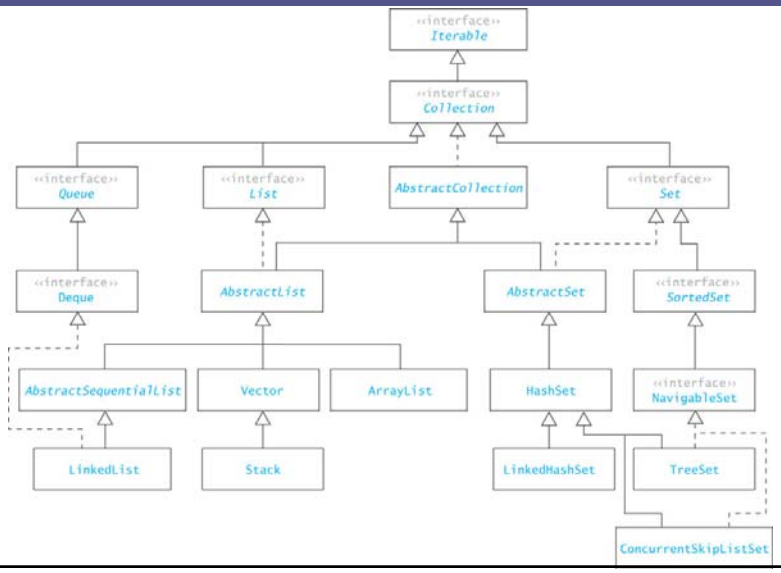
CHAPTER 2

Lists and the `Collections` Framework

The `Collection` Framework



Java Collections Interface

3

Key Basic Methods	
Modifier and Type	Method and Description
boolean	add(E e) Ensures that this collection contains the specified element (optional operation).
boolean	remove(Object o) Removes a single instance of the specified element from this collection, if it is present (optional operation).
boolean	contains(Object o) Returns true if this collection contains the specified element.
boolean	isEmpty() Returns true if this collection contains no elements.
int	size() Returns the number of elements in this collection.
boolean	equals(Object o) Compares the specified object with this collection for equality.
void	clear() Removes all of the elements from this collection (optional operation).

LIST Interface (Ordered Collection)

4

boolean	**add(E e)** **Appends the specified element to the end of this list (optional operation).**
void	add(int index, E element) Inserts the specified element at the specified position in this list (optional operation).
boolean	remove(Object o) Removes the first occurrence of the specified element from this list, if it is present (optional operation).
E	remove(int index) Removes the element at the specified position in this list (optional operation).
boolean	contains(Object o) Returns true if this list contains the specified element.
E	get(int index) Returns the element at the specified position in this list.

E	**set(int index, E element)** **Replaces the element at the specified position in this list with the specified element (optional operation).**
int	indexOf(Object o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
int	lastIndexOf(Object o) Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	isEmpty() Returns true if this list contains no elements.
int	size() Returns the number of elements in this list.
boolean	equals(Object o) Compares the specified object with this list for equality.
void	clear() Removes all of the elements from this list (optional operation).

Java.util.ArrayList<E>

5

boolean	**add(E e)** **Appends the specified element to the end of this list (optional operation).**
void	add(int index, E element) Inserts the specified element at the specified position in this list (optional operation).
boolean	remove(Object o) Removes the first occurrence of the specified element from this list, if it is present (optional operation).
E	remove(int index) Removes the element at the specified position in this list (optional operation).
boolean	contains(Object o) Returns true if this list contains the specified element.
E	get(int index) Returns the element at the specified position in this list.

E	**set(int index, E element)** **Replaces the element at the specified position in this list with the specified element (optional operation).**
int	indexOf(Object o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
int	lastIndexOf(Object o) Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	isEmpty() Returns true if this list contains no elements.
int	size() Returns the number of elements in this list.
boolean	equals(Object o) Compares the specified object with this list for equality.
void	clear() Removes all of the elements from this list (optional operation).

MyListInterface

6

```
public interface MyListInterface {
        public boolean add(Object item);
        public boolean add(int index, Object item);

        public E remove(int index);
        public E set(int index, Object item);
        public E get(int index);

        public boolean contains (Object item);

        public boolean isEmpty();
        public void clear();
        public int size();

} // interface MyListInterface
```

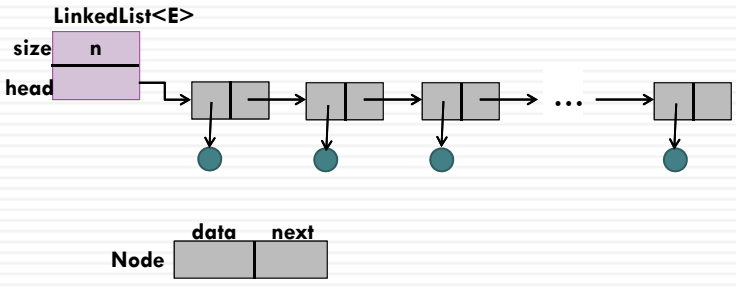
Comparing Performance

7

	Fixed Size Array	Dynamic Array
add(o)	O(1)	O(n)
add(i, o)	O(n)	O(n)
remove(i)	O(n)	O(n)
set(i, o)	O(1)	O(1)
get(i)	O(1)	O(1)
contains(o)	O(n)	O(n)
clear(), size(), isEmpty()	O(1)	O(1)

Going from static to dynamic array gives the flexibility of size but makes worst-case add(o) complexity O(n) because, each time the array is full, you need to expand, copy, and then add. Can we improve? Let's see another implementation: Linked Lists.

Single-Linked Lists



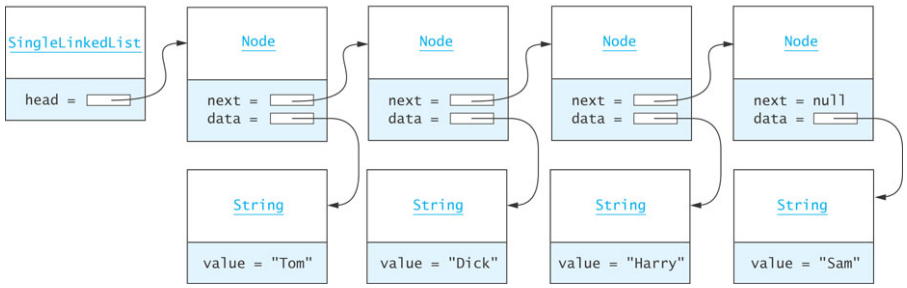
Single-Linked Lists

- A linked list is useful for inserting and removing at arbitrary locations
- The `ArrayList` is limited because its `add` and `remove` methods operate in linear (O(*n*)) time—requiring a loop to shift elements
- A linked list can add and remove elements at a known location in O(1) time
- In a linked list, instead of an index, each element is linked to the following element

A List Node

- A node can contain:
  - a data item
  - one or more links
- A link is a reference to a list node
- In our structure, the node contains a data field named `data` of type `E`
- and a reference to the next node, named `next`

Connecting Nodes



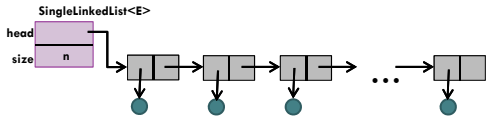
Connecting Nodes (cont.)

```
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");

tom.next = dick;
dick.next = harry;
harry.next = sam;
```

A Single-Linked List Class

A `SingleLinkedList` object has a data field `head`, the *list head*, which references the first list node. And another data field `size` – the number of entries/nodes in the list.




```
public class SingleLinkedList<E> {
  private Node<E> head = null;
  private int size = 0;
  ...
  private static class Node<E> {
  …
  } // class Node
} // class SingleLinkedList<E>
```

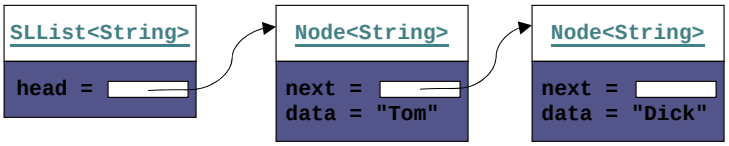
`SingleLinkedList` Class

Method	Behavior
`public E get(int index)`	Returns a reference to the element at position `index`.
`public E set(int index, E anEntry)`	Sets the element at position `index` to reference `anEntry`. Returns the previous value.
`public int size()`	Gets the current size of the `List`.
`public boolean add(E anEntry)`	Adds a reference to `anEntry` at the end of the `List`. Always returns `true`.
`public void add(int index, E anEntry)`	Adds a reference to `anEntry`, inserting it before the item at position `index`.
`int indexOf(E target)`	Searches for `target` and returns the position of the first occurrence, or –1 if it is not in the `List`.

Also need (in addition to constructors, and print method):

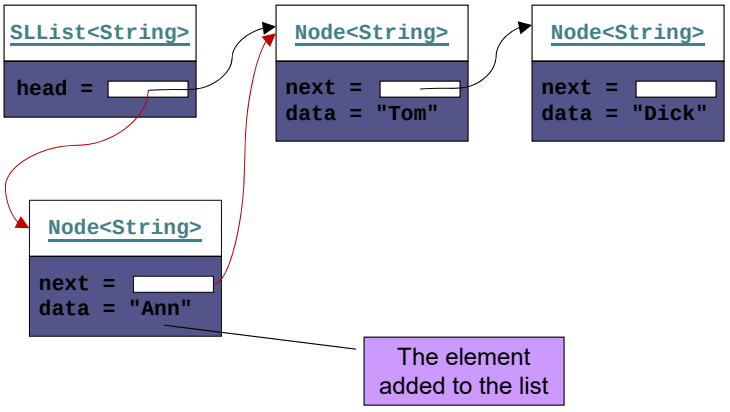
```
private void addFirst(Node n)
private void addAfter(Node n, E item)
private void removeFirst()
private E removeAfter(Node n)
```

SLList: An Example List




```
SingleLinkedList<String> SLList = new SingleLinkedList<String();
SLList.add(new Node("Tom"));
SLList.add(New Node("Dick"));
SLList.add(0, New Node("Ann"));
```

Implementing SLList.addFirst(E item)



Implementing `SLList.addFirst(E item)` (cont.)

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

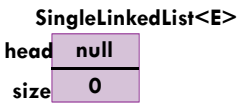
or, more simply …

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

This works even if `head` is null

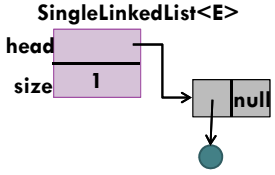
Implementing `SLList.addFirst(E item)` (cont.)

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



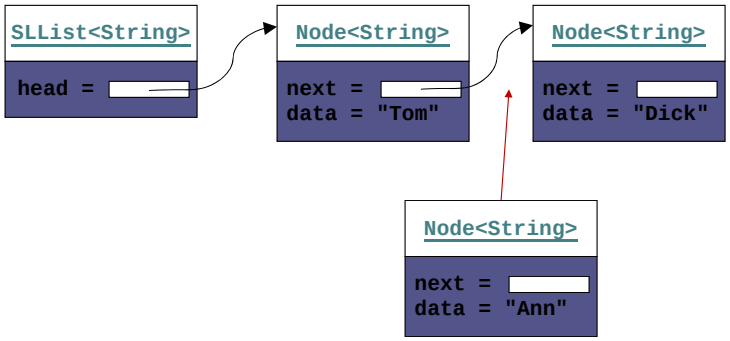
or, more simply …

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

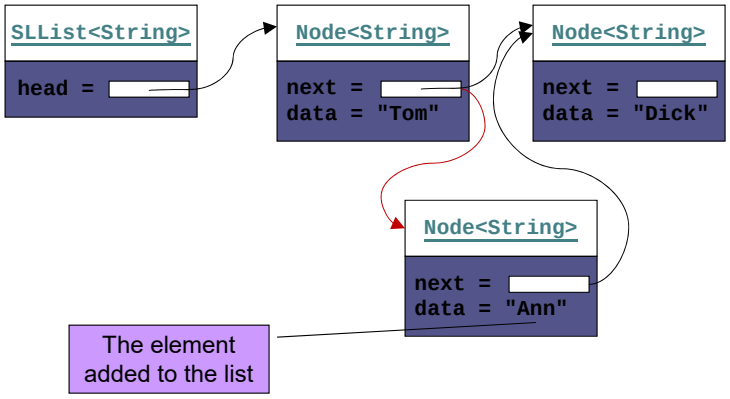


This works even if `head` is null

Implementing `addAfter(Node<E> node, E item)`



Implementing `addAfter(Node<E> node, E item)`



Implementing `addAfter(Node<E> node, E item)` (cont.)

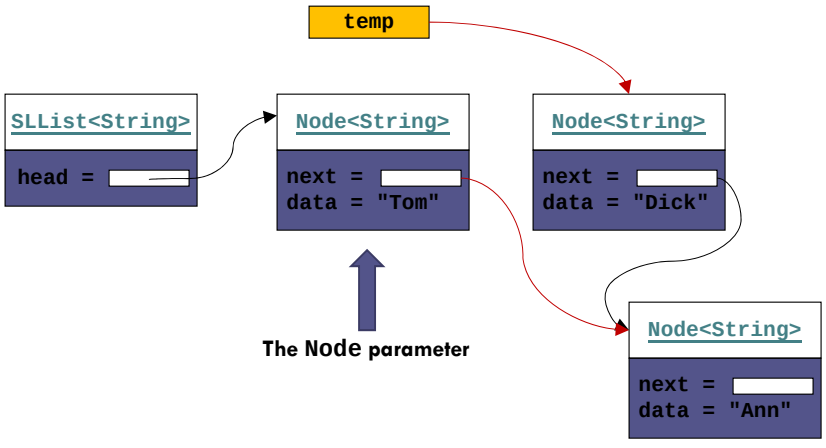
```
private void addAfter (Node<E> node, E item) {
  Node<E> temp = new Node<E>(item, node.next);
  node.next = temp;
  size++;
}
```

We declare this method `private` since it should not be called from outside the class. Later we will see how this method is used to implement the public add methods.

or, more simply ...

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

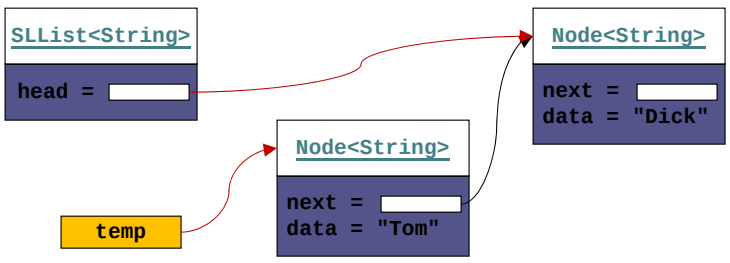
Implementing `removeAfter(Node<E> node)`



Implementing `removeAfter(Node<E> node)` (cont.)

```
private E removeAfter (Node<E> node) {
  Node<E> temp = node.next;
  if (temp != null) {
    node.next = temp.next;
    size--;
    return temp.data;
  } else {
    return null;
  }
} // removeAfter()
```

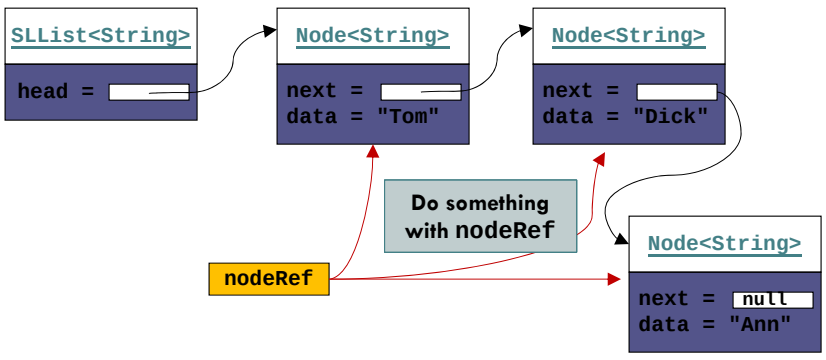
Implementing `SLList.removeFirst()`



Implementing SLList.removeFirst() (cont.)

```
private E removeFirst () {
  Node<E> temp = head;
  if (head != null) {
    head = head.next;
  }
  if (temp != null) {
    size--;
    return temp.data
  } else {
    return null;
  }
} // removeFirst()
```

Traversing a Single-Linked List



This is called a TRAVERSAL.

Traversing a Single-Linked List (cont.)

- `toString()`can be implemented with a traversal:

```
public String toString() {
  Node<String> nodeRef = head;
  StringBuilder result = new StringBuilder();
  while (nodeRef != null) {
    result.append(nodeRef.data);
    if (nodeRef.next != null) {
      result.append(" ==> ");
    }
    nodeRef = nodeRef.next;
  }
  return result.toString();
} // toString()
```

SLList.getNode(int)

- In order to implement methods required by the List interface, we need an additional helper method:

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

SingleLinkedList Class

Method	Behavior
`public E get(int index)`	Returns a reference to the element at position `index`.
`public E set(int index, E anEntry)`	Sets the element at position `index` to reference `anEntry`. Returns the previous value.
`public int size()`	Gets the current size of the `List`.
`public boolean add(E anEntry)`	Adds a reference to `anEntry` at the end of the `List`. Always returns `true`.
`public void add(int index, E anEntry)`	Adds a reference to `anEntry`, inserting it before the item at position `index`.
`int indexOf(E target)`	Searches for `target` and returns the position of the first occurrence, or –1 if it is not in the `List`.

Also need (in addition to constructors):

```
private void addFirst(Node n)
private void addAfter(Node n, E item)
private void removeFirst()
private E removeAfter(Node n)
```

public E get(int index)

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

public E set(int index, E newValue)

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

public void add(int index, E item)

```
public void add (int index, E item) {
 if (index < 0 || index > size) {
   throw new
     IndexOutOfBoundsException(Integer.toString(index));
 }
 if (index == 0) {
   addFirst(item);
 } else {
   Node<E> node = getNode(index-1);
   addAfter(node, item);
 }
} // add()
```

public boolean add(E item)

- To add an item to the end of the list

```
public boolean add (E item) {
  add(size, item);
  return true;
} // add()
```

Clear(), size(), isEmpty()


```
public void clear() {
   head = null;
   size = 0;
} // clear()

public int size() {
   return this.size();
} // size()

public boolean isEmpty() {
   return size == 0;
} // isEmpty()
```

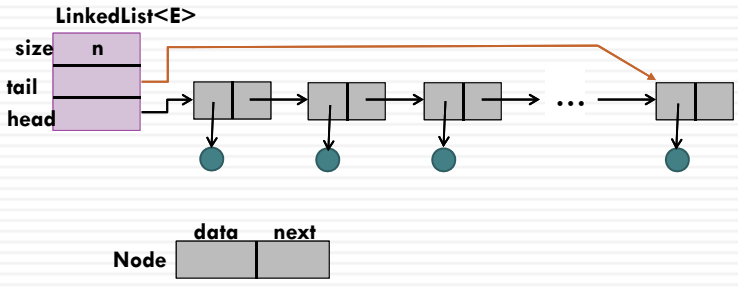
Comparing Performance

We can Make it O(1)!

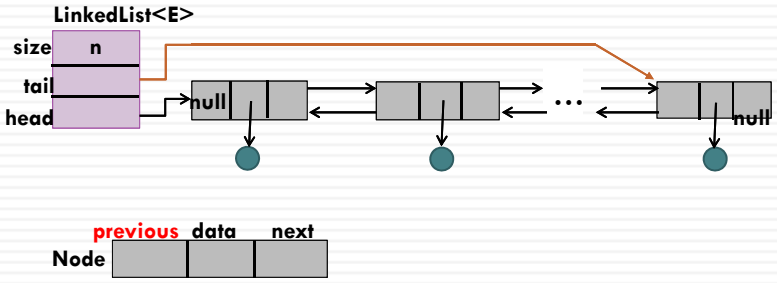
35

	Fixed Size Array	Dynamic Array	Single-Linked List
add(o)	O(1)	O(n)	O(n)
add(i, o)	O(n)	O(n)	O(n)
remove(i)	O(n)	O(n)	O(n)
set(i, o)	O(1)	O(1)	O(n)
get(i)	O(1)	O(1)	O(n)
contains(o)	O(n)	O(n)	O(n)
clear(), size(), isEmpty()	O(1)	O(1)	O(1)

Single-Linked Lists with a Tail



Double-Linked Lists with a tail



The `LinkedList` Class and the `Iterator`, `ListIterator`, and `Iterable` Interfaces

Section 2.7

The `LinkedList` Class

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 first node
- 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 traversal using `get()` calls in a linked list is O(n^2)

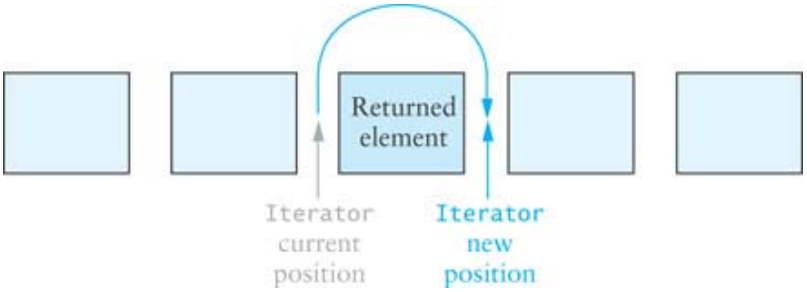
`Iterator` Interface

- The `Iterator` interface is defined in `java.util`
- The `List` interface declares the method `iterator` which 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 = aList.iterator();
while (iter.hasNext()) {
  int value = iter.next();
  // Do something with value
  ...
}
```

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 $O(n^2)$
  - `Iterator.remove` removes items without starting over at the beginning, so in general it is O(n)

Iterators and Removing Elements (cont.)

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

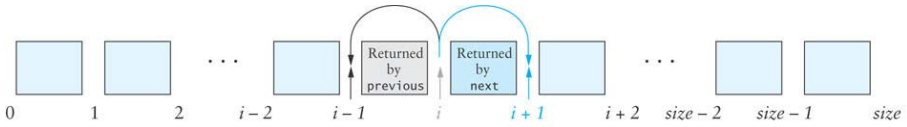
```
public static void removeDivisibleBy(LinkedList<Integer>
                                                aList, int div) {
  Iterator<Integer> iter = aList.iterator();
  while (iter.hasNext()) {
    int nextInt = iter.next();
    if (nextInt % div == 0) {
      iter.remove();
    }
  }
}
```

`ListIterator` Interface

- `Iterator` limitations
  - Traverses `List` only in the forward direction
  - Provides a `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

`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`



`ListIterator` Interface (cont.)

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.

`ListIterator` Interface (cont.)

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`.

Comparison of `Iterator` and `ListIterator`

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

Conversion Between `ListIterator` and an Index

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

Enhanced `for` Statement

- Java 5.0 introduced 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 (cont.)

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

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

`Iterable` Interface

- Each class that implements the `List` interface must provide an `iterator` method
- The `Collection` interface extends the `Iterable` interface
- All classes that implement the `List` interface (a subinterface of `Collection`) must provide an `iterator` method
- Allows use of the Java 5.0 *for-each* loop

```
public interface Iterable<E> {
    /** returns an iterator over the elements in this
    collection. */
  Iterator<E> iterator();
}
```

The Collections Framework Design

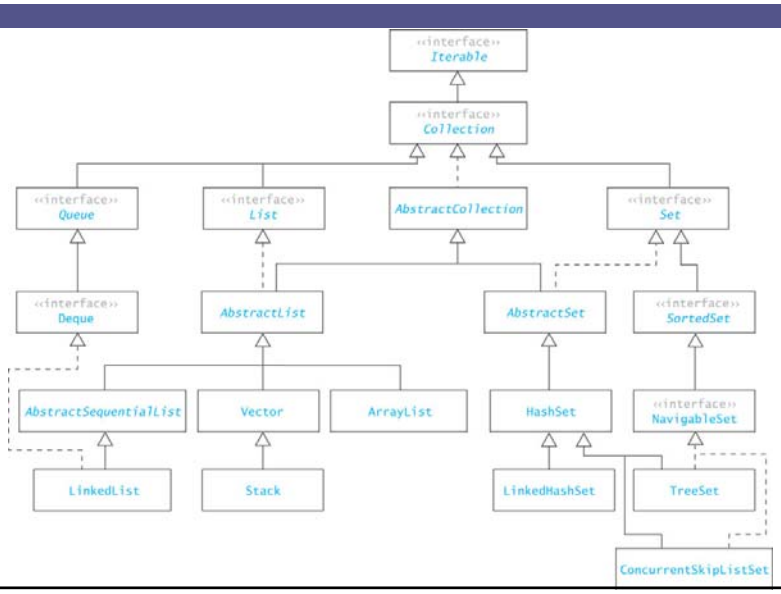
Section 2.9

The `Collection` Interface

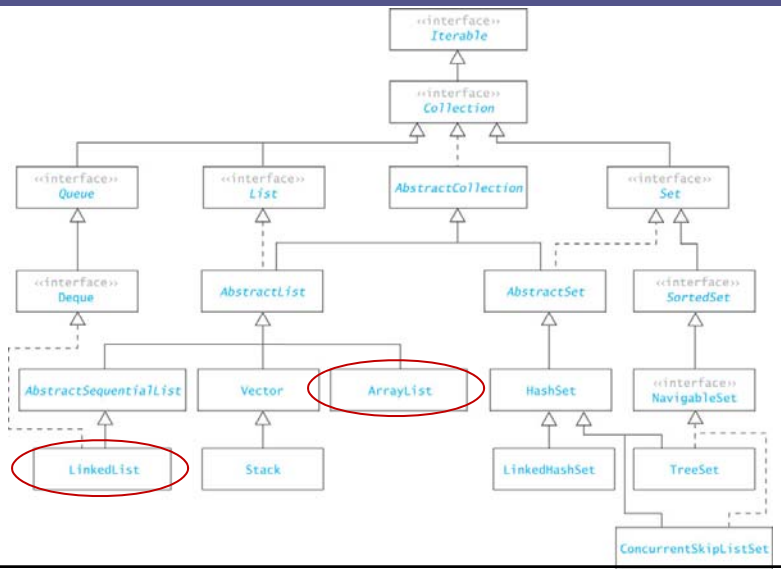
- Specifies a subset of methods in the `List` interface, specifically excluding
  - `add(int, E)`
  - `get(int)`
  - `remove(int)`
  - `set(int, E)`

  but including
  - `add(E)`
  - `remove(Object)`
  - the `iterator` method

The `Collection` Framework



The `Collection` Framework



Common Features of Collections

- Collections
  - grow as needed
  - hold references to objects
  - have at least two constructors: one to create an empty collection and one to make a copy of another collection

Common Features of Collections (cont.)

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

- In a general `Collection` the order of elements is not specified
- For collections implementing the `List` interface, the order of the elements is determined by the index

Common Features of Collections (cont.)

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

- In a general `Collection`, the position where an object is inserted is not specified
- In `ArrayList` and `LinkedList`, `add(E)` always inserts at the end and always returns `true`

`AbstractCollection`, `AbstractList`, **and** `AbstractSequentialList`

- The Java API includes several "helper" abstract classes to help build implementations of their corresponding interfaces
- By providing implementations for interface methods not used, the helper classes require the programmer to extend the `AbstractCollection` class and implement only the desired methods

Implementing a Subclass of `Collection<E>`

- Extend `AbstractCollection<E>`, which 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()`
- `AbstractList` implements `Iterator<E>` using the index