Class 05 – Lists

CSIS 3475 Data Structures and Algorithms

Lists

- A way to organize data
- Examples
  - To-do list
  - Gift lists
  - Grocery Lists
- Items in list have position
  - May or may not be important
- Items may be added anywhere

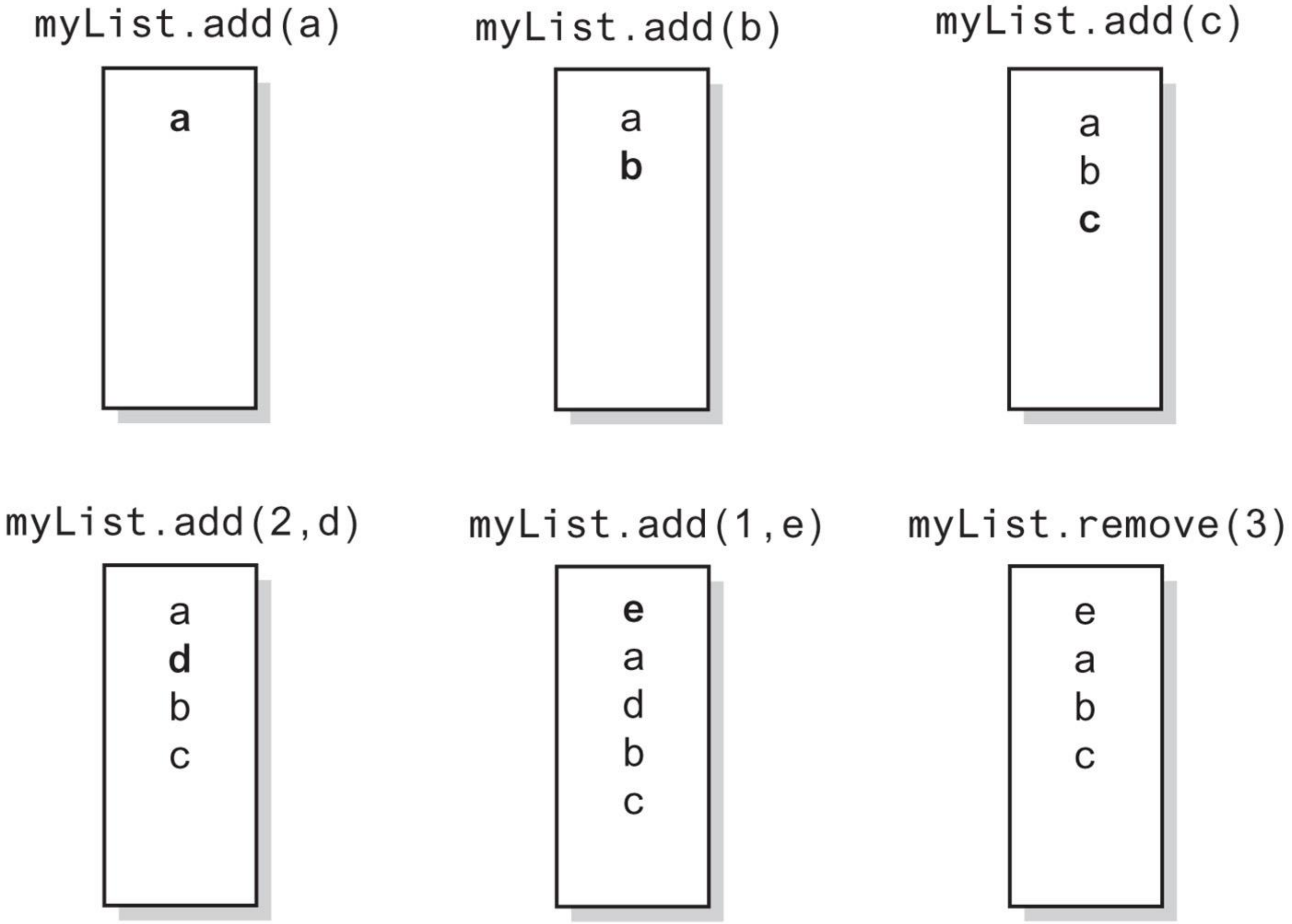
FIGURE 10-1
A to-do list

Specifications for the ADT List

- **`add(newEntry)`**
- **`add(newPosition, newEntry)`**
- **`remove(givenPosition)`**
- **`clear()`**
- **`replace(givenPosition, newEntry)`**
- **`getEntry(givenPosition)`**
- **`toArray()`**
- **`contains(anEntry)`**
- **`findEntry(anEntry)`**
- **`removeEntry(anEntry)`**
- **`size()`**
- **`isEmpty()`**

Specifications for the ADT List

- The effect of ADT list operations on an initially empty list



`myList.add(a)`
a

`myList.add(b)`
a
b

`myList.add(c)`
a
b
c

`myList.add(2,d)`
a
d
b
c

`myList.add(1,e)`
e
a
d
b
c

`myList.remove(3)`
e
a
b
c

© 2019 Pearson Education, Inc.

Specifications for the ADT List

- Data
  - A collection of objects in a specific order and having the same data type
  - The number of objects in the collection
- Note: this is different from the textbook – changes are highlighted in yellow

Pseudocode	UML	Description
add(newEntry)	**+add(newEntry: T): boolean**	Task: Adds newEntry to the end of the list. Input: newEntry is an object. Output: True if the addition was successful
add(newPosition, anEntry)	**+add(newPosition: integer, anEntry: T): boolean**	Task: Adds newEntry at position newPosition within the list. Position 1 indicates the first entry in the list. Input: newPosition is an integer, newEntry is an object. Output: True if the addition was successful
remove(givenPosition)	**+remove(givenPosition: integer): T**	Task: Removes and returns the entry at position givenPosition. Input: givenPosition is an integer. Output: Either returns the removed entry or throws exception if givenPosition is invalid for this list. Note that any value of givenPosition is invalid if the list is empty before the operation.
clear()	**+clear(): void**	Task: Removes all entries from the list. Input: None. Output: None.
replace(givenPosition, anEntry)	**+replace(givenPosition: integer, anEntry: T): T**	Task: Replaces the entry at position givenPosition with newEntry. Input: givenPosition is an integer, newEntry is an object. Output: Either returns the replaced entry or throws an exception if givenPosition is invalid for this list. Note that any value of givenPosition is invalid if the list is empty before the operation.

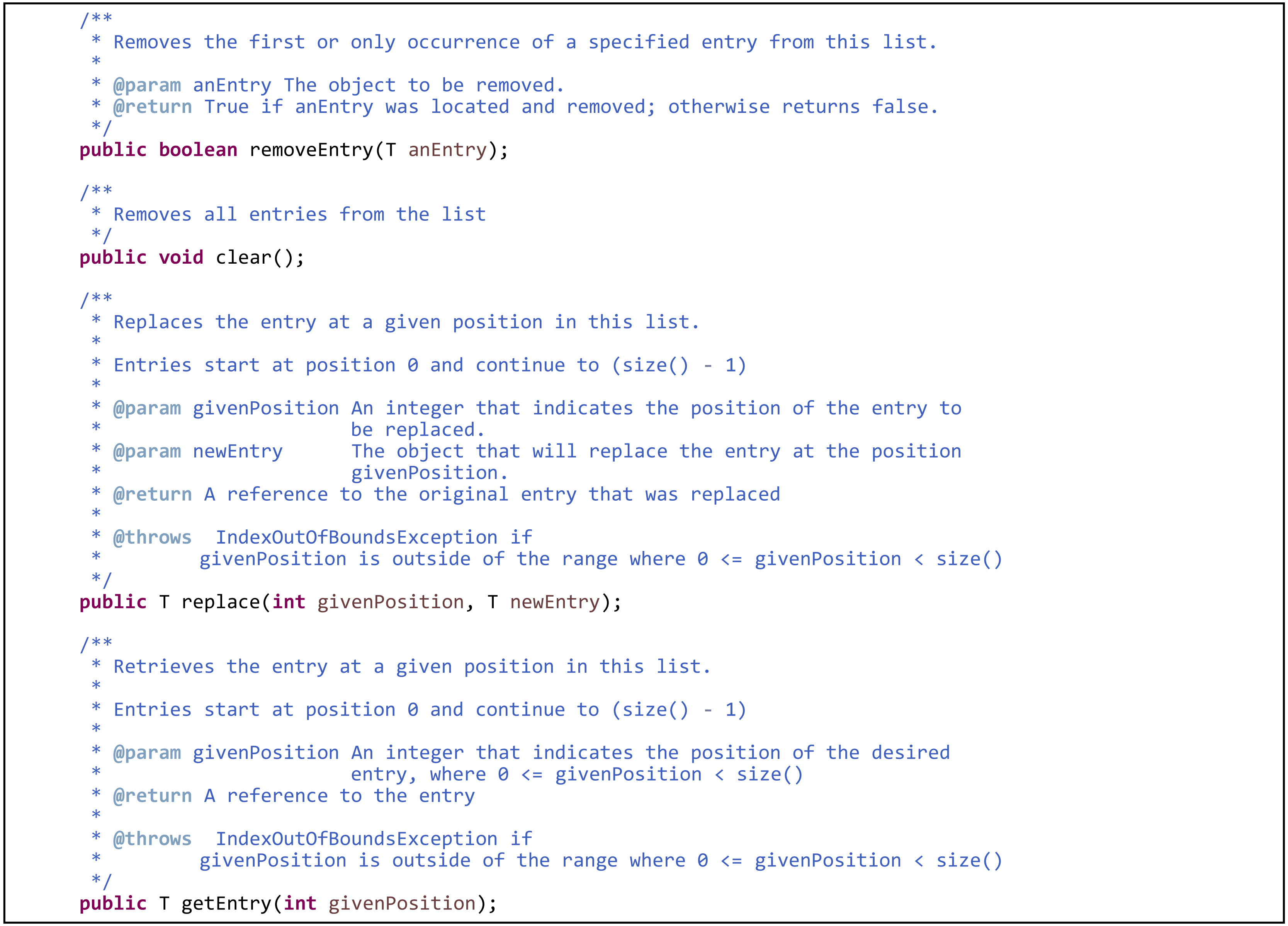
Specifications for the ADT List

Pseudocode	UML	Description
getEntry(givenPosition)	**+getEntry(givenPosition: integer): T**	Task: Retrieves the entry at position givenPosition. Input: givenPosition is an integer. Output:A reference to the original entry that was replaced or throws an exception if the position is invalid.
findEntry(anEntry)	**+findEntry(anEntry T): int**	Task: Find the first occurence of an entry in the list, and return its position. Input: anEntry is an object. Output: The position of the entry, or -1 if not found
removeEntry(anEntry)	**+removeEntry(anEntry T): boolean**	Task: Removes the first or only occurrence of a specified entry from this list. Input: anEntry is an object Output: Returns true if the entry was removed
toArray()	**+toArray: T[ ]**	Task: Retrieves all entries that are in the list in the order in which they occur. Input: None. Output: Returns a new array of the entries currently in the list.
contains(anEntry)	**+contains(anEntry: T): boolean**	Task: Sees whether the list contains anEntry. Input: anEntry is an object. Output: Returns true if anEntry is in the list, or false if not.
size()	**+size(): integer**	Task: Gets the number of entries currently in the list. Input: None. Output: Returns the number of entries currently in the list.
isEmpty()	**+isEmpty(): boolean**	Task: Sees whether the list is empty. Input: None. Output: Returns true if the list is empty, or false if not.

ListInterface

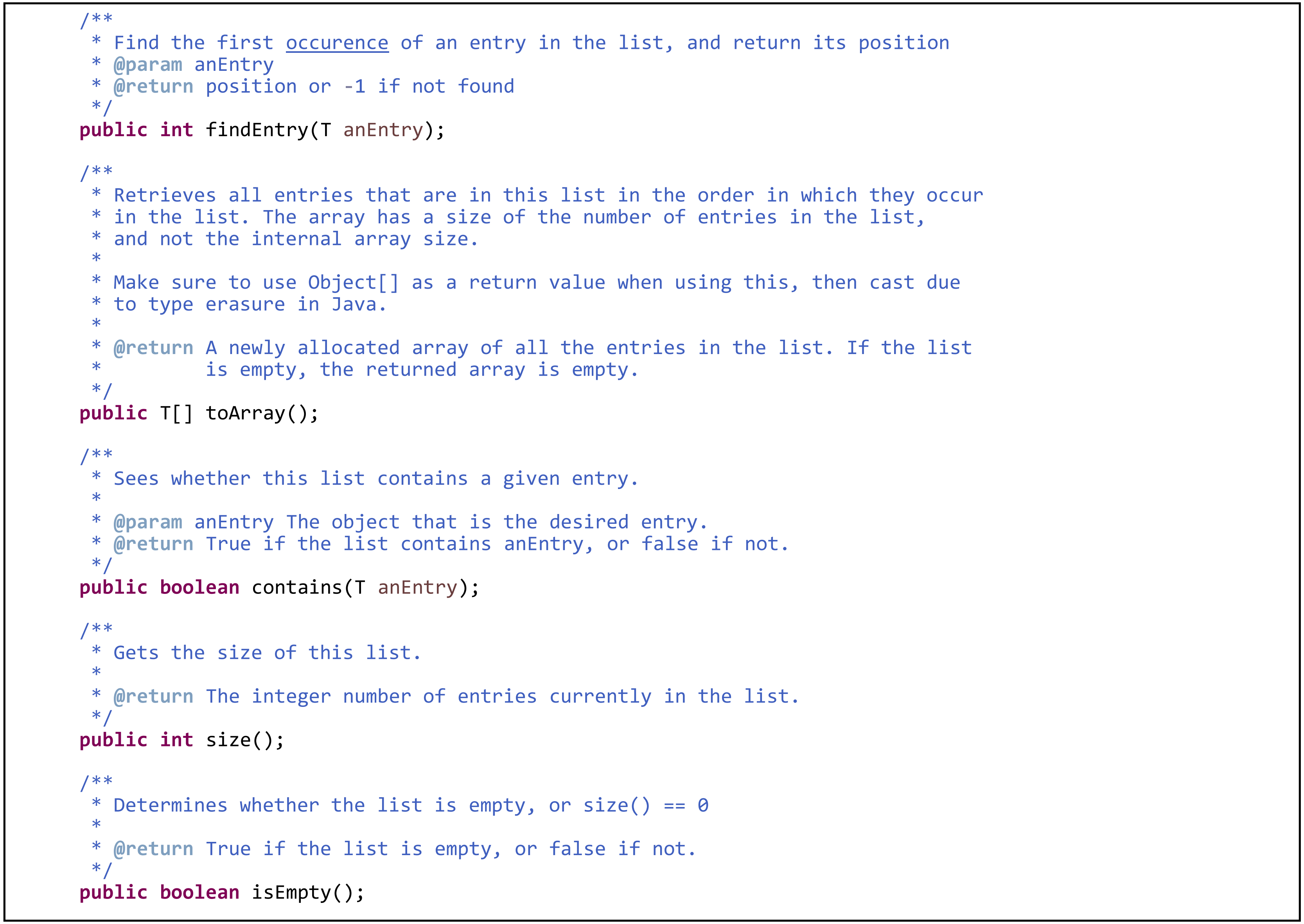
```java
/**
 * Adds a new entry to the end of this list. Entries currently in the list are
 * unaffected. The list's size is increased by 1.
 *
 * @param newEntry The object to be added as a new entry.
 * @return false if not enough room in the list to add the entry
 */
public boolean add(T newEntry);

/**
 * Adds a new entry at a specified position within this list. Entries originally
 * at and above the specified position are at the next higher position within
 * the list. The list's size is increased by 1.
 *
 * If the position is equal to the size of the list, simply add the entry to the list.
 *
 * Entries start at position 0 and continue to (size() - 1)
 *
 * @param newPosition An integer that specifies the desired position of the new
 *                    entry.
 * @param newEntry    The object to be added as a new entry.
 * @return false if not enough room in the list to add the entry
 */
public boolean add(int newPosition, T newEntry);

/**
 * Removes the entry at a given position from this list. Entries originally at
 * positions higher than the given position are at the next lower position
 * within the list, and the list's size is decreased by 1.
 *
 * Entries start at position 0 and continue to (size() - 1)
 *
 * @param givenPosition An integer that indicates the position of the entry to
 *                      be removed.
 * @return A reference to the removed entry
 *
 * @throws  IndexOutOfBoundsException if
 *             givenPosition is outside of the range where 0 <= givenPosition < size()
 */
public T remove(int givenPosition);
```

ListInterface



```java
    /**
     * Removes the first or only occurrence of a specified entry from this list.
     *
     * @param anEntry The object to be removed.
     * @return True if anEntry was located and removed; otherwise returns false.
     */
    public boolean removeEntry(T anEntry);

    /**
     * Removes all entries from the list
     */
    public void clear();

    /**
     * Replaces the entry at a given position in this list.
     *
     * Entries start at position 0 and continue to (size() - 1)
     *
     * @param givenPosition An integer that indicates the position of the entry to
     *                      be replaced.
     * @param newEntry      The object that will replace the entry at the position
     *                      givenPosition.
     * @return A reference to the original entry that was replaced
     *
     * @throws  IndexOutOfBoundsException if
     *        givenPosition is outside of the range where 0 <= givenPosition < size()
     */
    public T replace(int givenPosition, T newEntry);

    /**
     * Retrieves the entry at a given position in this list.
     *
     * Entries start at position 0 and continue to (size() - 1)
     *
     * @param givenPosition An integer that indicates the position of the desired
     *                      entry, where 0 <= givenPosition < size()
     * @return A reference to the entry
     *
     * @throws  IndexOutOfBoundsException if
     *        givenPosition is outside of the range where 0 <= givenPosition < size()
     */
    public T getEntry(int givenPosition);
```

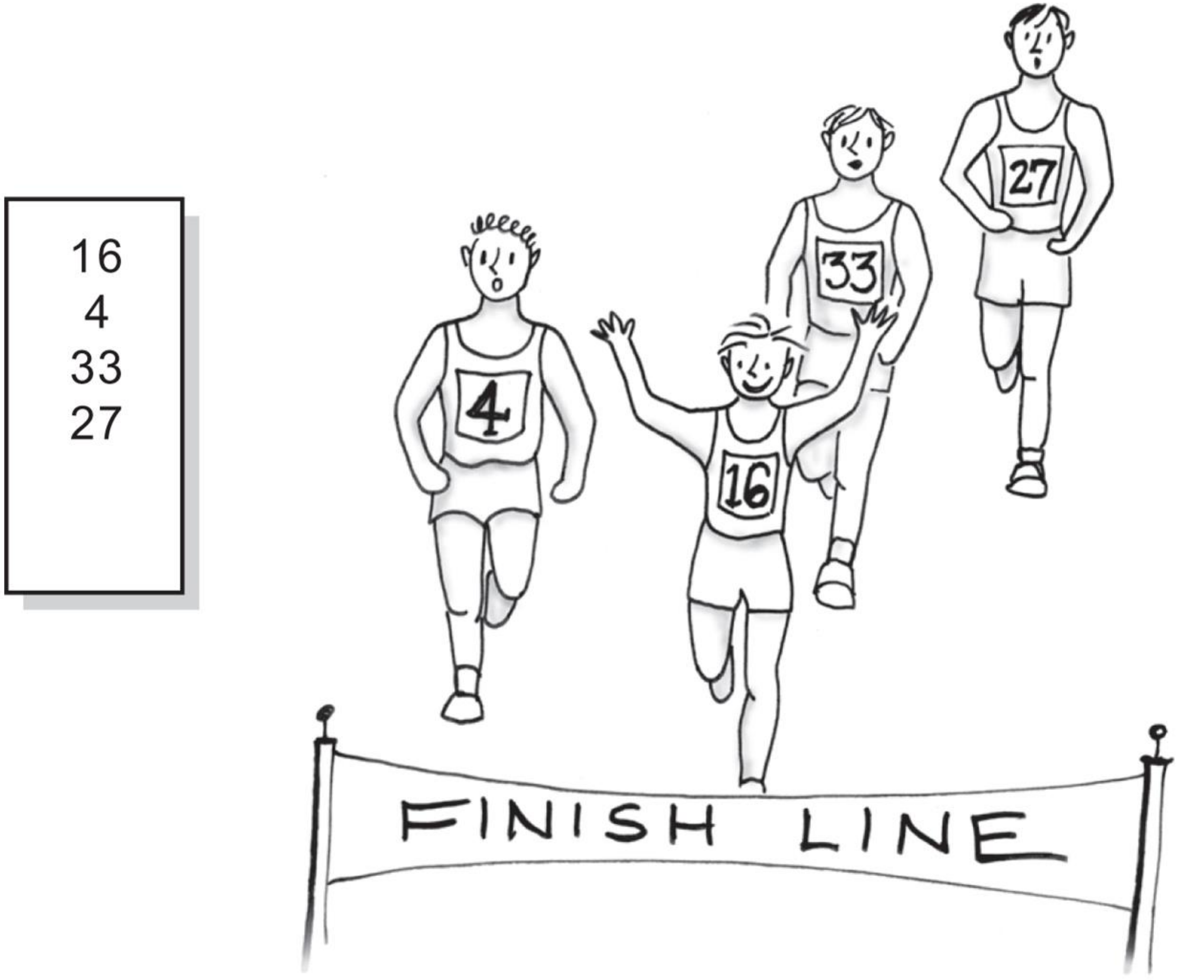
ListInterface



```
    /**
     * Find the first occurence of an entry in the list, and return its position
     * @param anEntry
     * @return position or -1 if not found
     */
    public int findEntry(T anEntry);

    /**
     * Retrieves all entries that are in this list in the order in which they occur
     * in the list. The array has a size of the number of entries in the list,
     * and not the internal array size.
     *
     * Make sure to use Object[] as a return value when using this, then cast due
     * to type erasure in Java.
     *
     * @return A newly allocated array of all the entries in the list. If the list
     *         is empty, the returned array is empty.
     */
    public T[] toArray();

    /**
     * Sees whether this list contains a given entry.
     *
     * @param anEntry The object that is the desired entry.
     * @return True if the list contains anEntry, or false if not.
     */
    public boolean contains(T anEntry);

    /**
     * Gets the size of this list.
     *
     * @return The integer number of entries currently in the list.
     */
    public int size();

    /**
     * Determines whether the list is empty, or size() == 0
     *
     * @return True if the list is empty, or false if not.
     */
    public boolean isEmpty();
```

Using the ADT List

- A list of numbers that identify runners in the order in which they finished



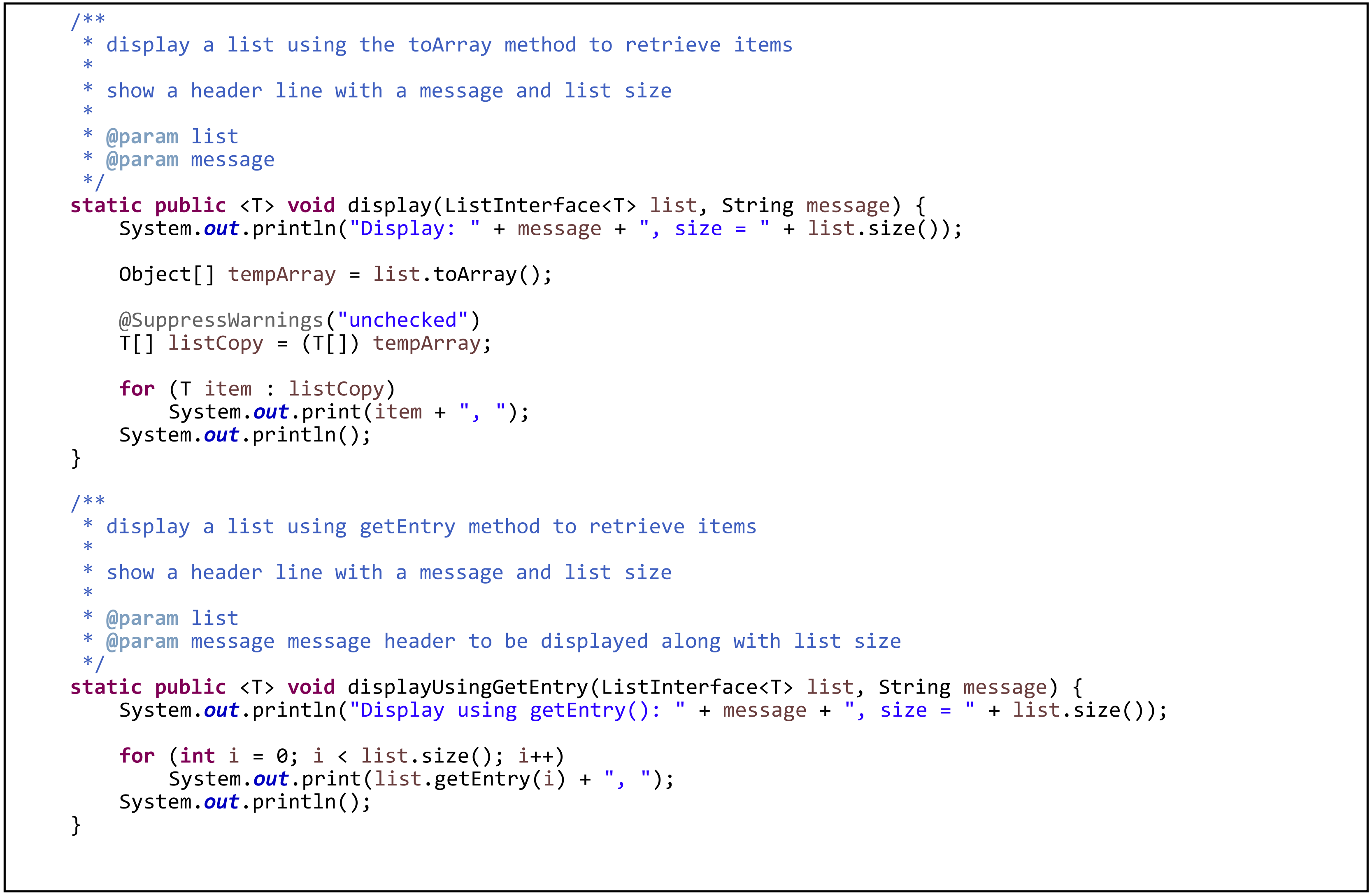
© 2019 Pearson Education, Inc.

Using ListInterface

- Add strings to the list, using any implementation
- Display using DemoUtilities class.

```
public class RoadRaceDemo {
    public static void main(String[] args) {

//       ListInterface<String> runnerList = new AList<>();
//       ListInterface<String> runnerList = new LList<>();
//       ListInterface<String> runnerList = new LListWithTail<>();
//       ListInterface<String> runnerList = new CompletedAList<>();
         ListInterface<String> runnerList = new CompletedLList<>();
//       ListInterface<String> runnerList = new CompletedLListWithTail<>();

         // runnerList has only methods in ListInterface

         runnerList.add("16"); // Winner
         runnerList.add(" 4"); // Second place
         runnerList.add("33"); // Third place
         runnerList.add("27"); // Fourth place

         DemoUtilities.display(runnerList, "List of Runners");
         DemoUtilities.displayUsingGetEntry(runnerList, "List of Runners");
    }
}
```

displayUsingGetEntry() with ListInterface



```java
/**
 * display a list using the toArray method to retrieve items
 *
 * show a header line with a message and list size
 *
 * @param list
 * @param message
 */
static public <T> void display(ListInterface<T> list, String message) {
    System.out.println("Display: " + message + ", size = " + list.size());

    Object[] tempArray = list.toArray();

    @SuppressWarnings("unchecked")
    T[] listCopy = (T[]) tempArray;

    for (T item : listCopy)
        System.out.print(item + ", ");
    System.out.println();
}

/**
 * display a list using getEntry method to retrieve items
 *
 * show a header line with a message and list size
 *
 * @param list
 * @param message message header to be displayed along with list size
 */
static public <T> void displayUsingGetEntry(ListInterface<T> list, String message) {
    System.out.println("Display using getEntry(): " + message + ", size = " + list.size());

    for (int i = 0; i < list.size(); i++)
        System.out.print(list.getEntry(i) + ", ");
    System.out.println();
}
```

List of Students

- Note that position in a list starts at 0
  - This is different from the textbook
- add **before** the indicated position
- To add to the end, use add(size(), entry), which is one past the last position

```java
public class StudentsDemo {
    public static void main(String[] args) {
//      ListInterface<String> alphaList = new AList<>();
//      ListInterface<String> alphaList = new LList<>();
//      ListInterface<String> alphaList = new LListWithTail<>();
        ListInterface<String> alphaList = new CompletedAList<>();
//      ListInterface<String> alphaList = new CompletedLList<>();
//      ListInterface<String> alphaList = new CompletedLListWithTail<>();


        alphaList.add(0, "Amy"); // Amy
        alphaList.add(1, "Elias"); // Amy Elias
        alphaList.add(1, "Bob"); // Amy Bob Elias
        alphaList.add(2, "Drew"); // Amy Bob Drew Elias
        alphaList.add(0, "Aaron"); // Aaron Amy Bob Drew Elias
        alphaList.add(3, "Carol"); // Aaron Amy Bob Carol Drew Elias

        DemoUtilities.display(alphaList, "List of Students");
        DemoUtilities.displayUsingGetEntry(alphaList, "List of Students");
    }
}
```

Using the ADT List

- A list of Name objects, rather than String

```
// Make a list of names as you think of them
ListInterface<Name> nameList = new AList<>();
Name amy = new Name("Amy", "Smith");
nameList.add(amy);
nameList.add(new Name("Tina", "Drexel");
nameList.add(new Name("Robert", "Jones");

Name secondName = nameList.getEntry(2);
```

Java Class Library: The Interface `List`

- Method headers from the interface List

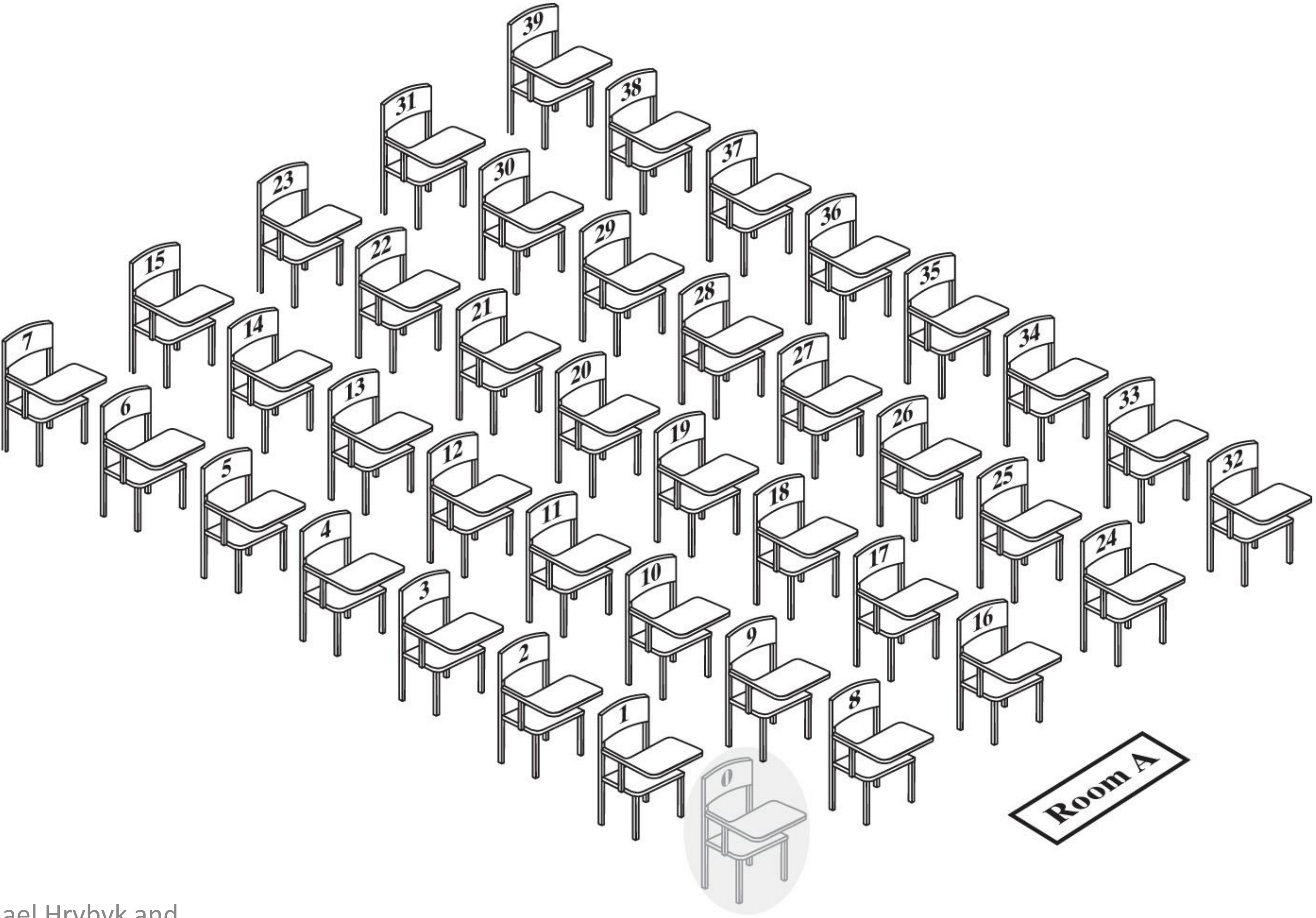
```
public void add(int index, T newEntry)
public T remove(int index)
public void clear()
public T set(int index, T anEntry) // Like replace
public  T get(int index) // Like getEntry
public boolean contains(Object anEntry)
public int size() // Like getLength
public boolean isEmpty()
```

Java Class Library: The Class `ArrayList`

- Available constructors
  - **`public ArrayList()`**
  - **`public ArrayList(int initialCapacity)`**
- Similar to **`java.util.vector`**
  - Can use either **`ArrayList`** or **`Vector`** as an implementation of the interface **`List`**.

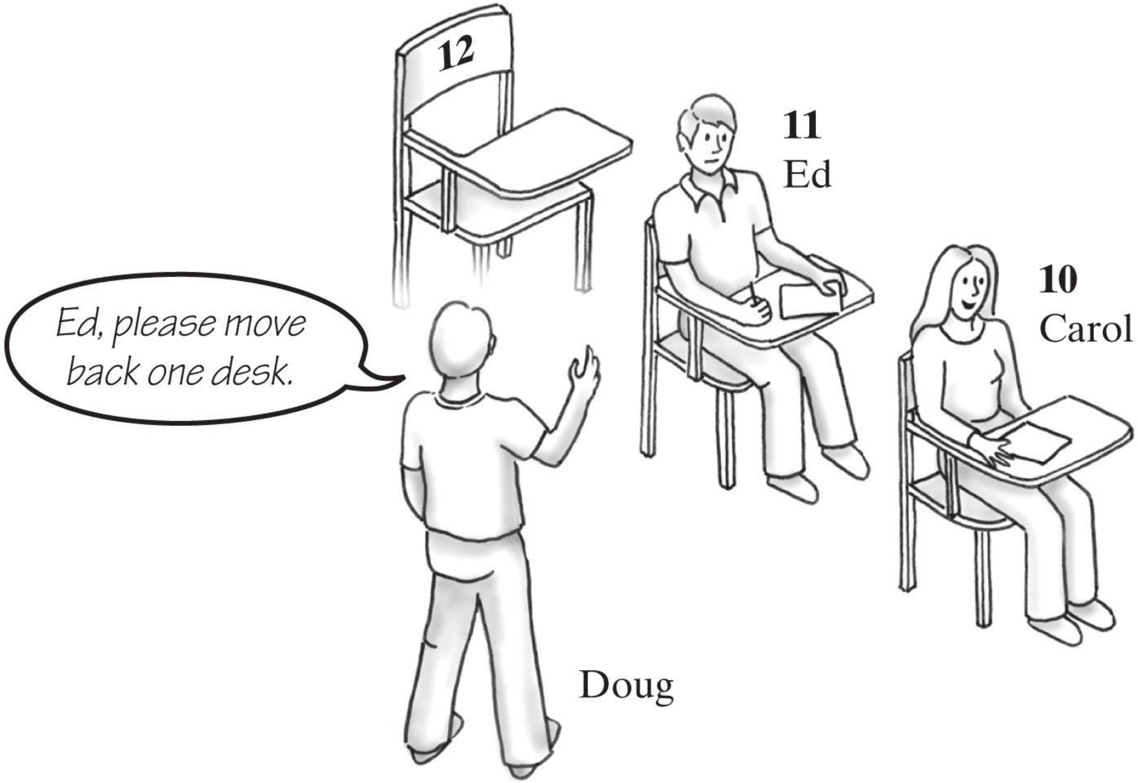
Using an Array to Implement the ADT List

- A classroom that contains desks in fixed positions

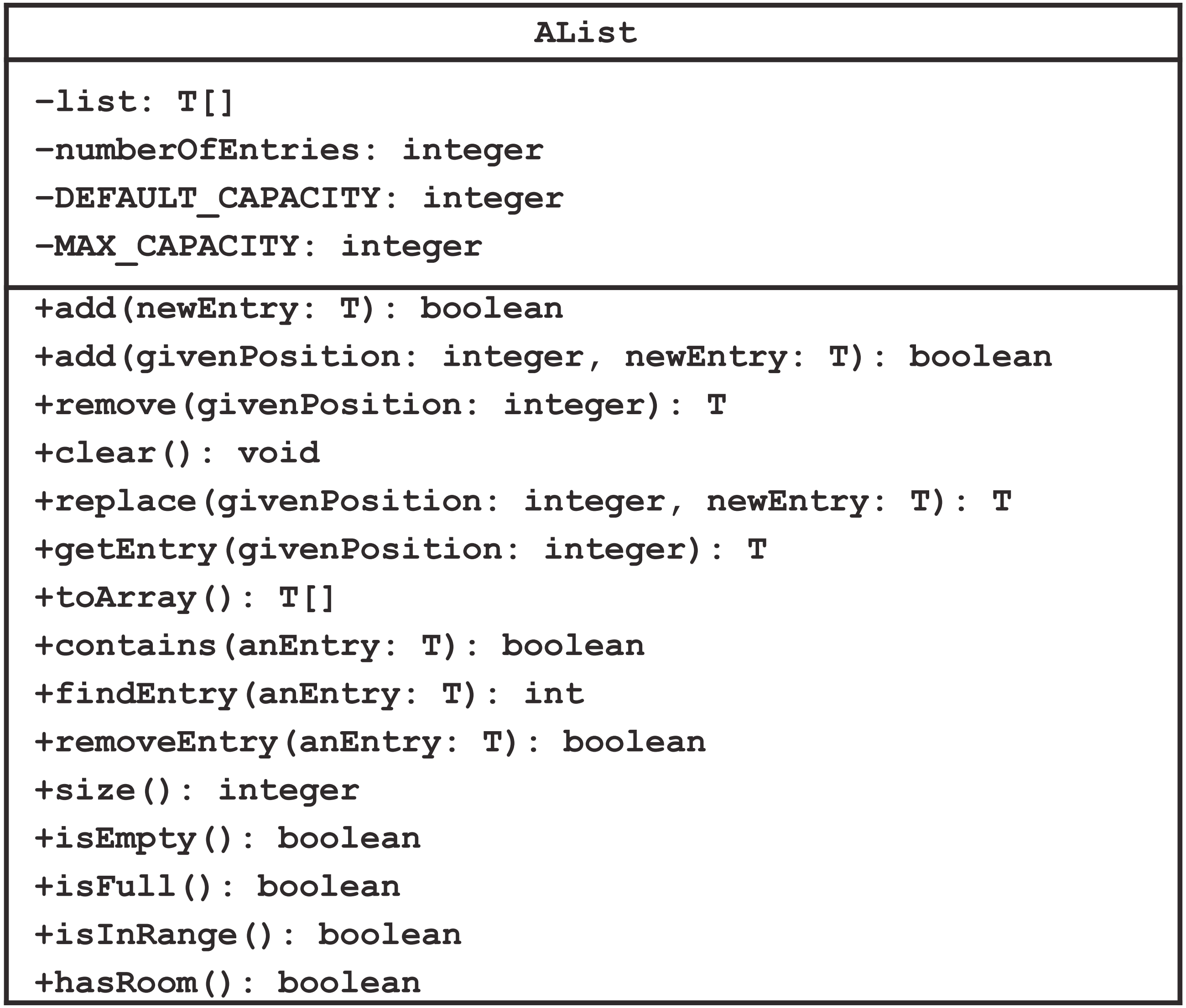


Using an Array to Implement the ADT List

- Seating a new student between two existing students: At least one other student must move

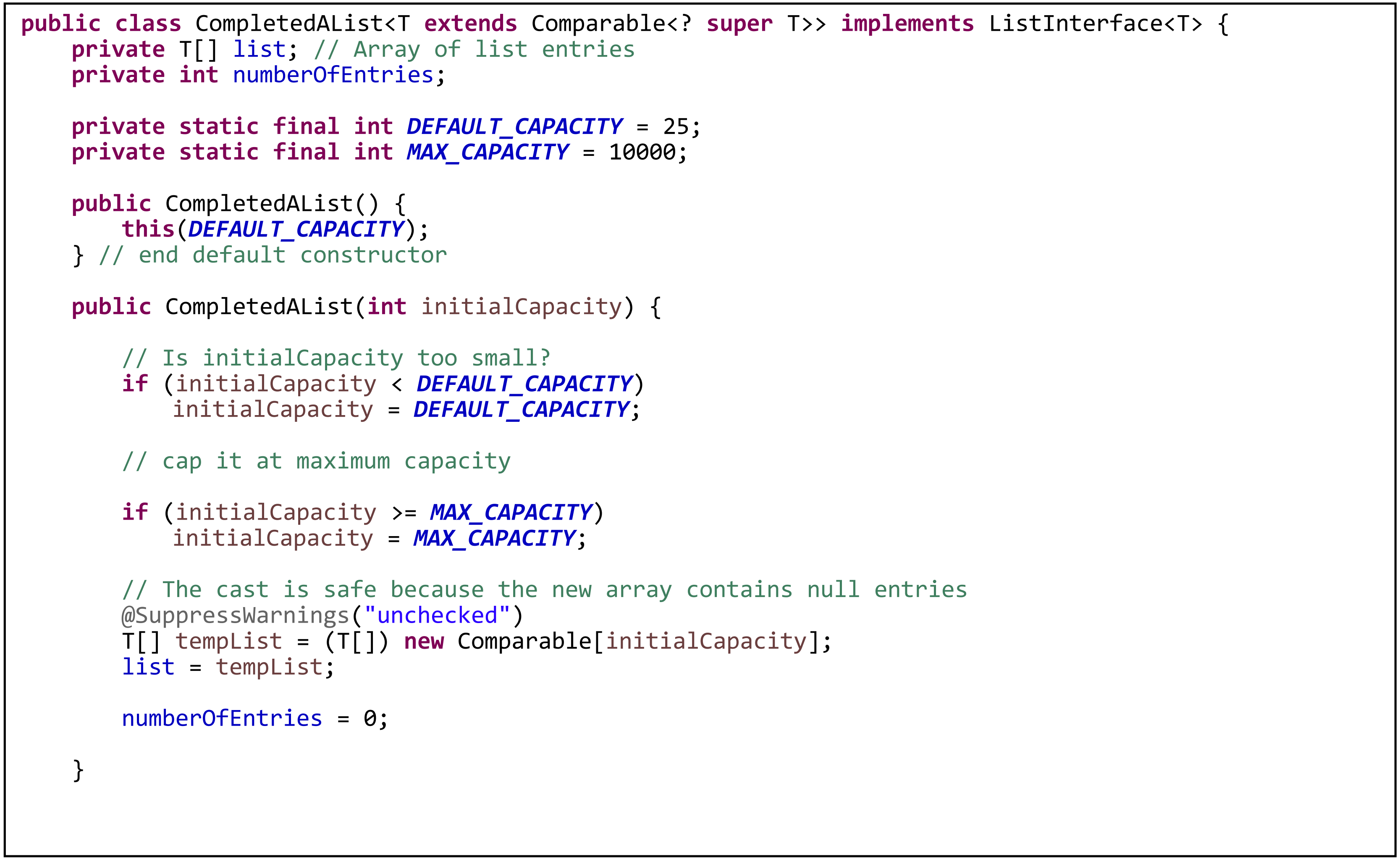


Using an Array to Implement the ADT List



```
AList
-list: T[]
-numberOfEntries: integer
-DEFAULT_CAPACITY: integer
-MAX_CAPACITY: integer
+add(newEntry: T): boolean
+add(givenPosition: integer, newEntry: T): boolean
+remove(givenPosition: integer): T
+clear(): void
+replace(givenPosition: integer, newEntry: T): T
+getEntry(givenPosition: integer): T
+toArray(): T[]
+contains(anEntry: T): boolean
+findEntry(anEntry: T): int
+removeEntry(anEntry: T): boolean
+size(): integer
+isEmpty(): boolean
+isFull(): boolean
+isInRange(): boolean
+hasRoom(): boolean
```

AList - constructors



```java
public class CompletedAList<T extends Comparable<? super T>> implements ListInterface<T> {
    private T[] list; // Array of list entries
    private int numberOfEntries;

    private static final int DEFAULT_CAPACITY = 25;
    private static final int MAX_CAPACITY = 10000;

    public CompletedAList() {
        this(DEFAULT_CAPACITY);
    } // end default constructor

    public CompletedAList(int initialCapacity) {

        // Is initialCapacity too small?
        if (initialCapacity < DEFAULT_CAPACITY)
            initialCapacity = DEFAULT_CAPACITY;

        // cap it at maximum capacity

        if (initialCapacity >= MAX_CAPACITY)
            initialCapacity = MAX_CAPACITY;

        // The cast is safe because the new array contains null entries
        @SuppressWarnings("unchecked")
        T[] tempList = (T[]) new Comparable[initialCapacity];
        list = tempList;

        numberOfEntries = 0;

    }
```

AList – basic methods

```
	public int size() {
		return numberOfEntries;
	}

	/**
	 * Determines if a position is in the proper range of the list
	 *
	 * @param givenPosition
	 * @return
	 */
	public boolean isInRange(int givenPosition) {
		return (givenPosition >= 0) && (givenPosition < numberOfEntries);
	}

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

AList – basic methods

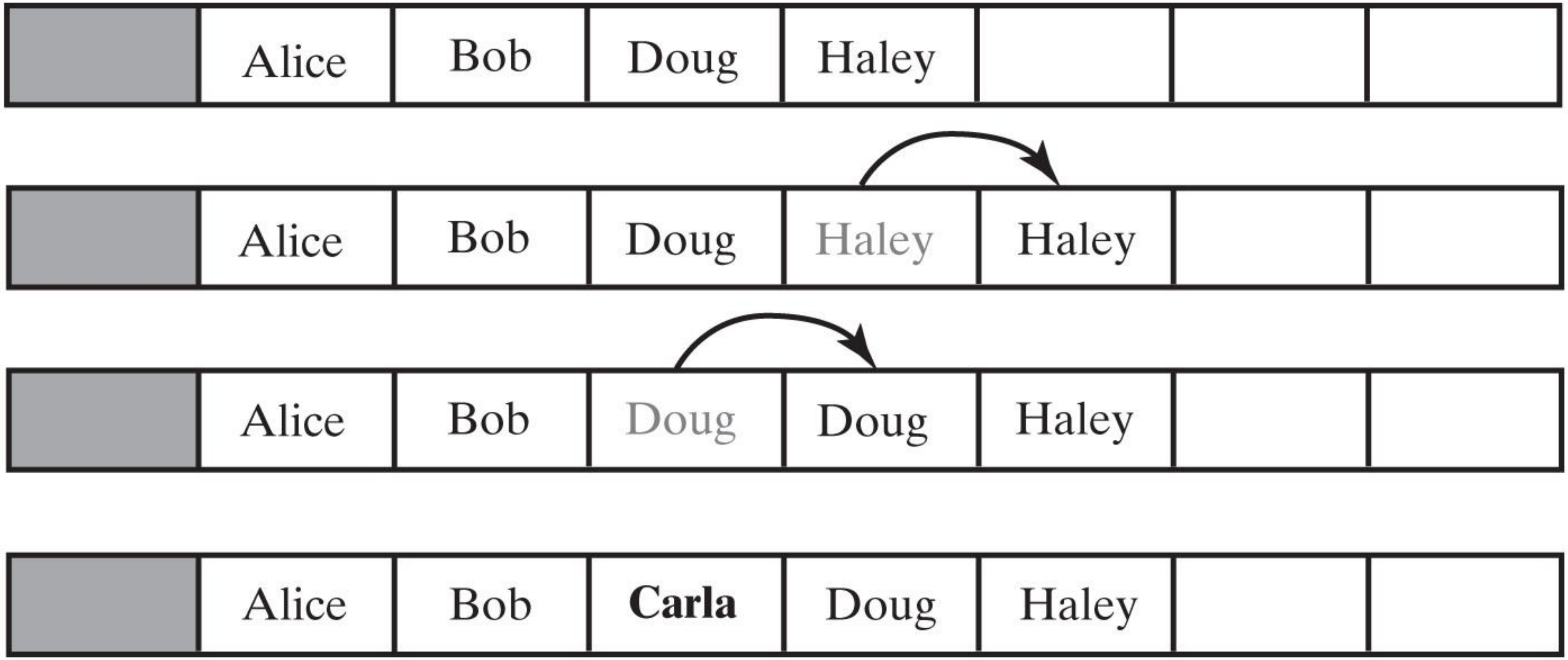
```java
/**
 * Determine if there is no more room in the list to add entries
 *
 * @return true if the list is full
 */
public boolean isFull() {
    if (numberOfEntries >= list.length)
        return false;
    else
        return true;
}

/**
 * Determines if there is room in the list to add entries
 *
 * @return true if there is room in the list
 */
public boolean hasRoom() {
    if (numberOfEntries < list.length)
        return true;
    else
        return false;
}

public T[] toArray() {
    return (T[]) Arrays.copyOf(list, numberOfEntries);
}
```

add() – simply adds to end of list

```
    public boolean add(T newEntry) {

        if (hasRoom() == false)
            return false;

        // set entry into the next slot of the array
        list[numberOfEntries] = newEntry;
        numberOfEntries++;

        return true;

    }
```

Adding an entry to the middle of an array

- Making room to insert Carla as the third entry in an array
- Shift up



	Alice	Bob	Doug	Haley			
	Alice	Bob	Doug	Haley	Haley		
	Alice	Bob	Doug	Doug	Haley		
	Alice	Bob	**Carla**	Doug	Haley		

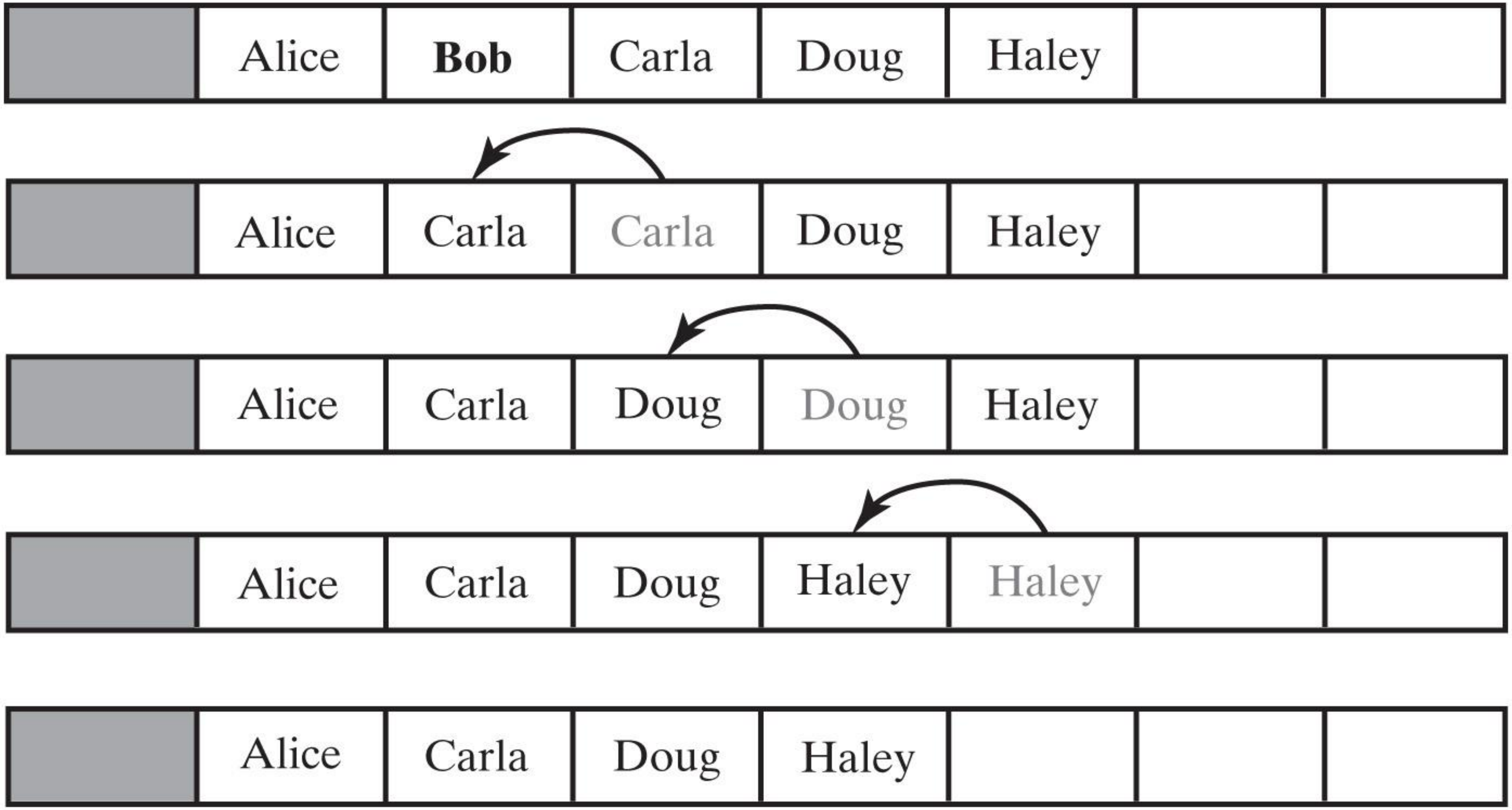
add() at a specific position

- Start at the end, and shift to the next position.
- Next position from the current end will be empty.
- Decrement to move down

```
    public boolean add(int newPosition, T newEntry) {

        // make sure we have room

        if(hasRoom() == false)
            return false;

        // if list is empty, ignore the position and add
        if (isEmpty()) {
            list[0] = newEntry;
            numberOfEntries = 1;
            return true;
        }

        // needs to be in range or in the last position (not greater)

        if (newPosition < 0 || newPosition > numberOfEntries)
            return false;

        // shift all entries starting at the desired position for the new entry
        // up by one to make room
        // notice we are starting from the back and shifting each to a new position

        for (int index = numberOfEntries - 1; index >= newPosition; index--) {
            list[index + 1] = list[index];
        }

        // now set new entry into the vacated position and increast the list count
        list[newPosition] = newEntry;

        numberOfEntries++;

        return true;

    }
```

->make space on particular position,

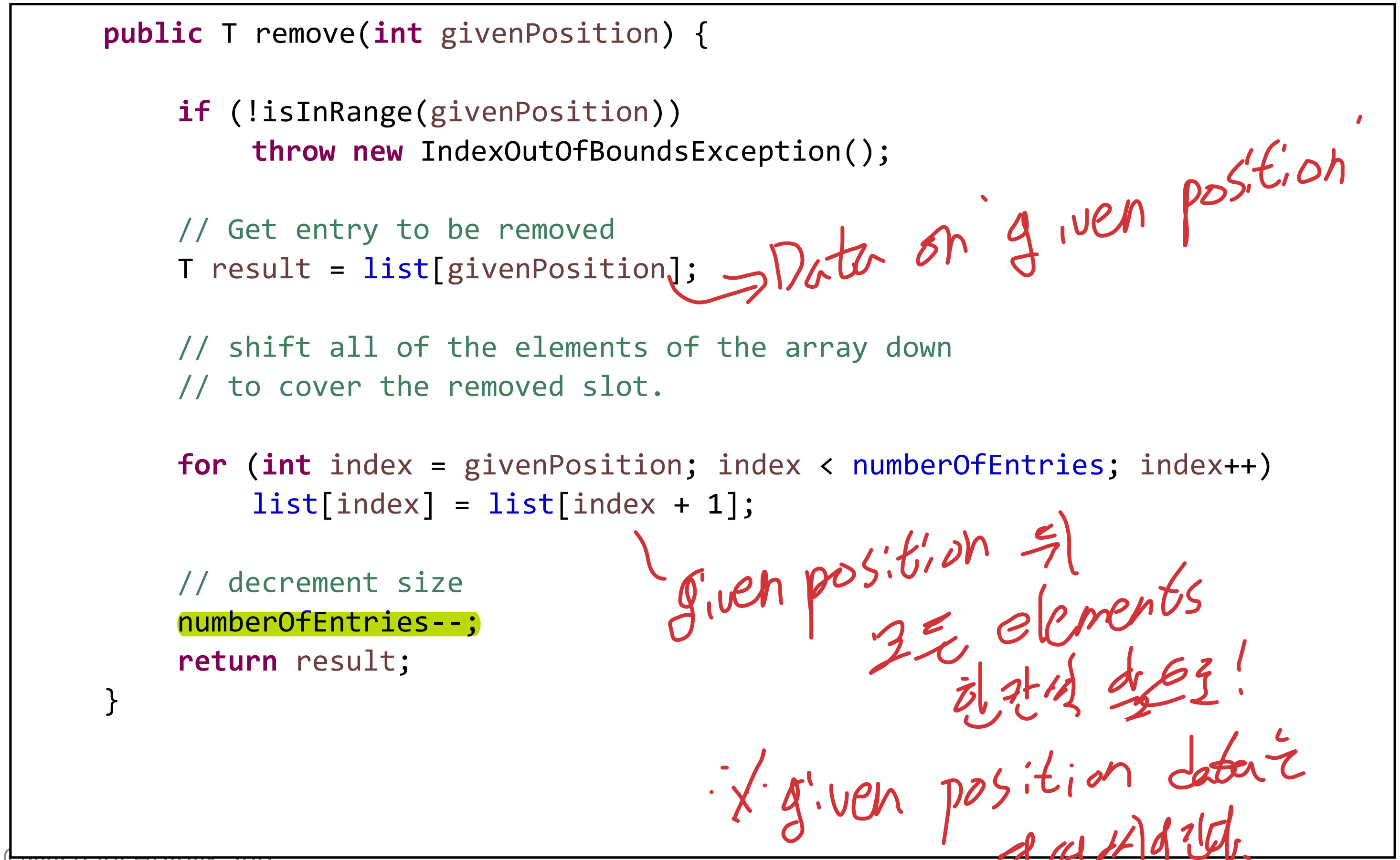
Removing Bob by shifting array entries



	Alice	Bob	Carla	Doug	Haley		
	Alice	Carla	Carla	Doug	Haley		
	Alice	Carla	Doug	Doug	Haley		
	Alice	Carla	Doug	Haley	Haley		
	Alice	Carla	Doug	Haley			

remove() at a specific position

- Start at the position, and shift from the next position.
- Position will be overwritten
- Increment to move up



```java
    public T remove(int givenPosition) {

        if (!isInRange(givenPosition))
            throw new IndexOutOfBoundsException();

        // Get entry to be removed
        T result = list[givenPosition];

        // shift all of the elements of the array down
        // to cover the removed slot.

        for (int index = givenPosition; index < numberOfEntries; index++)
            list[index] = list[index + 1];

        // decrement size
        numberOfEntries--;
        return result;
    }
```

→ Data on given position'

given position 뒤 모든 elements 한칸씩 앞으로!

∴ given position data는 덮어씌어진다.

removeEntry() and findEntry()

- To remove an entry, find its position first.
- Once found, call remove()
- To find an entry, iterate through the list

```
    public boolean removeEntry(T anEntry) {

        // find the entry, then remove it if it exists
        int position = findEntry(anEntry);

        if (position < 0)
            return false;
        else {
            if (remove(position) != null)
                return true;
            else
                return false;
        }
    }
    public int findEntry(T anEntry) {

        int position = 0;

        // if the entry exists, loop will end when found.

        while ((position < numberOfEntries) && !anEntry.equals(getEntry(position))) {
            position++;
        }

        // return -1 if not found as the entire list has been traversed
        return position == numberOfEntries ? -1 : position;

    }
```

getEntry() and contains()

- getEntry() simply returns the array entry for the position
- contains() simply calls findEntry()

```
public T getEntry(int givenPosition) {

      // if a legal position, return the data in the array slot

      if (!isInRange(givenPosition))
            throw new IndexOutOfBoundsException();

      return list[givenPosition];
}

public boolean contains(T anEntry) {
      return findEntry(anEntry) >= 0;
}
```

clear() – remove all entries

- Iterate through array and set entries to null
- Alternatives
  - ○ Successive calls to remove() - inefficient
  - ○ Create new array

```
public void clear() {

	// Clear entries but retain array; no need to create a new array
	for (int index = 0; index < numberOfEntries; index++)
		list[index] = null;

	numberOfEntries = 0;
}
```

Advantages of Linked Implementation

- Uses memory only as needed
- When entry removed, unneeded memory returned to system
- Avoids moving data when adding or removing entries

Linked list uses nodes

goto check

'Doubly liked node'

- Use of a doubly linked node
- Has previous and next node references
- Node class constructors below
  - Can set next, previous, and data

```
public class Node<T extends Comparable<? super T>> {
	private T data; // object with data to be held in the node
	private Node<T> next; // Link to next node
	private Node<T> previous; // Link to previous node

	public Node(T dataPortion) {
		data = dataPortion;
		next = null;
		previous = null;
	}

	public Node(T dataPortion, Node<T> nextNode) {
		data = dataPortion;
		next = nextNode;
		previous = null;
	}

	public Node(T dataPortion, Node<T> nextNode, Node<T> previousNode) {
		data = dataPortion;
		next = nextNode;
		previous = previousNode;
	}
```

Node set/get methods

```
	public T getData() {
		return data;
	}

	public void setData(T newData) {
		data = newData;
	}

	public Node<T> getNextNode() {
		return next;
	}

	public void setNextNode(Node<T> nextNode) {
		next = nextNode;
	}

	public Node<T> getPreviousNode() {
		return previous;
	}

	public void setPreviousNode(Node<T> previousNode) {
		previous = previousNode;
	}
```

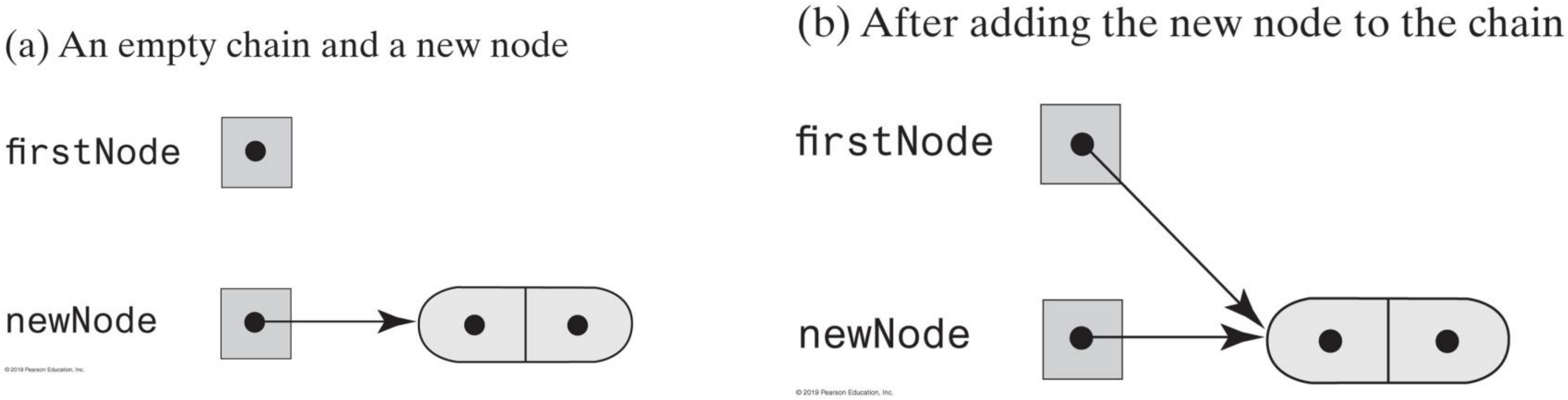
Adding a Node at Various Positions

- Possible cases:
  - Chain is empty
  - Adding node at chain's beginning
  - Adding node between adjacent nodes
  - Adding node to chain's end

Adding a Node to an empty chain

- This pseudocode establishes a new node for the given data

```
newNode references a new instance of Node
Place newEntry in newNode
firstNode = address of newNode
```



(a) An empty chain and a new node

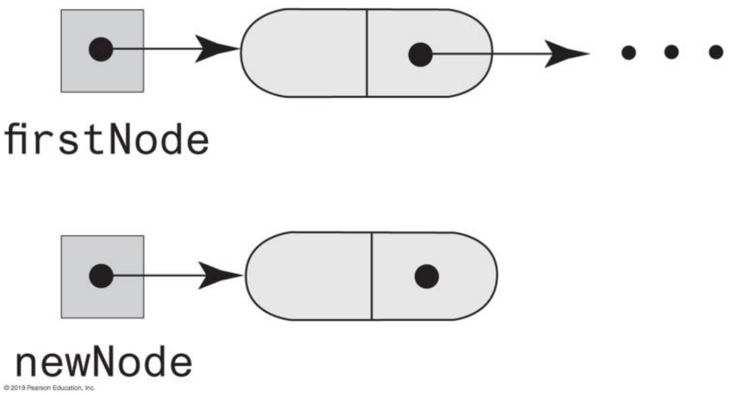
(b) After adding the new node to the chain

firstNode
newNode
© 2019 Pearson Education, Inc.
firstNode
newNode
© 2019 Pearson Education, Inc.

Adding a Node to the beginning of a chain

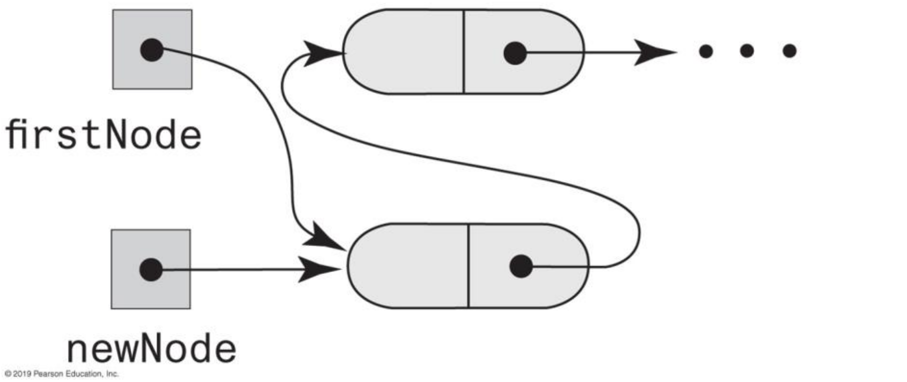
- This pseudocode describes the steps needed to add a node to the beginning of a chain.

```
newNode references a new instance of Node
Place newEntry in newNode
Set newNode's link to firstNode
Set firstNode to newNode
```



(a) A chain of nodes and a new node

© 2019 Pearson Education, Inc.



(b) After adding the new node to the beginning of the chain

© 2019 Pearson Education, Inc.

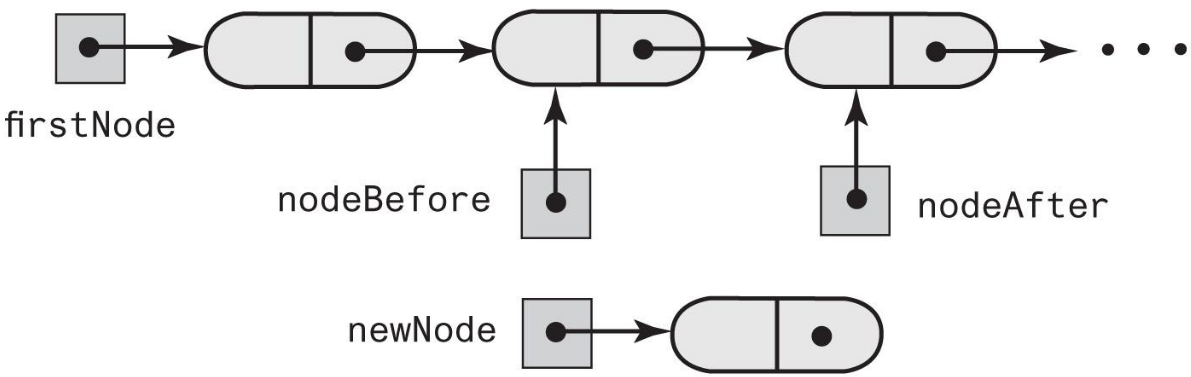
Adding a Node

- Pseudocode to add a node to a chain between two existing, consecutive nodes

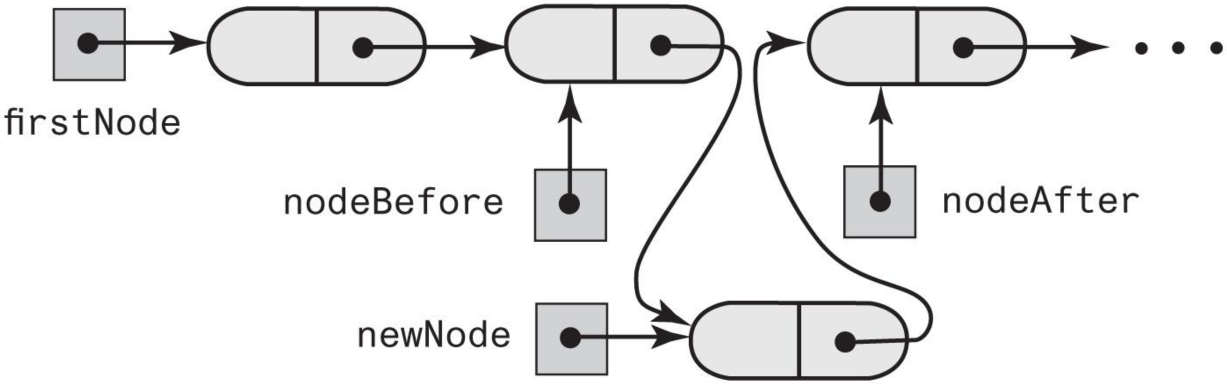
```
newNode references the new node
Place newEntry in newNode
Let nodeBefore reference the node that will be before the new node
Set nodeAfter to nodeBefore's link
Set newNode's link to nodeAfter
Set nodeBefore's link to newNode
```

Adding a Node between nodes

(a) A chain of nodes and a new node



(b) After adding the new node between adjacent nodes



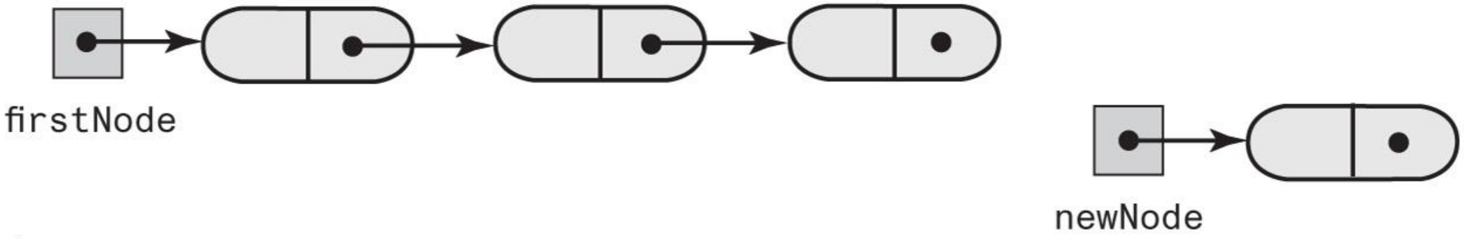
Adding a Node

- Steps to add a node at the end of a chain.

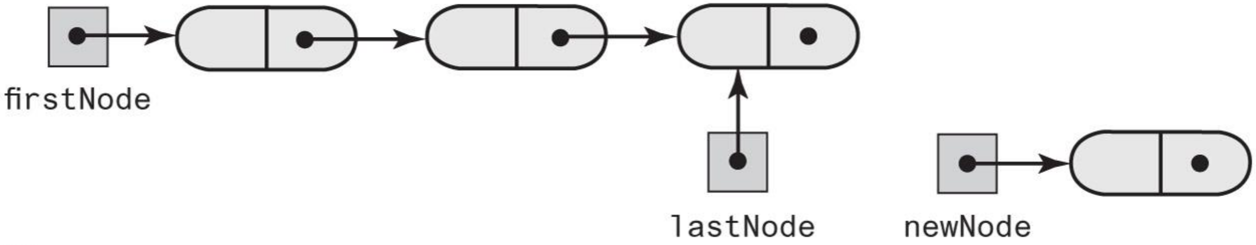
```
newNode references a new instance of Node
Place newEntry in newNode
Locate the last node in the chain
Place the address of newNode in this last node
```

Adding a Node to the end of a chain

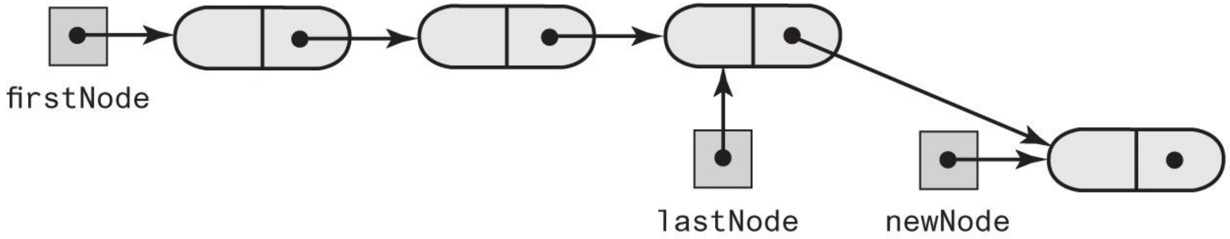
(a) A chain of nodes and a new node



(b) After locating the last node



(c) After adding the new node to the end of the chain



Removing a Node

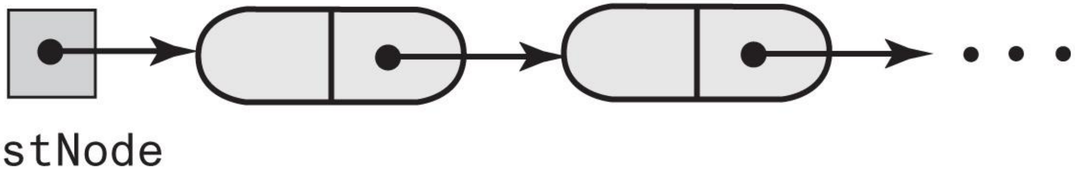
- Possible cases
    - Removing the first node
    - Removing a node other than first one

Removing the first node

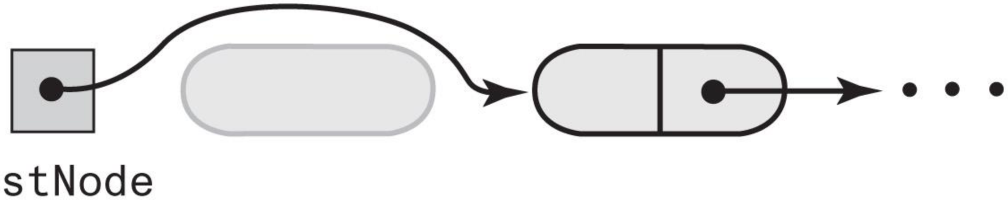
Set firstNode *to the link in the first node;* firstNode *now either references the second node or is* null *if the chain had only one node.*

Since all references to the first node no longer exist, the system automatically recycles the first node's memory.

(a) A chain of nodes



(b) After removing the first node



Removing a Node other than first

Let nodeBefore *reference the node before the one to be removed.*

Set nodeToRemove *to* nodeBefore*'s link;* nodeToRemove *now references the node to be removed.*

Set nodeAfter *to* nodeToRemove*'s link;* nodeAfter *now either references the node after the one to be removed or is* null.

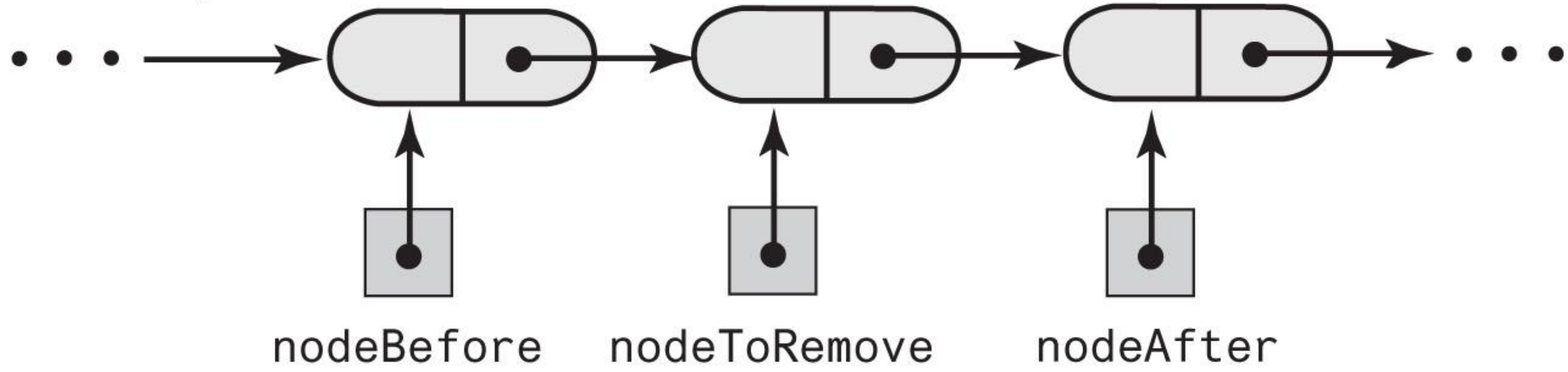
Set nodeBefore*'s link to* nodeAfter. *(*nodeToRemove *is now disconnected from the chain.)*

Set nodeToRemove *to* null.

Since all references to the disconnected node no longer exist, the system automatically recycles the node's memory.

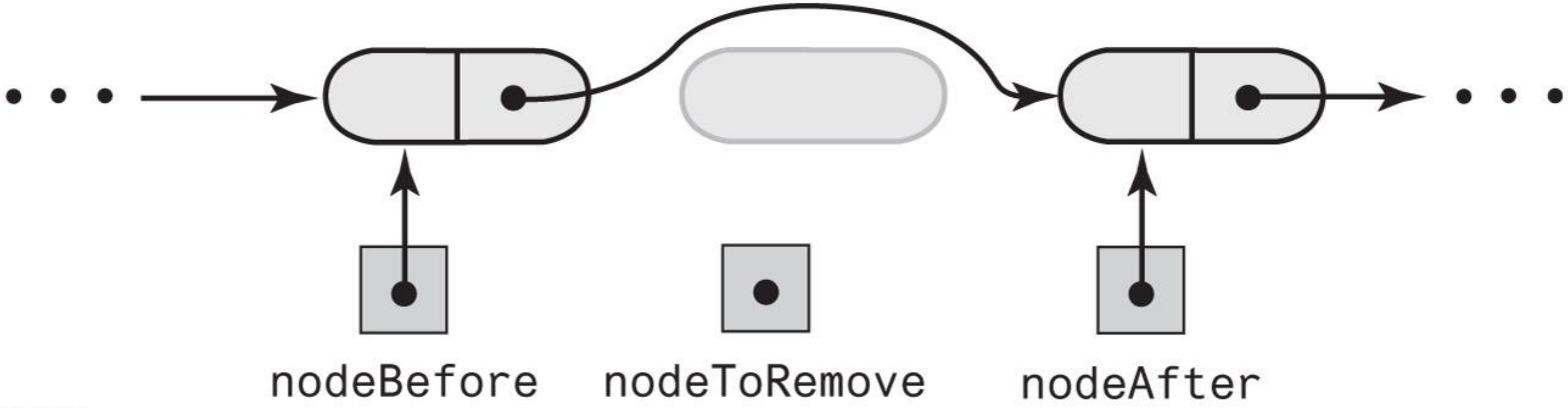
Removing an interior Node

(a) After locating the node to remove



© 2019 Pearson Education, Inc.

(b) After removing the node



© 2019 Pearson Education, Inc.

LList – linked list implementation

- Constructor – no need for capacity allocation

```java
public class CompletedLList<T extends Comparable<? super T>> implements ListInterface<T> {
       private Node<T> firstNode; // Reference to first node of chain
       private int numberOfEntries;

       public CompletedLList() {
              initializeDataFields();
       }

       /**
        * Reset all data in the list.
        */
       private void initializeDataFields() {
              firstNode = null;
              numberOfEntries = 0;
       }
```

getNodeAt()

- add/remove methods require this

```
    private Node<T> getNodeAt(int givenPosition) {

        if (isEmpty())
            return null;

        if (!isInRange(givenPosition))
            throw new IndexOutOfBoundsException();

        // Traverse the chain to locate the desired node

        Node<T> currentNode = firstNode;

        for (int counter = 0; counter < givenPosition; counter++)
            currentNode = currentNode.getNextNode();

        return currentNode;
    }
```

add() to the end of the list

- First need to find the end of this – requires traversal
  - use getNodeAt()

```
public boolean add(T newEntry) {
    Node<T> newNode = new Node<>(newEntry);

    if (isEmpty())
        firstNode = newNode;
    else {
        // Add to end of nonempty list
        Node<T> lastNode = getNodeAt(numberOfEntries - 1);

        // Make last node reference new node
        lastNode.setNextNode(newNode);
    }

    numberOfEntries++;

    // assume things always complete, but out of memory is possible

    return true;
}
```

add() at a given position

- If at the start, the new node becomes the first node, and the next node of the new node is set to the old first node.
- If at another location, get the prior node and the one after it. Insert the new node between, setting the next references appropriately

```java
    public boolean add(int givenPosition, T newEntry) {
        // add to the position before given position, so to add to the end allow for
        // *before* the last last + 1 (or numberOfEntries)
        if (isInRange(givenPosition) || givenPosition == numberOfEntries) {

            Node<T> newNode = new Node<>(newEntry);

            if(isEmpty())
                firstNode = newNode;
            else if (givenPosition == 0) {
                // at the start, so just insert at beginning of chain
                newNode.setNextNode(firstNode);
                firstNode = newNode;
            } else {
                // non empty list
                // if there was a doubly linked list this would be a bit simpler
                Node<T> nodeBefore = getNodeAt(givenPosition - 1);
                Node<T> nodeAfter = nodeBefore.getNextNode();

                // insert the new node between the existing nodes
                newNode.setNextNode(nodeAfter);
                nodeBefore.setNextNode(newNode);
            }
            numberOfEntries++;
            return true;
        } else
            return false;
    }
```

remove() an entry at a given position

```
public T remove(int givenPosition) {

	if (!isInRange(givenPosition))
		throw new IndexOutOfBoundsException();

	T result = null; // Return value

	if (givenPosition == 0) {
		// this is the first entry in the list
		// save the data, then remove the entry
		result = firstNode.getData();
		firstNode = firstNode.getNextNode();
	} else {
		// this is a non-empty list
		// find the node at the position before the one we are looking
		// for.
		// then get the next node, which is the one to remove
		Node<T> nodeBefore = getNodeAt(givenPosition - 1);
		Node<T> nodeToRemove = nodeBefore.getNextNode();

		// save the removed node's data

		result = nodeToRemove.getData();

		// finally, remove the node by setting the
		// pointer of the previous node to the remove node's
		// next. Skip around the removed node.
		Node<T> nodeAfter = nodeToRemove.getNextNode();
		nodeBefore.setNextNode(nodeAfter);
	}
	numberOfEntries--; // Update count

	return result;
}
```

Set Next Node as first Node.

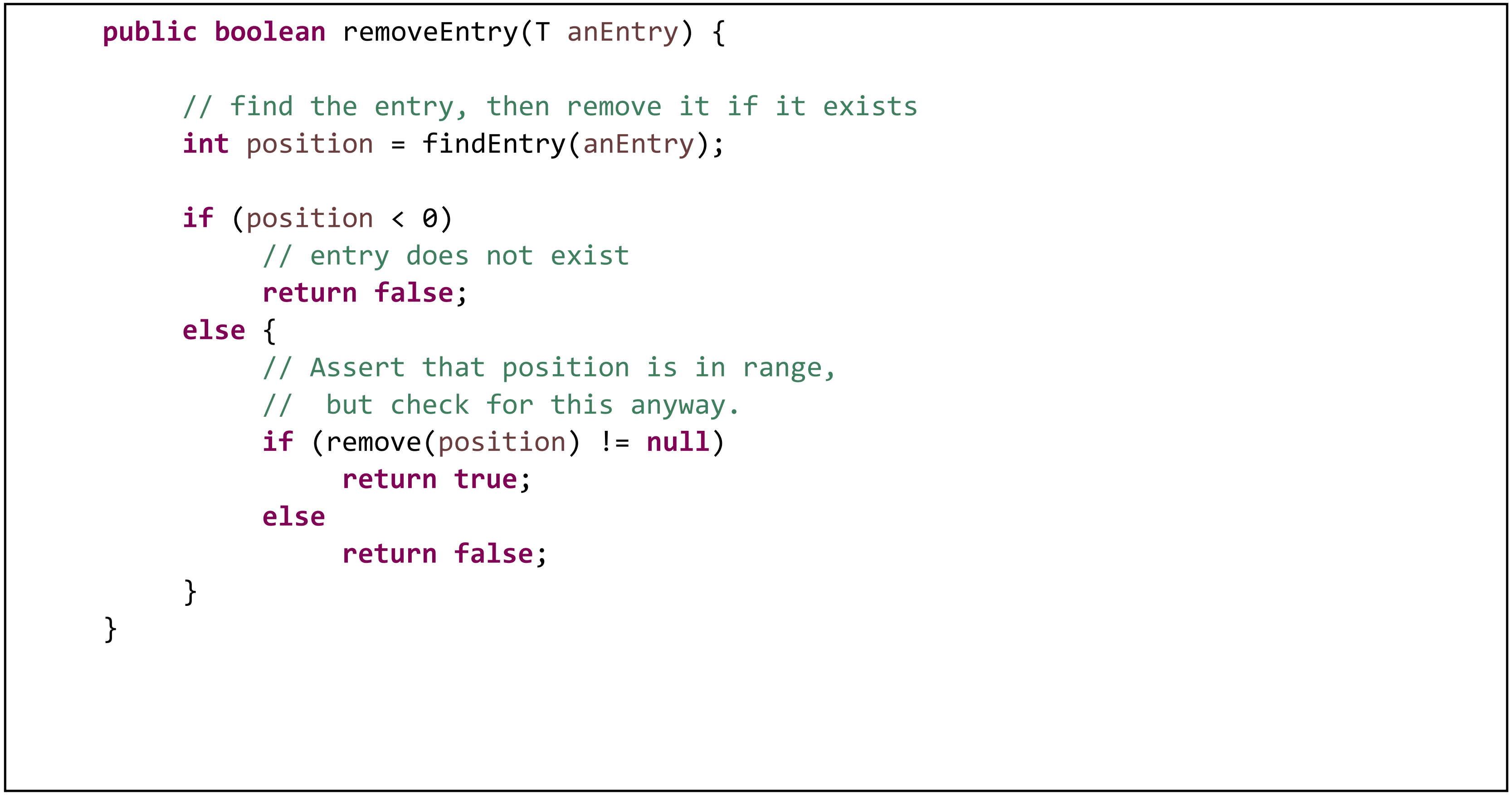
contains() and findEntry()

- findEntry() traverses the list until found
- contains() simply calls findEntry()

```java
public boolean contains(T anEntry) {
        return findEntry(anEntry) >= 0;
}

public int findEntry(T anEntry) {
        boolean found = false;

        // traverse the list looking for an entry

        int position = 0;
        Node<T> currentNode = firstNode;

        while (!found && (currentNode != null)) {
                if (anEntry.equals(currentNode.getData()))
                        found = true;
                else {
                        currentNode = currentNode.getNextNode();
                        position++;
                }
        }

        if (found == true)
                return position;
        else
                return -1;
}
```

removeEntry()

- find the entry, getting its position, then remove.
- not efficient, but keeps the code clean.



```
public boolean removeEntry(T anEntry) {

    // find the entry, then remove it if it exists
    int position = findEntry(anEntry);

    if (position < 0)
        // entry does not exist
        return false;
    else {
        // Assert that position is in range,
        //  but check for this anyway.
        if (remove(position) != null)
            return true;
        else
            return false;
    }
}
```

replace() and getEntry()

```java
public T replace(int givenPosition, T newEntry) {

    if (!isInRange(givenPosition))
        throw new IndexOutOfBoundsException();

    // find the node, then simply replace its data
    // Assert that desiredNode will not be node as
    // it is in the list range
    Node<T> desiredNode = getNodeAt(givenPosition);
    T originalEntry = desiredNode.getData();
    desiredNode.setData(newEntry);
    return originalEntry;
}

public T getEntry(int givenPosition) {

    if (!isInRange(givenPosition))
        throw new IndexOutOfBoundsException();

    // get the data from the node.
    // Assert that the node will not be null
    // as it is in the list range
    return getNodeAt(givenPosition).getData();

}
```

LList other methods

```
    public void clear() {
        initializeDataFields();
    }

    public int size() {
        return numberOfEntries;
    }

    public boolean isEmpty() {
        return numberOfEntries == 0;
    }

    /**
     * Return the first node in the list. Violates data hiding.
     *
     * Only used by iterator. In production, the subclass would be final,
     * or the iterator would be built in from the start.
     * @return
     */
    protected Node<T> getFirstNode() {
        return firstNode;
    }

    /**
     * Determines if a position is in the proper range of the list
     * @param givenPosition
     * @return
     */
    public boolean isInRange(int givenPosition) {
        return (givenPosition >= 0) && (givenPosition < numberOfEntries);
    }
```

toArray()

- need to traverse the list and copy each node’s data to an array slot.

```
public T[] toArray() {
    // The cast is safe because the new array contains null entries
    @SuppressWarnings("unchecked")
    T[] result = (T[]) new Comparable[numberOfEntries];

    // traverse the list copying the data at each step
    // note that the array will have zero entries but is not null
    //  if the size is zero.
    int index = 0;
    Node<T> currentNode = firstNode;
    while ((index < numberOfEntries) && (currentNode != null)) {
        result[index] = currentNode.getData();
        currentNode = currentNode.getNextNode();
        index++;
    }

    return result;
}
```

Testing and Examples

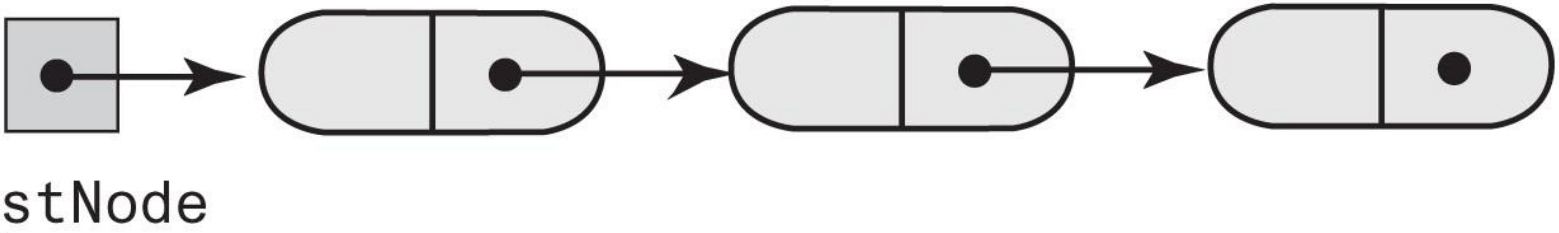
- See ListDemo and ListDemo2
- Both use display() from DemoUtilities class
- Both test AList and LList implementations and their variations.
  - Simply arrange uncommented code to perform the test of the implementation
- Output should be the same independent of implementation.

DemoUtilities display methods

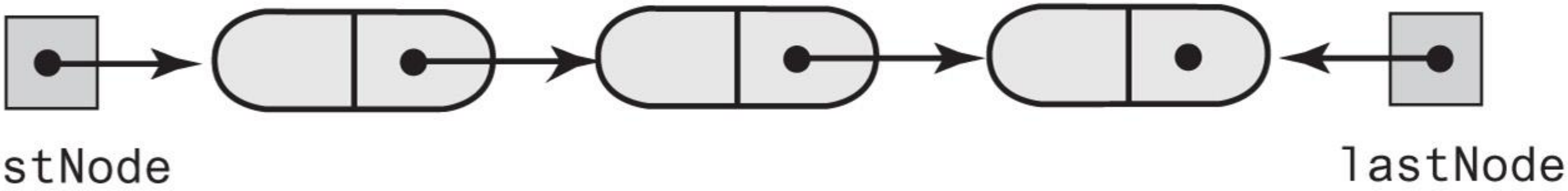
```
/**
 * display a list using the toArray method to retrieve items
 *
 * show a header line with a message and list size
 *
 * @param list
 * @param message
 */
static public <T> void display(ListInterface<T> list, String message) {
    System.out.println("Display: " + message + ", size = " + list.size());

    Object[] tempArray = list.toArray();

    @SuppressWarnings("unchecked")
    T[] listCopy = (T[]) tempArray;

    for (T item : listCopy)
        System.out.print(item + ", ");
    System.out.println();
}

/**
 * display a list using getEntry method to retrieve items
 *
 * show a header line with a message and list size
 *
 * @param list
 * @param message message header to be displayed along with list size
 */
static public <T> void displayUsingGetEntry(ListInterface<T> list, String message) {
    System.out.println("Display using getEntry(): " + message + ", size = " + list.size());

    for (int i = 0; i < list.size(); i++)
        System.out.print(list.getEntry(i) + ", ");
    System.out.println();
}
```

Using a Tail Reference

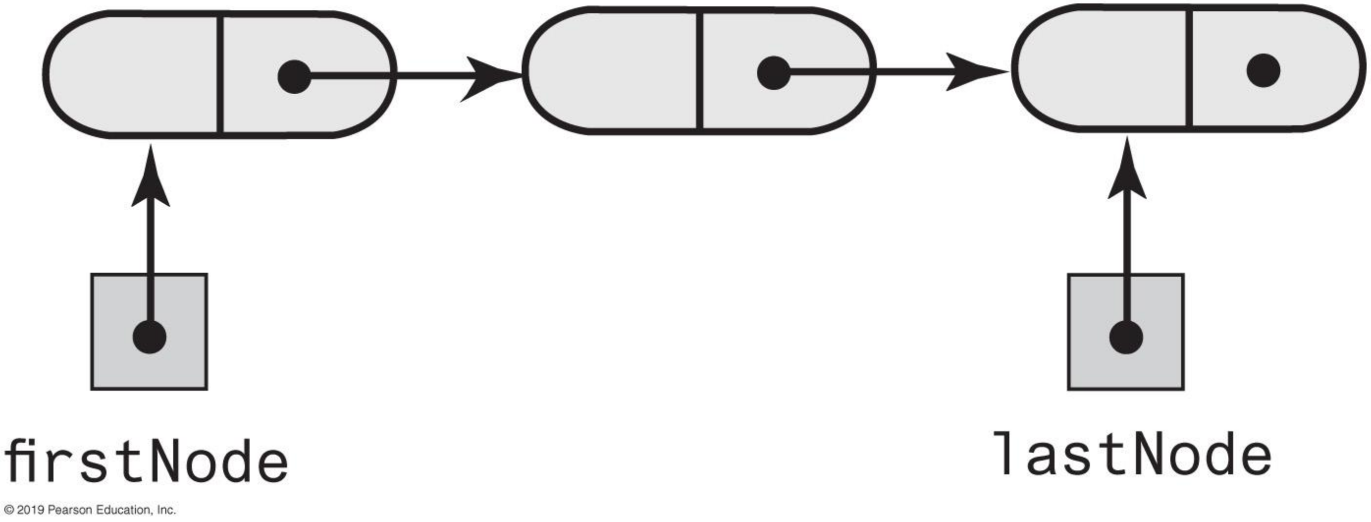
(a) With only a head reference



(b) With both a head reference and a tail reference

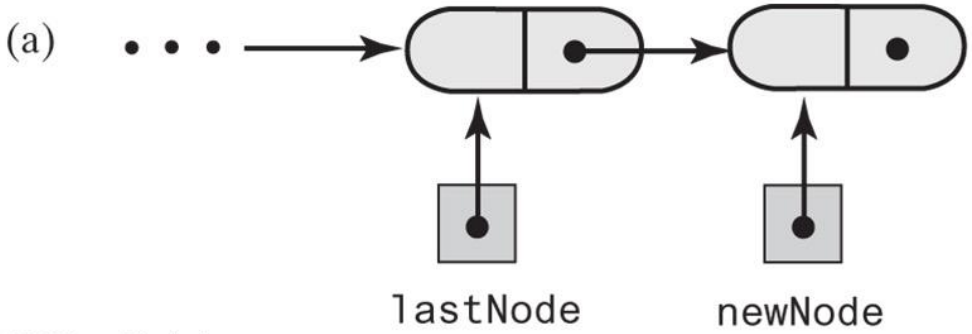


Linked chain with head and tail

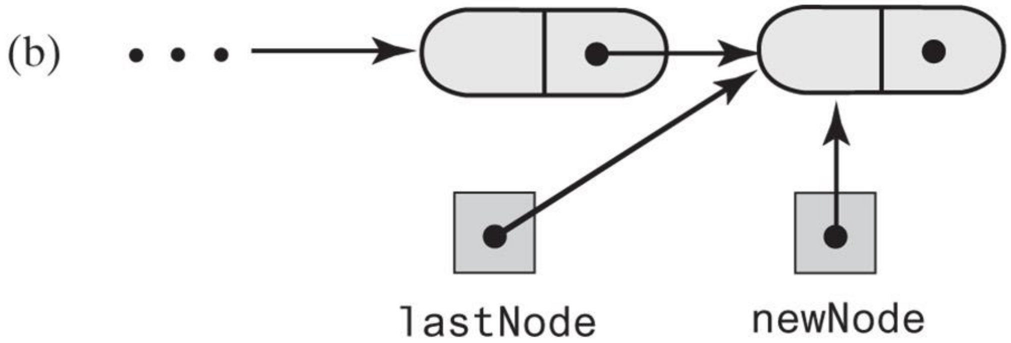


© 2019 Pearson Education, Inc.

Adding a node at the end of the chain



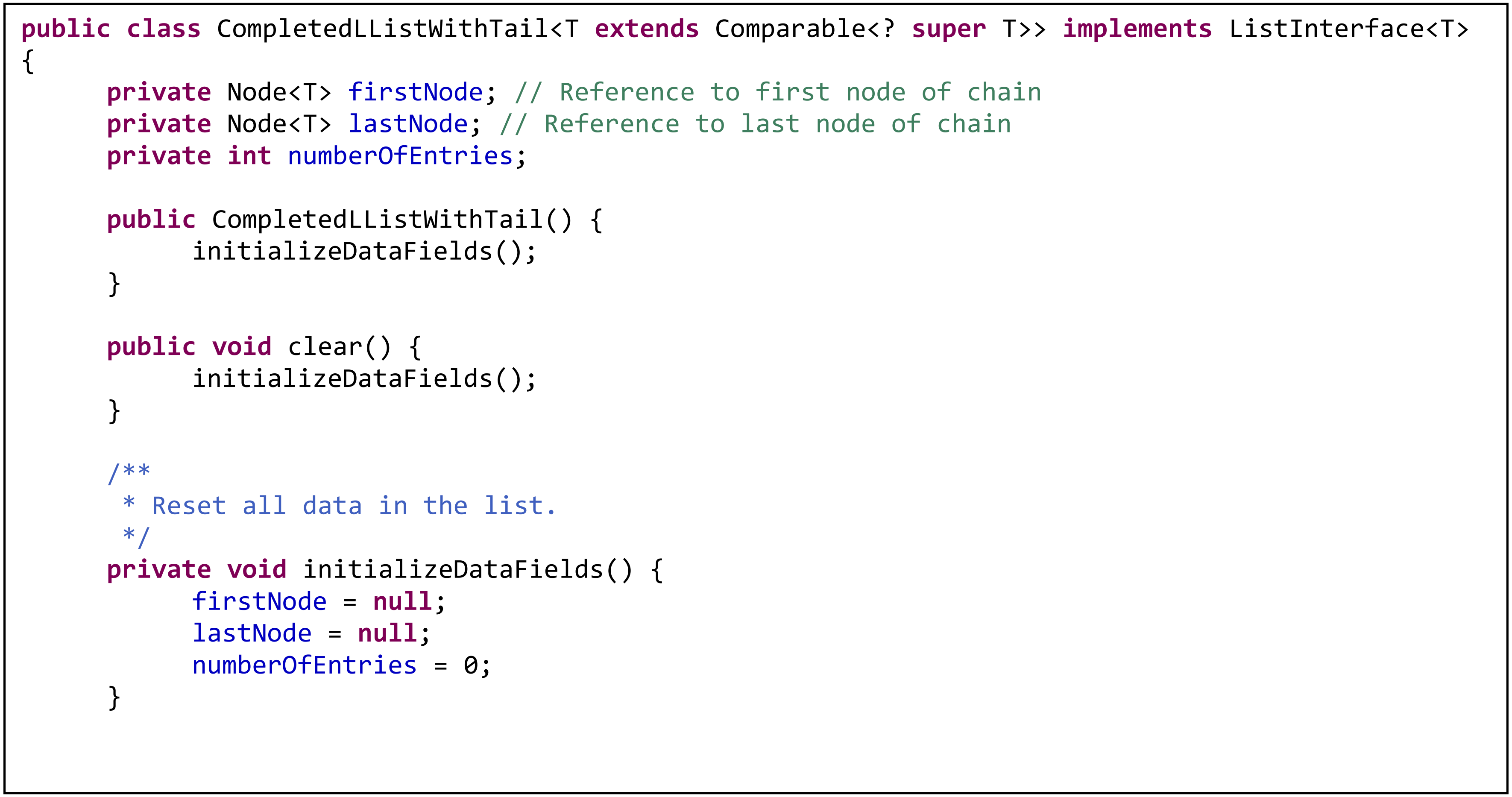
After executing
`lastNode.setNextNode(newNode);`



After executing
`lastNode = newNode;`

LListWithTail constructor

- Now has first and last nodes.
- Initialize by setting both to null



```java
public class CompletedLListWithTail<T extends Comparable<? super T>> implements ListInterface<T>
{
    private Node<T> firstNode; // Reference to first node of chain
    private Node<T> lastNode; // Reference to last node of chain
    private int numberOfEntries;

    public CompletedLListWithTail() {
        initializeDataFields();
    }

    public void clear() {
        initializeDataFields();
    }

    /**
     * Reset all data in the list.
     */
    private void initializeDataFields() {
        firstNode = null;
        lastNode = null;
        numberOfEntries = 0;
    }
```

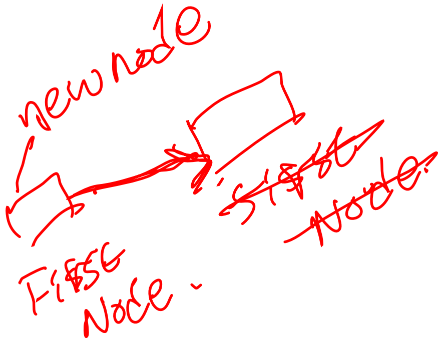
add() to the end of the list – Simple!

```java
    public boolean add(T newEntry) {
        Node<T> newNode = new Node<>(newEntry);

        if (isEmpty()) {
            firstNode = newNode;
        } else {
            // Add to end of nonempty list
            // Make current last node reference new node
            lastNode.setNextNode(newNode);
        }

        // reset lastNode to be the new one added

        lastNode = newNode;
        numberOfEntries++;

        // assume things always complete, but out of memory is possible

        return true;
    }
```

add() at a given position

- If at the end, simply adjust the last node
- No need to traverse the list with getNodeAt()

```
public boolean add(int givenPosition, T newEntry) {
    // add to the position before given position, so to add to the end allow for
    // *before* the last last + 1 (or numberOfEntries)
    if (isInRange(givenPosition) || givenPosition == numberOfEntries) {

        Node<T> newNode = new Node<>(newEntry);

        if (isEmpty()) {
            firstNode = newNode;
            lastNode = newNode;
        } else if (givenPosition == 0) {
            // at the start, so just insert at beginning of chain
            newNode.setNextNode(firstNode);
            firstNode = newNode;
        } else if (givenPosition == numberOfEntries) {
            // at the end of the chain, so simply modify last node
            lastNode.setNextNode(newNode);
            lastNode = newNode;
        } else {
            // is in the middle of the list
            // if there was a doubly linked list this would be a bit simpler
            Node<T> nodeBefore = getNodeAt(givenPosition - 1);
            Node<T> nodeAfter = nodeBefore.getNextNode();

            // insert the new node between the existing nodes
            newNode.setNextNode(nodeAfter);
            nodeBefore.setNextNode(newNode);
        }
        numberOfEntries++;
        return true;
    } else
        return false;
}
```

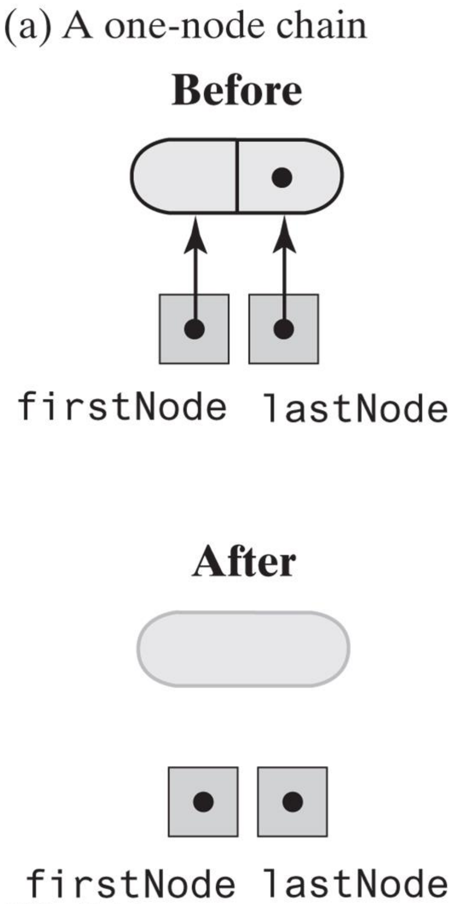


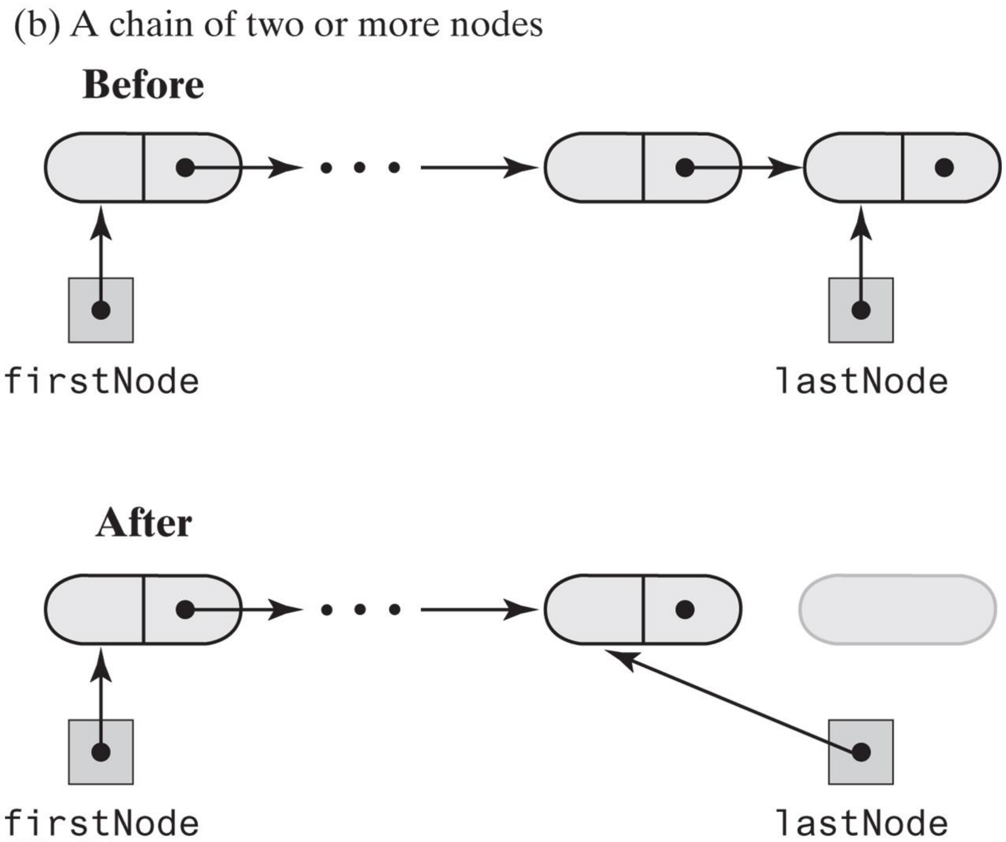
getNodeAt() optimizes using lastNode

```
    private Node<T> getNodeAt(int givenPosition) {

        if (isEmpty())
            return null;

        if (!isInRange(givenPosition))
            return null;

        // if the last position in the list is desired,
        // no need to traverse it

        if (givenPosition == numberOfEntries - 1)
            return lastNode;

        // Traverse the chain to locate the desired node

        Node<T> currentNode = firstNode;

        for (int counter = 0; counter < givenPosition; counter++)
            currentNode = currentNode.getNextNode();

        return currentNode;
    }
```

Removing a node

- Before and after removing the last node from a chain that has both head and tail references and contains one or more nodes





remove() just resets lastNode

```
    public T remove(int givenPosition) {

        if (!isInRange(givenPosition))
            throw new IndexOutOfBoundsException();

        T result = null; // Return value

        if (givenPosition == 0) {
            // this is the first entry in the list
            // save the data, then remove the entry
            result = firstNode.getData();
            firstNode = firstNode.getNextNode();
        } else {
            // this is a non-empty list
            // find the node at the position before the one we are looking
            // for.
            // then get the next node, which is the one to remove
            Node<T> nodeBefore = getNodeAt(givenPosition - 1);
            Node<T> nodeToRemove = nodeBefore.getNextNode();

            // save the removed node's data

            result = nodeToRemove.getData();

            // finally, remove the node by setting the
            // pointer of the previous node to the remove node's
            // next. Skip around the removed node.
            Node<T> nodeAfter = nodeToRemove.getNextNode();
            nodeBefore.setNextNode(nodeAfter);

            // if this is the last node, reset lastNode to the previous

            if (givenPosition == numberOfEntries - 1)
                lastNode = nodeBefore;
        }
        numberOfEntries--; // Update count

        return result;
    }
```

LListWithTail

- All other methods are exactly the same as LList!
- Reuse add(), getNodeAt() and remove()
- Cannot inherit from LList() as do not have access to Node<T> to be able to set lastNode.

Efficiency of Using a Chain

Operation	Alist	LList	LListWithTail
`add(newEntry)`	O(1)	O(*n*)	O(1)
`add(givenPosition, newEntry)`	O(*n*); O(n); O(1)	O(1); O(*n*)	O(1); O(*n*); O(1)
`toArray()`	O(*n*)	O(*n*)	O(*n*)
`remove(givenPosition)`	O(*n*); O(*n*); O(1)	O(1); O(*n*)	O(1); O(*n*)
`replace(givenPosition, newEntry)`	O(1)	O(1); O(*n*)	O(1); O(*n*); O(1)
`getEntry(givenPosition)`	O(1)	O(1); O(*n*)	O(1); O(*n*); O(1)
`contains(anEntry)`	O(*n*)	O(*n*)	O(*n*)
`clear(), getLength(), isEmpty()`	O(1)	O(1)	O(1)

Java Class Library: The Class `LinkedList`

- Implements the interface **`List`**
- **`LinkedList`** defines more methods than are in the interface `List`
- You can use the class **`LinkedList`** as implementation of ADT
  - **queue**
  - **deque**
  - or **list**.

Using Java library List

- Implement ListInterface using java library ArrayList,LinkedList

```java
public class CompletedListUsingLibraryLinkedList<T extends Comparable<? super T>>
    implements ListInterface<T>  {

    private List<T> list; // java library List interface

    public CompletedListUsingLibraryLinkedList() {
//      list = new LinkedList<>();
        list = new ArrayList<>();
    }

    public boolean add(T newEntry) {
        return list.add(newEntry);
    }
    public boolean add(int newPosition, T newEntry) {

        if(isInRange(newPosition)) {
            // it is in range of the current list, so add it
            list.add(newPosition, newEntry);
            return true;
        }
        else if(newPosition == size()) {
            // the request is to the next free slot past the end of the list,
            // so simply add it
            list.add(newEntry);
            return true;
        }
        else return false;
    }
    public T remove(int givenPosition) {
        if (!isInRange(givenPosition))
            throw new IndexOutOfBoundsException();
        return list.remove(givenPosition);
    }

    public boolean removeEntry(T anEntry) {
        return list.remove(anEntry);
    }
```

Using Java library List

```
    public void clear() {
        list.clear();
    }

    public T replace(int givenPosition, T newEntry) {
        if (!isInRange(givenPosition))
            throw new IndexOutOfBoundsException();
        return list.set(givenPosition, newEntry);
    }

    public T getEntry(int givenPosition) {
        if (!isInRange(givenPosition))
            throw new IndexOutOfBoundsException();
        return list.get(givenPosition);
    }

    public int findEntry(T anEntry) {
        return list.indexOf(anEntry);
    }

    @SuppressWarnings("unchecked")
    public T[] toArray() {
        T[] result = (T[]) new Comparable[list.size()];
        return list.toArray(result);
    }

    public boolean contains(T anEntry) {
        return list.contains(anEntry);
    }

    public int size() {
        return list.size();
    }

    public boolean isEmpty() {
        return list.isEmpty();
    }
```