

LECTURE

9

THE JAVA COLLECTIONS FRAMEWORK

Lecture Goals

- ❑ To introduce different data structures available in Java
- ❑ To learn how to use the collection classes supplied in the Java library
- ❑ To use iterators to traverse collections
- ❑ To choose appropriate collections for solving programming problems
- ❑ To study applications of stacks and queues

Contents

- ❑ An Overview of the Java Collections Framework
- ❑ Linked Lists
- ❑ Sets
- ❑ Maps
- ❑ Stacks Queues and Priority Queues
- ❑ Stack and Queue Applications

Basic Data Structures in Java

- ❑ Array

- int [] Box[]

- ArrayList<Integer> ArrayList<Box>

- ❑ Linked list

- LinkedList<Integer> LinkedList<Box>

- ❑ Stack

- LinkedList<Integer> LinkedList<Box>

Basic Data Structures in Java

- ❑ Queue

- LinkedList<Integer> LinkedList<Box>

- ❑ Set (operations: insert, search, delete)

- HashSet<Integer> HashSet<Box>

- TreeSet<Integer> TreeSet<Box>

- ❑ Keyed Set (operations: insert item with key, search by key, delete by key)

- HashMap<Integer, Student> HashMap<String, Person>

- TreeMap<Integer, Student>> TreeMap<String, Person>

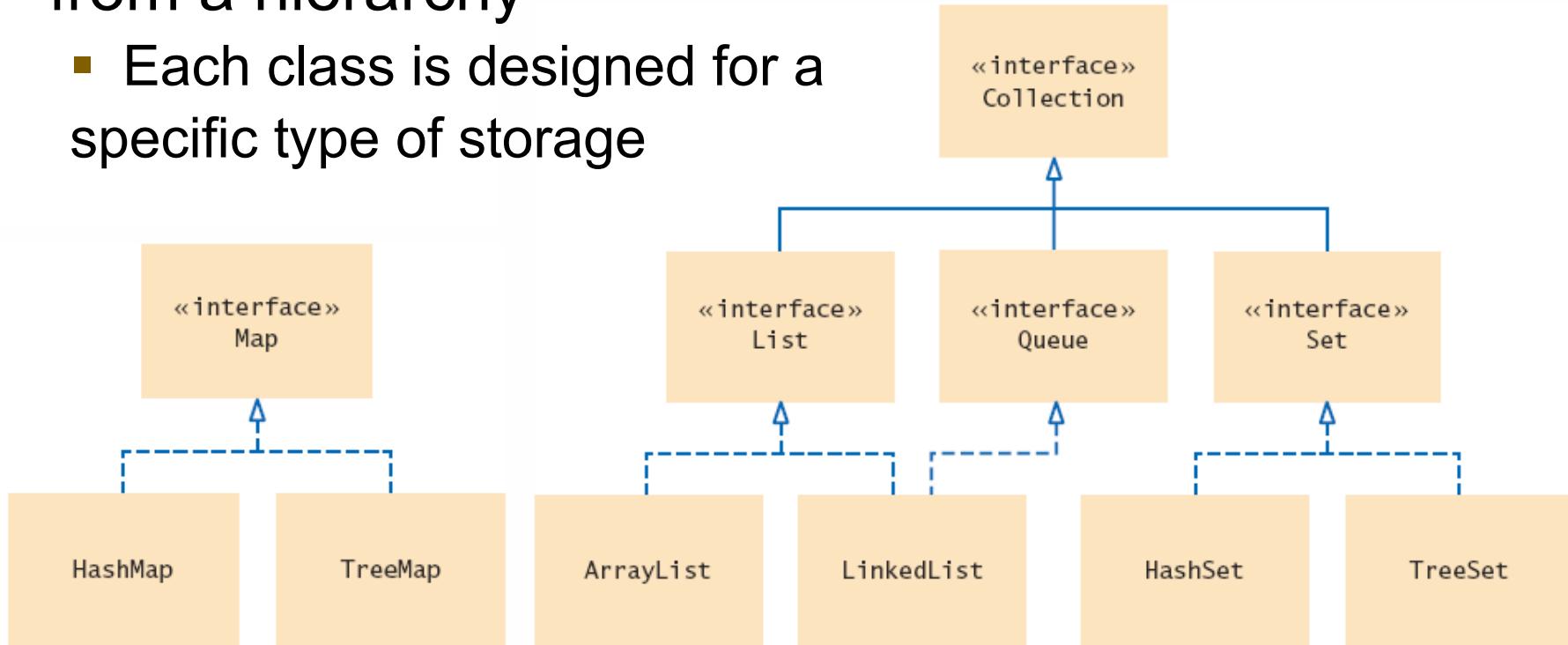
15.1 Java Collections Framework

- ❑ When you need to organize multiple objects in your program, you can place them into a collection
- ❑ The `ArrayList` class that was introduced in Chapter 6 is one of many collection classes that the standard Java library supplies
- ❑ Each interface type is implemented by one or more classes

A collection groups together elements and allows them to be accessed and retrieved later

Collections Framework Diagram

- Each collection class implements an interface from a hierarchy
 - Each class is designed for a specific type of storage



Lists and Sets

❑ Ordered Lists



- **ArrayList**
 - Stores a list of items in a dynamically sized array
- **LinkedList**
 - Allows speedy insertion and removal of items from the list

A **list** is a collection that maintains the order of its elements.

Lists and Sets

❑ Unordered Sets



■ HashSet

- Uses hash tables to speed up finding, adding, and removing elements

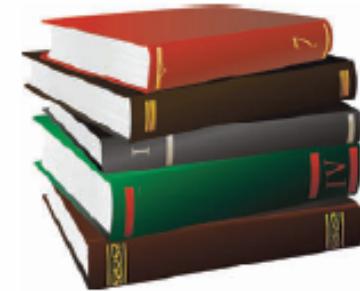
■ TreeSet

- Uses a binary tree to speed up finding, adding, and removing elements

A **set** is an unordered collection of unique elements.

Stacks and Queues

- ❑ Another way of gaining efficiency in a collection is to reduce the number of operations available
- ❑ Two examples are:
 - Stack
 - Remembers the order of its elements, but it does not allow you to insert elements in every position
 - You can only add and remove elements at the top
 - Queue
 - Add items to one end (the tail)
 - Remove them from the other end (the head)
 - Example: A line of people waiting for a bank teller



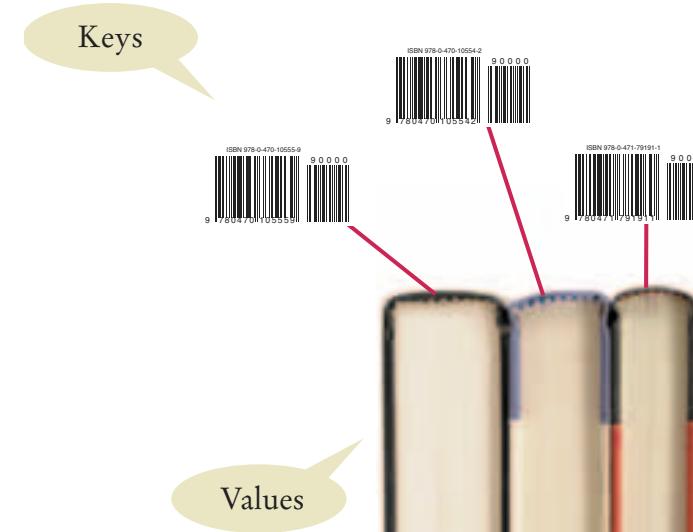
Maps

- ❑ A map stores keys, values, and the associations between them

- Example:
 - Barcode keys and books

A map keeps associations between key and value objects.

- ❑ Keys
 - Provides an easy way to represent an object (such as a numeric bar code)
- ❑ Values
 - The actual object that is associated with the key



The Collection Interface (1)

- ❑ List, Queue and Set are specialized interfaces that inherit from the Collection interface
 - All share the following commonly used methods

Table 1 The Methods of the Collection Interface

<pre>Collection<String> coll = new ArrayList<String>();</pre>	The <code>ArrayList</code> class implements the <code>Collection</code> interface.
<pre>coll = new TreeSet<String>()</pre>	The <code>TreeSet</code> class (Section 15.3) also implements the <code>Collection</code> interface.
<pre>int n = coll.size();</pre>	Gets the size of the collection. <code>n</code> is now 0.
<pre>coll.add("Harry"); coll.add("Sally");</pre>	Adds elements to the collection.
<pre>String s = coll.toString();</pre>	Returns a string with all elements in the collection. <code>s</code> is now "[Harry, Sally]"
<pre>System.out.println(coll);</pre>	Invokes the <code>toString</code> method and prints [Harry, Sally].

The Collection Interface (2)

Table 1 The Methods of the Collection Interface

<pre>coll.remove("Harry"); boolean b = coll.remove("Tom");</pre>	Removes an element from the collection, returning <code>false</code> if the element is not present. <code>b</code> is <code>false</code> .
<pre>b = coll.contains("Sally");</pre>	Checks whether this collection contains a given element. <code>b</code> is now <code>true</code> .
<pre>for (String s : coll) { System.out.println(s); }</pre>	You can use the “for each” loop with any collection. This loop prints the elements on separate lines.
<pre>Iterator<String> iter = coll.iterator()</pre>	You use an iterator for visiting the elements in the collection (see Section 15.2.3).

6.8 Array Lists

- ❑ When you write a program that collects values, you don't always know how many values you will have.
- ❑ In such a situation, a Java Array List offers two significant advantages:
 - Array Lists can grow and shrink as needed.
 - The `ArrayList` class supplies methods for common tasks, such as inserting and removing elements.

An Array List expands to hold as many elements as needed

Declaring and Using Array Lists

- ❑ The `ArrayList` class is part of the `java.util` package
 - It is a *generic* class
 - Designed to hold many types of objects
 - Provide the type of element during declaration
 - Inside `< >` as the ‘type parameter’:
 - The type must be a Class
 - Cannot be used for primitive types (`int`, `double`...)

```
ArrayList<String> names = new ArrayList<String>();
```

Syntax 6.4: Array Lists

Variable type Variable name An array list object of size 0
ArrayList<String> friends = new ArrayList<String>();

Use the
get and set methods
to access an element.

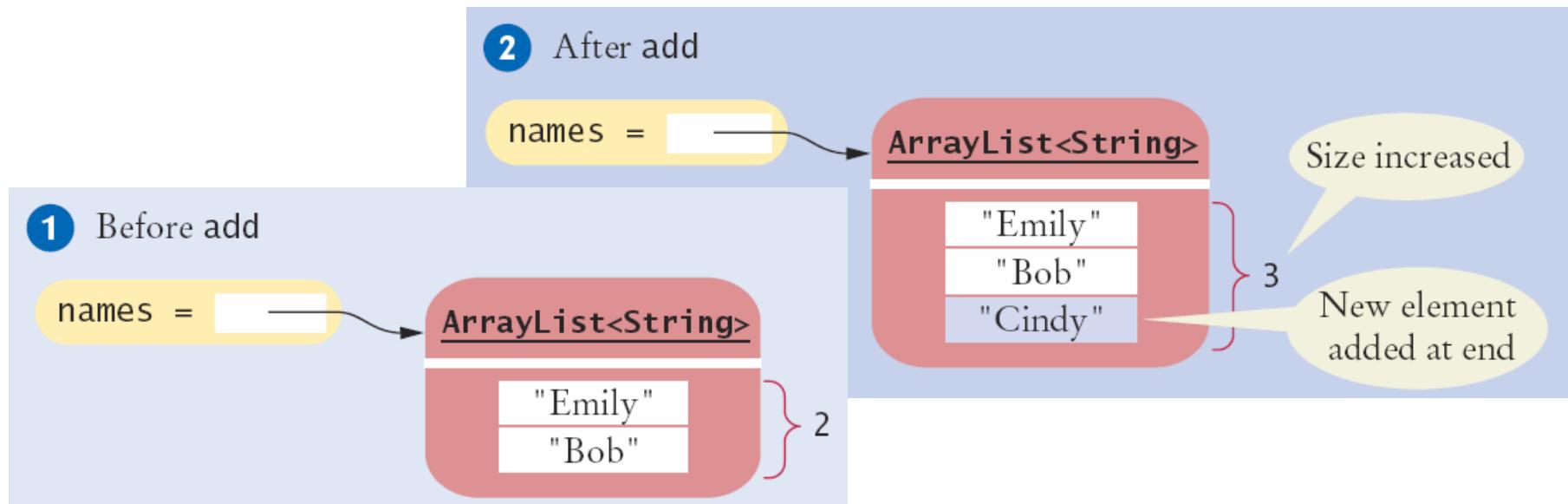
```
friends.add("Cindy");
String name = friends.get(i);
friends.set(i, "Harry");
```

The add method
appends an element to the array list,
increasing its size.

The index must be
 ≥ 0 and $< \text{friends.size}()$.

- ❑ ArrayList provides many useful methods:
 - add: add an element
 - get: return an element
 - remove: delete an element
 - set: change an element
 - size: current length

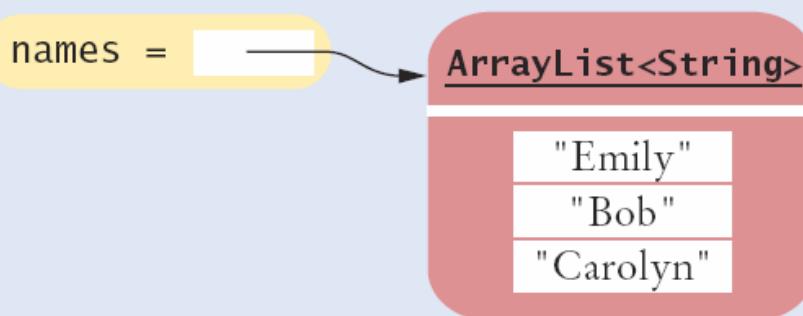
Adding an element with add()



- The `add` method has two versions:
 - Pass a new element to add to the end
`names.add("Cindy");`
 - Pass a location (index) and the new value to add
`names.add(1, "Cindy");` Moves all other elements

Adding an Element in the Middle

1 Before add



```
names.add(1, "Ann");
```



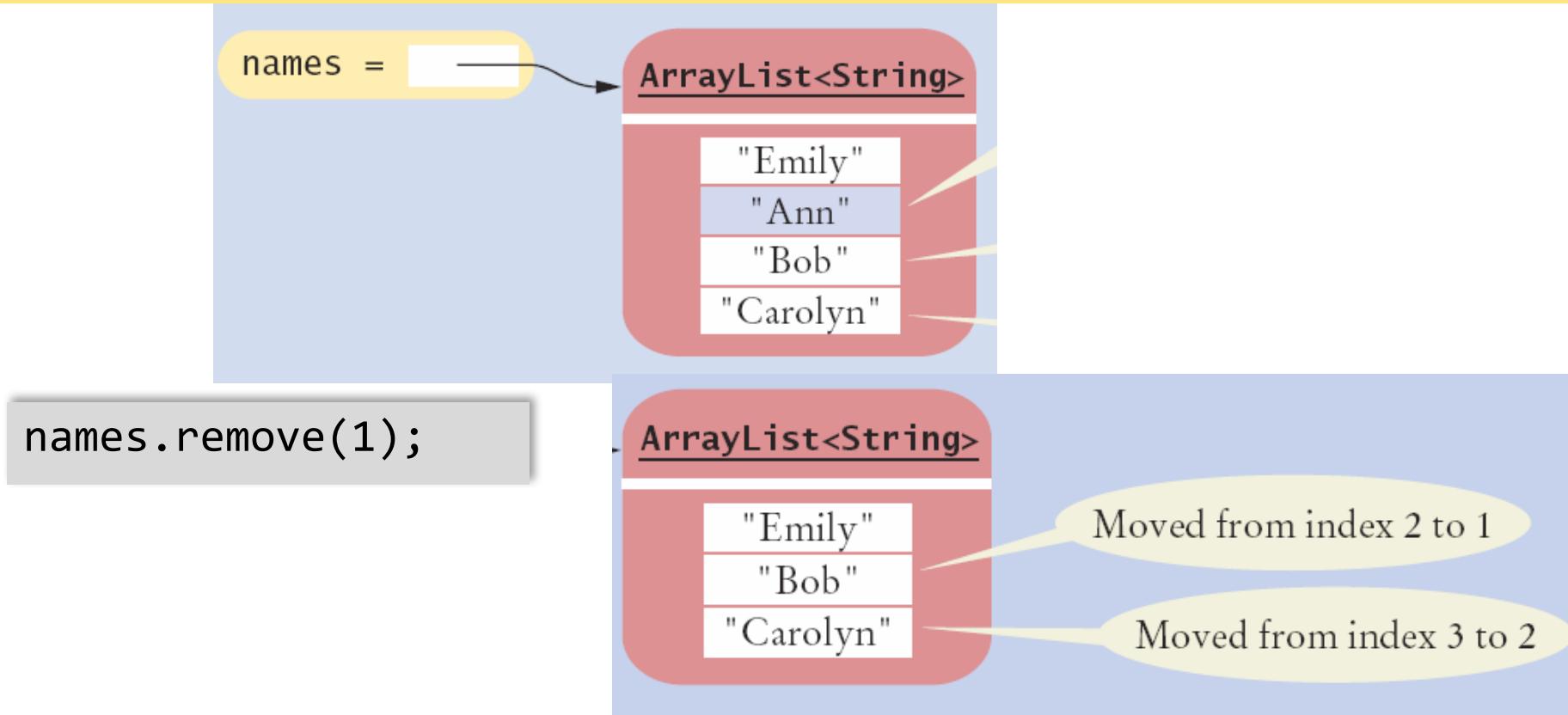
New element
added at index 1

Moved from index 1 to 2

Moved from index 2 to 3

- Pass a location (index) and the new value to add
Moves all other elements

Removing an Element



- Pass a location (index) to be removed
Moves all other elements

Using Loops with Array Lists

- You can use the enhanced for loop with Array Lists:

```
ArrayList<String> names = . . . ;
for (String name : names)
{
    System.out.println(name);
}
```

- Or ordinary loops:

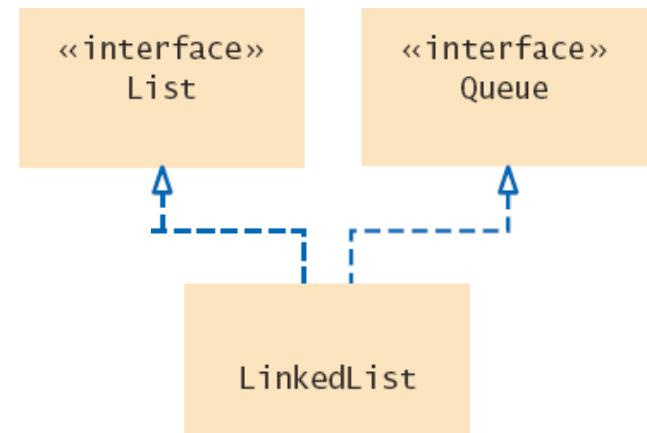
```
ArrayList<String> names = . . . ;
for (int i = 0; i < names.size(); i++)
{
    String name = names.get(i);
    System.out.println(name);
}
```

15.2 Linked Lists

- Linked lists use references to maintain an ordered lists of ‘nodes’
 - The ‘head’ of the list references the first node
 - Each node has a value and a reference to the next node



- They can be used to implement
 - A List Interface
 - A Queue Interface



Linked Lists Operations

❑ Efficient Operations

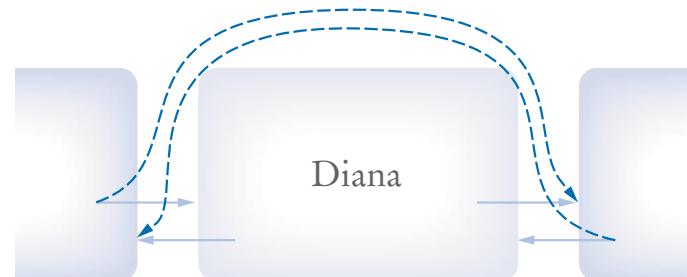
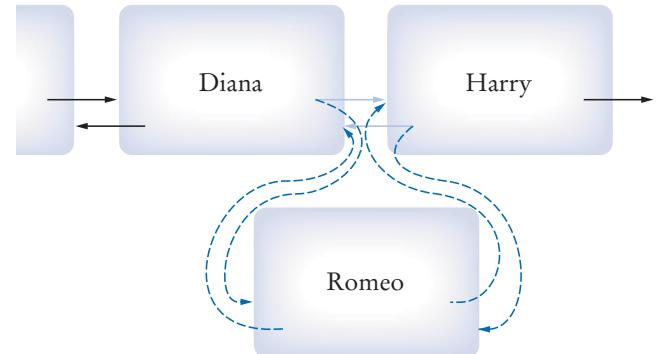
■ Insertion of a node

- Find the elements it goes between
- Remap the references

■ Removal of a node

- Find the element to remove
- Remap neighbor's references

■ Visiting all elements in order



❑ Inefficient Operations

■ Random access

Each instance variable is declared just like other variables we have used.

LinkedList: Important Methods

Table 2 Working with Linked Lists

<pre>LinkedList<String> list = new LinkedList<String>();</pre>	An empty list.
<pre>list.addLast("Harry");</pre>	Adds an element to the end of the list. Same as add.
<pre>list.addFirst("Sally");</pre>	Adds an element to the beginning of the list. list is now [Sally, Harry].
<pre>list.getFirst();</pre>	Gets the element stored at the beginning of the list; here "Sally".
<pre>list.getLast();</pre>	Gets the element stored at the end of the list; here "Harry".
<pre>String removed = list.removeFirst();</pre>	Removes the first element of the list and returns it. removed is "Sally" and list is [Harry]. Use removeLast to remove the last element.
<pre>ListIterator<String> iter = list.listIterator()</pre>	Provides an iterator for visiting all list elements (see Table 3 on page 676).

Generic Linked Lists

- ❑ The Collection Framework uses Generics
 - Each list is declared with a type field in < > angle brackets

```
LinkedList<String> employeeNames = . . .;
```

```
LinkedList<String>  
LinkedList<Employee>
```

List Iterators

- ❑ When traversing a `LinkedList`, use a `ListIterator`
 - Keeps track of where you are in the list.

```
LinkedList<String> employeeNames = . . .;  
ListIterator<String> iter = employeeNames.listIterator()
```

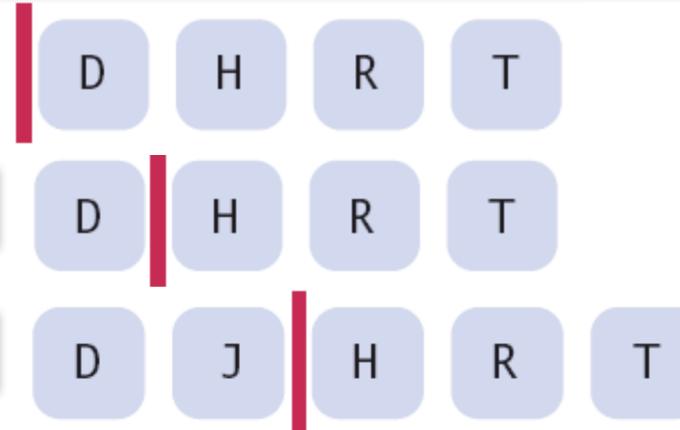
- ❑ Use an iterator to:
 - Access elements inside a linked list
 - Visit other than the first and the last nodes

Using Iterators

- Think of an iterator as pointing **between** two elements

```
ListIterator<String> iter = myList.listIterator()
```

Initial ListIterator position



- Note that the generic type for the `listIterator` must match the generic type of the `LinkedList`

Iterator and ListIterator Methods

- Iterators allow you to move through a list easily
 - Similar to an index variable for an array

Table 3 Methods of the Iterator and ListIterator Interfaces

<code>String s = iter.next();</code>	Assume that <code>iter</code> points to the beginning of the list [Sally] before calling <code>next</code> . After the call, <code>s</code> is "Sally" and the iterator points to the end.
<code>iter.previous();</code> <code>iter.set("Juliet");</code>	The <code>set</code> method updates the last element returned by <code>next</code> or <code>previous</code> . The list is now [Juliet].
<code>iter.hasNext()</code>	Returns <code>false</code> because the iterator is at the end of the collection.
<code>if (iter.hasPrevious())</code> { <code>s = iter.previous();</code> }	<code>hasPrevious</code> returns <code>true</code> because the iterator is not at the beginning of the list. <code>previous</code> and <code>hasPrevious</code> are <code>ListIterator</code> methods.
<code>iter.add("Diana");</code>	Adds an element before the iterator position (<code>ListIterator</code> only). The list is now [Diana, Juliet].
<code>iter.next();</code> <code>iter.remove();</code>	<code>remove</code> removes the last element returned by <code>next</code> or <code>previous</code> . The list is now [Diana].

Iterators and Loops

- ❑ Iterators are often used in while and “for-each” loops
 - `hasNext` returns true if there is a next element
 - `next` returns a reference to the value of the next element

```
while (iterator.hasNext())
{
    String name = iterator.next();
    // Do something with name
}
```

```
for (String name : employeeNames)
{
    // Do something with name
}
```

- Where is the iterator in the “for-next” loop?
 - It is used ‘behind the scenes’

Adding and Removing with Iterators

□ Adding

```
iterator.add("Juliet");
```

- A new node is added AFTER the Iterator
- The Iterator is moved past the new node

□ Removing

- Removes the object that was returned with the last call to next or previous
- It can be called only once after next or previous
- You cannot call it immediately after a call to add.

If you call the remove method improperly, it throws an IllegalStateException.

```
while (iterator.hasNext())
{
    String name = iterator.next();
    if (condition is true for name)
    {
        iterator.remove();
    }
}
```

ListDemo.java (1)

- Illustrates adding, removing and printing a list

```
1 import java.util.LinkedList;
2 import java.util.ListIterator;
3
4 /**
5  * This program demonstrates the LinkedList class.
6 */
7 public class ListDemo
8 {
9     public static void main(String[] args)
10    {
11         LinkedList<String> staff = new LinkedList<String>();
12         staff.addLast("Diana");
13         staff.addLast("Harry");
14         staff.addLast("Romeo");
15         staff.addLast("Tom");
16
17         // | in the comments indicates the iterator position
18
19         ListIterator<String> iterator = staff.listIterator(); // |DHRT
20         iterator.next(); // D|HRT
21         iterator.next(); // DH|RT
22 }
```

ListDemo.java (2)

```
23     // Add more elements after second element
24
25     iterator.add("Juliet"); // DHJ|RT
26     iterator.add("Nina"); // DHJN|RT
27
28     iterator.next(); // DHJNR|T
29
30     // Remove last traversed element
31
32     iterator.remove(); // DHJN|T
33
34     // Print all elements
35
36     System.out.println(staff);
37     System.out.println("Expected: [Diana, Harry, Juliet, Nina, Tom]");
38 }
39 }
```

Program Run

```
[Diana, Harry, Juliet, Nina, Tom]
Expected: [Diana, Harry, Juliet, Nina, Tom]
```

Break

15.3 Sets

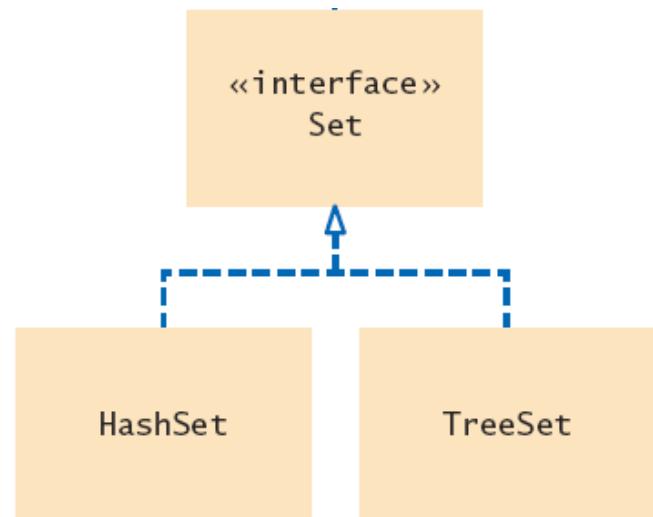
- A set is an unordered collection
 - It does not support duplicate elements
- The collection does not keep track of the order in which elements have been added
 - Therefore, it can carry out its operations more efficiently than an ordered collection

The HashSet and TreeSet classes both implement the Set interface.

Sets

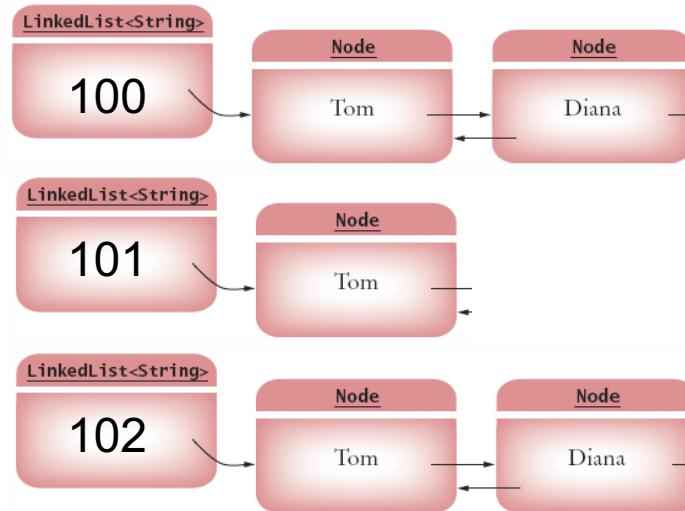
- ❑ HashSet: Stores data in a Hash Table
- ❑ TreeSet: Stores data in a Binary Tree
- ❑ Both implementations arrange the set elements so that finding, adding, and removing elements is efficient

Set implementations arrange the elements so that they can locate them quickly



Hash Table Concept

- ❑ Set elements are grouped into smaller collections of elements that share the same characteristic
 - It is usually based on the result of a mathematical calculation on the contents that results in an integer value
 - In order to be stored in a hash table, elements must have a method to compute their integer values



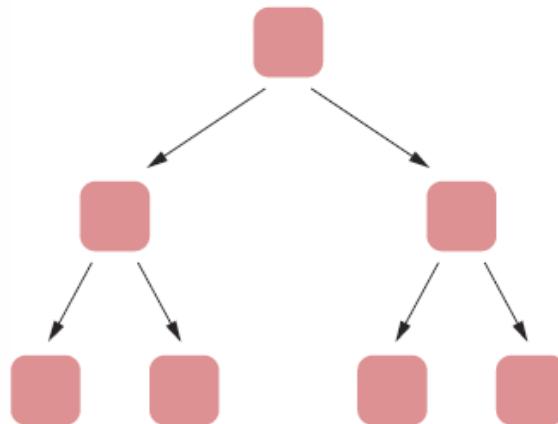
hashCode

- The method is called hashCode
 - If multiple elements have the same hash code, they are stored in a Linked list
- The elements must also have an equals method for checking whether an element equals another like:
 - String, Integer, Point, Rectangle, Color, and all collection classes

```
Set<String> names = new HashSet<String>();
```

Tree Concept

- Set elements are kept in sorted order
 - Nodes are not arranged in a linear sequence but in a tree shape



- In order to use a TreeSet, it must be possible to compare the elements and determine which one is “larger”

TreeSet

- Use TreeSet for classes that implement the Comparable interface
 - String and Integer, for example
 - The nodes are arranged in a ‘tree’ fashion so that each ‘parent’ node has up to two child nodes.
 - The node to the left always has a ‘smaller’ value
 - The node to the right always has a ‘larger’ value

```
Set<String> names = new TreeSet<String>();
```

Iterators and Sets

- Iterators are also used when processing sets
 - `hasNext` returns true if there is a next element
 - `next` returns a reference to the value of the next element
 - `add` via the iterator is not supported for `TreeSet` and `HashSet`

```
Iterator<String> iter = names.iterator();
while (iter.hasNext())
{
    String name = iter.next();
    // Do something with name
}
```

```
for (String name : names)
{
    // Do something with name
}
```

- Note that the elements are not visited in the order in which you inserted them.
- They are visited in the order in which the set keeps them:
 - Seemingly random order for a `HashSet`
 - Sorted order for a `TreeSet`

Working With Sets (1)

Table 4 Working with Sets

Set<String> names;	Use the interface type for variable declarations.
names = new HashSet<String>();	Use a TreeSet if you need to visit the elements in sorted order.
names.add("Romeo");	Now names.size() is 1.
names.add("Fred");	Now names.size() is 2.
names.add("Romeo");	names.size() is still 2. You can't add duplicates.
if (names.contains("Fred"))	The contains method checks whether a value is contained in the set. In this case, the method returns true.

Working With Sets (2)

Table 4 Working with Sets

<code>System.out.println(names);</code>	Prints the set in the format [Fred, Romeo]. The elements need not be shown in the order in which they were inserted.
<code>for (String name : names) { . . . }</code>	Use this loop to visit all elements of a set.
<code>names.remove("Romeo");</code>	Now <code>names.size()</code> is 1.
<code>names.remove("Juliet");</code>	It is not an error to remove an element that is not present. The method call has no effect.

SpellCheck.java (1)

```
1 import java.util.HashSet;
2 import java.util.Scanner;
3 import java.util.Set;
4 import java.io.File;
5 import java.io.FileNotFoundException;
6
7 /**
8      This program checks which words in a file are not present in a dictionary.
9 */
10 public class SpellCheck
11 {
12     public static void main(String[] args)
13         throws FileNotFoundException
14     {
15         // Read the dictionary and the document
16
17         Set<String> dictionaryWords = readWords("words");
18         Set<String> documentWords = readWords("alice30.txt");
19
20         // Print all words that are in the document but not the dictionary
21
22         for (String word : documentWords)
23         {
24             if (!dictionaryWords.contains(word))
25             {
26                 System.out.println(word);
27             }
28         }
29 }
```

SpellCheck.java (2)

```
29 }
30 /**
31      Reads all words from a file.
32      @param filename the name of the file
33      @return a set with all lowercased words in the file. Here, a
34          word is a sequence of upper- and lowercase letters.
35 */
36 public static Set<String> readWords(String filename)
37     throws FileNotFoundException
38 {
39     Set<String> words = new HashSet<String>();
40     Scanner in = new Scanner(new File(filename));
41     // Use any characters other than a-z or A-Z as delimiters
42     in.useDelimiter("[^a-zA-Z]+");
43     while (in.hasNext())
44     {
45         words.add(in.next().toLowerCase());
46     }
47     return words;
48 }
49 }
50 }
```

Program Run

```
neighbouring
croqueted
pennyworth
dutchess
comfits
xii
dinn
clamour
...
```

Programming Tip 15.1



❑ Use Interface References to Manipulate Data Structures

- It is considered good style to store a reference to a HashSet or TreeSet in a variable of type `Set`.

```
Set<String> words = new HashSet<String>();
```

- This way, you have to change only one line if you decide to use a TreeSet instead.

Programming Tip 15.1 (continued)

- Unfortunately the same is not true of the `ArrayList`, `LinkedList` and `List` classes
 - The `get` and `set` methods for random access are very inefficient
- Also, if a method can operate on arbitrary collections, use the `Collection` interface type for the parameter:

```
public static void removeLongWords(Collection<String> words)
```

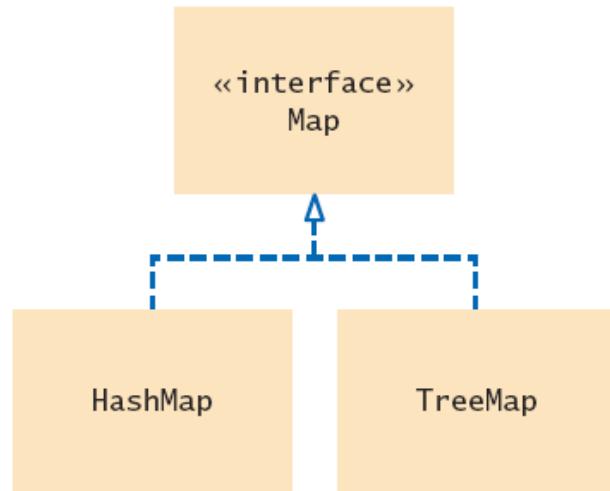
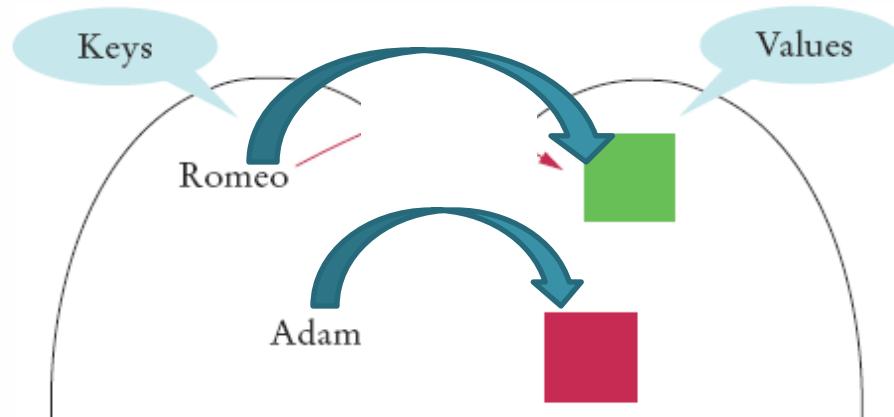
15.4 Maps

- ❑ A map allows you to associate elements from a key set with elements from a value collection.
 - The HashMap and TreeMap classes both implement the Map interface.
 - Use a map to look up objects by using a key.

Maps



```
Map<String, Color> favoriteColors = new HashMap<String, Color>();
```

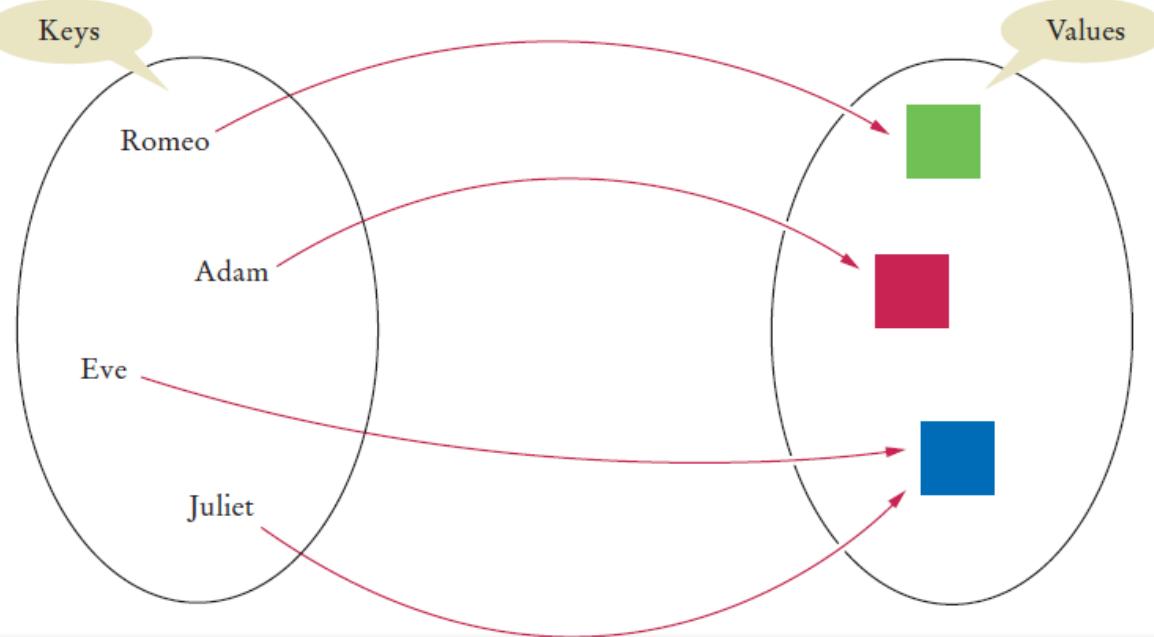


Working with Maps (Table 5)

<pre>Map<String, Integer> scores;</pre>	Keys are strings, values are Integer wrappers. Use the interface type for variable declarations.
<pre>scores = new TreeMap<String, Integer>();</pre>	Use a HashMap if you don't need to visit the keys in sorted order.
<pre>scores.put("Harry", 90); scores.put("Sally", 95);</pre>	Adds keys and values to the map.
<pre>scores.put("Sally", 100);</pre>	Modifies the value of an existing key.
<pre>int n = scores.get("Sally"); Integer n2 = scores.get("Diana");</pre>	Gets the value associated with a key, or null if the key is not present. n is 100, n2 is null.
<pre>System.out.println(scores);</pre>	Prints scores.toString(), a string of the form {Harry=90, Sally=100}
<pre>for (String key : scores.keySet()) { Integer value = scores.get(key); ... }</pre>	Iterates through all map keys and values.
<pre>scores.remove("Sally");</pre>	Removes the key and value.

Key Value Pairs in Maps

- Each key is associated with a value



```
Map<String, Color> favoriteColors = new HashMap<String, Color>();  
favoriteColors.put("Juliet", Color.RED);  
favoriteColors.put("Romeo", Color.GREEN);  
Color juliet'sFavoriteColor = favoriteColors.get("Juliet");  
favoriteColors.remove("Juliet");
```

Iterating through Maps

- To iterate through the map, use a `keySet` to get the list of keys:

```
Set<String> keySet = m.keySet();
for (String key : keySet)
{
    Color value = m.get(key);
    System.out.println(key + "->" + value);
}
```

To find all values in a map, iterate through the key set and find the values that correspond to the keys.

MapDemo.java

```
1 import java.awt.Color;
2 import java.util.HashMap;
3 import java.util.Map;
4 import java.util.Set;
5
6 /**
7     This program demonstrates a map that maps names to colors.
8 */
9 public class MapDemo
10 {
11     public static void main(String[] args)
12     {
13         Map<String, Color> favoriteColors = new HashMap<String, Color>();
14         favoriteColors.put("Juliet", Color.BLUE);
15         favoriteColors.put("Romeo", Color.GREEN);
16         favoriteColors.put("Adam", Color.RED);
17         favoriteColors.put("Eve", Color.BLUE);
18
19         // Print all keys and values in the map
20
21         Set<String> keySet = favoriteColors.keySet();
22         for (String key : keySet)
23         {
24             Color value = favoriteColors.get(key);
25             System.out.println(key + " : " + value);
26         }
27     }
28 }
```

Program Run

```
Juliet : java.awt.Color[r=0,g=0,b=255]
Adam : java.awt.Color[r=255,g=0,b=0]
Eve : java.awt.Color[r=0,g=0,b=255]
Romeo : java.awt.Color[r=0,g=255,b=0]
```

Steps to Choosing a Collection

1) Determine how you access values

- Values are accessed by an integer position. Use an `ArrayList`
 - Go to Step 2, then stop
- Values are accessed by a key that is not a part of the object
 - Use a `Map`.
- It doesn't matter. Values are always accessed “in bulk”, by traversing the collection and doing something with each value

2) Determine the element types or key/value types

- For a `List` or `Set`, a single type
- For a `Map`, the key type and the value type

Steps to Choosing a Collection

3) Determine whether element or key order matters

- Elements or keys must be sorted
 - Use a TreeSet or TreeMap. Go to Step 6
- Elements must be in the same order in which they were inserted
 - Your choice is now narrowed down to a LinkedList or an ArrayList
- It doesn't matter
 - If you chose a map in Step 1, use a HashMap and go to Step 5

Steps to Choosing a Collection

4) For a collection, determine which operations must be fast

- Finding elements must be fast
 - Use a HashSet and go to Step 5
- Adding and removing elements at the beginning or the middle must be fast
 - Use a LinkedList
- You only insert at the end, or you collect so few elements that you aren't concerned about speed
 - Use an ArrayList.

Steps to Choosing a Collection

- 5) For hash sets and maps, decide if you need to implement the equals and hashCode methods
 - If your elements do not support them, you must implement them yourself.
- 6) If you use a tree, decide whether to supply a comparator
 - If your element class does not provide it, implement the Comparable interface for your element class

Special Topic: Hash Functions



- Hashing can be used to find elements in a set data structure quickly, without making a linear search through all elements.
- A **hashCode** method computes and returns an integer value: the hash code.
 - Should be likely to yield different hash codes
 - Because hashing is so important, the Object class has a **hashCode** method that computes the hash code of any object x.

```
int h = x.hashCode();
```

Computing Hash Codes

- ❑ To put objects of a given class into a HashSet or use the objects as keys in a HashMap, the class should override the default `hashCode` method.
- ❑ A good `hashCode` method should work such that different objects are likely to have different hash codes.
 - It should also be efficient
 - A simple example for a String might be:

```
int h = 0;
for (int i = 0; i < s.length(); i++)
{
    h = h + s.charAt(i);
}
```

Computing Hash Codes

- But Strings that are permutations of another (such as "eat" and "tea") would all have the same hash code
- Better:
 - From the Java Library!

```
final int HASH_MULTIPLIER = 31;
int h = 0;
for (int i = 0; i < s.length(); i++)
{
    h = HASH_MULTIPLIER * h + s.charAt(i);
}
```

Sample Strings and HashCodes

- ❑ The `String` class implements a good example of a `hashCode` method
- ❑ It is possible for two or more distinct objects to have the same hash code: This is called a **collision**
 - A `hashCode` function should minimizes collisions

Table 6 Sample Strings and Their Hash Codes

String	Hash Code
"eat"	100184
"tea"	114704
"Juliet"	-2065036585
"Ugh"	84982
"VII"	84982

Computing Object Hash Codes

- You should have a good `hashCode` method for your own objects to store them efficiently
- Override `hashCode` methods in your own classes by combining the hash codes for the instance variables

```
public int hashCode()
{
    int h1 = name.hashCode();
    int h2 = new Double(area).hashCode();
    . . .
}
```

- Then combine the hash codes using a prime-number hash multiplier:

```
final int HASH_MULTIPLIER = 29;
int h = HASH_MULTIPLIER * h1 + h2;
return h;
}
```

hashCode and equals methods

- ❑ hashCode methods should be *compatible* with equals methods
 - If two objects are equal, their hashCodes should match
 - a hashCode method should use **all** instance variables
 - The hashCode method of the Object class uses the memory location of the object, not the contents

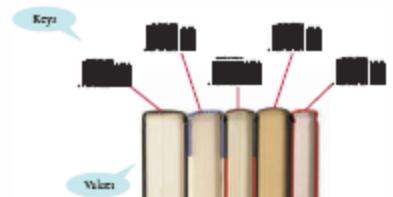
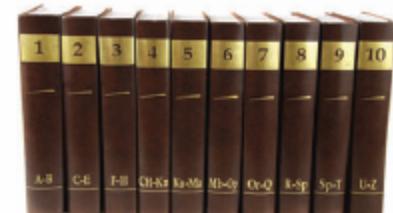
hashCode and equals methods

- Do not mix Object class hashCode or equals methods with your own:
 - Use an existing class such as String. Its hashCode and equals methods have already been implemented to work correctly.
 - Implement both hashCode and equals.
 - Derive the hash code from the instance variables that the equals method compares, so that equal objects have the same hash code
 - Implement neither hashCode nor equals. Then only identical objects are considered to be equal

Summary: Collections

- ❑ A collection groups together elements and allows them to be retrieved later

- A **list** is a collection that remembers the order of its elements
- A **set** is an unordered collection of unique elements
- A **map** keeps associations between key and value objects



Summary: Linked Lists

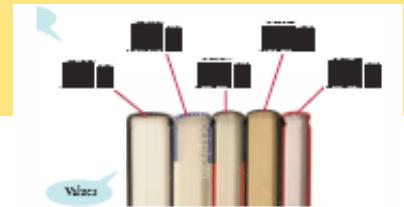
- A linked list consists of a number of nodes, each of which has a reference to the next node
 - Adding and removing elements in the middle of a linked list is efficient
 - Visiting the elements of a linked list in sequential order is efficient, but random access is not
 - You use a list iterator to access elements of a linked list



Summary: Choosing a Set

- ❑ The HashSet and TreeSet classes both implement the Set interface.
- ❑ Set implementations arrange the elements so that they can locate them quickly.
- ❑ You can form hash sets holding objects of type String, Integer, Double, Point, Rectangle, or Color.
- ❑ You can form tree sets for any class that implements the Comparable interface, such as String or Integer.
- ❑ Sets don't have duplicates. Adding a duplicate of an element that is already present is silently ignored.
- ❑ A set iterator visits the elements in the order in which the set implementation keeps them.
- ❑ You cannot add an element to a set at an iterator position.

Summary: Maps



□ Maps associate keys with values

- The HashMap and TreeMap classes both implement the Map interface
- To find all keys and values in a Map, iterate through the key set and find the values that correspond to the keys
- A hash function computes an integer value from an object.
- A good hash function minimizes **collisions**—identical hash codes for different objects.
- Override hashCode methods in your own classes by combining the hash codes for the instance variables.
- A class's hashCode method must be compatible with its equals method.