Lesson 11 – Part II
Generic Collections

Assoc. Prof. Marenglen Biba

(C) 2010 Pearson Education, Inc. All rights reserved.

In this Chapter you'll learn:

- What collections are.
- To use class `Arrays` for array manipulations.
- To form linked data structures using references, self-referential classes and recursion.
- The type-wrapper classes that enable programs to process primitive data values as objects.
- To use the collections framework (prebuilt data structure) implementations.
- To use collections framework methods (such as `search`, `sort` and `fill`) to manipulate collections.
- To use the collections framework interfaces to program with collections polymorphically.
- To use iterators to "walk through" a collection.
- To use persistent hash tables manipulated with objects of class `Properties`.
- To use synchronization and modifiability wrappers.

20.1 Introduction

20.2 Collections Overview

20.3 Type-Wrapper Classes for Primitive Types

20.4 Autoboxing and Auto-Unboxing

20.5 Interface `Collection` and Class `Collections`

20.6 Lists

20.6.1 `ArrayList` and `Iterator`

20.6.2 `LinkedList`

20.7 Collections Methods

20.7.1 Method `sort`

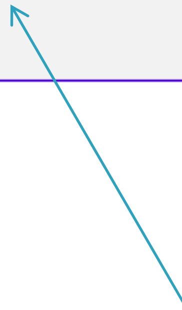
20.7.2 Method `shuffle`

20.7.3 Methods `reverse`, `fill`, `copy`, `max` and `min`

20.7.4 Method `binarySearch`

20.7.5 Methods `addAll`, `frequency` and `disjoint`

20.8 Stack Class of Package java.util
20.9 Class PriorityQueue and Interface Queue
20.10 Sets
20.11 Maps
20.12 Properties Class
20.13 Synchronized Collections
20.14 Unmodifiable Collections
20.15 Abstract Implementations
20.16 Wrap-Up



Not included in program

20.1 Introduction

- Java collections framework
  - prebuilt data structures
  - interfaces and methods for manipulating those data structures

20.2 Collections Overview

- A collection is a data structure — actually, an object — that can hold references to other objects.
  - Usually, collections contain references to objects that are all of the same type.
- Figure 20.1 lists some of the interfaces of the collections framework.
- Package `java.util`.

Interface	Description
`Collection`	The root interface in the collections hierarchy from which interfaces `Set`, `Queue` and `List` are derived.
`Set`	A collection that does not contain duplicates.
`List`	An ordered collection that can contain duplicate elements.
`Map`	Associates keys to values and cannot contain duplicate keys.
`Queue`	Typically a first-in, first-out collection that models a waiting line; other orders can be specified.

Fig. 20.1 | Some collections framework interfaces.

20.3 Type-Wrapper Classes for Primitive Types

- Each primitive type has a corresponding type-wrapper class (in package `java.lang`).
  - `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long` and `Short`.
- Each type-wrapper class enables you to manipulate primitive-type values as objects.
- Collections cannot manipulate variables of primitive types.
  - They can manipulate objects of the type-wrapper classes, because every class ultimately derives from `Object`.

20.3 Type-Wrapper Classes for Primitive Types (cont.)

- Each of the numeric type-wrapper classes — `Byte`, `Short`, `Integer`, `Long`, `Float` and `Double` — extends class `Number`.
- The type-wrapper classes are `final` classes, so you cannot extend them.
- Primitive types do not have methods, so the methods related to a primitive type are located in the corresponding type-wrapper class.

20.4 Autoboxing and Auto-Unboxing

- A boxing conversion converts a value of a primitive type to an object of the corresponding type-wrapper class.
- An unboxing conversion converts an object of a type-wrapper class to a value of the corresponding primitive type.
- These conversions can be performed automatically (called autoboxing and auto-unboxing).
- Example:
  - ```
    // create integerArray
    Integer[] integerArray = new Integer[ 5 ];

    // assign Integer 10 to integerArray[ 0 ]
    integerArray[ 0 ] = 10;

    // get int value of Integer
    ```
  - ```
    int value = integerArray[ 0 ];
    ```

20.5 Interface Collection and Class Collections

- Interface `Collection` is the root interface from which interfaces `Set`, `Queue` and `List` are derived.
- Interface `Set` defines a collection that does not contain duplicates.
- Interface `Queue` defines a collection that represents a waiting line.
- Interface `Collection` contains bulk operations for adding, clearing and comparing objects in a collection.

20.5 Interface `Collection` and Class `Collections`

- A `Collection` can be converted to an array.
- Interface `Collection` provides a method that returns an `Iterator` object, which allows a program to walk through the collection and remove elements from the collection during the iteration.
- Class `Collections` provides `static` methods that search, sort and perform other operations on collections.

20.6 Lists

- A `List` (sometimes called a sequence) is a `Collection` that can contain duplicate elements.
- `List` indices are zero based.
- In addition to the methods inherited from `Collection`, `List` provides methods for manipulating elements via their indices, manipulating a specified range of elements, searching for elements and obtaining a `ListIterator` to access the elements.
- Interface `List` is implemented by several classes, including `ArrayList`, `LinkedList` and `Vector`.
- Autoboxing occurs when you add primitive-type values to objects of these classes, because they store only references to objects.

20.6 Lists (cont.)

- Class `ArrayList` and `Vector` are resizable-array implementations of `List`.
- Inserting an element between existing elements of an `ArrayList` or `Vector` is an inefficient operation.
- A `LinkedList` enables efficient insertion (or removal) of elements in the middle of a collection.
- The primary difference between `ArrayList` and `Vector` is that `Vector`s are synchronized by default, whereas `ArrayList`s are not.

20.6.1 ArrayList and Iterator

- List method add adds an item to the end of a list.
- List method size returns the number of elements.
- List method get retrieves an individual element's value from the specified index.
- Collection method iterator gets an Iterator for a Collection.
- Iterator- method hasNext determines whether a Collection contains more elements.
  - Returns true if another element exists and false otherwise.
- Iterator method next obtains a reference to the next element.
- Collection method contains determine whether a Collection contains a specified element.
- Iterator method remove removes the current element from a Collection.

```
1  // Fig. 20.2: CollectionTest.java
2  // Collection interface demonstrated via an ArrayList object.
3  import java.util.List;
4  import java.util.ArrayList;
5  import java.util.Collection;
6  import java.util.Iterator;
7
8  public class CollectionTest
9  {
10    public static void main( String[] args )
11    {
12       // add elements in colors array to list
13       String[] colors = { "MAGENTA", "RED", "WHITE", "BLUE", "CYAN" };
14       List< String > list = new ArrayList< String >();
15
16       for ( String color : colors )
17          list.add( color ); // adds color to end of list
18
19       // add elements in removeColors array to removeList
20       String[] removeColors = { "RED", "WHITE", "BLUE" };
21       List< String > removeList = new ArrayList< String >();
22
```

Good practice to reference a collection via an interface-type variable—easier to change the collection later

Fig. 20.2 | `Collection` interface demonstrated via an `ArrayList` object. (Part 1 of 3.)

```
23         for ( String color : removeColors )
24            removeList.add( color );
25
26         // output list contents
27         System.out.println( "ArrayList: " );
28
29         for ( int count = 0; count < list.size(); count++ )
30            System.out.printf( "%s ", list.get( count ) );
31
32         // remove from list the colors contained in removeList
33         removeColors( list, removeList );
34
35         // output list contents
36         System.out.println( "\n\nArrayList after calling removeColors: " );
37
38         for ( String color : list )
39            System.out.printf( "%s ", color );
40      } // end main
41
```

Fig. 20.2 | `Collection` interface demonstrated via an `ArrayList` object. (Part 2 of 3.)

```
42      // remove colors specified in collection2 from collection1
43      private static void removeColors( Collection< String > collection1,
44         Collection< String > collection2 )
45      {
46         // get iterator
47         Iterator< String > iterator = collection1.iterator();
48
49         // loop while collection has items
50         while ( iterator.hasNext() )
51         {
52            if ( collection2.contains( iterator.next() ) )
53               iterator.remove(); // remove current Color
54         } // end while
55      } // end method removeColors
56   } // end class CollectionTest
```

Method works with any `Collection`

```
ArrayList:
MAGENTA RED WHITE BLUE CYAN

ArrayList after calling removeColors:
MAGENTA CYAN
```

Fig. 20.2 | `Collection` interface demonstrated via an `ArrayList` object. (Part 3 of 3.)

Common Programming Error 20.1

If a collection is modified by one of its methods after an iterator is created for that collection, the iterator immediately becomes invalid—operations performed with the iterator after this point throw `ConcurrentModificationException`*s. For this reason, iterators are said to be "fail fast."*

20.7 Collections Methods

- Class `Collections` provides several high-performance algorithms for manipulating collection elements.
- The algorithms (Fig. 20.5) are implemented as `static` methods.

Method	Description
`sort`	Sorts the elements of a `List`.
`binarySearch`	Locates an object in a `List`.
`reverse`	Reverses the elements of a `List`.
`shuffle`	Randomly orders a `List`'s elements.
`fill`	Sets every `List` element to refer to a specified object.
`copy`	Copies references from one `List` into another.
`min`	Returns the smallest element in a `Collection`.
`max`	Returns the largest element in a `Collection`.
`addAll`	Appends all elements in an array to a `Collection`.
`frequency`	Calculates how many collection elements are equal to the specified element.
`disjoint`	Determines whether two collections have no elements in common.

Fig. 20.5 | `Collections` methods.

Software Engineering Observation 20.4

The collections framework methods are polymorphic. That is, each can operate on objects that implement specific interfaces, regardless of the underlying implementations.

20.7.1 Method sort

- Method sort sorts the elements of a List
  - The elements must implement the Comparable interface.
  - The order is determined by the natural order of the elements' type as implemented by a compareTo method.
  - Method compareTo is declared in interface Comparable and is sometimes called the natural comparison method.
  - The sort call may specify as a second argument a Comparator object that determines an alternative ordering of the elements.

```java
1  // Fig. 20.6: Sort1.java
2  // Collections method sort.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6
7  public class Sort1
8  {
9     public static void main( String[] args )
10    {
11       String[] suits = { "Hearts", "Diamonds", "Clubs", "Spades" };
12
13       // Create and display a list containing the suits array elements
14       List< String > list = Arrays.asList( suits ); // create List
15       System.out.printf( "Unsorted array elements: %s\n", list );
16
17       Collections.sort( list ); // sort ArrayList
18
19       // output list
20       System.out.printf( "Sorted array elements: %s\n", list );
21    } // end main
22 } // end class Sort1
```

Fig. 20.6 | `Collections` method `sort`. (Part 1 of 2.)

```
Unsorted array elements: [Hearts, Diamonds, Clubs, Spades]
Sorted array elements: [Clubs, Diamonds, Hearts, Spades]
```

Fig. 20.6 | `Collections` method `sort`. (Part 2 of 2.)

20.7.1 Method `sort` (cont.)

- The `Comparator` interface is used for sorting a `Collection`'s elements in a different order.
- The `static` `Collections` method `reverseOrder` returns a `Comparator` object that orders the collection's elements in reverse order.

```
1  // Fig. 20.7: Sort2.java
2  // Using a Comparator object with method sort.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6
7  public class Sort2
8  {
9     public static void main( String[] args )
10    {
11       String[] suits = { "Hearts", "Diamonds", "Clubs", "Spades" };
12
13       // Create and display a list containing the suits array elements
14       List< String > list = Arrays.asList( suits ); // create List
15       System.out.printf( "Unsorted array elements: %s\n", list );
16
17       // sort in descending order using a comparator
18       Collections.sort( list, Collections.reverseOrder() );
19
20       // output List elements
21       System.out.printf( "Sorted list elements: %s\n", list );
22    } // end main
23 } // end class Sort2
```

Comparator reverses the sort order

Fig. 20.7 | Collections method sort with a Comparator object. (Part 1 of 2.)

```
Unsorted array elements: [Hearts, Diamonds, Clubs, Spades]
Sorted list elements: [Spades, Hearts, Diamonds, Clubs]
```

Fig. 20.7 | `Collections` method `sort` with a `Comparator` object. (Part 2 of 2.)

20.7.1 Method `sort` (cont.)

- Figure 20.8 creates a custom `Comparator` class, named `TimeComparator`, that implements interface `Comparator` to compare two `Time2` objects.
- Class `Time2`, declared in Fig. 8.5, represents times with hours, minutes and seconds.
- Class `TimeComparator` implements interface `Comparator`, a generic type that takes one type argument.
- A class that implements `Comparator` must declare a `compare` method that receives two arguments and returns a negative integer if the first argument is less than the second, 0 if the arguments are equal or a positive integer if the first argument is greater than the second.

```
1  // Fig. 20.8: TimeComparator.java
2  // Custom Comparator class that compares two Time2 objects.
3  import java.util.Comparator;
4
5  public class TimeComparator implements Comparator< Time2 >
6  {
7     public int compare( Time2 time1, Time2 time2 )
8     {
9        int hourCompare = time1.getHour() - time2.getHour(); // compare hour
10
11       // test the hour first
12       if ( hourCompare != 0 )
13          return hourCompare;
14
15       int minuteCompare =
16          time1.getMinute() - time2.getMinute(); // compare minute
17
18       // then test the minute
19       if ( minuteCompare != 0 )
20          return minuteCompare;
21
22       int secondCompare =
23          time1.getSecond() - time2.getSecond(); // compare second
```

Custom Comparator for Time2 objects

Fig. 20.8 | Custom Comparator class that compares two Time2 objects. (Part 1 of 2.)

```
24
25        return secondCompare; // return result of comparing seconds
26     } // end method compare
27  } // end class TimeComparator
```

Fig. 20.8 | Custom `Comparator` class that compares two `Time2` objects. (Part 2 of 2.)

```java
1  // Fig. 20.9: Sort3.java
2  // Collections method sort with a custom Comparator object.
3  import java.util.List;
4  import java.util.ArrayList;
5  import java.util.Collections;
6
7  public class Sort3
8  {
9     public static void main( String[] args )
10    {
11       List< Time2 > list = new ArrayList< Time2 >(); // create List
12
13       list.add( new Time2(  6, 24, 34 ) );
14       list.add( new Time2( 18, 14, 58 ) );
15       list.add( new Time2(  6, 05, 34 ) );
16       list.add( new Time2( 12, 14, 58 ) );
17       list.add( new Time2(  6, 24, 22 ) );
18
19       // output List elements
20       System.out.printf( "Unsorted array elements:\n%s\n", list );
21
```

Fig. 20.9 | `Collections` method `sort` with a custom `Comparator` object. (Part 1 of 2.)

```
22         // sort in order using a comparator
23         Collections.sort( list, new TimeComparator() );
24
25         // output List elements
26         System.out.printf( "Sorted list elements:\n%s\n", list );
27      } // end main
28   } // end class Sort3
```

Time2 objects could not be sorted before creating the TimeComparator; technique can be used to make objects of almost any class sortable

```
Unsorted array elements:
[6:24:34 AM, 6:14:58 PM, 6:05:34 AM, 12:14:58 PM, 6:24:22 AM]
Sorted list elements:
[6:05:34 AM, 6:24:22 AM, 6:24:34 AM, 12:14:58 PM, 6:14:58 PM]
```

Fig. 20.9 | Collections method sort with a custom Comparator object. (Part 2 of 2.)

20.7.2 Method `shuffle`

- Method `shuffle` randomly orders a `List`’s elements.

```
 1  // Fig. 20.10: DeckOfCards.java
 2  // Card shuffling and dealing with Collections method shuffle.
 3  import java.util.List;
 4  import java.util.Arrays;
 5  import java.util.Collections;
 6
 7  // class to represent a Card in a deck of cards
 8  class Card
 9  {
10     public static enum Face { Ace, Deuce, Three, Four, Five, Six,
11        Seven, Eight, Nine, Ten, Jack, Queen, King  };
12     public static enum Suit { Clubs, Diamonds, Hearts, Spades };
13
14     private final Face face; // face of card
15     private final Suit suit; // suit of card
16
17     // two-argument constructor
18     public Card( Face cardFace, Suit cardSuit )
19     {
20         face = cardFace; // initialize face of card
21         suit = cardSuit; // initialize suit of card
22     } // end two-argument Card constructor
```

Fig. 20.10 | Card shuffling and dealing with `Collections` method `shuffle`. (Part 1 of 5.)

```
23
24      // return face of the card
25      public Face getFace()
26      {
27         return face;
28      } // end method getFace
29
30      // return suit of Card
31      public Suit getSuit()
32      {
33         return suit;
34      } // end method getSuit
35
36      // return String representation of Card
37      public String toString()
38      {
39         return String.format( "%s of %s", face, suit );
40      } // end method toString
41   } // end class Card
42
```

Fig. 20.10 | Card shuffling and dealing with `Collections` method `shuffle`. (Part 2 of 5.)

```
43  // class DeckOfCards declaration
44  public class DeckOfCards
45  {
46     private List< Card > list; // declare List that will store Cards
47
48     // set up deck of Cards and shuffle
49     public DeckOfCards()
50     {
51        Card[] deck = new Card[ 52 ];
52        int count = 0; // number of cards
53
54        // populate deck with Card objects
55        for ( Card.Suit suit : Card.Suit.values() )
56        {
57           for ( Card.Face face : Card.Face.values() )
58           {
59              deck[ count ] = new Card( face, suit );
60              ++count;
61           } // end for
62        } // end for
63
```

Fig. 20.10 | Card shuffling and dealing with `Collections` method `shuffle`. (Part 3 of 5.)

```
64          list = Arrays.asList( deck ); // get List
65          Collections.shuffle( list );  // shuffle deck
66       } // end DeckOfCards constructor
67
68       // output deck
69       public void printCards()
70       {
71          // display 52 cards in two columns
72          for ( int i = 0; i < list.size(); i++ )
73             System.out.printf( "%-19s%s", list.get( i ),
74                ( ( i + 1 ) % 4 == 0 ) ? "\n" : "" );
75       } // end method printCards
76
77       public static void main( String[] args )
78       {
79          DeckOfCards cards = new DeckOfCards();
80          cards.printCards();
81       } // end main
82    } // end class DeckOfCards
```

Shuffles the contents of a collection

Fig. 20.10 | Card shuffling and dealing with `Collections` method `shuffle`. (Part 4 of 5.)

```
Deuce of Clubs        Six of Spades         Nine of Diamonds      Ten of Hearts
Three of Diamonds     Five of Clubs         Deuce of Diamonds     Seven of Clubs
Three of Spades       Six of Diamonds       King of Clubs         Jack of Hearts
Ten of Spades         King of Diamonds      Eight of Spades       Six of Hearts
Nine of Clubs         Ten of Diamonds       Eight of Diamonds     Eight of Hearts
Ten of Clubs          Five of Hearts        Ace of Clubs          Deuce of Hearts
Queen of Diamonds     Ace of Diamonds       Four of Clubs         Nine of Hearts
Ace of Spades         Deuce of Spades       Ace of Hearts         Jack of Diamonds
Seven of Diamonds     Three of Hearts       Four of Spades        Four of Diamonds
Seven of Spades       King of Hearts        Seven of Hearts       Five of Diamonds
Eight of Clubs        Three of Clubs        Queen of Clubs        Queen of Spades
Six of Clubs          Nine of Spades        Four of Hearts        Jack of Clubs
Five of Spades        King of Spades        Jack of Spades        Queen of Hearts
```

Fig. 20.10 | Card shuffling and dealing with `Collections` method `shuffle`. (Part 5 of 5.)

20.7.3 Methods reverse, fill, copy, max and min

- Collections method reverse reverses the order of the elements in a List
- Method fill overwrites elements in a List with a specified value.
- Method copy takes two arguments—a destination List and a source List.
  - Each source List element is copied to the destination List.
  - The destination List must be at least as long as the source List; otherwise, an IndexOutOfBoundsException occurs.
  - If the destination List is longer, the elements not overwritten are unchanged.
- Methods min and max each operate on any Collection.
  - Method min returns the smallest element in a Collection, and method max returns the largest element in a Collection.

```java
1  // Fig. 20.11: Algorithms1.java
2  // Collections methods reverse, fill, copy, max and min.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6
7  public class Algorithms1
8  {
9     public static void main( String[] args )
10    {
11       // create and display a List< Character >
12       Character[] letters = { 'P', 'C', 'M' };
13       List< Character > list = Arrays.asList( letters ); // get List
14       System.out.println( "list contains: " );
15       output( list );
16
17       // reverse and display the List< Character >
18       Collections.reverse( list ); // reverse order the elements
19       System.out.println( "\nAfter calling reverse, list contains: " );
20       output( list );
21
```

Fig. 20.11 | Collections methods reverse, fill, copy, max and min. (Part 1 of 3.)

```
22         // create copyList from an array of 3 Characters
23         Character[] lettersCopy = new Character[ 3 ];
24         List< Character > copyList = Arrays.asList( lettersCopy );
25
26         // copy the contents of list into copyList
27         Collections.copy( copyList, list );
28         System.out.println( "\nAfter copying, copyList contains: " );
29         output( copyList );
30
31         // fill list with Rs
32         Collections.fill( list, 'R' );
33         System.out.println( "\nAfter calling fill, list contains: " );
34         output( list );
35      } // end main
36
37      // output List information
38      private static void output( List< Character > listRef )
39      {
40         System.out.print( "The list is: " );
41
42         for ( Character element : listRef )
43            System.out.printf( "%s ", element );
```

Fig. 20.11 | `Collections` methods `reverse`, `fill`, `copy`, `max` and `min`. (Part 2 of 3.)

```
44
45         System.out.printf( "\nMax: %s", Collections.max( listRef ) );
46         System.out.printf( "  Min: %s\n", Collections.min( listRef ) );
47      } // end method output
48   } // end class Algorithms1
```

```
list contains:
The list is: P C M
Max: P  Min: C

After calling reverse, list contains:
The list is: M C P
Max: P  Min: C

After copying, copyList contains:
The list is: M C P
Max: P  Min: C

After calling fill, list contains:
The list is: R R R
Max: R  Min: R
```

Fig. 20.11 | `Collections` methods `reverse`, `fill`, `copy`, `max` and `min`. (Part 3 of 3.)

20.7.5 Methods `addAll`, `frequency` and `disjoint`

- `Collections` method `addAll` takes two arguments—a `Collection` into which to insert the new element(s) and an array that provides elements to be inserted.
- `Collections` method `frequency` takes two arguments — a `Collection` to be searched and an `Object` to be searched for in the collection.
  - Method `frequency` returns the number of times that the second argument appears in the collection.
- `Collections` method `disjoint` takes two `Collection`s and returns `true` if they have no elements in common.

```
 1  // Fig. 20.13: Algorithms2.java
 2  // Collections methods addAll, frequency and disjoint.
 3  import java.util.ArrayList;
 4  import java.util.List;
 5  import java.util.Arrays;
 6  import java.util.Collections;
 7
 8  public class Algorithms2
 9  {
10     public static void main( String[] args )
11     {
12        // initialize list1 and list2
13        String[] colors = { "red", "white", "yellow", "blue" };
14        List< String > list1 = Arrays.asList( colors );
15        ArrayList< String > list2 = new ArrayList< String >();
16
17        list2.add( "black" ); // add "black" to the end of list2
18        list2.add( "red" ); // add "red" to the end of list2
19        list2.add( "green" ); // add "green" to the end of list2
20
21        System.out.print( "Before addAll, list2 contains: " );
22
```

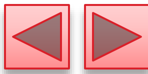
Fig. 20.13 | Collections methods addAll, frequency and disjoint. (Part 1 of 3.)

```java
23        // display elements in list2
24        for ( String s : list2 )
25           System.out.printf( "%s ", s );
26
27        Collections.addAll( list2, colors ); // add colors Strings to list2
28
29        System.out.print( "\nAfter addAll, list2 contains: " );
30
31        // display elements in list2
32        for ( String s : list2 )
33           System.out.printf( "%s ", s );
34
35        // get frequency of "red"
36        int frequency = Collections.frequency( list2, "red" );
37        System.out.printf(
38           "\nFrequency of red in list2: %d\n", frequency );
39
40        // check whether list1 and list2 have elements in common
41        boolean disjoint = Collections.disjoint( list1, list2 );
42
43        System.out.printf( "list1 and list2 %s elements in common\n",
44           ( disjoint ? "do not have" : "have" ) );
45     } // end main
46  } // end class Algorithms2
```

Fig. 20.13 | `Collections` methods `addAll`, `frequency` and `disjoint`. (Part 2 of 3.)

```
Before addAll, list2 contains: black red green
After addAll, list2 contains: black red green red white yellow blue
Frequency of red in list2: 2
list1 and list2 have elements in common
```

Fig. 20.13 | `Collections` methods `addAll`, `frequency` and `disjoint`. (Part 3 of 3.)



Exercise 1

- *Student Poll.* Figure 7.8 contains an array of survey responses that’s hard coded into the program.
- Suppose we wish to process survey results that are stored in a file.
- This exercise requires two separate programs. First, create an application that prompts the user for survey responses and outputs
- each response to a file.
- Use a Formatter to create a file called numbers.txt. Each integer should be written using method format.
- Then modify the program in Fig. 7.8 to read the survey responses from numbers.txt.
- The responses should be read from the file by using a Scanner. Use method nextInt to input one integer at a time from the file.
- The program should continue to read responses until it reaches the end of the file. The results should be output to the text file "output.txt".

From Lesson 5 Part-2

- Figure 7.8 uses arrays to summarize the results of data collected in a survey:
  - *Forty students were asked to rate the quality of the food in the student cafeteria on a scale of 1 to 10 (where 1 means awful and 10 means excellent). Place the 40 responses in an integer array, and summarize the results of the poll.*
- Array `responses` is a 40-element `int` array of the survey responses.
- 11-element array `frequency` counts the number of occurrences of each response (1 to 10).
  - Each element is initialized to zero by default.
  - We ignore `frequency[0]`.

End of Class

- End of the course
- Hope you have enjoyed the course
- Good luck and have fun!

(C) 2010 Pearson Education, Inc. All rights reserved.