Lesson 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
- Generics
- The Java Collections framework (hierarchy)

Day - 1

Introduction to data structures

Section 2.0

What is data?

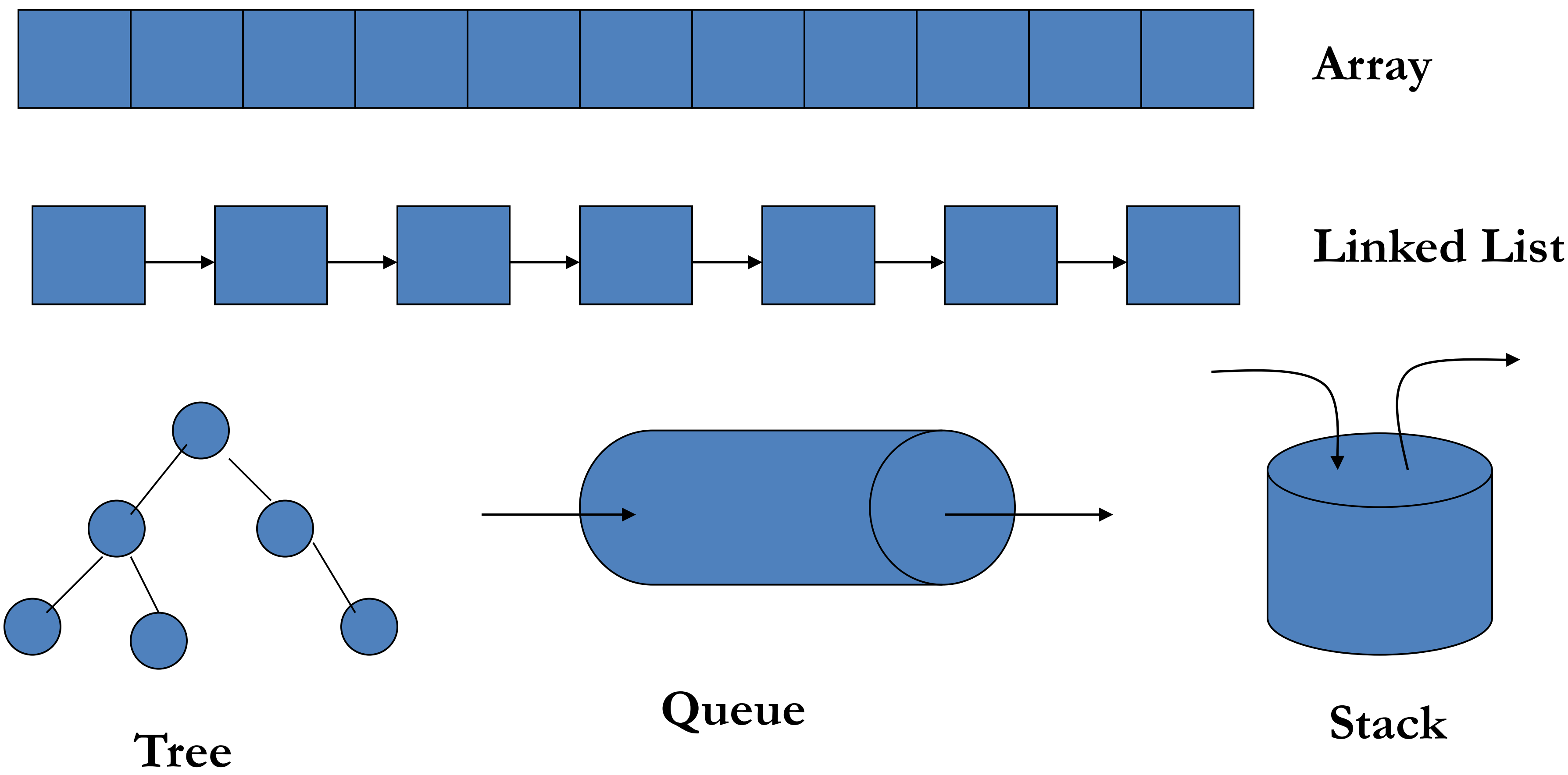
- Data
  - A collection of facts from which conclusion may be drawn
  - e.g. Data: Temperature 35°C; Conclusion: It is hot.

- Types of data
  - Textual: For example, your name (John)
  - Numeric: For example, your ID (090254)
  - Audio: For example, your voice
  - Video: For example, your voice and picture

What is data structure?

- A particular way of storing and organizing data in a computer so that it can be used efficiently and effectively.
- Data structure is the logical or mathematical model of a particular organization of data.
- A group of data elements grouped together under one name.
  - For example, an array of integers

Types of data structures



There are many, but we named a few. We'll learn these data structures in great detail!

The need for data structures

- Goal: to **organize data**

- Criteria: to facilitate **efficient**
  - **storage** of data
  - **retrieval** of data
  - **manipulation** of data

- Design Issue:
  - **select and design** appropriate data types

(This is the main motivation to learn and understand data structures)

Data structure operations

- **Traversing**
  - Accessing each data element exactly once so that certain items in the data may be processed
- **Searching**
  - Finding the location of the data element (key) in the structure
- **Insertion**
  - Adding a new data element to the structure

Data structure operations (cont.)

- **Deletion**
  - Removing a data element from the structure
- **Sorting**
  - Arrange the data elements in a logical order (ascending/descending)
- **Merging**
  - Combining data elements from two or more data structures into one

Linear or sequential search

- The *sequential search* (also called the *linear search*) is the simplest search algorithm.
- It is also the least efficient.
- It simply examines each element sequentially, starting with the first element, until it finds the key element or it reaches the end of the array.
- Demo : TestLinearSearch.java
- Animation : https://visualgo.net/en/list

Binary search

- The *binary search* is the standard algorithm for searching through a sorted sequence.
- It is much more efficient than the sequential search, but it does require that the elements be in order.
- It repeatedly divides the sequence in two, each time restricting the search to the half that would contain the element.
- You might use the binary search to look up a word in a dictionary.
- Demo : TestBinarySearch.java
- Animation : https://visualgo.net/en/bst

Ordered array & advantages

- used a binary search to locate the position where a new item will be inserted with the help of find() method.
- Items should be stored in an ordered manner.
- The major advantage is that search times are much faster than in an unordered array.
- The disadvantage is that insertion and deletion takes longer because all the data items must be moved up to make room.

Ordered array applications

- Ordered arrays are therefore useful in situations in which searches are frequent, but insertions and deletions are not.
- An ordered array might be appropriate for a database of company employees
- A retail store inventory, on the other hand, would not be a good candidate for an ordered array because the frequent insertions and deletions, as items arrived in the store and were sold, would run slowly.

Algorithm Efficiency and Big-O

Section 2.1

What is Algorithm?

An algorithm is a set of instructions to be followed to solve a problem.

- There can be more than one solution (more than one algorithm) to solve a given problem.

- An algorithm can be implemented using different programming languages on different platforms.

Algorithmic Performance

There are *two aspects* of algorithmic performance:

Time

Instructions take time.
How fast does the algorithm perform?
What affects its runtime?

Space

Data structures take space
What kind of data structures can be used?
How does choice of data structure affect the runtime?

We will focus on time:

How to estimate the time required for an algorithm

Analysis of Algorithms

When we analyze algorithms, we should employ mathematical techniques that analyze algorithms independently of *specific implementations, computers, or data*.

To analyze algorithms:

- First, we start to count the number of significant operations in a particular solution to assess its efficiency.
- Then, we will express the efficiency of algorithms using growth functions.

The Execution Time of Algorithms

Each operation in an algorithm (or a program) has a cost.
Each operation takes a certain of time.

`count = count + 1;` → take a certain amount of time, but it is constant

A sequence of operations:

```
count = count + 1;
sum = sum + count;
```

Cost: c1
Cost: c2

Total Cost = c1 + c2

The Execution Time of Algorithms (cont.)

Example: Simple Loop

Code	Cost	Times
`i = 1;`	c1	1
`sum = 0;`	c2	1
`while (i <= n) {`	c3	n+1
`i = i + 1;`	c4	n
`sum = sum + i;`	c5	n
`}`		

Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*c5

➔ The time required for this algorithm is proportional to n

The Execution Time of Algorithms (cont.)

Example: Nested Loop

Code	**Cost**	**Times**
`i=1;`	c1	1
`sum = 0;`	c2	1
`while (i <= n) {`	c3	n+1
`    j=1;`	c4	n
`    while (j <= n) {`	c5	n*(n+1)
`        sum = sum + i;`	c6	n*n
`        j = j + 1;`	c7	n*n
`  }`		
`  i = i +1;`	c8	n
`}`		

Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*(n+1)*c5+n*n*c6+n*n*c7+n*c8

➔ The time required for this algorithm is proportional to n^2

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

Big-O Notation

- The O() can be thought of as an abbreviation of "order of magnitude“ to describe the performance of an Algorithm.
- 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 and how ***n*** is increasing/decreasing.
- Assuming a loop body consists of only 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 . . .

O(*n*) - Linear Growth Rate

- If processing time increases in proportion to the number of inputs *n*, the algorithm grows at a linear rate which is described as O(n) or "a growth rate of order *n*"

```
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
}
```

O(n^2) - 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;
}
```

O(log *n*) Logarithms

- The inverse of raising something to a power is called a logarithm. Here logarithms are used to calculate the number of steps necessary to perform certain operations.
- ***Refer the next slide***
- Example : If the size is 100 requires the growth rate of 6.64.
  - First find log(100) for base 10 and multiply by 3.322(helps to convert base 10 to base 2) or else directly calculate for logarithms base 2.
  - Log(100) = 2 (base 10) ; 2*3.322 = 6.644 (base 2)

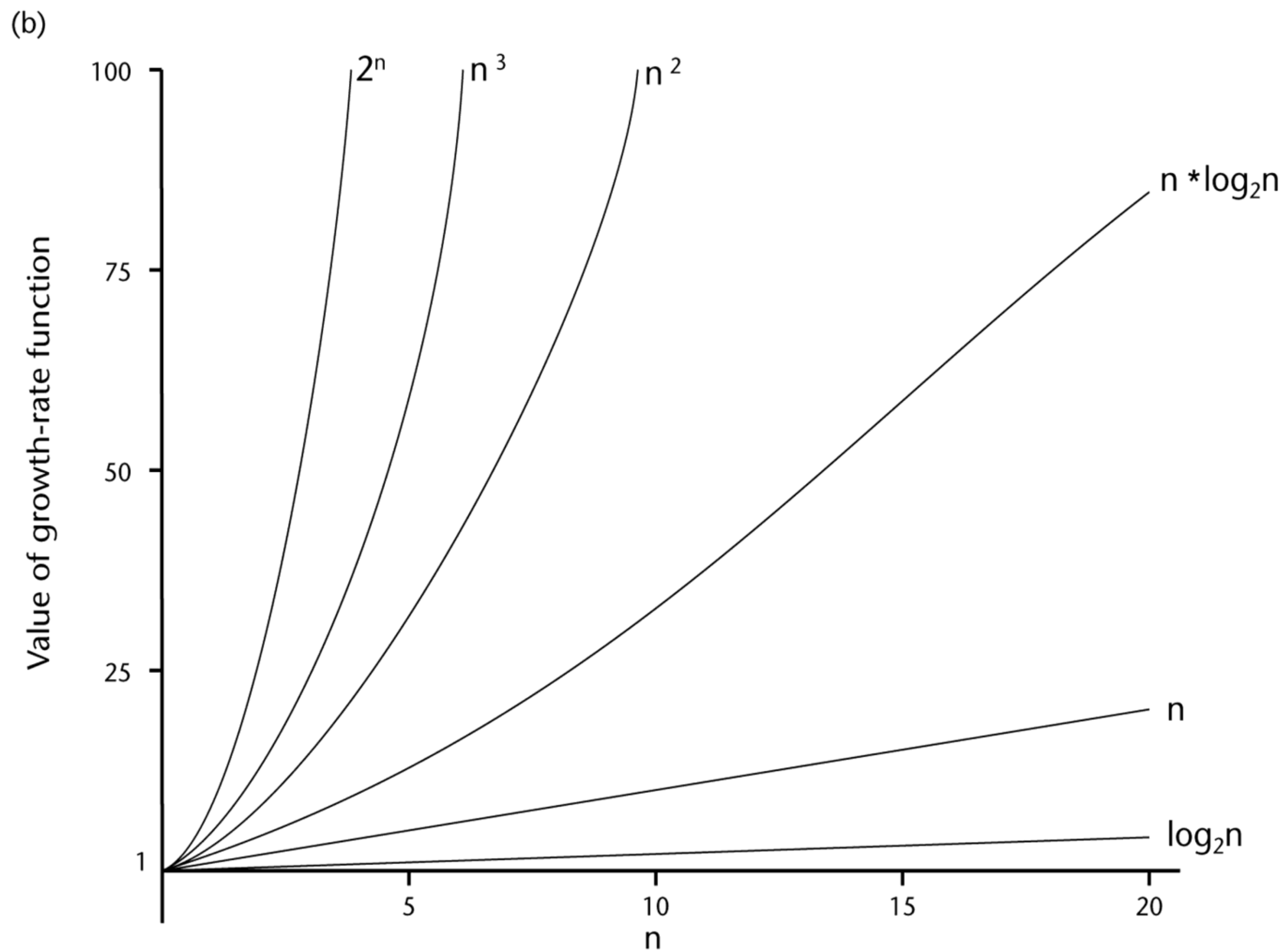
Common Growth Rates

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

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}

Comparison of Growth Rates



What to Analyze

An algorithm can require different times to solve different problems of the same size.

Eg. Searching an item in a list of n elements using sequential search. ➔ Cost: 1,2,...,n

Worst-Case Analysis –The maximum amount of time that an algorithm require to solve a problem of size n.

This gives an upper bound for the time complexity of an algorithm.

Normally, we try to find worst-case behavior of an algorithm.

Best-Case Analysis –The minimum amount of time that an algorithm require to solve a problem of size n.

The best case behavior of an algorithm is NOT so useful.

Average-Case Analysis –The average amount of time that an algorithm require to solve a problem of size n.

Sometimes, it is difficult to find the average-case behavior of an algorithm.

We have to look at all possible data organizations of a given size n, and their distribution probabilities of these organizations.

Worst-case analysis is more common than average-case analysis.

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

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?

1. **Type-safety:** can hold only a single type of objects in generics. It doesn't allow to store other objects.
2. **Type casting is not required:** Avoids the need to downcast from Object.
3. **Compile-Time Checking:** It is checked at compile time so problem will not occur at runtime.

* *More details in week 2*

Day - 2

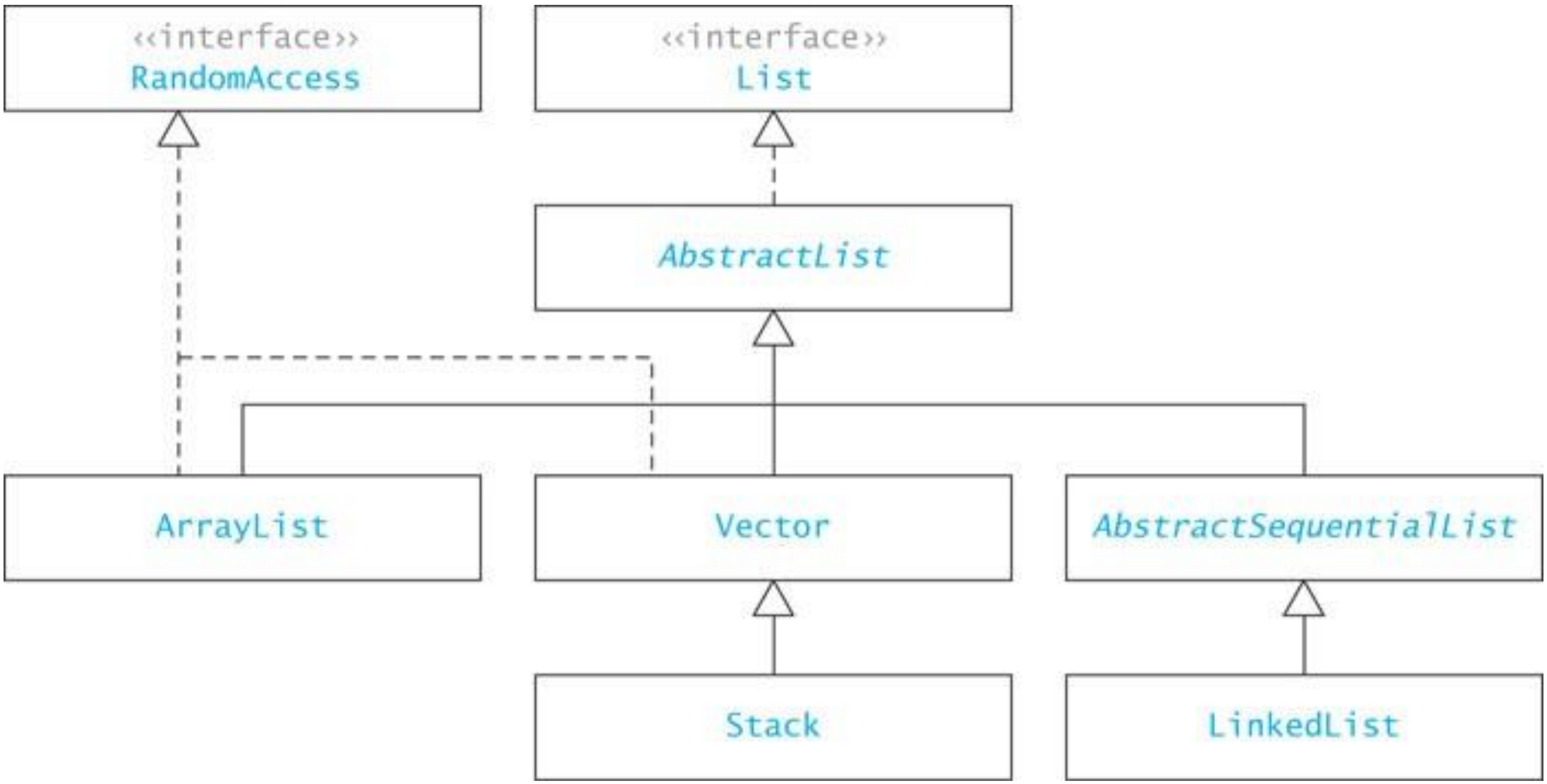
The List Interface and ArrayList Class

Section 2.2

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

`java.util.List` Interface and its Implementers



`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

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>();
```

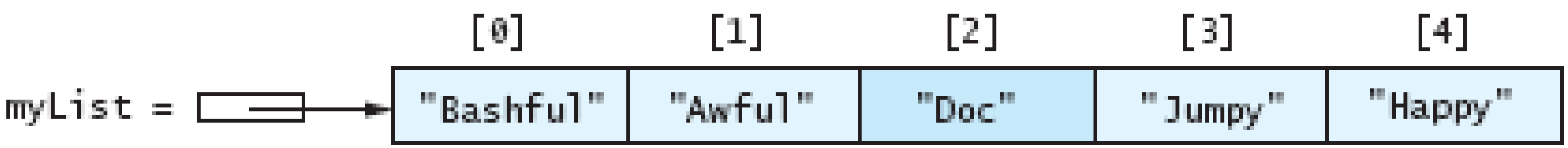
- The initial List is empty and has a default initial capacity of 10 elements
- To add strings to the list,

```
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");
```



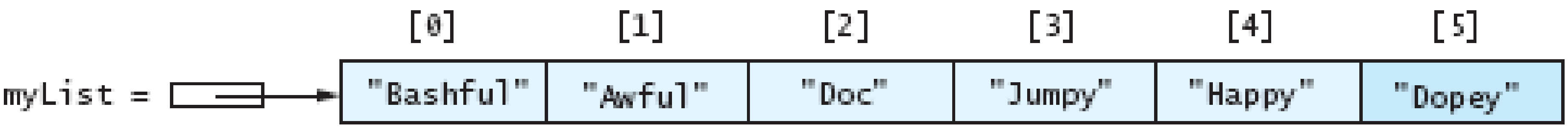
After insertion of "Doc" before the third element

- 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:

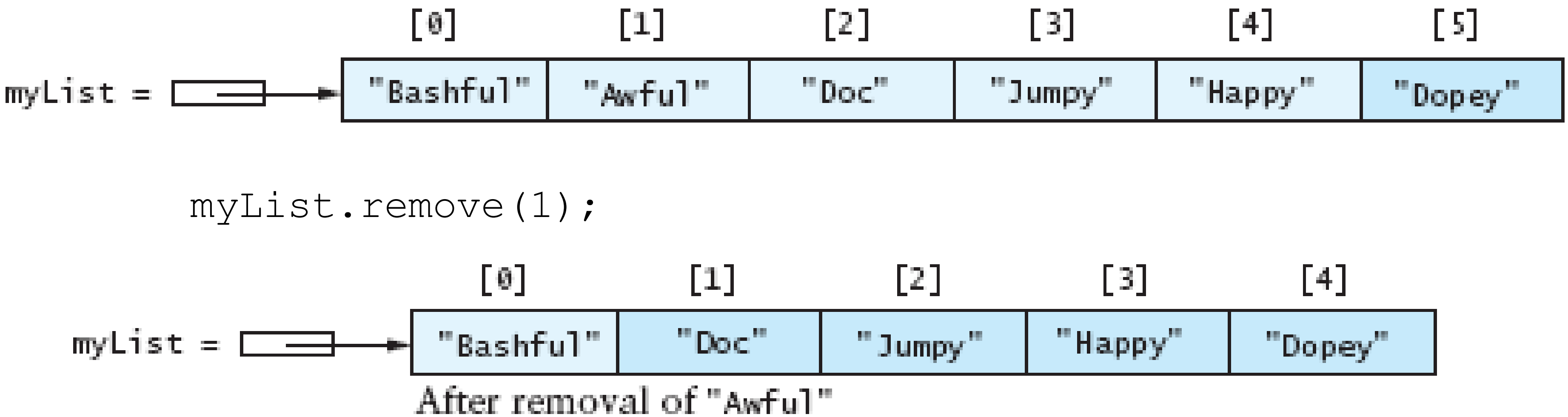
```
myList.add("Dopey");
```



After insertion of "Dopey" at the end

`ArrayList` Class (cont.)

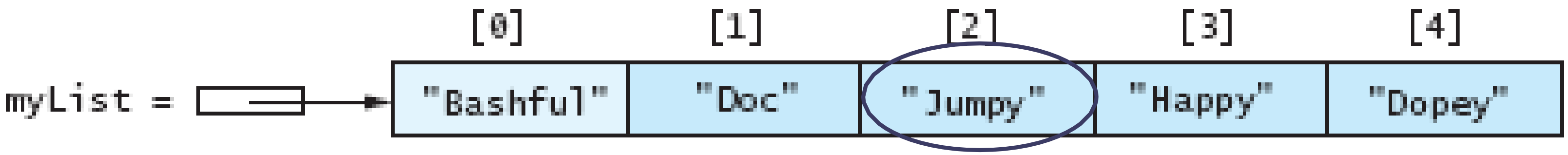
- Removing an element:



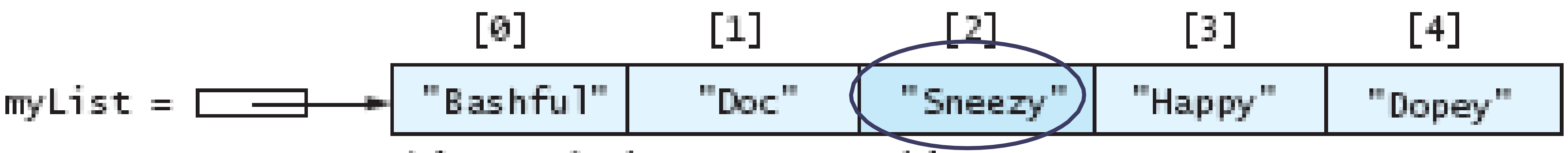
- The strings referenced by [2] to [5] have changed to [1] to [4]

`ArrayList` Class (cont.)

- You may also replace an element:

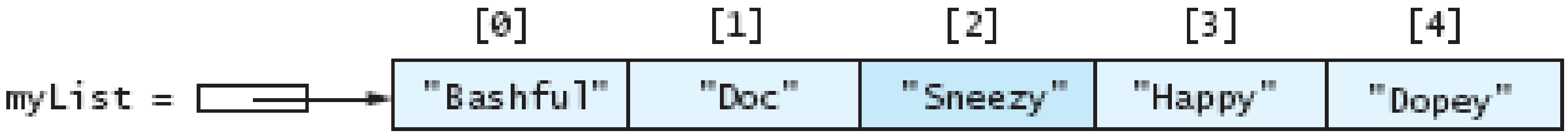



```
myList.set(2, "Sneezy");
```



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

Applications of `ArrayList`

Section 2.3

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.size();
  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;
}

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

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

Add a `DirectoryEntry` object

Phone Directory Application (cont.)

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

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

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

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

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

Phone Directory Application (cont.)

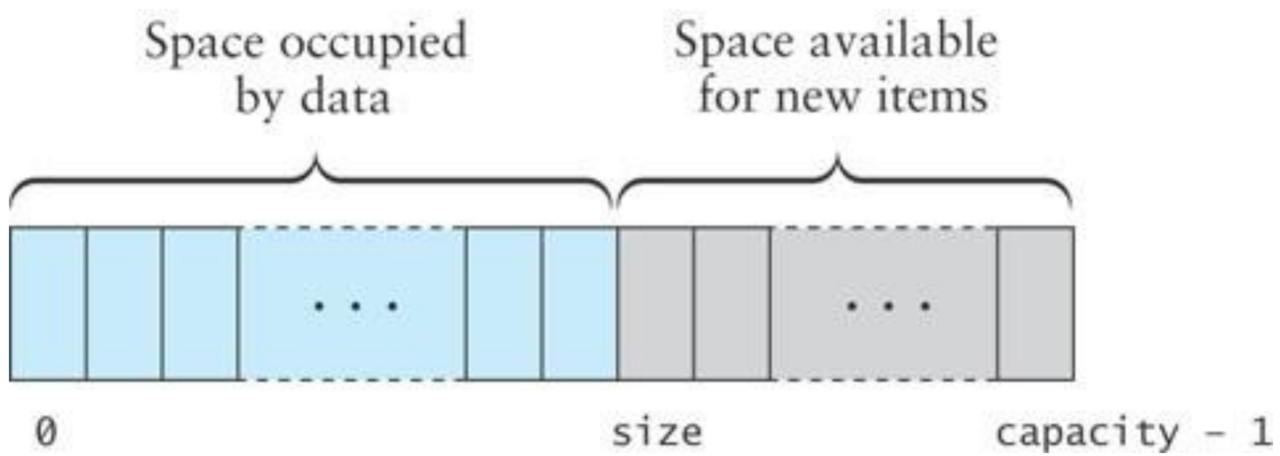
```
public class DirectoryEntry {
  String name;
  String number;
}

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 2.4

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[]

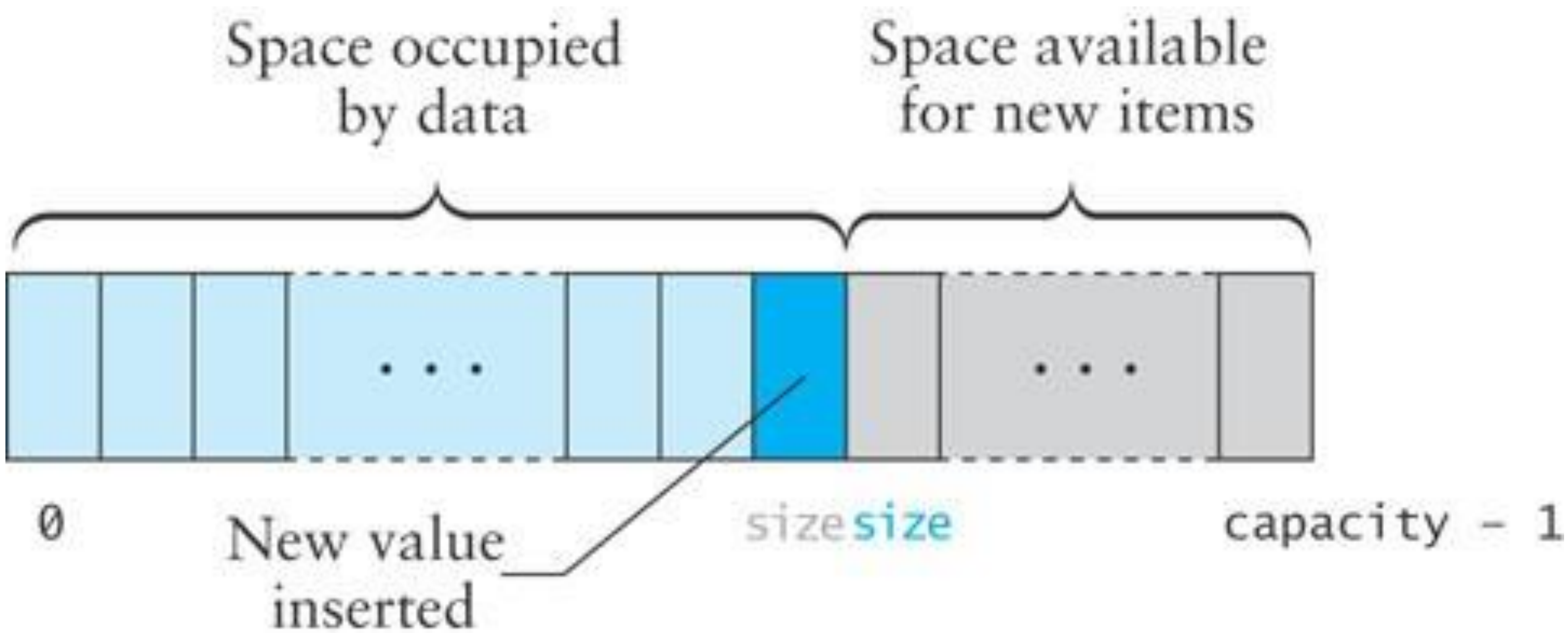
Although this may cause a compiler warning, it's ok

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

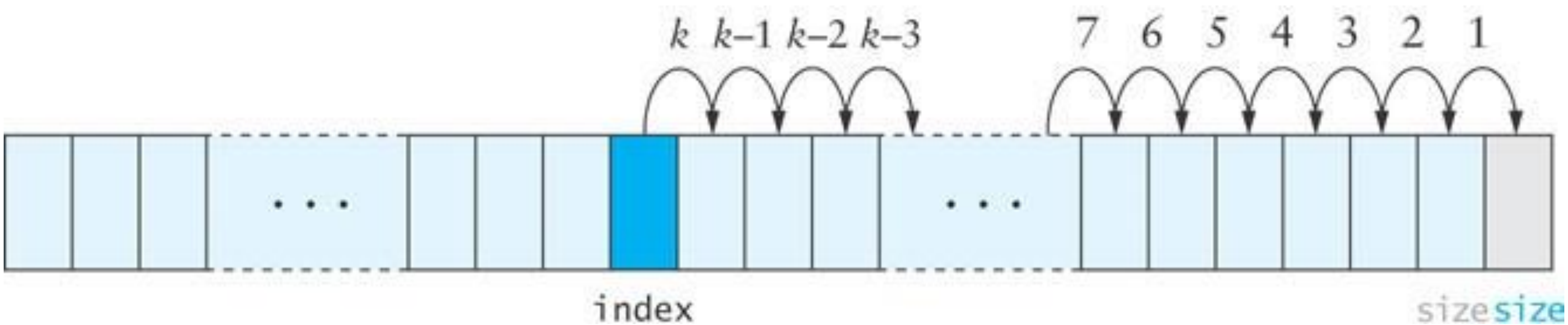
Implementing `ArrayList.add(E)` (cont.)

- 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



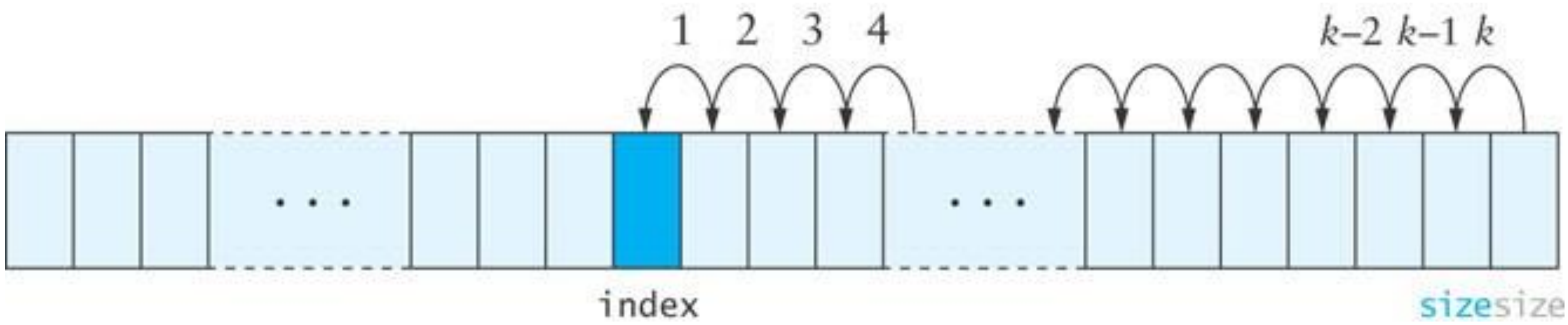
Implementing `ArrayList.add(index,E)`

```
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;
The **java.util.Arrays.copyOf(int[] original,int newLength)** method copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.

`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;

- See Demo code : w1l2 package – arraylist and arraylistapi (sub package)

`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

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.

Day – 3 & 4

Single-Linked Lists

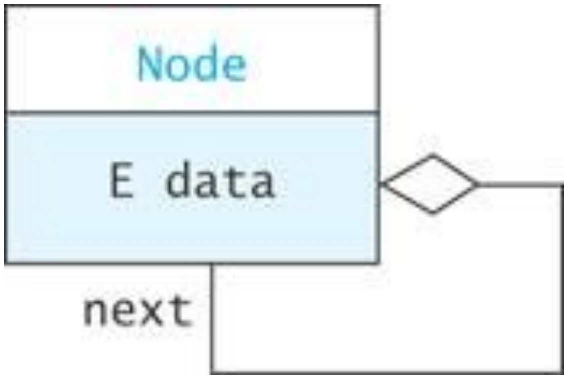
Section 2.5

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 (cont.)

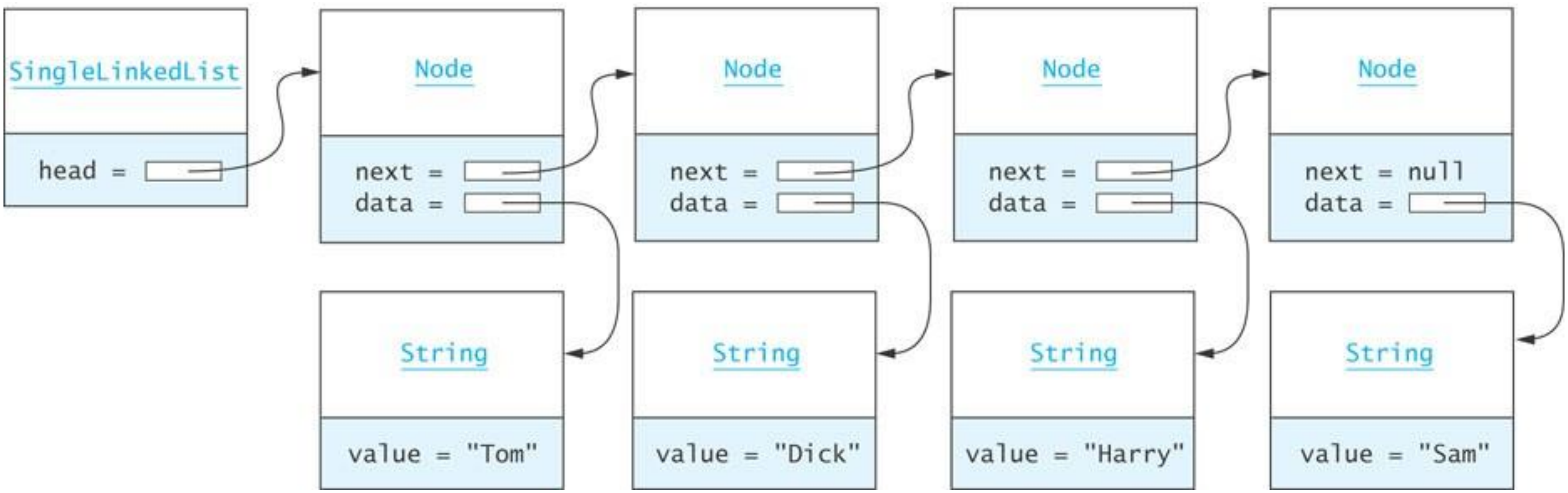
```
private static class Node<E> {
  private E data;
  private Node<E> next;

  /** 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;
  }
}
```

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

Static inner classes are also called *nested classes*

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

Connecting Nodes



Connecting Nodes (cont.)

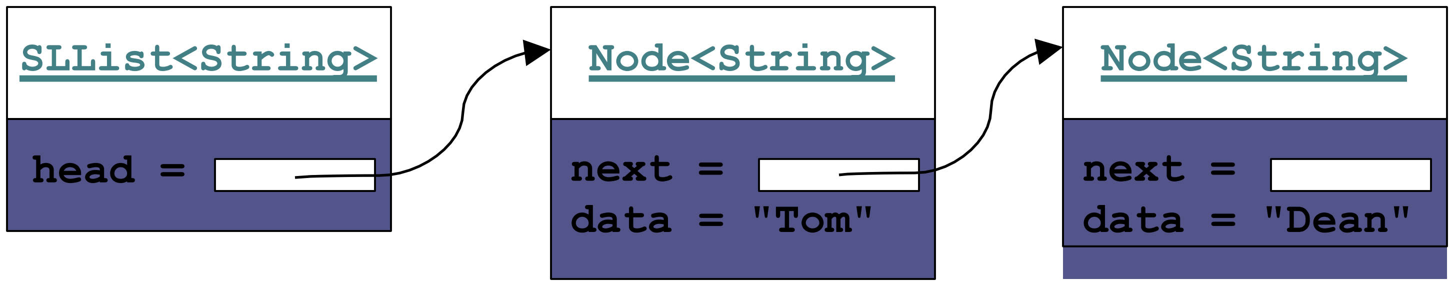
```
Node<String> tom = new Node<String>("Tom");
Node<String> dean = new Node<String>("Dean");
Node<String> harry = new Node<String>("Harry");
Node<String> sam = new Node<String>("Sam");

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

A Single-Linked List Class

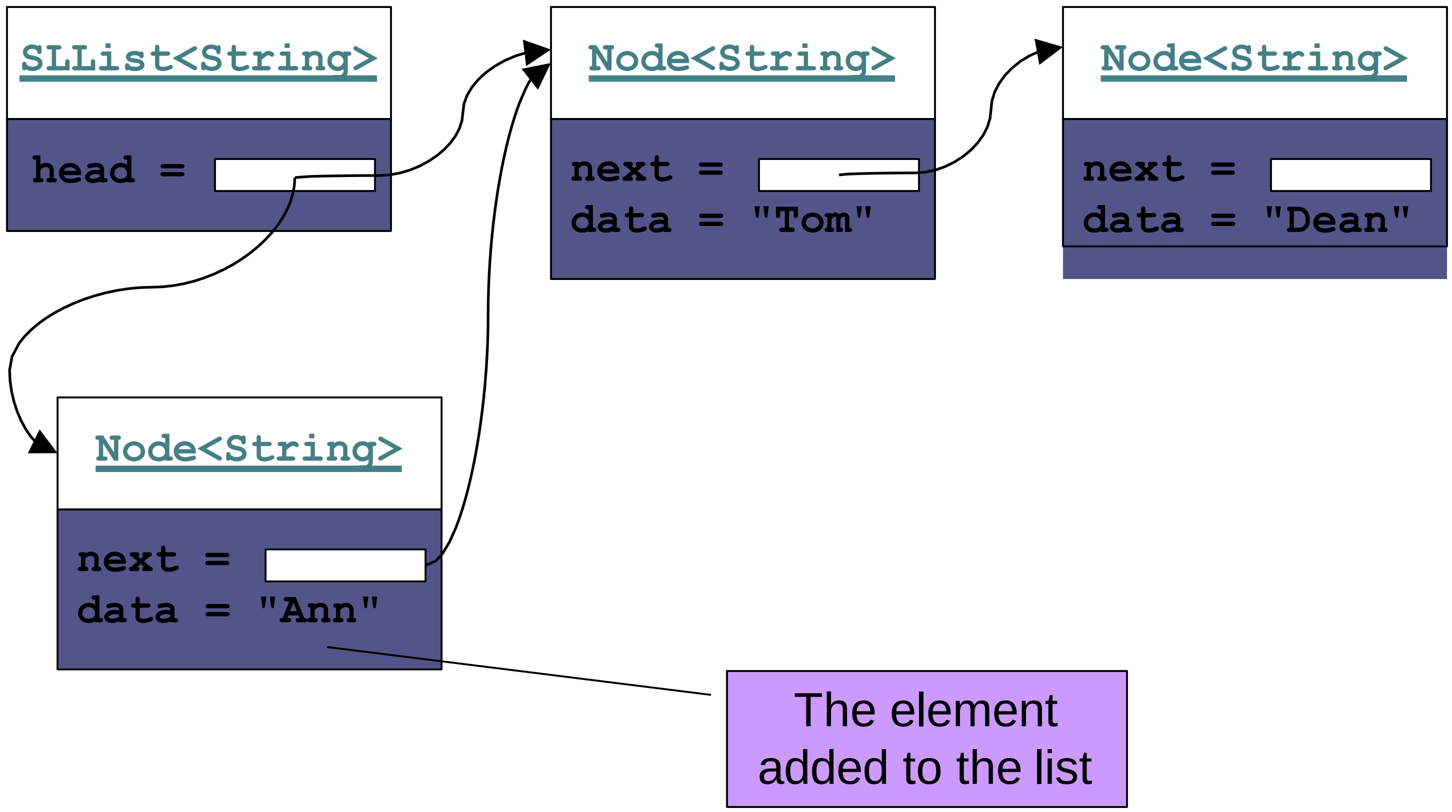
- Generally, we do not have individual references to each node.
- A `SingleLinkedList` object has a data field `head`, the *list head*, which references the first list node

```
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.)

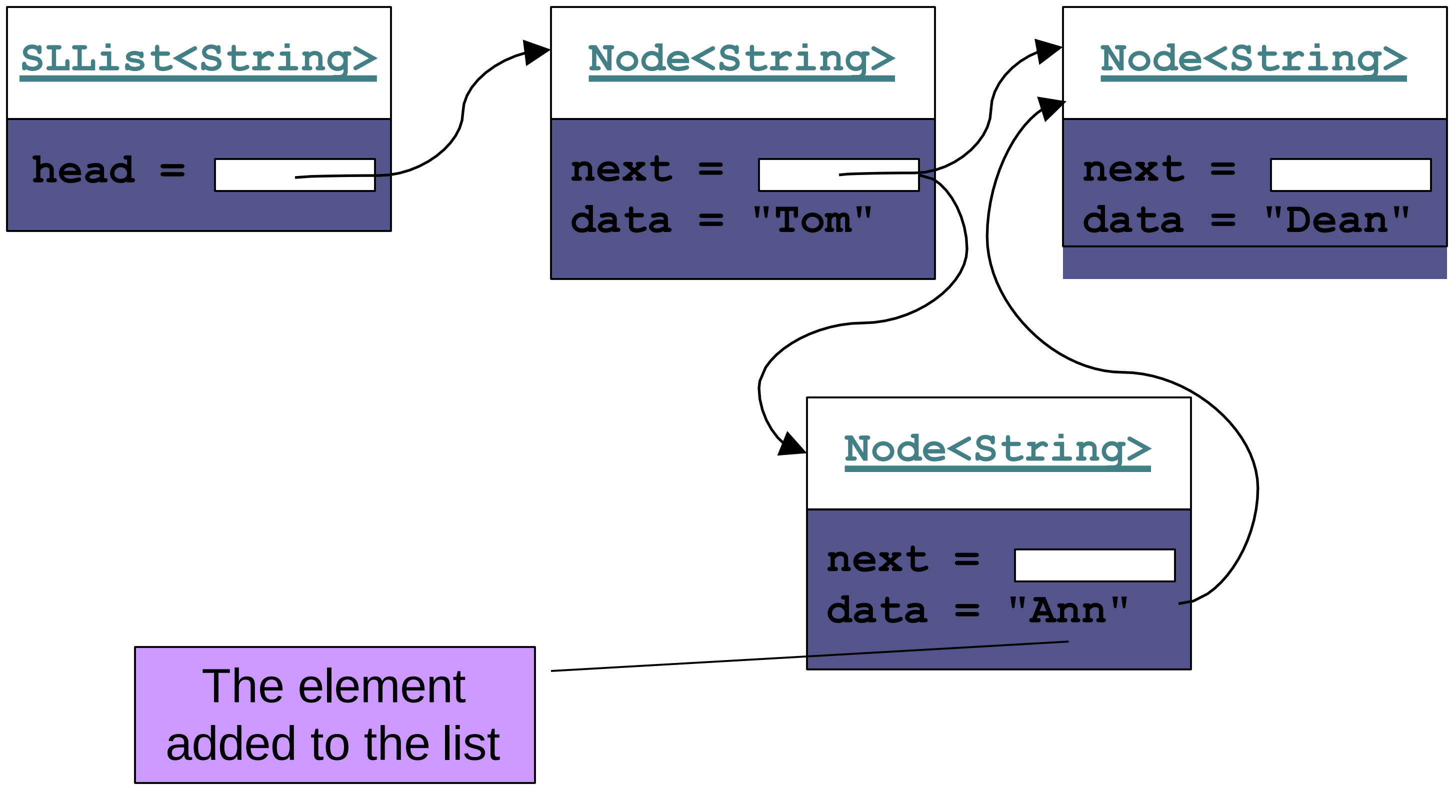
```
private void addFirst (E item) {
  Node<E> temp = new Node<E>(item, head);
  head = temp;
  size++;
}
```

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.)

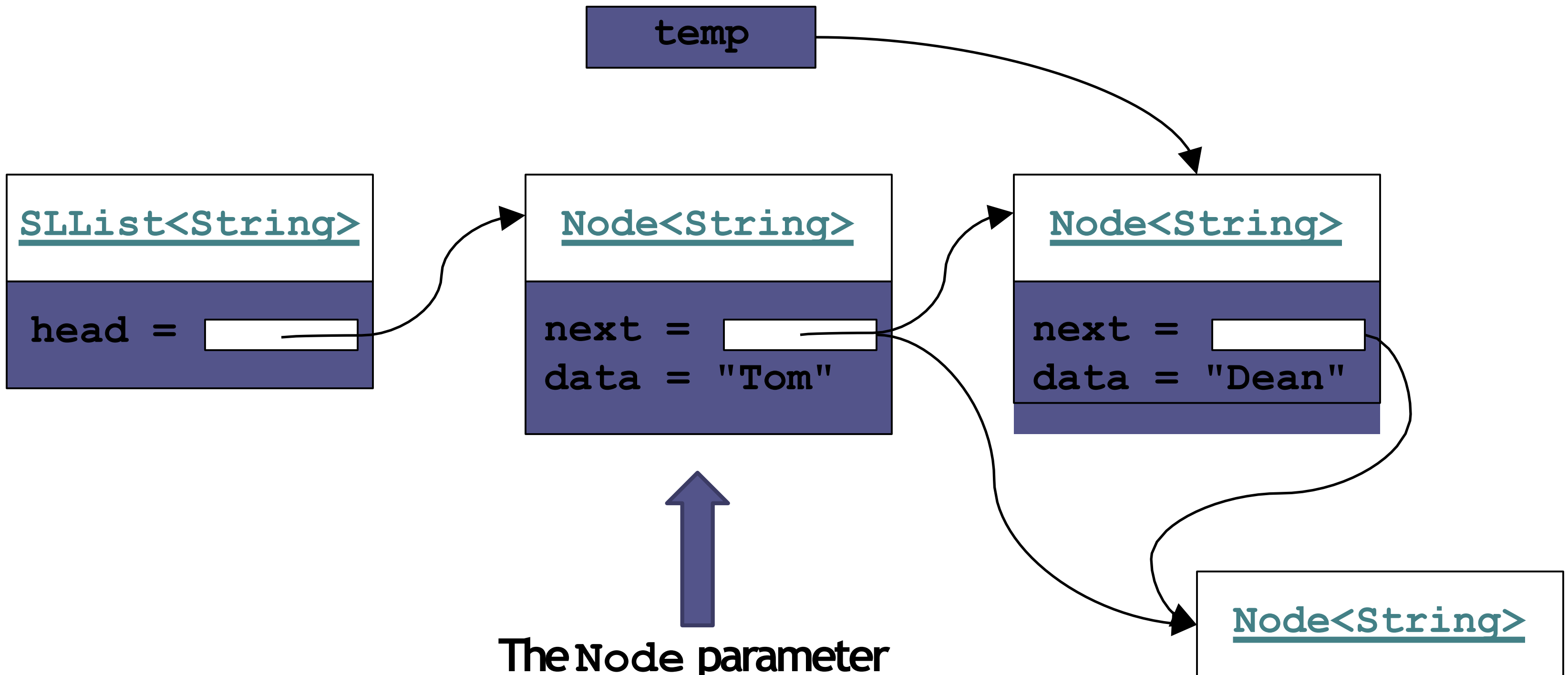
```
private void addAfter (Node<E> node, E item) {
  Node<E> temp = new Node<E>(item, node.next);
  node.next = temp;
  size++;
}
```

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++;
}
```

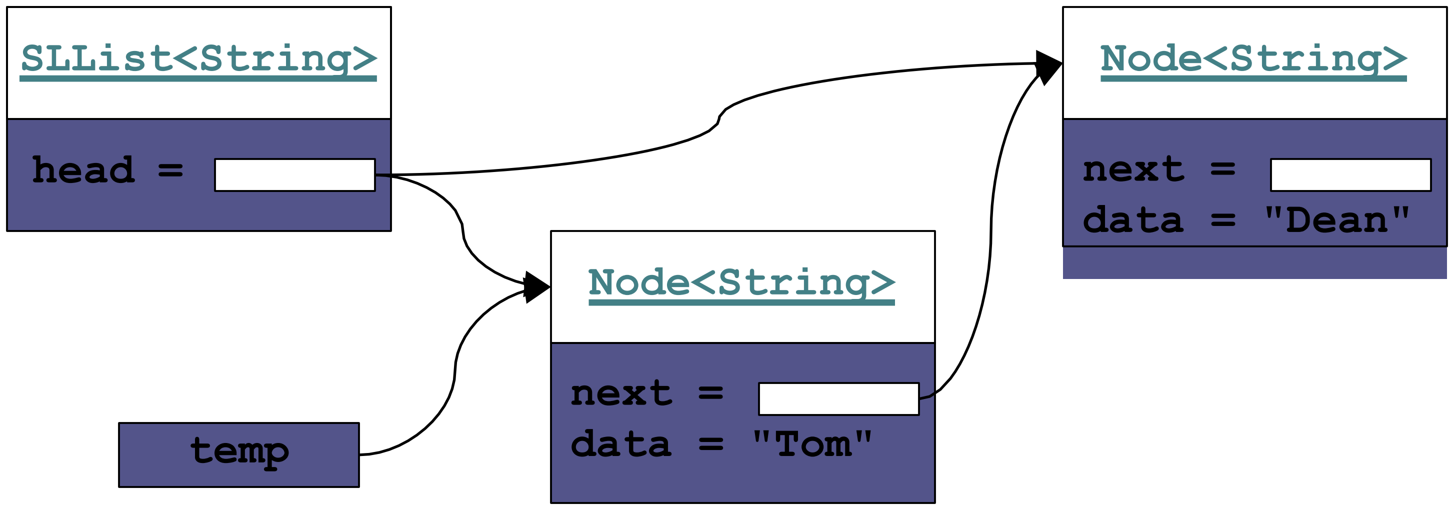
Implementing `removeAfter(Node<E> node)`



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;
  }
}
```

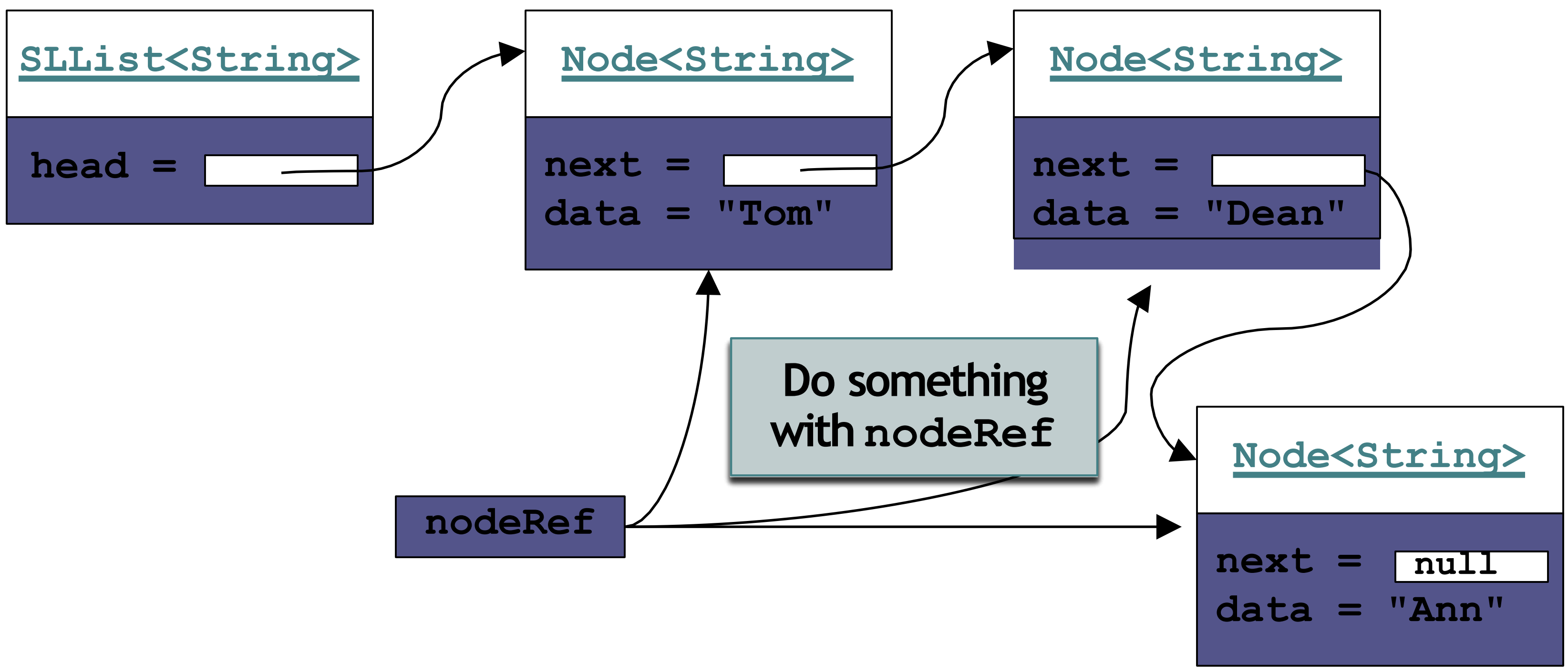
Implementing `SLList.removeFirst()`



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

```
private E removeFirst () {
  Node<E> temp = head;
  if (head != null) {
    head = head.next;
    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 = newValue;
  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;
}
```

- Demo : SingleLinkedList.java

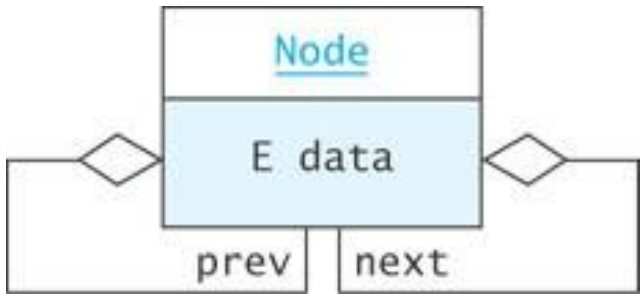
Double-Linked Lists and Circular Lists

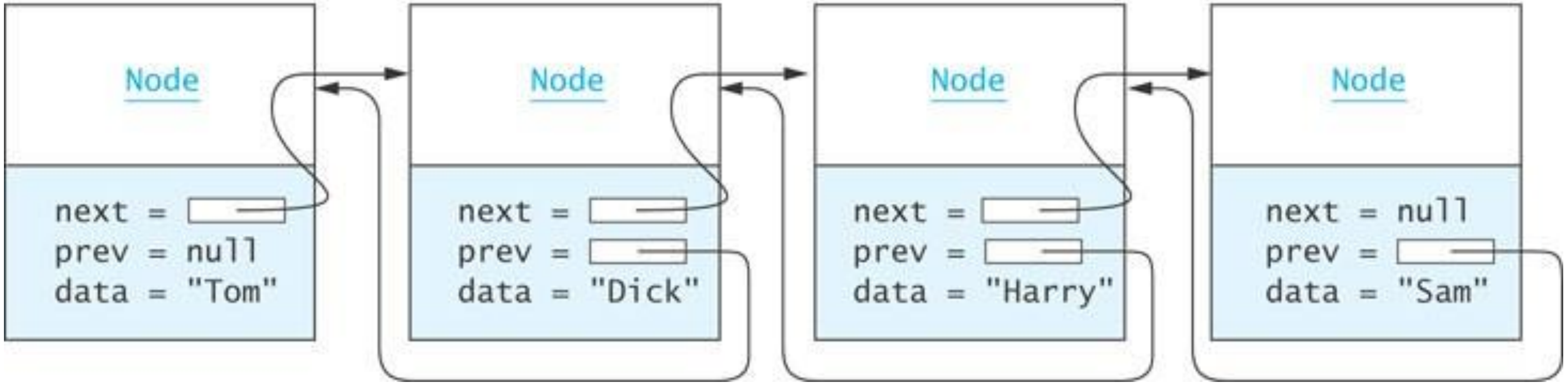
Section 2.6

Double-Linked Lists

- Limitations of a singly-linked list include:
  - 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*

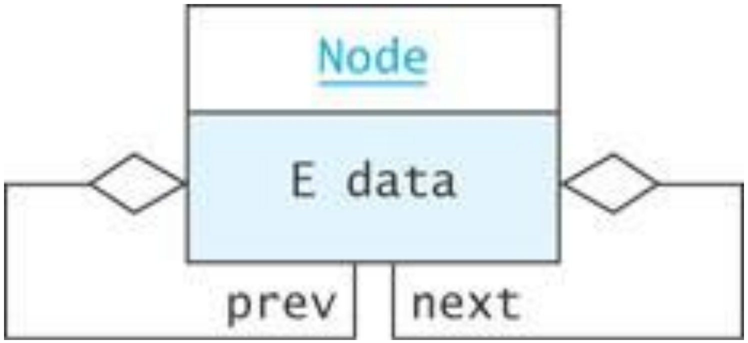
Double-Linked Lists (cont.)



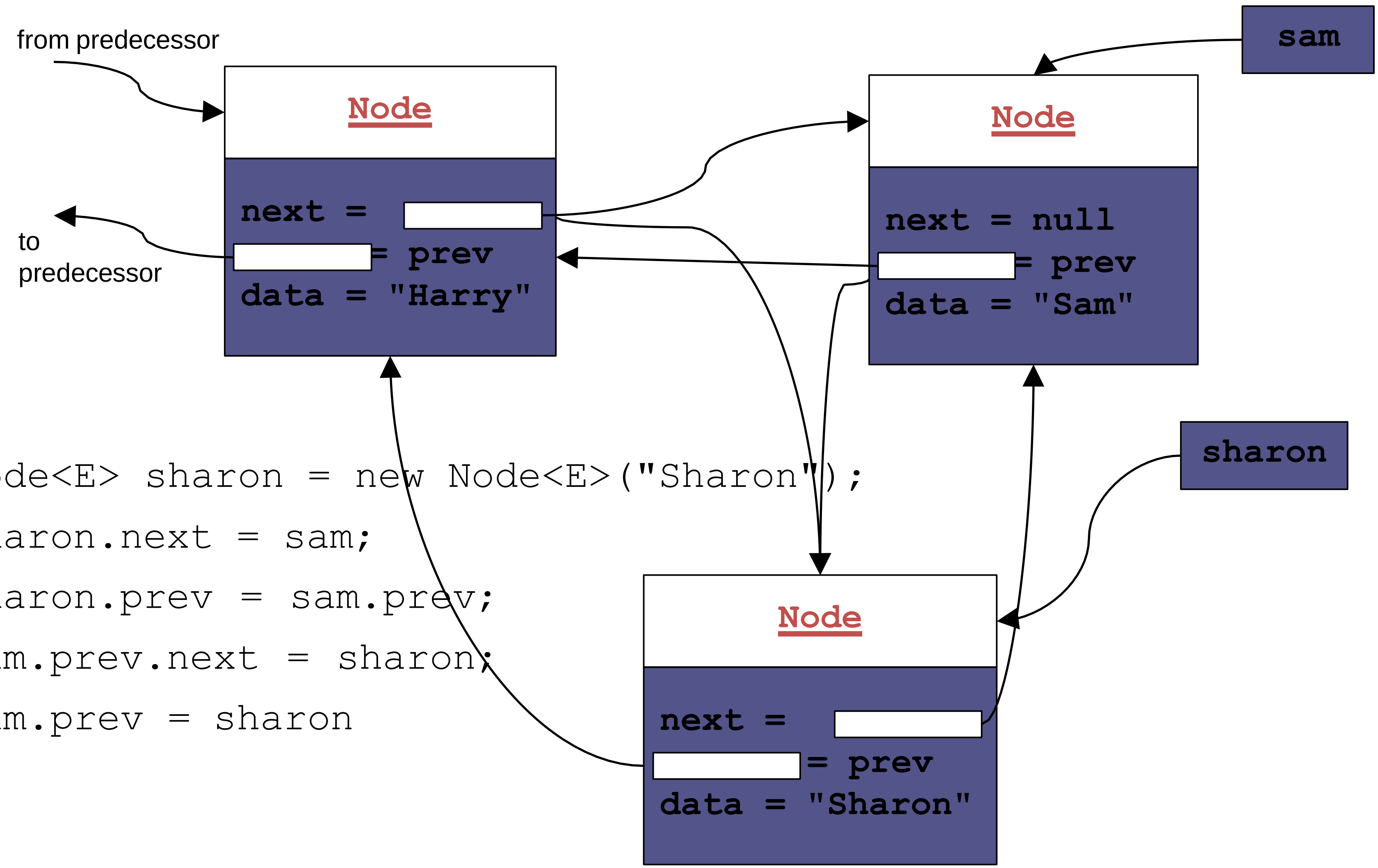


Node Class

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

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

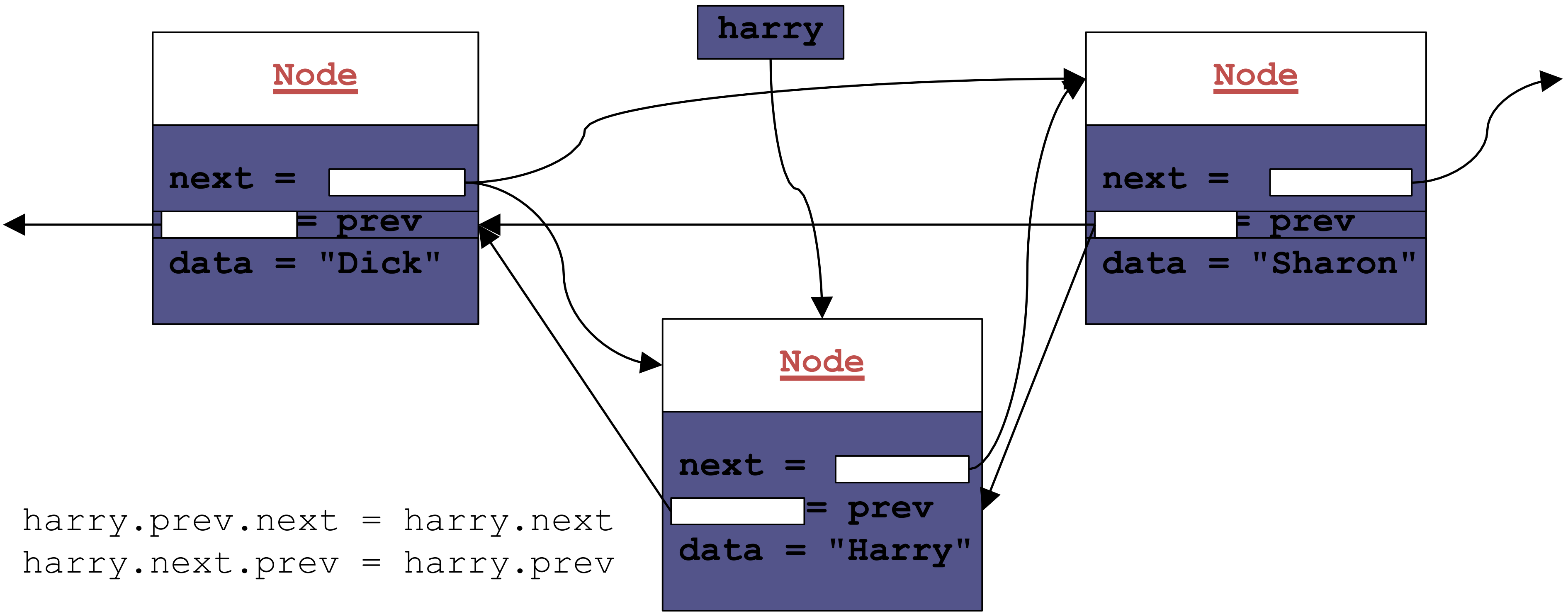


Inserting into a Double-Linked List




```
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

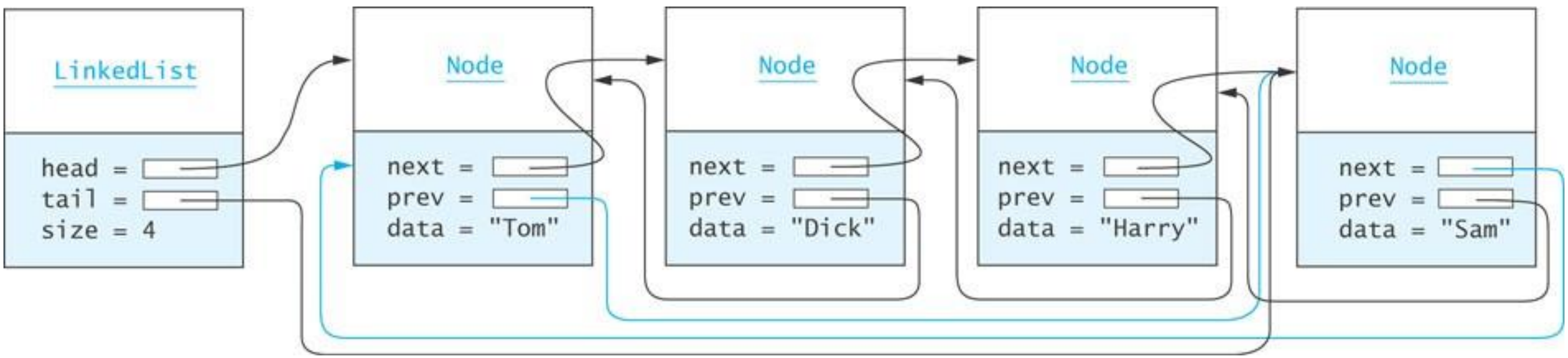
- So far we have worked only with internal nodes
- As with the single-linked class, it is best to access the internal nodes with a double-linked list object

- 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*)
- Demo : MyStringLinkedList.java, DoublyLinkedApp.java

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!
- We are not discussing the implementation of Circular Linked List

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.

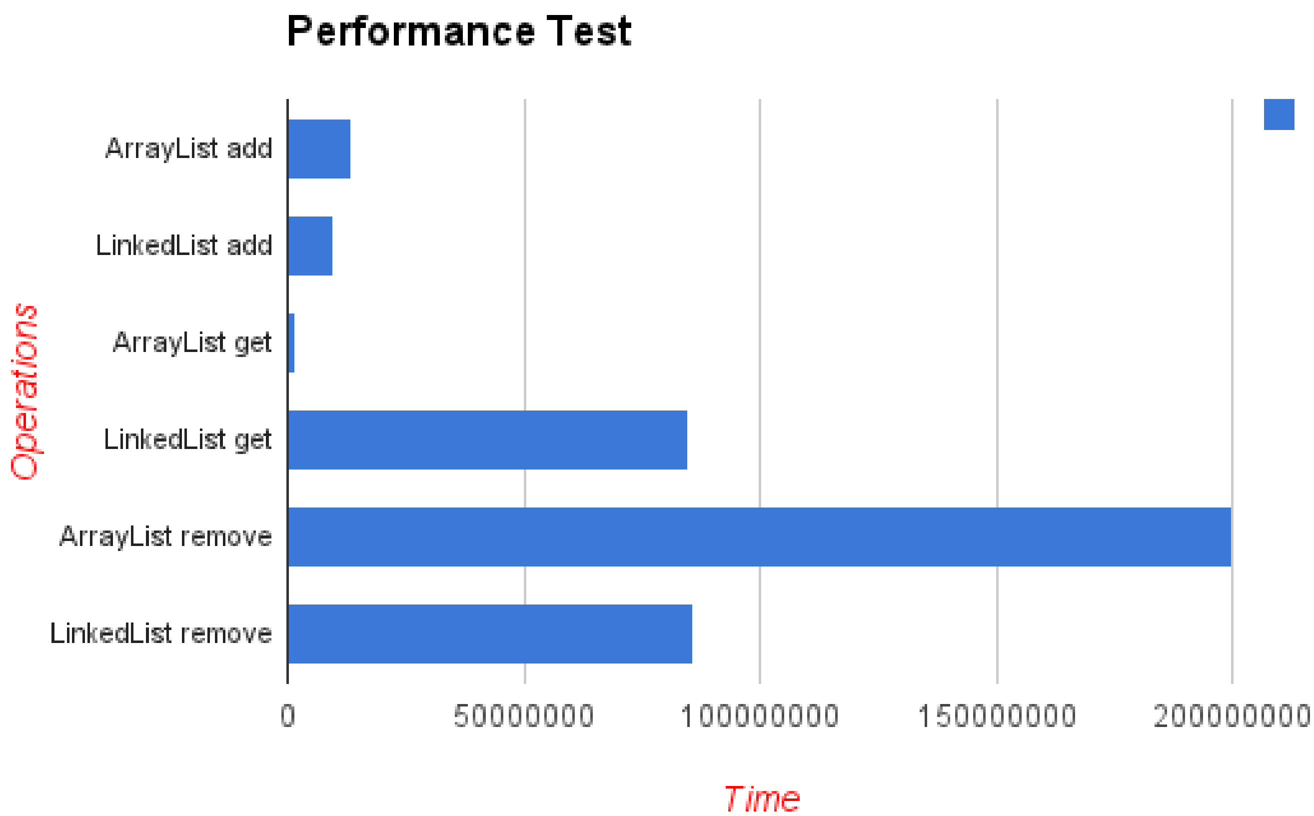
Java Collection Framework

- **Creation of Linked List**

```
LinkedList <Integer>list = new LinkedList<Integer>();
list.add(10);
list.add(20);
int   size    =  list.size();
```

ListMethods.doc

When to use LinkedList?



Day - 5

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)

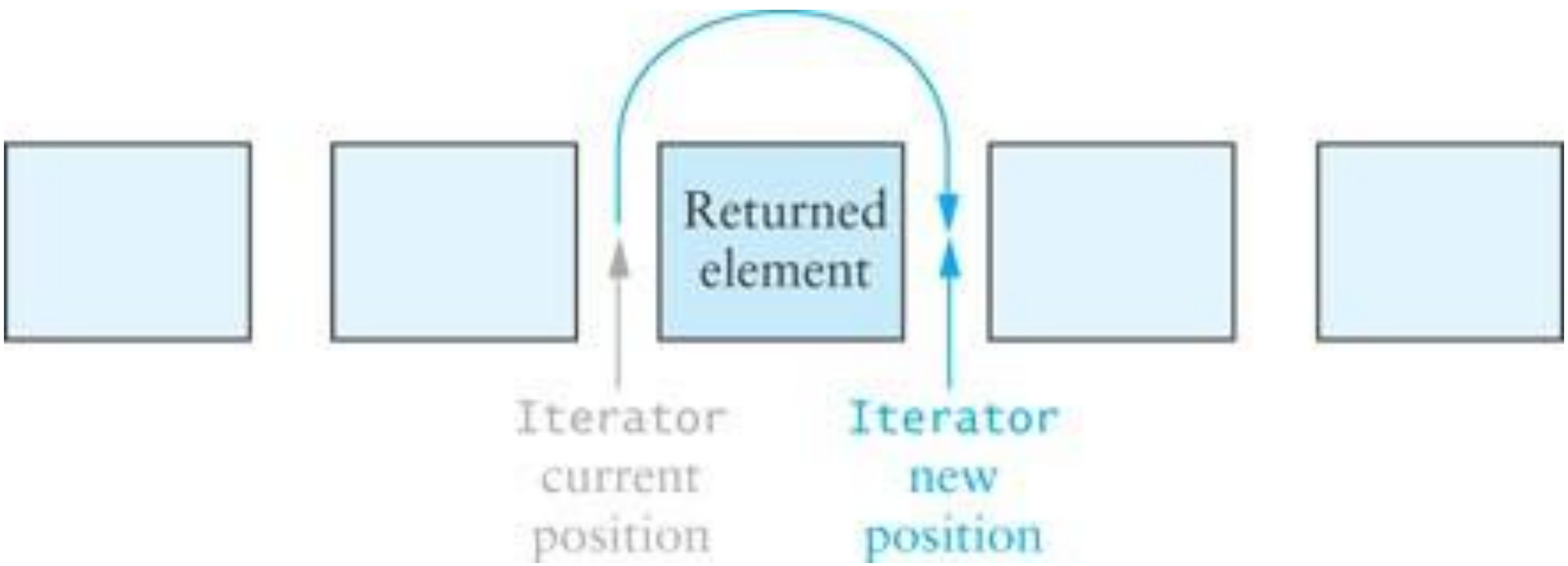
`Iterator` Interface

- 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.
  - `Iterator.remove` removes items without starting over at the beginning.

Iterators and Removing Elements (cont.)

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

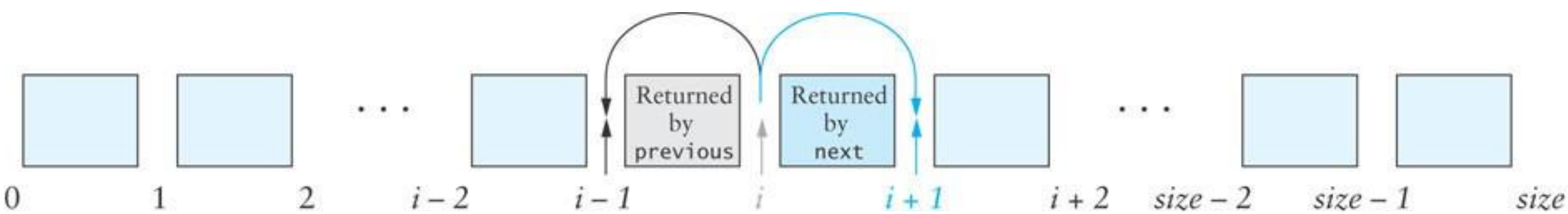
```
public static void removeDivisibleBy(LinkedList<Integer>
                                           aList, int div) {
  Iterator<Integer> iter = aList.iterator();
  while (iter.hasNext()) {
    int nextInt = iter.next();
    if (nextInt % div == 0) {
      iter.remove();
    }
  }
}
```

`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`.

Refer : DemoCode\w1l2\linkedlist\api

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();
}
```

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 Collection Framework Design

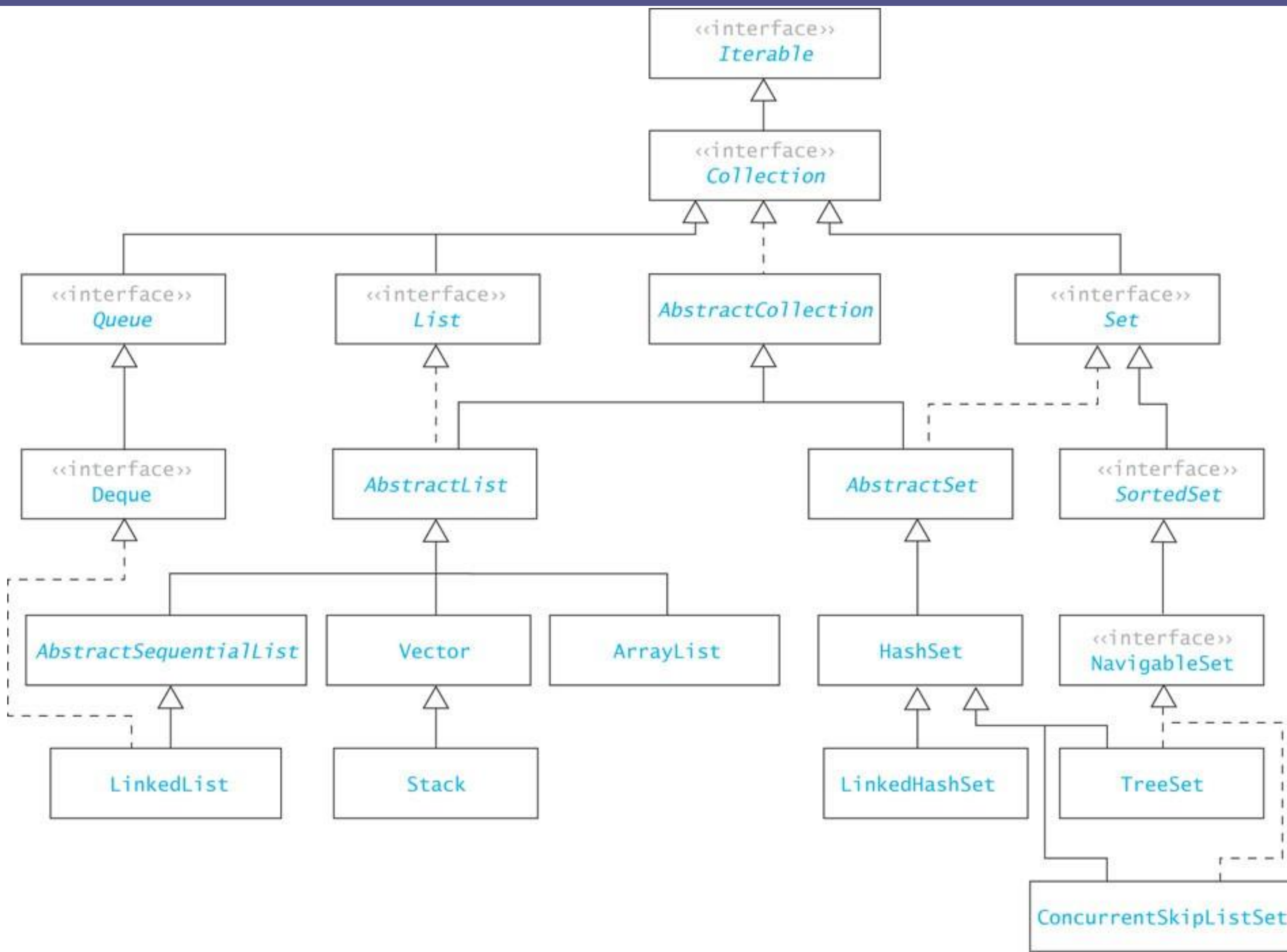
Section 2.9

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 Collection

- Collection
  - 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 Collection (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(can't use index for accessing)

- 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

COMPARABLE AND COMPARATOR INTERFACE

Comparable Inteface

- *The* **Comparable** *interface defines the* **compareTo** *method for comparing objects.*

```
package java.lang;
public interface Comparable<E>{
public int compareTo(E o);
}
```

- The **compareTo** method determines the order of this object with the specified object **o** and returns a negative integer, zero, or a positive integer if this object is less than, equal to, or greater than object **o**.
- Refer : DemoCode\ w1l2\ sorting\ComparableDemo.java

Comparator Interface

- **Comparator** *can be used to compare the objects of a class that doesn’t implement* **Comparable.**
- To do so, define a class that implements the **java.util.Comparator<T>** interface.
- The **Comparator<T>** interface has two methods, **compare** and **equals.**

public int compare(T element1, T element2)

Returns a negative value if **element1** is less than **element2,**
a positive value if **element1** is greater than **element2,**
and zero if they are equal.

Refer : DemoCode\w1l2\ sorting\ArrayListSort.java, TestSorting.java

Difference between Comparable and Comparator

- If any class implements **comparable** interface then collection of that object can be sorted automatically using Collection.sort() or Arrays.sort().Object will be sort on the basis of compareTo method in that class.
- Using **Comparator** interface, we can write different sorting based on different attributes of objects to be sorted. You can use anonymous comparator to compare at particular line of code or other class can implement this interface to sort.

  public void sort(Collection obj,Comparator c): is used to sort the elements of List by the given comparator.

Sorting

- To accomplish this, you specify your own ordering on a class using the **Comparator** interface, whose only method is **compare()**. Like lists, in j2se5.0, Comparators are parameterized.

- The compare() method is expected to behave in the following way (so it can be used in conjunction with the Collections API):

For objects a and b,

- compare(a,b) returns a negative number if a is “less than” b
- compare(a,b) returns a positive number if a is “greater than” b
- compare(a,b) returns 0 if a “equals” b

Sorting (Consist with equals)

- If compare is not used in a "sensible" way, it will lead to unexpected results when used by utilities like `Collections.sort`.

<u>The compare contract</u> It must be true that:

- a is "less than" b if and only if b is "greater than" a
- if a is "less than" b and b is "less than" c, then a must be "less than" c.

It *should* also be true that the `Comparator` is *consistent with equals*; in other words:

- `compare(a,b) == 0` if and only if `a.equals(b)`

If a `Comparator` is not consistent with equals, problems can arise when using different container classes. For instance, the `contains` method of a Java `List` uses equals to decide if an object is in a list. However, containers that maintain the order relationship among elements (like `TreeSet` – more on this one later) check whether the output of compare is 0 to implement `contains`.

- refer : DemoCode\ w1l2\ sorting\employee