CHAPTER 2
Lists and the Collections Framework

The Collection Framework
## Java Collections Interface

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

## LIST Interface (Ordered Collection)

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>add(E e)</td>
</tr>
<tr>
<td></td>
<td>Appends the specified element to the end of this list (optional operation).</td>
</tr>
<tr>
<td>void</td>
<td>add(int index, E element)</td>
</tr>
<tr>
<td></td>
<td>Inserts the specified element at the specified position in this list (optional operation).</td>
</tr>
<tr>
<td>boolean</td>
<td>remove(Object o)</td>
</tr>
<tr>
<td></td>
<td>Removes the first occurrence of the specified element from this list, if it is present (optional operation).</td>
</tr>
<tr>
<td>E</td>
<td>remove(int index)</td>
</tr>
<tr>
<td></td>
<td>Removes the element at the specified position in this list (optional operation).</td>
</tr>
<tr>
<td>boolean</td>
<td>contains(Object o)</td>
</tr>
<tr>
<td></td>
<td>Returns true if this list contains the specified element.</td>
</tr>
<tr>
<td>E</td>
<td>get(int index)</td>
</tr>
<tr>
<td></td>
<td>Returns the element at the specified position in this list.</td>
</tr>
<tr>
<td>boolean</td>
<td>set(int index, E element)</td>
</tr>
<tr>
<td></td>
<td>Replaces the element at the specified position in this list with the specified element (optional operation).</td>
</tr>
<tr>
<td>int</td>
<td>indexOf(Object o)</td>
</tr>
<tr>
<td></td>
<td>Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.</td>
</tr>
<tr>
<td>int</td>
<td>lastIndexOf(Object o)</td>
</tr>
<tr>
<td></td>
<td>Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.</td>
</tr>
<tr>
<td>boolean</td>
<td>isEmpty()</td>
</tr>
<tr>
<td></td>
<td>Returns true if this list contains no elements.</td>
</tr>
<tr>
<td>int</td>
<td>size()</td>
</tr>
<tr>
<td></td>
<td>Returns the number of elements in this list.</td>
</tr>
<tr>
<td>boolean</td>
<td>equals(Object o)</td>
</tr>
<tr>
<td></td>
<td>Compares the specified object with this list for equality.</td>
</tr>
<tr>
<td>void</td>
<td>clear()</td>
</tr>
<tr>
<td></td>
<td>Removes all of the elements from this list (optional operation).</td>
</tr>
</tbody>
</table>
### Java.util.ArrayList<E>

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add(E e)</code></td>
<td>Appends the specified element to the end of this list (optional operation).</td>
</tr>
<tr>
<td><code>add(int index, E element)</code></td>
<td>Inserts the specified element at the specified position in this list (optional operation).</td>
</tr>
<tr>
<td><code>remove(Object o)</code></td>
<td>Removes the first occurrence of the specified element from this list, if it is present (optional operation).</td>
</tr>
<tr>
<td><code>remove(int index)</code></td>
<td>Removes the element at the specified position in this list (optional operation).</td>
</tr>
<tr>
<td><code>set(int index, E element)</code></td>
<td>Replaces the element at the specified position in this list with the specified element (optional operation).</td>
</tr>
<tr>
<td><code>indexOf(Object o)</code></td>
<td>Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.</td>
</tr>
<tr>
<td><code>lastIndexOf(Object o)</code></td>
<td>Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.</td>
</tr>
<tr>
<td><code>isEmpty()</code></td>
<td>Returns true if this list contains no elements.</td>
</tr>
<tr>
<td><code>size()</code></td>
<td>Returns the number of elements in this list.</td>
</tr>
<tr>
<td><code>equals(Object o)</code></td>
<td>Compares the specified object with this list for equality.</td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>Removes all of the elements from this list (optional operation).</td>
</tr>
</tbody>
</table>

### MyListInterface

```java
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

<table>
<thead>
<tr>
<th></th>
<th>Fixed Size Array</th>
<th>Dynamic Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(o)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>add(i, o)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>remove(i)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>set(i, o)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>get(i)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>contains(o)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>clear(), size(), isEmpty()</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

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 (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.

```java
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

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

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)

The element added to the list
Implementing `SLList.addFirst(E item)` (cont.)

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

or, more simply ...

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

This works even if `head` is null

```
SingleLinkedList<E>
head: null
size: 0
```

```
SingleLinkedList<E>
head: 1
size: 1
```
Implementing `addAfter(Node<E> node, E item)`

```
SLList<String>
head =
next =
data = "Tom"

Node<String>
next =
data = "Dick"

Node<String>
next =
data = "Ann"
```

The element added to the list
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++;
}

or, more simply ...

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

Implementing removeAfter(Node<E> node)

The Node parameter

SLList<String>

Node<String>

Node<String>

Node<String>

next = 
data = "Tom"

next = 
data = "Dick"

next = 
data = "Ann"
Implementing `removeAfter(Node<E> node)` (cont.)

```java
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
\texttt{SLList.removeFirst()} (cont.)

```java
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:
  ```java
  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:
  ```java
  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

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

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) {
    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)

```java
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)

```java
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

<table>
<thead>
<tr>
<th></th>
<th>Fixed Size Array</th>
<th>Dynamic Array</th>
<th>Single-Linked List</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(o)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>add(i, o)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>remove(i)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>set(i, o)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>get(i)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>contains(o)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>clear(), size(), isEmpty()</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

We can Make it 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

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

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.

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

**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`

```java
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:

```java
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.

![Diagram showing the positions of ListIterator](image)

### ListIterator Interface (cont.)

<table>
<thead>
<tr>
<th>Method</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>void add(E obj)</td>
<td>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.</td>
</tr>
<tr>
<td>boolean hasNext()</td>
<td>Returns true if next will not throw an exception.</td>
</tr>
<tr>
<td>boolean hasPrevious()</td>
<td>Returns true if previous will not throw an exception.</td>
</tr>
<tr>
<td>E next()</td>
<td>Returns the next object and moves the iterator forward. If the iterator is at the end, the NoSuchElementException is thrown.</td>
</tr>
<tr>
<td>int nextIndex()</td>
<td>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.</td>
</tr>
<tr>
<td>E previous()</td>
<td>Returns the previous object and moves the iterator backward. If the iterator is at the beginning of the list, the NoSuchElementException is thrown.</td>
</tr>
<tr>
<td>int previousIndex()</td>
<td>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.</td>
</tr>
<tr>
<td>void remove()</td>
<td>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 NoSuchElementException is thrown.</td>
</tr>
<tr>
<td>void set(E obj)</td>
<td>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 NoSuchElementException is thrown.</td>
</tr>
</tbody>
</table>
ListIterator Interface (cont.)

<table>
<thead>
<tr>
<th>Method</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>public ListIterator&lt;?super E&gt; listIterator()</td>
<td>Returns a ListIterator that begins just before the first list element.</td>
</tr>
<tr>
<td>public ListIterator&lt;?super E&gt; listIterator(int index)</td>
<td>Returns a ListIterator that begins just before position index.</td>
</tr>
</tbody>
</table>

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

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

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

```java
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[]`

```java
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

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

[Diagram of 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.)

- 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

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

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