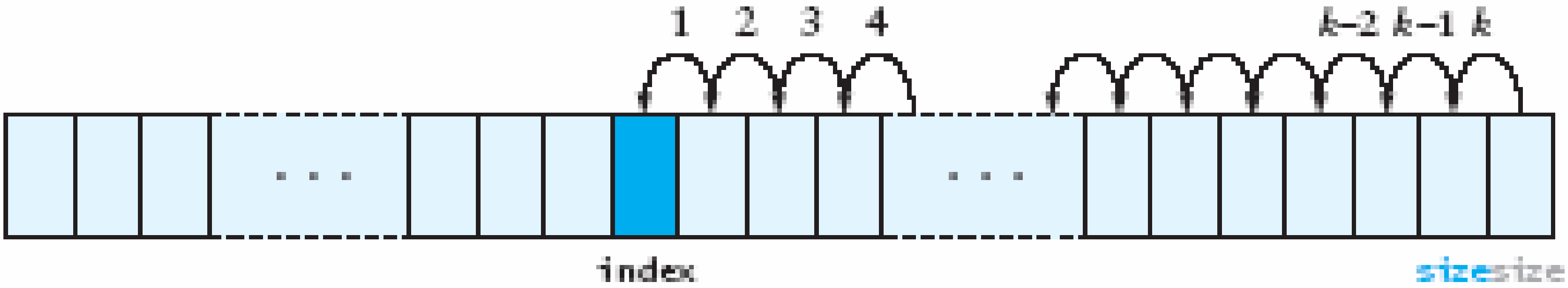
# Implementing **`ArrayList.add(int,E)`** (3)

```
public void add (int index, E anEntry) {
  ...
  // shift data: arraycopy may be faster
  System.arraycopy(theData, index,
                   theData, index+1,
                   size-index);

  ...
}
```

# Implementing **`ArrayList.remove(int)`**

**FIGURE 4.6**

Removing an Item from an Array



# Implementing **`ArrayList.remove(int)`** (2)

```
public E remove (int index) {
  if (index < 0 || index >= size) {
    throw new
      IndexOutOfBoundsException(index);
  }
  E returnValue = theData[index];
  for (int i = index + 1; i < size; i++) {
    theData[i-1] = theData[i];
  }
  size--;
  return returnValue;
}
```

# Implementing **`ArrayList.remove(int)`** (3)

```
public E remove (int index) {
    ...
    System.arraycopy(theData, index+1,
                     theData, index,
                     size-(index+1));
    ...
}
```

# Implementing **`ArrayList.get(int)`**

```
public E get (int index) {
  if (index < 0 || index >= size) {
    throw new
      IndexOutOfBoundsException(index);
  }
  return theData[index];
}
```

# Implementing `ArrayList.set(int,E)`

```
public E set (int index, E newValue) {
  if (index < 0 || index >= size) {
    throw new
      IndexOutOfBoundsException(index);
  }
  E oldValue = theData[index];
  theData[index] = newValue;
  return oldValue;
}
```

# Performance of **`KWArrayList`** and the **`Vector`** Class

- Set and get methods execute in constant time: O(1)
- Inserting or removing general elements is linear time: O(n)
- Adding at end is (usually) constant time: O(1)
  - With our reallocation policy the *average* is O(1)
  - The worst case is O(n) because of reallocation
- Initial release of Java API contained the **`Vector`** class which has similar functionality to the **`ArrayList`**
  - Both contain the same methods
- New applications should use **`ArrayList`** rather than **`Vector`**
- **`Stack`** is a subclass of **`Vector`**

# The **`Stack`** Class

- **`Stack<E>`** is a subclass of **`Vector<E>`**
- It supports the following additional methods:

```
boolean empty();
E peek();
E pop();
E push(E item);
int search(Object o);
```

# Singly-Linked Lists and Doubly-Linked Lists

- The **`ArrayList`** **`add`** and **`remove`** methods are O(n)
  - Because they need to shift the underlying array
- Linked list overcomes this:
  - Add/remove items anywhere in constant time: O(1)
- Each element (node) in a linked list stores:
  - The element information, of type **`E`**
  - A *link* to the *next* node
  - A *link* to the *previous* node (optional)

# A List Node

- A node contains:
  - A (reference to a) data item
  - One or more *links*
- A link is a reference to a list node
- The node class is usually defined inside another class:
  - It is a hidden *inner class*
- The details of a node should be kept private

# List Nodes for Singly-Linked Lists

**FIGURE 4.12**

Node and Link

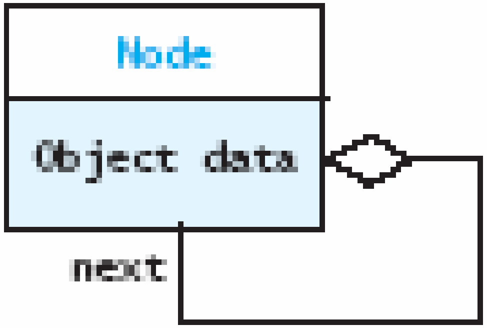
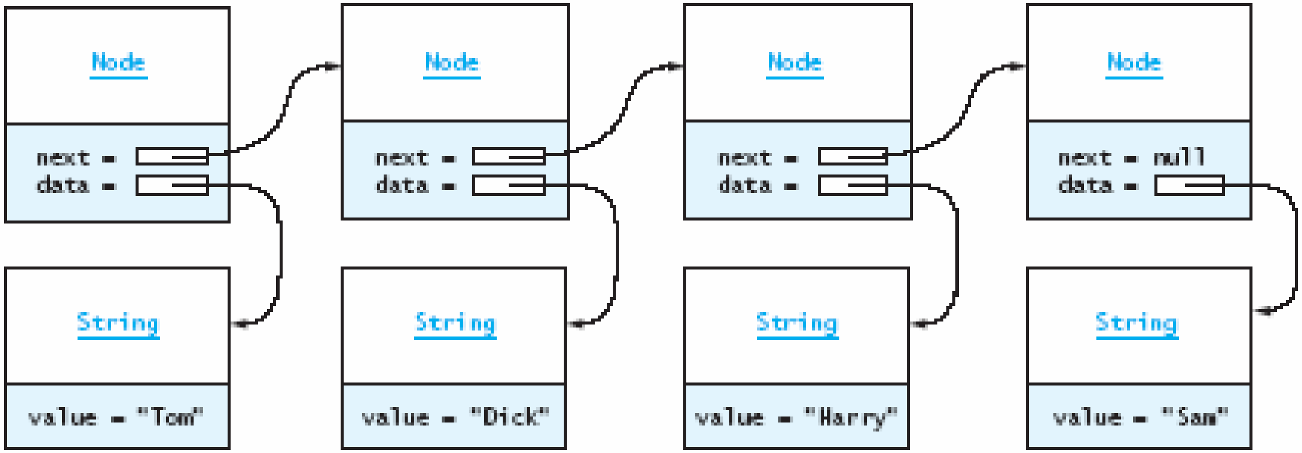


**FIGURE 4.13**

Nodes in a Linked List



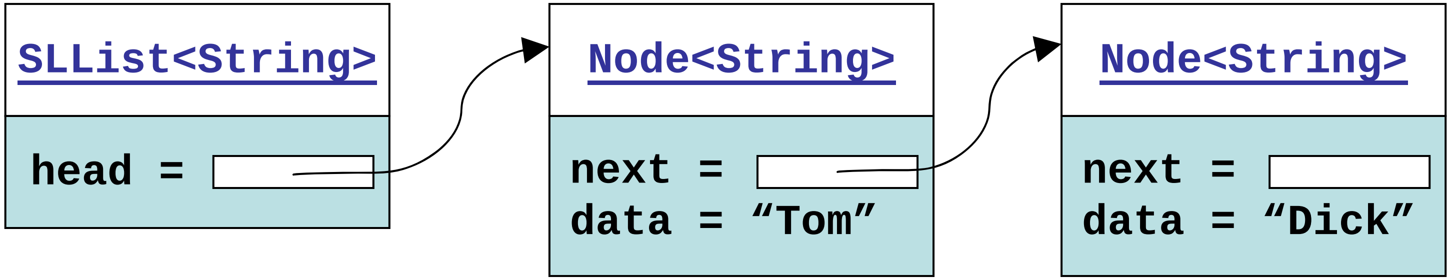
# List Nodes for Singly-Linked Lists

```
// Note: all private members of a private inner
// class are visible to the containing class
private static class Node<E> {
  private E data;
  private Node<E> next;

  private Node (E data, Node<E> node) {
    this.data = data;
    this.next = node;
  }

  private Node (E data) {
    this(data, (Node<E>)null);
  }
}
```

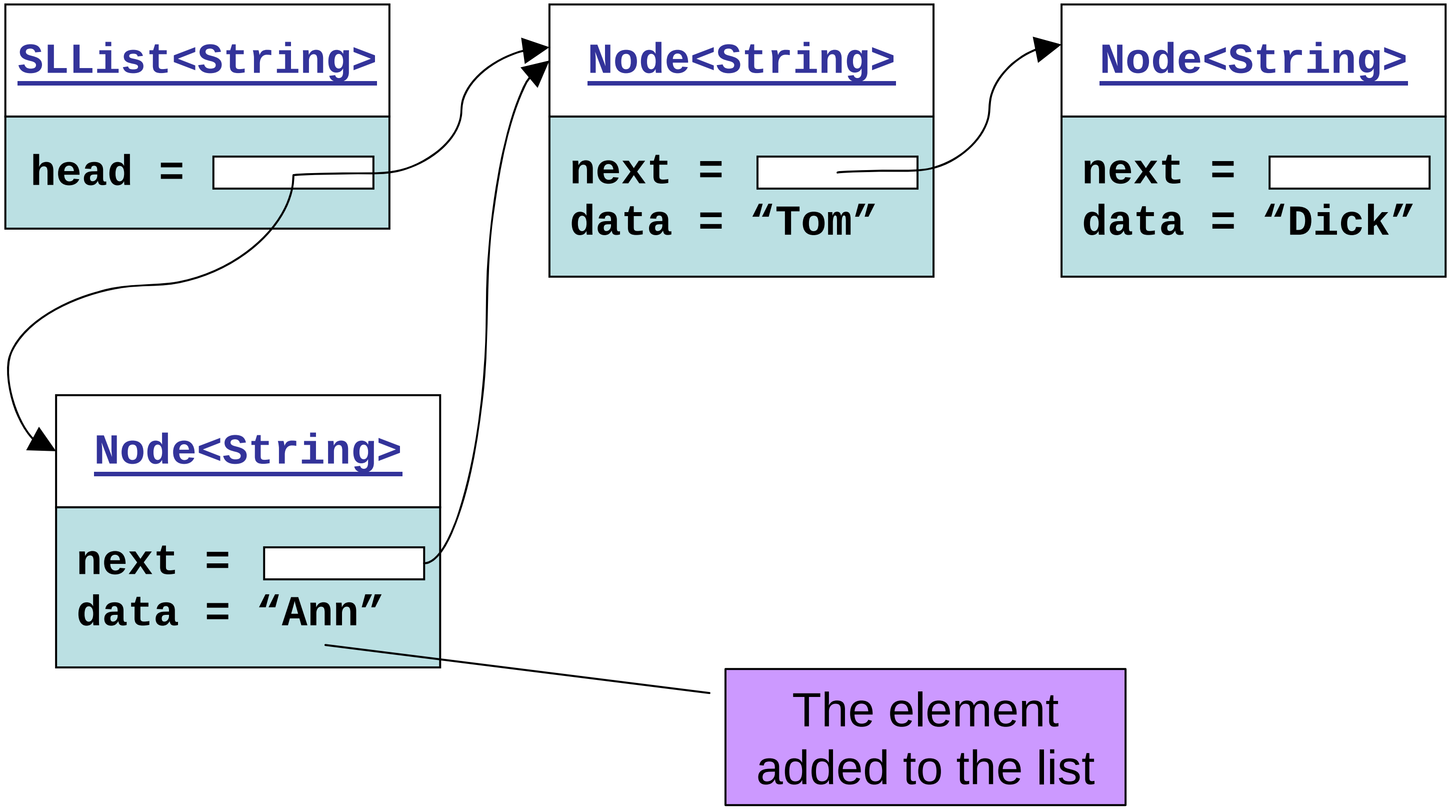
# Implementing **SLList**

```
public class SLList<E> implements List<E> {
  private Node<E> head = null;

  private static class Node<E> { ... }

  public SLList () { }

  ...
}
```

# Implementing **`SLList`**: An Example List



# Implementing **`SLList.addFirst(E)`**



SLList<String>

head =

Node<String>

next =
data = “Tom”

Node<String>

next =
data = “Dick”

Node<String>

next =
data = “Ann”

The element added to the list

# Implementing **`SLList`** Add Before First (2)

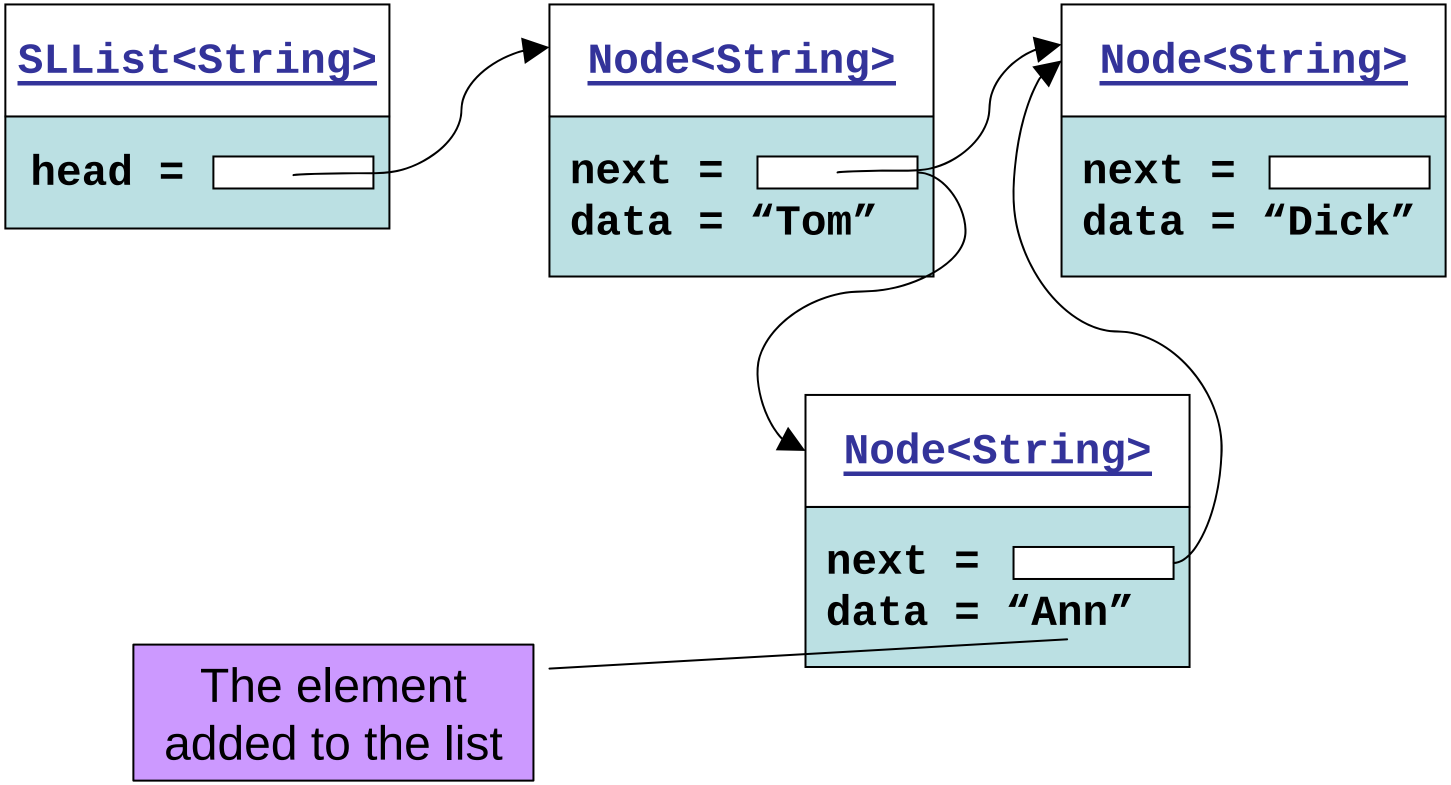
```
private void addFirst (E item) {
  Node<E> temp = new Node<E>(item, head);
  head = temp;
}
```

... *or, more simply* ...

```
private void addFirst (E item) {
  head = new Node<E>(item, head);
}
```

Note: This works fine even if **`head`** is **`null`**.

# Implementing **`addAfter(Node<E>, E)`**



The element added to the list

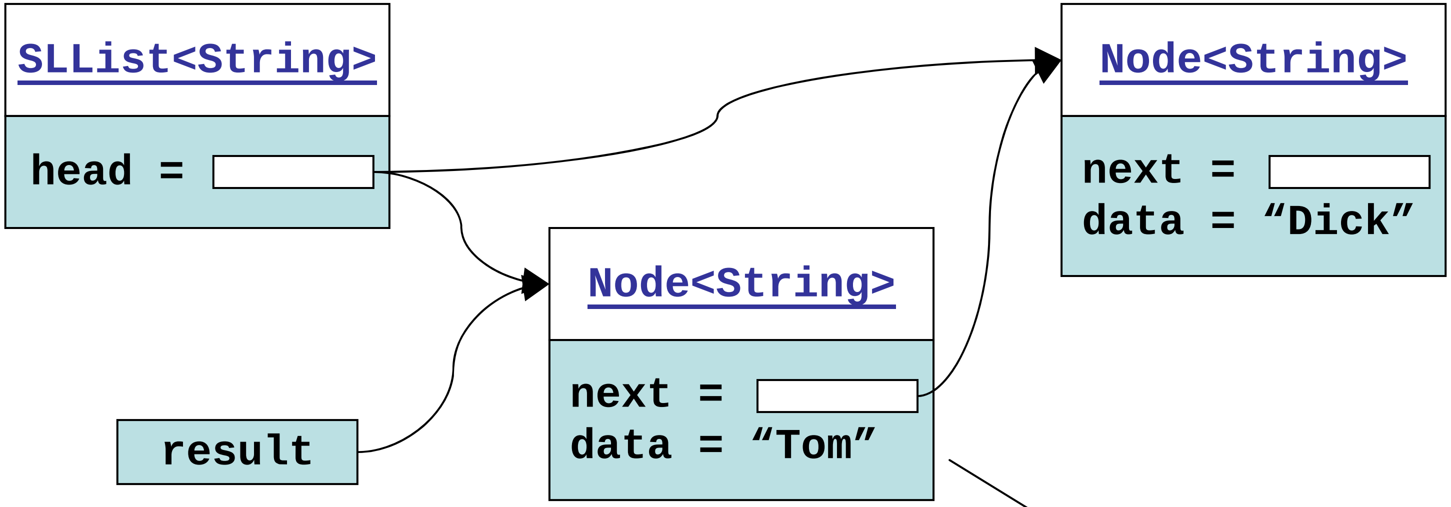
# Implementing **`addAfter(Node<E>, E)`**

```
private void addAfter (Node<E> node, E item) {
  Node<E> temp = new Node<E>(item, node.next);
  node.next = temp;
}
```

... *or, more simply* ...

```
private void addAfter (Node<E> node, E item) {
  node.next = new Node<E>(item, node.next);
}
```

# Implementing `SLList.removeFirst()`

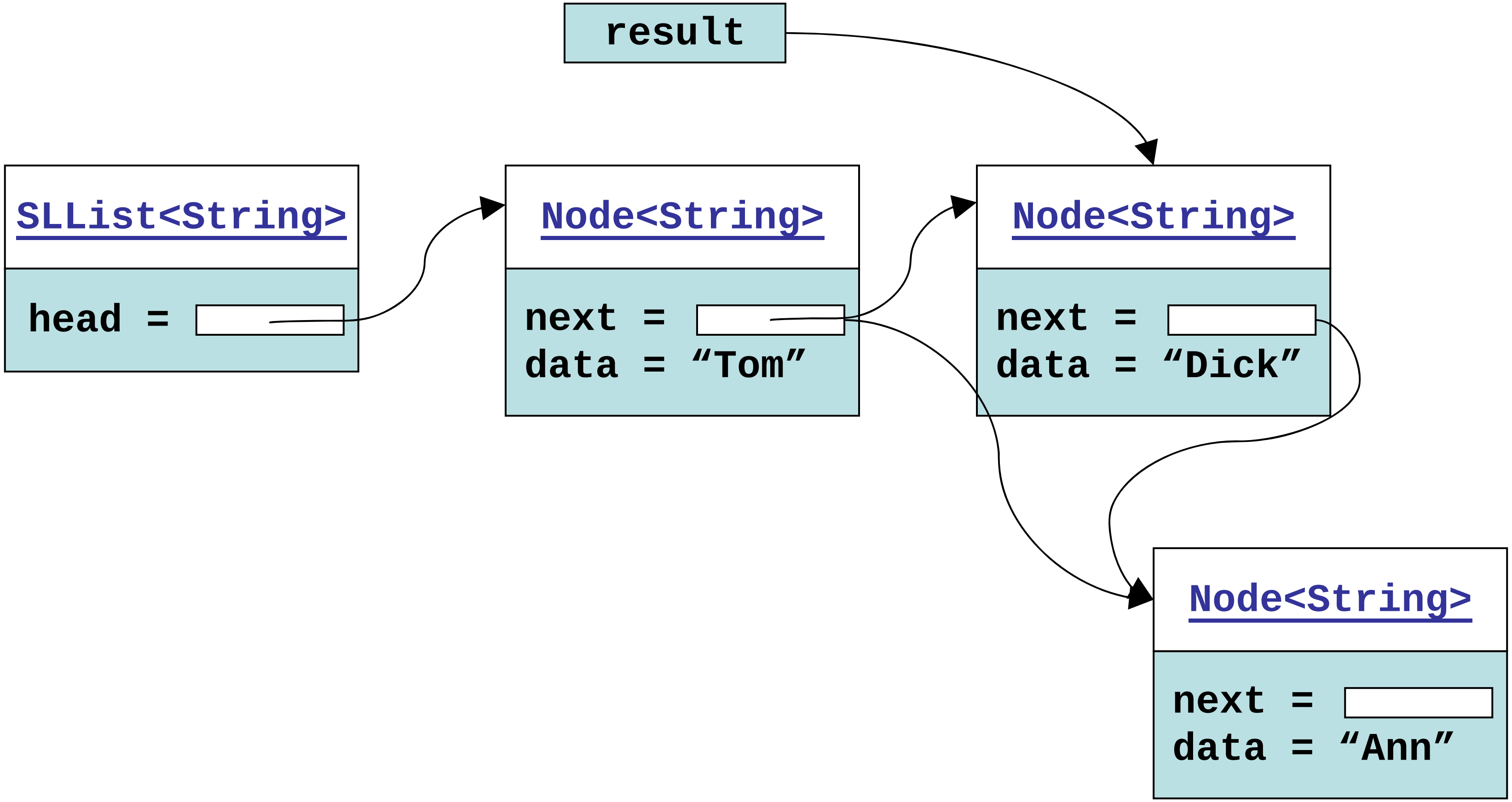


Node is unlinked, but data may live on

# Implementing **`SLList.removeFirst()`**

```
private Node<E> removeFirst () {
  Node<E> result = head;
  if (head != null) {
    head = head.next;
  }
  return result;
}
```

# Implementing **`removeAfter(Node<E>)`**



# Implementing **removeAfter(Node<E>)**

```
private Node<E> removeAfter (Node<E> node) {
  Node<E> result = node.next;
  if (result != null) {
    node.next = result.next;
  }
  return result;
}
```

# Using **add/remove First/After** for **SLList**

```
// first, we need a helper method:
// getNode(i) gets the ith Node, where
// the first Node is numbered 0

private Node<E> getNode (int index) {
  Node<E> node = head;
  for (int i = 0; i < index; ++i) {
    if (node == null) break;
    node = node.next;
  }
  return node;
}
```

# **SLList** Helper Method Summary

```
private void addFirst (E item)
private void addAfter (Node<E> node, E item)

private Node<E> removeFirst ()
private Node<E> removeAfter (Node<E> node)

private Node<E> getNode (int index)
```

We now show how to use these to implement **SLList** methods ...

# SLList.get(int)

```
public E get (int index) {
  Node<E> node = getNode(index);
  if (index < 0 || node == null) {
    throw
      new IndexOutOfBoundsException(index);
  }
  return node.data;
}
```

# SLList.set(int)

```
public E set (int index, E newValue) {
  Node<E> node = getNode(index);
  if (index < 0 || node == null) {
    throw
      new IndexOutOfBoundsException(index);
  }
  E result = node.data;
  node.data = newValue;
  return result;
}
```

# SLList.add(int, E)

```
public void add (int index, E anEntry) {
  if (index == 0) {
    addFirst(anEntry);  // index 0 is special
  } else {
    Node<E> node = getNode(index-1);
    if (index < 0 || node == null) {
      throw new
        IndexOutOfBoundsException(index);
    }
    addAfter(node, anEntry);
  }
}
```

# SLList.remove(int)

```
public E remove (int index) {
  Node<E> removed = null;
  if (index == 0) {
    removed = removeFirst(index);
  } else if (index > 0) {
    Node<E> node = getNode(index-1);
    if (node != null) {
      removed = removeAfter(node);
    }
  }
  if (removed == null) {
    throw new
      IndexOutOfBoundsException(index);
  }
  return removed.data;
}
```

# SLList.add(E)

```
private Node<E> getLast () {
  Node<E> node = head;
  if (node != null) {
    while (node.next != null)
      node = node.next;
  }
  return node;
}
public void add (E anEntry) {
  if (head == null) {
    addFirst(anEntry);
  } else {
    addAfter(getLast(), anEntry);
  }
}
```

# `SLList` Performance

- get/set(i) = O(i)
- add/remove(i) = O(i)
  - add(0) = O(1)
  - add at end = O(n)
- Hardly seems an improvement over **`ArrayList`**!
- We can easily improve add-at-end
- We can also speed up some common patterns ...

# Making **`add(E)`** Faster

- Idea: add a field that remembers the *last* element
- What needs fixing?
  - Constructor: set it to null
  - Add/remove: update it as necessary

# Supporting **`last`** for **`SLList`**

```
private void addFirst (E item) {
  Node<E> temp = new Node<E>(item, head);
  if (head == null) { last = temp; }
  head = temp;
}

private void addAfter (Node<E> node, E item) {
  Node<E> temp = new Node<E>(item, node.next);
  if (last == node) { last = temp; }
  node.next = temp;
}
```

# Supporting `last` for `SLList` (2)

```
private Node<E> removeFirst () {
  Node<E> result = head;
  if (last == head) { last = null;      }
  if (head != null) { head = head.next; }
  return result;
}
private Node<E> removeAfter (Node<E> node) {
  Node<E> result = node.next;
  if (last == result) { last = node; }
  if (result != null) {
    node.next = result.next;
  }
  return result;
}
```

# Using **`last`** for **`SLList.add(E)`**

```
public void add (E anEntry) {
  if (head == null) {
    addFirst(anEntry);
  } else {
    addAfter(last, anEntry);
  }
}
```

# Making Other Operations Faster

- Idea: remember a recent *index* and its **`Node`**
- Complicates existing routines (of course)
- Helps when accessing same or forward
- No help when accessing backward

(We won’t pursue it in detail ....)

# Doubly-Linked Lists

- Limitations of a singly-linked list include:
  - Can insert only after a referenced node
  - Removing node requires pointer to previous node
  - Can traverse list only in the forward direction
- We can remove these limitations:
  - Add a pointer in each node to the previous node: *doubly-linked list*

# Doubly-Linked Lists: Recall Singly-Linked

**FIGURE 4.14**
After Inserting bob

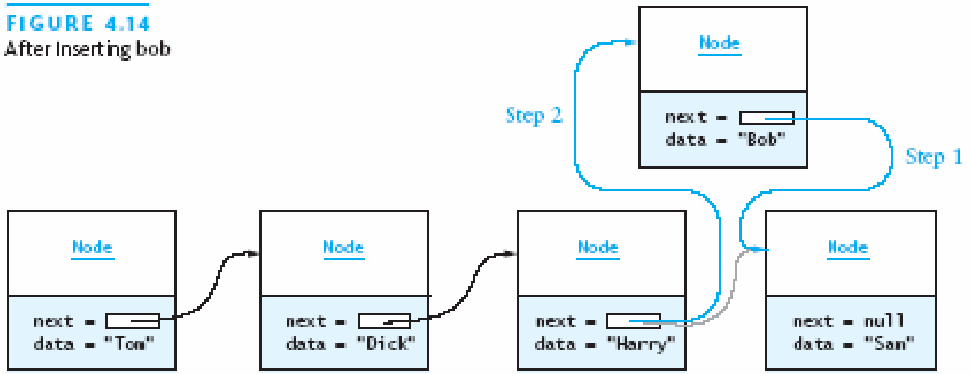
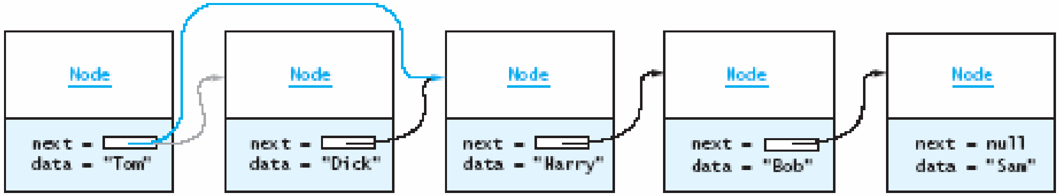


**FIGURE 4.15**
After Removing Node Following tom



# Doubly-Linked Lists, The Diagrams

**FIGURE 4.16**
Double-Linked List
Node UML Diagram

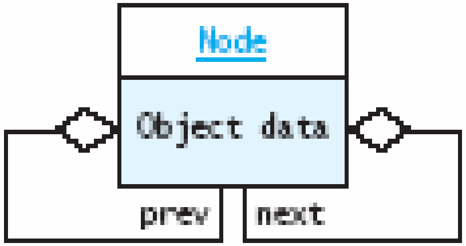
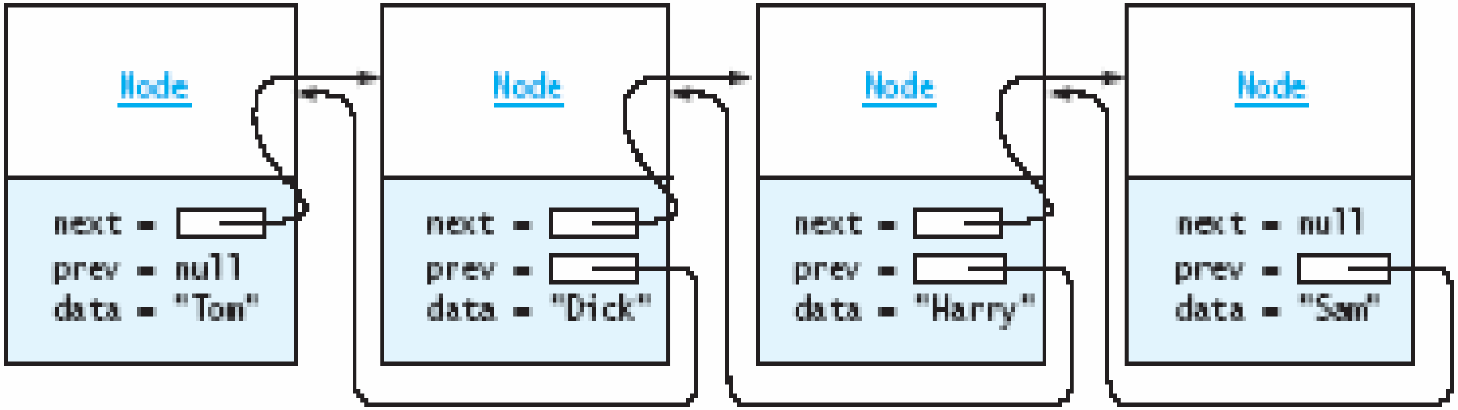
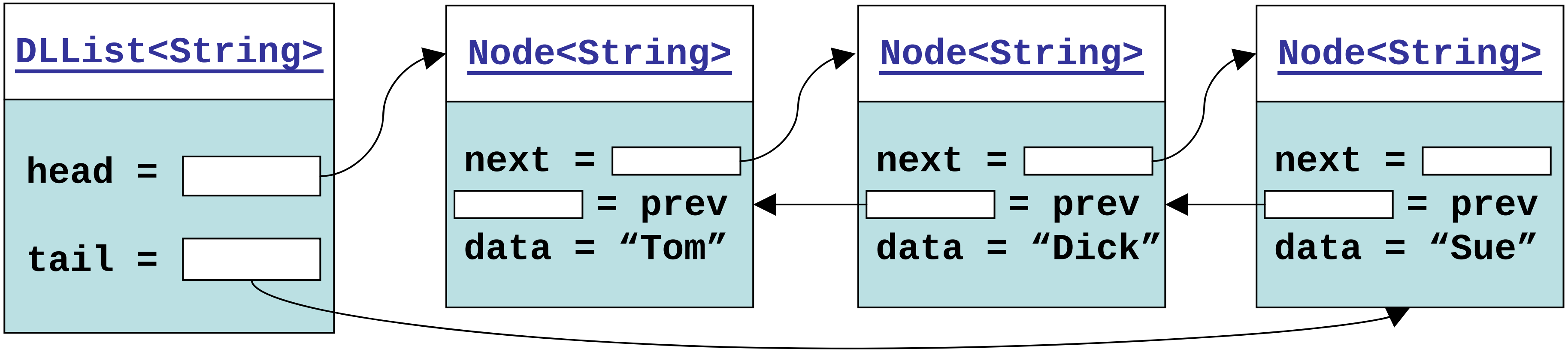


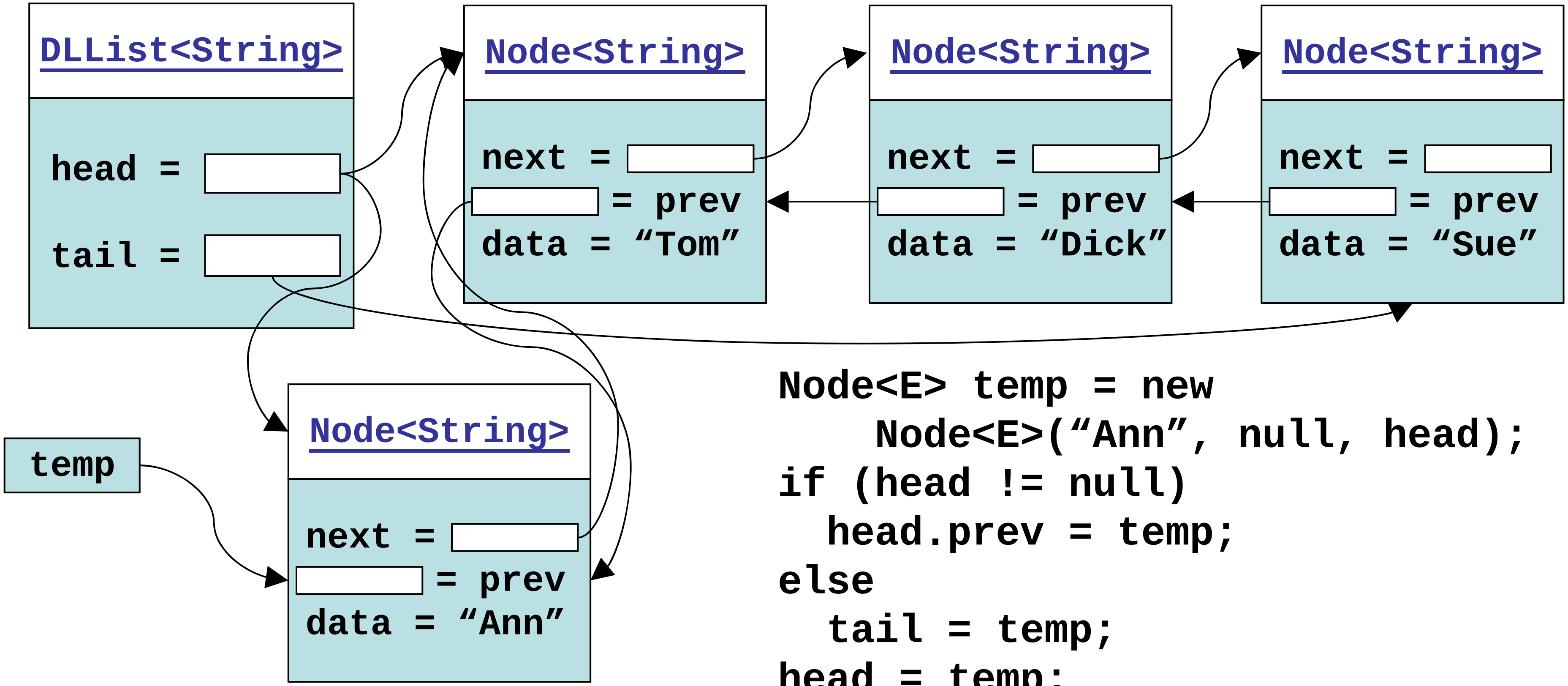
**FIGURE 4.17**
A Double-Linked List



# Implementing **`DLList`**: An Example List



# Inserting Into **`DLList`** at Head



**DLList<String>**

head =

tail =

**Node<String>**

next =

= prev

data = “Tom”

**Node<String>**

next =

= prev

data = “Dick”

**Node<String>**

next =

= prev

data = “Sue”

temp

**Node<String>**

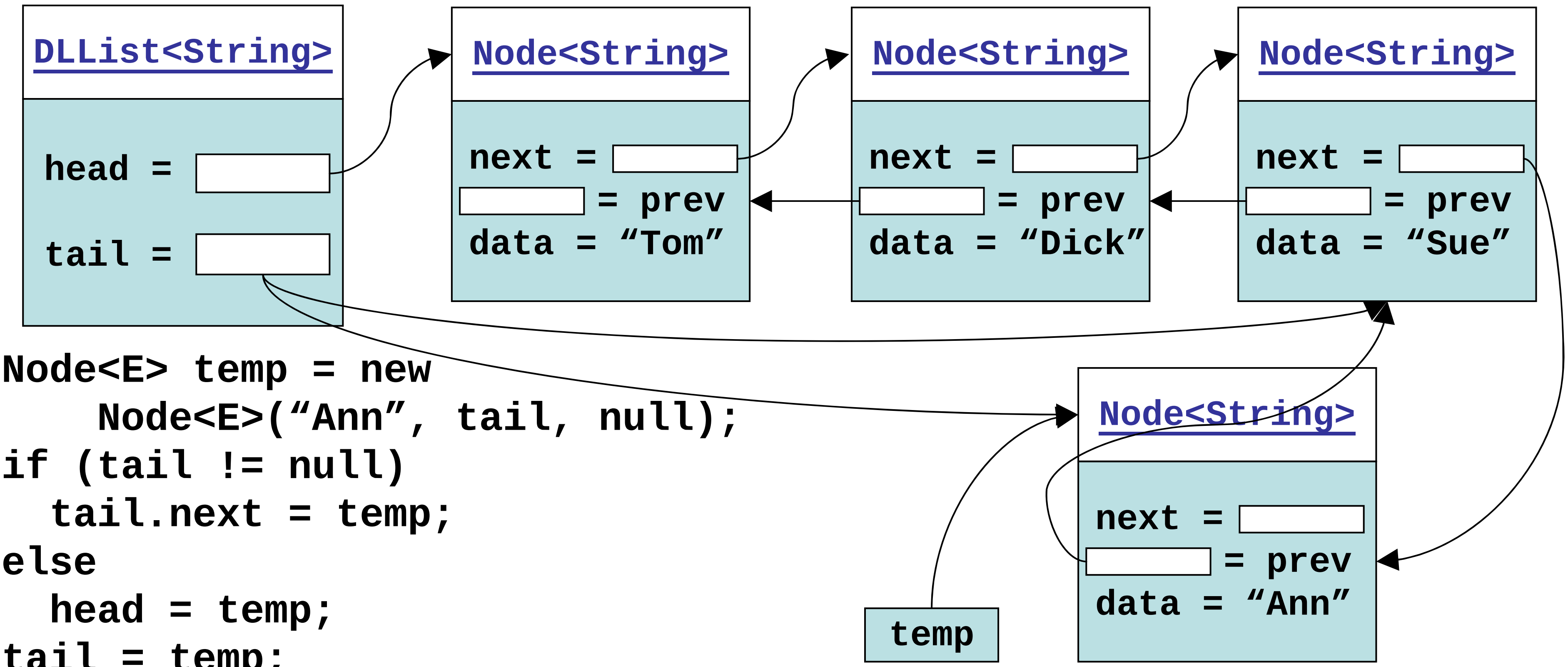
next =

= prev

data = “Ann”

```
Node<E> temp = new
    Node<E>(“Ann”, null, head);
if (head != null)
  head.prev = temp;
else
  tail = temp;
head = temp;
```

# Inserting Into **`DLList`** at Tail



**DLList<String>**

head =

tail =

**Node<String>**

next =

= prev

data = “Tom”

**Node<String>**

next =

= prev

data = “Dick”

**Node<String>**

next =

= prev

data = “Sue”

**Node<String>**

next =

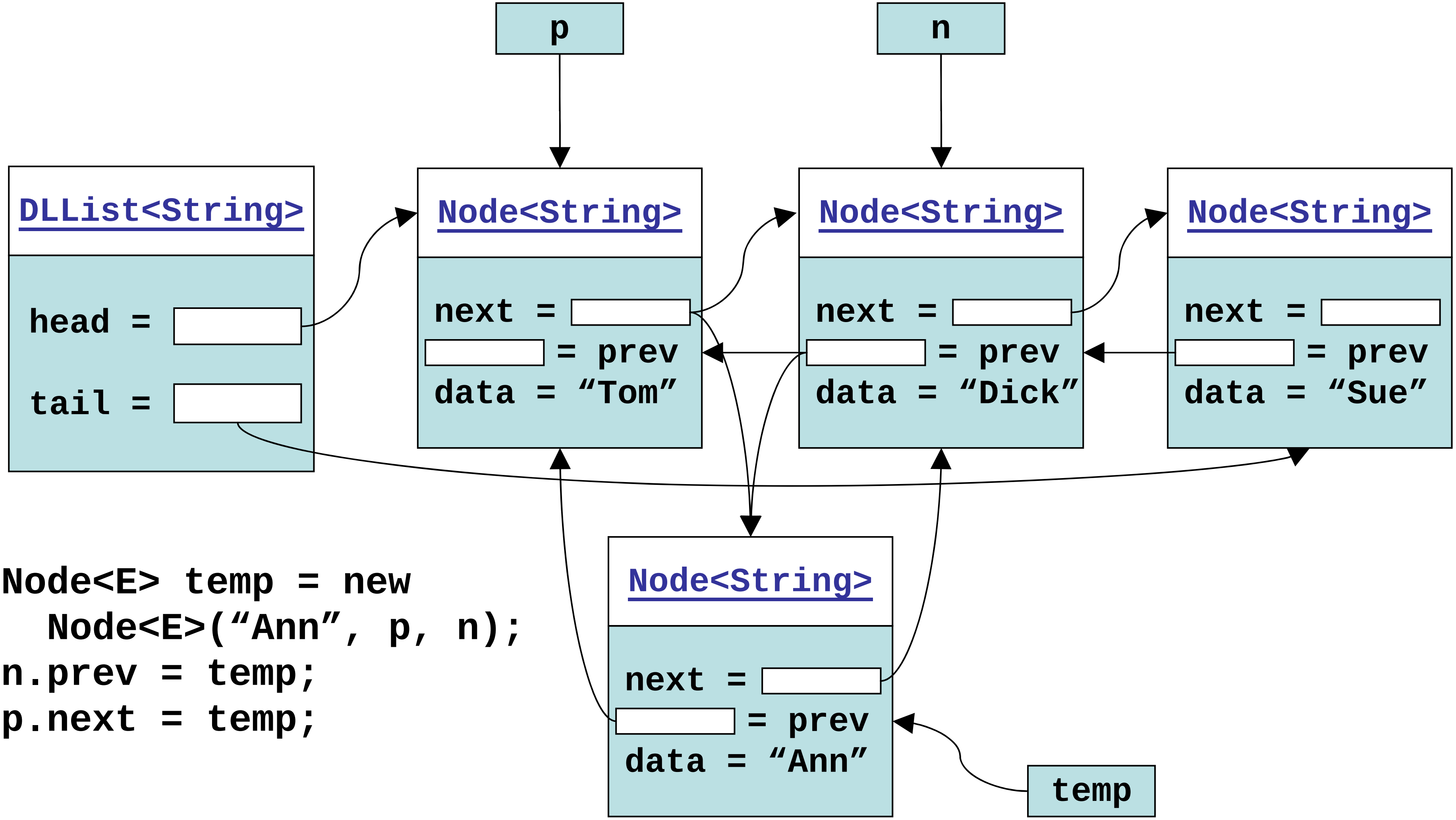
= prev

data = “Ann”

temp

```
Node<E> temp = new
    Node<E>(“Ann”, tail, null);
if (tail != null)
  tail.next = temp;
else
  head = temp;
tail = temp;
```

# Inserting Into **`DLList`** in Middle



p

n

DLList<String>

head =

tail =

Node<String>

next =

= prev

data = “Tom”

Node<String>

next =

= prev

data = “Dick”

Node<String>

next =

= prev

data = “Sue”

Node<String>

next =

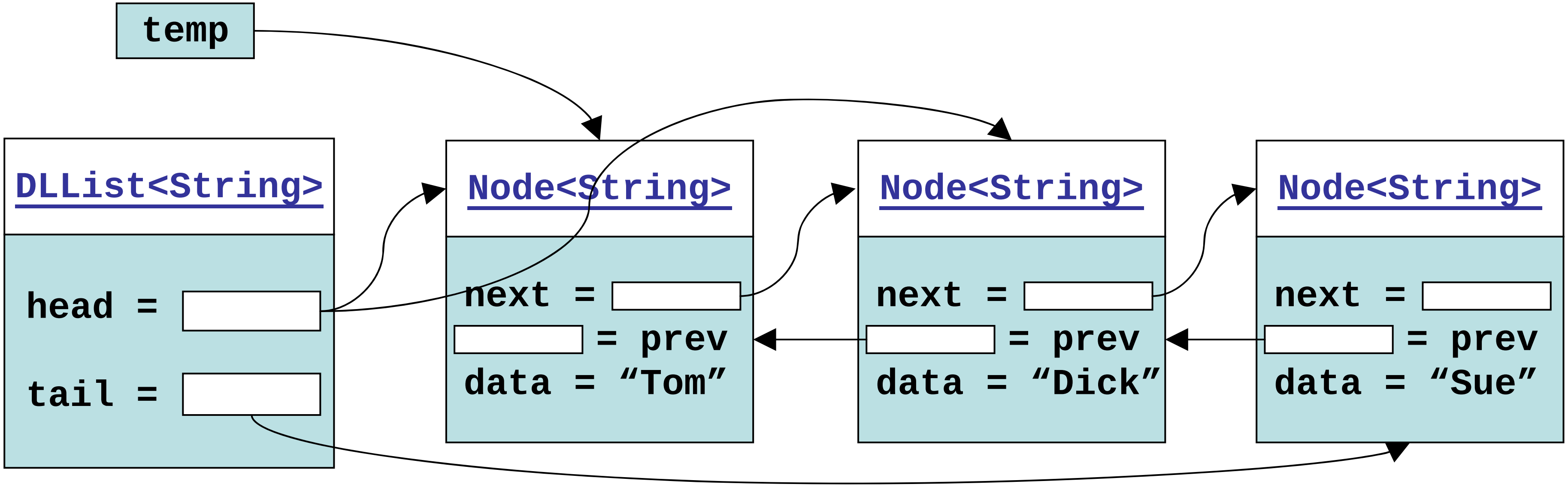
= prev

data = “Ann”

temp

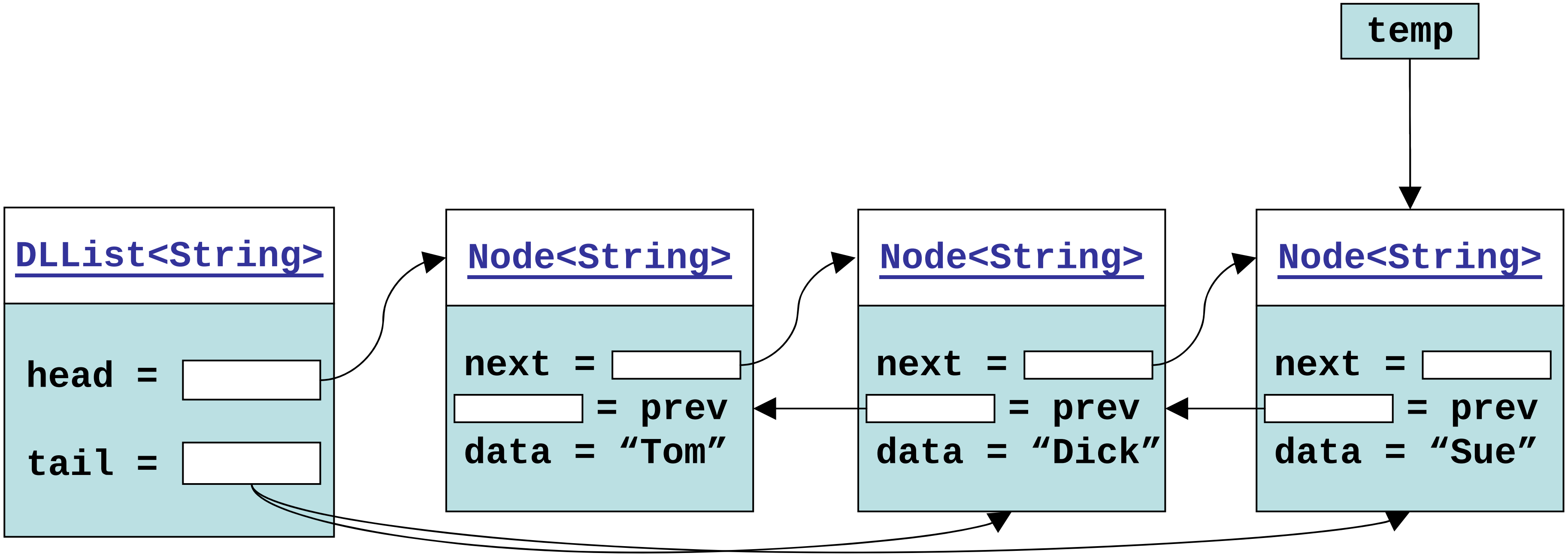
```
Node<E> temp = new
  Node<E>(“Ann”, p, n);
n.prev = temp;
p.next = temp;
```

# Removing From **DLList** at Head



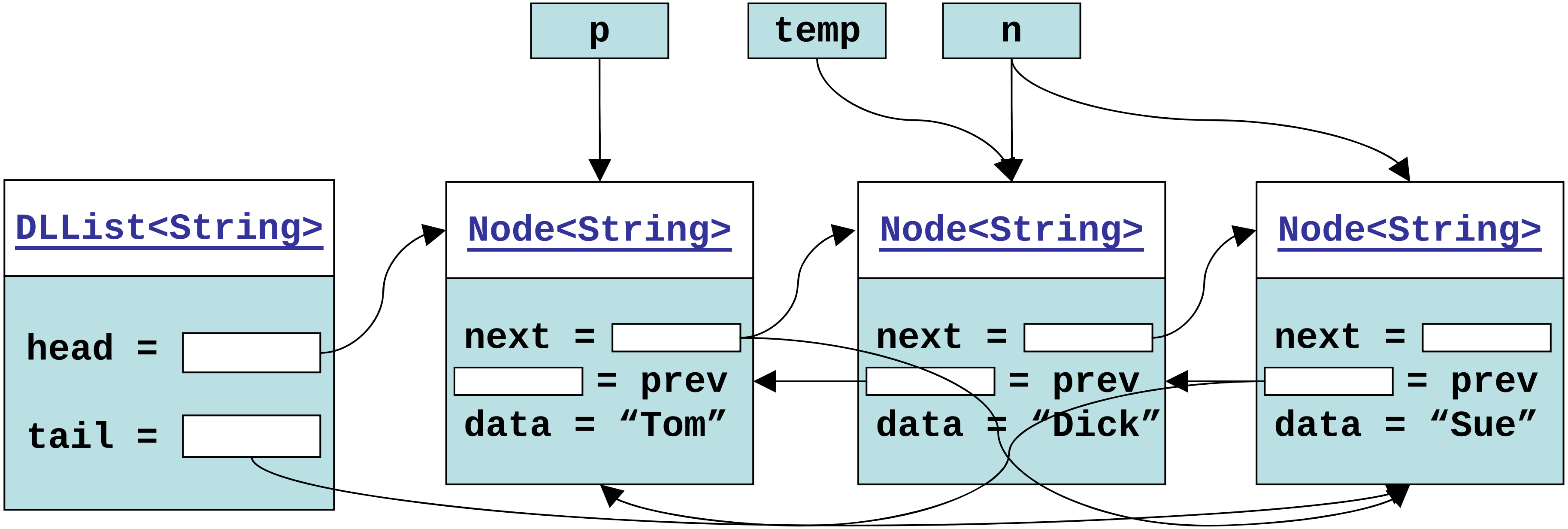
```
temp = head;
if (head != null)
  head = head.next;
if (head != null)
  head.prev = null;
else
  tail = null;
```

# Removing From **`DLList`** at Tail



```
temp = tail;
if (tail != null)
  tail = tail.prev;
if (tail != null)
  tail.next = null;
else
  head = null;
```

# Removing From **DLList** in Middle

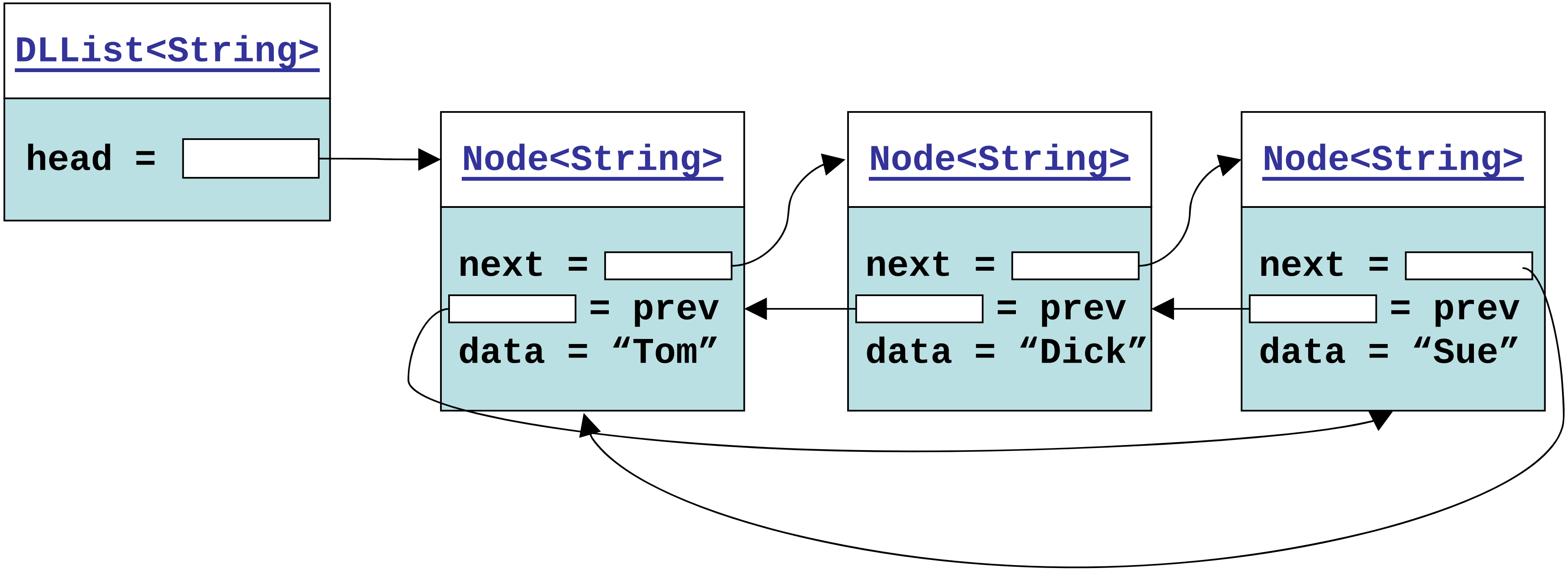


```
Node<E> temp = n;
n = n.next;
p.next = n;
n.prev = p;
```

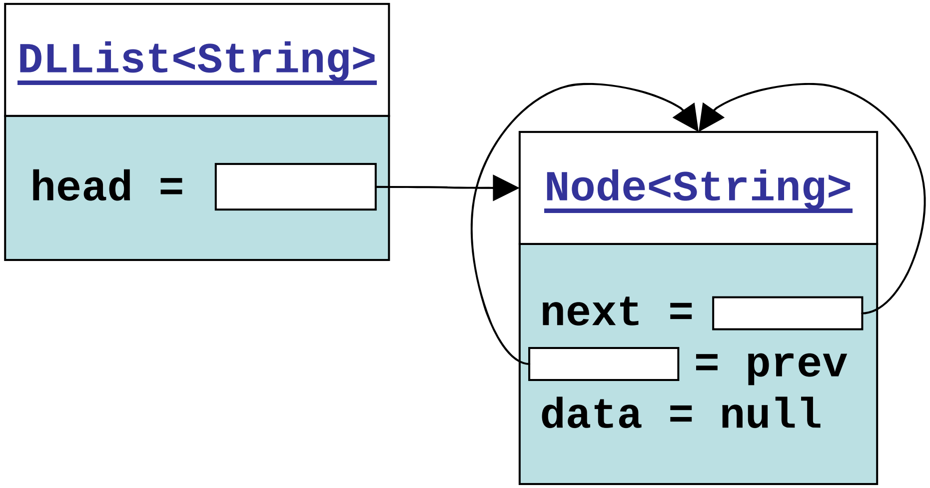
# Circular Lists

- Circular doubly-linked list:
  - Link last node to the first node, and
  - Link first node to the last
- *Advantages*:
  - Can easily keep going “past” ends
  - Can visit all elements from any starting point
  - Can never fall off the end of a list
- *Disadvantage*: code must avoid infinite loop!
- Can also build singly-linked circular lists:
  - Traverse in forward direction only

# Implementing **`DLList`** Circularly



# Implementing **`DLList`** With a “Dummy” Node



- The “dummy” node is always present
- Eliminates null pointer cases
  - Even for an empty list
- Effect is to simplify the code
- Helps for singly-linked and non-circular too

# Putting It Together

- The **`LinkedList`** class uses a **`DLList`** with dummy node
- To support cheap insertion/deletion/traversal we introduce: **`ListIterator`**
  - A moveable pointer “into” a **`LinkedList`**
- Go through implementation of such a **`LinkedList`** class

# The **`LinkedList`** Class

- Part of the Java API
- Implements the **`List`** interface using a double-linked list

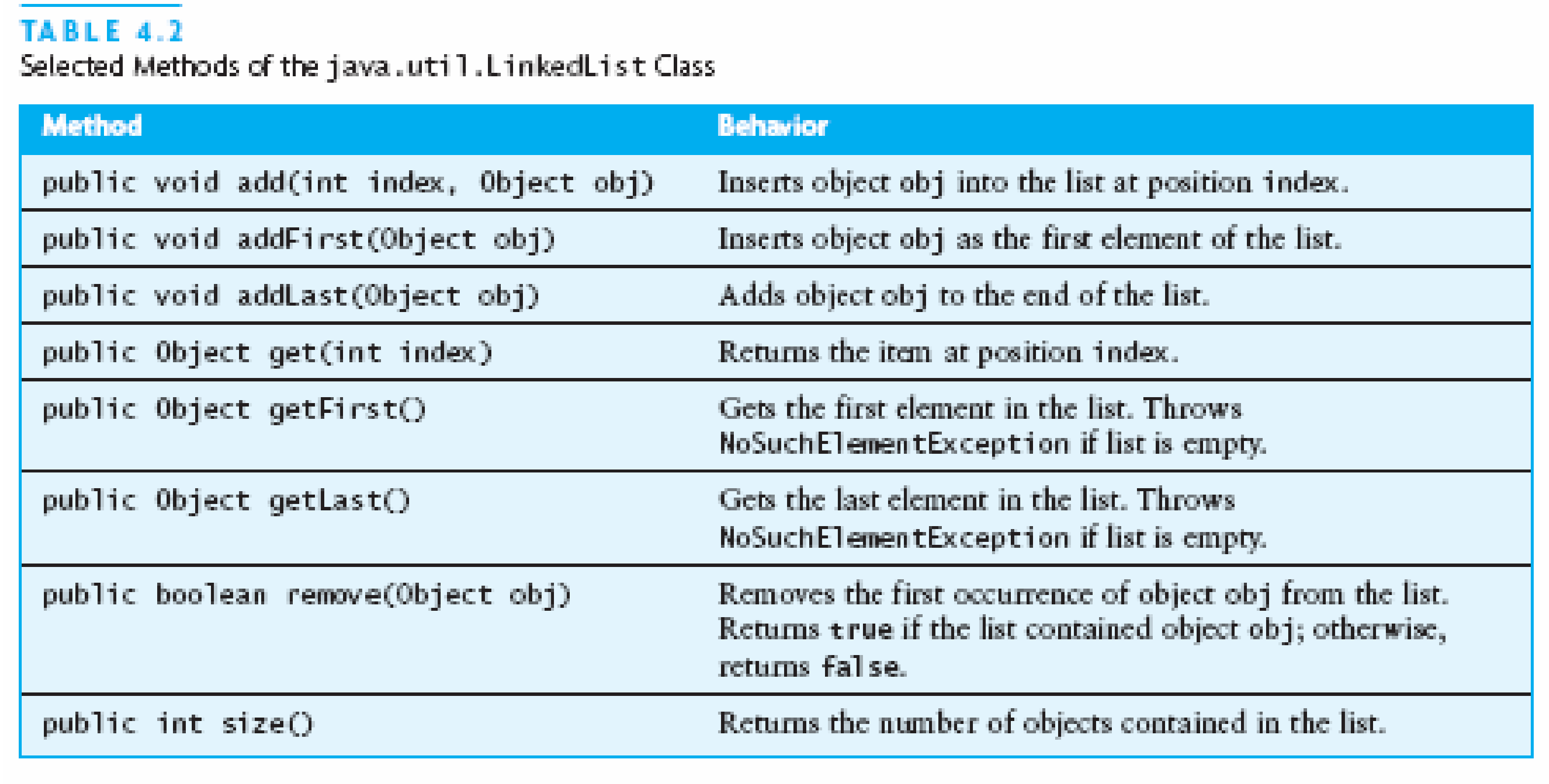


**TABLE 4.2**

Selected Methods of the `java.util.LinkedList` Class

| Method | Behavior |
|---|---|
| `public void add(int index, Object obj)` | Inserts object `obj` into the list at position `index`. |
| `public void addFirst(Object obj)` | Inserts object `obj` as the first element of the list. |
| `public void addLast(Object obj)` | Adds object `obj` to the end of the list. |
| `public Object get(int index)` | Returns the item at position `index`. |
| `public Object getFirst()` | Gets the first element in the list. Throws `NoSuchElementException` if list is empty. |
| `public Object getLast()` | Gets the last element in the list. Throws `NoSuchElementException` if list is empty. |
| `public boolean remove(Object 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` Interface

- The `Iterator` interface is defined in `java.util`
- The `List` interface declares the method `iterator`
  - Returns an `Iterator` object
  - That iterates over the elements of that list
- An `Iterator` does not refer to a particular object at any time

# The **`Iterator`** Interface (2)

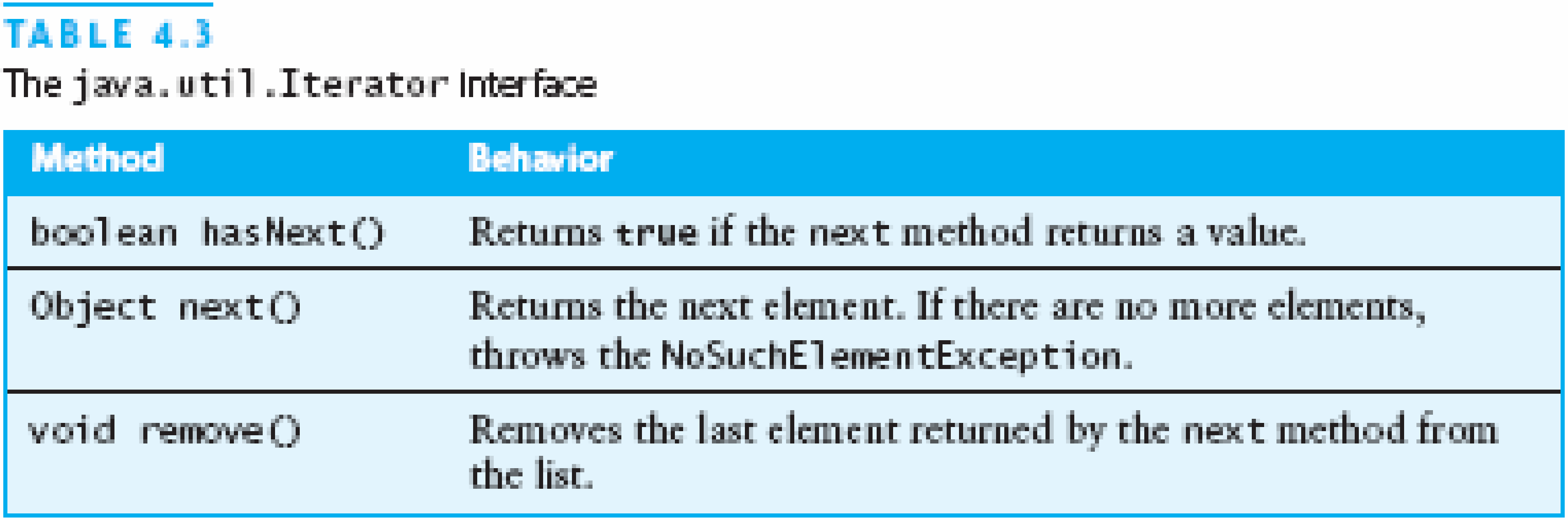


**TABLE 4.3**

The `java.util.Iterator` Interface

| Method | Behavior |
|---|---|
| `boolean hasNext()` | Returns `true` if the `next` method returns a value. |
| `Object 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 from the list. |

# The `Iterator` Interface: Typical Use

```
List<E> lst = ...;
Iterator<E> iter = lst.iterator();
while (iter.hasNext()) {
  System.out.println(iter.next().toString());
}
```

*Alternatively (Java 5.0 for-each loop):*

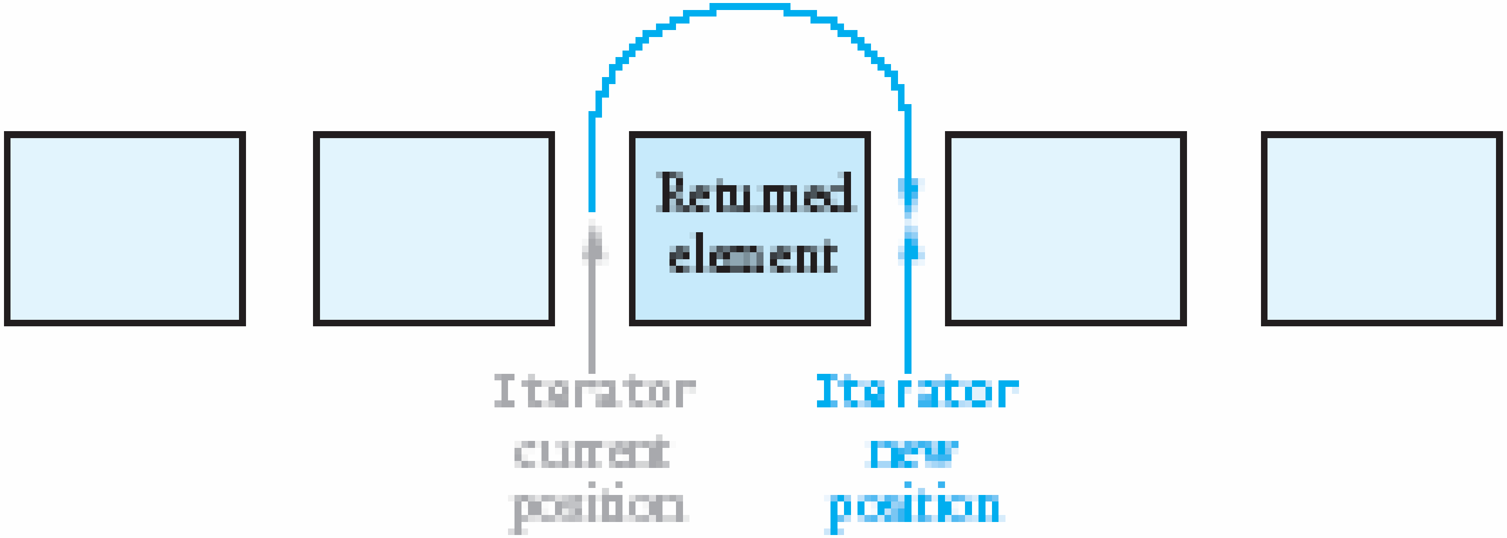
```
for (E elem : lst) {
  System.out.println(elem.toString());
}
```

# Picture of an **`Iterator`**

- *Point:* An **`Iterator`** is conceptually *between* elements

FIGURE 4.23

Advancing an Iterator via the next Method



# Iterators and Removing Elements

- Interface **`Iterator`** supports removing: **`void remove()`**
- What it does is delete the *most recent element returned*
- So, you must invoke **`next()`** before each **`remove()`**
- What about **`LinkedList.remove`**?
  - It must walk down the list, then remove
  - So in general it is O(n)
  - Versus **`Iterator.remove`**, which is O(1)
- Further, you should not mix them:
  - Most iterators fail, throwing **`ConcurrentModificationException`**, if you make changes any other way

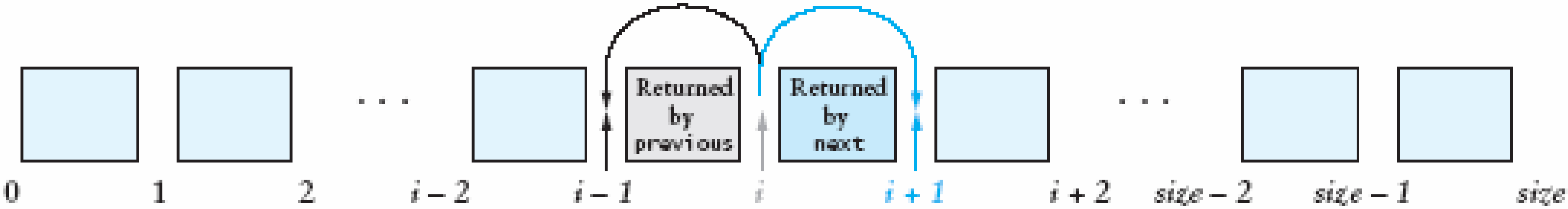
# The **ListIterator** Interface

- **Iterator** limitations
  - Can traverse **List** only in the forward direction
  - Provides **remove** method, but no **add**
  - Must advance iterator using your own loop if not starting from the beginning of the list
- **ListIterator** adds to Iterator, overcoming these limitations
- As with **Iterator**, **ListIterator** best imagined as being positioned *between* elements of the list

# Imagining `ListIterator`

- Diagram also illustrates the numbering of positions

**FIGURE 4.23**
The `ListIterator`



# The `ListIterator` Interface (continued)

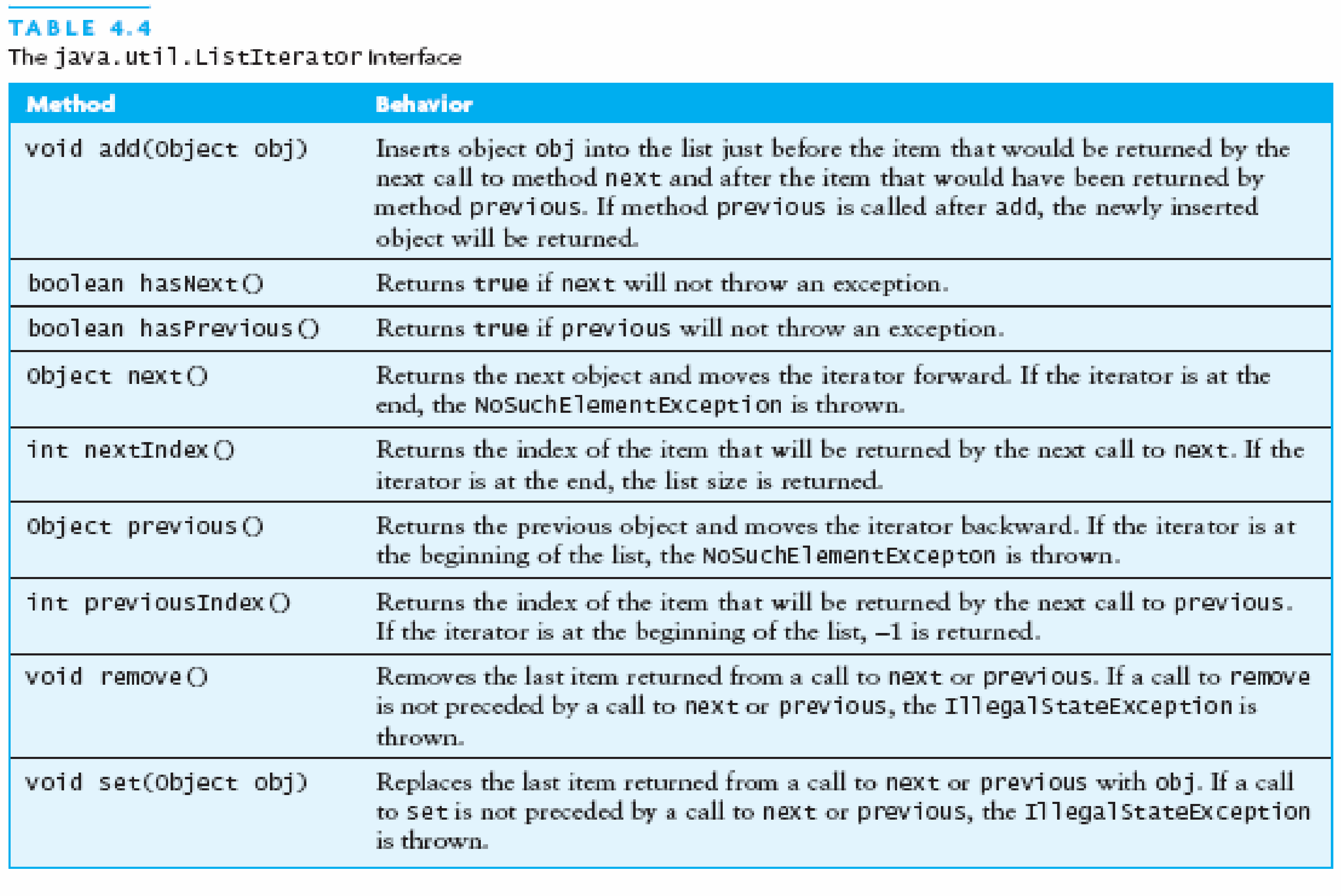


**TABLE 4.4**
The `java.util.ListIterator` Interface

| Method | Behavior |
|---|---|
| `void add(Object 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. |
| `Object 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. |
| `Object 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(Object 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. |

# Obtaining a `ListIterator`

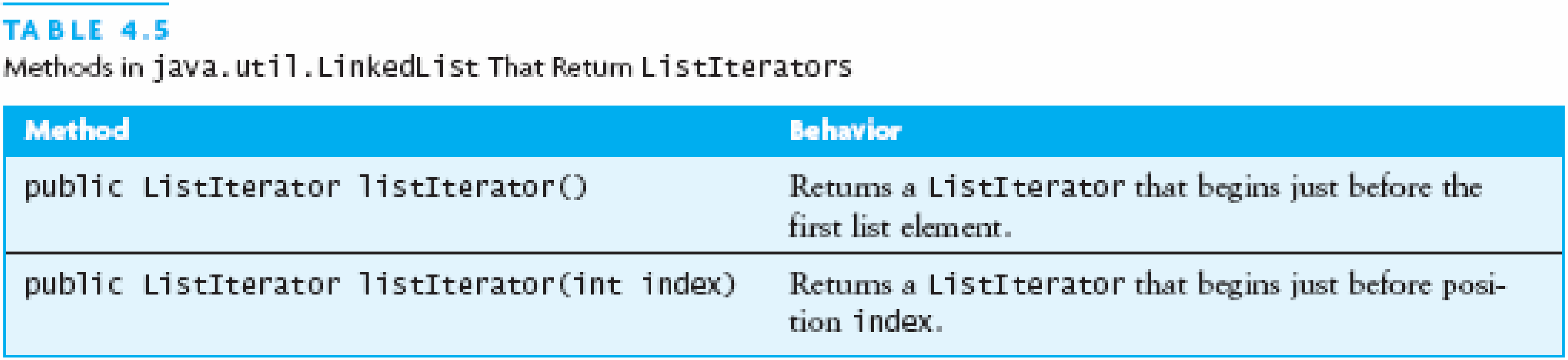


**TABLE 4.5**

Methods in `java.util.LinkedList` That Return `ListIterators`

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

# Comparison of `Iterator` and `ListIterator`

- `ListIterator` is a subinterface of `Iterator`
  - Classes that implement `ListIterator` provide all the capabilities of both
- `Iterator`:
  - Requires fewer methods
  - Can iterate over more general data structures
- `Iterator` required by the `Collection` interface
  - `ListIterator` required only by the `List` interface

# What `ListIterator` Adds

- Traversal in both directions
  - Methods **`hasPrevious()`**, **`previous()`**
  - Methods **`hasNext()`**, **`next()`**
- Obtaining next and previous index
- Modifications:
  - Method **`add(E)`** to add before “cursor” position
  - Method **`remove()`** to remove last returned
  - Method **`set(E)`** to set last returned

# Conversion Between **`ListIterator`** and Index

- **`ListIterator`**:
  - Method **`nextIndex()`** returns index of item to be returned by **`next()`**
  - Method **`previousIndex()`** returns index of item to be returned by **`previous()`**
- Class **`LinkedList`** has method **`listIterator(int index)`**
  - Returns a **`ListIterator`** positioned so **`next()`** will return item at position **`index`**

# One More Interface: **`Iterable`**

- Implemented by types providing a standard **`Iterator`**
- Allows use of Java 5.0 for-each loop

```
public interface Iterable<E> {
  Iterator<E> iterator();
}
```

# Implementing **DLList**

```
public class DLList<E> implements List<E> {
  private static class Node<E> {...}

  int size = 0;
  private Node<E> head;

  public DLList () {
    head = new this.Node<E>((E)null, null, null);
    head.next = head;
    head.prev = head;
  }
}
```

# Implementing **`DLList`** (2)

```
private static class Node<E> {
  private E data;
  private Node<E> prev;
  private Node<E> next;

  Node(E data, Node<E> prev, Node<E> next) {
    this.data = data;
    this.prev = prev;
    this.next = next;
  }
}
```

# Implementing **`DLList`** (3)

```
// internal helper method
private Node<E> addBefore (Node<E> n, E e) {
  Node<E> newNode = new Node<E>(e, n.prev, n);
  n.prev.next = newNode;
  n.prev      = newNode;
  ++size;
  return newNode;
}
```

# Implementing **`DLList`** (4)

```
// internal helper method
private E remove (Node<E> n) {
  if (n == head)
    throw new NoSuchElementException();
  E result = n.data;
  n.next.prev = n.prev;
  n.prev.next = n.next;
  n.prev = n.next = null;
  n.data = null;
  --size;
  return result;
}
```

# Implementing **`DLList`** (5)

```
// internal helper method
private Node<E> entry (int idx) {
  if (idx < 0 || idx >= size)
    throw new IndexOutOfBoundsException(...);
  Node<E> n = head;
  for (int i = 0; i <= idx; ++i)
    n = n.next;
  return n;
}
```

# Implementing **`DLList`** (6)

```
// internal helper method: smarter version
private Node<E> entry (int idx) {
  if (idx < 0 || idx >= size)
    throw new IndexOutOfBoundsException(...);
  Node<E> n = head;
  if (idx < (size / 2)) {  // forward, if closer
    for (int i = 0; i <= idx; ++i)
      n = n.next;
  } else {  // backward if that is closer
    for (int i = size; i > index; --i)
      n = n.prev;
  }
  return n;
}
```

# Implementing **`DLList`** (7)

```
public E getFirst () {
  if (size == 0)
    throw new NoSuchElementException();
  return head.next.data;
}

public E getLast () {
  if (size == 0)
    throw new NoSuchElementException();
  return head.prev.data;
}
```

# Implementing **DLList** (8)

```
public E removeFirst () {
  return remove(head.next);
}

public E removeLast () {
  return remove(head.prev);
}

public void addFirst (E e) {
  addBefore(head.next, e);
}

public void addLast (E e) {
  addBefore(head, e);
}
```

# Implementing **`DLList`** (9)

```
public E get (int idx) {
  return entry(idx).data;
}

public void set (int idx, E e) {
  Node<E> n = entry(idx);
  E result = n.data;
  n.data = e;
  return result;
}
```

# Implementing **`DLList`** (10)

```
public void add (int idx, E e) {
  addBefore((idx == size ? head : entry(idx),
            e);
}

public E remove (int idx) {
  return remove(entry(idx));
}
```

# Implementing **`DLList.LIter`**

```
public class DLList ... {

  private class LIter implement ListIterator<E> {
    // Note: not a static class
    // So: has implied reference to the DLList
    // Because of that, E is considered already
    //   defined, so do not write LIter<E> above

    private Node<E> lastReturned = head;
    private Node<E> nextNode;
    private nextIdx;
    ...
  }
}
```

# Implementing **DLList.LIter** (2)

```
// Constructor: returns LIter set to index idx
LIter (int idx) {
  if (idx < 0 || idx > size)
    throw new IndexOutOfBoundsException(...);
  nextNode = head.next;
  for (nextIdx = 0; nextIdx < idx; ++nextIdx) {
    nextNode = nextNode.next;
  }
  // can do same improvement as for entry(int)
}
```

# Implementing **`DLList.LIter`** (3)

```
public boolean hasNext () {
  return (nextIdx < size);
}

public E next () {
  if (nextIdx >= size)
    throw new NoSuchElementException();
  lastReturned = nextNode;
  nextNode = nextNode.next;
  ++nextIdx;
  return lastReturned.data;
}

public int nextIndex () { return nextIdx; }
```

# Implementing **`DLList.LIter`** (4)

```
public boolean hasPrevious () {
  return (nextIdx > 0);
}

public E previous () {
  if (nextIdx <= 0)
    throw new NoSuchElementException();
  lastReturned = nextNode = nextNode.prev;
  --nextIdx;
  return lastReturned.data;
}

public int previousIndex () { return nextIdx-1; }
```

# Implementing **`DLList.LIter`** (5)

```
public void set (E e) {
  if (lastReturned == head)
    throw new IllegalStateException();
  lastReturned.data = e;
}

public add (E e) {
  lastReturned = head;
  addBefore(nextNode, e);
  ++nextIdx;
}
```

# Implementing **`DLList.LIter`** (6)

```
public void remove () { // remove last returned
  Node<E> lastNext = lastReturned.next;
  try {
    LinkedList.this.remove(lastReturned);
  } catch (NoSuchElementException e) {
    throw new IllegalStateException();
  }
  if (nextNode == lastReturned)
    nextNode = lastNext;  // forward case
  else
    --nextIdx;            // backward case
  lastReturned = head;
}
```

# An Application: *Ordered* Lists

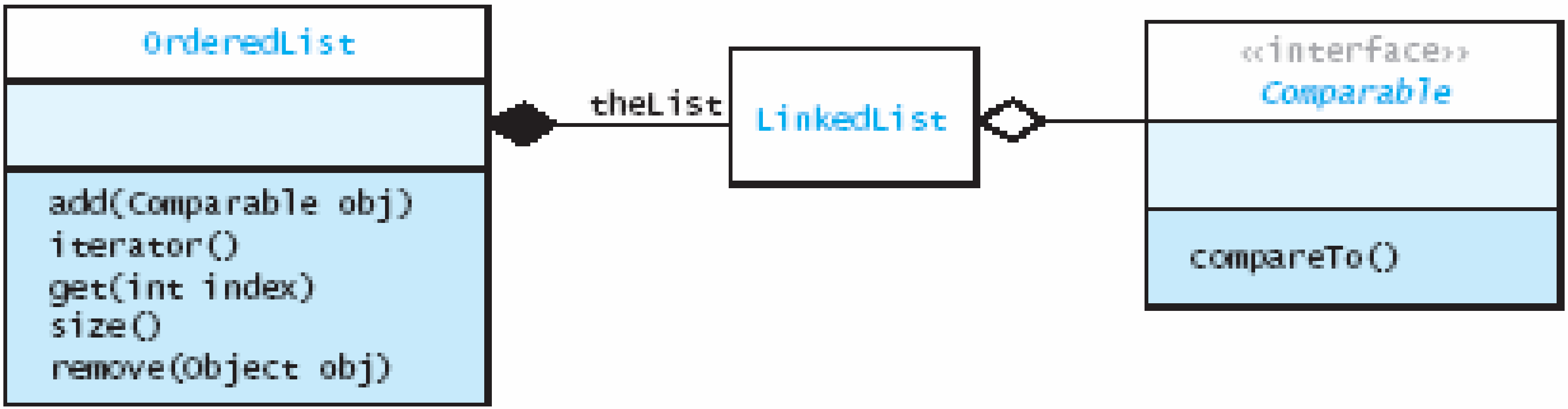
- Want to maintain a list of names
- Want them in alphabetical order at all times
- *Approach:* Develop an **`OrderedList`** class
  - For reuse, good if can work with other types:

```
public interface Comparable<E> {
  int compareTo(E e);
  // <0 if this <  e
  //  0 if this == e
  // >0 if this >  e
}
```

# Class Diagram for Ordered Lists (old)

**FIGURE 4.31**

`OrderedList` Class Diagram



# Skeleton of **`OrderedList`**

```
import java.util.*;
public class
      OrderedList<E extends Comparable<E>>
    implements Iterable<E> {
  private LinkedList<E> lst =
    new LinkedList<E>();
  public void add (E e) {...}
  public E get (int idx) {...}
  public int size() {...}
  public E remove (E e) {...}
  public Iterator iterator() {...}
}
```

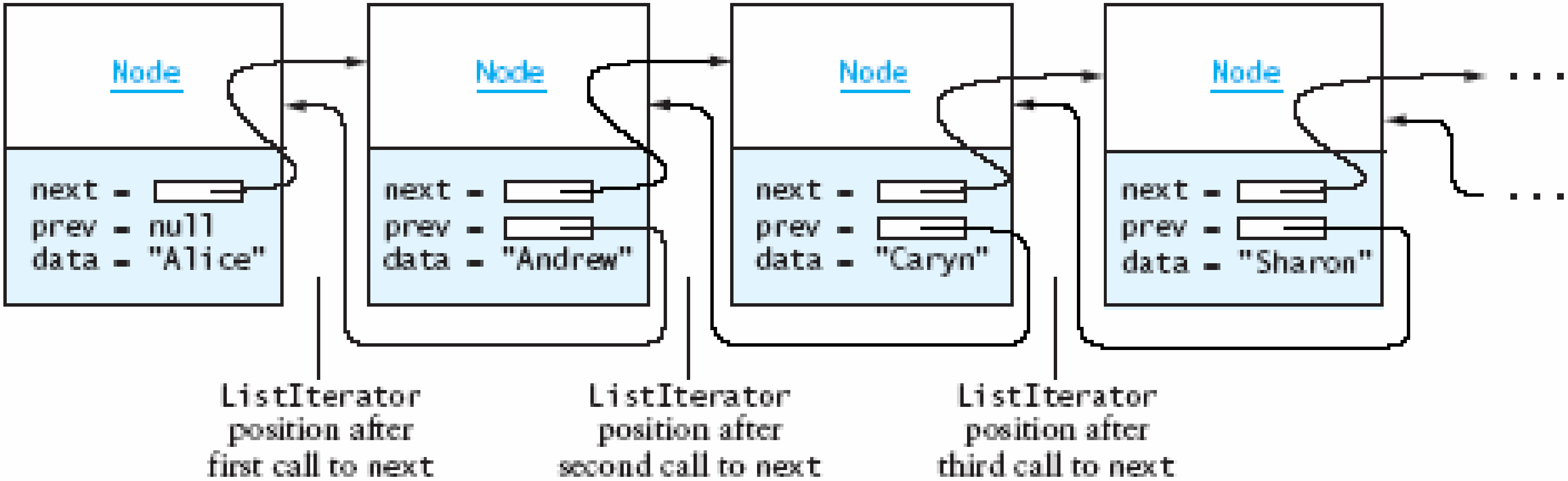
# Inserting Into an **`OrderedList`**

- *Strategy* for inserting new element e:
  - Find first element > e
  - Insert e before that element
- Two cases:
  - No such element: e goes at the end of the list
  - Element e2 > e:
    - Iterator will be positioned after e2, so ...
    - Back up by one and insert e

# Inserting Diagrammed

**FIGURE 4.32**

Inserting `"Bill"` before `"Caryn"` in an Ordered List



Should insert new element here

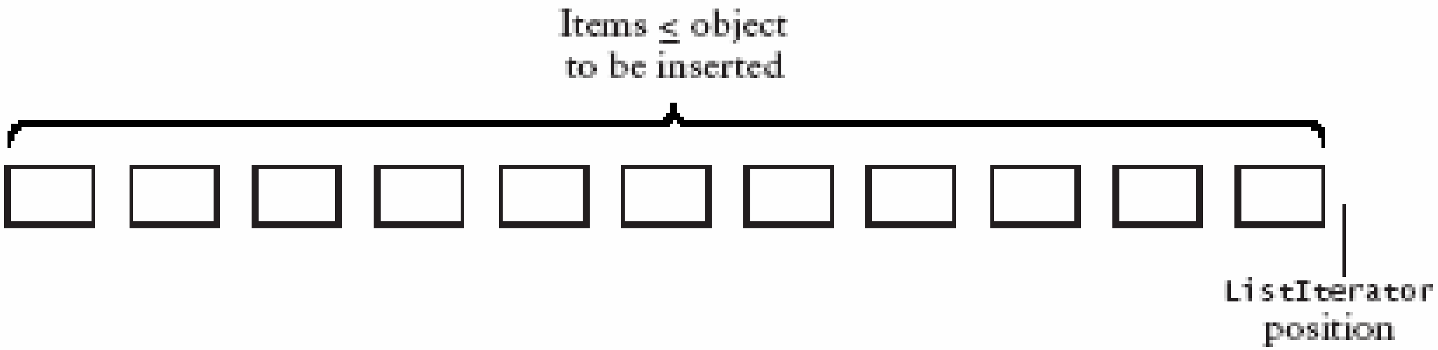
Iterator will be positioned here

# Inserting Diagrammed (2)

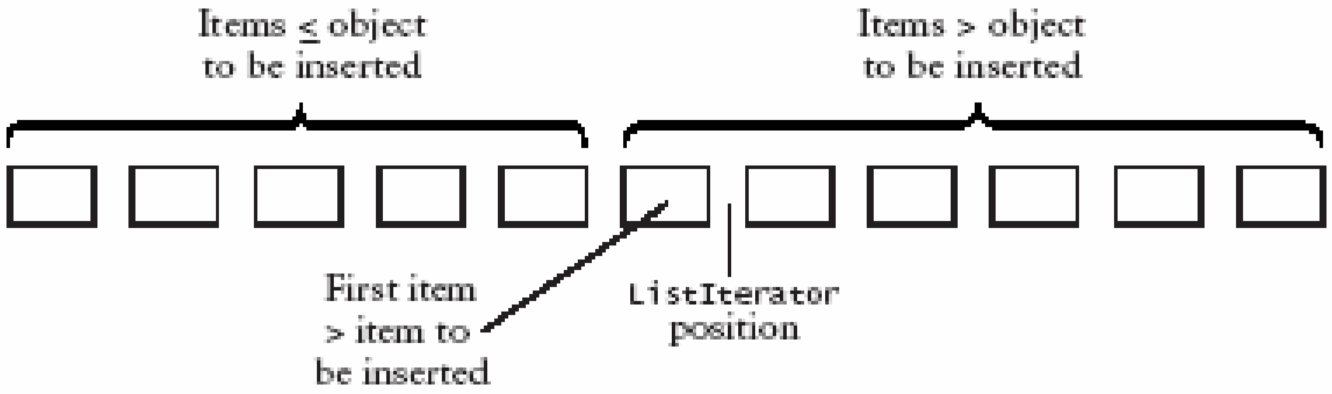
**FIGURE 4.33**

Attempted Insertion into an Ordered List

Case 1: Inserting at the end of a list



Case 2: Inserting in the middle of a list



# OrderedList.add

```
public void add (E e) {
  ListIterator<E> iter =
    lst.listIterator();
  while (iter.hasNext()) {
    if (e.compareTo(iter.next()) < 0) {
      // found element > new one
      iter.previous();  // back up by one
      iter.add(e);      // add new one
      return;           // done
  }
  iter.add(e);  // will add at end
}
```

# `OrderedList.add` (variant)

```
public void add (E e) {
  ListIterator<E> iter =
    lst.listIterator();
  while (iter.hasNext()) {
    if (e.compareTo(iter.next()) < 0) {
      // found element > new one
      iter.previous();  // back up by one
      break;
  }
  iter.add(e);  // add where iterator is
}
```

# Other Methods Can Use Delegation

```
public E get (int idx) {
  return lst.get(idx);
}
public int size () {
  return lst.size();
}
public E remove (E e) {
  return lst.remove(e);
}
public Iterator iterator() {
  return lst.iterator();
}
```

# Testing **`OrderedList`**

```
OrderList<Integer> test =
  new OrderedList<Integer>
// Fill with randomly chosen integers
Random rand = new Random();
for (int i = 0; i < START_SIZE; ++i) {
  int val = random.nextInt(MAX_INT);
  // 0 <= val < MAX_INT
  test.add(val);
}
test.add(-1);  // adds at beginning
test.add(MAX_INT);  // adds at end
printAndCheck(test);
```

# Testing **`OrderedList`** (2)

```
public static void printAndCheck (
    OrderedList<Integer> test) {
  int prev = test.get(0);
  for (int thisOne : test) {
    System.out.println(thisOne);
    if (prev > thisOne)
      System.out.println(
        “***FAILED, value is “ + thisOne);
    prev = thisOne;
  }
}
```

# Testing **`OrderedList`** (3)

```
// some remove tests:
Integer first = test.get(0);
Integer last  = test.get(test.size()-1);
Integer mid   = test.get(test.size()/2);
test.remove(first);
test.remove(last);
test.remove(mid);
printAndCheck(test);
```

# Lists That “Expose” the **Node** Objects

- Suppose we have **E** objects and lists such that:
  - Each **E** object is on one list (or none)
  - We wish to move **E** objects between lists
- Not efficient to search for item in list
  - Need to iterate down list to find element
- May wish to reduce allocation of **Node** objects
- *Idea:* Allow access to **Node** for an **E** object
- *But:* Want this still to be safe

# API of Lists That “Expose” the **Node** Objects

```
public class DLList<E> {
  // class exposed
  public static class Node<E> {
    public E data;  // can expose this
    private Node<E> prev;  // protect!
    private Node<E> next;  // protect!

    private Node<E> () { } // protect!
    private Node<E> (E data,  // protect!
      Node<E> prev, Node<E> next) {...}
  }
}
```

# API of Lists That “Expose” **Node** Objects (2)

```
public class DLList<E> {
  // methods to return a Node
  public Node<E> addNode (int idx, E e) {
    ... }

  public Node<E> addNode (E) {...}

  public Node<E> getNode (int idx) {...}
}
```

# API of Lists That “Expose” **Node** Objects (3)

```
public class DLList<E> {
  // methods that use a Node
  public void add (int idx, Node<E> n) {
    ... }

  public Node<E> add (Node<E> n) {...}

  public static void remove (Node<E> n) {
    ...}
}
```

# Using Lists That “Expose” **Node** Objects

```
Student[] students =
  new Student[nStudents];

DLList<Student>[] group =
  (DLList<Student>[]) new DLList[nGroups];

for (int i = 0; i < nGroups; ++i)
  group[i] = new DLList<Student>();

Node<Student>[] nodes =
  (Node<Student>[]) new Node[nStudents];
```

# Using Lists That “Expose” **Node** Objects (2)

```
for (int j = 0; j < nStudents; ++j) {
  Student s = students[j];
  int g = s.getGroup();
  Node<Student> n = groups[g].addNode(s);
  nodes[j] = n;
}

// change group of student j to newGrp
Node<Student> n = nodes[j];
DLList<Student>.remove(n);
groups[newGrp].add(n);
students[j].setGroup(newGrp);
```

# The **`Collection`** Hierarchy

***`Collection<E>`*** interface, root of the hierarchy

implements ***`Iterable<E>`***

**`AbstractCollection<E>`**

abstract class, holds some shared methods

***`List<E>`*** interface, root of **`List`** hierarchy

***`Set<E>`*** interface, root of **`Set`** hierarchy

***`Queue<E>`*** interface

# The **`List`** Hierarchy: Access By Index

***`List<E>`***

- **`AbstractList<E>`**
  - abstract class, extends **`AbstractCollection<E>`**
  - **`ArrayList<E>`**
  - **`Vector<E>`**
    - **`Stack<E>`**
  - **`AbstractSequentialList<E>`** abstract class
    - **`LinkedList<E>`** implements ***`Queue<E>`***

# The **`Set`** Hierarchy: Access By Element

***`Set<E>`***

- **`AbstractSet<E>`**
  - abstract class, extends **`AbstractCollection<E>`**
  - **`HashSet<E>`** **`contains(E)`** is fast (on average)
    - **`LinkedHashSet<E>`** iterate in insertion order
  - **`TreeSet<E>`** implements **`SortedSet<E>`**
  - ***`SortedSet<E>`*** interface, maintains **`E`** order

# The Collection Hierarchy (old)

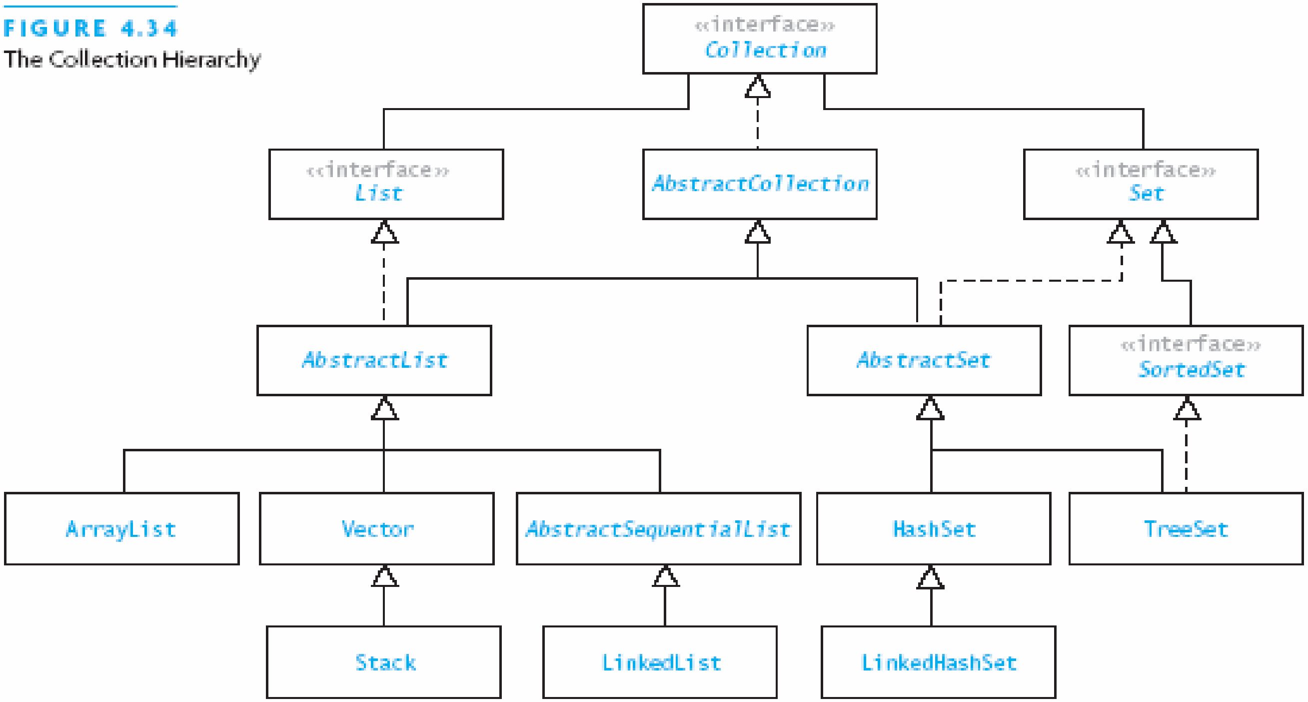


**FIGURE 4.34**
The Collection Hierarchy

# Common Features of Collections

- **`Collection`** specifies a set of common methods
- Fundamental features include:
  - Collections *grow* as needed
  - Collections hold *references to objects*
  - Collections have at least two constructors
    - Create an empty collection of that kind
    - Create a copy of another collection

# Common Features of Collections

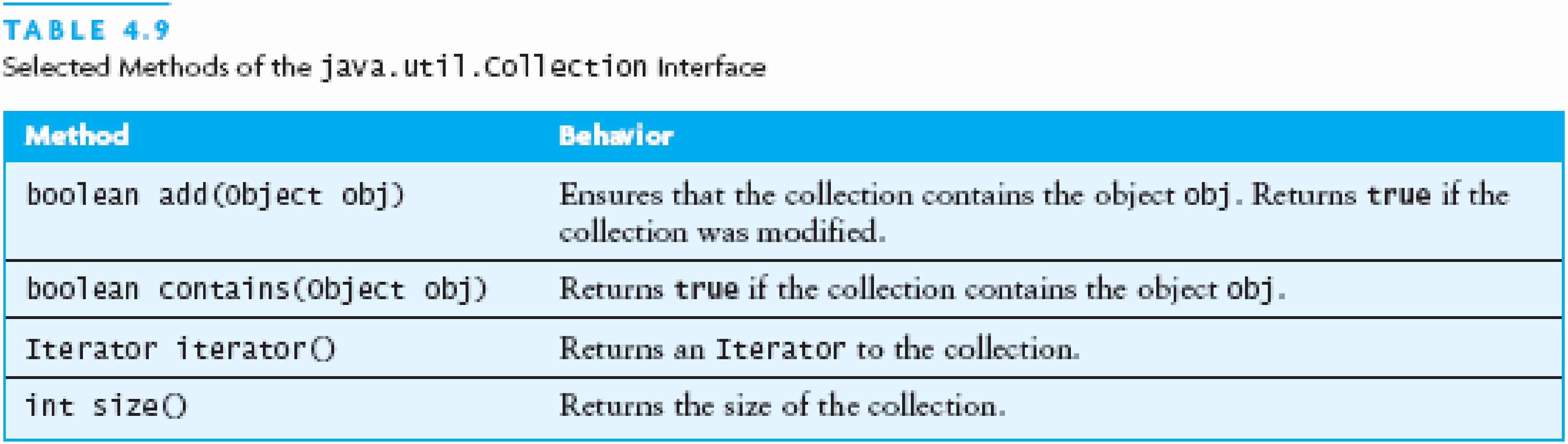


**TABLE 4.9**

Selected Methods of the `java.util.Collection` Interface

| Method | Behavior |
|---|---|
| `boolean add(Object obj)` | Ensures that the collection contains the object `obj`. Returns `true` if the collection was modified. |
| `boolean contains(Object obj)` | Returns `true` if the collection contains the object `obj`. |
| `Iterator iterator()` | Returns an `Iterator` to the collection. |
| `int size()` | Returns the size of the collection. |

# Implementing a Subclass of `Collection<E>`

- Extend **`AbstractCollection<E>`**
- It implements most operations
- You need to implement only:
  - **`add(E)`**
  - **`size()`**
  - **`iterator()`**
  - An inner class that implements **`Iterator<E`**>

# Implementing a Subclass of **`List<E>`**

- Extend **`AbstractList<E>`**
- You need to implement only:
  - **`add(int, E)`**
  - **`get(int)`**
  - **`remove(int)`**
  - **`set(int E)`**
  - **`size()`**
- It implements **`Iterator<E`**> using the index

# Implementing a Subclass of **`List<E>`** (2)

- Extend **`AbstractSequentialList<E>`**
- You need to implement only:
  - **`listIterator()`**
  - **`size()`**
  - An inner class implementing **`ListIterator<E>`**