Iterators

Chapter 15

THIRD EDITION

Data Structures and Abstractions with Java™

FRANK M. CARRANO

Contents

Contents

Objectives

- Describe concept of iterator
- Use iterator to traverse, manipulate a list
- Implement in Java separate class iterator, inner class iterator for list
- Describe pros and cons of separate class iterators, inner class iterators

What Is an Iterator?

- Program component
  - Enables you to step through, or
  - Traverse, a collection of data
- During a traversal
  - Each data item conside
- When we write loops
  - They traverse, iterate through whole list
- Consider interface **`Iterator`**, Listing 15-1

Note: Code listing files must be in same folder as PowerPoint files for links to work

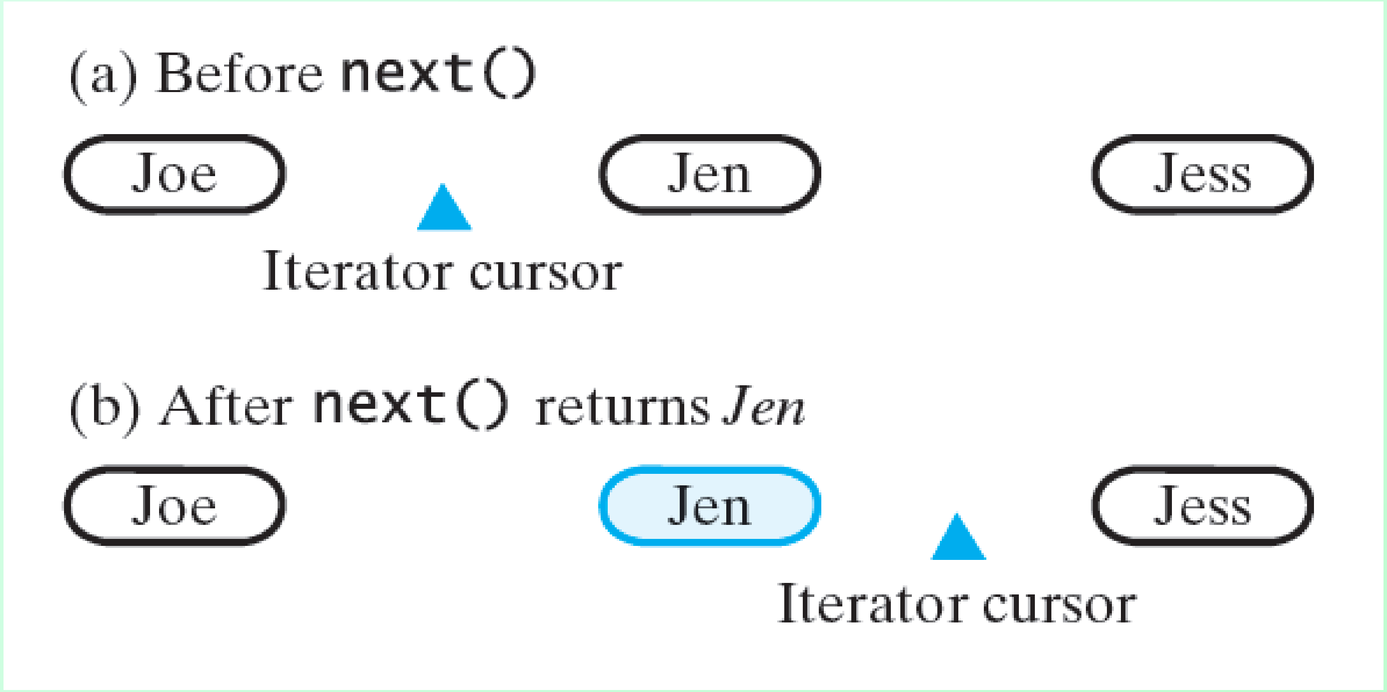


Figure 15-1 The effect of a call to next on a list iterator

Using the Interface `Iterator`

- Implement iterator methods within their own class
  - Can be private, separate from ADT class
  - Or private, inner class of ADT
- Consider list of strings

```
ListInterface<String> nameList = new
   LList<String>();
nameList.add("Jamie");
nameList.add("Joey");
nameList.add("Rachel");
```

Using the Interface **`Iterator`**

- Create an instance of **`SeparateIterator`**

```
Iterator<String> nameIterator = new
SeparateIterator<String>(nameList);
```

- Connects the iterator **`nameIterator`** to the list **`nameList`**

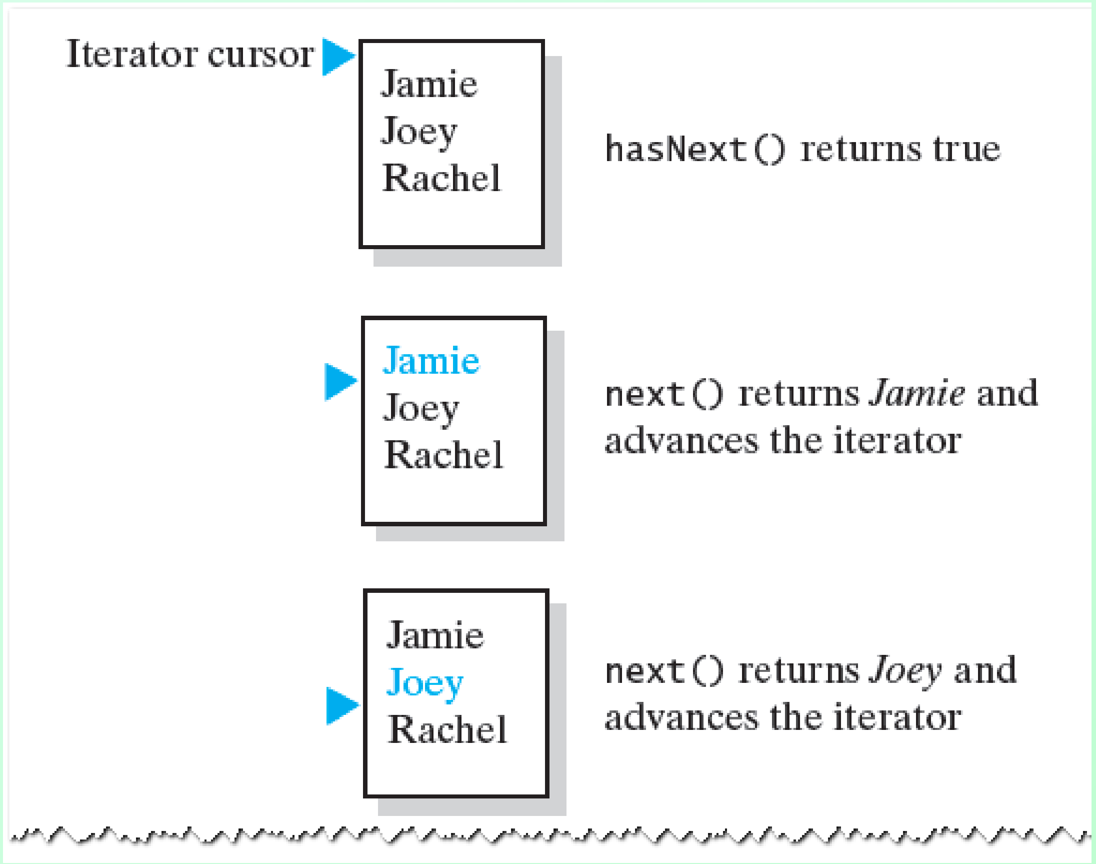

Iterator cursor
Jamie
Joey
Rachel
hasNext() returns true
Jamie
Joey
Rachel
next() returns *Jamie* and advances the iterator
Jamie
Joey
Rachel
next() returns *Joey* and advances the iterator

Figure 15-2 The effect of the iterator methods **`hasNext`** and **`next`** on a list

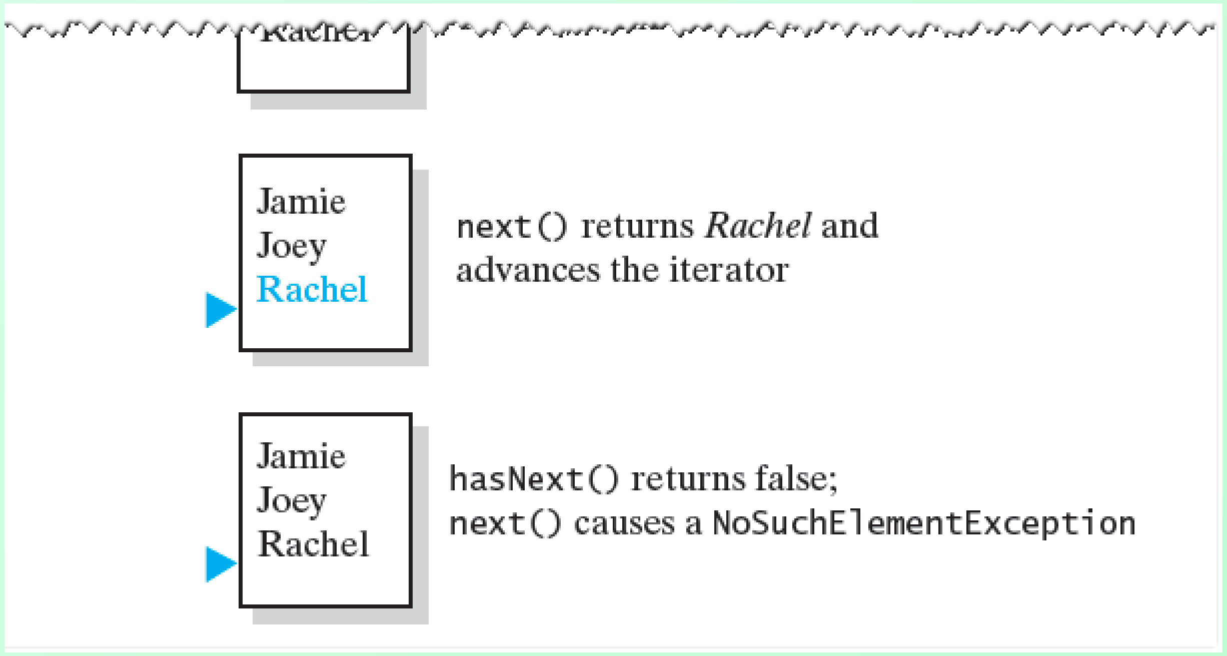


Figure 15-2 The effect of the iterator methods **`hasNext`** and **`next`** on a list

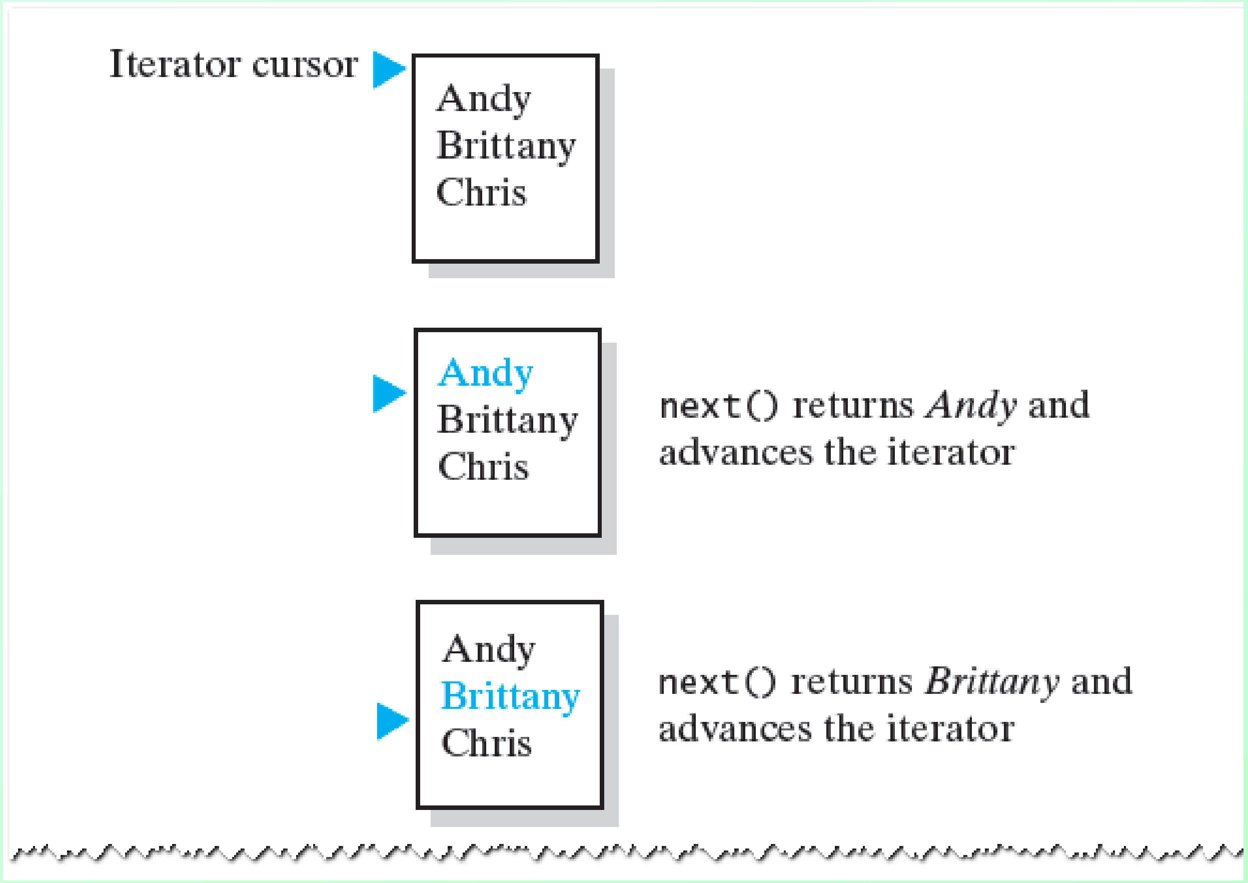

Iterator cursor
Andy
Brittany
Chris
Andy
Brittany
Chris
next() returns *Andy* and advances the iterator
Andy
Brittany
Chris
next() returns *Brittany* and advances the iterator

Figure 15-3 The effect of the iterator methods next and remove on a list

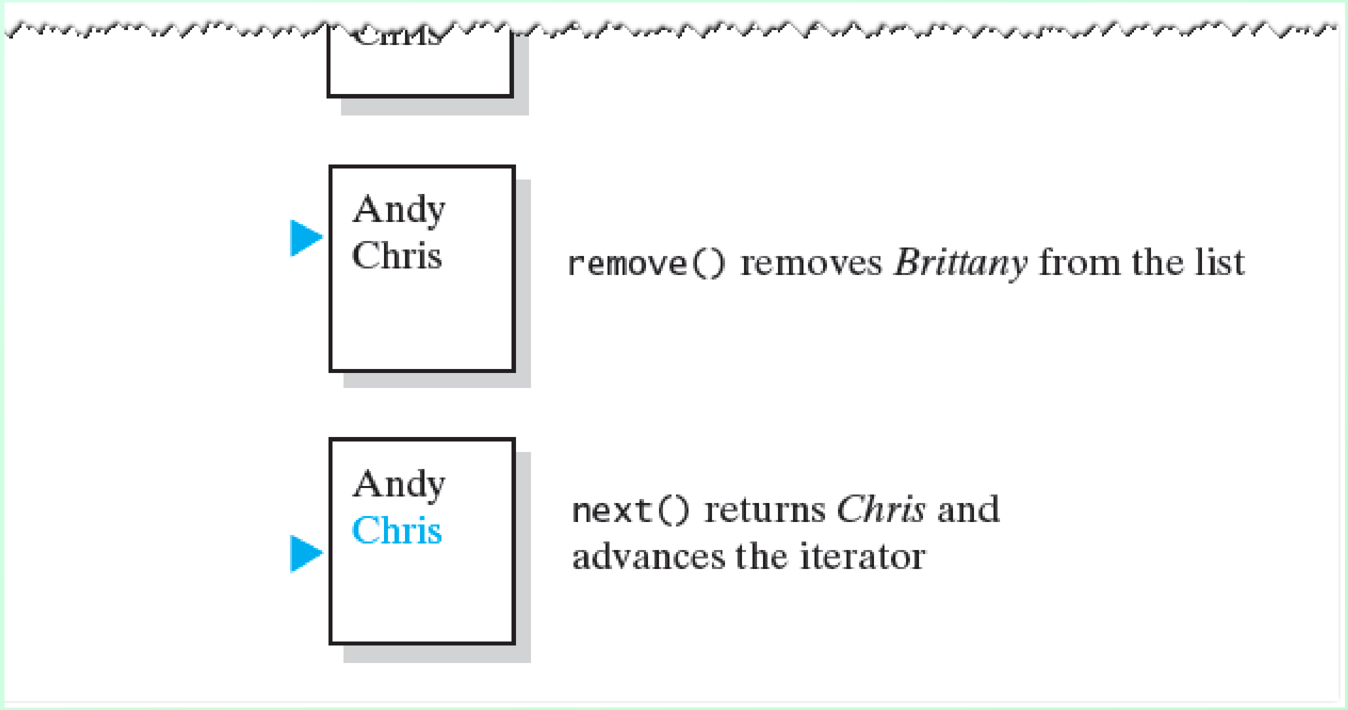

Chris
Andy
Chris
remove() removes *Brittany* from the list
Andy
Chris
next() returns *Chris* and
advances the iterator

Figure 15-3 The effect of the iterator methods next and remove on a list

Question 1 Assume that nameList contains the names Jamie, Joey, and Rachel, as it does in Segment 15.6. What output is produced by the following Java statements?

```
Iterator<String> nameIterator = new  SeparateIterator<String>(nameList);
nameIterator.next();
nameIterator.next();
nameIterator.remove();
System.out.println(nameIterator.hasNext());
System.out.println(nameIterator.next());
```

Question 2 Assume that nameList is an instance of a class that implements ListInterface, and nameIterator is defined as in the previous question. If nameList contains at least three strings, write Java statements that display the list's third entry.

Question 1 Assume that nameList contains the names Jamie, Joey, and Rachel, as it does in Segment 15.6. What output is produced by the following Java statements?

```
Iterator<String> nameIterator = new  SeparateIterator<String>(nameList);
nameIterator.next();
nameIterator.next();
nameIterator.remove();
System.out.println(nameIterator.hasNext());
System.out.println(nameIterator.next());
```

```
true
Rachel
```

Question 2 Assume that nameList is an instance of a class that implements ListInterface, and nameIterator is defined as in the previous question. If nameList contains at least three strings, write Java statements that display the list's third entry.

```
nameIterator.next();
nameIterator.next();
System.out.println(nameIterator.next());
```

Question 3   Given nameList and nameIterator as described in the previous question, write statements that display the even-numbered entries in the list. That is, display the second entry, the fourth entry, and so on.

Question 4 Given nameList and nameIterator as described in Question 2, write statements that remove all entries from the list.

Question 3 Given nameList and nameIterator as described in the previous question, write statements that display the even-numbered entries in the list. That is, display the second entry, the fourth entry, and so on.

```
nameIterator.next();                            // skip first entry; list has > 1 entry
while (nameIterator.hasNext())
{   System.out.println(nameIterator.next());    // display even-numbered entry
    if (nameIterator.hasNext())
        nameIterator.next();                    // skip odd-numbered entry
}
```

Question 4 Given nameList and nameIterator as described in Question 2, write statements that remove all entries from the list.

```
while (nameIterator.hasNext())
{   nameIterator.next();
    nameIterator.remove();
}
```

Multiple Iterators

- Possible to have multiple iterators of the same list in progress simultaneously
- View code which counts number of times that *Jane* appears in a list of names [Listing 15-A]
- Output of Listing 15-A (below 15-A code)

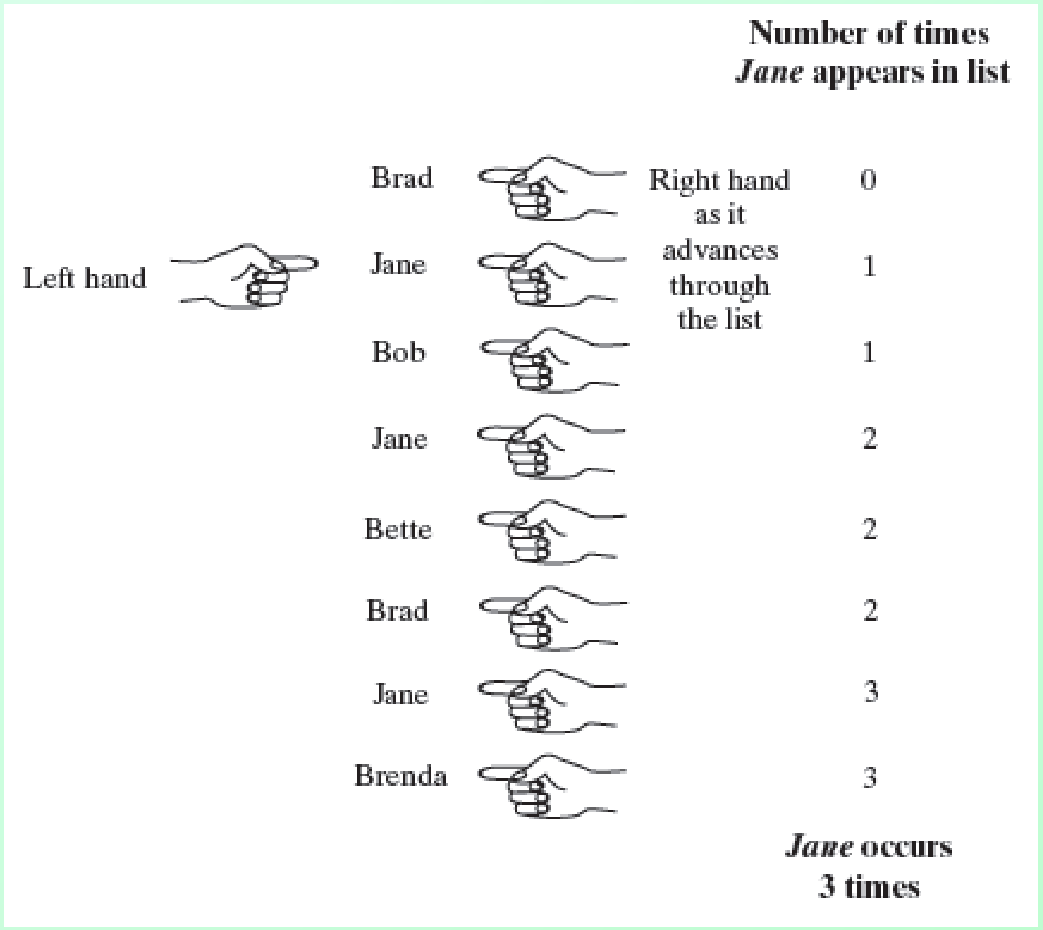


Figure 15-4 Counting the number of times that Jane appears in a list of names

A Separate Class Iterator

- Implementation of class **`SeparateIterator`**
  - Implements the interface **`java.util.Iterator`**
- View source code, Listing 15-2
- Note: definition of **`SeparateIterator`** independent of a particular implementation

Question 5 What does the method hasNext return when the list is empty? Why?

Question 5 What does the method hasNext return when the list is empty? Why?

False. When the list is empty, both nextPosition and list.getLength() are zero.

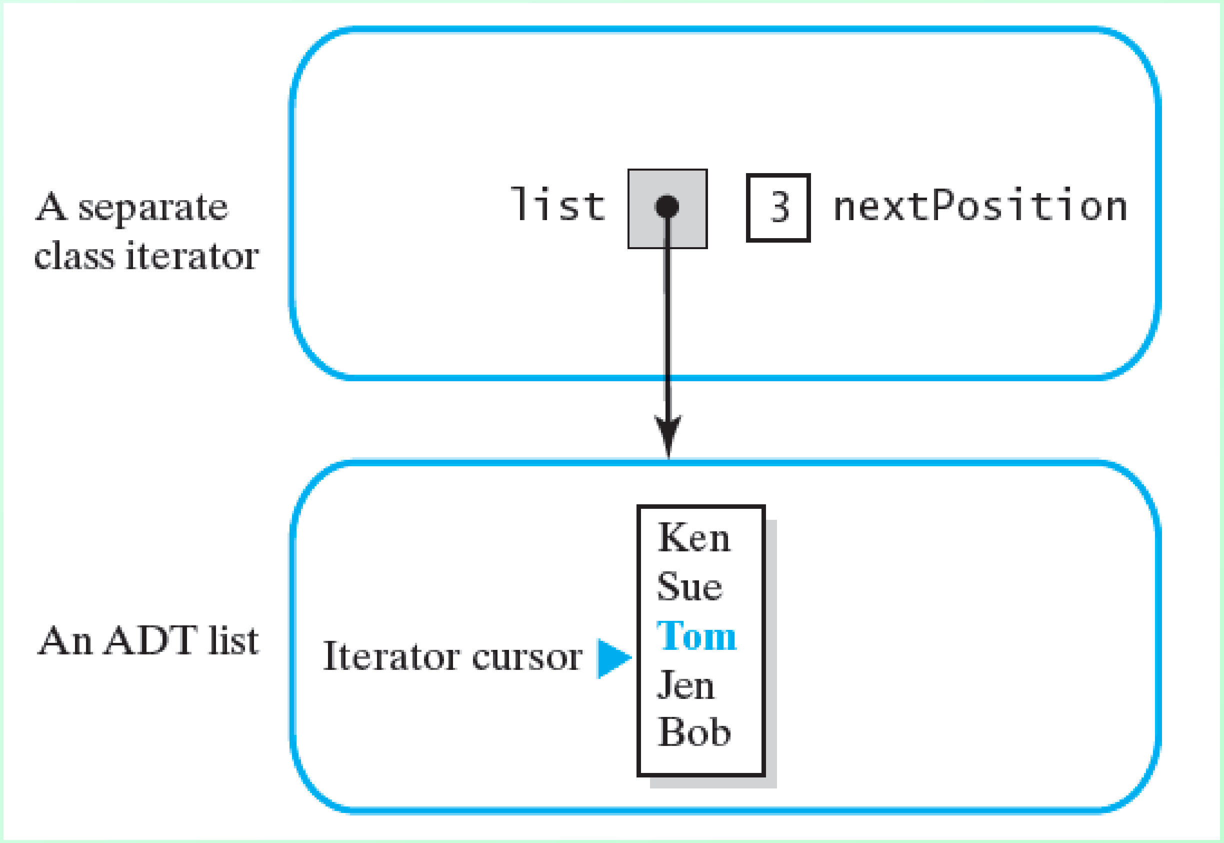


Figure 15-5 A separate class iterator with a reference to an ADT, an indicator of its position within the iteration, and no knowledge of the ADT's implementation

Question 6 The work performed by the method next depends upon the implementation of the ADT list that is ultimately used. For which implementation of the list, array-based or linked, will next use the most execution time? Why?

Question 6 The work performed by the method next depends upon the implementation of the ADT list that is ultimately used. For which implementation of the list, array-based or linked, will next use the most execution time? Why?

Linked. The particular implementation of the list affects the amount of work that the method getEntry must perform. For an array-based implementation, getEntry accesses the required entry directly and immediately. For a linked implementation, getEntry must traverse a chain of nodes to find the desired entry. This takes more time to accomplish than accessing an array entry.

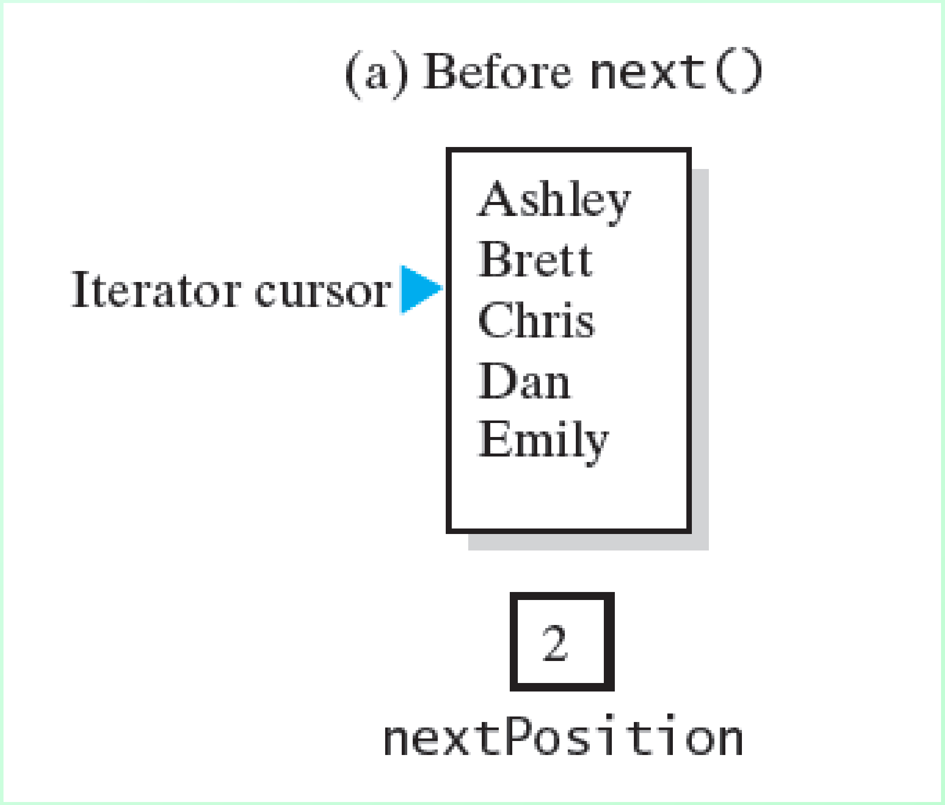


Figure 15-6 A list and **`nextPosition`**
(a) just before the call to **`next`**;

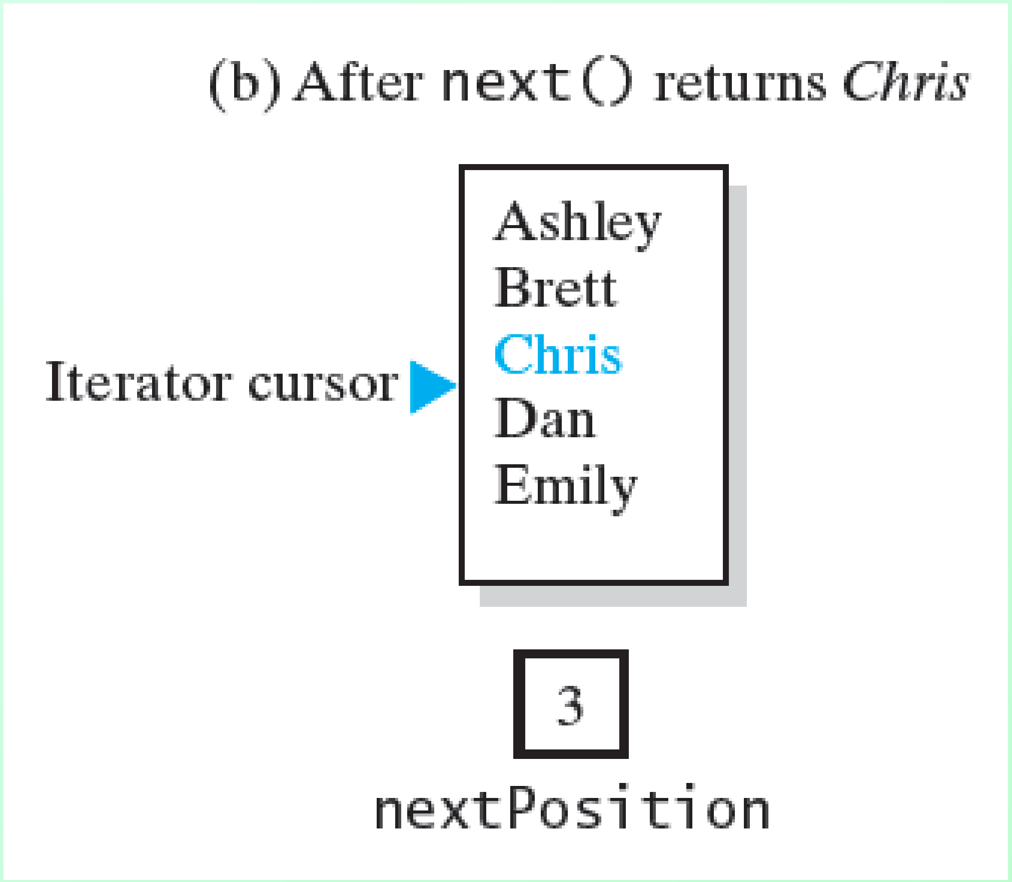


Figure 15-6 A list and **`nextPosition`** (b) just after the call to **`next`** but before the call to **`remove`**

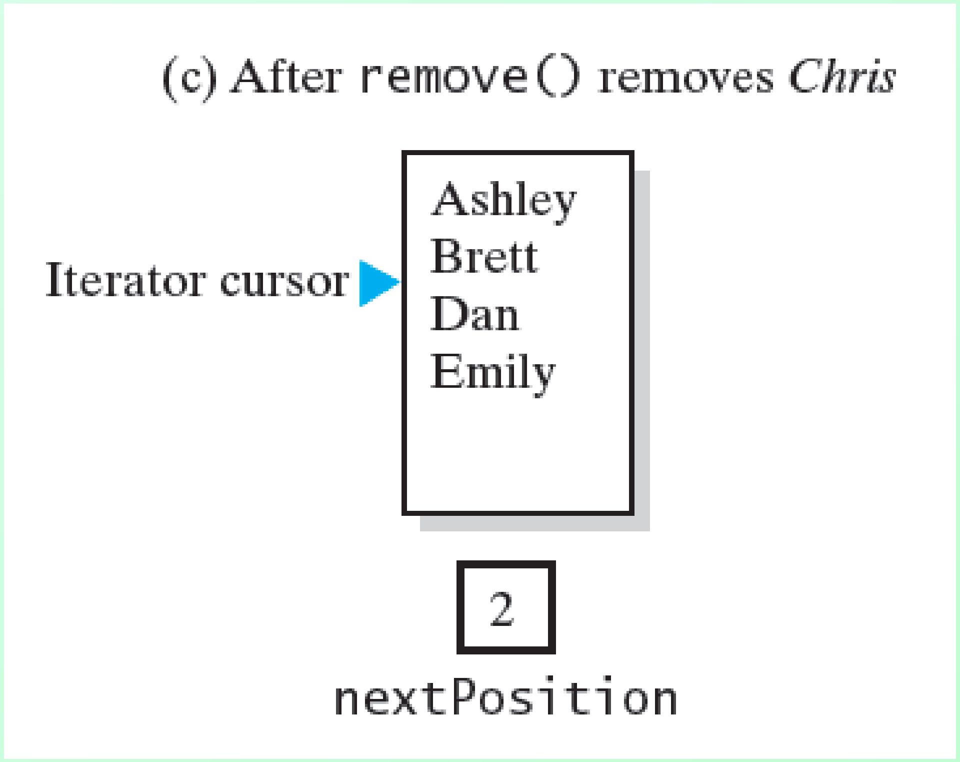


Figure 15-6 A list and **`nextPosition`**
(c) after the call to **`remove`**

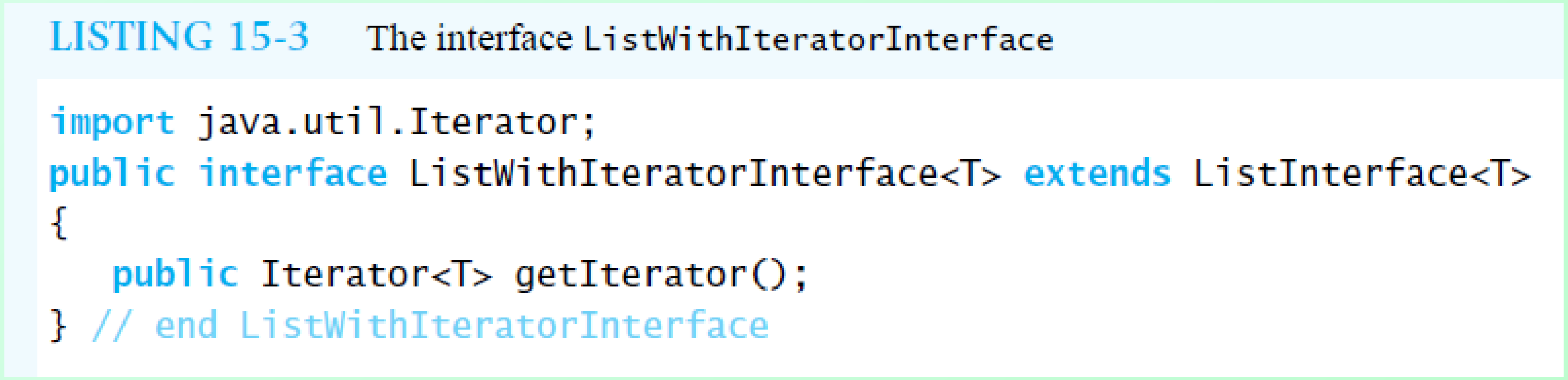
Linked Implementation

- Define the methods specified in `Iterator` within new inner class
  - A class that implements the ADT list
  - Needs another method that client can use to create an iterator

```
public Iterator<T> getIterator()
{
   return new IteratorForLinkedList();
} // end getIterator
```

Linked Implementation

- New interface needed
  Listing 15-3


LISTING 15-3 The interface `ListWithIteratorInterface`

```
import java.util.Iterator;
public interface ListWithIteratorInterface<T> extends ListInterface<T>
{
   public Iterator<T> getIterator();
} // end ListWithIteratorInterface
```

- Listing 15-4, an outline of the class **`LinkedListWithIterator`**

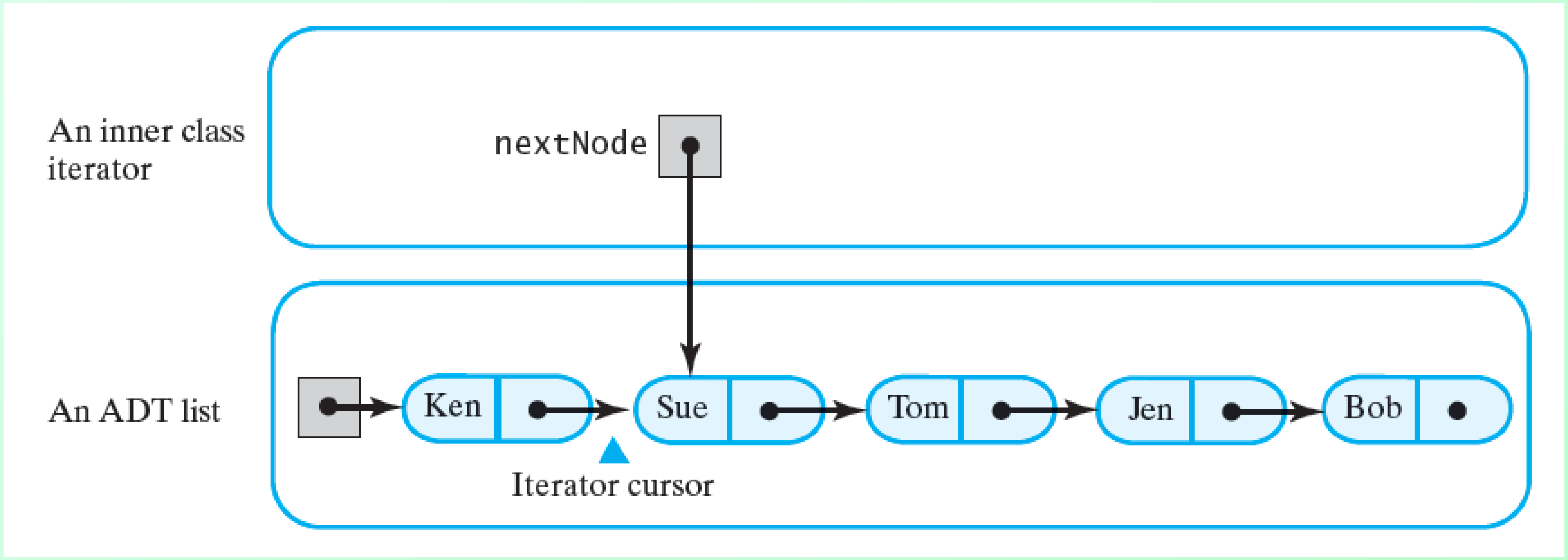


Figure 15-7 An inner class iterator with direct access to the linked chain that implements the ADT

Question 7 What does the method hasNext return when the list is empty? Why?

Question 8 Given the class LinkedListWithIterator, what Java statements create the iterators nameIterator and countingIterator mentioned in Segment 15.11?

Question 7 What does the method hasNext return when the list is empty? Why?

False. When the list is empty, firstNode, and therefore nextNode , is null.

Question 8 Given the class LinkedListWithIterator, what Java statements create the iterators nameIterator and countingIterator mentioned in Segment 15.11?

Create the iterators by writing

```
   Iterator<String> nameIterator = nameList.getIterator();
   Iterator<String> countingIterator = nameList.getIterator();
```

Array-Based Implementation

- Iterator will support the **`remove`** method
- Adaptation of **`Alist`** class, Chapter 13
- Listing 15-5
  - Implements interface **`ListWithIteratorInterface`**
  - Also includes method **`getIterator`**
  - Contains the inner class **`IteratorForArrayList,`** implements interface **`Iterator.`**

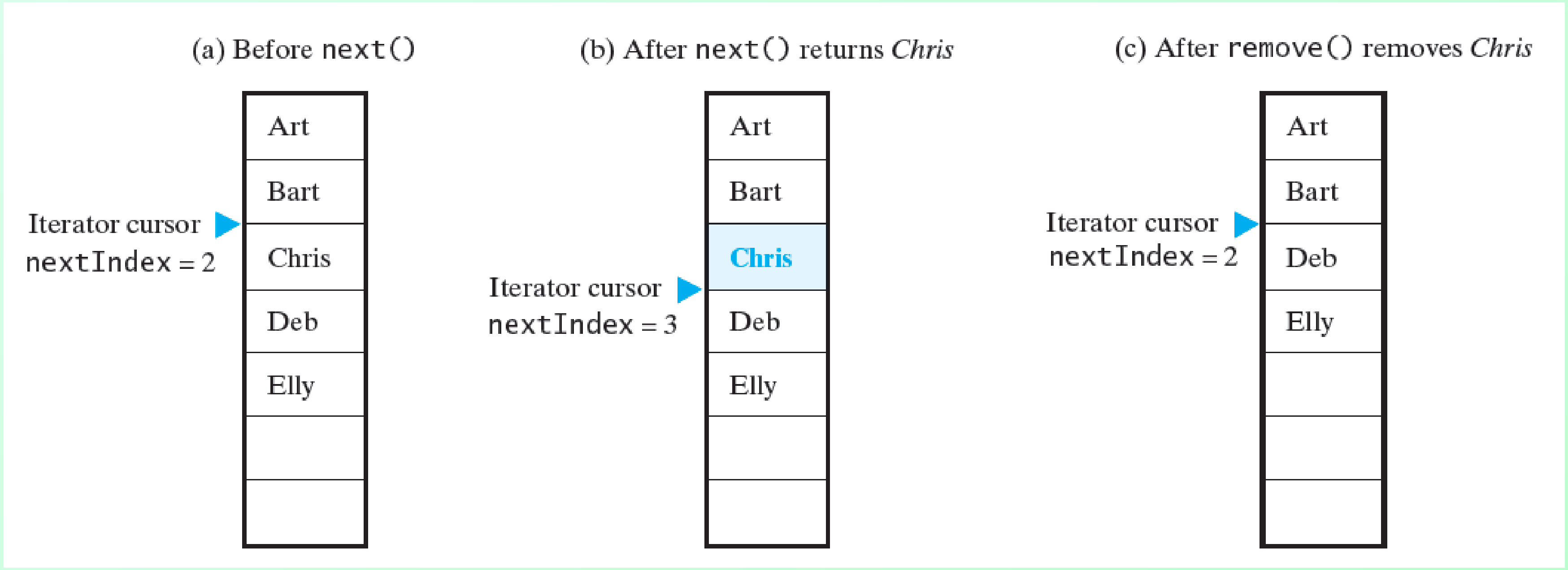


FIGURE 15-8 The array of list entries and **`nextIndex`** (a) just before the call to **`next`**; (b) just after the call to **`next`** but before the call to **`remove`**; (c) after the call to **`remove`**

Question 10 Consider the list and the calls to next and remove in Figure 15-8.

a. What would a call to next return if it occurred after the call to remove in Figure 15-8c?

b. What would a call to next return if it occurred after the call to next in Figure 15-8b?

Question 11 What changes would be necessary to the methods in the inner class IteratorForArrayList if its constructor set nextIndex to - 1 instead of 0?

Question 10 Consider the list and the calls to next and remove in Figure 15-8.

a. What would a call to next return if it occurred after the call to remove in Figure 15-8c?

a. Deb.

b. What would a call to next return if it occurred after the call to next in Figure 15-8b?

a. Deb.

Question 11 What changes would be necessary to the methods in the inner class IteratorForArrayList if its constructor set nextIndex to - 1 instead of 0?

Originally, nextIndex is the index of the next entry that next will return. The change makes nextIndex the index of the last entry that next returned. Thus, the following changes are needed:

- hasNext should compare nextIndex to numberOfEntries - 1 instead of numberOfEntries
- next should increment nextIndex before accessing list[nextIndex]
- remove should remove the entry at nextIndex + 1

Why Are Iterator Methods in Their Own Class?

- Inner class iterators have direct access to structure containing ADT's data
- Execute faster than separate class iterators
- Consider Listing 15-6
  - Modified linked implementation
  - Differences with Listing 15-4 highlighted

Why Are Iterator Methods in Their Own Class?

- Consider this traversal

```
myList.resetTraversal();
while (myList.hasNext())
   System.out.println(myList.next());
```

- Quick traversal, but …
  - Only one traversal can be in progress at a time
  - Resulting ADT has too many operations

Question 13 Suppose that you want to omit the method resetTraversal.

a. Could the default constructor initialize nextNode to firstNode? Explain.

b. Could the add methods initialize nextNode to firstNode? Explain.

Question 13 Suppose that you want to omit the method resetTraversal.

a. Could the default constructor initialize nextNode to firstNode? Explain.

No. The default constructor creates an empty list. If it set nextNode to firstNode, nextNode would be set to null.

b. Could the add methods initialize nextNode to firstNode? Explain.

Yes, but with a disadvantage. Each addition to the list would set nextNode to firstNode. After creating a list, you could traverse it. However, the only way you could reset the traversal to the list's beginning would be to add another entry to the list.

The Interface **`ListIterator`**

- A second interface for iterators
  - Listing 15-7
- Extends **`Iterator`**
  - Includes methods **`hasNext`**, **`next`**, and **`remove`**

The Interface **`ListIterator`**

- Methods **`remove`**, **`add`**, and **`set`** are optional
  - Can choose not to provide
  - Must have an implementation that throws exception **`UnsupportedOperationException`**

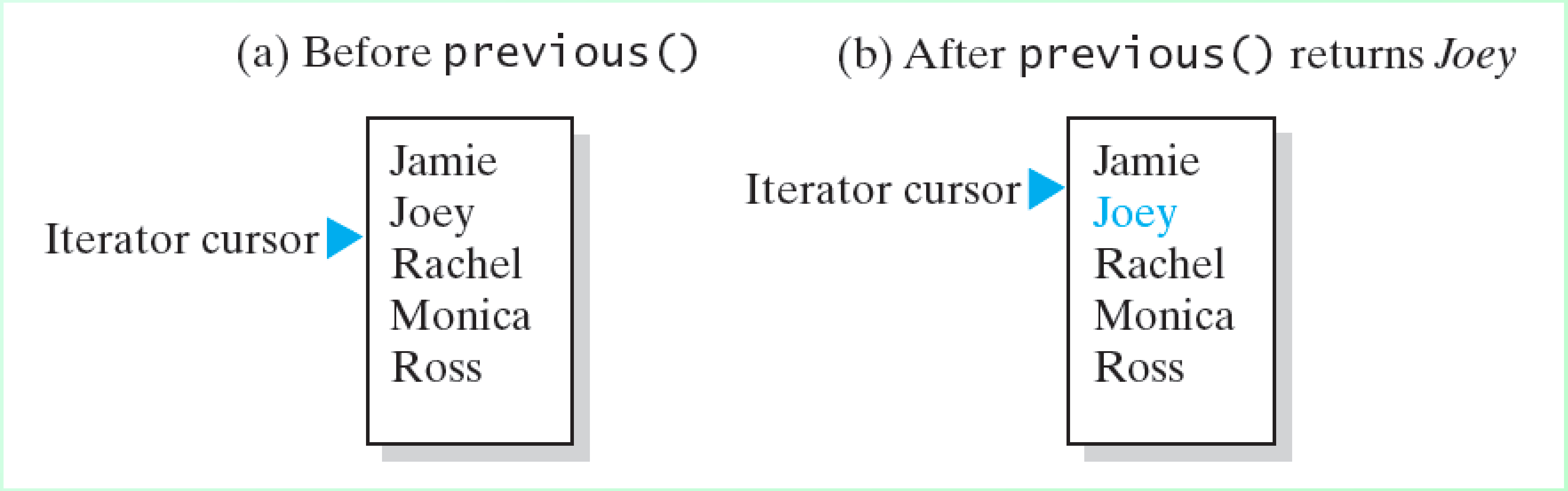


Figure 15-9 The effect of a call to **`previous`** on a list

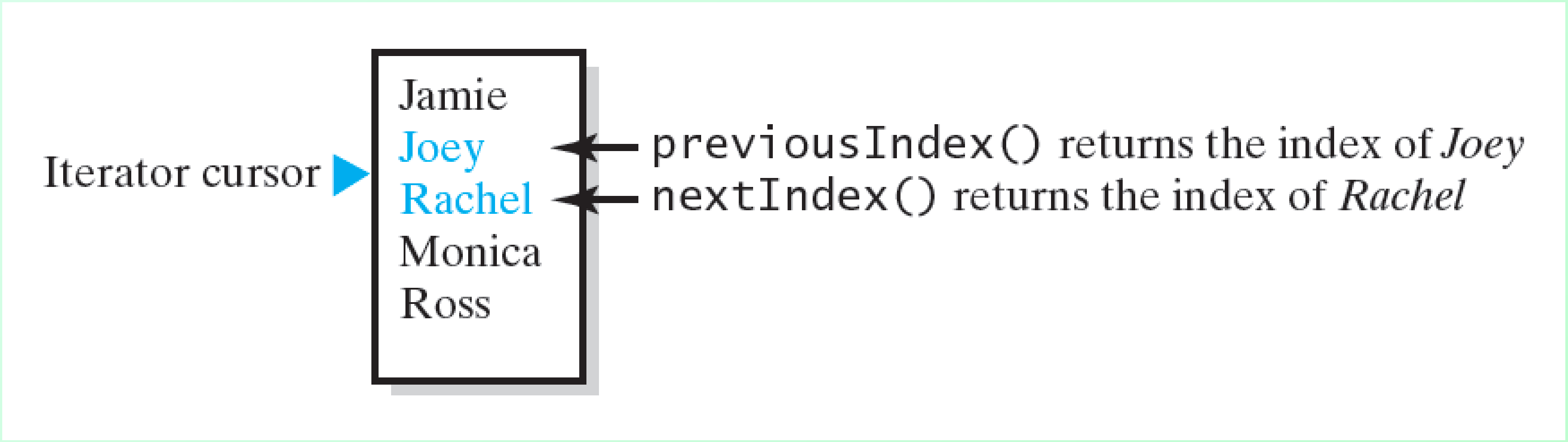

Iterator cursor
Jamie
Joey
Rachel
Monica
Ross
previousIndex() returns the index of *Joey*
nextIndex() returns the index of *Rachel*

Figure 15-10 The indices returned by the methods **`nextIndex`** and **`previousIndex`**

Using Interface **`ListIterator`**

Given

- Interface **`ListIterator`** implemented as inner class of class that implements ADT list.
- Iterator includes the operations **`add`**, **`remove`**, and **`set`**.
- Method **`getIterator`** is added to ADT list.
- List **`nameList`** contains: *Jess, Jim, Josh*
- Iterator **`traverse`** is defined

```
ListIterator<String> traverse =
 nameList.getIterator();
```

Using Interface **`ListIterator`**

- Statements

```
System.out.println("nextIndex     " + traverse.nextIndex());
System.out.println("hasNext       " + traverse.hasNext());
System.out.println("previousIndex " + traverse.previousIndex());
System.out.println("hasPrevious   " + traverse.hasPrevious());
```

- Produce output

```
nextIndex     0
hasNext       true
previousIndex -1
hasPrevious   false
```

Using Interface **`ListIterator`**

- Then, statements

```
System.out.println("next      " + traverse.next());
System.out.println("nextIndex " + traverse.nextIndex());
System.out.println("hasNext   " + traverse.hasNext());
```

- Produce output

```
next      Jess
nextIndex 1
hasNext   true
```

Using Interface `ListIterator`

- Finally, statements

```
System.out.println("previousIndex " + traverse.previousIndex());
System.out.println("hasPrevious   " + traverse.hasPrevious());
System.out.println("previous      " + traverse.previous());
System.out.println("nextIndex     " + traverse.nextIndex());
System.out.println("hasNext       " + traverse.hasNext());
System.out.println("next          " + traverse.next());
```

- Produce output

```
previousIndex 0
hasPrevious   true
previous      Jess
nextIndex     0
hasNext       true
next          Jess
```

Question 15   If the iterator's position is between the first two entries of the previous list, write Java statements that replace Josh with Jon.

Question 16 If the iterator's position is between Ashley and Jim, write Java statements that add Miguel right after Jim.

Question 15 If the iterator’s position is between the first two entries of the previous list, write Java statements that replace Josh with Jon.

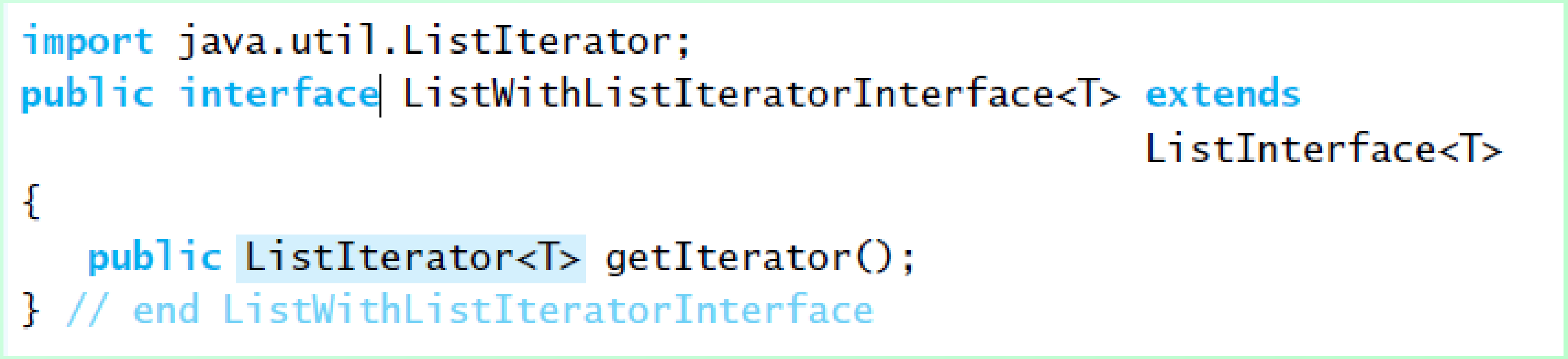
```
traverse.next();        // return Jim
traverse.next();        // return Josh
traverse.set("Jon");  // replace Josh
```

Question 16 If the iterator’s position is between Ashley and Jim, write Java statements that add Miguel right after Jim.

```
traverse.next();              // return Jim
traverse.add("Miguel");  // add Miguel after Jim
```

Array-Based Implementation of Interface **`ListIterator`**

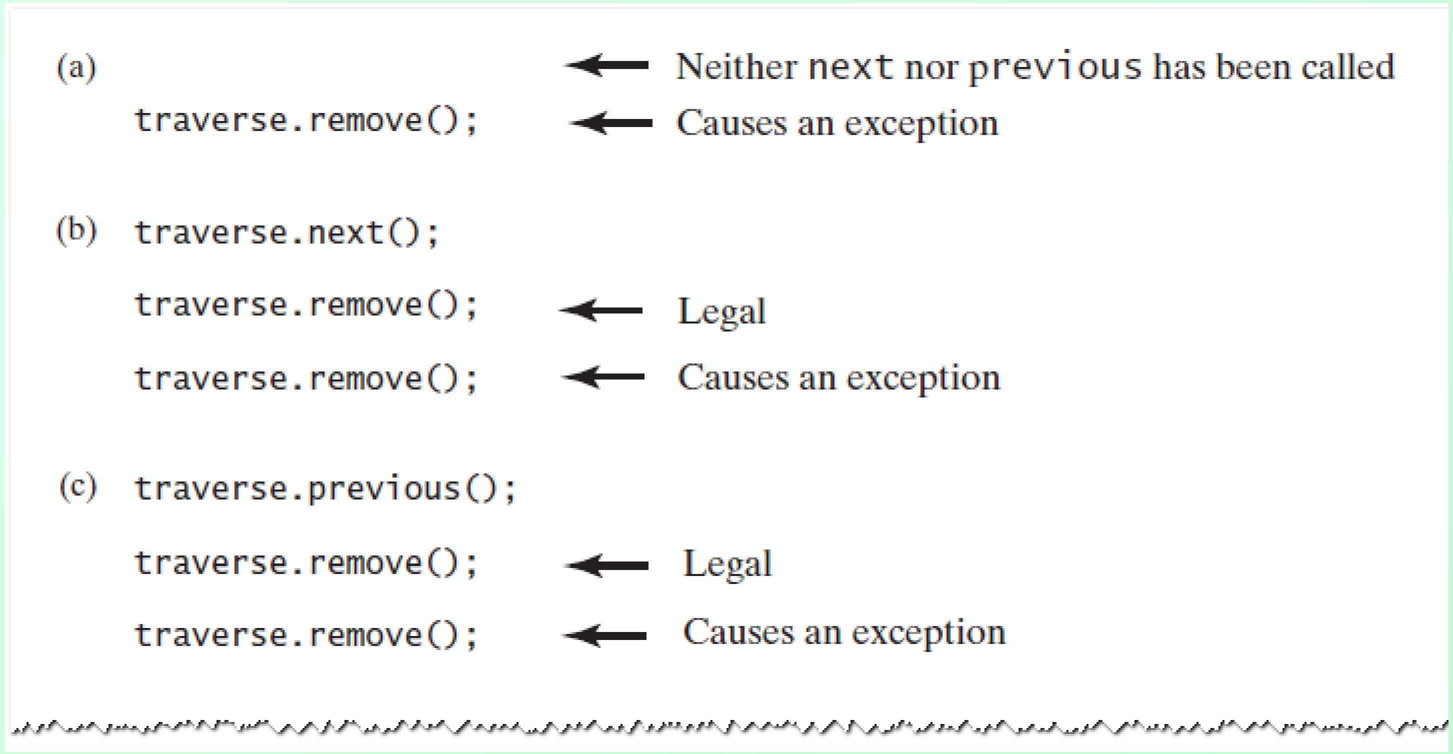
- The interface **`ListWithListIteratorInterface`**
- Listing 15-8



```
import java.util.ListIterator;
public interface ListWithListIteratorInterface<T> extends
                                                     ListInterface<T>
{
   public ListIterator<T> getIterator();
} // end ListWithListIteratorInterface
```

Array-Based Implementation of Interface **`ListIterator`**

- The class that implements the ADT list
  Listing 15-9
- Class **`ArrayListWithListIterator`**
  Listing 15-10
- Consider how remove and set will throw **`IllegalStateException`**.
  - Happens when
    - **`next`** or **`previous`** was not called, or
    - **`remove`** or **`add`** has been called since the last call to **`next`** or **`previous`**

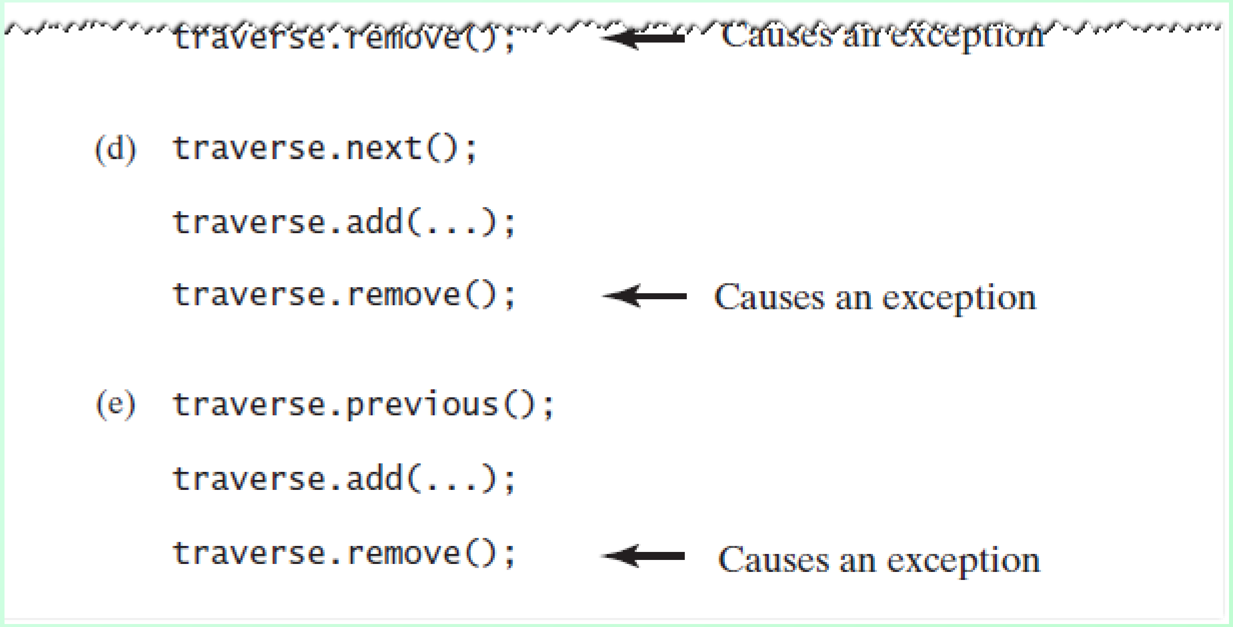


```
(a)                              ← Neither next nor previous has been called
    traverse.remove();           ← Causes an exception

(b) traverse.next();
    traverse.remove();           ← Legal
    traverse.remove();           ← Causes an exception

(c) traverse.previous();
    traverse.remove();           ← Legal
    traverse.remove();           ← Causes an exception
```

Figure 15-11 Possible contexts in which the method **`remove`** of the iterator **`traverse`** throws an exception when called



```
    traverse.remove();        ←  Causes an exception

(d) traverse.next();
    traverse.add(...);
    traverse.remove();        ←  Causes an exception

(e) traverse.previous();
    traverse.add(...);
    traverse.remove();        ←  Causes an exception
```

Figure 15-11 Possible contexts in which the method **`remove`** of the iterator **`traverse`** throws an exception when called

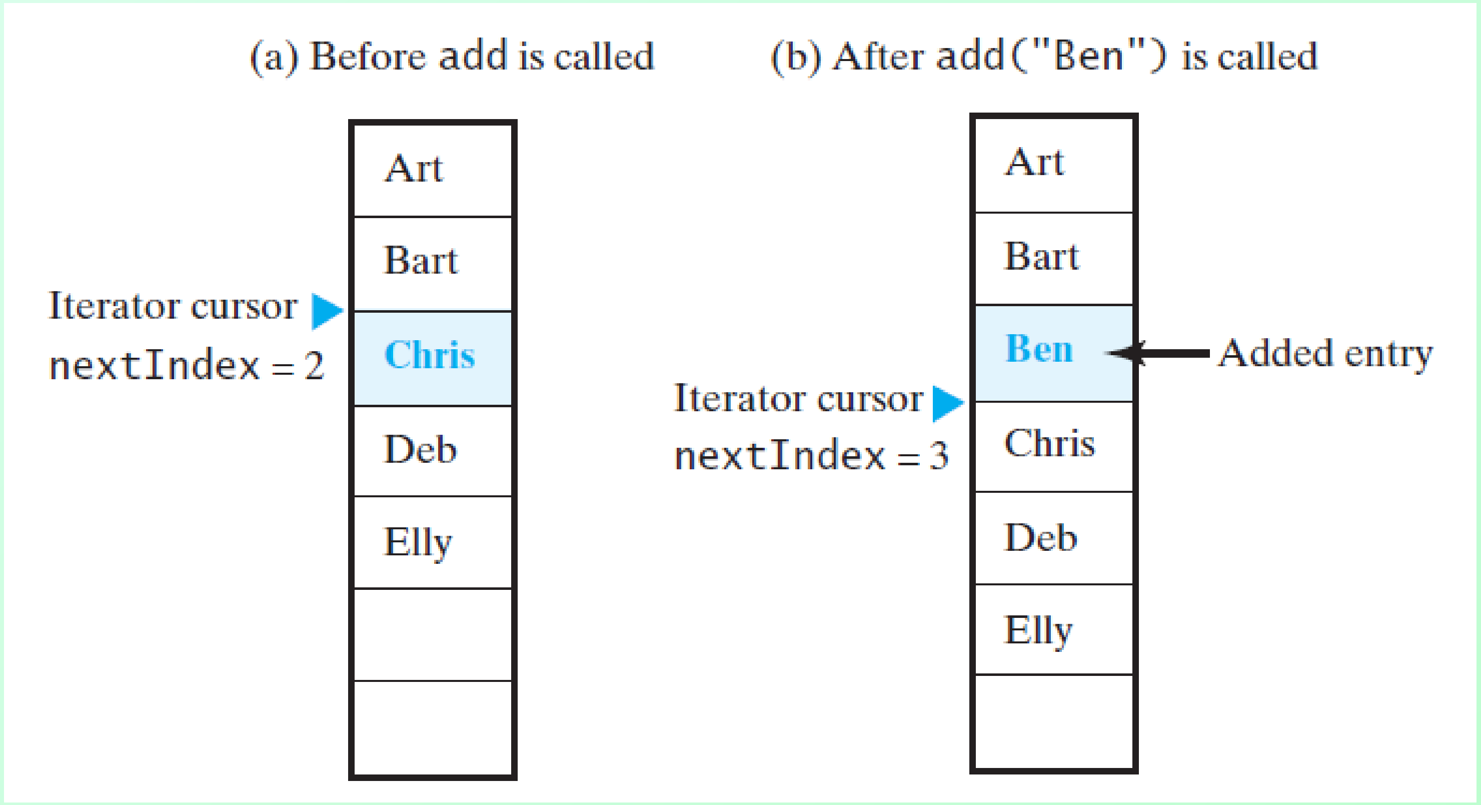


Figure 15-12 The array of list entries and **`nextIndex`** (a) just before the call to **`add`**; (b) just after the call to **`add`**

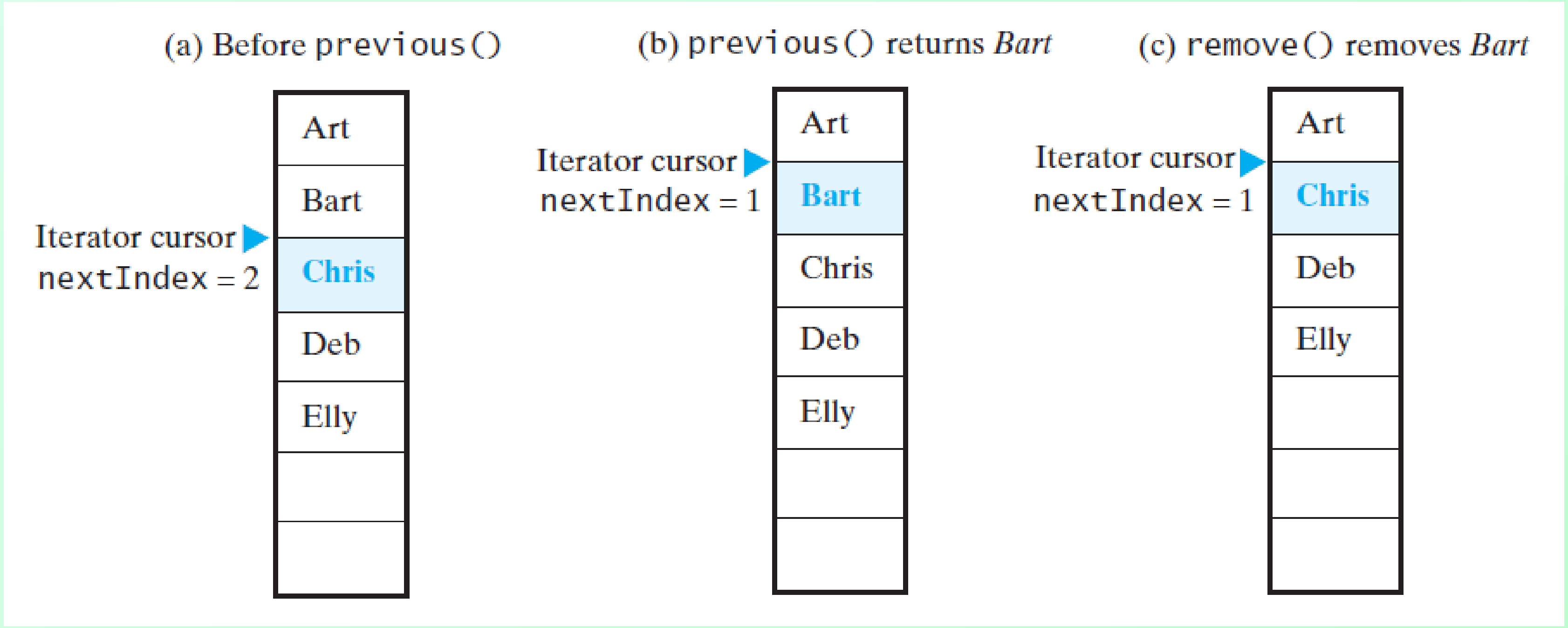
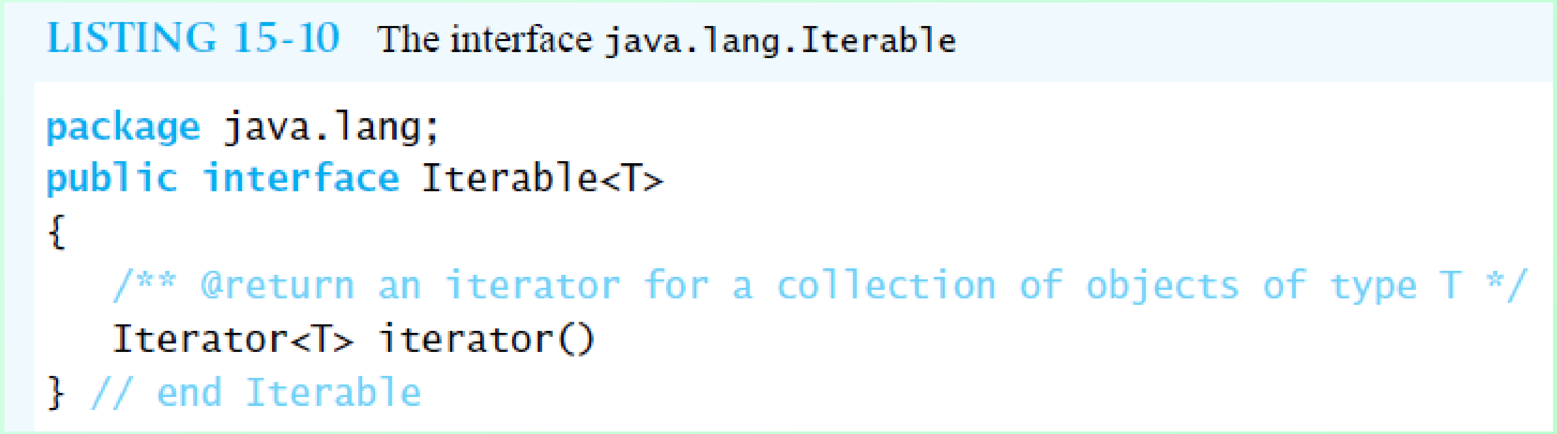


Figure 15-13 The array of list entries and **`nextIndex`** (a) just before the call to **`previous`**; (b) just after the call to **`previous`** but before the call to **`remove`**; (c) after the call to **`remove`**

Java Class Library: The Interface **`Iterable`**

- Listing 15-11

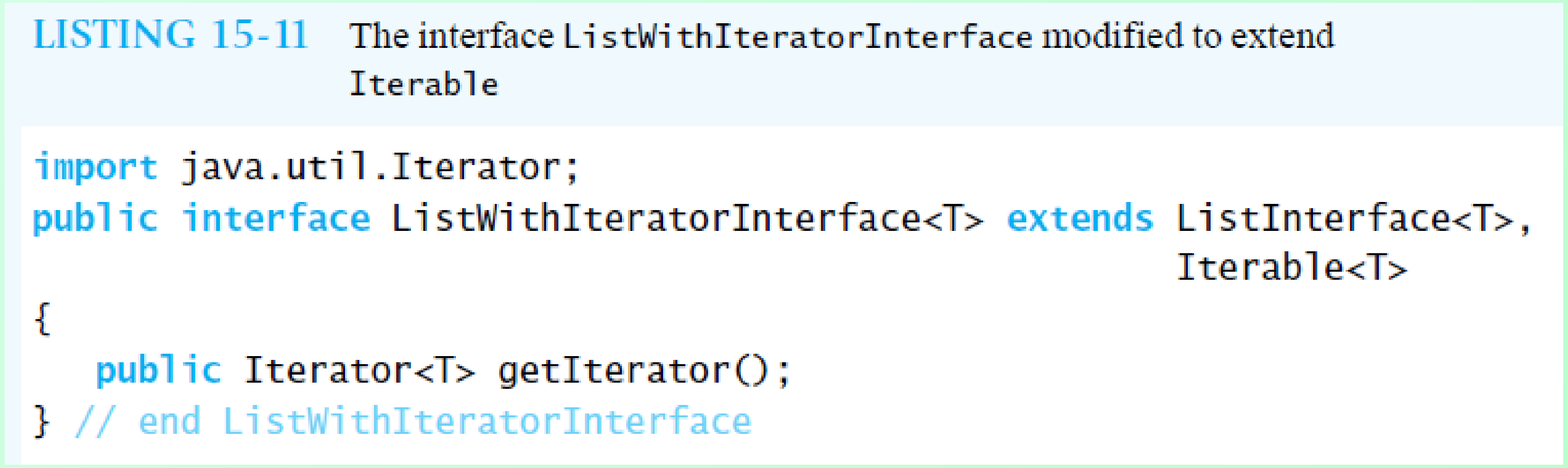


LISTING 15-10 The interface `java.lang.Iterable`

```
package java.lang;
public interface Iterable<T>
{
   /** @return an iterator for a collection of objects of type T */
   Iterator<T> iterator()
} // end Iterable
```

Java Class Library: The Interface **`Iterable`**

- Listing 15-12



LISTING 15-11 The interface `ListWithIteratorInterface` modified to extend `Iterable`

```
import java.util.Iterator;
public interface ListWithIteratorInterface<T> extends ListInterface<T>,
                                                      Iterable<T>
{
   public Iterator<T> getIterator();
} // end ListWithIteratorInterface
```

`Iterable` and for-each Loops

- Can use a for-each loop to traverse
- Given

```
ListWithIteratorInterface<String> nameList =
                          new LinkedListWithIterator<String>();
nameList.add("Joe");
nameList.add("Jess");
nameList.add("Josh");
nameList.add("Jen");
```

- Then

```
for (String name : nameList)
   System.out.print(name + " ");
System.out.println();
```

End

Chapter 15