Chapter 5

The Collection ADT

OBJECT-ORIENTED
DATA STRUCTURES
USING

Java™

FOURTH EDITION

NELL DALE
DANIEL T. JOYCE
CHIP WEEMS

Chapter 5: The Collection ADT

5.1 The Collection Interface

- Stacks and queues restrict access to data based on the order in which the data was stored

- Sometimes we need to retrieve information regardless of the order in which it is stored—content based access—for example obtaining student information based on an ID number

- the Collection ADT is the most basic ADT that provides the required functionality

Assumptions for our Collections

- Support addition, removal, and retrieval of elements

- Content based access is based on equality of objects (using the equals method)

- Allow duplicate elements, do not allow `null` elements

- `add` and `remove` operations return a `boolean` indicating success

```java
package ch05.collections;
public interface CollectionInterface<T>
{
  boolean add(T element);
  // Attempts to add element to this collection. Returns true if successful, false otherwise.

  T get(T target);
  // Returns an element e from this collection such that e.equals(target).
  // If no such element exists, returns null.

  boolean contains(T target);
  // Returns true if this collection contains an element e such that
  // e.equals(target); otherwise returns false.

  boolean remove (T target);
  // Removes an element e from this collection such that e.equals(target)
  // and returns true. If no such element exists, returns false.

  boolean isFull();
  boolean isEmpty();
  int size();
}
```

5.2 Array-Based Collection Implementation

- Basic Approach - a collection of N elements is held in the first N locations of an array, locations 0 to N-1

- Maintain an instance variable, `numElements`

numElements: 4

	[0]	[1]	[2]	[3]	[4]	[5]
elements:	"BRA"	"MEX"	"PER"	"ARG"		

Beginning of class

```
package ch05.collections;
public class ArrayCollection<T> implements CollectionInterface<T>
{
  protected final int DEFCAP = 100; // default capacity
  protected T[] elements;            // array to hold collection's elements
  protected int numElements = 0;     // number of elements

  // set by find method
  protected boolean found;  // true if target found, otherwise false
  protected int location;   // indicates location of target if found

  public ArrayCollection()
  {
    elements = (T[]) new Object[DEFCAP];
  }

  public ArrayCollection(int capacity)
  {
    elements = (T[]) new Object[capacity];
  }
. . .
```

Adding an element

	[0]	[1]	[2]	[3]	[4]	[5]
elements:	"BRA"	"MEX"	"PER"	"ARG"		

Add "COL"

numElements: 4̸ 5

	[0]	[1]	[2]	[3]	[4]	[5]
elements:	"BRA"	"MEX"	"PER"	"ARG"	"COL"	

The add method

```
public boolean add(T element)
// Attempts to add element to this collection.
// Returns true if successful, false otherwise.
{
    if (isFull())
        return false;
    else
    {
        elements[numElements] = element;
        numElements++;
        return true;
    }
}
```

The helper `find` method

- protected access

- uses Sequential Search … O(N)

- sets instance variables `found` and `location`

- simplifies
  - remove
  - contains
  - get

The helper `find` method

```
protected void find(T target)
{
    location = 0;
    found = false;
    while (location < numElements)
    {
        if (elements[location].equals(target))
        {
            found = true;
            return;
        }
        else
            location++;
    }
}
```
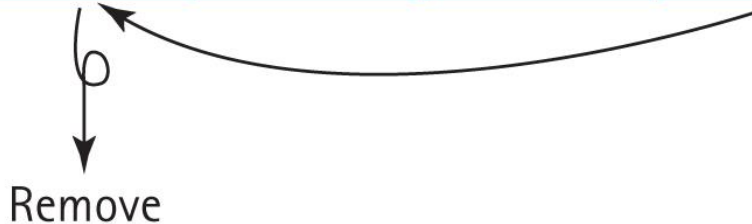
Removing an element

numElements: ~~4~~ 5

	[0]	[1]	[2]	[3]	[4]	[5]
elements:	"BRA"	"MEX"	"PER"	"ARG"	"COL"	

Remove "MEX"

numElements: ~~4~~ ~~5~~ 4

	[0]	[1]	[2]	[3]	[4]	[5]
elements:	"BRA"	"MEX"	"PER"	"ARG"	"COL"	

Remove

The `remove` method

```
public boolean remove (T target)
// Removes an element e from this collection
// such that e.equals(target) and returns true;
// if no such element exists, returns false.
{
    find(target);
    if (found)
    {
        elements[location] = elements[numElements - 1];
        elements[numElements - 1] = null;
        numElements--;
    }
    return found;
}
```

The `contains` and `get` methods

```
public boolean contains (T target)
{
    find(target);
    return found;
}


public T get(T target)
{
    find(target);
    if (found)
        return elements[location];
    else
        return null;
}
```

5.3 Application: Vocabulary Density

- The vocabulary density of a text is the total number of words in the text divided by the number of unique words in the text.

- Used by computational linguists to help analyze texts.

- We define a word to be a sequence of letter characters (A through Z) plus the apostrophe character (').

Vocabulary Density Calculation

```
Initialize variables and objects
while there are more words to process
    Get the next word
    if the collection does not contain the word
        Add the word to the collection
    Increment total number of words
Display (total number words/size of the collection)
```

The application

- Instructors can now review the application, `VocabularyDensity`, found in the `ch05.apps` package, plus the notes on pages 307-308.

- The code is short and simple—this is because so much of the complexity is handled by the `ArrayCollection` class. This is a good example of the power and utility of abstraction.
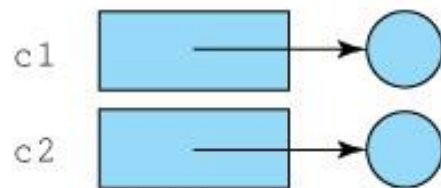
5.4 Comparing Objects Revisited

- Collection operations require comparing objects for equality

- Section 5.5 "Sorted Array-Based Collection Implementation" requires comparing objects for order

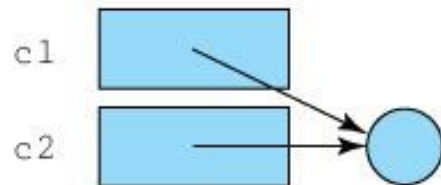- This section reviews the `equals` and `compareTo` operations

Using the comparison (==) operator

intA 10
intB 10
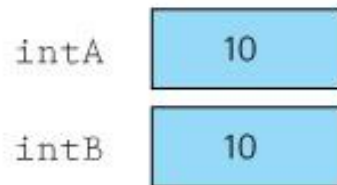
"intA == intB" evaluates to true

c1
c2

"c1 == c2" evaluates to false

c1
c2

"c1 == c2" evaluates to true

Using the `equals` method

- Since `equals` is exported from the `Object` class it can be used with objects of any Java class.

- For example, If `c1` and `c2` are objects of the class `Circle`, then we can compare them using

      `c1.equals(c2)`

- But this method, as defined in the `Object` class, acts much the same as the comparison operator. It returns true if and only if the two variables reference the same object.

- However, we can redefine the `equals` method to fit the goals of the class.

Defining an `equals` method

- A reasonable definition for equality of Circle objects is that they are equal if they have equal radii.

- To realize this approach we define the equals method of our `Circle` class to use the `radius` attribute:

```
@Override
public boolean equals(Object obj)
{
    if (obj == this)
        return true;
    else
    if (obj == null || obj.getClass() != this.getClass())
        return false;
    else
    {
        Circle c = (Circle) obj;
        return (this.radius == c.radius);
    }
}
```

The `FamousPerson` class

- A famous person object has attributes representing name (first name and last name), the year they were born, and some short interesting fact about the person
- The class is in the `support` package
- It exports getters plus `equals`, `compareTo`, and `toString` methods.

The `equals` method

```java
@Override
public boolean equals(Object obj)
{
    if (obj == this)
        return true;
    else
    if (obj == null || obj.getClass() != this.getClass())
        return false;
    else
    {
        FamousPerson fp = (FamousPerson) obj;
        return (this.firstName.equals(fp.firstName) &&
                this.lastName.equals(fp.lastName));
    }
}
```

Identifying objects

- Objects are identified by their "key".

- The key of a class, from the point of view of an application, is the set of attributes that are used to determine the identity of an object of the class, for that application.

- For example
  - the key of a `Circle` object is its `radius` attribute
  - the key of a `FamousPerson` object is the combination of its `firstName` and `lastName` attributes

Retrieving objects based on their key

- Note that two objects can be equal although they are not identical

- Keys form the basis of retrieval

- Stored: Requested:

  firstName: Ada   firstName: Ada
  lastName: Lovelace   lastName: Lovelace
  yearOfBirth: 1815   yearOfBirth: 0
  fact: first programmer   fact:

- Instructors can review and demonstrate the `CSInfo` application found in the `ch05.applications` package

Ordering objects

- In addition to checking objects for equality, there is another type of comparison we need.

- To support a sorted collection we need to be able to tell when one object is less than, equal to, or greater than another object.

- The Java library provides an interface, called `Comparable`, which can be used to ensure that a class provides this functionality.

The `Comparable` Interface

- The `Comparable` interface consists of exactly one abstract method:

```
public int compareTo(T o);
// Returns a negative integer, zero, or a positive
// integer as this object is less than, equal to,
// or greater than the specified object.
```

- The `compareTo` method returns an integer value that indicates the relative "size" relationship between the object upon which the method is invoked and the object passed to the method as an argument.

A `compareTo` Example

```
public int compareTo(FamousPerson other)
// Precondition: 'other' is not null
//
// Compares this FamousPerson with 'other' for order. Returns a
// negative integer, zero, or a positive integer as this object
// is less than, equal to, or greater than 'other'.
{
  if (!this.lastName.equals(other.lastName))
    return this.lastName.compareTo(other.lastName);
  else
    return this.firstName.compareTo(other.firstName);
}
```

- This `compareTo` method uses the `compareTo` method of the String class.

- Note that the `equals` method and the `compareTo` method of our `Circle` class are compatible with each other.

- By convention the `compareTo` method of a class should support the standard order of a class. We call the order established by a class `compareTo` method the **natural order** of the class.

5.5 Sorted Array-Based Collection Implementation

- The `ArrayCollection` class features fast `add` (O(1)) but slow `get`, `contains`, and `remove` (O(N))

- Many applications require fast retrieval (`get` and `contains`) as they access the collection repeatedly to obtain information

- A sorted array approach permits use of the Binary Search for `find`, therefore speeding up both `get` and `contains`
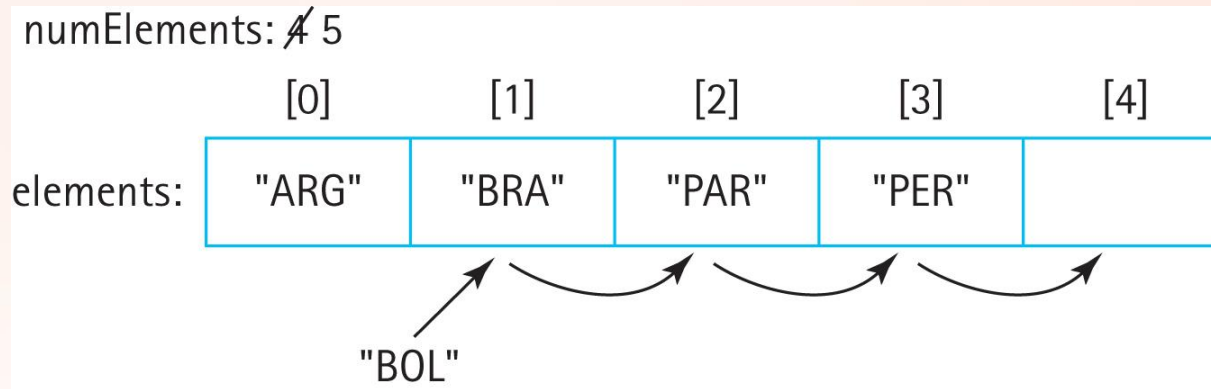
Comparable Elements

- We will use the `compareTo` method of the element class to keep underlying array sorted
- We specify as a precondition of the `add` method that its argument is comparable to the previous objects added to the collection.
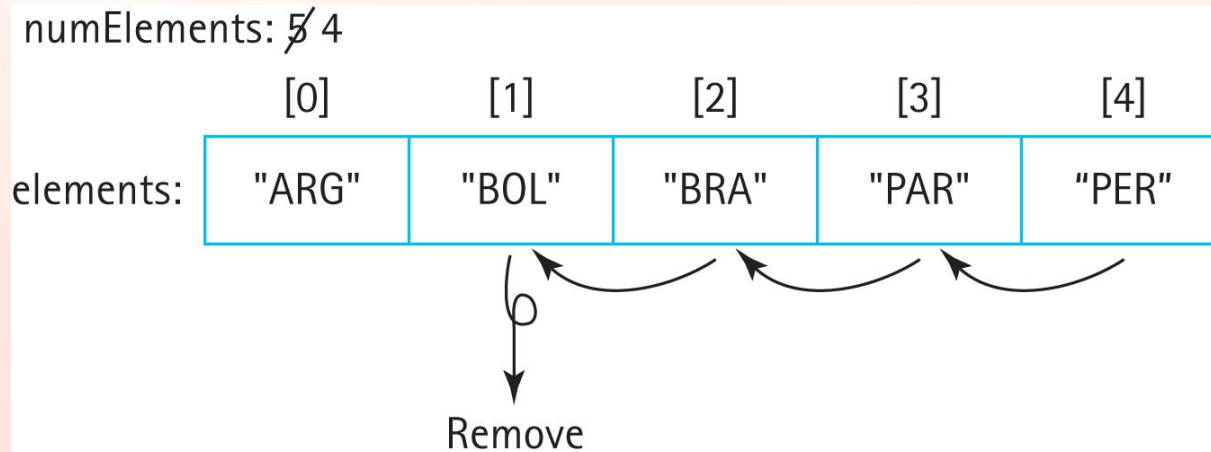
The `SortedArrayCollection` class

- Similar to `ArrayCollection` class but …
- Unbounded (uses a protected `enlarge` method)
- the `find` method uses the Binary Search algorithm
  - cast elements as `Comparable` (some compilers may generate a warning)
- add and remove must preserve the underlying order of the array (see next slide)

The `add` and `remove` operations

numElements: ~~4~~ 5

	[0]	[1]	[2]	[3]	[4]
elements:	"ARG"	"BRA"	"PAR"	"PER"	

"BOL"

add "BOL"

numElements: ~~5~~ 4

	[0]	[1]	[2]	[3]	[4]
elements:	"ARG"	"BOL"	"BRA"	"PAR"	"PER"

Remove

remove "BOL"

Sample Application Results

Table 5.1 Results of Vocabulary Density Experiment[1]

Text	File Size	Results		Array-Collection	Sorted-Array-Collection
Shakespeare's 18th Sonnet	1 KB	words: unique: density:	114 83 1.37	20 msecs	23 msecs
Shakespeare's *Hamlet*	177 KB	words: unique: density:	32,247 4,790 6.73	236 msecs	128 msecs
Linux Word File	400 KB	words: unique: density:	45,404 45,371 1.00	9,100 msecs	182 msecs
Melville's *Moby-Dick*	1,227 KB	words: unique: density:	216,113 17,497 12.35	2,278 msecs or 2.3 seconds	382 msecs
The Complete Works of William Shakespeare	5,542 KB	words: unique: density:	900,271 26,961 33.39	9.7 seconds	1.2 seconds
Webster's Unabridged Dictionary	28,278 KB	words: unique: density:	4,669,130 206,981 22.56	4.7 minutes	9.5 seconds
11th Edition of the *Encyclopaedia Britannica*	291,644 KB	words: unique: density:	47,611,399 695,531 68.45	56.4 minutes	2.5 minutes
Mashup	608,274 KB	words: unique: density:	102,635,256 1,202,099 85.38	10 hours	7.2 minutes

Implementing ADTs "by Copy" or "by Reference"

- When designing an ADT we have a choice about how to handle the elements—"by copy" or "by reference."

  - **By Copy:** The ADT manipulates copies of the data used in the client program. Making a valid copy of an object can be a complicated process.

  - **By Reference:** The ADT manipulates references to the actual elements passed to it by the client program. This is the most commonly used approach and is the approach we use throughout this textbook.

"By Copy" Notes

- Valid copies of an object are typically created using the object's `clone` method.
- Classes that provide a `clone` method must indicate this to the runtime system by implementing the `Cloneable` interface.
- Drawbacks:
  - Copy of object might not reflect up-to-date status of original object
  - Copying objects takes time, especially if the objects are large and require complicated deep-copying methods.
  - Storing extra copies of objects also requires extra memory.

"By Reference" Notes

- Because the client program retains a reference to the element, we say we have exposed the contents of the collection ADT to the client program.

- The ADT allows direct access to the individual elements of the collection by the client program through the client program's own references.

- Drawbacks:
  – We create aliases of our elements, therefore we must deal with the potential problems associated with aliases.
  – This situation is especially dangerous if the client program can use an alias to change an attribute of an element that is used by the ADT to determine the underlying organization of the elements – for example if it changes the key value for an element stored in a sorted list.

An Example

- The next three slides show the results of a sequence of operations when each of the two approaches is used to store a sorted list:
  - We have three objects that hold a person's name and weight (Slide 1)
  - We add the three objects onto a list that sorts objects by the variable weight
  - We transform one of the original objects with a diet method, that changes the weight of the object
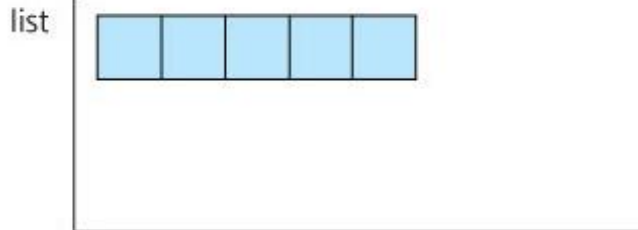
Example Step 1: The Three Objects

By Copy Approach

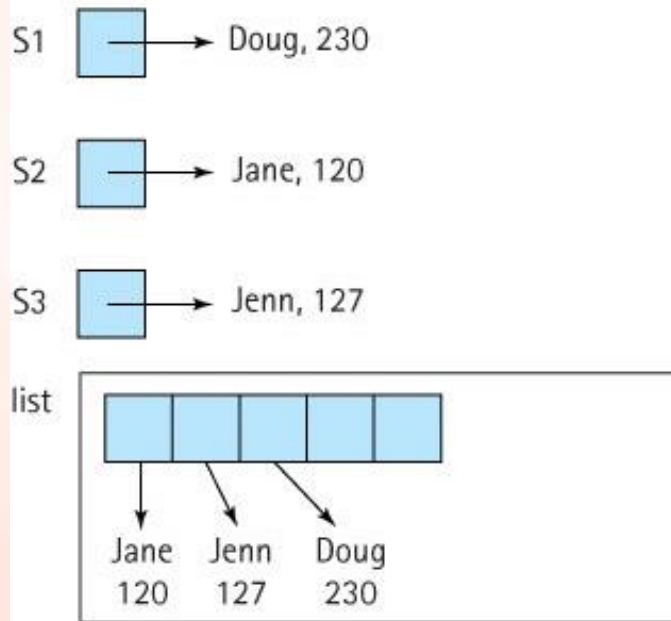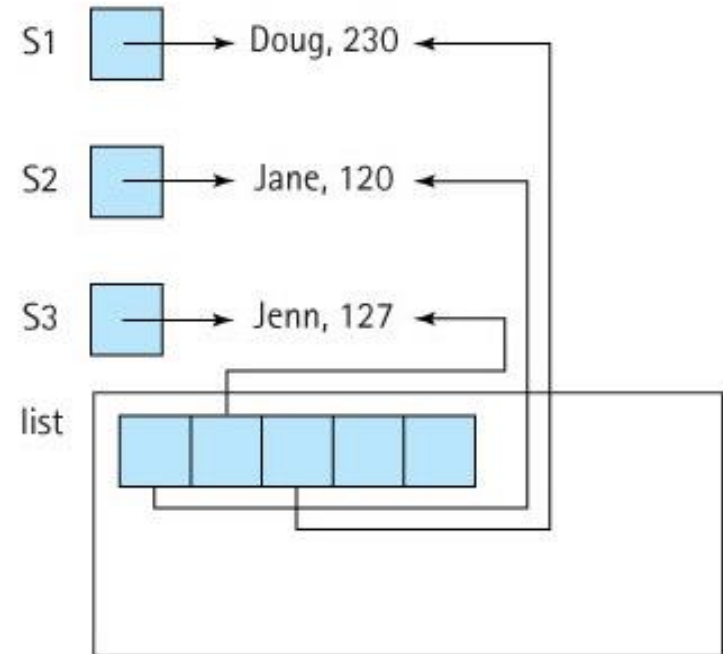By Reference Approach
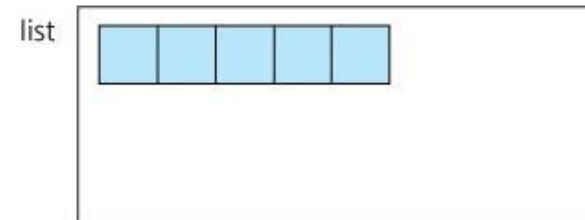
Example Step 2: Add Objects to List

By Copy Approach

By Reference Approach

Example Step 3: S1.diet(-105)

By Copy Approach

By Reference Approach
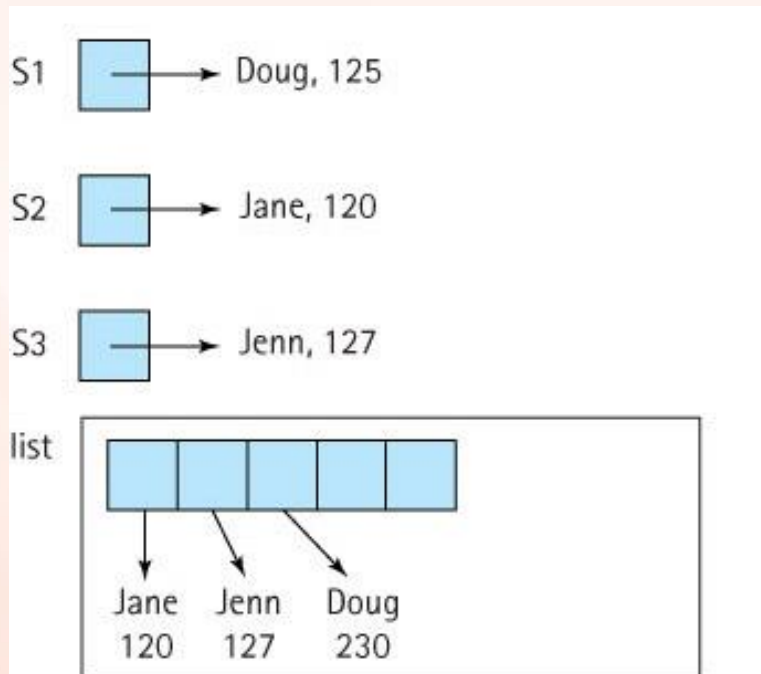
Problem: List copy is out of date
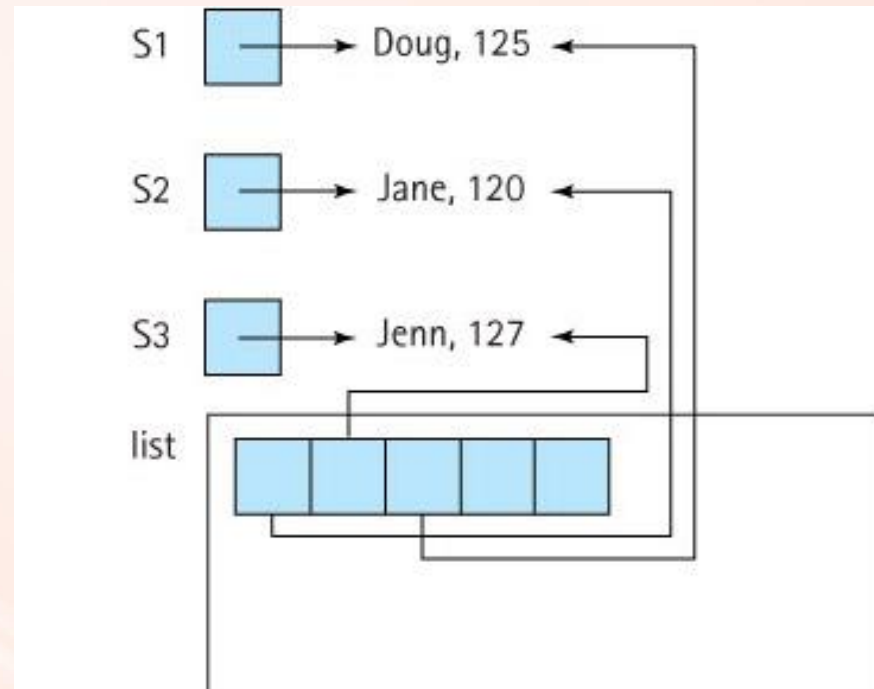
Problem: List is no longer sorted

Which approach is better?

- That depends.

- If processing time and space are issues, and if we are comfortable counting on the application programs to behave properly, then the "by reference" approach is probably best.

- If we are not too concerned about time and space (maybe our list objects are not too large), but we are concerned with maintaining careful control over the access to and integrity of our lists, then the "by copy" approach is probably best.

- The suitability of either approach depends on what the list is used for.

5.6 Link-Based Collection Implementation

- Internal representation: unsorted linked list

- Reuses design/code from previous classes

- Code is in the `ch05.collections` package

- The `find` method sets the `boolean found` plus two "pointers" if element is found: `previous` and `location`

- The next slide lists array-based and link-based `find` side-by-side for comparison

Array-Based

```
protected void find(T target)
{
  location = 0;
  found = false;
  while (location < numElements)
  {
    if elements[location].equals(target))
    {
      found = true;
      return;
    }
    else
      location++;
  }
}
```
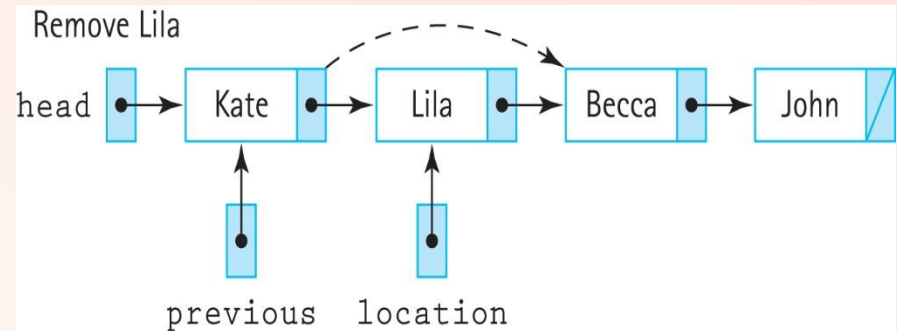
Link-Based

```
protected void find(T target)
{
  location = head;
  found = false;
  while (location != null)
  {
    if (location.getInfo().equals(target))
    {
      found = true;
      return;
    }
    else
    {
      previous = location;
      location = location.getLink();
    }
  }
}
```

Removing an element


```
public boolean remove (T target)
{
   find(target);
   if (found)
   {
     if (head == location)
        head = head.getLink();    // remove first node
     else
        previous.setLink(location.getLink());  // remove node at location
     numElements--;
   }
   return found;
}
```

Our Collection Architecture

ch05.collections

<<interface>>
CollectionInterface<T>

+ *add(T element): boolean*
+ *get(T target): T*
+ *contains(T target): boolean*
+ *remove(T target): boolean*
+ *isFull( ): boolean*
+ *isEmpty( ): boolean*
+ *size( ): int*

Key:
- - - -> uses
- - - -▷ implements

ch05.collections

ArrayCollection<T>

+ ArrayCollection( )
+ ArrayCollection(int capacity)
. . .

ch05.collections

LinkedCollection<T>

+ LinkedCollection( )
. . .

support

LLNode<T>

+ LLNode(T info)
+ setInfo(T info): void
+ getInfo( ): T
+ setLink(LLNode<T> link): void
+ getLink( ): LLNode<T>

ch05.collections

SortedArrayCollection<T>

+ SortedArrayCollection( )
+ SortedArrayCollection(int capacity)
. . .

java.lang

<<interface>>
Comparable<T>

+ *compareTo(T o): int*

Comparing Collection Implementations

Table 5.2 Comparison of Collection Implementations

Storage Structure	ArrayCollection Unsorted array	SortedArrayCollection Sorted array	LinkedCollection Unsorted linked list
Space	Bounded by original capacity	Unbounded—invokes `enlarge` method as needed	Unbounded—grows and shrinks
Class constructor	$O(N)$	$O(N)$	$O(1)$

`size`
`isEmpty`
`isFull`
	$O(1)$	$O(1)$	$O(1)$

`contains`	$O(N)$	$O(\log_2 N)$	$O(N)$
`get`	$O(N)$	$O(\log_2 N)$	$O(N)$
`add`	$O(1)$	$O(N)$	$O(1)$
`remove`	$O(N)$	$O(N)$	$O(N)$

5.7 Collection Variations

- The Collection ADT offers simple but crucial functionality–the ability to store and retrieve information.

- This functionality sits at the heart of information processing.

- Data structures, file systems, memory/storage, databases, the Cloud, the Internet all involve, at their core, storing and retrieving information.

- There are many variations of collections, including the lists, search trees, maps, hash tables, and priority queues we study in the upcoming chapters

The Java Collections Framework

- The Java library provides a robust collections framework

- At the center of the framework is the `Collection` interface, found in the `java.util` package of the library. This interface supports 11 subinterfaces including `Deque`, `List`, and `Set` and has 33 implementing classes

- If you are interested in learning more about the Java Collections Framework, reference the extensive documentation available at Oracle's website.

The Bag ADT

- On pages 332 and 333 we define a `BagInterface` that extends `CollectionInterface` with the additional methods `grab`, `count`, `removeAll` and `clear`

- Implementation is left as an exercise

The `BagInterface`

```java
package ch05.collections;

public interface BagInterface<T> extends CollectionInterface
{

  T grab();
  // If this bag is not empty, removes and returns a random
  // element of the bag; otherwise returns null.


  int count(T target);
  // Returns a count of all elements e in this collection
  // such that  e.equals(target).


  int removeAll(T target);
  // Removes all elements e from this collection such that
  // e.equals(target) and returns the number of elements removed.


  void clear();
  // Empties this bag so that it contains zero elements.

}
```

The Set ADT

- Our collection ADTs allow duplicate elements. If we disallow duplicate elements, we have a collection commonly known as a Set.

- The Set ADT models the mathematical set that is typically defined as a collection of distinct objects.

Set ADT Implementations

- We can implement a `Set` class by copying and changing the code from one of our collection implementations—the only method we need to change is the `add` method.

- The new `add` method could be designed to check if the element argument is not already in the collection, and if not it would add the element and return `true`.

- Otherwise, of course, it returns `false`.

Set ADT Implementations

- We can also implement a `Set` class by extending a previous class and overwriting `add`
  - See `BasicSet1` of the `ch05.collections` package, which extends the `LinkedCollection` class

- We can also implement a `Set` class by wrapping a previous class and re-writing `add`
  - See `BasicSet2` of the `ch05.collections` package, which wraps an object of the `LinkedCollection` class.