CHAPTER 2

Lists and the `Collections` Framework

Chapter Objectives

- The `List` interface
- Writing an array-based implementation of `List`
- Linked list data structures:
  - Singly-linked
  - Doubly-linked
  - Circular
- Big-O notation and algorithm efficiency
- Implementing the `List` interface as a linked list
- The `Iterator` interface
- Implementing `Iterator` for a linked list
- Testing strategies
- The Java `Collections` framework (hierarchy)

Introduction

- A *list* is a collection of elements, each with a position or index
- *Iterators* facilitate sequential access to lists
- Classes `ArrayList`, `Vector`, and `LinkedList` are *subclasses* of abstract class `AbstractList` and *implement* the `List` interface

The `List` Interface and `ArrayList` Class

Section 2.1

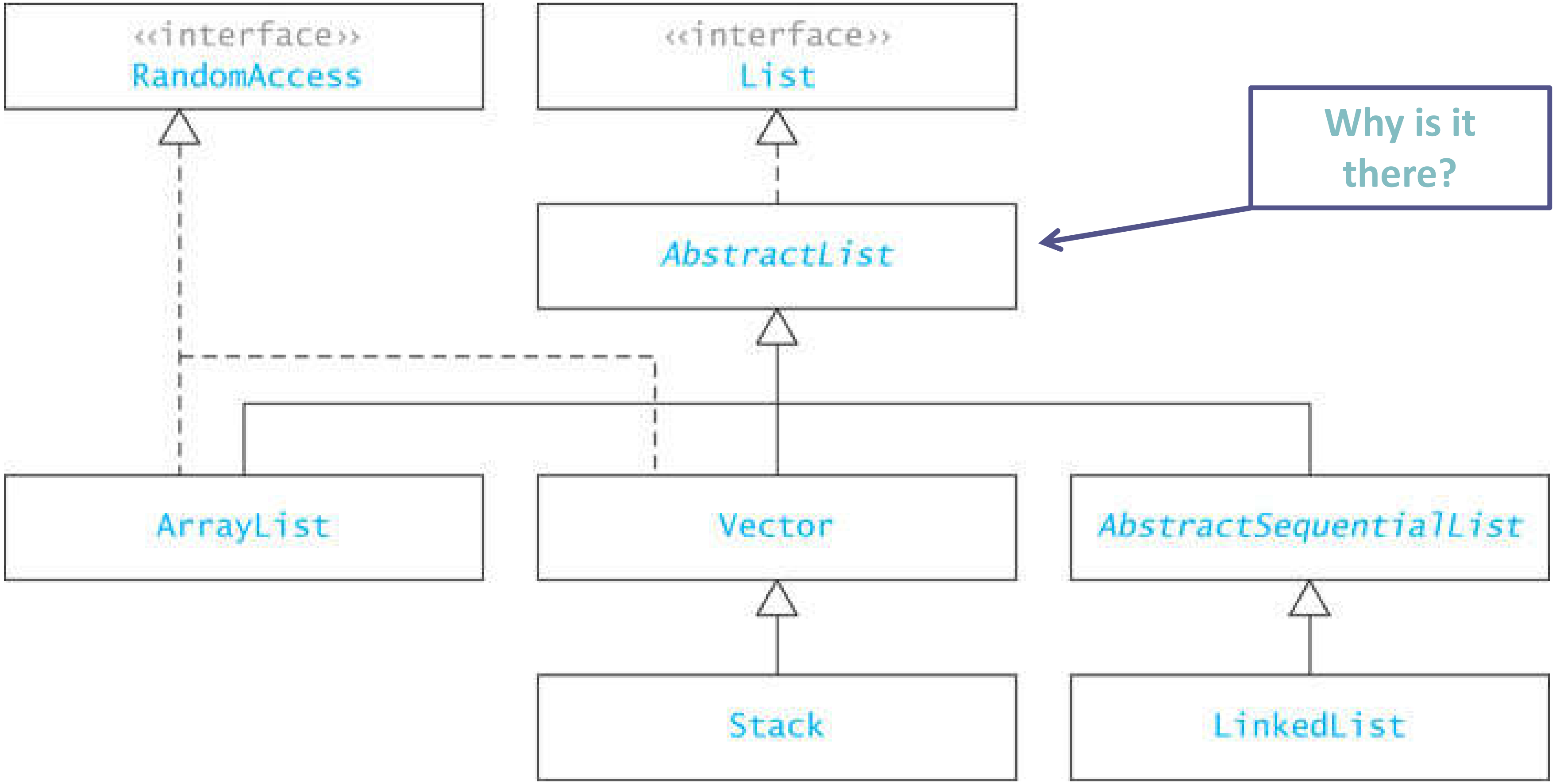
`List` **Interface and** `ArrayList` **Class**

- An array is an indexed structure
- In an indexed structure,
  - elements may be accessed in any order using subscript values
  - elements can be accessed in sequence using a loop that increments the subscript
- With the Java `Array` object, you cannot
  - increase or decrease its length (length is fixed)
  - add an element at a specified position without shifting elements to make room
  - remove an element at a specified position and keep the elements contiguous without shifting elements to fill in the gap

`List` Interface and `ArrayList` Class (cont.)

- Java provides a `List` interface as part of its API `java.util`
- Classes that implement the `List` interface provide the functionality of an indexed data structure and offer many more operations
- A sample of the operations:
  - Obtain an element at a specified position
  - Replace an element at a specified position
  - Find a specified target value
  - Add an element at either end
  - Remove an element from either end
  - Insert or remove an element at any position
  - Traverse the list structure without managing a subscript
- All classes introduced in this chapter support these operations, but they do not support them with the same degree of efficiency

`java.util.List` Interface and its Implementers



`List` Interface and `ArrayList` Class

- Unlike the `Array` data structure, classes that implement the `List` interface cannot store primitive types
- Classes must store values as objects
- This requires you to wrap primitive types, such an `int` and `double` in object wrappers, in these cases, `Integer` and `Double`

`ArrayList` **Class**

- The simplest class that implements the List interface
- An improvement over an array object
- Use when:
  - you will be adding new elements to the end of a list
  - you need to access elements quickly in any order

`ArrayList` Class (cont.)

- To declare a `List` “object” whose elements will reference `String` objects:

```
List<String> myList = new ArrayList<String>();
```

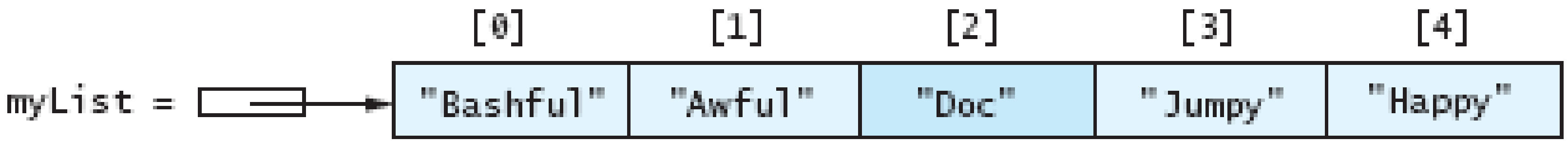
- The initial List is empty and has a default initial capacity of 10 elements
- To add strings to the list,

```
myList.add("Bashful");
myList.add("Awful");
myList.add("Jumpy");
myList.add("Happy");
```

`ArrayList` **Class** (cont.)

- Adding an element with subscript 2:

```
myList.add(2, "Doc");
```



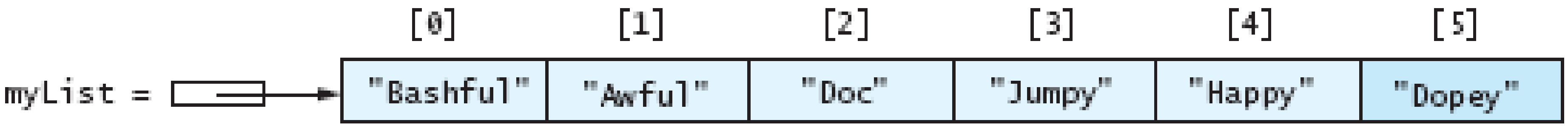
After insertion of "Doc" before the third element

- Notice that the subscripts of `"Jumpy"` and `"Happy"` have changed from [2],[3] to [3],[4]

ArrayList **Class** (cont.)

- When no subscript is specified, an element is added at the end of the list:

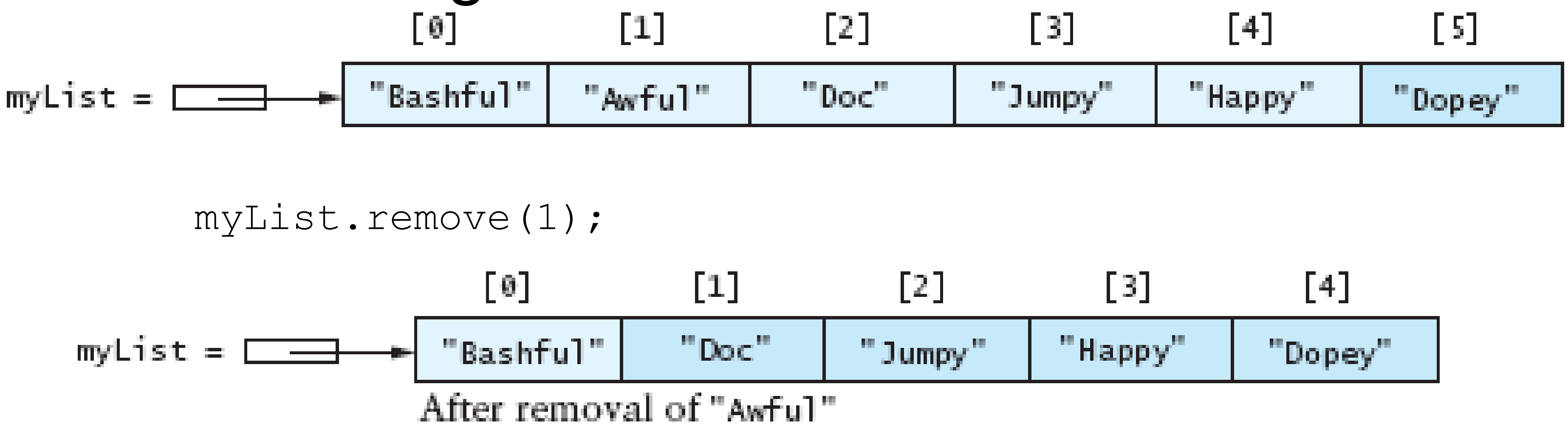
```
myList.add("Dopey");
```



After insertion of "Dopey" at the end

`ArrayList` Class (cont.)

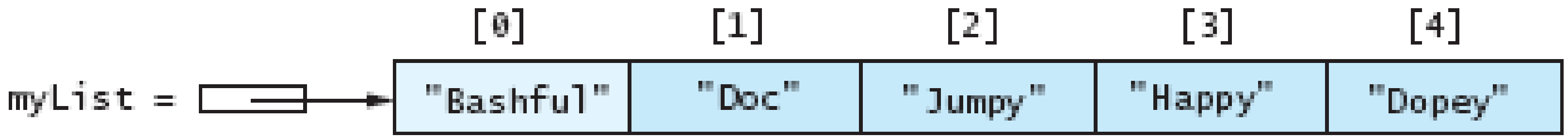
- Removing an element:



- The strings referenced by [2] to [5] have changed to [1] to [4]

`ArrayList` **Class** (cont.)

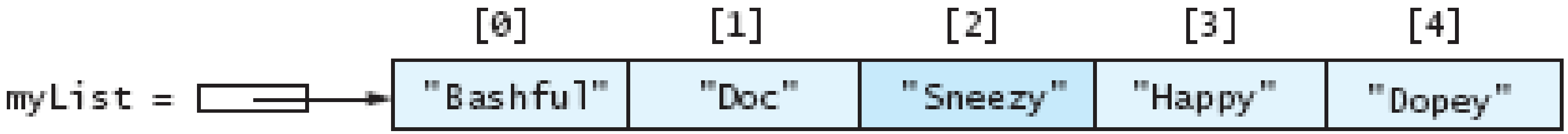
- You may also replace an element:




```
myList.set(2, "Sneezy");
```

After replacing "Jumpy" with "Sneezy"

`ArrayList` Class (cont.)

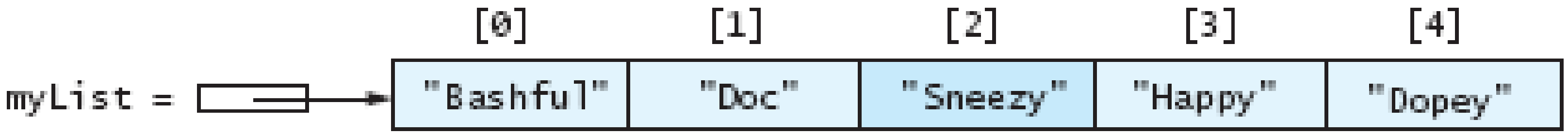


- You cannot access an element using a bracket index as you can with arrays (`array[1]`)
- Instead, you must use the `get()` method:

```
String dwarf = myList.get(2);
```

- The value of `dwarf` becomes `"Sneezy"`

`ArrayList` **Class** (cont.)



- You can also search an `ArrayList`:

```
myList.indexOf("Sneezy");
```

- This returns `2` while

```
myList.indexOf("Jumpy");
```

- returns `-1` which indicates an unsuccessful search

Generic Collections

- The statement

```
List<String> myList = new
   ArrayList<String>();
```

  uses a language feature called *generic collections* or *generics*
- The statement creates a `List` of `String`; only references of type `String` can be stored in the list
- `String` in this statement is called a *type parameter*
- The type parameter sets the data type of all objects stored in a collection

Generic Collections (cont.)

- The general declaration for generic collection is

```
            CollectionClassName<E> variable =
                          new CollectionClassName<E>();
```

- The `<E>` indicates a type parameter
- Adding a noncompatible type to a generic collection will generate an error during compile time
- However, primitive types will be autoboxed:

```
ArrayList<Integer> myList = new ArrayList<Integer>();
myList.add(new Integer(3)); // ok
myList.add(3); // also ok! 3 is automatically wrapped
                  in an Integer object
myList.add(new String("Hello")); // generates a type
                                   incompatability error
```

Why Use Generic Collections?

- Better type-checking: catch more errors, catch them earlier
- Documents intent
- Avoids the need to downcast from `Object`

Specification of the `ArrayList` Class

Method	Behavior
`public E get(int index)`	Returns a reference to the element at position `index`.
`public E set(int index, E anEntry)`	Sets the element at position `index` to reference `anEntry`. Returns the previous value.
`public int size()`	Gets the current size of the `ArrayList`.
`public boolean add(E anEntry)`	Adds a reference to `anEntry` at the end of the `ArrayList`. Always returns `true`.
`public void add(int index, E anEntry)`	Adds a reference to `anEntry`, inserting it before the item at position `index`.
`int indexOf(E target)`	Searches for `target` and returns the position of the first occurrence, or −1 if it is not in the `ArrayList`.
`public E remove(int index)`	Returns and removes the item at position `index` and shifts the items that follow it to fill the vacated space.

Applications of `ArrayList`

Section 2.2

Example Application of `ArrayList`

```
ArrayList<Integer> someInts = new ArrayList<Integer>();
int[] nums = {5, 7, 2, 15};
for (int i = 0; i < nums.length; i++) {
  someInts.add(nums[i]);
}

// Display the sum
int sum = 0;
for (int i = 0; i < someInts.size(); i++) {
  sum += someInts.get(i);
}
System.out.println("sum is " + sum);
```

Example Application of `ArrayList` (cont.)

```
ArrayList<Integer> someInts = new ArrayList<Integer>();
int[] nums = {5, 7, 2, 15};
for (int i = 0; i < nums.length; i++) {
  someInts.add(nums[i]);
}

// Display the sum
int sum = 0;
for (int i = 0; i < someInts
  sum += someInts.get(i);
}
System.out.println("sum is " + sum);
```

`nums[i]` is an `int`; it is automatically wrapped in an `Integer` object

Phone Directory Application

```
public class DirectoryEntry {
  String name;
  String number;
}
```

Create a class for objects stored in the directory

Phone Directory Application (cont.)

```
public class DirectoryEntry {
  String name;
  String number;
}

private ArrayList<DirectoryEntry> theDirectory =
          new ArrayList<DirectoryEntry>();
```

Create the directory

Phone Directory Application (cont.)

```
public class DirectoryEntry {
  String name;
  String number;
}
```

Add a `DirectoryEntry` object

```
private ArrayList<DirectoryEntry> theDirectory =
          new ArrayList<DirectoryEntry>();

theDirectory.add(new DirectoryEntry("Jane Smith",
                                    "555-1212"));
```

Phone Directory Application (cont.)

```
public class Direc
  String name;
  String number;
}

private ArrayList<DirectoryEntry> theDirectory =
          new ArrayList<DirectoryEntry>();

theDirectory.add(new DirectoryEntry("Jane Smith",
                                     "555-1212"));

int index = theDirectory.indexOf(new DirectoryEntry(aName,
                                                    ""));
```

Method `indexOf` searches `theDirectory` by applying the `equals` method for class `DirectoryEntry`. Assume `DirectoryEntry`'s `equals` method compares `name` fields.

Phone Directory Application (cont.)

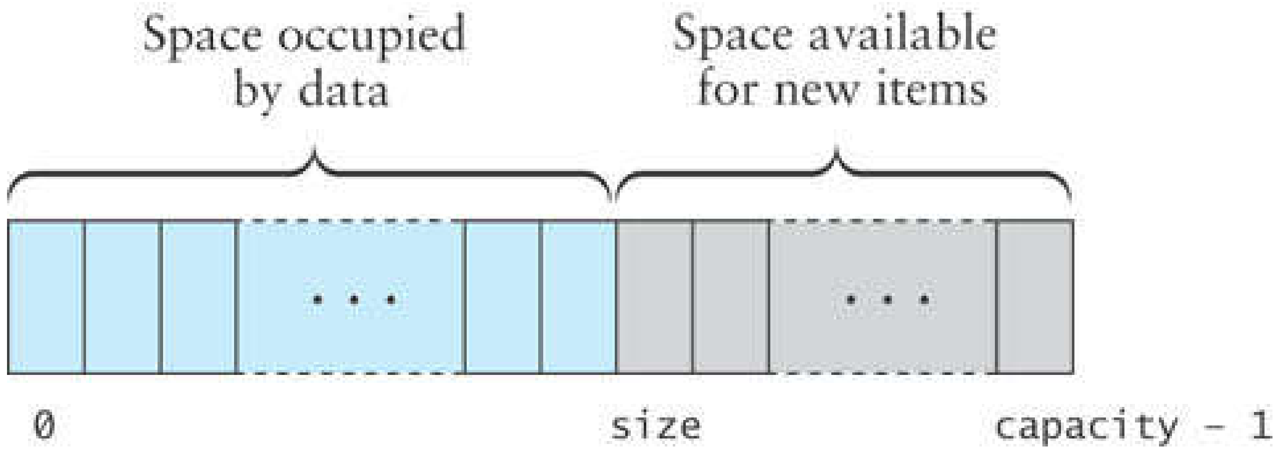
```
public class DirectoryEntry {
  String name;
  String number;
}

private ArrayList<DirectoryEntry> theDirectory =
          new ArrayList<DirectoryEntry>();

theDirectory.add(new DirectoryEntry("Jane Smith", "555-1212"));

int index = theDirectory.indexOf(new DirectoryEntry(aName, ""));

if (index != -1)
  dE = theDirectory.get(index);
else
  dE = null;
```

Implementation of an `ArrayList` Class

Section 2.3

Implementing an `ArrayList` Class

- `KWArrayList`: a simple implementation of `ArrayList`
  - Physical size of array indicated by data field *capacity*
  - Number of data items indicated by the data field *size*



`KWArrayList` Fields

```
import java.util.*;
/** This class implements some of the methods of the Java
ArrayList class
*/
public class KWArrayList<E> {
  // Data fields
  /** The default initial capacity */
  private static final int INITIAL_CAPACITY = 10;

  /** The underlying data array */
  private E[] theData;

  /** The current size */
  private int size = 0;

  /** The current capacity */
  private int capacity = 0;
}
```

`KWArrayList` Constructor

```
public KWArrayList () {
    capacity = INITIAL_CAPACITY;
    theData = (E[]) new Object[capacity];
}
```

This statement allocates storage for an array of type `Object` and then casts the array object to type E[]

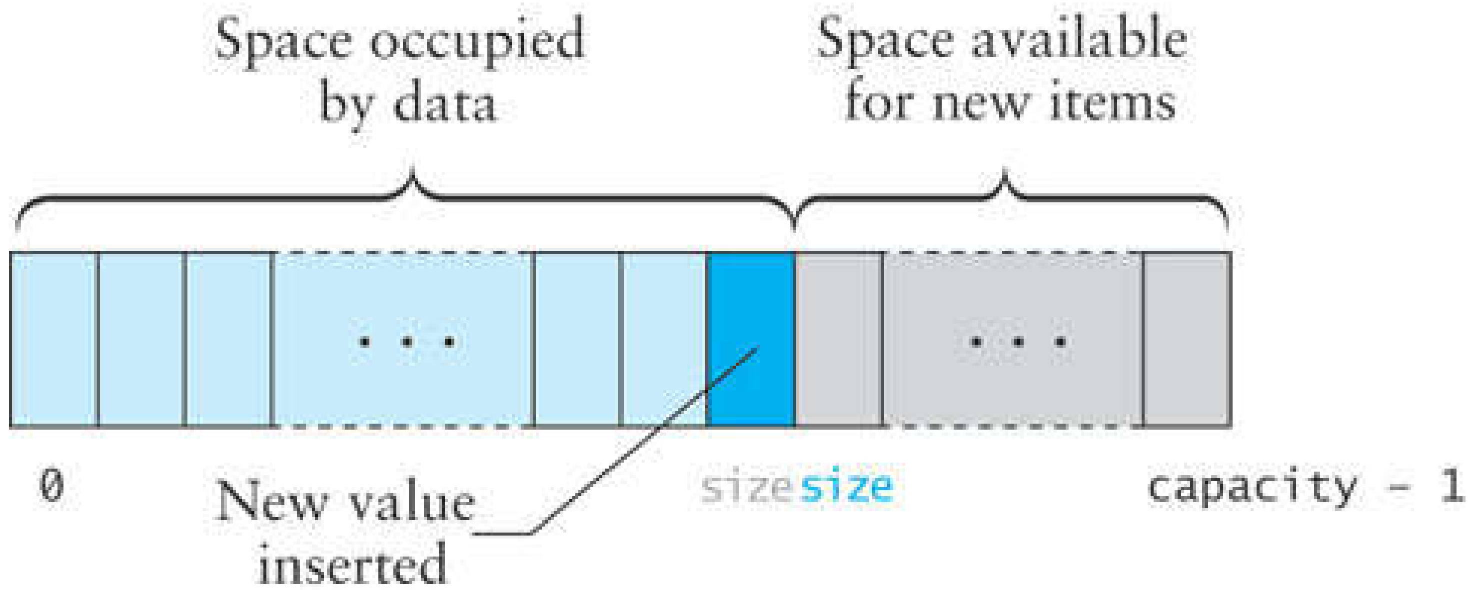
Although this may cause a compiler warning, it's ok

Implementing `ArrayList.add(E)`

- We will implement two add methods
- One will append at the end of the list
- The other will insert an item at a specified position

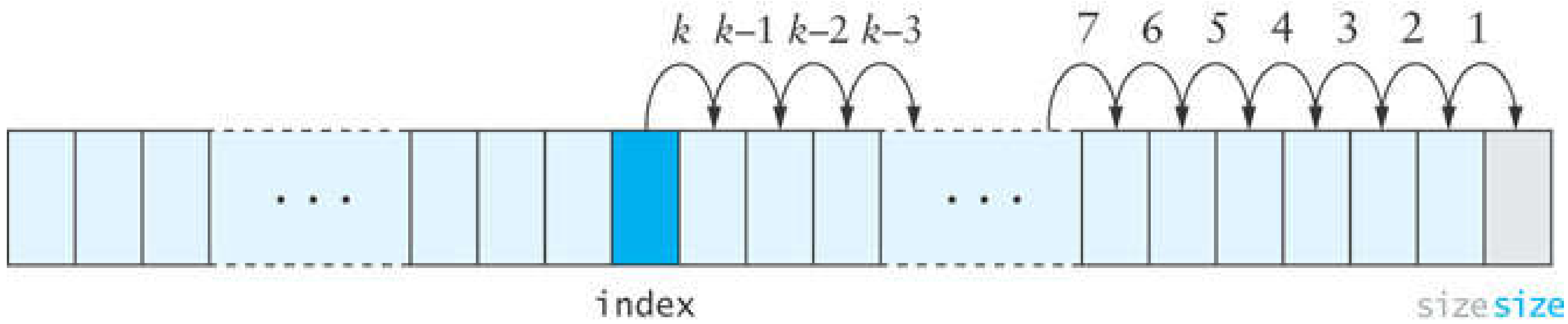
Implementing `ArrayList.add(E)` (cont.)

- If `size` is less than capacity, then to append a new item
  1. insert the new item at the position indicated by the value of `size`
  2. increment the value of `size`
  3. return `true` to indicate successful insertion



Implementing `ArrayList.add(int index,E anEntry)`

- To insert into the middle of the array, the values at the insertion point are shifted over to make room, beginning at the end of the array and proceeding in the indicated order



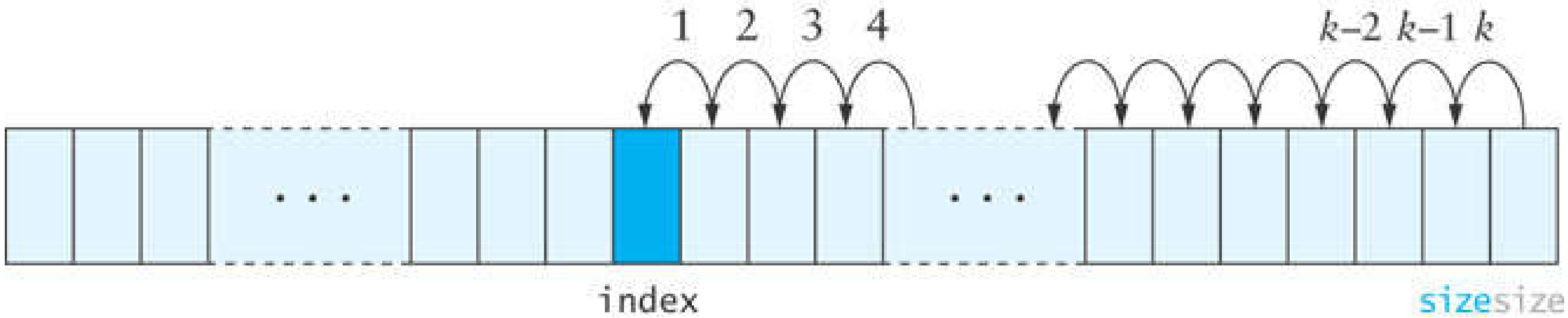
Implementing `ArrayList.add(index,E)`

```
public void add (int index, E anEntry) {

  // check bounds
  if (index < 0 || index > size) {
    throw new ArrayIndexOutOfBoundsException(index);
  }

  // Make sure there is room
  if (size >= capacity) {
    reallocate();
  }

  // shift data
  for (int i = size; i > index; i--) {
    theData[i] = theData[i-1];
  }

  // insert item
  theData[index] = anEntry;
  size++;
}
```

`set` and `get` Methods

```
public E get (int index) {
  if (index < 0 || index >= size) {
    throw new ArrayIndexOutOfBoundsException(index);
  }
  return theData[index];
}

public E set (int index, E newValue) {
  if (index < 0 || index >= size) {
    throw new ArrayIndexOutOfBoundsException(index);
  }
  E oldValue = theData[index];
  theData[index] = newValue;
  return oldValue;
}
```

remove Method



- When an item is removed, the items that follow it must be moved forward to close the gap
- Begin with the item closest to the removed element and proceed in the indicated order

`remove` **Method** (cont.)

```
public E remove (int index) {

  if (index < 0 || index >= size) {
    throw new ArrayIndexOutOfBoundsException(index);
  }

  E returnValue = theData[index];

  for (int i = index + 1; i < size; i++) {
    theData[i-1] = theData[i];
  }

  size--;
  return returnValue;
}
```

`reallocate` Method

- Create a new array that is twice the size of the current array and then copy the contents of the new array

```
private void reallocate () {
  capacity *= 2;
  theData = Arrays.copyOf(theData, capacity);
}
```

`reallocate` Method (cont.)

```
private void reallocate () {
  capacity *= 2;
  theData = Arrays.copyOf(theData, capacity);
}
```

The reason for doubling is to spread out the cost of copying; we discuss this further in the next section

`KWArrayList` as a Collection of Objects

- Earlier versions of Java did not support generics; all collections contained only `Object` elements
- To implement `KWArrayList` this way,
  - remove the parameter type `<E>` from the class heading,
  - replace each reference to data type `E` by `Object`
  - The underlying data array becomes

```
private Object[] theData;
```

`Vector` **Class**

- The Java API `java.util` contains two very similar classes, `Vector` and `ArrayList`
- New applications normally use `ArrayList` rather than `Vector` as `ArrayList` is generally more efficient
- `Vector` class is *synchronized*, which means that multiple threads can access a `Vector` object without conflict

Algorithm Efficiency and Big-O

Section 2.4

Algorithm Efficiency and Big-O

- Getting a precise measure of the performance of an algorithm is difficult
- Big-O notation expresses the performance of an algorithm as a function of the number of items to be processed
- This permits algorithms to be compared for efficiency
- For more than a certain number of data items, some problems cannot be solved by any computer

Linear Growth Rate

- If processing time increases in proportion to the number of inputs *n*, the algorithm grows at a linear rate

```
public static int search(int[] x, int target) {
  for(int i=0; i < x.length; i++) {
    if (x[i]==target)
      return i;
  }
  return -1; // target not found
}
```

Linear Gro

- If the target is not present, the `for` loop will execute `x.length` times
- If the target is present the `for` loop will execute (on average) `(x.length + 1)/2` times
- Therefore, the total execution time is directly proportional to `x.length`
- This is described as a growth rate of order *n* OR
- O(n)

□ If processing ti
the number of
a linear rate

```
public static int search(int[] x, int target) {
  for(int i=0; i < x.length; i++) {
    if (x[i]==target)
      return i;
  }
  return -1; // target not found
}
```

n x m Growth Rate

- Processing time can be dependent on two different inputs

```
public static boolean areDifferent(int[] x, int[] y) {
  for(int i=0; i < x.length; i++) {
    if (search(y, x[i]) != -1)
      return false;
  }
  return true;
}
```

n x m Growth Rate (cont.)

- Processing tim
different inputs

- The `for` loop will execute `x.length` times
- But it will call `search`, which will execute `y.length` times
- The total execution time is proportional to (`x.length * y.length`)
- The growth rate has an order of n x m or
- O(n x m)

```
public static boolean areDifferent(int[] x, int[] y) {
  for(int i=0; i < x.length; i++) {
    if (search(y, x[i]) != -1)
      return false;
  }
  return true;
}
```

Quadratic Growth Rate

- If processing time is proportional to the square of the number of inputs *n*, the algorithm grows at a quadratic rate

```
public static boolean areUnique(int[] x) {
  for(int i=0; i < x.length; i++) {
    for(int j=0; j < x.length; j++) {
      if (i != j && x[i] == x[j])
        return false;
    }
  }
  return true;
}
```

Quadratic Growth Rate (cont.)

- If processing time … number of inputs … rate

- The `for` loop with `i` as index will execute `x.length` times
- The `for` loop with `j` as index will execute `x.length` times
- The total number of times the inner loop will execute is $(\texttt{x.length})^2$
- The growth rate has an order of n^2 or
- $O(n^2)$

```
public static boolean areUnique(int[] x) {
  for(int i=0; i < x.length; i++) {
    for(int j=0; j < x.length; j++) {
      if (i != j && x[i] == x[j])
        return false;
    }
  }
  return true;
}
```

Big-O Notation

- The O() in the previous examples can be thought of as an abbreviation of "order of magnitude"
- A simple way to determine the big-O notation of an algorithm is to look at the loops and to see whether the loops are nested
- Assuming a loop body consists only of simple statements,
  - a single loop is O(n)
  - a pair of nested loops is $O(n^2)$
  - a nested pair of loops inside another is $O(n^3)$
  - and so on . . .

Big-O Notation (cont.)

- You must also examine the *number of times* a loop is executed

```
for(i=1; i < x.length; i *= 2) {
   // Do something with x[i]
}
```

- The loop body will execute k-1 times, with `i` having the following values:

  1, 2, 4, 8, 16, . . ., 2^k

  until 2^k is greater than `x.length`
- Since 2^{k-1} = `x.length` < 2^k and $\log_2 2^k$ is k, we know that k-1 = $\log_2$(`x.length`) < k
- Thus we say the loop is O(log n) (in analyzing algorithms, we use logarithms to the base 2)
- Logarithmic functions grow slowly as the number of data items n increases

Formal Definition of Big-O

- Consider the following program structure:

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
     Simple Statement
  }
}
for (int i = 0; i < n; i++) {
   Simple Statement 1
   Simple Statement 2
   Simple Statement 3
   Simple Statement 4
   Simple Statement 5
}
Simple Statement 6
Simple Statement 7
. . .
Simple Statement 30
```

Formal Definition of Big-O (cont.)

- Consider the following program structure:

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
      Simple Statement
  }
}
for (int i = 0; i < n; i++) {
   Simple Statement 1
   Simple Statement 2
   Simple Statement 3
   Simple Statement 4
   Simple Statement 5
}
Simple Statement 6
Simple Statement 7
...
Simple Statement 30
```

This nested loop executes a *Simple Statement* n^2 times

Formal Definition of Big-O (cont.)

- Consider the following program structure:

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
     Simple Statement
  }
}
for (int i = 0; i < n; i++) {
   Simple Statement  1
   Simple Statement 2
   Simple Statement 3
   Simple Statement 4
   Simple Statement 5
}
Simple Statement 6
Simple Statement 7
. . .
Simple Statement 30
```

This loop executes 5 *Simple Statements n* times (5*n*)

Formal Definition of Big-O (cont.)

- Consider the following program structure:

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
      Simple Statement
  }
}
for (int i = 0; i < n; i++) {
   Simple Statement 1
   Simple Statement 2
   Simple Statement 3
   Simple Statement 4
   Simple Statement 5
}
Simple Statement 6
Simple Statement 7
. . .
Simple Statement 30
```

Finally, 25 *Simple Statements* are executed

Formal Definition of Big-O (cont.)

- Consider the following program structure:

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
      Simple Statement
  }
}
for (int i = 0; i < n; i++) {
   Simple Statement 1
   Simple Statement 2
   Simple Statement 3
   Simple Statement 4
   Simple Statement 5
}
Simple Statement 6
Simple Statement 7
. . .
Simple Statement 30
```

We can conclude that the relationship between processing time and *n* (the number of date items processed) is:

$$T(n) = n^2 + 5n + 25$$

Formal Definition of Big-O (cont.)

- In terms of T(n),

$$T(n) = O(f(n))$$

- There exist
  - two constants, n_0 and c, greater than zero, and
  - a function, f(n),
- such that for all $n > n_0$, cf(n) >= T(n)
- In other words, as n gets sufficiently large (larger than n_0), there is some constant c for which the processing time will always be less than or equal to cf(n)
- cf(n) is an upper bound on performance

Formal Definition of Big-O (cont.)

- The growth rate of f(n) will be determined by the fastest growing term, which is the one with the largest exponent
- In the example, an algorithm of

$$O(n^2 + 5n + 25)$$

is more simply expressed as

$$O(n^2)$$

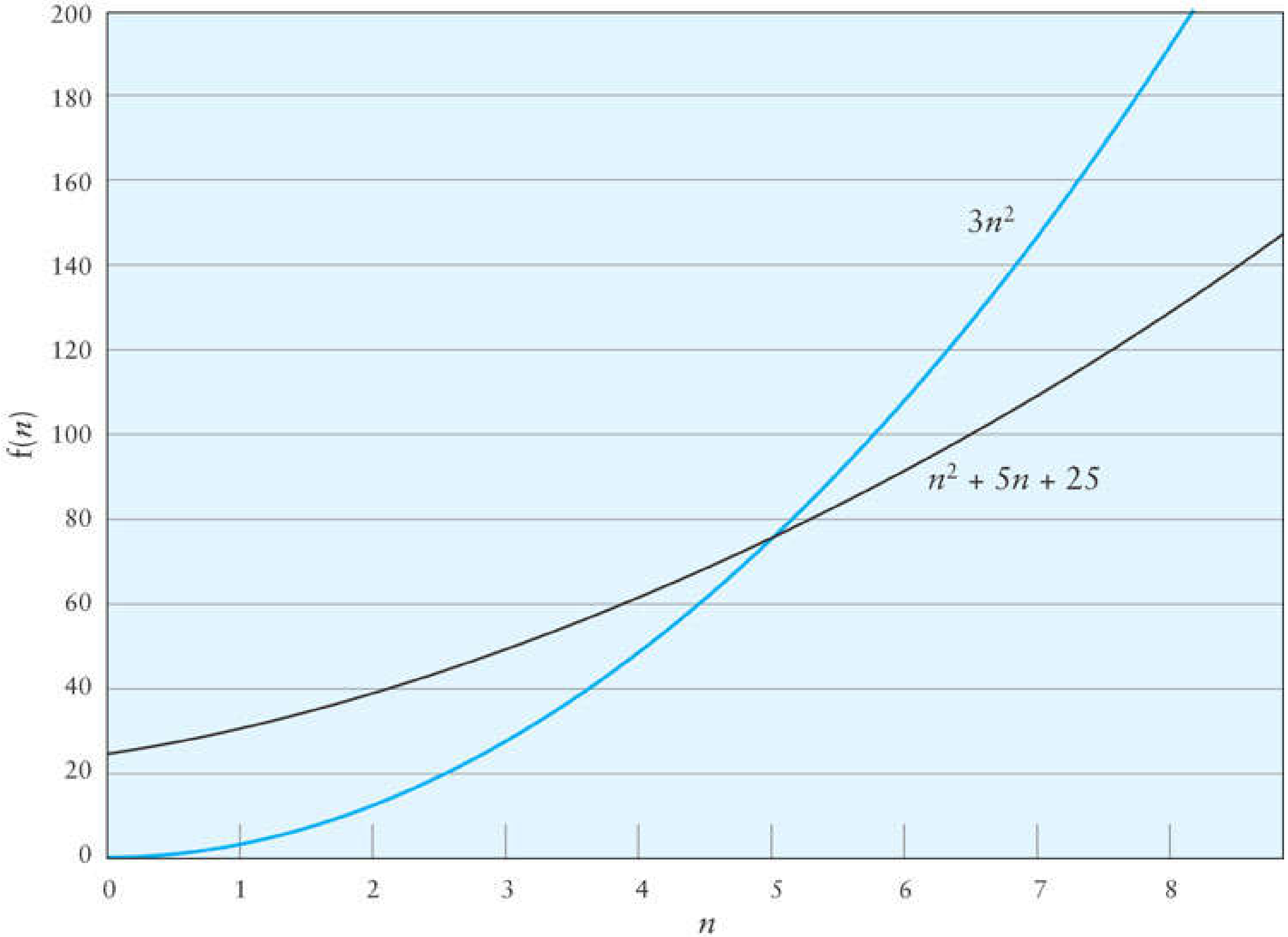
- In general, it is safe to ignore all constants and to drop the lower-order terms when determining the order of magnitude

Big-O Example 1

- Given $T(n) = n^2 + 5n + 25$, show that this is $O(n^2)$
- Find constants n_0 and c so that, for all $n > n_0$, $cn^2 > n^2 + 5n + 25$
  - Find the point where $cn^2 = n^2 + 5n + 25$
  - Let $n = n_0$, and solve for c

    $c = 1 + 5/\,n_0, + 25/\,n_0{}^2$
- When n_0 is 5, c = (1 + 5/5 + 25/25), c is 3
- So, $3n^2 > n^2 + 5n + 25$ for all $n > 5$
- Other values of n_0 and c also work

Big-O Example 1 (cont.)



Big-O Example 2

- Consider the following loop

```
for (int i = 0; i < n; i++) {
   for (int j = i + 1; j < n; j++) {
         3 simple statements
   }
}
```

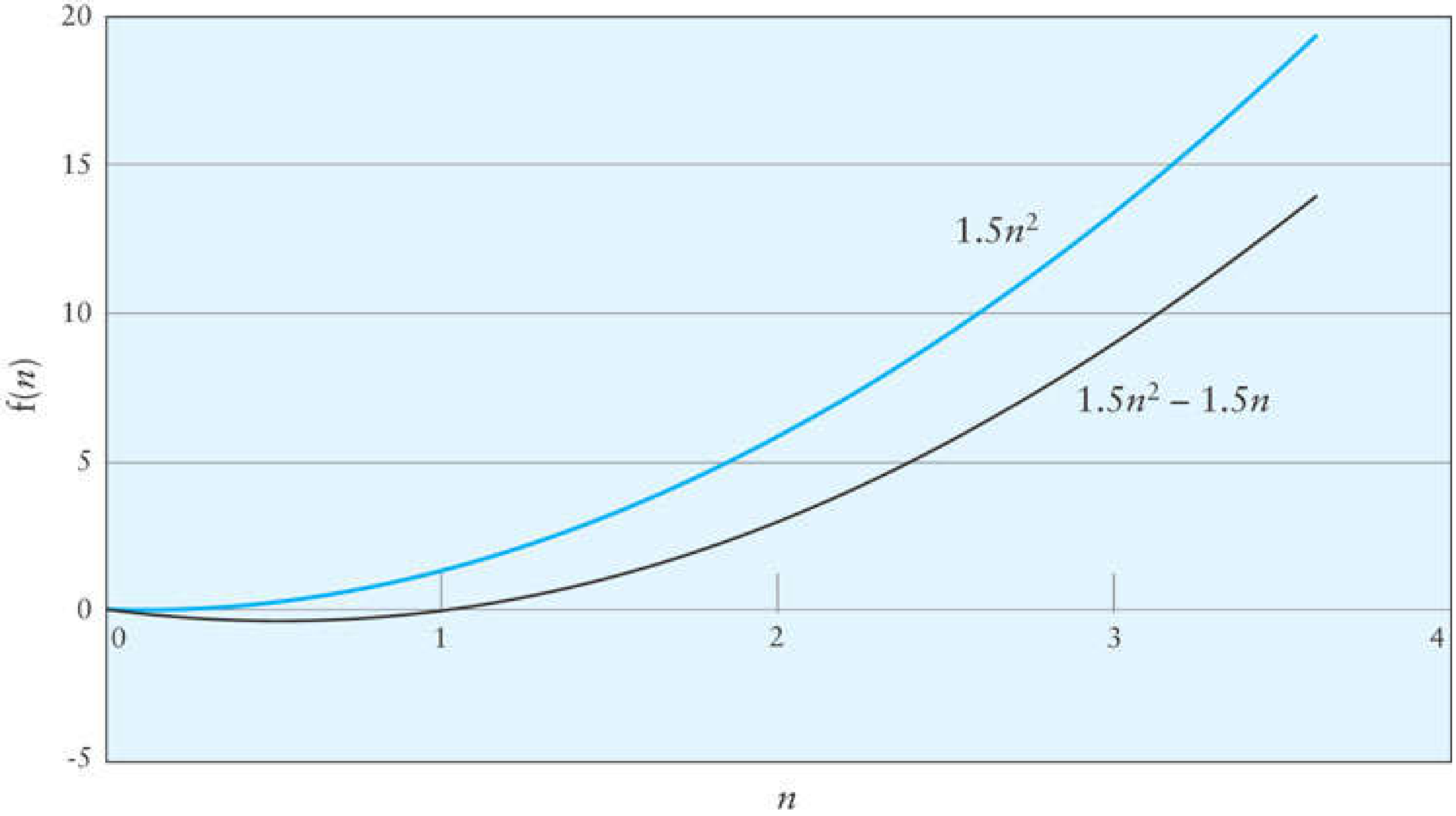
- $T(n) = 3(n - 1) + 3\,(n - 2) + \ldots + 3$
- Factoring out the 3,

  $3(n - 1 + n - 2 + n - 3 + \ldots + 1)$
- $1 + 2 + \ldots + n - 1 = (n \times (n\text{-}1))/2$

Big-O Example 2 (cont.)

- Therefore T(n) = $1.5n^2 - 1.5n$
- When $n = 0$, the polynomial has the value 0
- For values of $n > 1$, $1.5n^2 > 1.5n^2 - 1.5n$
- Therefore T(n) is O(n^2) when n_0 is 1 and c is 1.5

Big-O Example 2 (cont.)



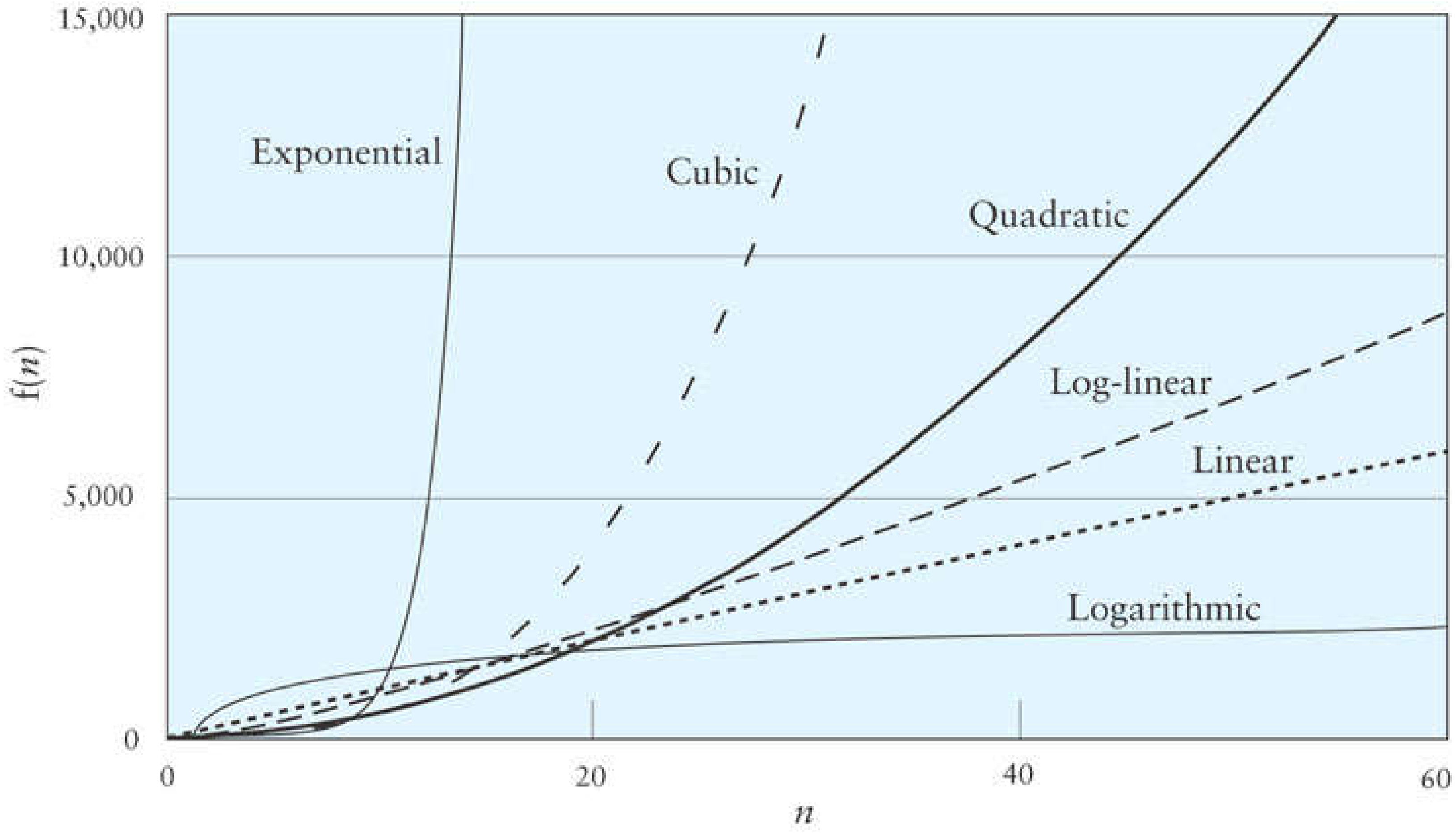
Symbols Used in Quantifying Performance

Symbol	Meaning
$T(n)$	The time that a method or program takes as a function of the number of inputs, n. We may not be able to measure or determine this exactly.
$f(n)$	Any function of n. Generally, $f(n)$ will represent a simpler function than $T(n)$, for example, n^2 rather than $1.5n^2 - 1.5n$.
$\mathbf{O}(f(n))$	Order of magnitude. $\mathbf{O}(f(n))$ is the set of functions that grow no faster than $f(n)$. We say that $T(n) = \mathbf{O}(f(n))$ to indicate that the growth of $T(n)$ is bounded by the growth of $f(n)$.

Common Growth Rates

Big-O	Name
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Log-linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential
$O(n!)$	Factorial

Different Growth Rates



Effects of Different Growth Rates

O(f(*n*))	f(50)	f(100)	f(100)/f(50)
O(1)	1	1	1
O(log *n*)	5.64	6.64	1.18
O(*n*)	50	100	2
O(*n* log *n*)	282	664	2.35
$O(n^2)$	2500	10,000	4
$O(n^3)$	12,500	100,000	8
$O(2^n)$	1.126×10^{15}	1.27×10^{30}	1.126×10^{15}
$O(n!)$	3.0×10^{64}	9.3×10^{157}	3.1×10^{93}

Algorithms with Exponential and Factorial Growth Rates

- Algorithms with exponential and factorial growth rates have an effective practical limit on the size of the problem they can be used to solve
- With an $O(2^n)$ algorithm, if 100 inputs takes an hour then,
  - 101 inputs will take 2 hours
  - 105 inputs will take 32 hours
  - 114 inputs will take 16,384 hours (almost 2 years!)

Algorithms with Exponential and Factorial Growth Rates (cont.)

- Encryption algorithms take advantage of this characteristic
- Some cryptographic algorithms can be broken in $O(2^n)$ time, where n is the number of bits in the key
- A key length of 40 is considered breakable by a modern computer,
- but a key length of 100 bits will take a billion-billion (10^{18}) times longer than a key length of 40

Performance of `KWArrayList`

- The `set` and `get` methods execute in constant time: O(1)
- Inserting or removing general elements is linear time: O(*n*)
- Adding at the end is (usually) constant time: O(1)
  - With our reallocation technique the average is O(1)
  - The worst case is O(n) because of reallocation

Single-Linked Lists

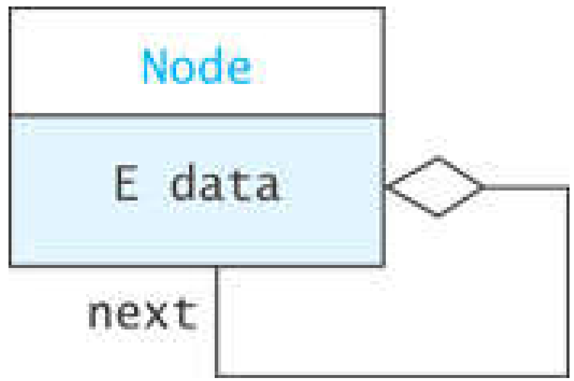
Section 2.5

Single-Linked Lists

- A linked list is useful for inserting and removing at arbitrary locations
- The `ArrayList` is limited because its `add` and `remove` methods operate in linear (O(n)) time—requiring a loop to shift elements
- A linked list can add and remove elements at a known location in O(1) time
- In a linked list, instead of an index, each element is linked to the following element

A List Node

- A node can contain:
  - a data item
  - one or more links
- A link is a reference to a list node
- In our structure, the node contains a data field named `data` of type `E`
- and a reference to the next node, named `next`



List Nodes for Single-Linked Lists

```
private static class Node<E> {
  private E data;
  private Node<E> next;

  /** Creates a new node with a null next field
      @param dataItem  The data stored
  */
  private Node(E dataItem) {
    data = dataItem;
    next = null;
  }

 /** Creates a new node that references another node
      @param dataItem  The data stored
      @param nodeRef  The node referenced by new node
  */
  private Node(E dataItem, Node<E> nodeRef) {
    data = dataItem;
    next = nodeRef;
  }
}
```

List Nodes for Single-Linked Lists (cont.)

```
private static class Node<E> {
  private E data;
  private Node<E> next;

  /** Creates a new node with a null next
      @param dataItem  The data stored
  */
  private Node(E data) {
    data = dataItem;
    next = null;
  }

 /** Creates a new node that references another node
      @param dataItem  The data stored
      @param nodeRef  The node referenced by new node
  */
  private Node(E dataItem, Node<E> nodeRef) {
    data = dataItem;
    next = nodeRef;
  }
}
```

The keyword `static` indicates that the `Node<E>` class will not reference its outer class

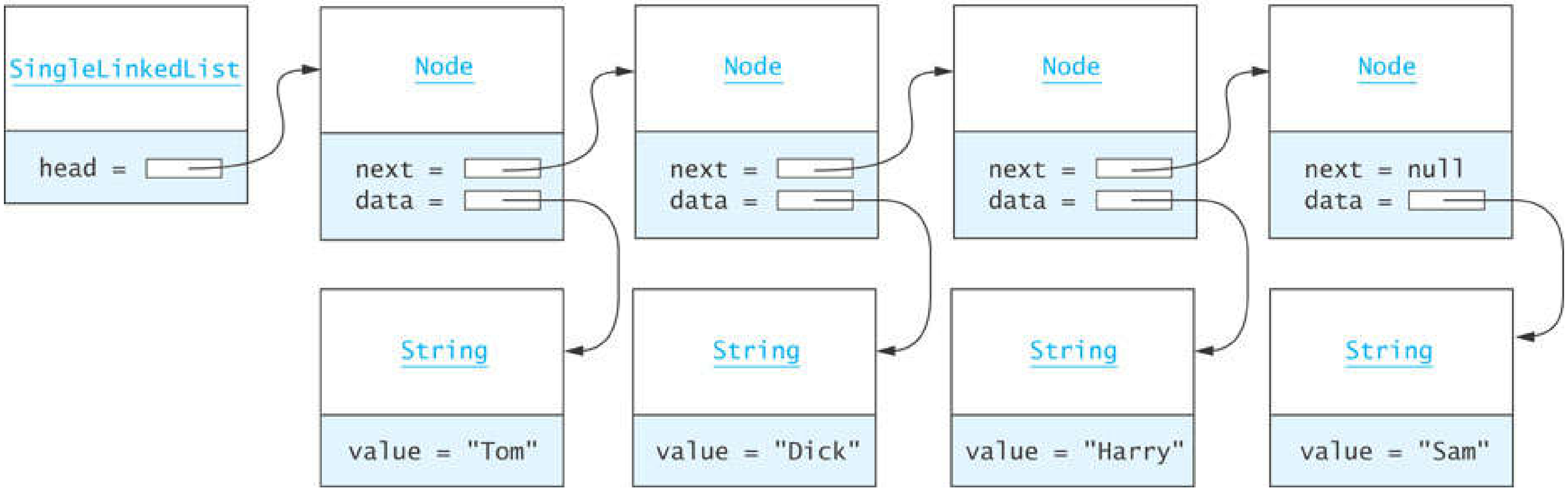
Static inner classes are also called *nested classes*

List Nodes for Single-Linked Lists (cont.)

```java
private static class Node<E> {
  private E data;
  private Node<E> next;

  /** Creates a new node with a null next field
      @param dataItem  The data stored
  */
  private Node(E dataItem) {
    data = dataItem;
    next = null;
  }

 /** Creates a new node that references another node
      @param dataItem  The data stored
      @param nodeRef  The node referenced by new node
  */
  private Node(E dataItem, Node<E> nodeRef) {
    data = dataItem;
    next = nodeRef;
  }
}
```

Generally, all details of the Node class should be private. This applies also to the data fields and constructors.

Connecting Nodes



Connecting Nodes (cont.)

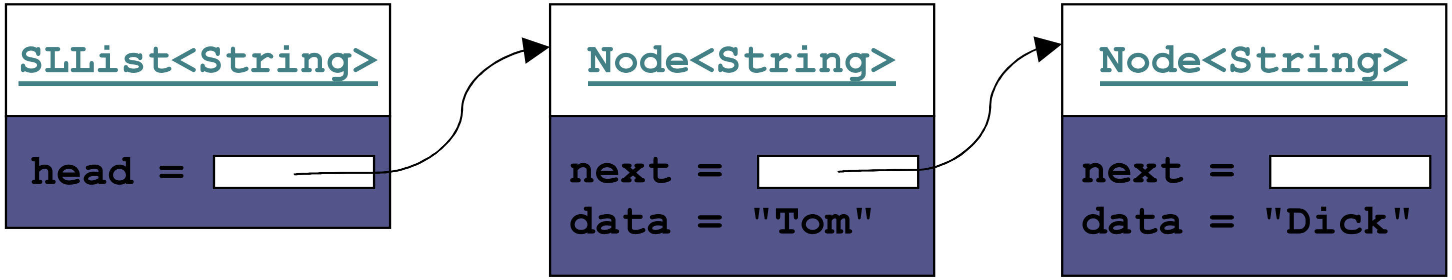
```
Node<String> tom = new Node<String>("Tom");
Node<String> dick = new Node<String>("Dick");
Node<String> harry = new Node<String>("Harry");
Node<String> sam = new Node<String>("Sam");

tom.next = dick;
dick.next = harry;
harry.next = sam;
```

A Single-Linked List Class

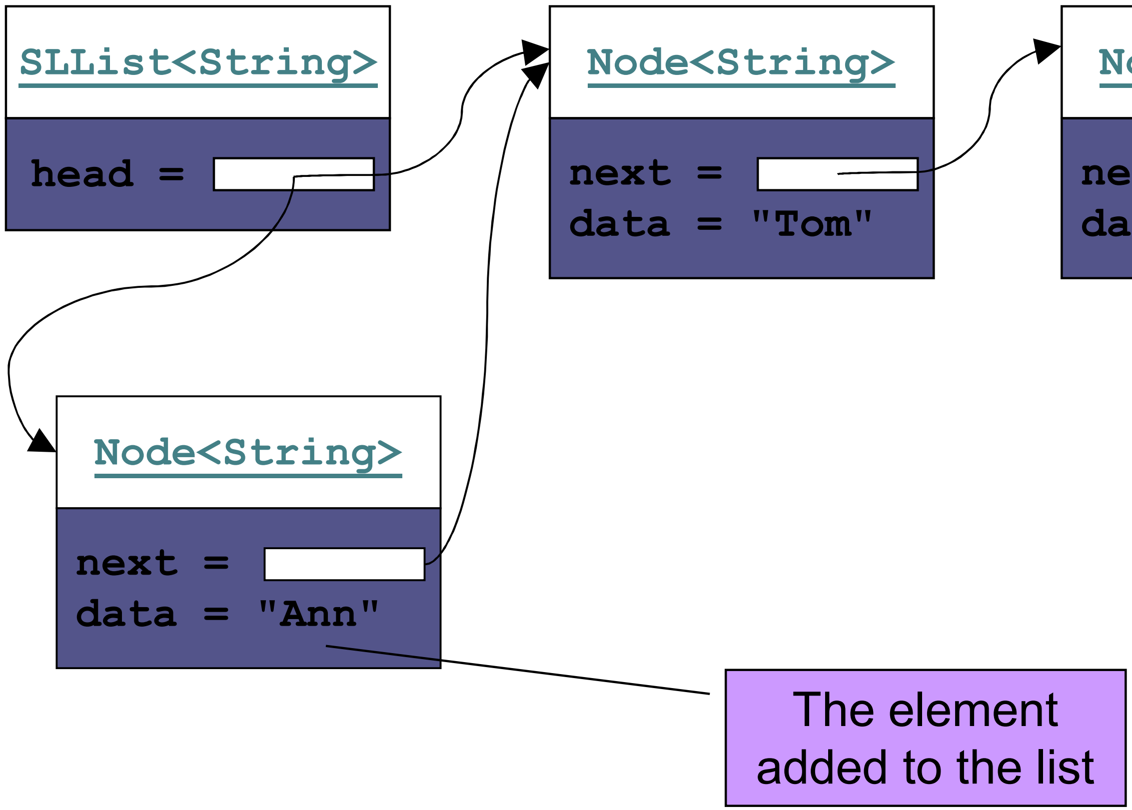
- Generally, we do not have individual references to each node.
- A `SingleLinkedList` object has a data field `head`, the *list head*, which references the first list node

```
public class SingleLinkedList<E> {
  private Node<E> head = null;
  private int size = 0;
  ...
}
```

`SLList`: An Example List



Implementing `SLList.addFirst(E item)`



Implementing `SLList.addFirst(E item)` (cont.)

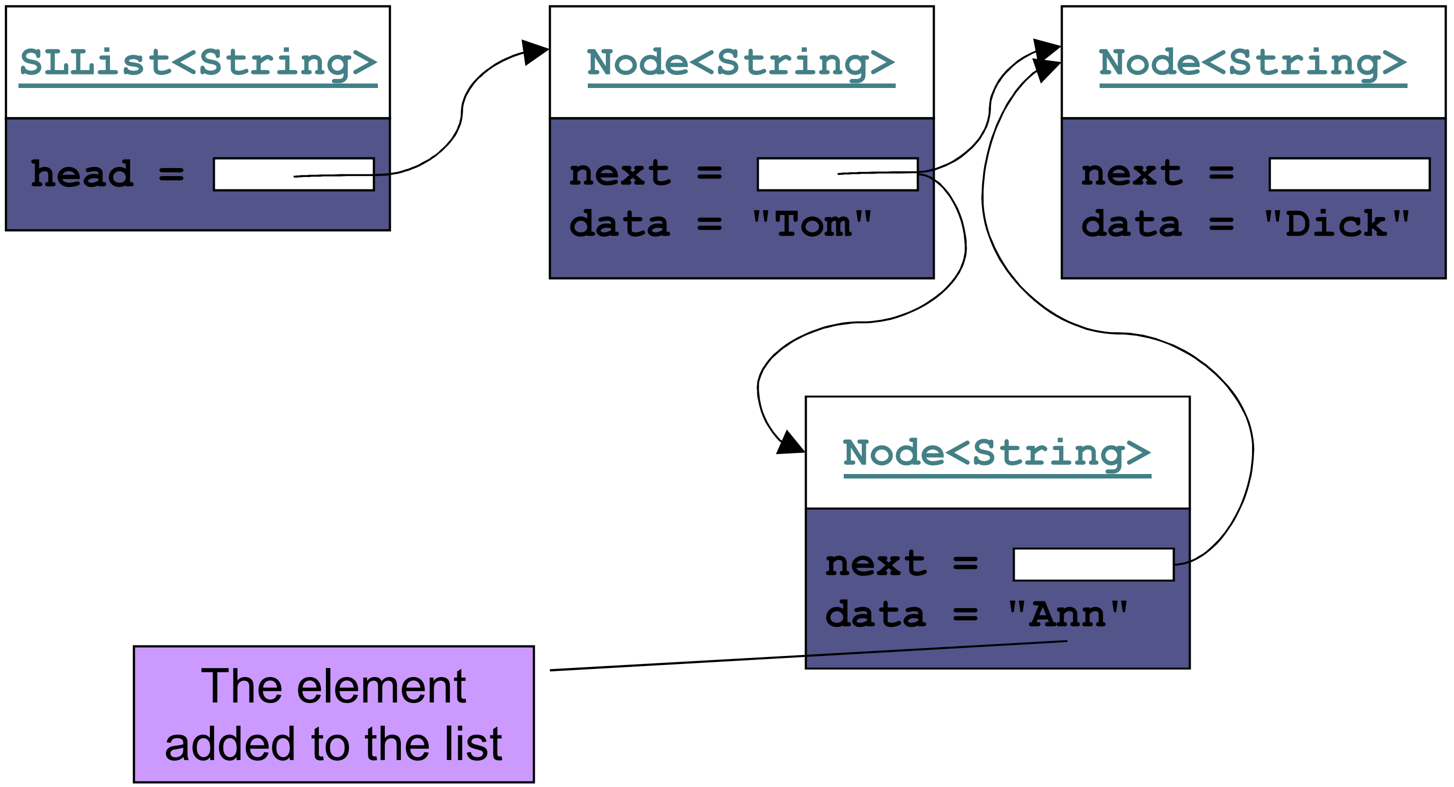
```
private void addFirst (E item) {
  Node<E> temp = new Node<E>(item, head);
  head = temp;
  size++;
}
```

or, more simply ...

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

This works even if `head` is null

Implementing `addAfter(Node<E> node, E item)`



Implementing `addAfter(Node<E> node, E item)` (cont.)

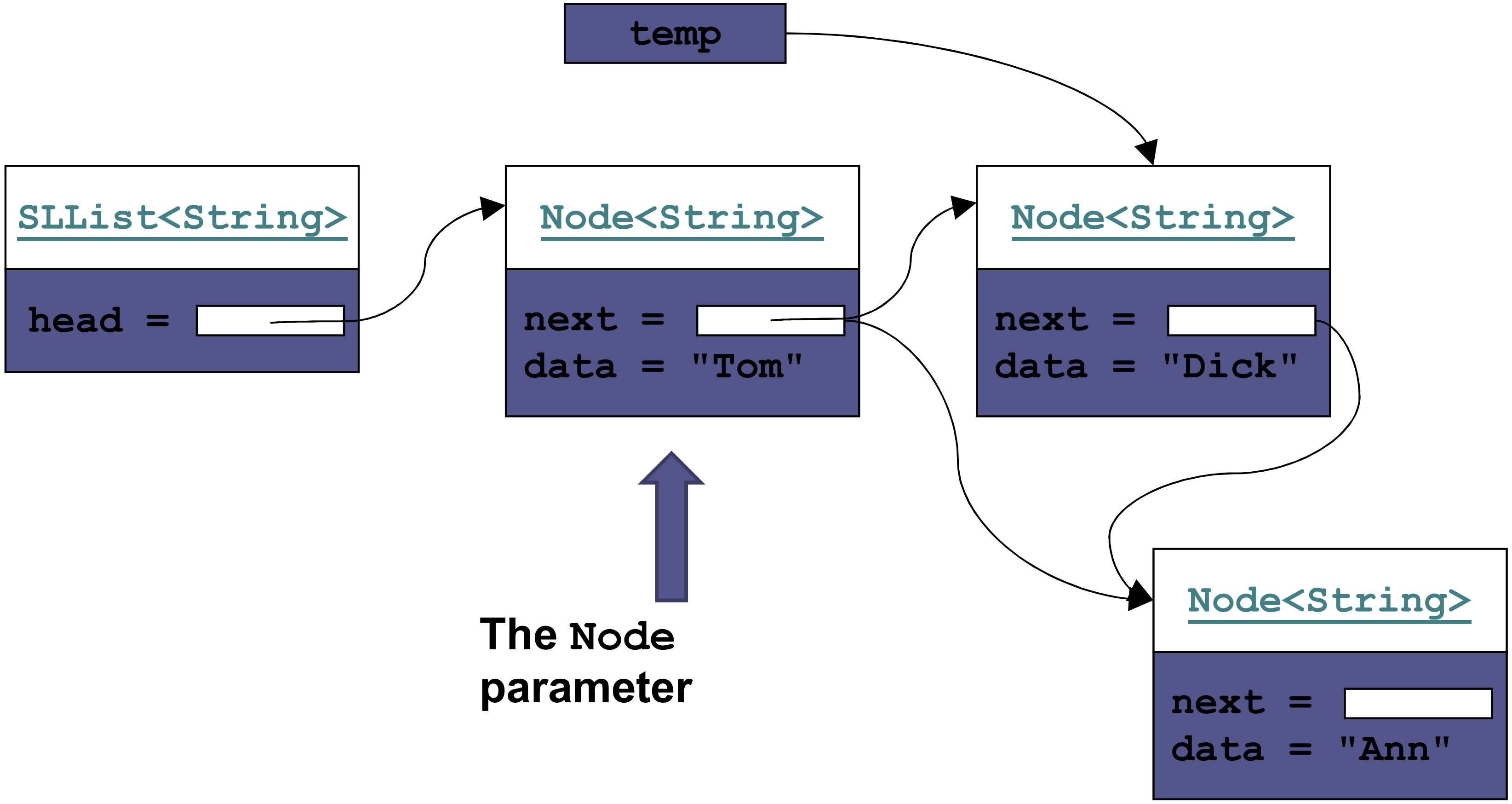
```
private void addAfter (Node<E> node, E item) {
  Node<E> temp = new Node<E>(item, node.next);
  node.next = temp;
  size++;
}
```

We declare this method **`private`** since it should not be called from outside the class. Later we will see how this method is used to implement the public add methods.

or, more simply ...

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

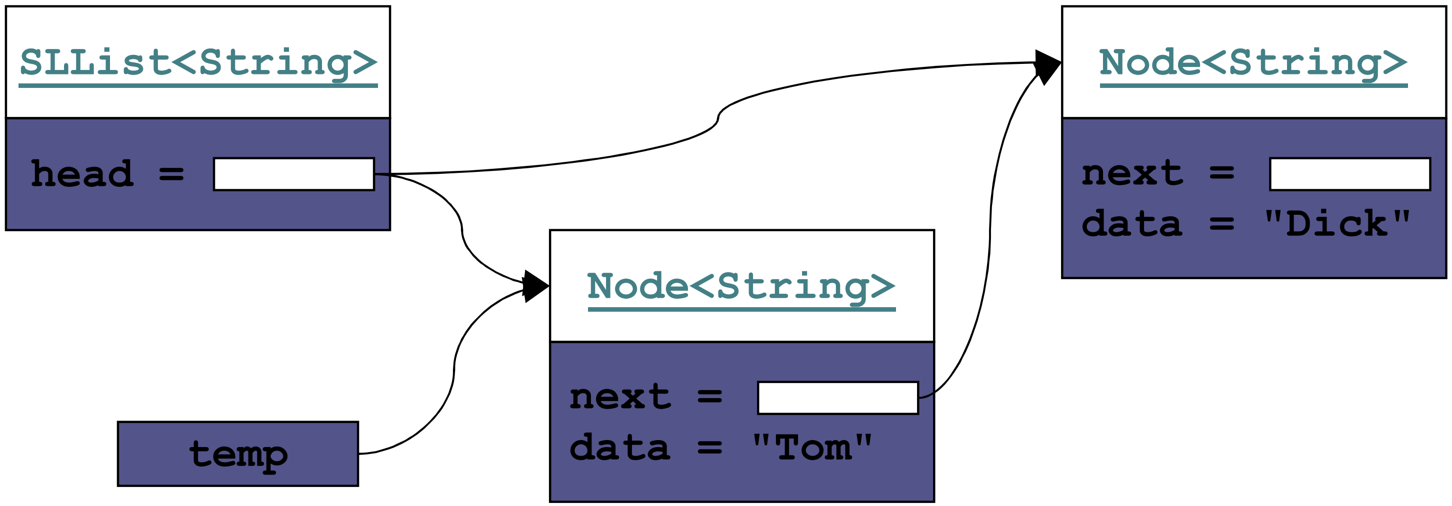
Implementing `removeAfter(Node<E> node)`



Implementing `removeAfter(Node<E> node)` (cont.)

```
private E removeAfter (Node<E> node) {
  Node<E> temp = node.next;
  if (temp != null) {
    node.next = temp.next;
    size--;
    return temp.data;
  } else {
    return null;
  }
}
```

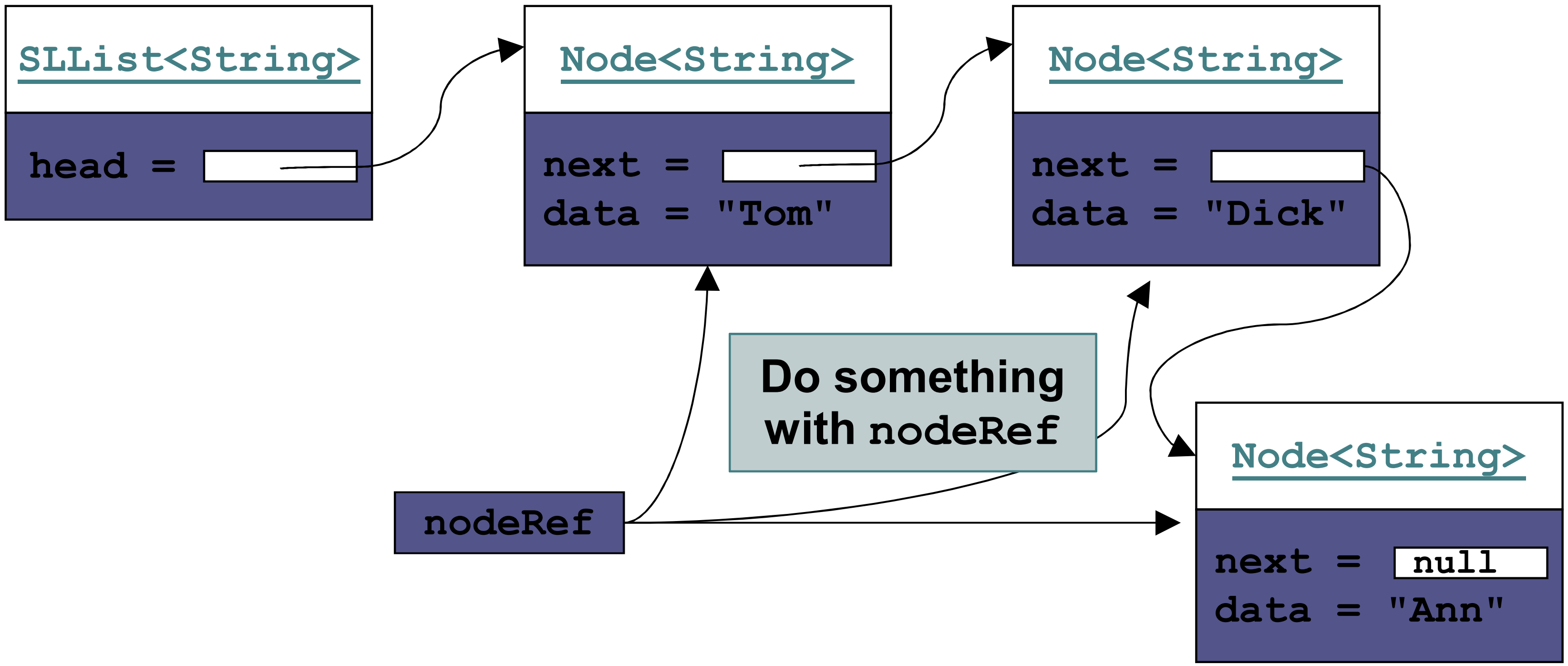
Implementing `SLList.removeFirst()`



Implementing `SLList.removeFirst()` (cont.)

```
private E removeFirst () {
  Node<E> temp = head;
  if (head != null) {
    head = head.next;
  }
  if (temp != null) {
    size--;
    return temp.data;
  } else {
    return null;
  }
}
```

Traversing a Single-Linked List



Traversing a Single-Linked List (cont.)

- `toString()` can be implemented with a traversal:

```
public String toString() {
  Node<String> nodeRef = head;
  StringBuilder result = new StringBuilder();
  while (nodeRef != null) {
    result.append(nodeRef.data);
    if (nodeRef.next != null) {
      result.append(" ==> ");
    }
    nodeRef = nodeRef.next;
  }
  return result.toString();
}
```

SLList.getNode(int)

- In order to implement methods required by the List interface, we need an additional helper method:

```
private Node<E> getNode(int index) {
  Node<E> node = head;
  for (int i=0; i<index && node != null; i++) {
    node = node.next;
  }
  return node;
}
```

Completing the `SingleLinkedList` Class

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

public E get(int index)

```
public E get (int index) {
 if (index < 0 || index >= size) {
    throw new
      IndexOutOfBoundsException(Integer.toString(index));
   }
  Node<E> node = getNode(index);
  return node.data;
}
```

public E set(int index, E newValue)

```
public E set (int index, E anEntry) {
  if (index < 0 || index >= size) {
    throw new
       IndexOutOfBoundsException(Integer.toString(index));
  }
  Node<E> node = getNode(index);
  E result = node.data;
  node.data = anEntry;
  return result;
}
```

public void add(int index, E item)

```
public void add (int index, E item) {
 if (index < 0 || index > size) {
   throw new
     IndexOutOfBoundsException(Integer.toString(index));
 }
 if (index == 0) {
   addFirst(item);
 } else {
   Node<E> node = getNode(index-1);
   addAfter(node, item);
 }
}
```

public boolean add(E item)

- To add an item to the end of the list

```
public boolean add (E item) {
    add(size, item);
    return true;
}
```

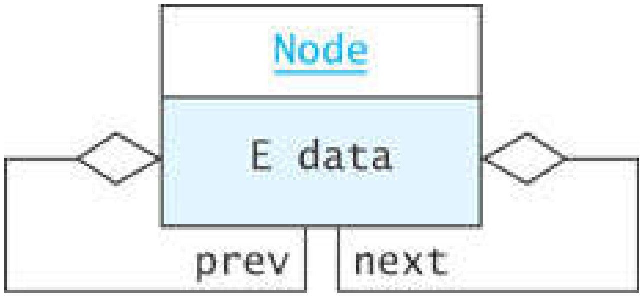
Double-Linked Lists and Circular Lists

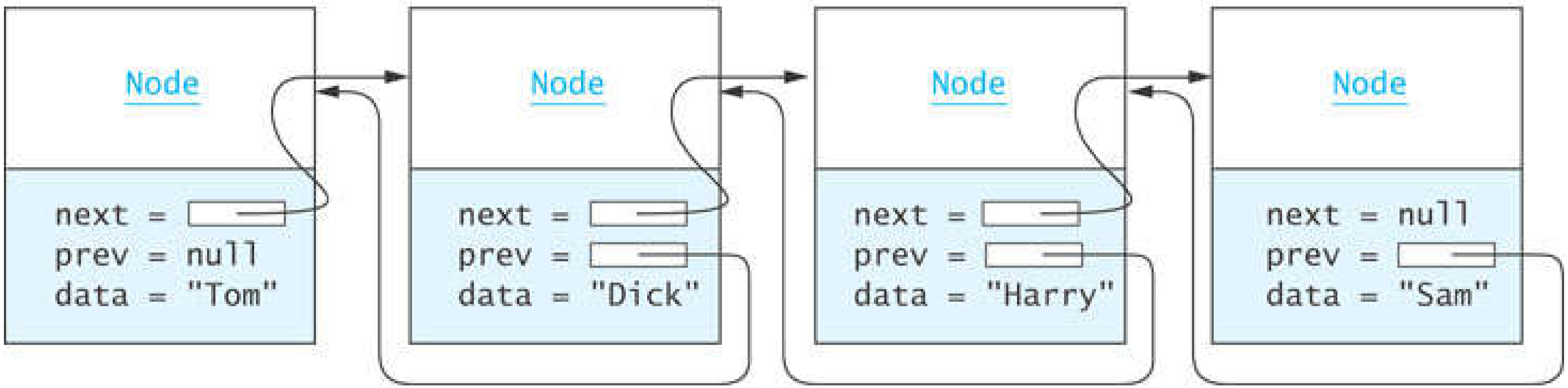
Section 2.6

Double-Linked Lists

- Limitations of a singly-linked list include:
  - Insertion at the front is O(1); insertion at other positions is O(*n*)
  - Insertion is convenient only after a referenced node
  - Removing a node requires a reference to the previous node
  - We can traverse the list only in the forward direction
- We can overcome these limitations:
  - Add a reference in each node to the previous node, creating a *double-linked list*

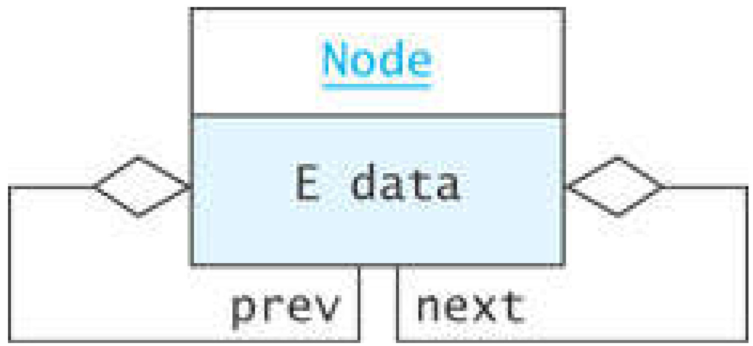
Double-Linked Lists (cont.)



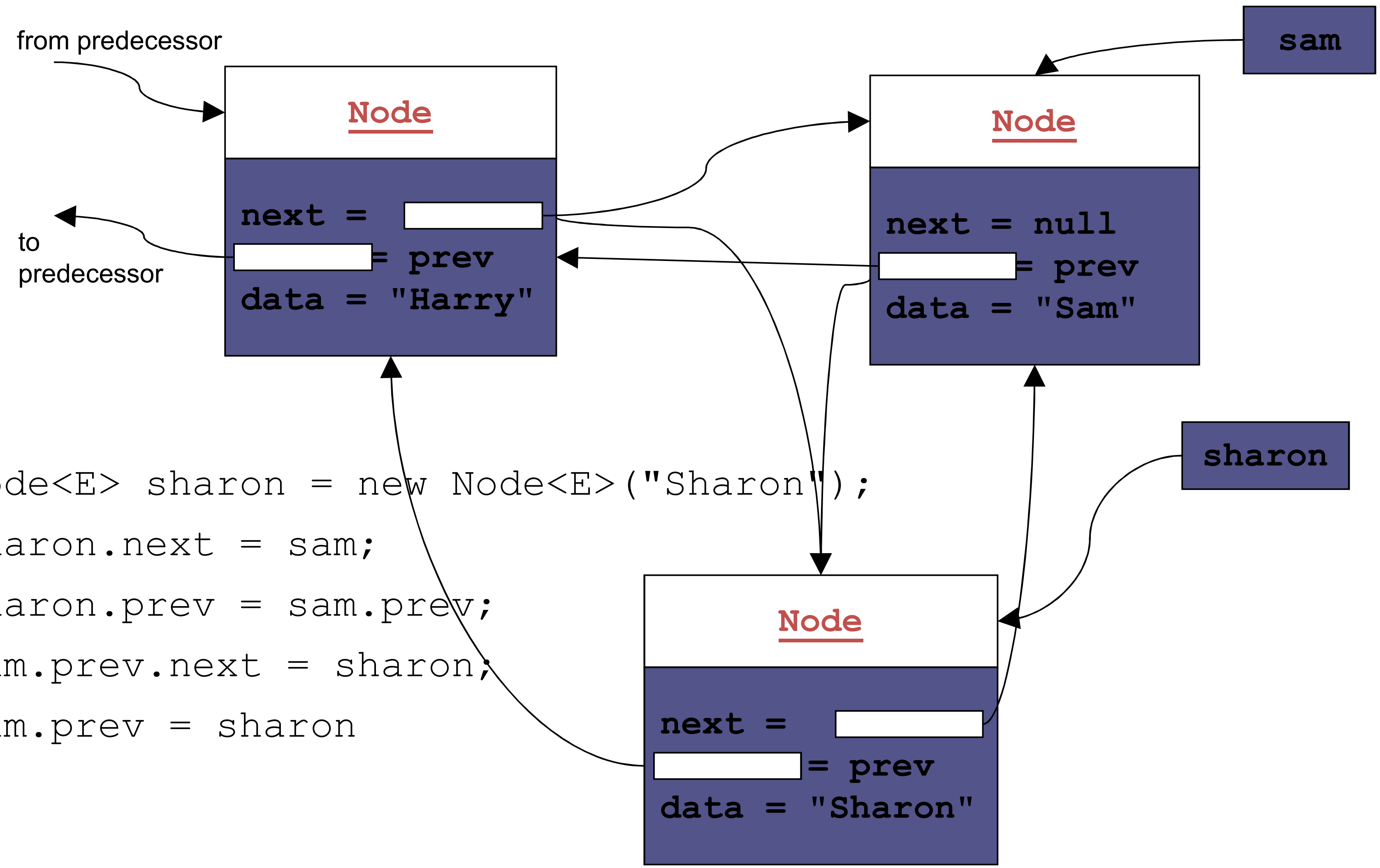


Node **Class**

```
private static class Node<E> {
  private E data;
  private Node<E> next = null;
  private Node<E> prev = null;

  private Node(E dataItem) {
    data = dataItem;
  }
}
```

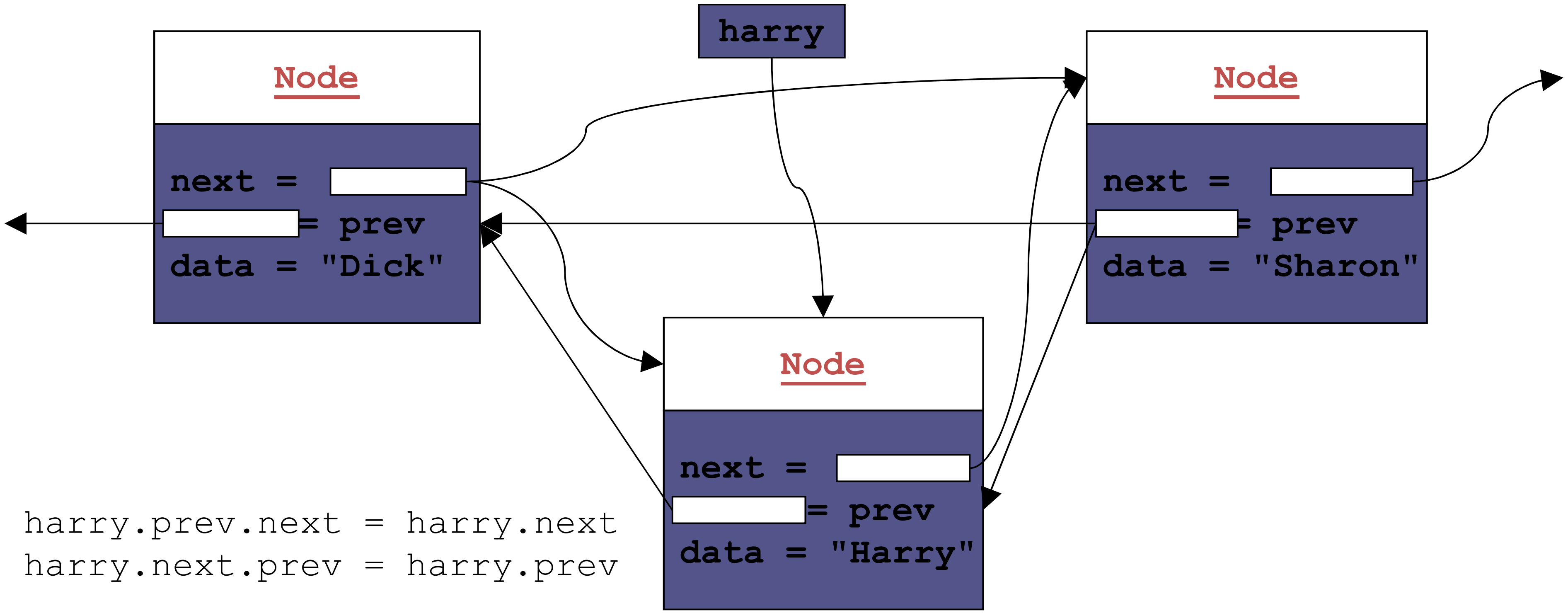


Inserting into a Double-Linked List



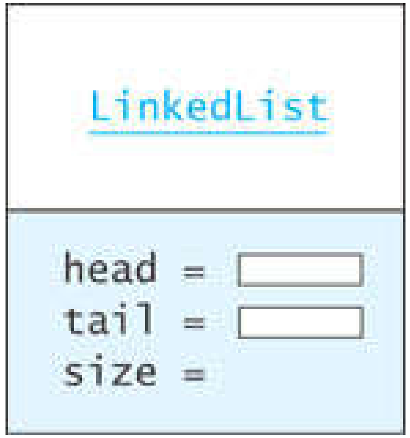

```
Node<E> sharon = new Node<E>("Sharon");
sharon.next = sam;
sharon.prev = sam.prev;
sam.prev.next = sharon;
sam.prev = sharon
```

Removing from a Double-Linked List




```
harry.prev.next = harry.next
harry.next.prev = harry.prev
```

A Double-Linked List Class

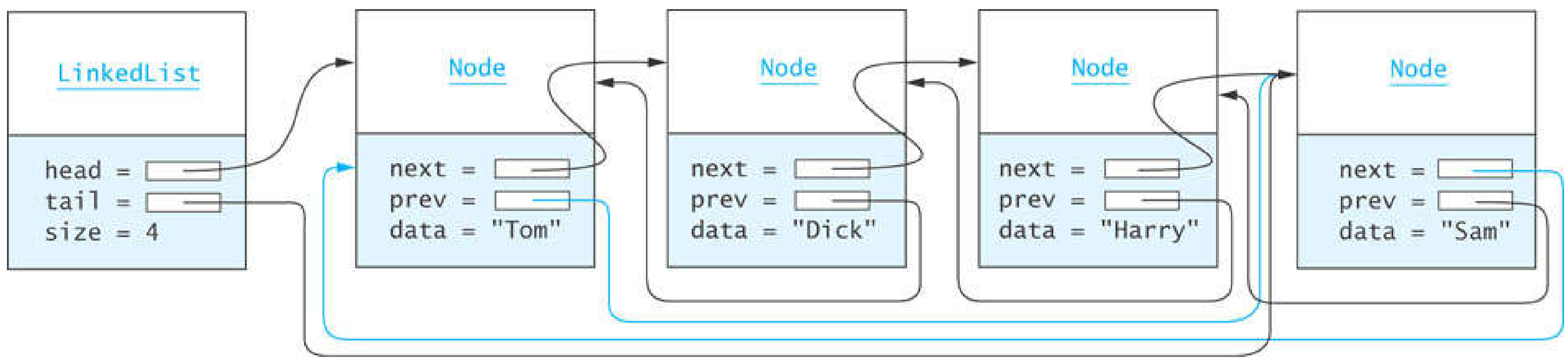


- So far we have worked only with internal nodes
- As with the single-linked class, it is best to access the internal nodes with a double-linked list object
- A double-linked list object has data fields:
  - `head` (a reference to the first list `Node`)
  - `tail` (a reference to the last list `Node`)
  - `size`
- Insertion at either end is O(1); insertion elsewhere is still O(*n*)

Circular Lists

- Circular double-linked list:
  - Link last node to the first node, and
  - Link first node to the last node
- We can also build singly-linked circular lists:
  - Traverse in forward direction only
- **Advantages:**
  - Continue to traverse even after passing the first or last node
  - Visit all elements from any starting point
  - Never fall off the end of a list
- **Disadvantage**: Code must avoid an infinite loop!

Circular Lists (cont.)



The `LinkedList` Class and the `Iterator`, `ListIterator`, and `Iterable` Interfaces

Section 2.7

The `LinkedList` Class

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.

The Iterator

- An iterator can be viewed as a moving place marker that keeps track of the current position in a particular linked list
- An `Iterator` object for a list starts at the first node
- The programmer can move the `Iterator` by calling its `next` method
- The `Iterator` stays on its current list item until it is needed
- An `Iterator` traverses in O(n) while a list traversal using `get()` calls in a linked list is O(n^2)

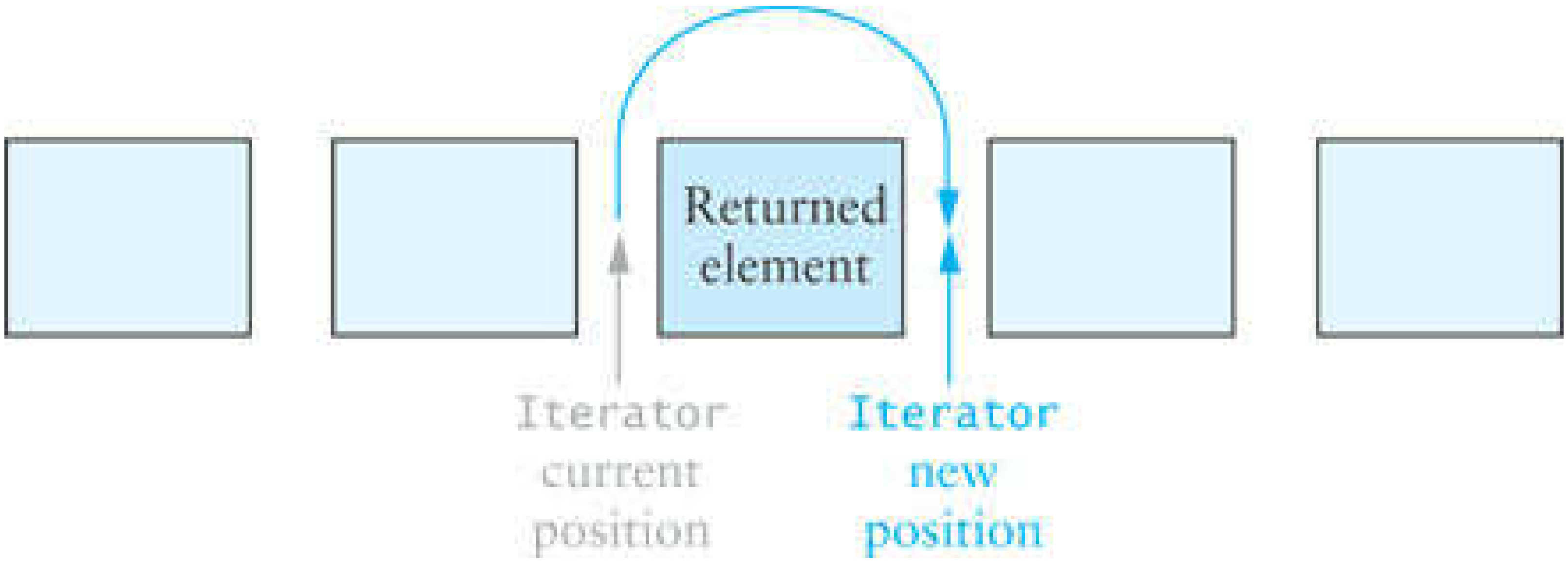
`Iterator` Interface

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

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** (cont.)

- An `Iterator` is conceptually *between* elements; it does not refer to a particular object at any given time



`Iterator` **Interface** (cont.)

- In the following loop, we process all items in `List<Integer>` through an `Iterator`

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

Iterators and Removing Elements

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

Iterators and Removing Elements (cont.)

- To remove all elements from a list of type `Integer` that are divisible by a particular value:

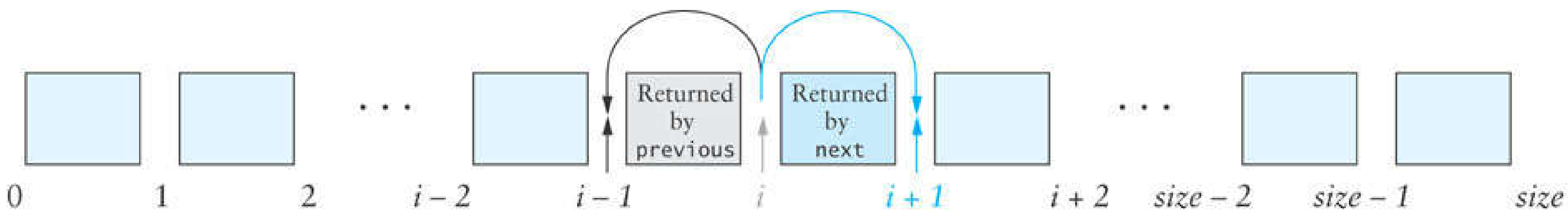
```
public static void removeDivisibleBy(LinkedList<Integer>
                                              aList, int div) {
  Iterator<Integer> iter = aList.iterator();
  while (iter.hasNext()) {
    int nextInt = iter.next();
    if (nextInt % div == 0) {
      iter.remove();
    }
  }
}
```

`ListIterator` Interface

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

`ListIterator` Interface (cont.)

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



ListIterator Interface (cont.)

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.

`ListIterator` Interface (cont.)

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`.

Comparison of `Iterator` and `ListIterator`

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

Conversion Between `ListIterator` and an Index

- `ListIterator`:
  - `nextIndex()` returns the index of item to be returned by `next()`
  - `previousIndex()` returns the index of item to be returned by `previous()`
- `LinkedList` has method `listIterator(int index)`
  - Returns a `ListIterator` positioned so `next()` will return the item at position `index`

Conversion Between `ListIterator` and an Index (cont.)

- The `listIterator (int index)` method creates a new `ListIterator` that starts at the beginning, and walks down the list to the desired position – generally an O(n) operation

Enhanced `for` Statement

- Java 5.0 introduced an enhanced `for` statement
- The enhanced `for` statement creates an `Iterator` object and implicitly calls its `hasNext` and `next` methods
- Other `Iterator` methods, such as `remove`, are not available

Enhanced `for` Statement (cont.)

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

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

Enhanced `for` Statement (cont.)

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

```
sum = 0;
for (int nextInt : myList) {
  sum += nextInt;
}
```

Enhanced `for` Statement (cont.)

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

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

`Iterable` **Interface**

- Each class that implements the `List` interface must provide an `iterator` method
- The `Collection` interface extends the `Iterable` interface
- All classes that implement the `List` interface (a subinterface of `Collection`) must provide an `iterator` method
- Allows use of the Java 5.0 *for-each* loop

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

Implementation of a Double-Linked List Class

Section 2.8

KWLinkedList

- We will define a `KWLinkedList` class which 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` which you should

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

KWLinkedList (cont.)

```
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;

    . . .
```

Add **Method**

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

To use a `ListIterator` object to implement add:

1. Obtain an iterator that is positioned just before the `Node` at position `index`
2. Insert a new `Node` containing `obj` before the `Node` currently referenced by this iterator

```
/** Add an item at the specified
     index.
    @param index The index at
            which the object is
            to be inserted
    @param obj The object to be
            inserted
    @throws
       IndexOutOfBoundsException
            if the index is out
             of range
            (i < 0 || i > size())
*/
public void add(int index, E obj)
{
  listIterator(index).add(obj);
}
```

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

`Get` Method

1. Obtain a reference, `nodeRef`, to the node at position `index`
2. Return the contents of the `Node` referenced by `nodeRef`

```
/** Get the element at position
    index.
    @param index Position of
           item to be retrieved
    @return The item at index
*/
public E get(int index) {
  return
    listIterator(index).next();
}
```

Other `Add` and `Get` Methods

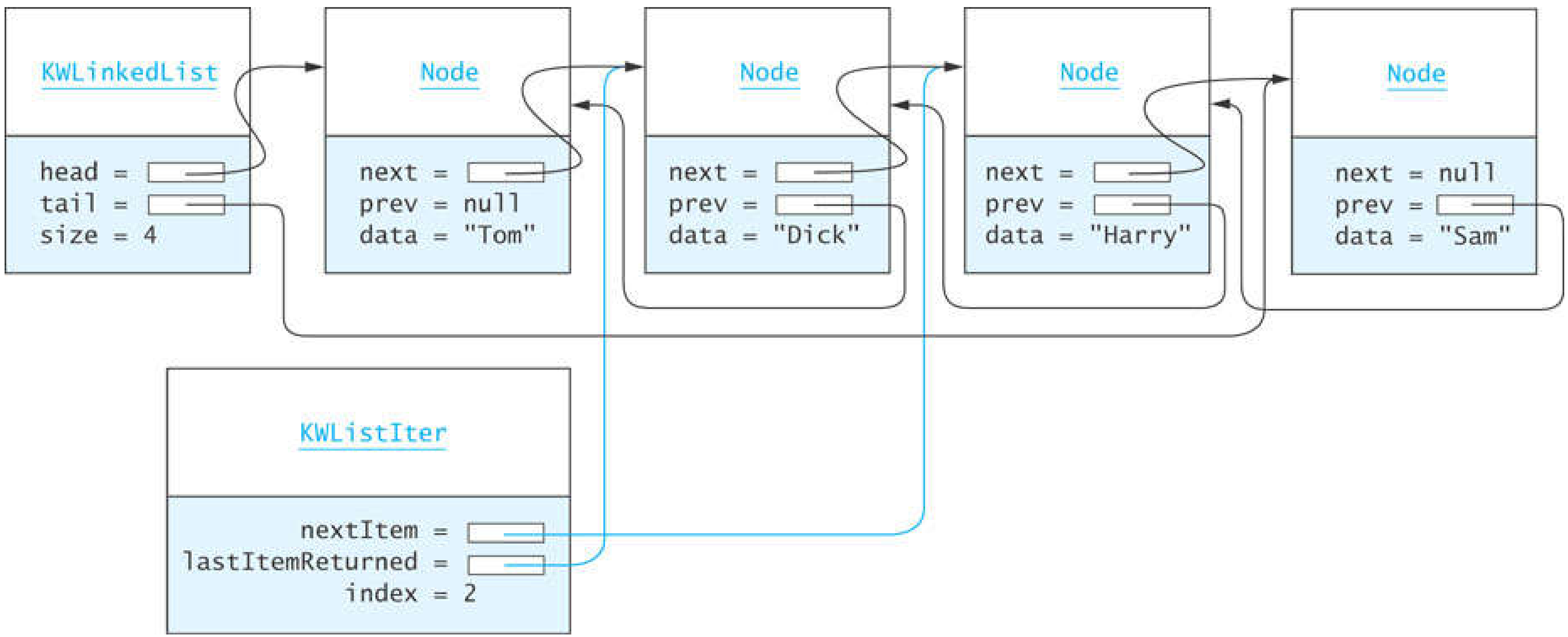
```
public void addFirst(E item) {
  add(0, item);
}

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

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

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

Implementing the `ListIterator` Interface

- `KWListIter` is an inner class of `KWLinkedList` which 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 (cont.)



Implementing the `ListIterator` Interface (cont.)

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

Constructor

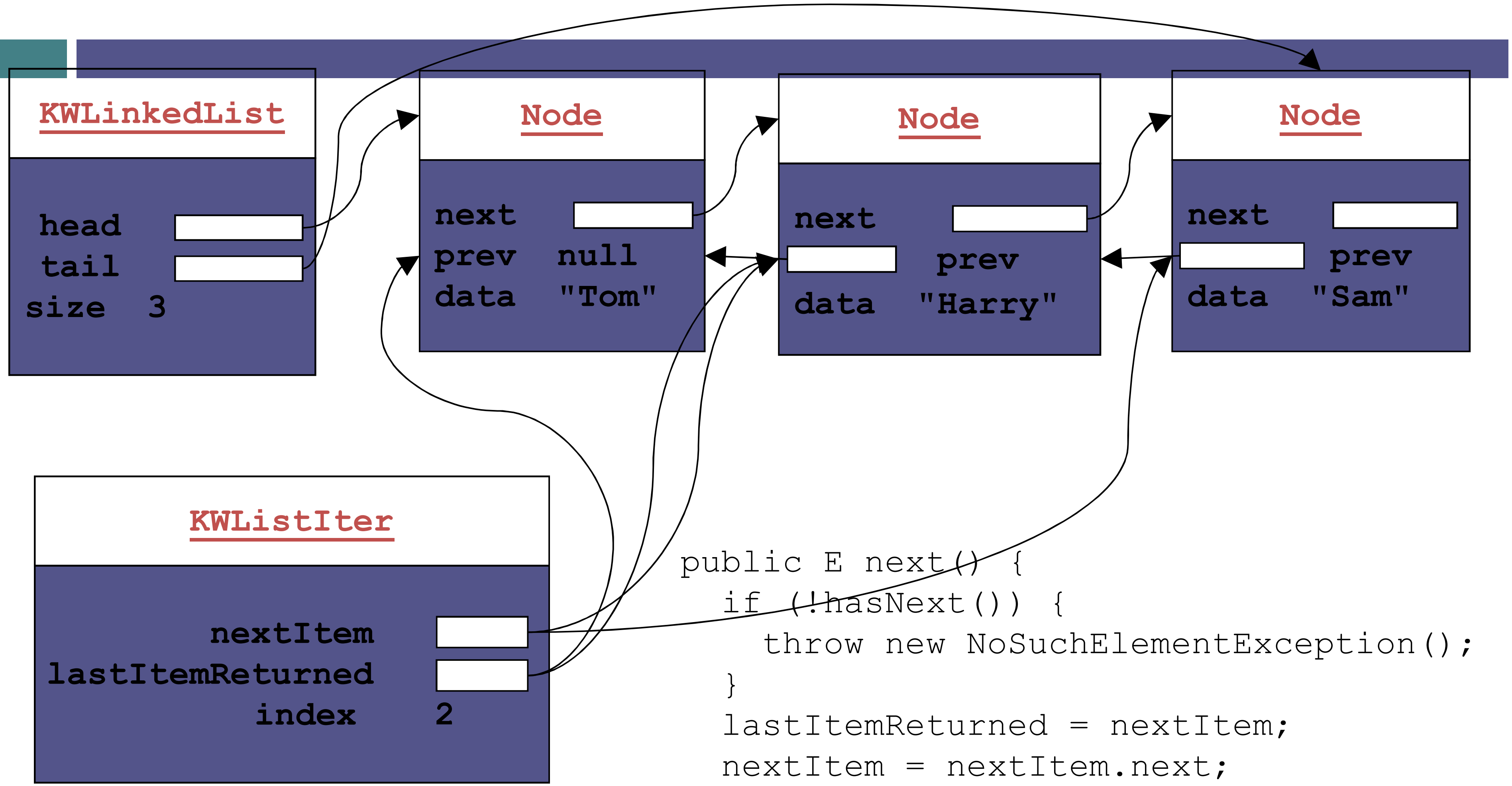
```
public KWListIter(int i) {
  // Validate i parameter.
  if (i < 0 || i > size) {
    throw new IndexOutOfBoundsException("Invalid index " + i);
  }
  lastItemReturned = null; // No item returned yet.
  // Special case of last item
  if (i == size) {
    index = size;
    nextItem = null;
  }
  else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

The `hasNext()` Method

- tests to see if `nextItem` is `null`

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

Advancing the Iterator




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

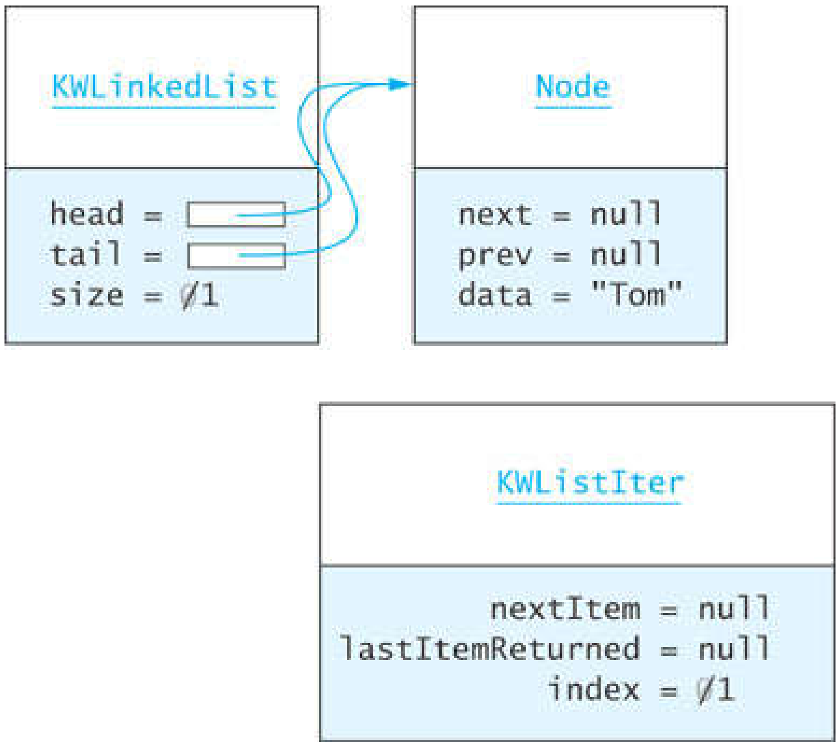
Previous Methods

```
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 four 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

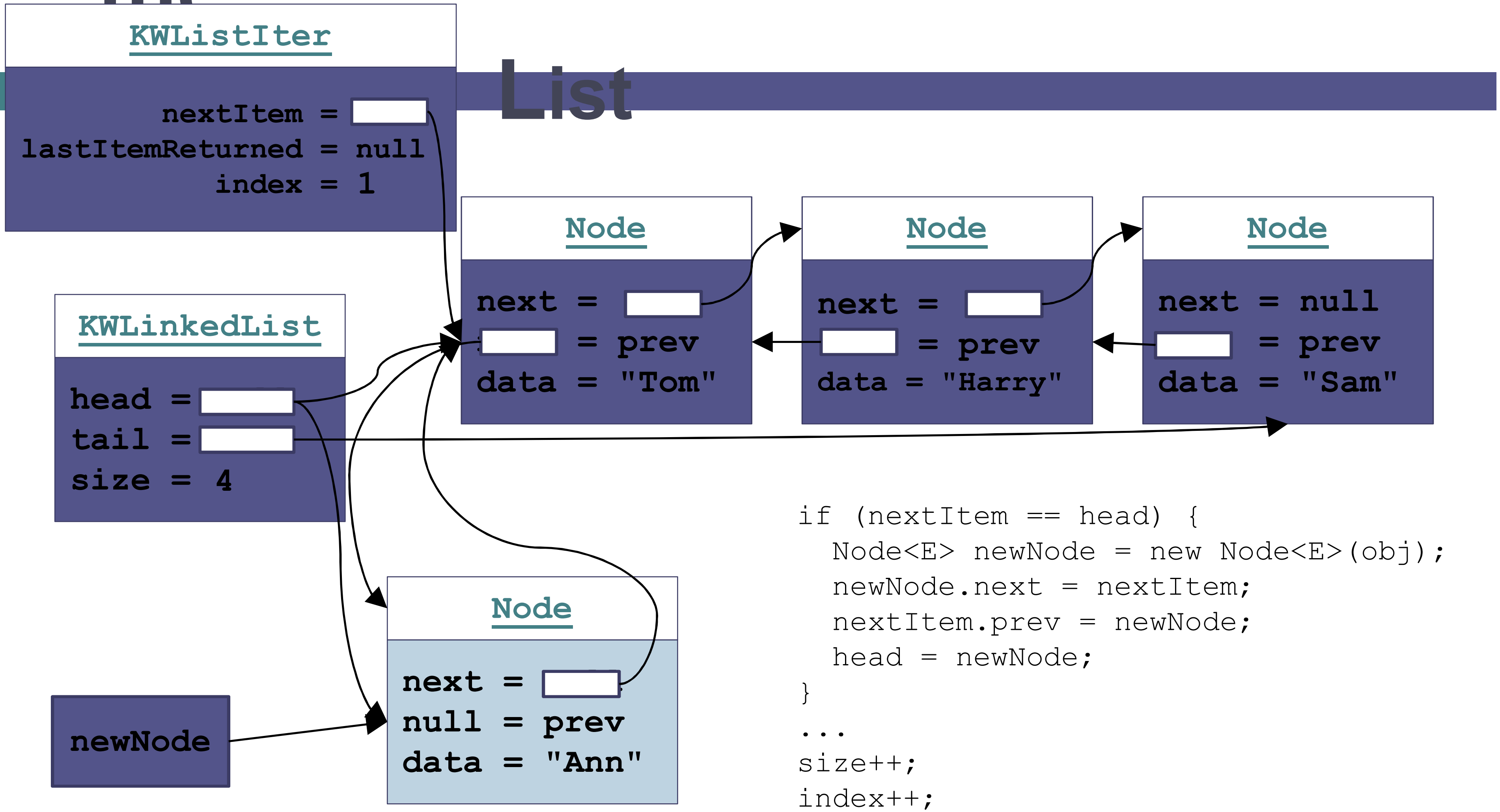
Adding to an Empty List



(after insertion)

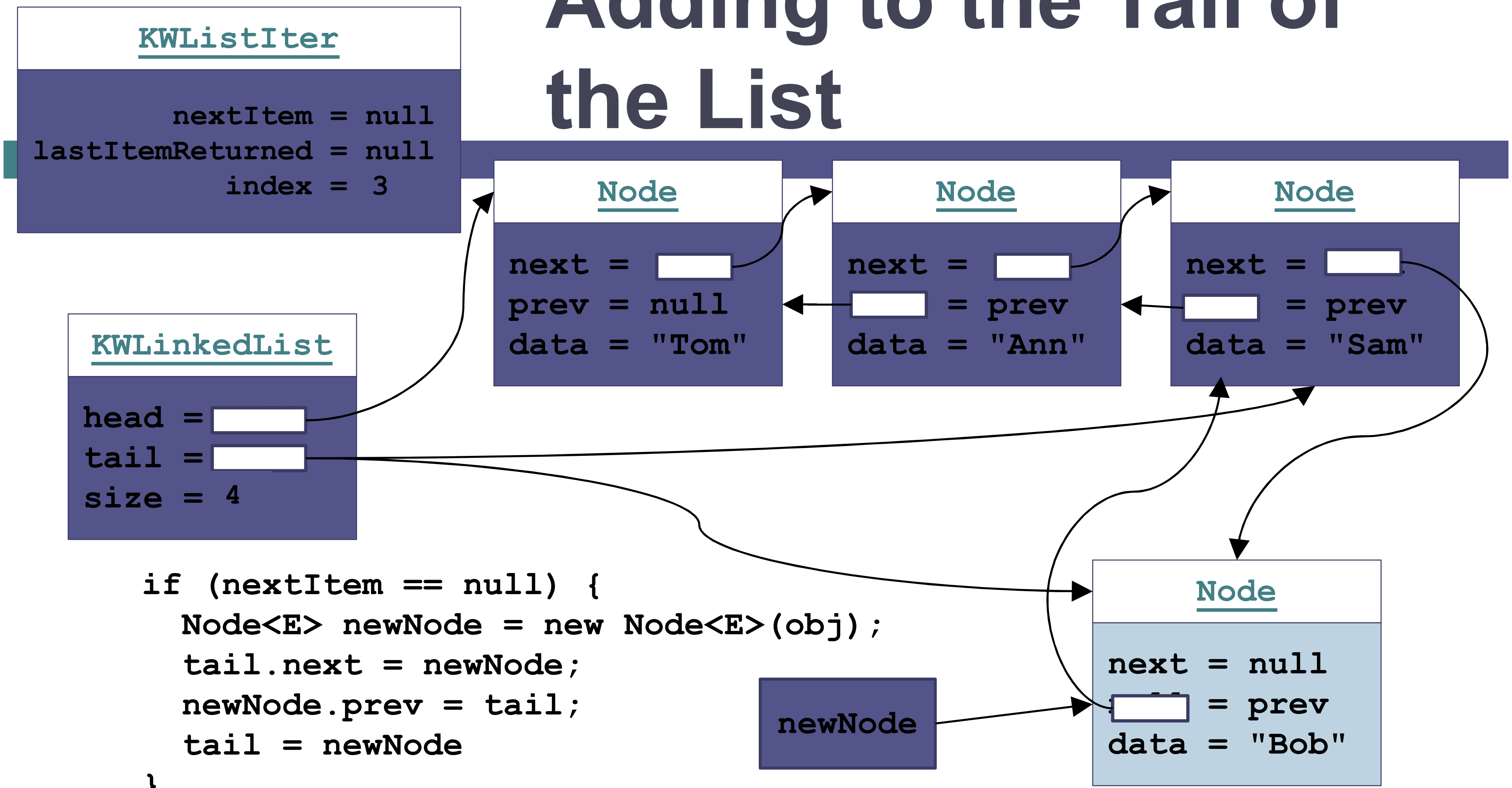
```
if (head == null) {
   head = new Node<E>(obj);
   tail = head;
}
...
size++
```

Adding to the Head of the List



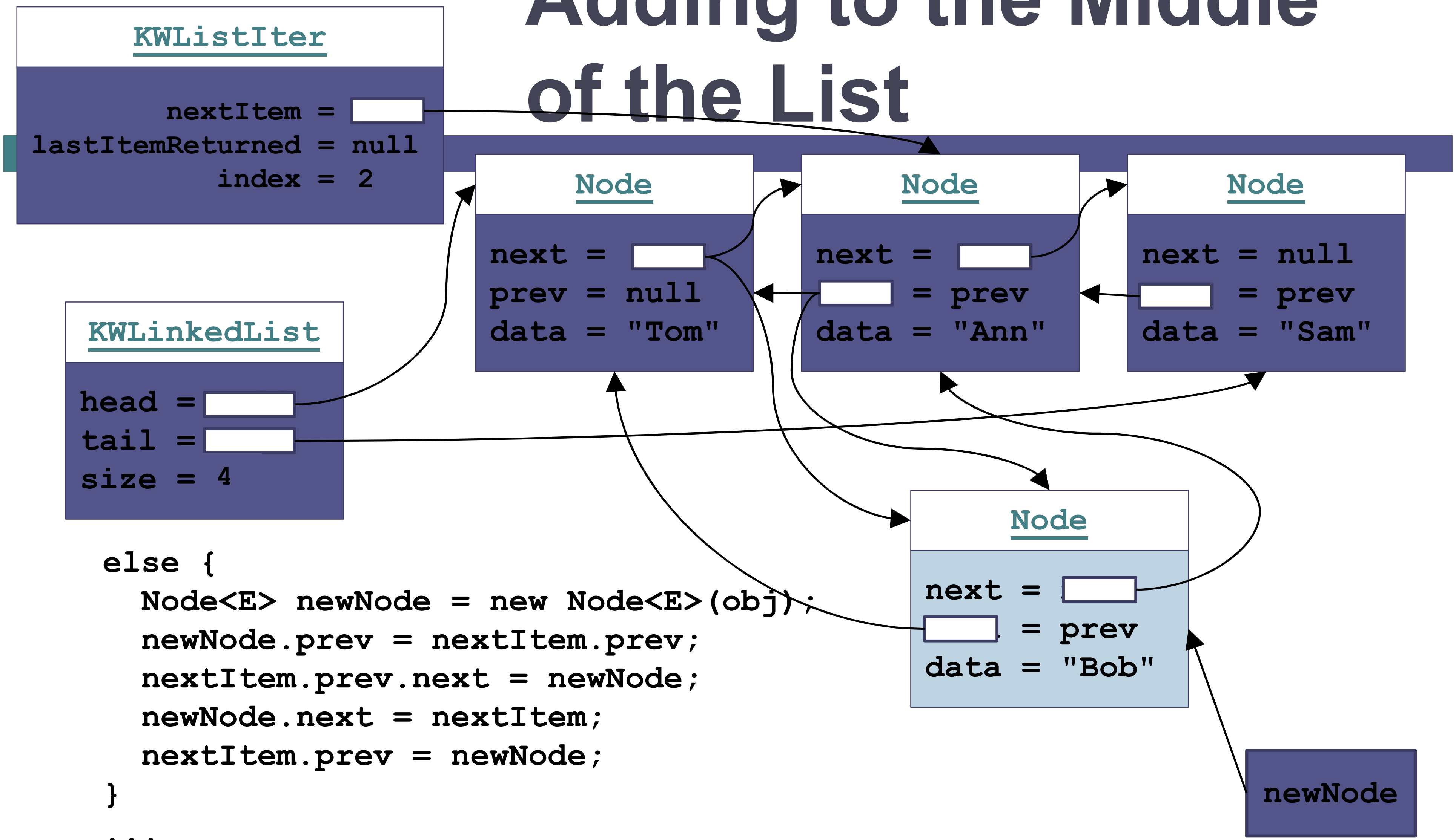

```
if (nextItem == head) {
  Node<E> newNode = new Node<E>(obj);
  newNode.next = nextItem;
  nextItem.prev = newNode;
  head = newNode;
}
...
size++;
index++;
```

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++;
```

Inner Classes: Static and Nonstatic

- `KWLinkedList` contains two inner classes:
  - `Node<E>` is declared static: there is no need for it to access the data fields of its parent class, `KWLinkedList`
  - `KWListIter` cannot be declared static because its methods access and modify data fields of `KWLinkedList`'s parent object which created it
- An inner class which is not static contains an implicit reference to its parent object and can reference the fields of its parent object
- Since its parent class is already defined with the parament `<E>`, `KWListIter` cannot be declared as `KWListIter<E>`; if it were, an *incompatible types* syntax error would occur

The Collections Framework Design

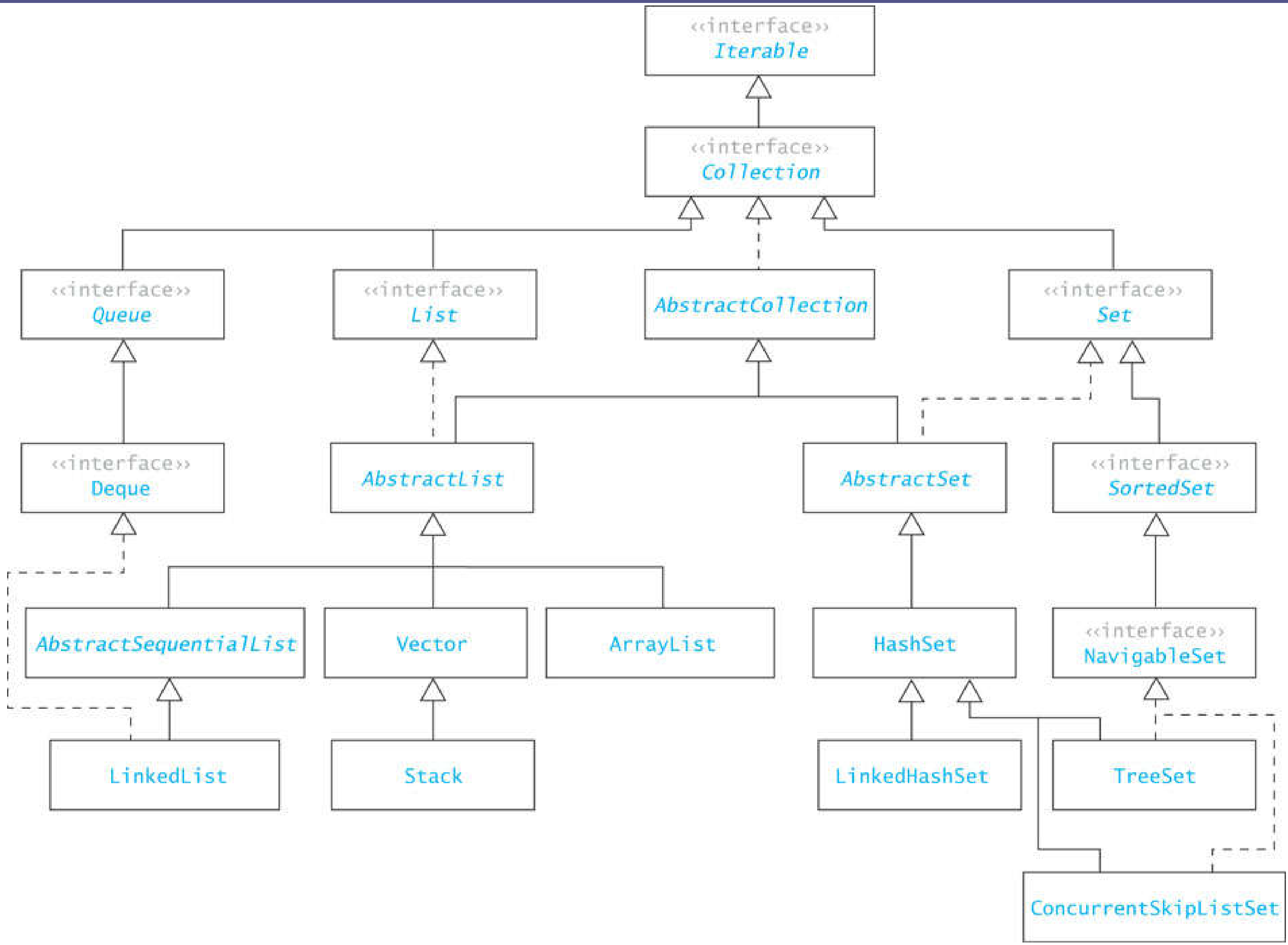
Section 2.9

The `Collection` Interface

- Specifies a subset of methods in the `List` interface, specifically excluding
  - `add(int, E)`
  - `get(int)`
  - `remove(int)`
  - `set(int, E)`

  but including
  - `add(E)`
  - `remove(Object)`
  - the `iterator` method

The `Collection` Framework



Common Features of Collections

- Collections
  - grow as needed
  - hold references to objects
  - have at least two constructors:
    - one to create an empty collection and
    - one to make a copy of another collection

Common Features of Collections (cont.)

Method	Behavior
`boolean add(E obj)`	Ensures that the collection contains the object `obj`. Returns `true` if the collection was modified.
`boolean contains(E obj)`	Returns `true` if the collection contains the object `obj`.
`Iterator<E> iterator()`	Returns an `Iterator` to the collection.
`int size()`	Returns the size of the collection.

- In a general `Collection` the order of elements is not specified
- For collections implementing the `List` interface, the order of the elements is determined by the index

Common Features of Collections (cont.)

Method	Behavior
`boolean add(E obj)`	Ensures that the collection contains the object `obj`. Returns `true` if the collection was modified.
`boolean contains(E obj)`	Returns `true` if the collection contains the object `obj`.
`Iterator<E> iterator()`	Returns an `Iterator` to the collection.
`int size()`	Returns the size of the collection.

- In a general `Collection`, the position where an object is inserted is not specified
- In `ArrayList` and `LinkedList`, `add(E)` always inserts at the end and always returns `true`

`AbstractCollection`, `AbstractList`, and `AbstractSequentialList`

- The Java API includes several "helper" abstract classes to help build implementations of their corresponding interfaces
- By providing implementations for interface methods not used, the helper classes require the programmer to extend the `AbstractCollection` class and implement only the desired methods

Implementing a Subclass of `Collection<E>`

- Extend `AbstractCollection<E>`, which implements most operations
- You need to implement only:
  - `add(E)`
  - `size()`
  - `iterator()`
  - an inner class that implements `Iterator<E>`

Implementing a Subclass of `List<E>`

- Extend `AbstractList<E>`
- You need to implement only:
  - `add(int, E)`
  - `get(int)`
  - `remove(int)`
  - `set(int, E)`
  - `size()`
- `AbstractList` implements `Iterator<E>` using the index

`AbstractCollection`, `AbstractList`, and `AbstractSequentialList`

- Another more complete way to declare `KWArrayList` is:

```
public class KWArrayList<E> extends AbstractList<E>
                            implements List<E>
```

- Another more complete, way to declare `KWLinkedLinkedList` is:

```
public class KWLinkedList<E> extends
                     AbstractSequentialList<E>
                     implements List<E>
```

`List` and `RandomAccess` Interfaces

- Accessing a `LinkedList` using an index requires an O(n) traversal of the list until the index is located
- The `RandomAccess` interface is applied to list implementations in which indexed operations are efficient (e.g. `ArrayList`)
- An algorithm can test to see if a parameter of type `List` is also of type `RandomAccess` and, if not, take appropriate measures to optimize indexed operations

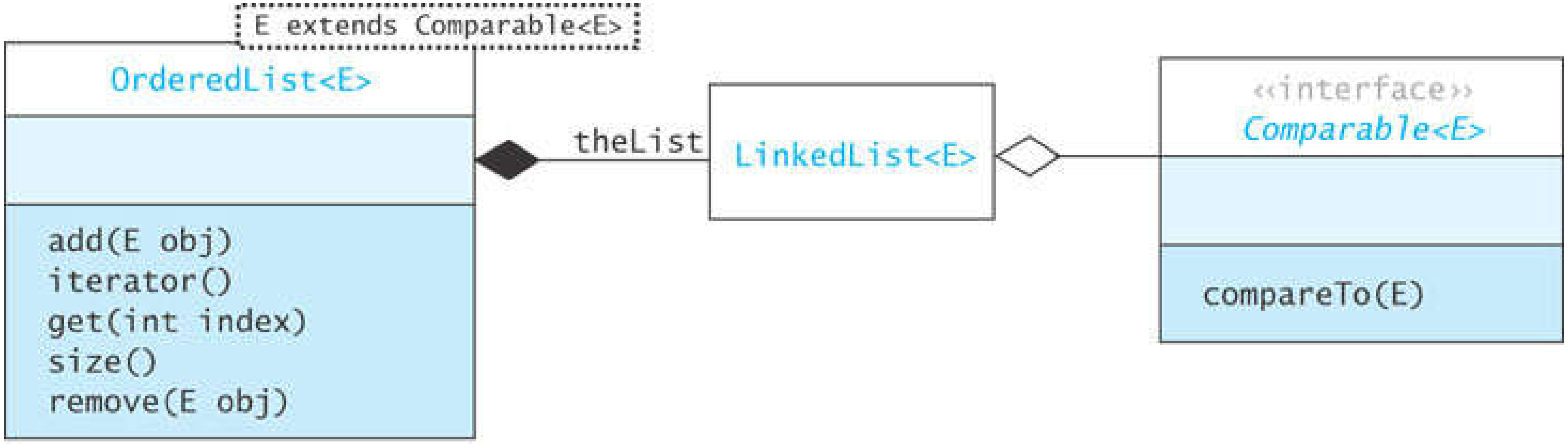
Application of the `LinkedList` Class

Section 2.10

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)
  - Implement a `Comparable` interface by providing a `compareTo(E)` method
  - Use a `LinkedList` class as a component of the `OrderedList`
    - if `OrderedList` extended `LinkedList`, the user could use `LinkedList`'s `add` methods to add an element out of order

Class Diagram for `OrderedList`



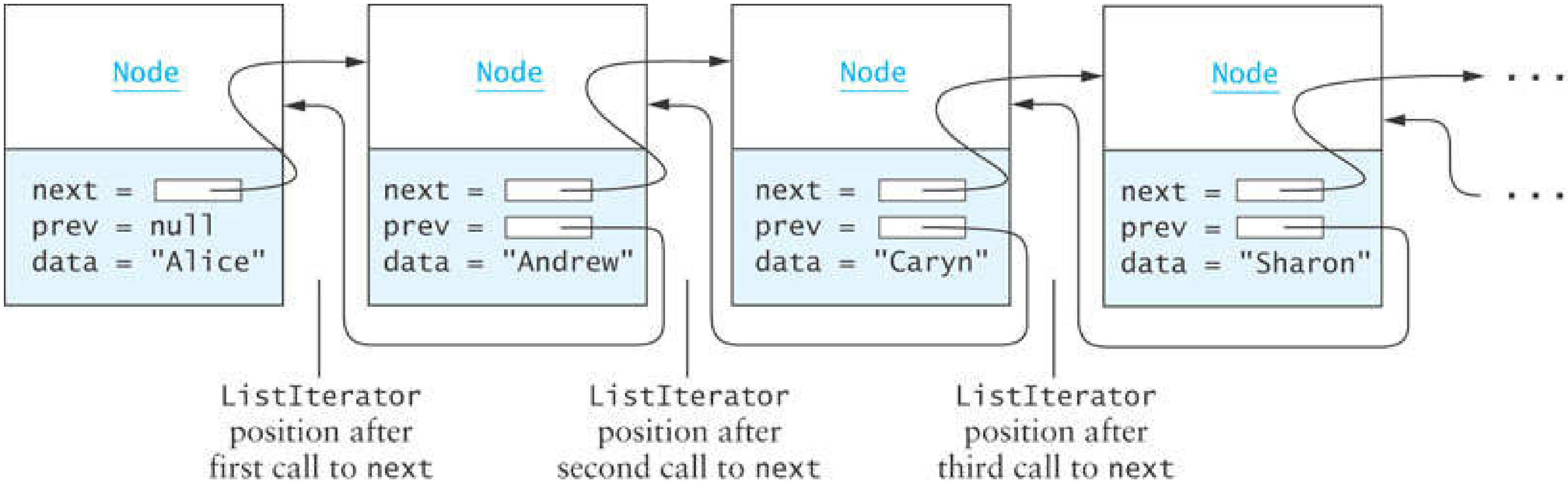
Design

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`

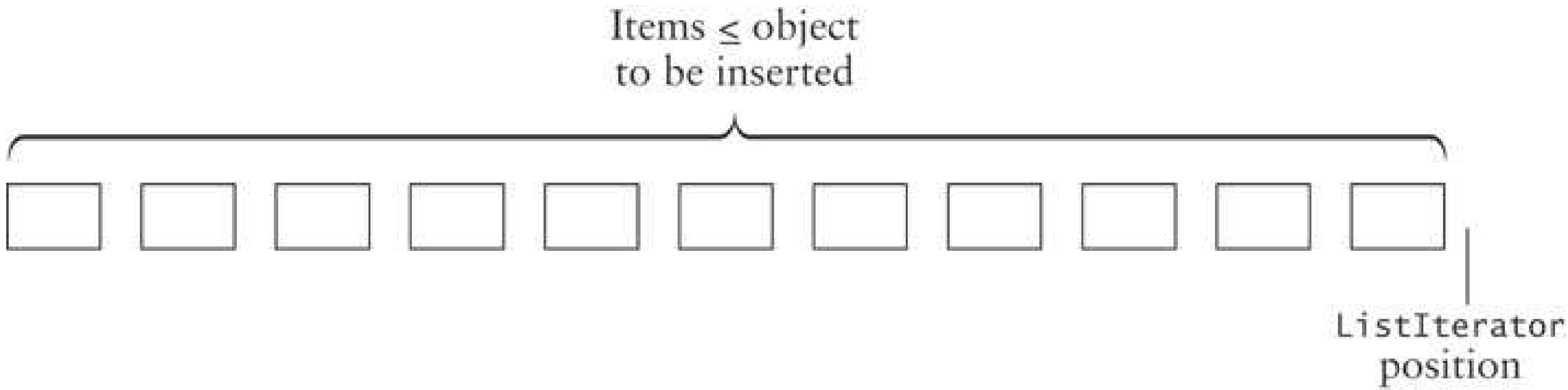
- Strategy for inserting new element `e`:
  - Find first item > `e`
  - Insert `e` before that item
- Refined with an iterator:
  - Create `ListIterator` that starts at the beginning of the list
  - While the `ListIterator` is not at the end of the list and `e` >= the next item
    - Advance the `ListIterator`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

Inserting Diagrammed

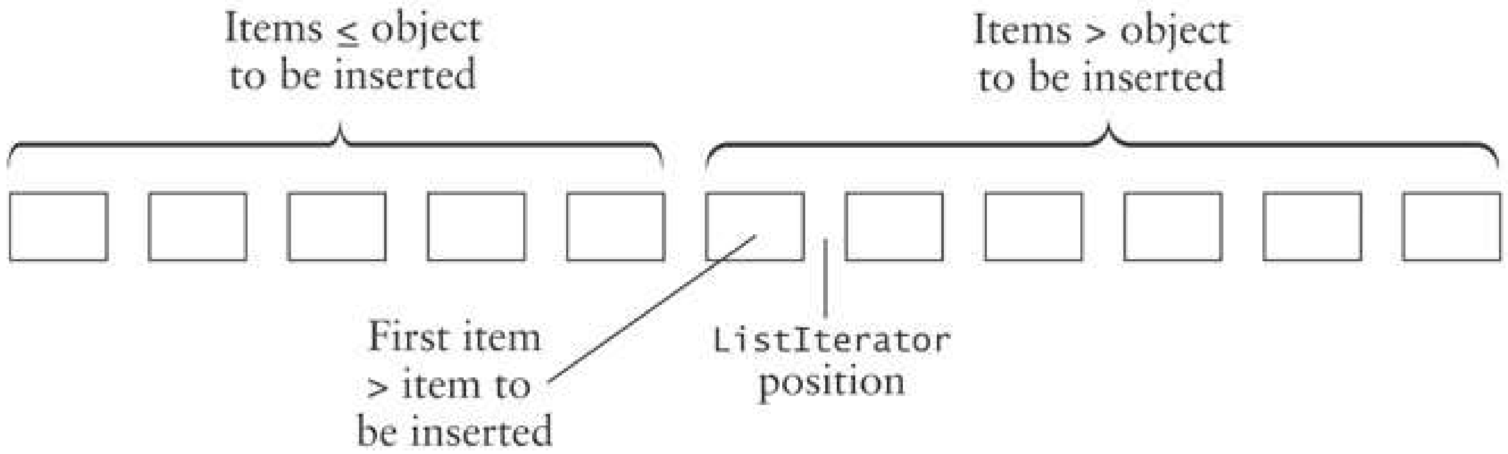


Inserting Diagrammed (cont.)

Case 1: Inserting at the end of a list



Case 2: Inserting in the middle of a list



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
      iter.previous();  // back up by one
      iter.add(e);      // add new one
      return;           // done
    }
  }
  iter.add(e);  // will add at end
}
```

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);
}
// returns an iterator positioned before the first element
public Iterator iterator() {
  return theList.iterator();
}
```

Testing

Section 2.11

Testing

- Testing runs a program or part of a program under controlled conditions to verify that results are as expected
- Testing detects program defects after the program compiles (all syntax error have been removed)
- While extremely useful, testing cannot detect the absence of all defects in complex programs

Testing Levels

- **Unit testing**: tests the smallest testable piece of the software, often a class or a sufficiently complex method
- **Integration testing**: tests integration among units
- **System testing**: tests the whole program in the context in which it will be used
- **Acceptance testing**: system testing designed to show that a program meets its functional requirements

Types of Testing

- Black-box testing:
  - tests the item (method, class, or program) based on its interfaces and functional requirements
  - is also called *closed-box* or *functional* testing
  - is accomplished by varying input parameters across the allowed range and outside the allowed range, and comparing with independently calculated results

Types of Testing (cont.)

- White-box testing:
  - tests the item (method, class, or program) with knowledge of its internal structure
  - is also called *glass-box, open-box,* or *coverage* testing
  - exercises as many paths through the element as possible
  - provides appropriate coverage
    - statement – ensures each statement is executed at least once
    - branch – ensures each choice of branch (if , switch, loops) is taken
    - path – tests each path through a method

Preparations for Testing

- A test plan should be developed early in the design stage—the earlier an error is detected, the easier and less expensive it is to correct it
- Aspects of test plans include deciding:
  - how the software will be tested
  - when the tests will occur
  - who will do the testing
  - what test data will be used

Testing Tips for Program Systems

1. Carefully document method operation, parameter, and class attributes using comments; follow Javadoc conventions
2. Leave a trace of execution by displaying the method name as you enter it
3. Display values of all input parameters upon entering a method and values of any class attributes accessed by the method
4. Display values of all method outputs after returning from a method, together with any class attributes that are modified by a method

Testing Tips for Program Systems (cont.)

- An efficient way to display values of parameters, return values, and class attributes:

```
private static final boolean TESTING = true; // or false to
                                             // disable

if (TESTING) {
  // Code for output statements
}
```

- Remove these features when you are satisfied with the testing results
- You can define different `boolean` flags for different tests

Developing the Test Data

- In black-box testing, test data should check for all expected inputs as well as unanticipated data
- In white-box testing, test data should be designed to ensure all combinations of paths through the code are executed

Testing Boundary Conditions

- Example

```
for (int i = 0; i < x.length; i++) {
  if (x[i] == target)
    return i;
}
```

- Test the boundary conditions (for white-box and black-box testing) when `target` is:
  - first element (`x[0] == target` is `true`)
  - last element (`x[length-1] == target` is `true`)
  - not in array (`x[i] == target` is always `false`)
  - present multiple times (`x[i] == target` for more than one value of `i`)

Testing Boundary Conditions (cont.)

```
for (int i = 0; i < x.length; i++) {
  if (x[i] == target)
    return i;
}
```

- Test for the typical situation when `target` is:
  - somewhere in the middle
- and for the boundary conditions when the array has
  - only one element
  - no elements

Testing Boundary Conditions (cont.)

```
public static void main(String[] args) {
    int[] x = {5, 12, 15, 4, 8, 12, 7}; // Array to search.
    // Test for target as first element.
    verify(x, 5, 0);
    // Test for target as last element.
    verify(x, 7, 6);
    // Test for target not in array.
    verify(x, -5, -1);
    // Test for multiple occurrences of target.
    verify(x, 12, 1);
    // Test for target somewhere in middle.
    verify(x, 4, 3);
    // Test for 1-element array.
    x = new int[1];
    x[0] = 10;
    verify(x, 10, 0);
    verify(x, -10, -1);
    // Test for an empty array.
    x = new int[0];
    verify(x, 10, -1);
}
```

Testing Boundary Conditions (cont.)

```
private static void verify(int[] x, int target, int expected) {
    int actual = search(x, target);
    System.out.print("search(x, " + target + ") is "
                     + actual + ", expected " + expected);
    if (actual == expected)
          System.out.println(": Pass");
    else
          System.out.println(": ****Fail");
}
```

Testing Boundary Conditions (cont.)

```
search(x, 5) is 0, expected 0:  Pass
search(x, 7) is 6, expected 6:  Pass
search(x, -5) is -1, expected -1:  Pass
search(x, 12) is 1, expected 1:  Pass
search(x, 4) is 3, expected 3:  Pass
search(x, 10) is 0, expected 0:  Pass
search(x, -10) is -1, expected -1:  Pass
search(x, 10) is -1, expected -1:  Pass
```

Stubs

- *Stubs* are method placeholders for methods called by other classes, but not yet implemented
- Stubs allowing testing as classes are being developed
- A sample stub:

```
public void save() {
  System.out.println("Stub for save has been called");
  modified = false;
}
```

Stubs (cont.)

- Stubs can
  - print out value of inputs
  - assign predictable values to outputs
  - change the state of variables

Preconditions and Postconditions

- A *precondition* is a statement of any assumptions or constraints on the input parameters before a method begins execution
- A *postcondition* describes the result of executing the method, including any change to the object's state
- A method's preconditions and postconditions serve as a contract between a method caller and the method programmer

```
/** Method Save
    pre: the initial directory contents are read from a
          data file
    post: writes the directory contents back to a data
           file
*/
public void save() {
. . .
}
```

Drivers

- Another testing tool
- A driver program
  - declares any necessary object instances and variables
  - assigns values to any of the method's inputs (specified by the preconditions)
  - calls the method
  - displays the outputs returned by the method
- Driver program code can be added to a class's `main` method (each class can have a `main` method; only one `main` method - the one you designate to execute - will run)

Finally

- JUnit, a popular program for Java projects, helps you develop testing programs (see Appendix C)
- Many IDEs are shipped with debugger programs you can use for testing

Testing `OrderedList`

- To test an `OrderedList`,
  - store a collection of randomly generated integers in an `OrderedList`
  - test insertion at beginning of list: insert a negative integer
  - test insertion at end of list: insert an integer larger than any integer in the list
  - create an iterator and iterate through the list, displaying an error if any element is smaller than the previous element
  - remove the first element, the last element, and a middle element, then traverse to show that order is maintained

Testing `OrderedList` (cont.)

```
Class TestOrderedList

import java.util.*;

public class TestOrderedList {
          /** Traverses ordered list and displays each element.
            Displays an error message if an element is out of order.
            @param testList An ordered list of integers
          */
          public static void traverseAndShow(OrderedList<Integer> testList) {
            int prevItem = testList.get(0);

            // Traverse ordered list and display any value that
            // is out of order.
            for (int thisItem : testList) {
                    System.out.println(thisItem);
                                 if (prevItem > thisItem)
                        System.out.println("*** FAILED, value is "
                                                          + thisItem);
                     prevItem = thisItem;
            }
             }

public static void main(String[] args) {
            OrderedList<Integer> testList = new OrderedList<Integer>();
            final int MAX_INT = 500;
            final int START_SIZE = 100;
```

(cont. on next slide)

Testing `OrderedList` (cont.)

```
    // Create a random number generator.
    Random random = new Random();
    for (int i = 0; i < START_SIZE; i++) {
            int anInteger = random.nextInt(MAX_INT);
            testList.add(anInteger);
    }

    // Add to beginning and end of list.
    testList.add(-1);
    testList.add(MAX_INT + 1);
    traverseAndShow(testList); // Traverse and display.

    // Remove first, last, and middle elements.
    Integer first = testList.get(0);
    Integer last = testList.get(testList.size() - 1);
    Integer middle = testList.get(testList.size() / 2);
    testList.remove(first);
    testList.remove(last);
    testList.remove(middle);
    traverseAndShow(testList); // Traverse and display.
     }
}
```