Class 05 – Lists with Iterators

CSIS 3475 Data Structures and Algorithms

What Is an Iterator?

- An object that traverses a collection of data
- During iteration, each data item is considered once
  - Possible to modify item as accessed
- Should implement as a distinct class that interacts with the ADT

The Java Interface `Iterator`

```
package java.util;
public interface Iterator<T>
  {
    /** Detects whether this iterator has completed its traversal
     and gone beyond the last entry in the collection of data.
     @return True if the iterator has another entry to return. */
    public boolean hasNext();

    /** Retrieves the next entry in the collection and
     advances this iterator by one position.
     @return A reference to the next entry in the iteration,
     if one exists.
     @throws NoSuchElementException if the iterator had reached the
     end already, that is, if hasNext() is false. */
    public T next();

    /** Removes from the collection of data the last entry that
     next() returned. A subsequent call to next() will behave
     as it would have before the removal.
     Precondition: next() has been called, and remove() has not
     been called since then. The collection has not been altered
     during the iteration except by calls to this method.
     @throws IllegalStateException if next() has not been called, or
     if remove() was called already after the last call to next().
     @throws UnsupportedOperationException if the iterator does
     not permit a remove operation. */
    public void remove(); // Optional method
  } // end Iterator
```

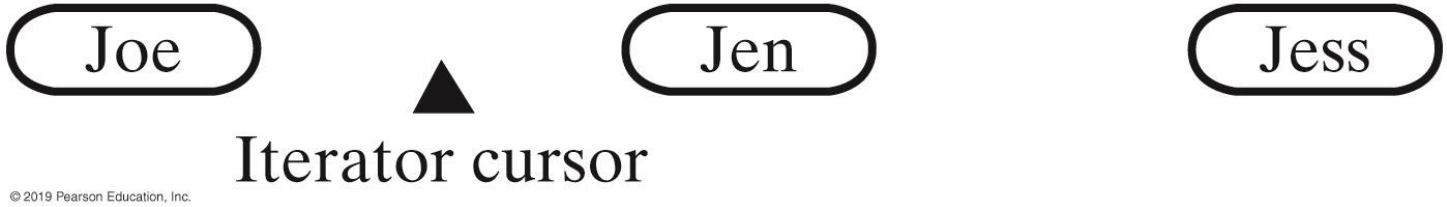
The Java Interface `Iterator`

- Possible positions of an iterator's cursor within a collection

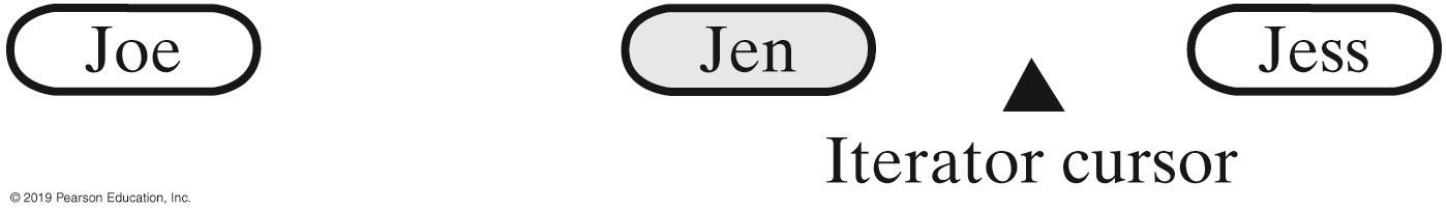
The Java Interface Iterator

- Effect on a collection's iterator by a call to `next` and subsequent call to `remove`

(a) Before `next()` exceutes



(b) After `next()` returns *Jen*



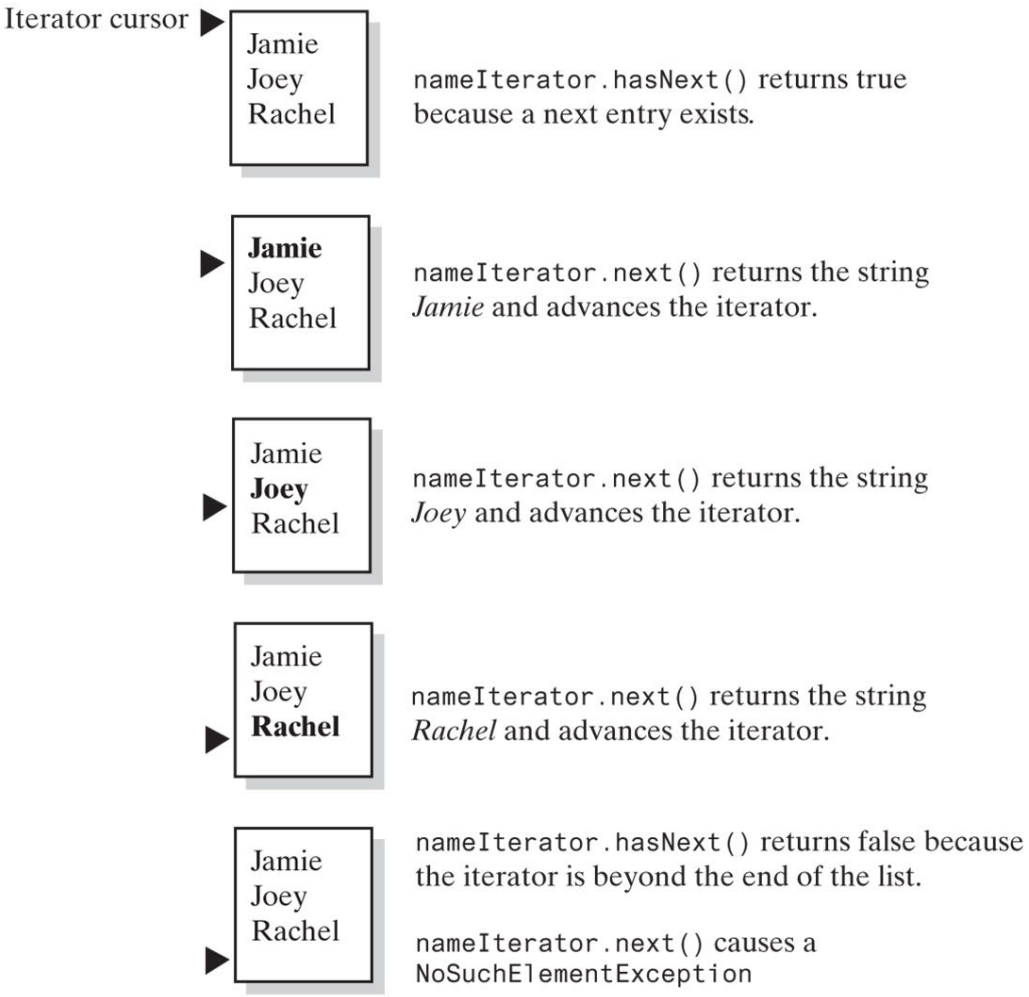
(c) After a subsequent `remove()` deletes *Jen*

The Interface `Iterable`

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

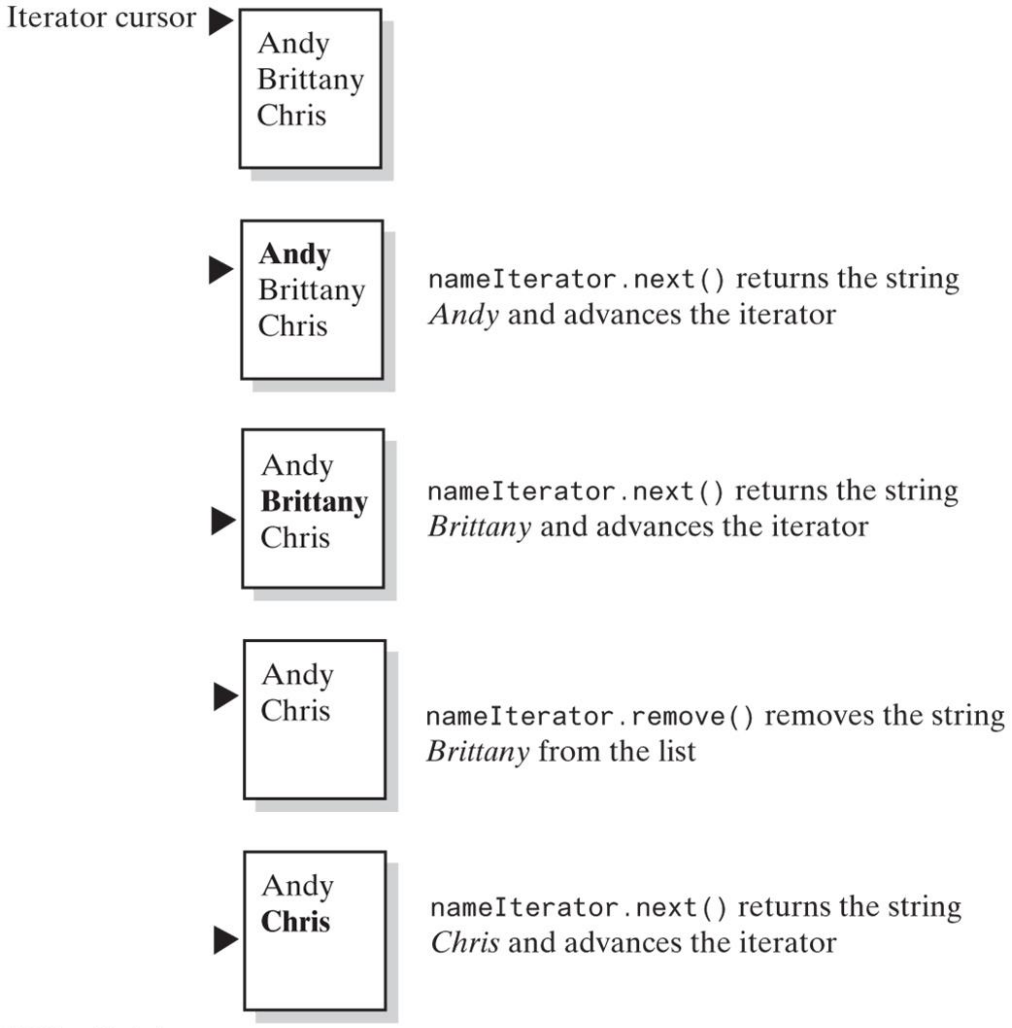
Using the Java Interface Iterator

- The effect of the iterator methods `hasNext` and `next` on a list



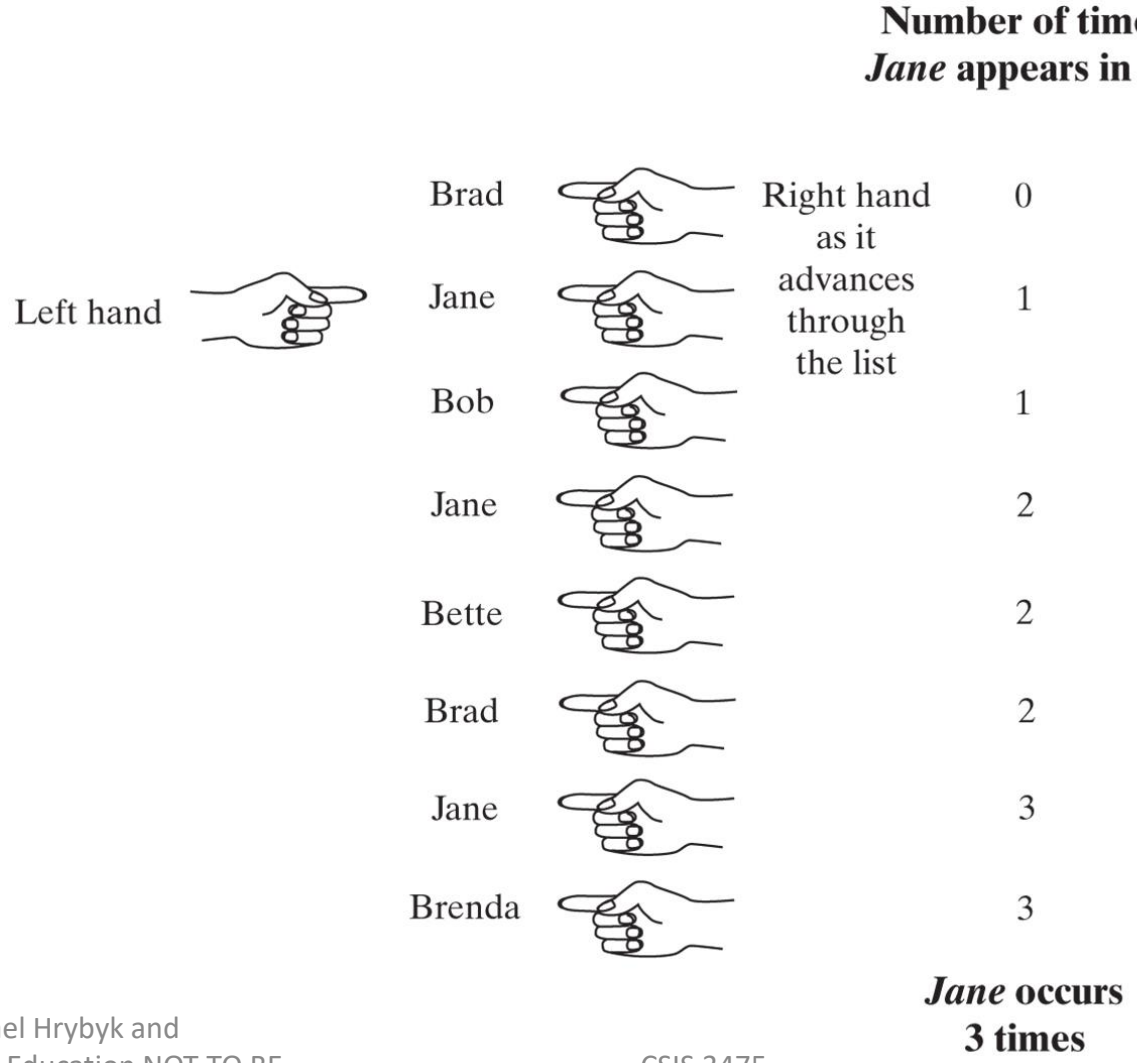
Using the Java Interface Iterator

- The effect of the iterator methods `next` and `remove` on a list



Using the Java Interface Iterator

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



Multiple Iterators

- Counts the occurrence of each name
- See [AL]ListWithIteratorDemo

```
Iterator<String> nameIterator = nameList.iterator();
while (nameIterator.hasNext()) {
    String currentName = nameIterator.next();
    int nameCount = 0;
    Iterator<String> countingIterator = nameList.iterator();
    while (countingIterator.hasNext()) {
        String nextName = countingIterator.next();
        if (currentName.equals(nextName))
            nameCount++;
    }
    System.out.println(currentName + " occurs " + nameCount + " times.");
}
```

```
package java.util;
public interface ListIterator<T> extends Iterator<T>
  {
    /** Detects whether this iterator has gone beyond the last
     entry in the list.
     @return True if the iterator has another entry to return when
     traversing the list forward; otherwise returns false. */
    public boolean hasNext();

    /** Retrieves the next entry in the list and
     advances this iterator by one position.
     @return A reference to the next entry in the iteration,
     if one exists.
     @throws NoSuchElementException if the iterator is at the end,
     that is, if hasNext() is false. */
    public T next();

    /** Removes from the list the last entry that either next()
     or previous() has returned.
     Precondition: next() or previous() has been called, but the
     iterator's remove() or add() method has not been called
     since then. That is, you can call remove only once per
     call to next() or previous(). The list has not been altered
     during the iteration except by calls to the iterator's
     remove(), add(), or set() methods.
     @throws IllegalStateException if next() or previous() has not
     been called, or if remove() or add() has been called
     already after the last call to next() or previous().
     @throws UnsupportedOperationException if the iterator does not
     permit a remove operation. */
    public void remove(); // Optional method
```

Java's interface `java.util.ListIterator`

These three methods are in the interface **`Iterator`**; they are duplicated here for reference and to show new behavior for **`remove`**.

Java's interface `java.util.ListIterator`

```
/** Detects whether this iterator has gone before the first
 entry in the list.
 @return True if the iterator has another entry to visit when
 traversing the list backward; otherwise returns false. */
public boolean hasPrevious();

/** Retrieves the previous entry in the list and moves this
 iterator back by one position.
 @return A reference to the previous entry in the iteration, if
 one exists.
 @throws NoSuchElementException if the iterator has no previous
 entry, that is, if hasPrevious() is false. */
public T previous();

/** Gets the index of the next entry.
 @return The index of the list entry that a subsequent call to
 next() would return. If next() would not return an entry
 because the iterator is at the end of the list, returns
 the size of the list. Note that the iterator numbers
 the list entries from 0 instead of 1. */
public int nextIndex();

/** Gets the index of the previous entry.
 @return The index of the list entry that a subsequent call to
 previous() would return. If previous() would not return
 an entry because the iterator is at the beginning of the
 list, returns –1. Note that the iterator numbers the
 list entries from 0 instead of 1. */
public int previousIndex();
```

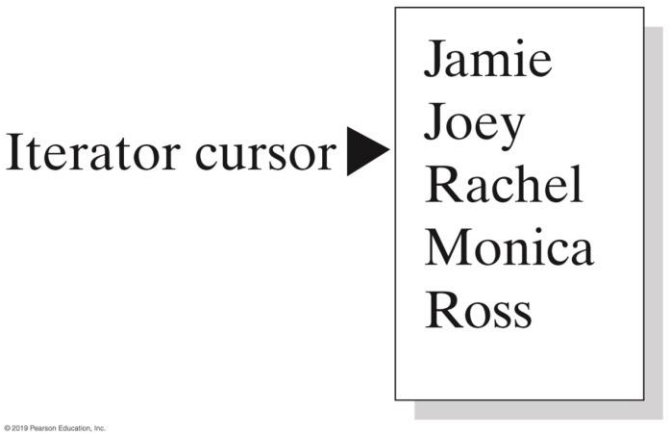
Java's interface `java.util.ListIterator`

```
/** Adds an entry to the list just before the entry, if any,
 that next() would have returned before the addition. This
 addition is just after the entry, if any, that previous()
 would have returned. After the addition, a call to
 previous() will return the new entry, but a call to next()
 will behave as it would have before the addition.
 Further, the addition increases by 1 the values that
 nextIndex() and previousIndex() will return.
 @param newEntry An object to be added to the list.
 @throws ClassCastException if the class of newEntry prevents the
 addition to the list.
 @throws IllegalArgumentException if some other aspect of
 newEntry prevents the addition to the list.
 @throws UnsupportedOperationException if the iterator does not
 permit an add operation. */
public void add(T newEntry); // Optional method

/** Replaces the last entry in the list that either next()
 or previous() has returned.
 Precondition: next() or previous() has been called, but the
 iterator's remove() or add() method has not been called since then.
 @param newEntry An object that is the replacement entry.
 @throws ClassCastException if the class of newEntry prevents the
 addition to the list.
 @throws IllegalArgumentException if some other aspect of newEntry
                                  prevents the addition to the list.
 @throws IllegalStateException if next() or previous() has not been called,
                               or if remove() or add() has been called already
                               after the last call to next() or previous().
 @throws UnsupportedOperationException if the iterator does not permit a set operation. */
public void set(T newEntry); // Optional method
} // end ListIterator
```

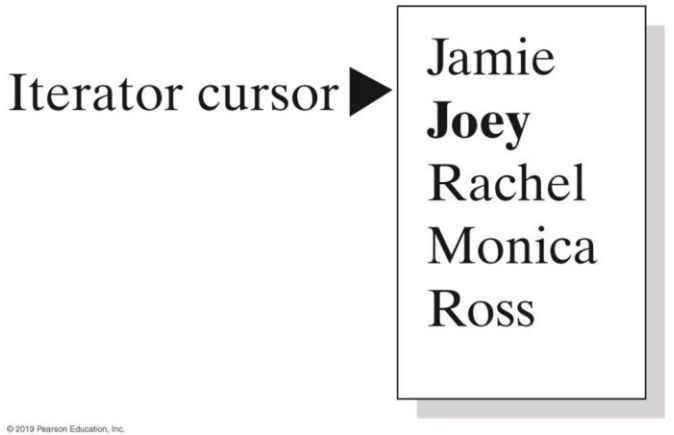
Using the Java Interface `ListIterator`

- The effect of a call to previous on a list

(a) Before `previous()`

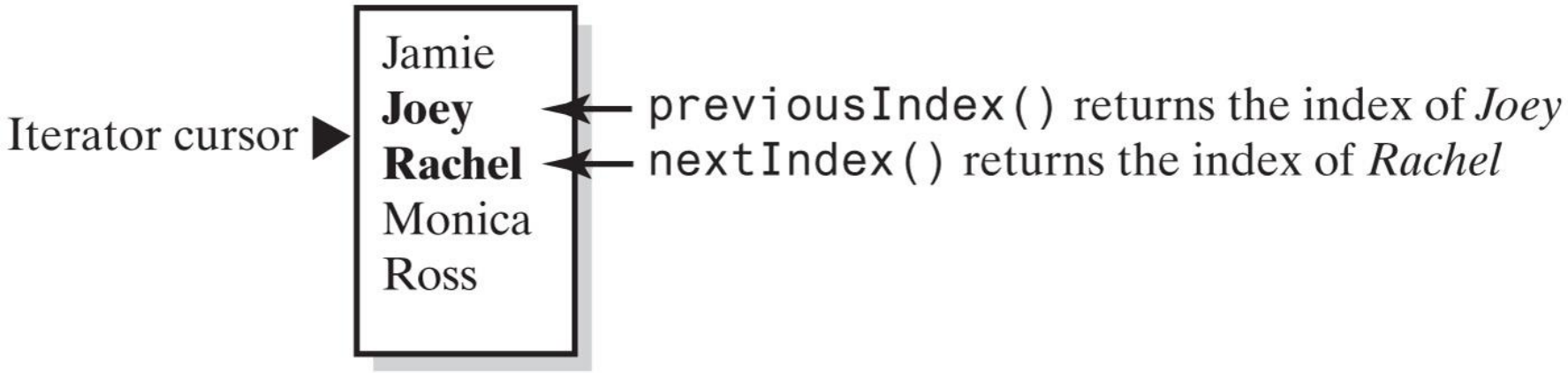


(b) After `previous()` returns *Joey*



Using the Java Interface `ListIterator`

- The indices returned by the methods `nextIndex` and `previousIndex`



The Interface `List` Revisited

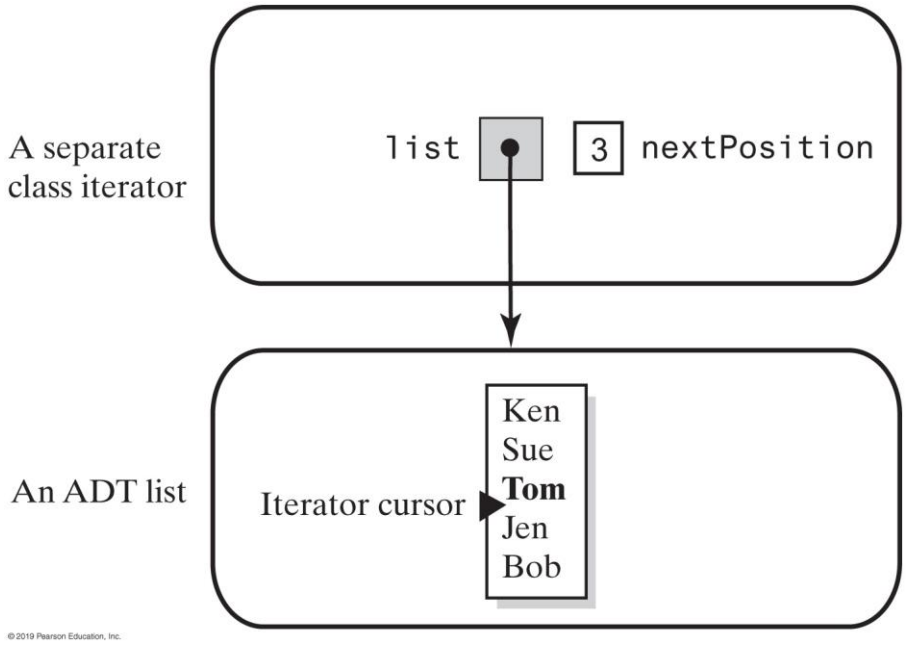
- Method set replaces entry that either next or previous just returned.
- Method add inserts an entry into list just before iterator's current position
- Method remove removes list entry that last call to either next or previous returned

Iterators

- An iterator
  - An object that enables you to traverse entries in a data collection
- Possible way to provide an ADT with traversal operations
  - Define them as ADT operations
- Better way
  - Implement the iterator methods within their own class

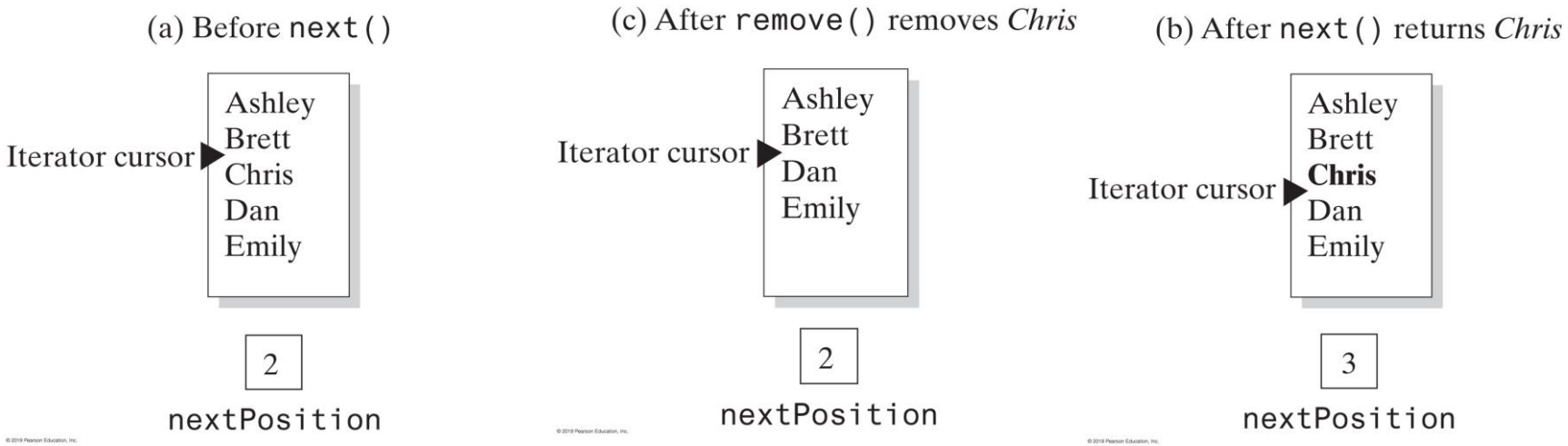
Separate Class Iterator

- 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



Separate Class Iterator

- Changes to a list and `nextPosition` when removing Chris from the list



SeparateIterator class

```
public class CompletedSeparateListIterator<T> implements Iterator<T> {
    private ListInterface<T> list;
    private int nextPosition; // Position of entry last returned by next()
    private boolean wasNextCalled; // Needed by remove

    public CompletedSeparateListIterator(ListInterface<T> myList) {
        list = myList;
        nextPosition = 0;
        wasNextCalled = false;
    }

    public boolean hasNext() {
        return nextPosition < list.size();
    }

    public T next() {
        if (hasNext()) {
            wasNextCalled = true;
            T nextEntry = list.getEntry(nextPosition);
            nextPosition++;
            return nextEntry;
        } else {
            throw new NoSuchElementException("No next element in list");
        }
    }

    public void remove() {
        if (wasNextCalled) {
            // nextPosition was incremented by the call to next(), so
            // it is the position number of the entry to be removed
            list.remove(nextPosition - 1);
            nextPosition--; // A subsequent call to next() must be
                            // unaffected by this removal
            wasNextCalled = false; // Reset flag
        } else
            throw new IllegalStateException("Illegal call to remove(); " + "next() was not called.");
    }
}
```

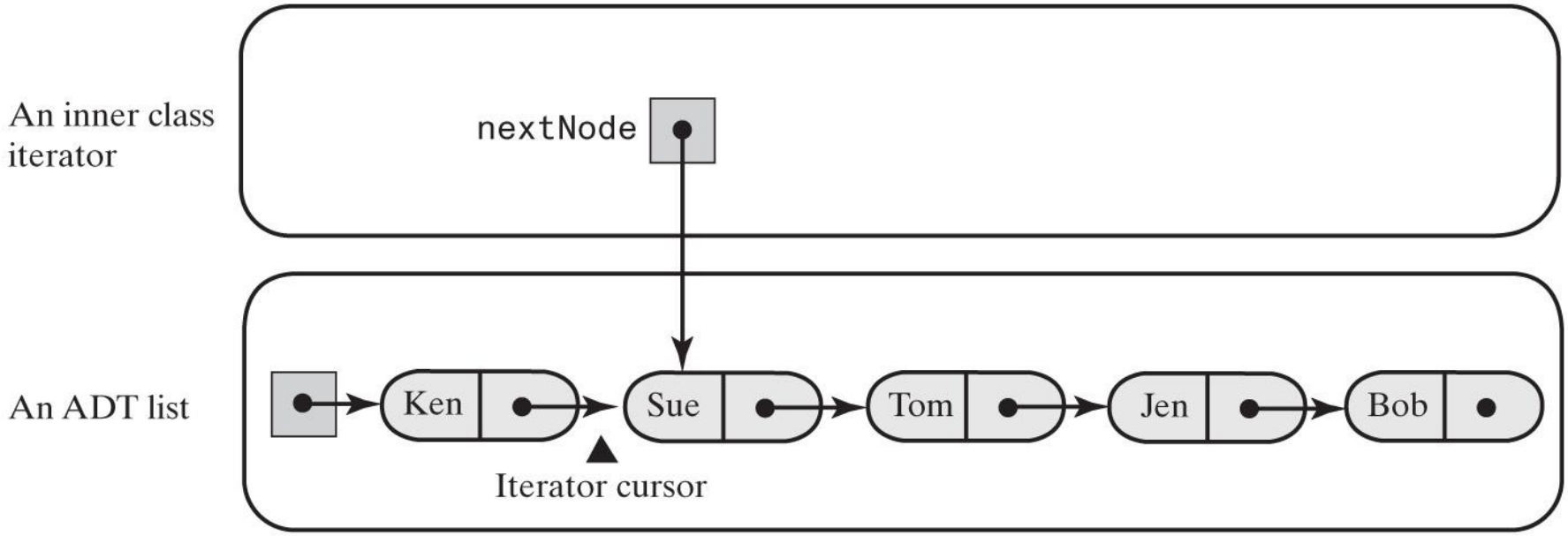
SeparateIterator class usage

- See LListSeparateIteratorDemo testIteratorOperations()

```
        System.out.println("Create a list: ");
        ListInterface<String> myList = new CompletedAList<>();
        System.out.println("Testing add to end: Add 15, 25, 35, 45, 55, 65, 75, 85, 95");
        myList.add("15");
        myList.add("25");
        myList.add("35");
        myList.add("45");
        myList.add("55");
        myList.add("65");
        myList.add("75");
        myList.add("85");
        myList.add("95");

        System.out.println("\n------------------------\n");
        System.out.println("Testing Iterator's hasNext and next methods:");

        System.out.println("\n\nList should contain\n15 25 35 45 55 65 75 85 95");
        System.out.println("\n\nUsing ADT list operations, the list contains ");
        displayList(myList);

        System.out.println("\n\nUsing Iterator methods, the list contains");
        Iterator<String> myIterator = new CompletedSeparateIterator<>(myList);

        while (myIterator.hasNext())
            System.out.print(myIterator.next() + " ");
```

Inner Class Iterator

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



Inner Class Iterator

- Simple interface, just returns an Iterator to be used on the list

```
import java.util.Iterator;

/**
 * An interface for the ADT list that has an iterator.
 *
 * @author Frank M. Carrano
 * @author Timothy M. Henry
 * @version 5.0
 */
public interface ListWithIteratorInterface<T> extends ListInterface<T>, Iterable<T> {
          public Iterator<T> getIterator();
}
```

LListWithTraversal – using inner class

- Different from text – extends LList.
- But then has no access to head of list, so need to get first node for traversal – NOT GOOD PRACTICE

```
public class CompletedLListWithTraversal<T extends Comparable<? super T>>
    extends CompletedLList<T> implements Iterator<T> {

    private Node<T> nextNode; // Node containing next entry in iteration

    public CompletedLListWithTraversal() {
        super();
        resetTraversal();
    }

    public boolean hasNext() {
        return nextNode != null;
    }

    public T next() {
        if (hasNext()) {
            Node<T> returnNode = nextNode; // Get next node
            nextNode = nextNode.getNextNode(); // Advance iterator

            return returnNode.getData(); // Return next entry in iteration
        } else {
            throw new NoSuchElementException("No next element in list");
        }
    }

    /**
     * Sets the traversal to the beginning of the list. This method is NOT in the
     * interface Iterator.
     */
    public void resetTraversal() {
        nextNode = super.getFirstNode();
    }



}
```

Use of LListWithTraversal

- See LListWithTraversalDemo

```
CompletedLListWithTraversal<String> myList = new CompletedLListWithTraversal<>();
System.out.println("Testing add to end: Add 15, 25, 35, 45, 55, 65, 75, 85, 95");
myList.add("15");
myList.add("25");
myList.add("35");
myList.add("45");
myList.add("55");
myList.add("65");
myList.add("75");
myList.add("85");
myList.add("95");

DemoUtilities.displayUsingGetEntry(myList, "List should be: 15, 25, 35, 45, 55, 65, 75, 85, 95");

System.out.println("Testing Iterator's hasNext and next methods:");

System.out.println("Using Iterator methods, the list contains: ");
myList.resetTraversal();

while (myList.hasNext())
      System.out.print(myList.next() + ", ");
System.out.println();


System.out.println("Return iterator to beginning of list");
myList.resetTraversal(); // Reset iterator to beginning
```

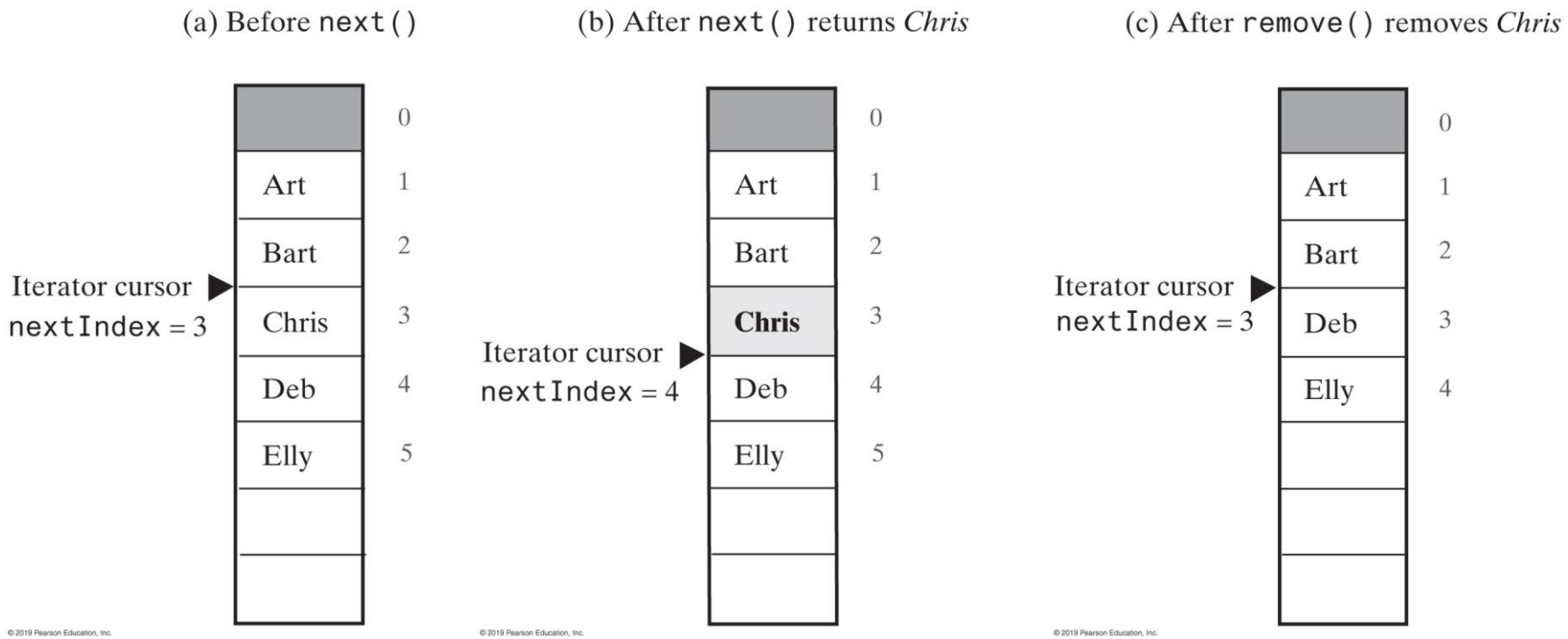
Array-Based Implementation of the Interface `ListIterator`

- The interface `ListWithListIteratorInterface`

```
/**  An interface for the ADT list that has an iterator implementing
   the interface ListIterator.  */
public interface ListWithListIteratorInterface<T>
                                    extends Iterable<T>, ListInterface<T>
{
   public ListIterator<T> getIterator();
} // end ListWithListIteratorInterface
```

Iterators for Array-Based Lists

- Changes to the array of list entries and `nextIndex` when removing Chris from the list



AListWithIterator

- Same as LList, this class simply extends AList
- This works better than LList because we have access to indices

```
public class CompletedAListWithIterator<T extends Comparable<? super T>>
   extends CompletedAList<T> implements ListWithIteratorInterface<T> {

   @Override
   public Iterator<T> iterator() {
      return new IteratorForArrayList();
   }

   @Override
   public Iterator<T> getIterator() {
      return iterator();
   }

// other methods are here, see next slide


}
```

Inner Iterator class for AList

```
private class IteratorForArrayList implements Iterator<T> {
    private int nextIndex; // Index of next entry in the iteration
    private boolean wasNextCalled; // Needed by remove

    private IteratorForArrayList() {
        nextIndex = 0; // Iteration begins at list's first entry
        wasNextCalled = false;
    }

    public boolean hasNext() {
        return nextIndex < size();
    }

    public T next() {
        if (hasNext()) {
            wasNextCalled = true;
            T nextEntry = getEntry(nextIndex);
            nextIndex++; // Advance iterator

            return nextEntry;
        } else {
            throw new NoSuchElementException("No next element in list");
        }

    }

    public void remove() {
        if (wasNextCalled) {
            // nextIndex was incremented by the call to next, so it is
            // 1 larger than the position number of the entry to be removed
            CompletedAListWithIterator.this.remove(nextIndex - 1);
            nextIndex--; // Index of next entry in iteration
            wasNextCalled = false; // Reset flag
        } else
            throw new IllegalStateException("Illegal call to remove(); " + "next() was not called.");
    }
}
```

Why Are Iterator Methods in Their Own Class?

- These traversal methods can execute quickly
  - They have direct access to the underlying data structure
- Disadvantages
  - Only one traversal at a time
  - Operation such as resetTraversal necessary – “interface bloat”

Using ListIterator

- ListIterator is an Iterator for the Java library List interface classes.
- Has the full set of methods.
- Implement as an inner class
- Need to track whether we are moving backwards or forwards, so keep state
  - See AListWithListIterator

```
public class CompletedAListWithListIterator<T extends Comparable<? super T>>
          extends CompletedAList<T>
          implements ListWithListIteratorInterface<T> {

     public ListIterator<T> getIterator() {
          return new ListIteratorForAList();
     }

     public Iterator<T> iterator() {
          return getIterator();
     }

     /**
      * Movement direction.
      *
      */
     private enum Move {
          NEXT, PREVIOUS
     }
```

next() – keeping forward state as we move

```
private class ListIteratorForAList implements ListIterator<T> {
     private int nextIndex; // Index of next entry in the iteration
     private boolean isRemoveOrSetLegal;
     private Move lastMove;

     private ListIteratorForAList() {
          nextIndex = 0; // Iteration begins at list's first entry
          isRemoveOrSetLegal = false;
          lastMove = null;
     }

     public boolean hasNext() {
          return nextIndex < size();
     }

     public T next() {
          if (hasNext()) {
               lastMove = Move.NEXT;
               isRemoveOrSetLegal = true;

               T nextEntry = getEntry(nextIndex);
               nextIndex++; // Advance iterator

               return nextEntry;
             } else {
                  throw new NoSuchElementException("No next element in list");
             }

     }
```

previous()

```
public boolean hasPrevious() {
    // is there a prior slot?
    // if nextIndex is past the end of the list, there is.
    return (nextIndex > 0) && (nextIndex <= size());
}

public T previous() {
    if (hasPrevious()) {
        // reset the direction
        lastMove = Move.PREVIOUS;
        isRemoveOrSetLegal = true;

        T previousEntry = getEntry(nextIndex - 1);

        nextIndex--; // Move iterator back
        return previousEntry;
    } else
        throw new NoSuchElementException(
                "Illegal call to previous(); " + "iterator is before beginning of list.");
}
```

Inner Class Iterator for Array-Based Lists (Part 1)

- Possible contexts in which the method `remove` of the iterator traversal throws an exception when called

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

(b)
```
traverse.next();
traverse.remove();
```
← Legal

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

(c)
```
traverse.previous();
traverse.remove();
```
← Legal

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

Inner Class Iterator for Array-Based Lists (Part 2)

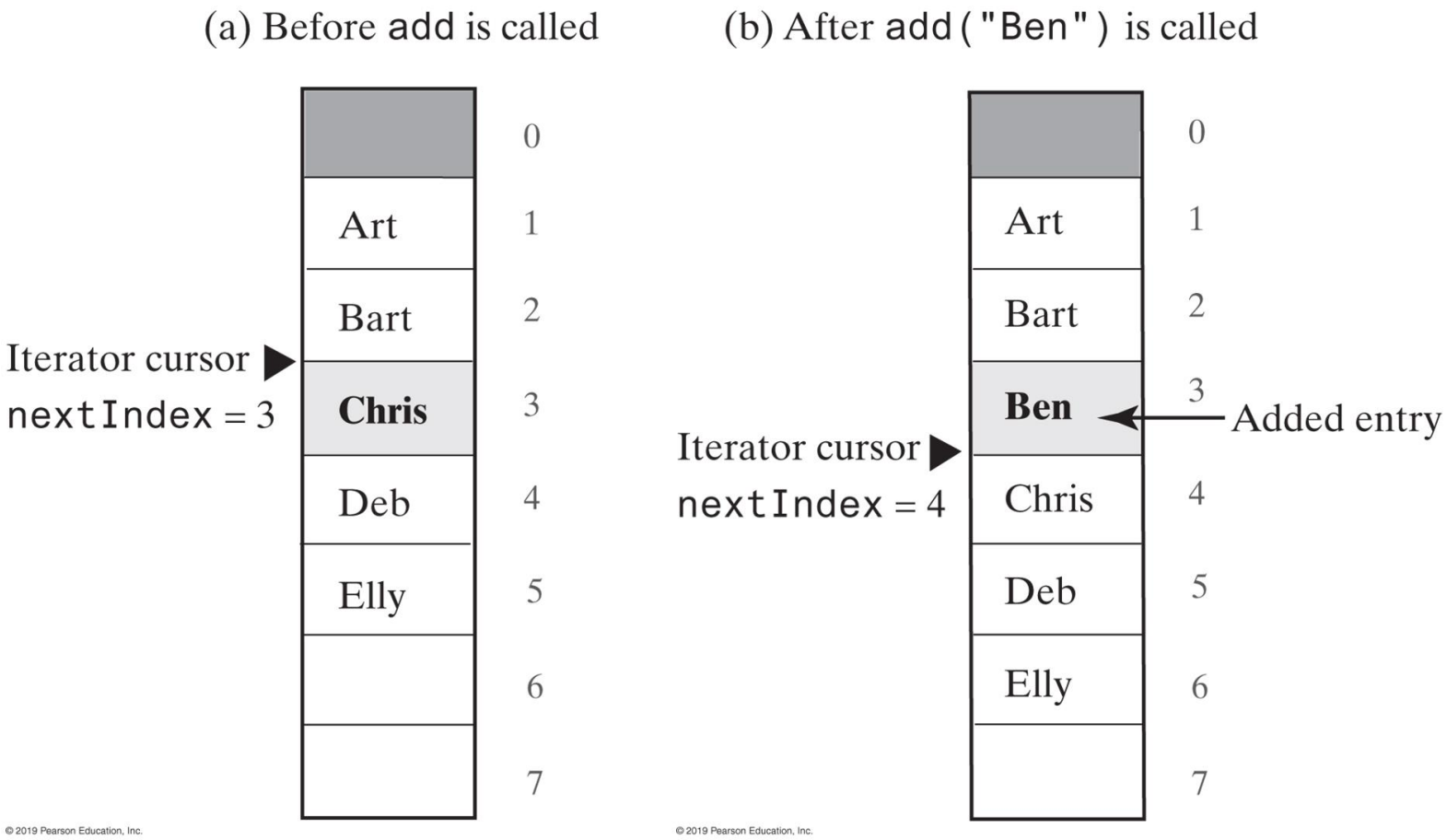
- Possible contexts in which the method `remove` of the iterator traversal throws an exception when called

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

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

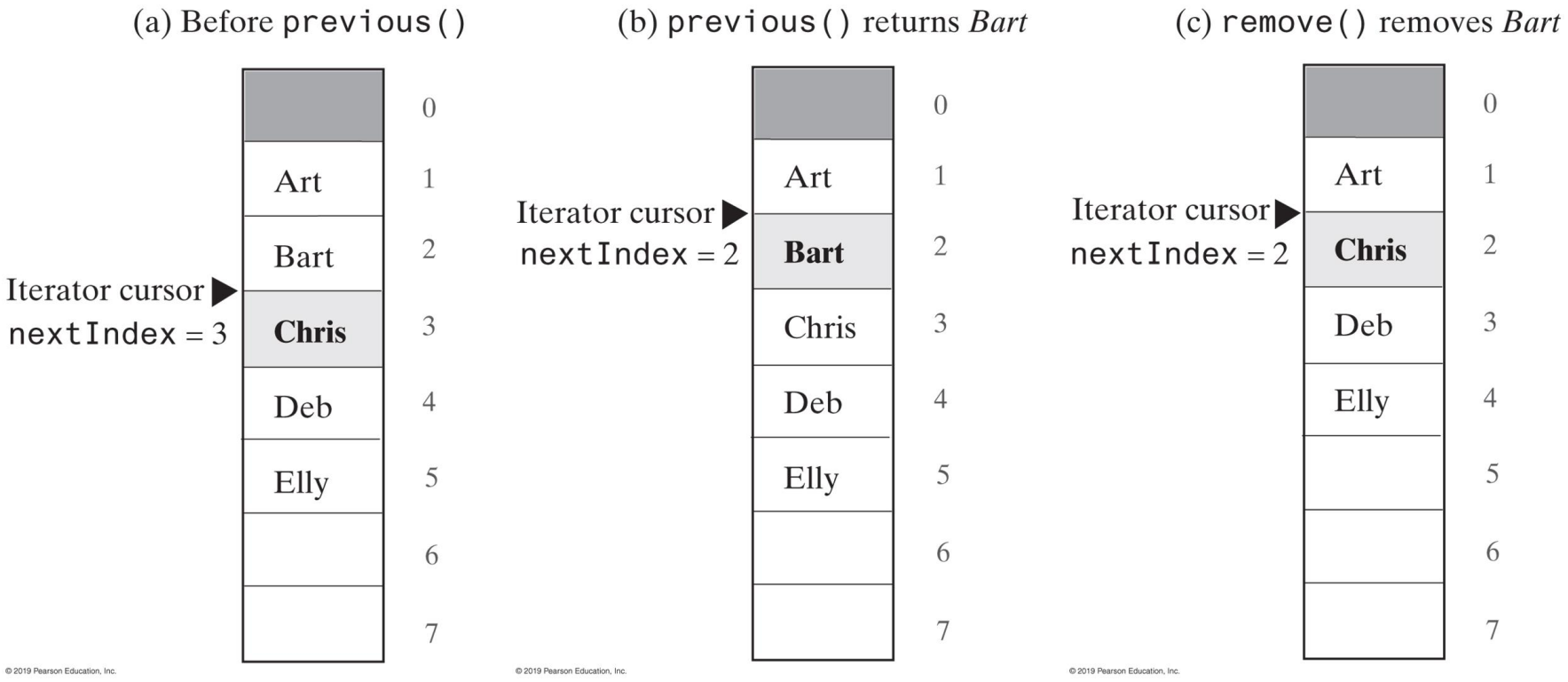
The Inner Class

- Changes to the array of list entries and `nextIndex` when adding Ben to the list



The Inner Class

- Changes to the array of list entries and `nextIndex` when removing Chris from the list



next/previous index(), add()

- add() just calls the outer class's add method

```
public int nextIndex() {
     int result;

     if (hasNext())
          result = nextIndex;
     else
          result = size(); // End-of-list flag

     return result;
}

public int previousIndex() {
     int result;

     if (hasPrevious())
          result = nextIndex - 1;
     else
          result = -1; // Beginning-of-list flag

     return result;
}

public void add(T newEntry) {

     // once we add, our direction is no longer set.

     isRemoveOrSetLegal = false;

     // Insert newEntry immediately before the the iterator's current position
     CompletedAListWithListIterator.this.add(nextIndex, newEntry);
     nextIndex++;
}
```

remove()

- Keep track of state, depends on whether next() or previous has been called.
- If neither, then we have a problem

```
public void remove() {
   if (isRemoveOrSetLegal) {
      isRemoveOrSetLegal = false;

      if (lastMove.equals(Move.NEXT)) {
         // next() called, but neither add() nor remove() has been
         // called since.

         // Remove entry last returned by next().

         // nextIndex is 1 more than the index of the entry
         // returned by next()
         CompletedAListWithListIterator.this.remove(nextIndex - 1);
         nextIndex--; // Move iterator back
      } else {
         // previous() called, but neither add() nor remove() has been
         // called since

         // Remove entry last returned by previous().

         // nextIndex is the index of the entry returned by previous().
         CompletedAListWithListIterator.this.remove(nextIndex);
      }
   } else
      throw new IllegalStateException("Illegal call to remove(); " + "next() or previous() not called, OR "
           + "add() or remove() called since then.");
}
```

set()

- similar to remove(), we need to keep track of state in order to determine which entry to replace.

```
    public void set(T newEntry) {
       if (isRemoveOrSetLegal) {
          if (lastMove.equals(Move.NEXT)) {
             replace(nextIndex - 1, newEntry);
          } else {
             replace(nextIndex, newEntry);
          }
       } else
          throw new IllegalStateException("Illegal call to set(); " + "next() or previous() not called,
OR "
                + "add() or remove() called since then.");
    }
```