Iterator

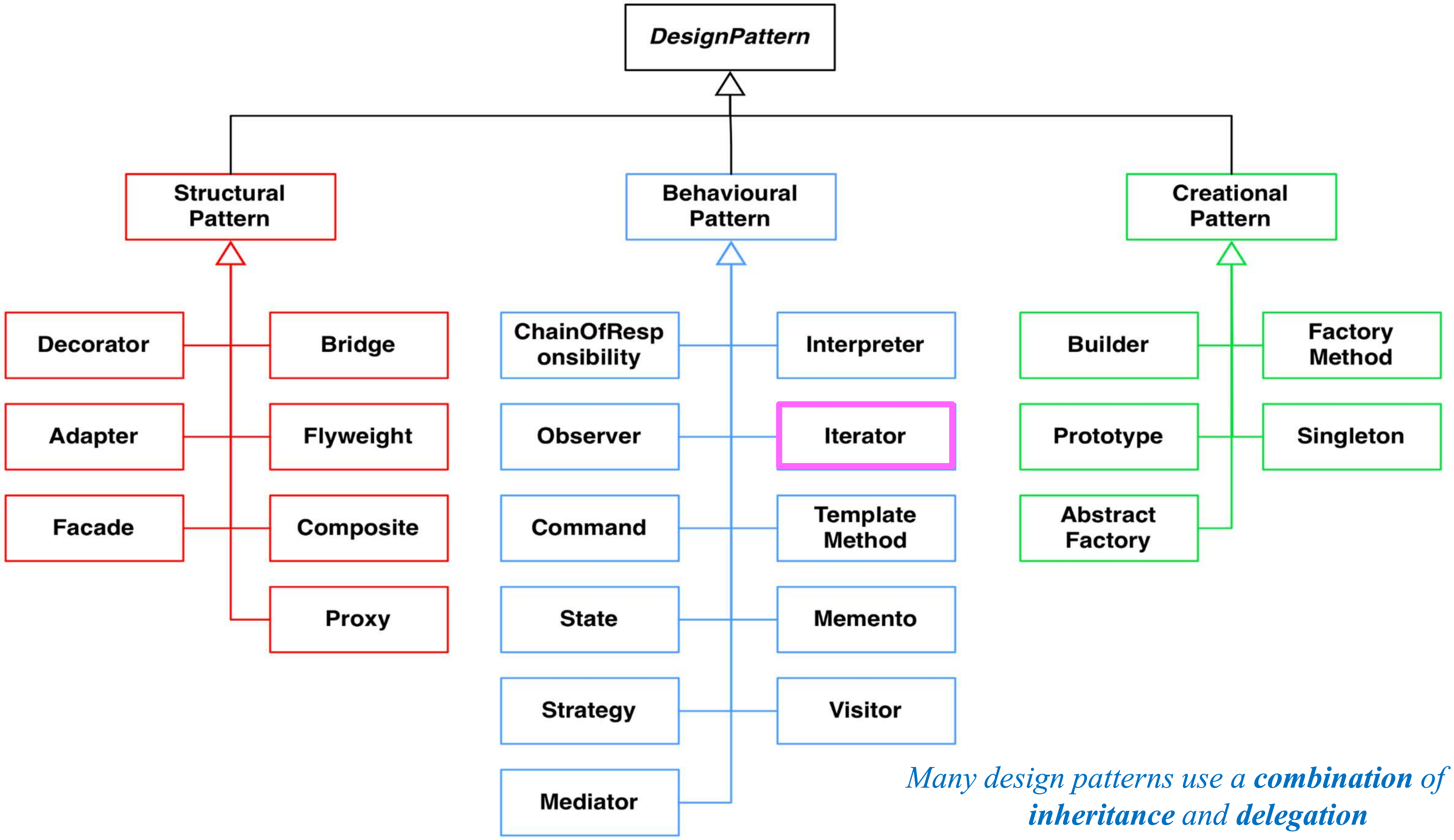
Dr. Youna Jung

Northeastern University

yo.jung@northeastern.edu

ITERATOR PATTERNS

Taxonomy of Design Patterns (23 Patterns)



Many design patterns use a ***combination*** *of* ***inheritance*** *and* ***delegation***

GoF: Iterator Pattern

- **Problem** – To access all members of a collection, must perform a specialized traversal **for each data structure**.
  - ✓ Introduces undesirable **dependences**
  - ✓ Does **not generalize** to other collections.
- **Solution**
  - ✓ Provide a **standard iterator** object supplied by **all** data **structures**.
  - ✓ **Results** are communicated to clients via a **standard interface**.
- **Consequence**
  - ✓ **Iteration order** is fixed by the **implementation**, **not** the **client**.
  - ✓ **Missing** various potentially **useful operations** (add(), set(), etc.).

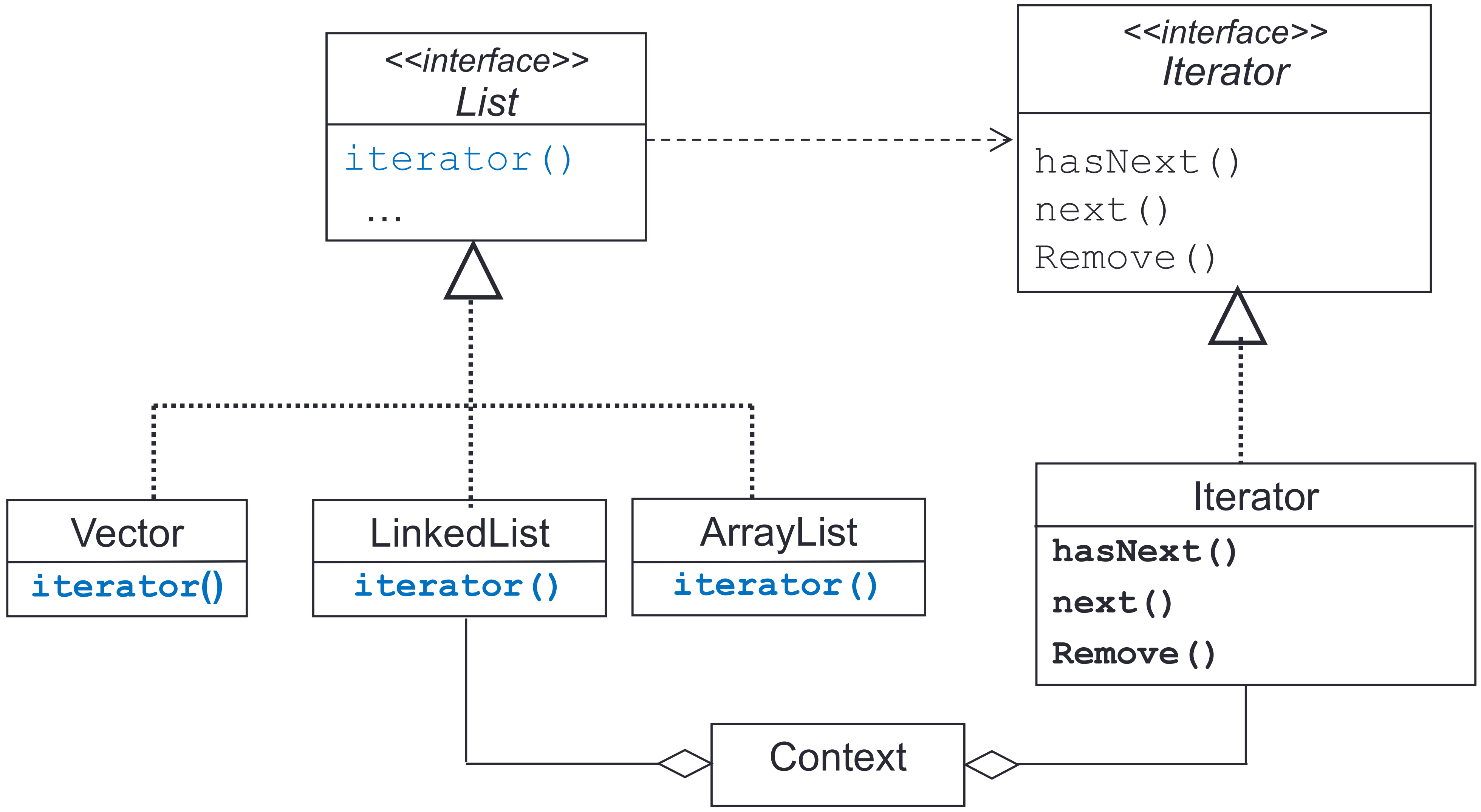
Iterator in Java

<<*interface*>>
Iterator

`boolean` **`hasNext()`**	*Returns true if the iteration has more elements.*
`Object` **`next()`**	*Returns the next element in the iteration*
`void` **`remove()`**	*Removes the most recently visited element*

```java
List<BankAccount> bank = new ArrayList<BankAccount>();
 bank.add(new BankAccount("One", 0.01));
 // ...
 bank.add(new BankAccount("Nine thousand", 9000.00));

 String ID = "Two";
 Iterator<BankAccount> i = bank.iterator();
 while(i.hasNext())  {
    if(i.next().getID().equals(searchAcct.getID()))
      System.out.println("Found " + ref.getID());
 }
```

Java: Iterator and Collections

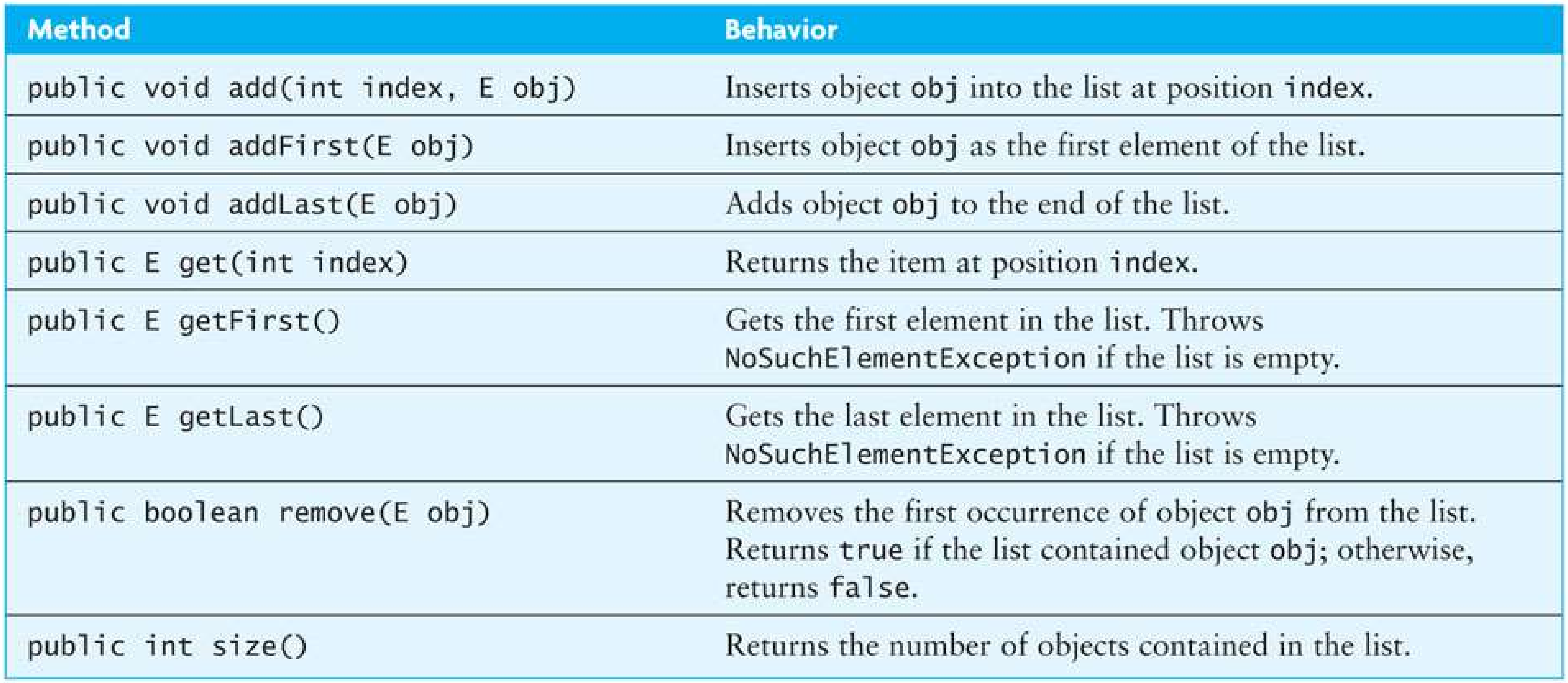


Example Use

```
import java.util.*;
public class IterateOverList {
    public static void main(String[] args) {
        // Change ArrayList to List
        List<String> names = new ArrayList<String>();
        names.add("Chris");
        names.add("Casey");
        names.add("Kim");
        Iterator<String> itr = names.iterator();
        while (itr.hasNext())
            System.out.println(itr.next());
    }
}
```

`LINKEDLIST` CLASS AND **`ITERATOR`**, **`LISTITERATOR`**, AND **`ITERABLE`** INTERFACES

Methods of Class `LinkedList<E>`



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

Iterator

- An iterator can be viewed as a moving place marker that **keeps track of the current position** in a particular linked list
- An `Iterator` object for a list starts at the list **head**
- The programmer can move the `Iterator` by calling its `next()` method.
- The `Iterator` stays on its current list item until it is needed

Iterator

- An `Iterator` traverses in $O(n)$ while a **linked list** traverse using get() calls in a linked list is $O(n^2)$

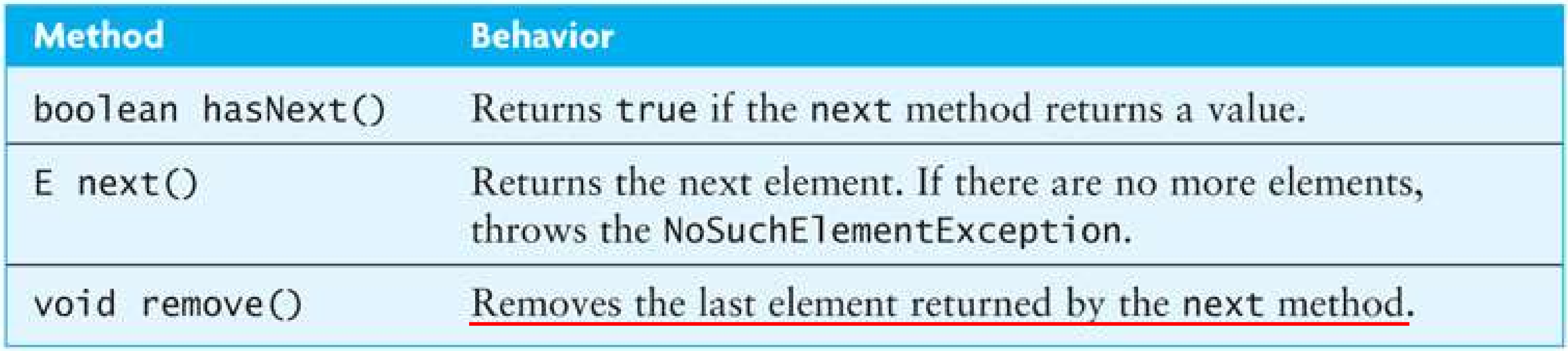
```
// Access each list element.
for (int index = 0; index < aList.size();index++) {
    E nextElement = aList.get(index);
    // Do something with the element at
       position index (nextElement)
     . . .
}
```

```
Node<E> nodeRef = head;
for (int j = 0; j < index; j++) {
    nodeRef = nodeRef.next;
}
```

$O(N^2)$

`Iterator` Interface

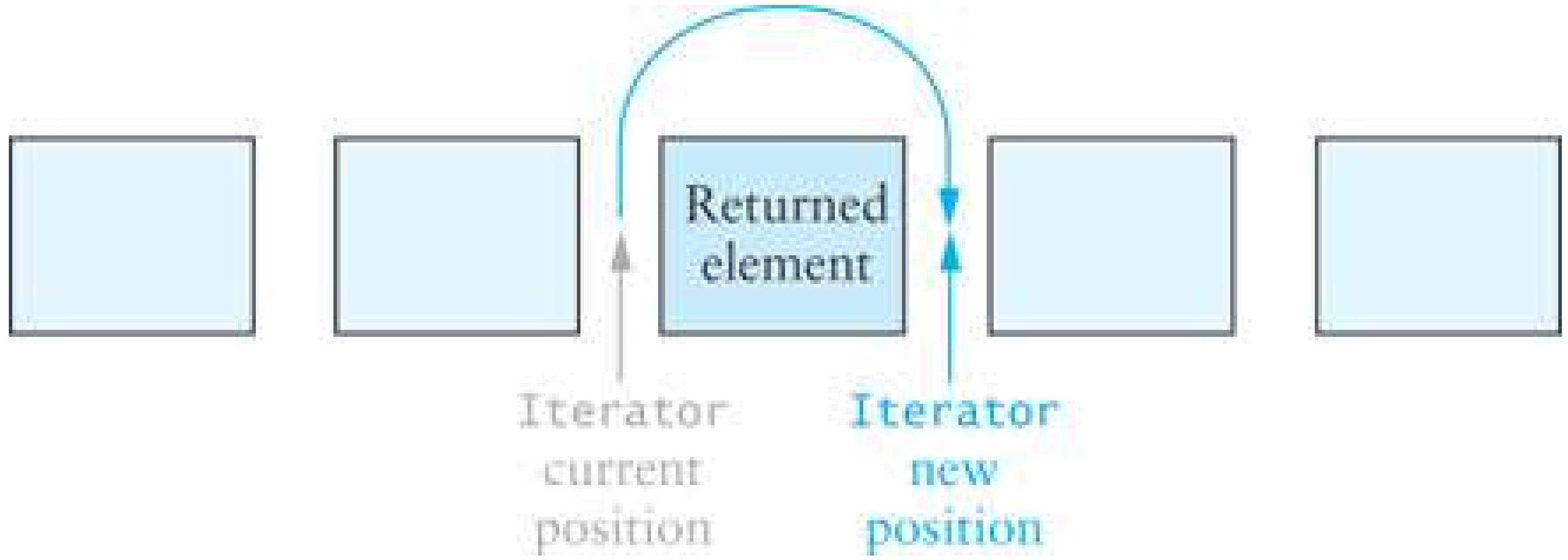
- The `Iterator` interface is defined in `java.util`
- declares the method `iterator`
  - ✓ **returns** an `Iterator` object that iterates over the elements of that list



Method	Behavior
`boolean hasNext()`	Returns `true` if the `next` method returns a value.
`E next()`	Returns the next element. If there are no more elements, throws the `NoSuchElementException`.
`void remove()`	Removes the last element returned by the `next` method.

Iterator Interface

- An Iterator is conceptually **between** elements
- It does not refer to a ~~particular~~ object at any given time



Process all items using `Iterator`

- **Process all items** in **`List<Integer>`** through an **`Iterator`**

```
Iterator<Integer> iter = lst.iterator();
while (iter.hasNext()) {
  int value = itr.next();
  // Do something with value
  ...
}
```

Iterators and Removing Elements

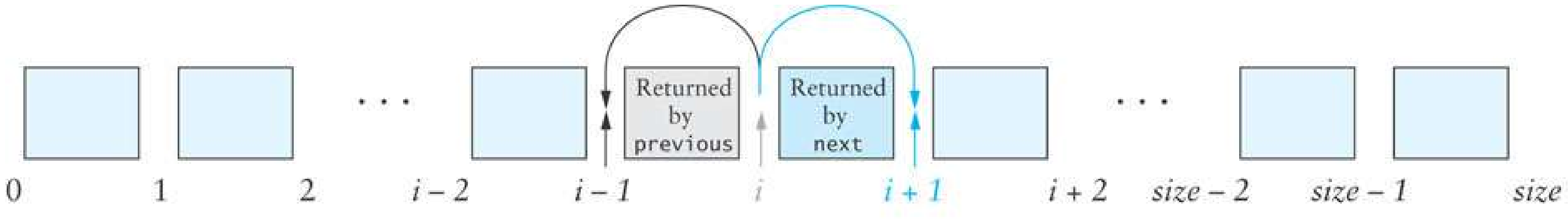
- You can use the `Iterator.remove()` to remove items from a list as you access them
- `remove()` deletes **the most recent element returned**
  - You must call **`next()`** before each **`remove()`**;
  - otherwise, an **`IllegalStateException`** will be thrown
- **`LinkedList.remove`** vs. **`Iterator.remove`**:
  - **`LinkedList.remove`** must walk down the list each time, then remove, so in general it is **$O(n^2)$**
  - **`Iterator.remove`** removes items without starting over at the beginning, so in general it is **$O(1)$**
    - An item to be removed is already returned by next()

`ListIterator` Interface

- ❑ `Iterator` limitations
  - ✓ **Traverses `List` only** in the **forward** direction
  - ✓ Provides `remove()` method, but **no `add()`** method
  - ✓ You must advance the `Iterator` using your own loop if you do not start from the beginning of the list
- ❑ `ListIterator` extends `Iterator`, overcoming these limitations

`ListIterator` Interface (cont.)

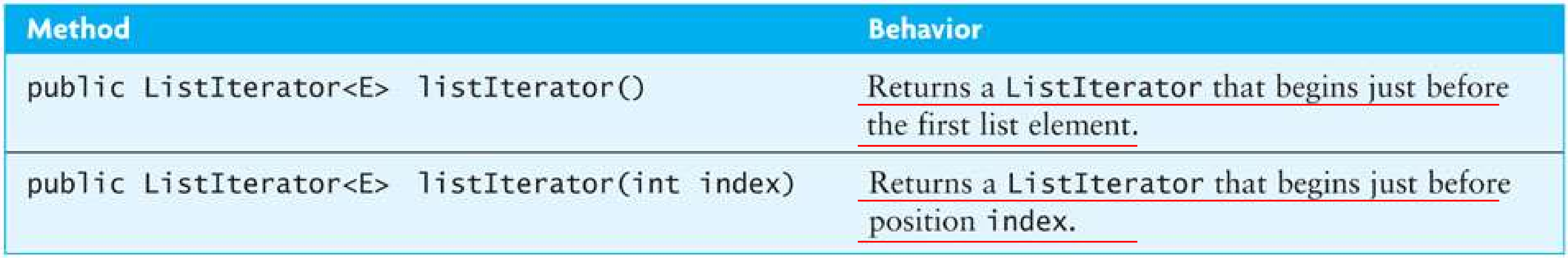
- As with `Iterator`, **`ListIterator`** is conceptually positioned between elements of the list
- **`ListIterator` positions** are assigned an index from 0 to **`size`**



ListIterator<E> Interface

Method	Behavior
`void add(E obj)`	Inserts object `obj` into the list just before the item that would be returned by the next call to method `next` and after the item that would have been returned by method `previous`. If method `previous` is called after `add`, the newly inserted object will be returned.
`boolean hasNext()`	Returns `true` if `next` will not throw an exception.
`boolean hasPrevious()`	Returns `true` if `previous` will not throw an exception.
`E next()`	Returns the next object and moves the iterator forward. If the iterator is at the end, the `NoSuchElementException` is thrown.
`int nextIndex()`	Returns the index of the item that will be returned by the next call to `next`. If the iterator is at the end, the list size is returned.
`E previous()`	Returns the previous object and moves the iterator backward. If the iterator is at the beginning of the list, the `NoSuchElementExcepton` is thrown.
`int previousIndex()`	Returns the index of the item that will be returned by the next call to `previous`. If the iterator is at the beginning of the list, −1 is returned.
`void remove()`	Removes the last item returned from a call to `next` or `previous`. If a call to `remove` is not preceded by a call to `next` or `previous`, the `IllegalStateException` is thrown.
`void set(E obj)`	Replaces the last item returned from a call to `next` or `previous` with `obj`. If a call to `set` is not preceded by a call to `next` or `previous`, the `IllegalStateException` is thrown.

Methods that return **ListIterators**



Method	Behavior
`public ListIterator<E> listIterator()`	Returns a `ListIterator` that begins just before the first list element.
`public ListIterator<E> listIterator(int index)`	Returns a `ListIterator` that begins just before position `index`.

Iterator V.S. ListIterator

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

Enhanced `for` (= for each loop)

- The enhanced `for` statement creates an `Iterator` object and **implicitly** calls its `hasNext()` and `next()` methods
  - ✓ Other `Iterator` methods, such as `remove`, are not available in the enhanced for statement

Enhanced `for` Statement (cont.)

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

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

Enhanced `for` Statement (cont.)

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

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

`Iterable` Interface

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

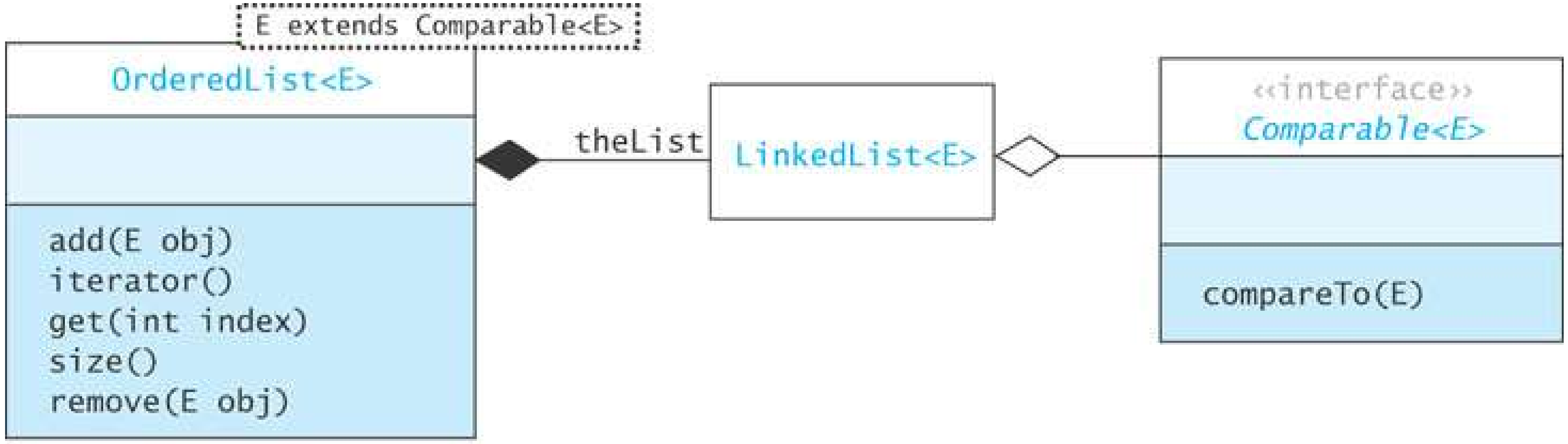
```
public interface Iterable<E> {
   Iterator<E> iterator();
}
```

APPLICATION OF THE `LINKEDLIST` CLASS

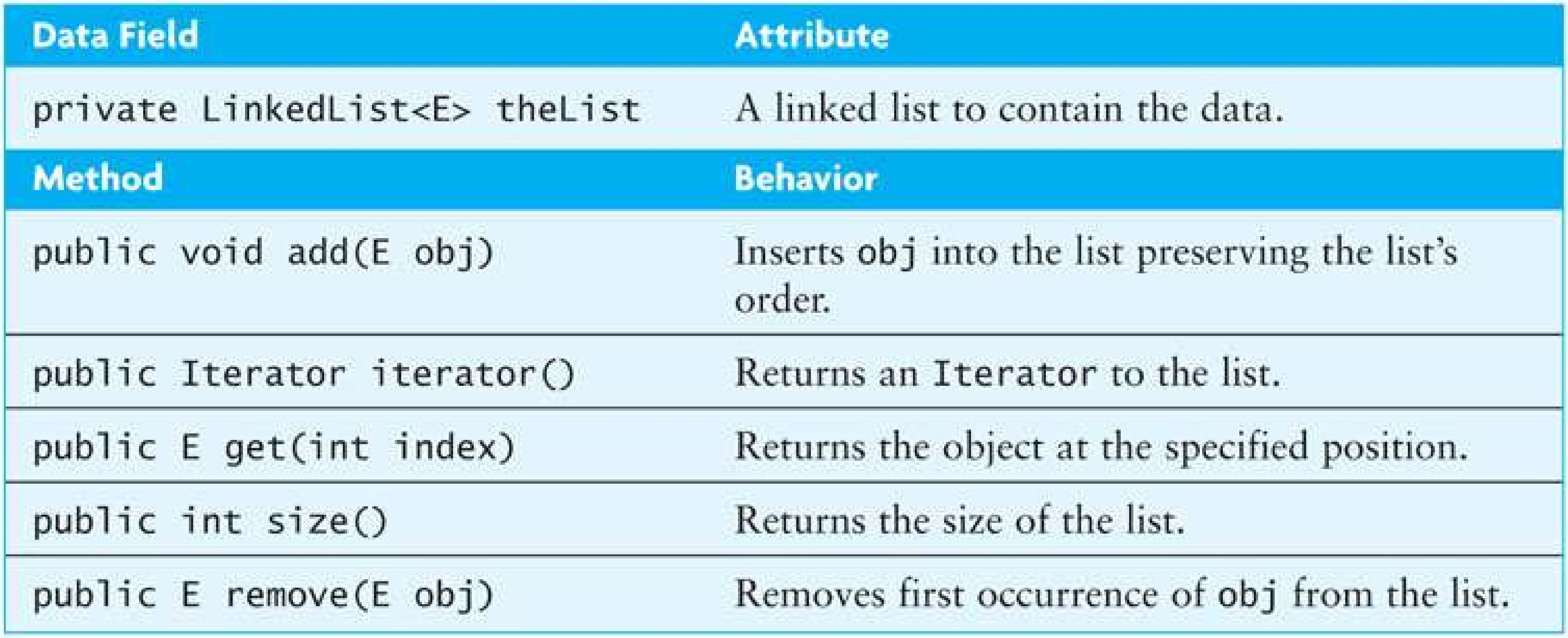
An Application: Ordered Lists

- We want to maintain a **list** of **names** in **alphabetical order** at all times
- **Approach**
  - Develop an **`OrderedList`** class (which can be used for other applications)
    - Use a **`LinkedList`** class as a component of the **`OrderedList`**
      - Implement a **`Comparable`** interface by providing a `compareTo(E)` method

OrderedList



Design of OrderedList



Data Field	Attribute
`private LinkedList<E> theList`	A linked list to contain the data.
Method	**Behavior**
`public void add(E obj)`	Inserts `obj` into the list preserving the list's order.
`public Iterator iterator()`	Returns an `Iterator` to the list.
`public E get(int index)`	Returns the object at the specified position.
`public int size()`	Returns the size of the list.
`public E remove(E obj)`	Removes first occurrence of `obj` from the list.

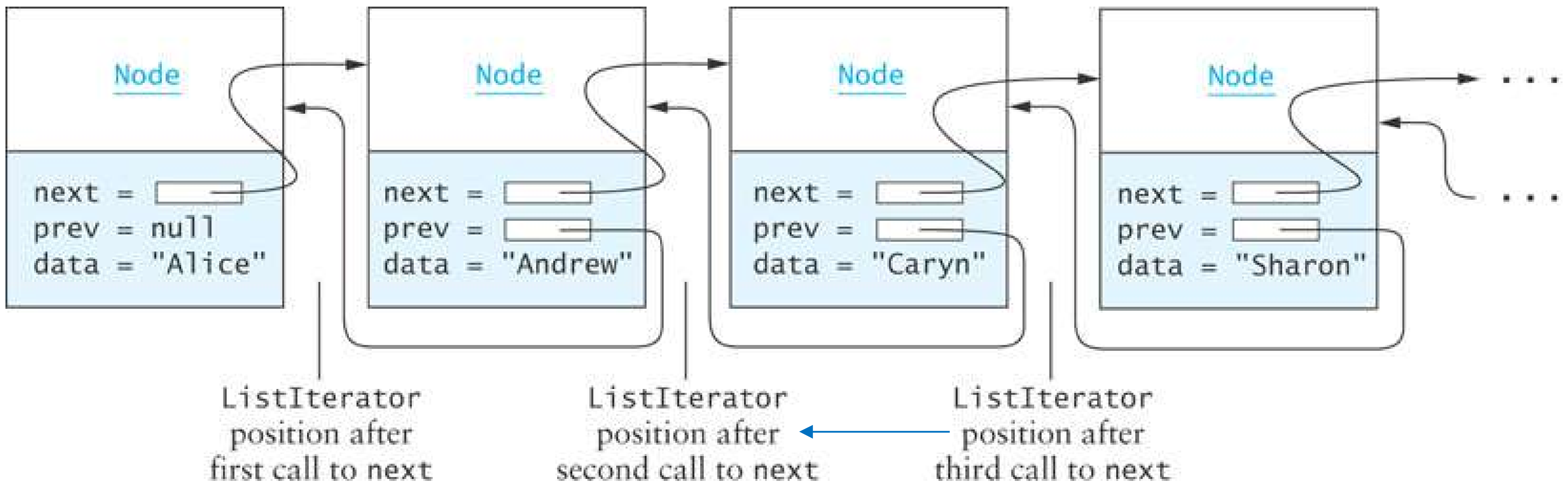
Inserting into an `OrderedList`

- Strategy for inserting new element `e`:
  - Find first item > `e`
  - Insert `e` before that item
- Refined with an iterator:
  - Create `ListIterator` that starts at the beginning of the list
  - While the `ListIterator` is not at the end of the list and `e` >= the next item
    - Advance the `ListIterator`
  - Insert `e` **before the current** `ListIterator` position

OrderedList.add()

```
public void add (E e) {
  ListIterator<E> iter = theList.listIterator();
  while (iter.hasNext()) {
    if (e.compareTo(iter.next()) < 0) {
      // found element > new one
         at that time, iter passed the element already
      iter.previous();  // back up by one
      iter.add(e);      // add new one
      return;           // done        }
  }
  iter.add(e);  // will add at end
}
```

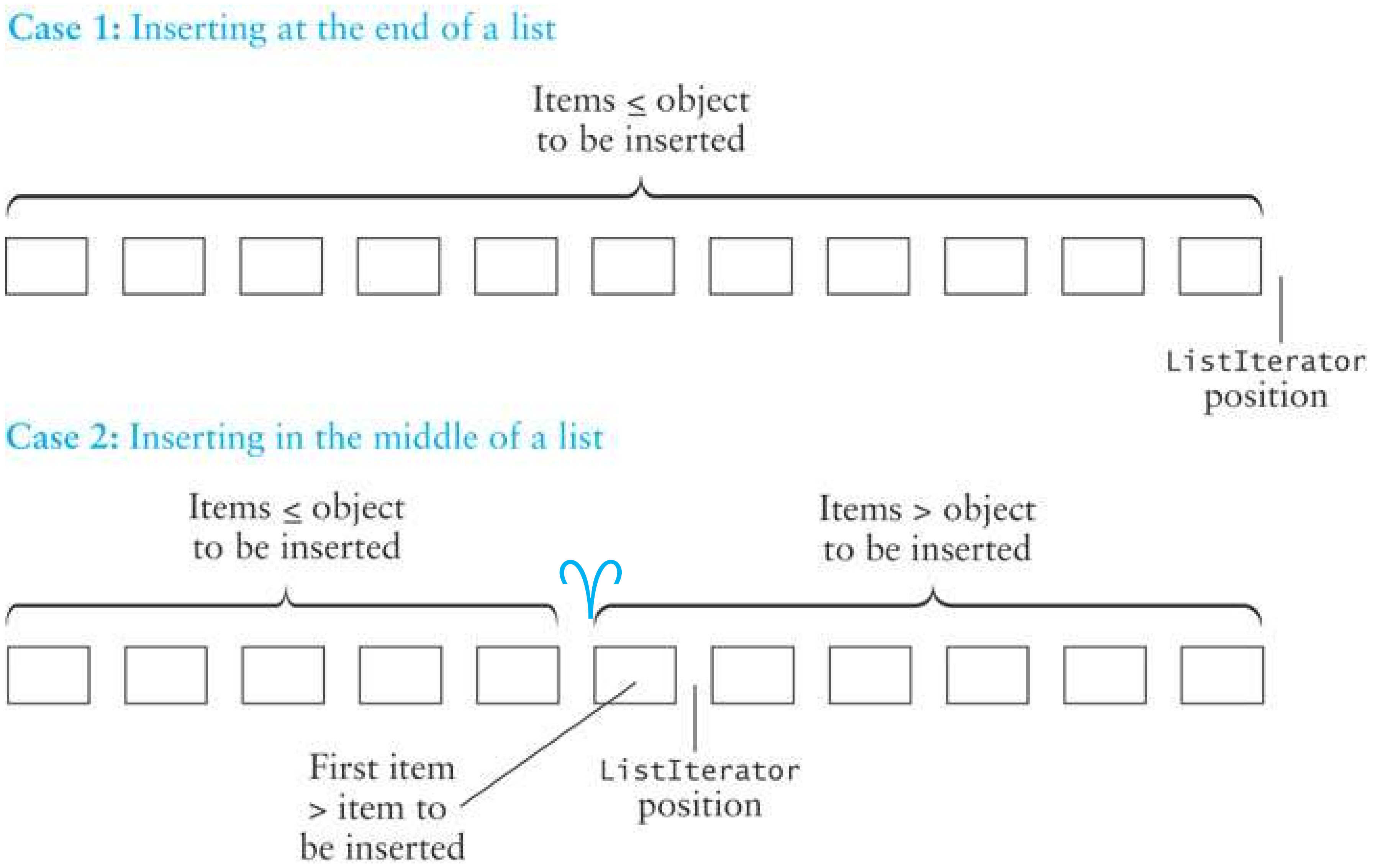
Inserting "Bill"



Node
next =
prev = null
data = "Alice"
Node
next =
prev =
data = "Andrew"
Node
next =
prev =
data = "Caryn"
Node
next =
prev =
data = "Sharon"
. . .
. . .
ListIterator position after first call to next
ListIterator position after second call to next
ListIterator position after third call to next

- Start at list head and call next() to advance iterator to first node whose data is larger than "Bill".
- Node with data "Caryn" is first node with data larger than "Bill".
- Iterator has passed the node that follows "Bill".
- Call previous() to move iterator back one position and insert a new node with data "Bill.

Inserting at End of List is Special Case


Case 1: Inserting at the end of a list
Items ≤ object to be inserted
ListIterator position
Case 2: Inserting in the middle of a list
Items ≤ object to be inserted
Items > object to be inserted
First item > item to be inserted
ListIterator position

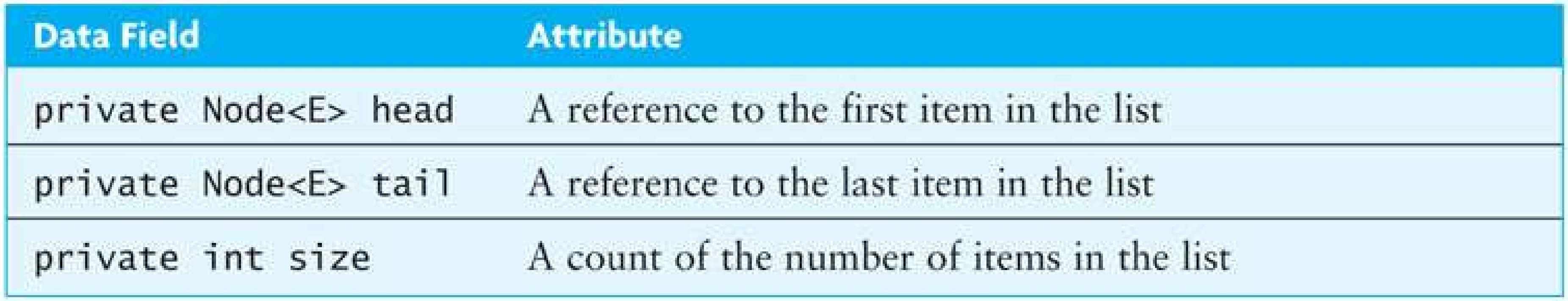
Using Delegation to Implement the Other Methods

```
public E get (int index) {
   return theList.get(index);
}
public int size () {
   return theList.size();
}
public E remove (E e) {
   return theList.remove(e);
}
public Iterator iterator() {
   return theList.iterator();
}
```

IMPLEMENTATION OF A DOUBLE-LINKED LIST CLASS

`KWLinkedList` Data Fields

- We will define a `KWLinkedList` class that implements some of the methods of the `List` interface
- The `KWLinkedList` class is for demonstration purposes only; Java provides **a standard `LinkedList` class** in `java.util` that you should use in your programs



Data Field	Attribute
`private Node<E> head`	A reference to the first item in the list
`private Node<E> tail`	A reference to the last item in the list
`private int size`	A count of the number of items in the list

Implementing **KWLinkedList**

```
import java.util.*;

/** Class KWLinkedList implements a double linked list and
 *   a ListIterator. */

public class KWLinkedList <E> {
    // Data Fields
    private Node <E> head = null;

    private Node <E> tail = null;

    private int size = 0;
      . . .
```

`KWLinkedList.add()`

```
public void add(int index, E obj) {
  listIterator(index).add(obj);
}
```

Algorithm for method `add`:

1. Obtain a reference, `nodeRef`, to the node at position `index`
2. Insert a new `Node` containing `obj` before the node referenced by `nodeRef`

Use a `ListIterator` to implement the algorithm:

1. Obtain an iterator that is positioned just before the `Node` currently at position `index`
2. Use method `add` to insert the new object before the `Node` referenced by this iterator

It is not necessary to declare a local `ListIterator`; the method call `listIterator` returns an anonymous `ListIterator` object

KWLinkedList.get()

```
public E get(int index) {
   return  listIterator(index).next();
}
```

Algorithm for **get()**:

1. Obtain a reference, `nodeRef`, to the node at position `index`
2. Insert a new `Node` containing `obj` before the node referenced by `nodeRef`

Use a **ListIterator** to implement **get()**

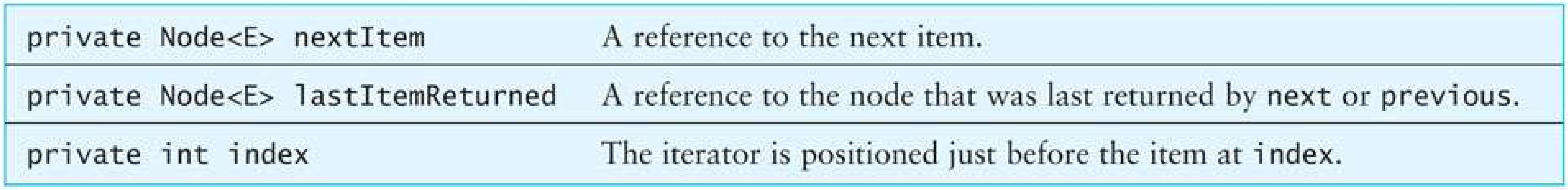
1. Obtain an iterator that is positioned just before the object at position `index`
2. Use method `next` to get the object currently referenced by this iterator

add() and get()

```
public void addFirst(E item) {
   add(0, item);
}

public void addLast(E item) {
   add(size, item);
}

public E getFirst() {
   return head.data;
}

public E getLast() {
   return tail.data;
}
```

Data Fields of **`KWListIter`**

- **`KWListIter`** is an inner class of **`KWLinkedList`** that implements the `ListIterator` interface

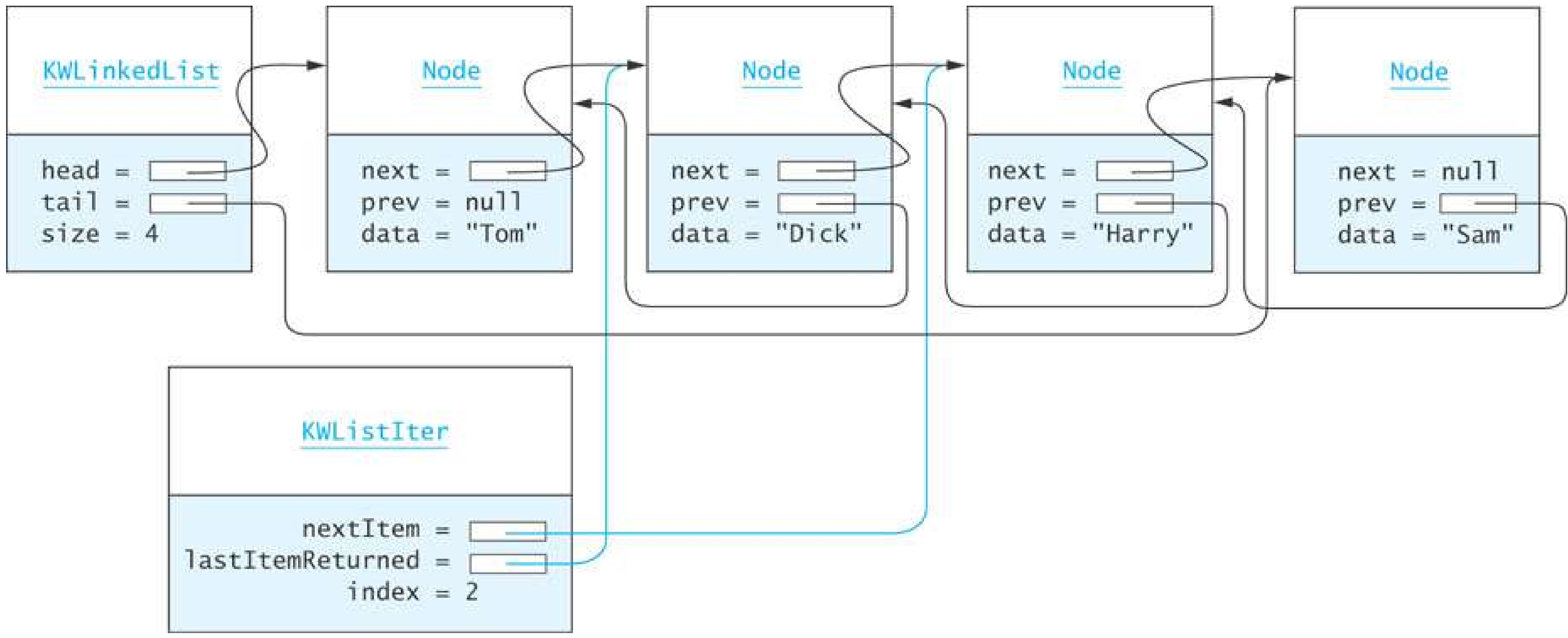


`private Node<E> nextItem`	A reference to the next item.
`private Node<E> lastItemReturned`	A reference to the node that was last returned by `next` or `previous`.
`private int index`	The iterator is positioned just before the item at `index`.

Implementing the `ListIterator` Interface

```
private class KWListIter implements ListIterator<E> {
   private Node <E> nextItem;
   private Node <E> lastItemReturned;
   private int index = 0;
  ...
```

KWLinkedList with KWListIter



Constructor for `KWListIter`

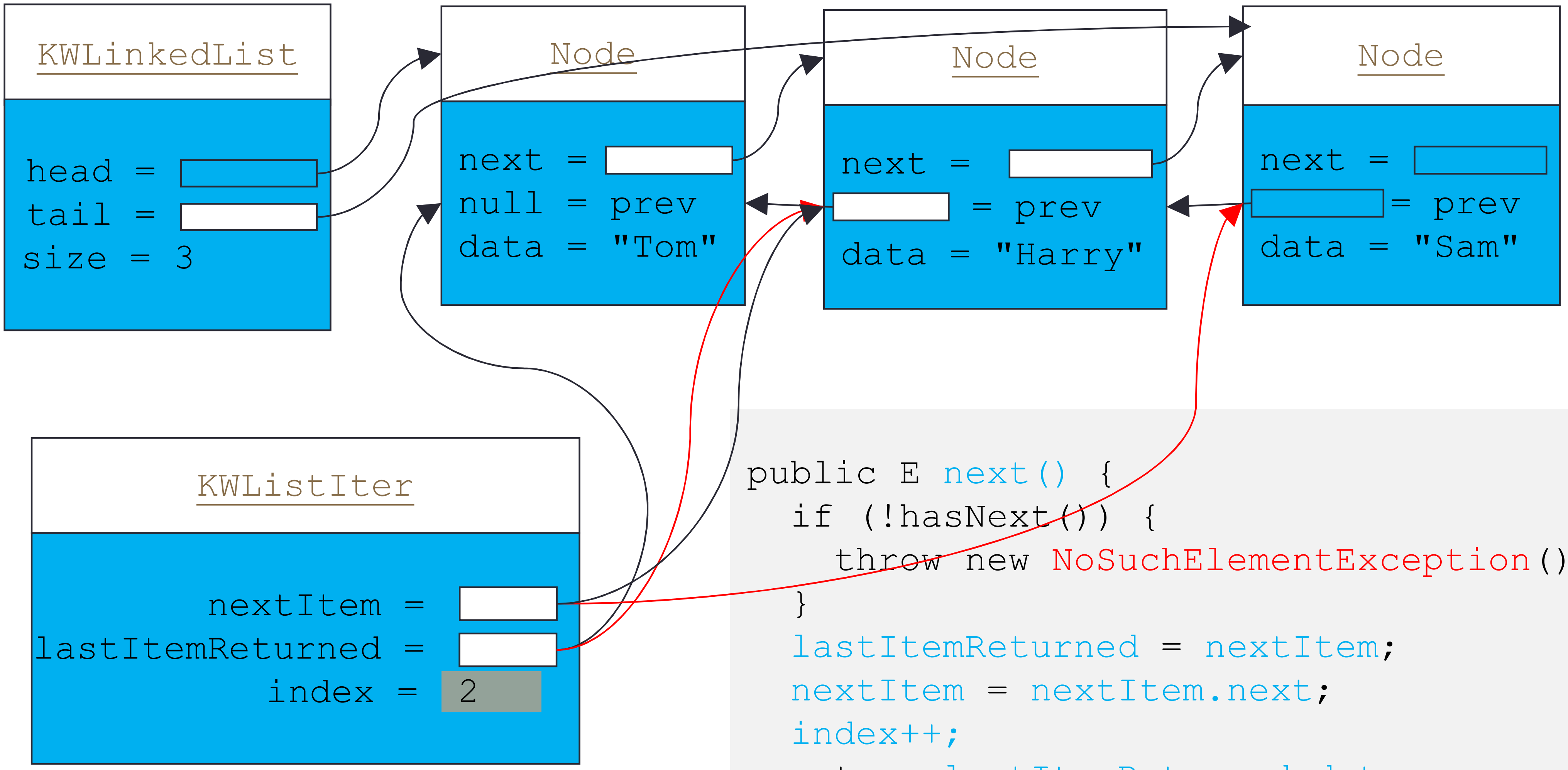
```
public KWListIter(int i) {
  // Validate i parameter.
  if (i < 0 || i > size) {
    throw new IndexOutOfBoundsException("Invalid index " + i);
  }
  lastItemReturned = null; // No item returned yet.
  // Special case of last item.
  if (i == size) {
    index = size;
    nextItem = null;   }
  else { // Start at the beginning
    nextItem = head;
    for (index = 0; index < i; index++) {
      nextItem = nextItem.next;     }
  }
}
```

`hasNext()` Method

- Tests to see if **`nextItem`** is **`null`**

```
public boolean hasnext() {
   return nextItem != null;
}
```

next() Methods



```
public E next() {
   if (!hasNext()) {
      throw new NoSuchElementException();
   }
   lastItemReturned = nextItem;
   nextItem = nextItem.next;
   index++;
   return lastItemReturned.data;
}
```

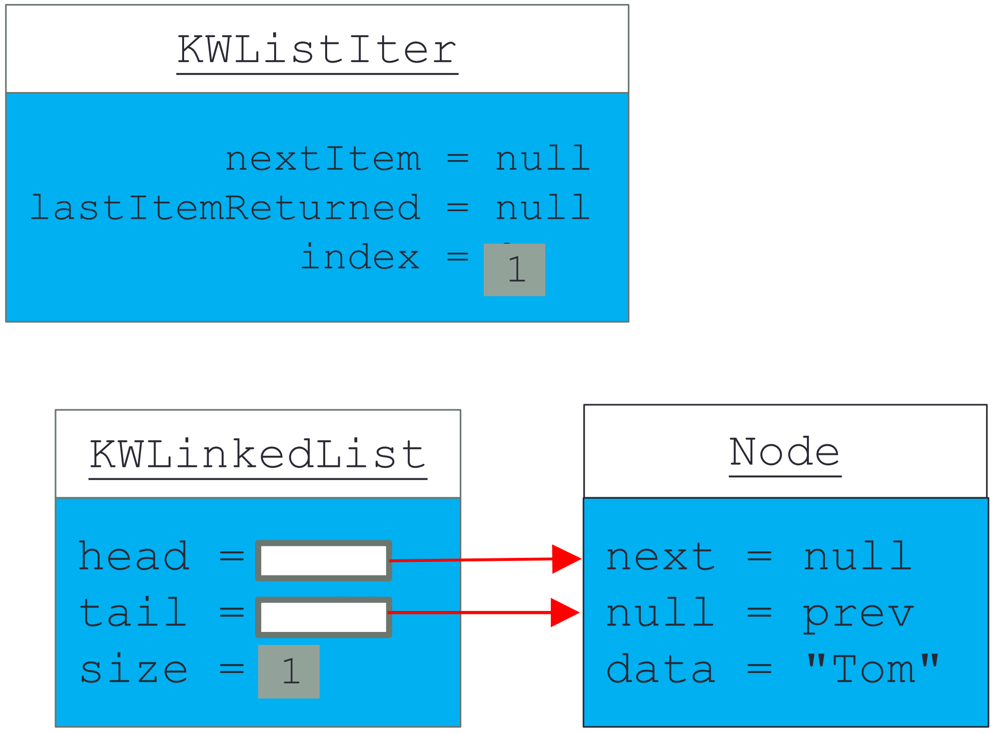
previous() Methods

```
public boolean hasPrevious() {
  return (nextItem == null && size != 0) || nextItem.prev != null;
}

public E previous() {
  if (!hasPrevious()) {
    throw new NoSuchElementException();
  }
  if (nextItem == null) { // Iterator past the last element
    nextItem = tail;
  }
  else {
    nextItem = nextItem.prev;
  }
  lastItemReturned = nextItem;
  index--;
  return lastItemReturned.data;
}
```

The `Add()` Method

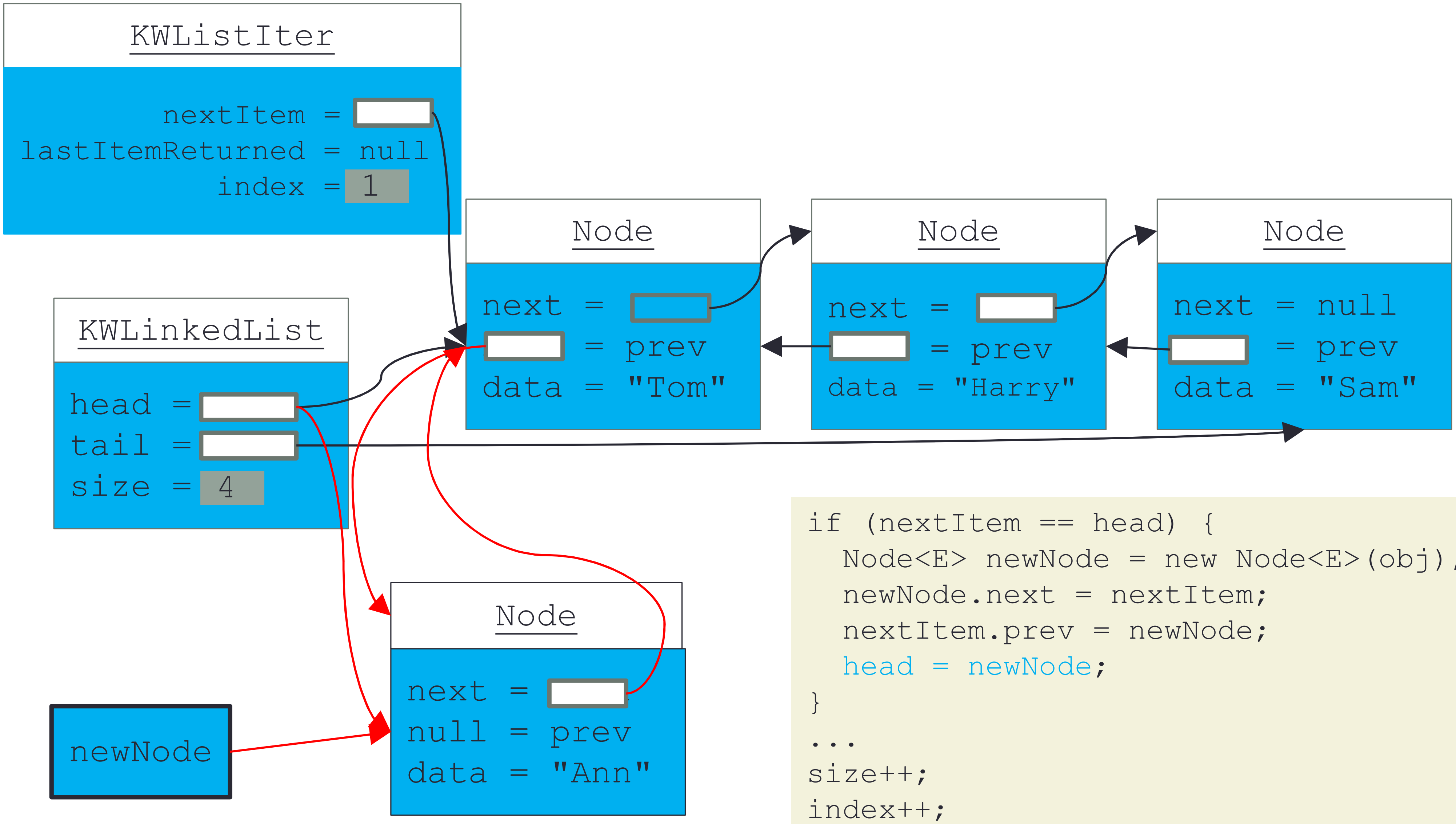
- When **adding**, there are 4 **cases** to address:
  - ✓ Add to an **empty list**
  - ✓ Add to the **head** of the list
  - ✓ Add to the **tail** of the list
  - ✓ Add to the **middle** of the list

Adding to an Empty List



```
if (head == null) {
   head = new Node<E>(obj);
   tail = head;
}
...
size++;
index++;
```

Adding to the Head of the List

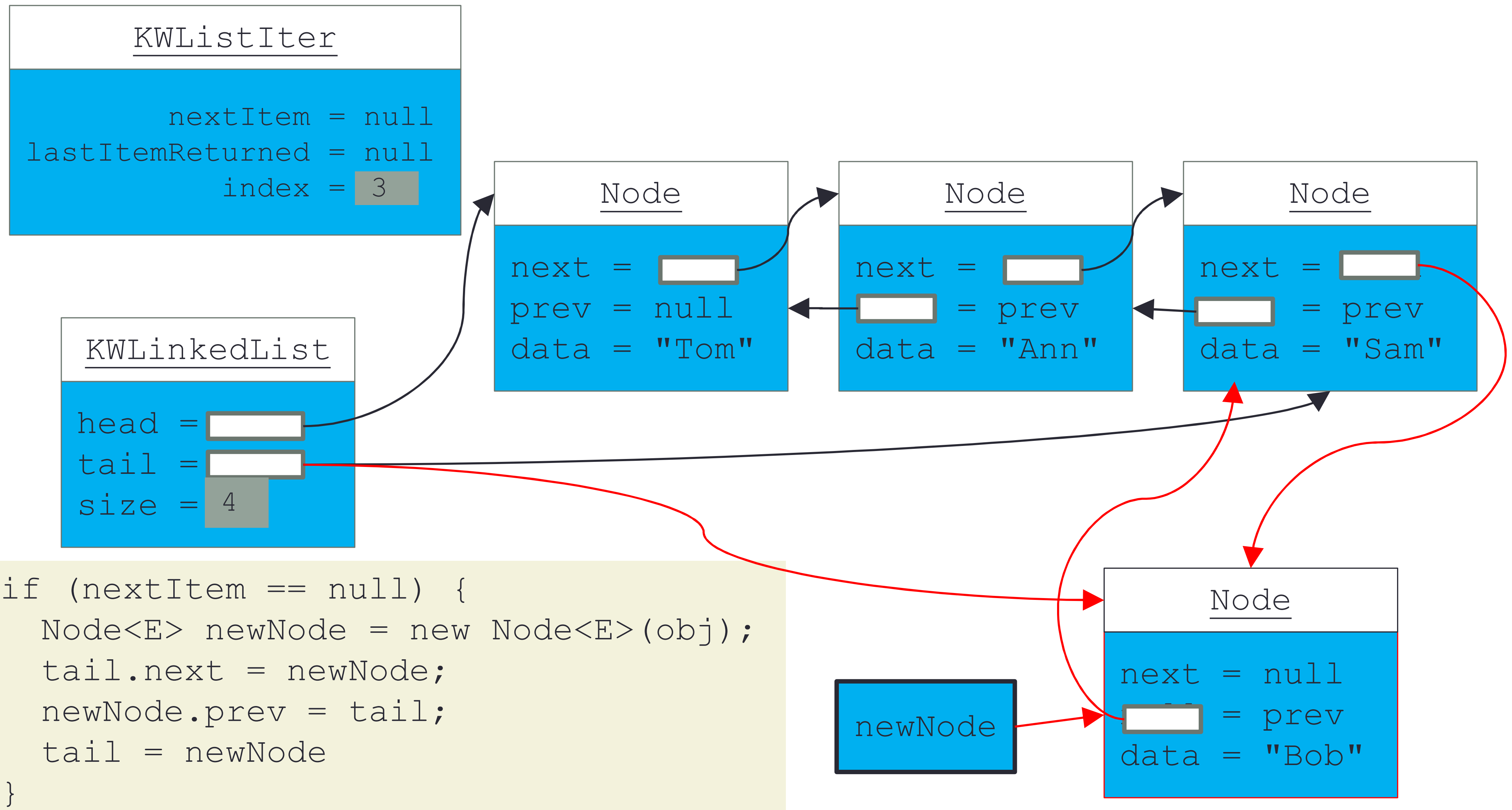


```
if (nextItem == head) {
  Node<E> newNode = new Node<E>(obj);
  newNode.next = nextItem;
  nextItem.prev = newNode;
  head = newNode;
}
...
size++;
index++;
```

Slide 49

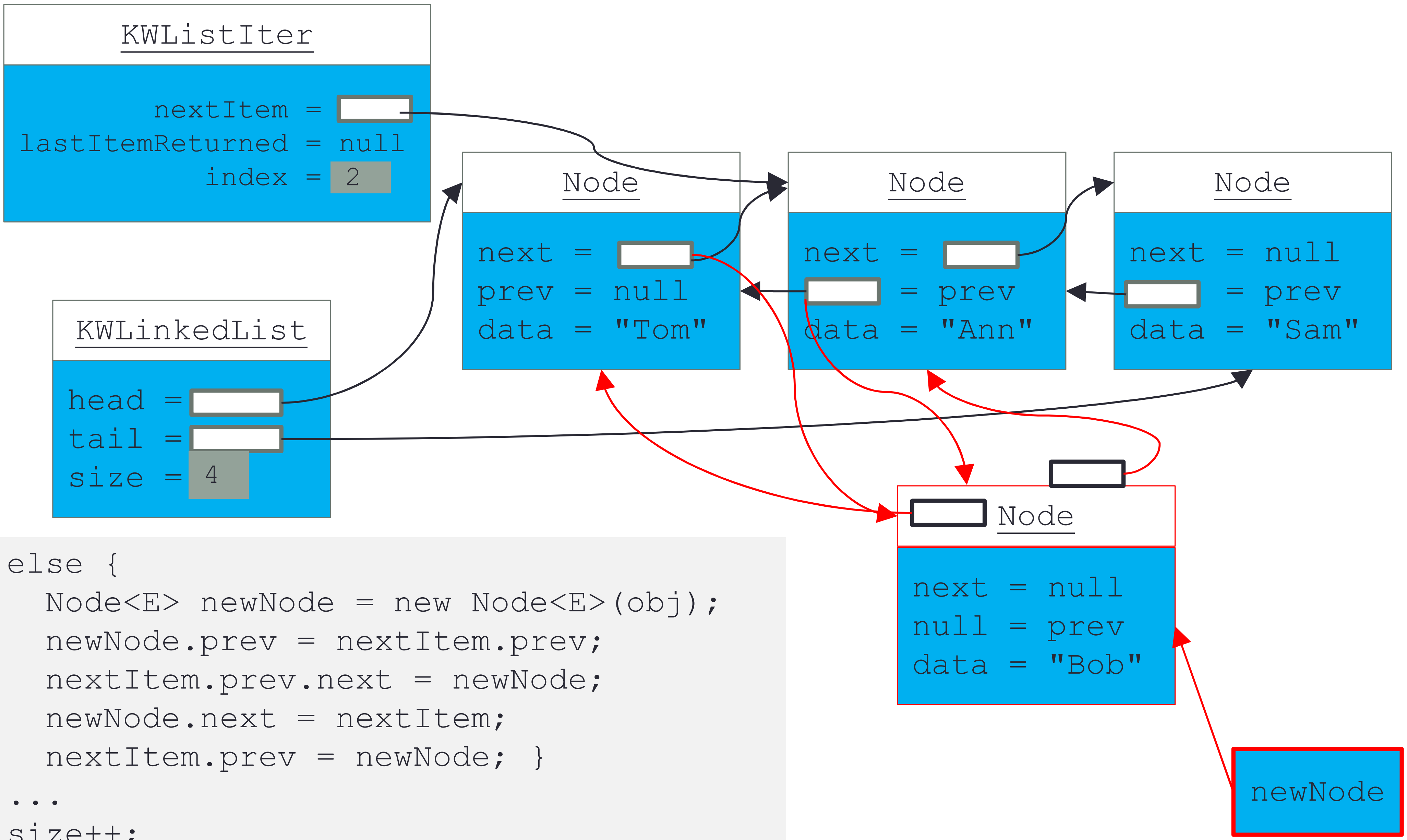
59472	Comp: Avoid overlap. 005947, 9/26/2020
EK2	I can't redraw. this is the best I can do. Elliot Koffman, 11/20/2020

Adding to the Tail of the List



```
if (nextItem == null) {
  Node<E> newNode = new Node<E>(obj);
  tail.next = newNode;
  newNode.prev = tail;
  tail = newNode
}
...
size++;
index++;
```

Adding to the Middle of the List



```
else {
  Node<E> newNode = new Node<E>(obj);
  newNode.prev = nextItem.prev;
  nextItem.prev.next = newNode;
  newNode.next = nextItem;
  nextItem.prev = newNode; }
...
size++;
index++;
```