

## **Chapter 4**

# **Lists and the Collection Interface**

## Arrays: properties

- An array is an ordered, indexed structure
- Arbitrary elements can be selected using their subscript value in constant time
- Elements may be accessed in sequence using a loop
- Arrays store primitive data types

# Arrays: limitations

We cannot:

- Increase or decrease the length of an array
- Add an elements without shifting the other elements to make room
- Remove elements without shifting other elements to fill in the resulting gap

# List Interface

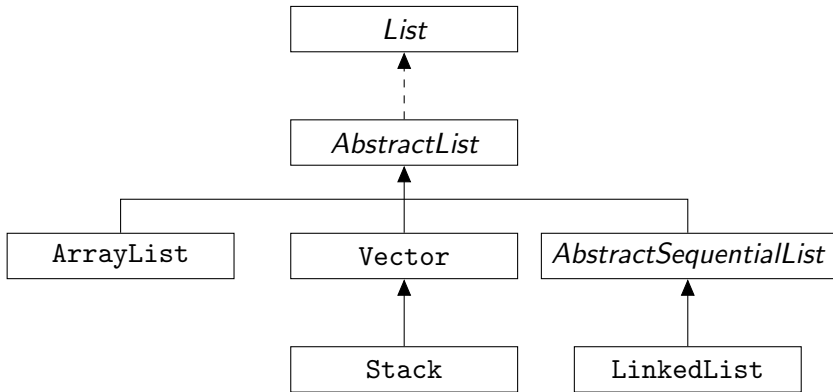
Methods in the **List** interface include:

- Finding a specified target
- Adding an element to either end
- Removing an item from either end
- Traversing the list structure without a subscript

Not all classes perform the allowed operations with the same degree of efficiency.

Lists store references to Objects (**Autoboxing** facilitates this).

# List Interface



# ArrayList

`ArrayList` is a simple class that implements the `List` interface.

Has more features than an array.

`ArrayLists` are often used when we want to add new elements to the end of a list, but also need to be able to access the elements stored in the list in arbitrary order.

Generics allow the definition of a collection that contains references to objects of a specific type.

```
List<String> myList = new ArrayList<String>();
```

Only references to objects of type `String` can be stored in `myList`, and all items retrieved are of type `String`.

# ArrayList

```
myList.add("Bashful");  
myList.add("Awful");  
myList.add("Jumpy");  
myList.add("Happy");
```

index: 0	1	2	3
"Bashful"	"Awful"	"Jumpy"	"Happy"

```
myList.add("Doc",2);
```

index: 0	1	2	3	4
"Bashful"	"Awful"	"Doc"	"Jumpy"	"Happy"

```
myList.remove(1);
```

index: 0	1	2	3
"Bashful"	"Doc"	"Jumpy"	"Happy"

# An Implementation of ArrayList

The text develops an implementation of a `ArrayList` class called `KWArrayList`.

The elements of the `ArrayList` are stored in an array named `theData`.

The physical size of the array is stored in a data field named `capacity`.

The number of elements is stored in a data field named `size`.

How can we implement insertion?

How can we implement deletion?

# Performance of ArrayList

`set` and `get` methods execute in `constant` time.

Inserting or removing elements is `linear` time.

The initial release of the Java API contained a `Vector` class which has similar functionality to `ArrayList`.

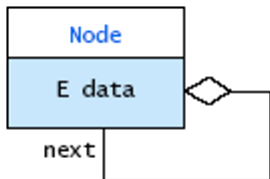
New applications should use `ArrayList` rather than `Vector`.

# Linked Lists

With the appropriate reference, linked lists can add or remove items anywhere in the list in constant time.

Each element (node) in a linked list stores information and a link to the next, and optionally previous, node

# List Nodes



A **Node** contains a data item and one or more links.

In Java, a link is a reference to a **Node**.

In Java, nodes are generally defined inside of the class that uses them (i.e. **SingleLinkedList**).

Classes defined inside other classes are called **inner classes**.

Inner classes are known only within the scope of the enclosing outer classes (unless **static**). Inner classes also have access to all of the members of the outer classes.

# A Node Class

```
private static class Node<E> {  
    // Data and link fields  
    private E data;  
    private Node next;  
  
    // Constructors  
    private Node(E dataItem) {  
        data = dataItem;  
        next = null;  
    }  
  
    private Node(E dataItem, Node<E> nodeRef) {  
        data = dataItem;  
        next = nodeRef;  
    }  
}
```

## Creating and Connecting Nodes

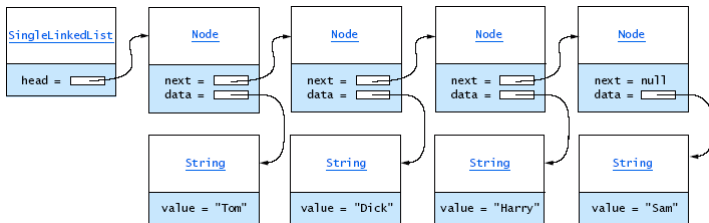
```
// Create four Nodes
Node<String> tom = new Node<String>("Tom");
Node<String> dick = new Node<String>("Dick");
Node<String> harry = new Node<String>("Harry");
Node<String> sam = new Node<String>("Sam");

// Link them together
tom.next = dick;
dick.next = harry;
harry.next = sam;
```

Trace the code...

# A Linked List

- Storing references to each **Node** is wasteful of memory
- Only one reference is actually needed: the **head**



# A Single-Linked List Class

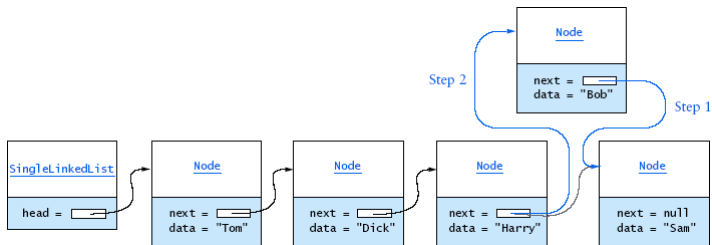
```
public class SingleLinkedList<E> {  
  
    private Node<E> head = null;  
  
    ...  
  
    // Add an item to the front of the list  
    public void addFirst(E item) {  
        head = new Node<E>(item, head);  
    }  
}
```

---

```
SingleLinkedList<String> names = new SingleLinkedList<String>();  
names.addFirst("Sam");  
names.addFirst("Harry");  
names.addFirst("Dick");  
names.addFirst("Tom");
```

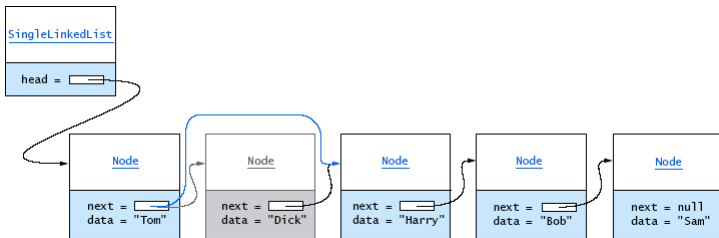
# Insertion in a Linked List

```
Node<String> bob = new Node<String>("Bob");  
bob.next = harry.next; // Step 1  
harry.next = bob;     // Step 2
```



# Deletion in a Linked List

```
tom.next = tom.next.next;
```



## Traversing a Linked List: pseudocode

```
Set nodeRef to reference the first node {  
while nodeRef is not null  
    // Visit the current Node  
    process the node referenced by nodeRef  
  
    // Advance by one node  
    nodeRef = nodeRef.next;  
}
```

# Properties of Single-Linked Lists

Insertion or deletion after a “known” reference is **constant** time.

Insertion or deletion at an arbitrary positions without a reference is **linear** time.

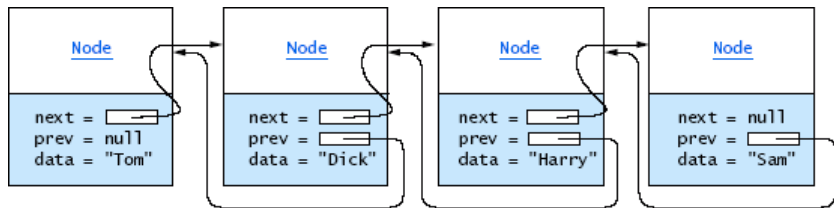
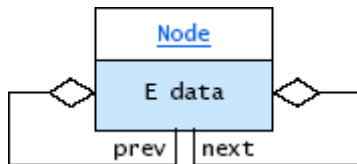
Can insert a node only after a referenced node.

Can remove a node only if we have a reference to its predecessor node.

Can traverse the list only in the forward direction.

Some of these limitations can be addressed by adding a reference in each node to the previous node.

# Double-Linked List



## A Revised Node Class

```
private static class Node<E> {
    // Data and link fields
    private E data;
    private Node next = null; // link to successor
    private Node prev = null; // link to predecessor

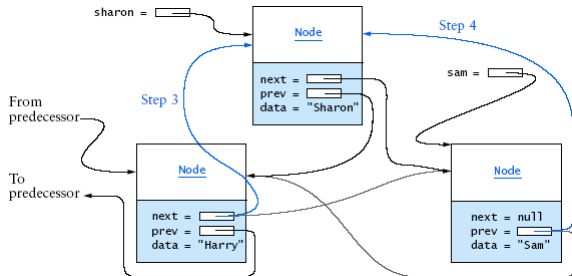
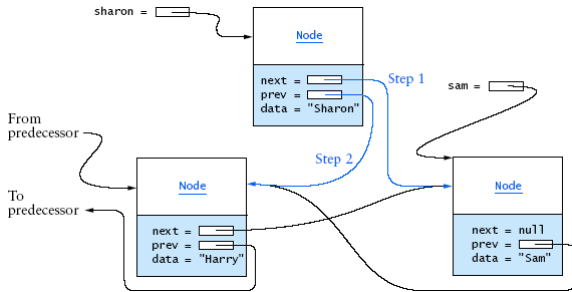
    // Constructor
    private Node(E dataItem) {
        data = dataItem;
    }
}
```

## Inserting a Node in a Double-Linked List

```
Node<String> sharon = new Node<String>("Sharon");  
  
sharon.next = sam;           // Step 1  
  
sharon.prev = sam.prev;     // Step 2  
  
sam.prev.next = sharon;     // Step 3  
  
sam.prev = sharon;          // Step 4
```

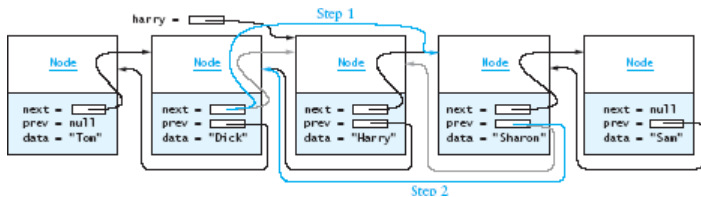
# Insertions in a Double Linked List

## Inserting after "Harry"



# Deletion in a Double Linked List

```
harry.prev.next = harry.next; // Step 1  
harry.next.prev = harry.prev; // Step 2
```



## Variations on a theme

- We need a *head* pointer to gain access to all the Nodes in a list
- A *tail* pointer may also be added
- An integer *size* to record the number of Nodes may be useful
- We may also form and maintain a *circular* list

# Circular Lists

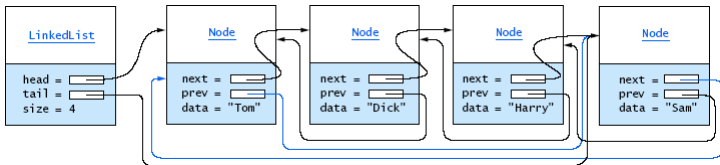
Circular linked lists link the last node of a double linked list to the first node and the first to the last.

You can traverse in forward or reverse direction even after you have passed the last or first node.

You can visit all the list elements from any starting point (never fall off the end of a list again).

However it can be easy to write an infinite loop over a circular list.

# Circular Lists



# LinkedList

`LinkedList` is part of the Java API and implements the `List` interface using a double-linked list.

Method	Behavior
<code>void add(int index, E obj)</code>	Inserts <code>obj</code> into the list at position <code>index</code>
<code>void addFirst(E obj)</code>	Inserts <code>obj</code> at the front of the list
<code>void addLast(E obj)</code>	Inserts <code>obj</code> at the at the end of the list
<code>E get(int index)</code>	Returns the element at position <code>index</code>
<code>E getFirst()</code>	Returns the first element in the list
<code>E getLast()</code>	Returns the last element in the list
<code>boolean remove(E obj)</code>	Removes the first occurrence of <code>obj</code> from the list
<code>int size()</code>	Returns the number of elements in the list

## Traversing a List — Naive Approach

Suppose `myList` is a list object

```
// Access each list element
for (int i = 0; i < myList.size(); i++) {
    E nextElement = myList.get(i);

    // Process nextElement in some way
    ...
}
```

- When `myList.get(i)` is invoked,  $i + 1$  references will be inspected, in order to “walk” down the list to the  $i$ th node
- Thus, traversing a list of size  $n$  makes

$$1 + 2 + 3 + \dots + n = \mathcal{O}(n^2)\text{references}$$

- Processing  $n$  elements in  $\mathcal{O}(n^2)$  time is very inefficient

We need a better way!

# Using an Iterator

```
// Obtain an Iterator for the list
Iterator<E> itr = myList.iterator();

// Access each list element
while (itr.hasNext()) {
    E nextElement = itr.next();

    // Process nextElement in some way
    ...
}
```

- When `itr.next()` is invoked, only  $\mathcal{O}(1)$  time is required
- Thus, traversing a list of size  $n$  takes

$$\underbrace{k + k + k + \dots + k}_{n \text{ terms}} = \mathcal{O}(n) \text{ time}$$

- Processing  $n$  elements in  $\mathcal{O}(n)$  time is efficient

# The Iterator Interface

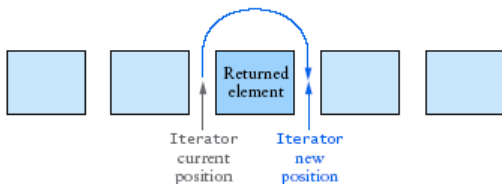
The interface `Iterator` is defined as part of `java.util`.

The `List` interface declares the method `iterator()`, which returns an `Iterator` object that will iterate over the elements of that list.

An `Iterator` does not refer to or point to a particular node, but points between nodes.

# The Iterator Interface

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



## Another Example

```
/** Remove all the even values from a list.  
    Pre: LinkedList myList contains Integer values.  
    Post: All even values in myList have been removed.  
*/  
public static void removeEvens(LinkedList<Integer> myList) {  
  
    // Create an Iterator  
    Iterator<Integer> itr = myList.iterator();  
  
    // Traverse the list, removing any even values  
    while (itr.hasNext()) {  
        int nextInt = itr.next();  
  
        if (nextInt % 2 == 0)  
            itr.remove();  
    }  
}
```

# The ListIterator Interface

**Iterators** have a few limitations:

- Can only traverse the List in the forward direction
- Provides only a **remove** method; there is no **add** method
- Must advance an iterator using your own loop if starting position is not at the beginning of the list

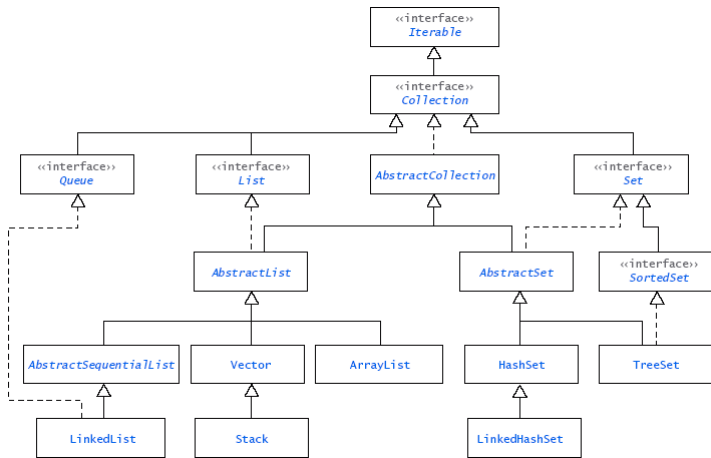
**ListIterator** is an extension of the **Iterator** interface that overcomes these limitations.

# The ListIterator Interface

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

# The Collection Hierarchy

Both `ArrayList` and `LinkedList` represent a collection of objects



The `Collection` interface specifies common methods

Fundamental features include:

- Collections can grow as needed
- Collections can hold references to objects
- Collections have at least two constructors

TABLE 4.9

Selected Methods of the `java.util.Collection<E>` Interface

Method	Behavior
<code>boolean add(E obj)</code>	Ensures that the collection contains the object <code>obj</code> . Returns <code>true</code> if the collection was modified.
<code>boolean contains(E obj)</code>	Returns <code>true</code> if the collection contains the object <code>obj</code> .
<code>Iterator&lt;E&gt; iterator()</code>	Returns an <code>Iterator</code> to the collection.
<code>int size()</code>	Returns the size of the collection.