16.7 `Collections` Methods

Class `Collections` provides several high-performance algorithms for manipulating collection elements. The algorithms (Fig. 16.5) are implemented as `static` methods. The methods `sort`, `binarySearch`, `reverse`, `shuffle`, `fill` and `copy` operate on `List`s. Methods `min`, `max`, `addAll`, `frequency` and `disjoint` operate on `Collection`s.

Software Engineering Observation 16.4

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

Method	Description
`sort`	Sorts the elements of a `List`.
`binarySearch`	Locates an object in a `List`, using the efficient binary search algorithm which we introduced in Section 7.15 and discuss in detail in Section 19.4.

`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. 16.5

Some `Collections` methods.

16.7.1 Method `sort`

Method `sort` sorts the elements of a `List`. The elements’ type must implement interface `Comparable`. The order is determined by the natural order of the elements’ type as implemented by a `compareTo` method. For example, the natural order for numeric values is ascending order, and the natural order for `String`s is based on their lexicographical ordering (Section 14.3). 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.

Sorting in Ascending Order

Figure 16.6 uses `Collections` method `sort` to order the elements of a `List` in *ascending* order (line 15). Line 12 creates `list` as a `List` of `String`s. Lines 13 and 16 each use an *implicit* call to the `list`’s `toString` method to output the list contents in the format shown in the output.

```
 1   // Fig. 16.6: 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      public static void main(String[] args) {
 9         String[] suits = {"Hearts", "Diamonds", "C
10
11         // Create and display a list containing th
12         List<String> list = Arrays.asList(suits);
13         System.out.printf("Unsorted array elements
14
15         Collections.sort(list); // sort ArrayList
16         System.out.printf("Sorted array elements:
17      }
18   }
```

```
Unsorted array elements: [Hearts, Diamonds, Clubs, Sp
Sorted array elements: [Clubs, Diamonds, Hearts, Spad
```

Fig. 16.6

`Collections` method `sort`.

Sorting in Descending Order

Figure 16.7 sorts the same list of strings used in Fig. 16.6 in *descending* order. The example introduces the `Comparator` interface, which is used for sorting a `Collection`’s elements in a different order. Line 16 calls `Collections`’s method `sort` to order the `List` in descending order. The `static` `Collections` **method** `reverseOrder` returns a `Comparator` object that orders the collection’s elements in reverse order. Because the collection being sorted is a `List<String>`, `reverseOrder` returns a `Comparator<String>`.

```
// Fig. 16.7: Sort2.java
// Using a Comparator object with method sort.
import java.util.List;
import java.util.Arrays;
import java.util.Collections;

public class Sort2 {
   public static void main(String[] args) {
        String[] suits = {"Hearts", "Diamonds", "C

        // Create and display a list containing th
        List<String> list = Arrays.asList(suits);
        System.out.printf("Unsorted array elements

        // sort in descending order using a compar
        Collections.sort(list, Collections.reverse
        System.out.printf("Sorted list elements: %
    }
}
```

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

Fig. 16.7

`Collections` method `sort` with a `Comparator` object.

Sorting with a `Comparator`

Figure 16.8 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.

```
// Fig. 16.8: TimeComparator.java
// Custom Comparator class that compares two Tim
import java.util.Comparator;

public class TimeComparator implements Comparato
   @Override
   public int compare(Time2 time1, Time2 time2)
      int hourDifference = time1.getHour() - tim

      if (hourDifference != 0) { // test the hou
         return hourDifference;
      }

      int minuteDifference = time1.getMinute() -

      if (minuteDifference != 0) { // then test
         return minuteDifference;
      }

      int secondDifference = time1.getSecond() -
      return secondDifference;
   }
}
```

Fig. 16.8

Custom `Comparator` class that compares two `Time2` objects.

Class `TimeComparator` implements interface `Comparator`, a generic type that takes one type argument (in this case `Time2`). 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. Method `compare` (lines 6–22) performs comparisons between `Time2` objects. Line 8 calculates the difference between the hours of the two `Time2` objects. If the hours are different (line 10), then we return this value. If this value is *positive*, then the first hour is greater than the second and the first time is greater than the second. If this value is *negative*, then the first hour is less than the second and the first time is less than the second. If this value is zero, the hours are the same and we must test the minutes (and maybe the seconds) to determine which time is greater.

Figure 16.9 sorts a list using the custom `Comparator` class `TimeComparator`. Line 9 creates an `ArrayList` of `Time2` objects. Recall that both `ArrayList` and `List` are generic types and accept a type argument that specifies the element type of the collection. Lines 11–15 create five `Time2` objects and add them to this list. Line 21 calls method `sort`, passing it an object of our `TimeComparator` class (Fig. 16.8).

```
1   // Fig. 16.9: Sort3.java
2   // Collections method sort with a custom Compara
3   import java.util.List;
4   import java.util.ArrayList;
5   import java.util.Collections;
6
7   public class Sort3 {
8      public static void main(String[] args) {
9         List<Time2> list = new ArrayList<>(); // c
10
```

```
      list.add(new Time2(6, 24, 34));
      list.add(new Time2(18, 14, 58));
      list.add(new Time2(6, 5, 34));
      list.add(new Time2(12, 14, 58));
      list.add(new Time2(6, 24, 22));

      // output List elements
      System.out.printf("Unsorted array elements

      // sort in order using a comparator
      Collections.sort(list, new TimeComparator(

      // output List elements
      System.out.printf("Sorted list elements:%n
   }
}
```

```
Unsorted array elements:
[6:24:34 AM, 6:14:58 PM, 6:05:34 AM, 12:14:58 PM, 6:2
Sorted list elements:
[6:05:34 AM, 6:24:22 AM, 6:24:34 AM, 12:14:58 PM, 6:1
```

Fig. 16.9

`Collections` method `sort` with a custom `Comparator` object.

16.7.2 Method `shuffle`

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

Chapter 7 presented a card shuffling and dealing simulation that shuffled a deck of cards with a loop. Figure 16.10 uses method `shuffle` to shuffle a deck of `Card` objects that might be used in a card-game simulator.

Class `Card` (lines 8–32) represents a card in a deck of cards. Each `Card` has a face and a suit. Lines 9–11 declare two `enum` types—`Face` and `Suit`—which represent the face and the suit of the card, respectively. Method `toString` (lines 29–31) returns a `String` containing the face and suit of the `Card` separated by the string `"of "`. When an `enum` constant is converted to a `String`, the constant's identifier is used as the `String` representation. Normally we would use all uppercase letters for `enum` constants. In this example, we chose to use capital letters for only the first letter of each `enum` constant because we want the card to be displayed with initial capital letters for the face and the suit (e.g., `"Ace of Spades"`).

```
 1   // Fig. 16.10: DeckOfCards.java
 2   // Card shuffling and dealing with Collections m
 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      public enum Face {Ace, Deuce, Three, Four, Fi
10         Seven, Eight, Nine, Ten, Jack, Queen, King
11      public enum Suit {Clubs, Diamonds, Hearts, Sp
12
13      private final Face face;
14      private final Suit suit;
```

```java

   // constructor
   public Card(Face face, Suit suit) {
      this.face = face;
      this.suit = suit;
   }

   // return face of the card
   public Face getFace() {return face;}

   // return suit of Card
   public Suit getSuit() {return suit;}

   // return String representation of Card
   public String toString() {
      return String.format("%s of %s", face, sui
   }
}

// class DeckOfCards declaration
public class DeckOfCards {
   private List<Card> list; // declare List that

   // set up deck of Cards and shuffle
   public DeckOfCards() {
      Card[] deck = new Card[52];
      int count = 0; // number of cards

      // populate deck with Card objects
      for (Card.Suit suit : Card.Suit.values())
         for (Card.Face face : Card.Face.values(
            deck[count] = new Card(face, suit);
            ++count;
         }
      }

      list = Arrays.asList(deck); // get List
      Collections.shuffle(list); // shuffle deck
   }

```

```java
    // output deck
    public void printCards() {
        // display 52 cards in four columns
        for (int i = 0; i < list.size(); i++) {
            System.out.printf("%-19s%s", list.get(i
                ((i + 1) % 4 == 0) ? System.lineSepa
        }
    }

    public static void main(String[] args) {
        DeckOfCards cards = new DeckOfCards();
        cards.printCards();
    }
}
```

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. 16.10

Card shuffling and dealing with `Collections` method `shuffle`.

Lines 44–49 populate the `deck` array with cards that have unique face and suit combinations. Both `Face` and `Suit` are `public static enum` types of class `Card`. To use these `enum` types outside of class `Card`, you must qualify each `enum`’s type name with the name of the class in which it resides (i.e., `Card`) and a dot (`.`) separator. Hence, lines 44 and 45 use `Card.Suit` and `Card.Face` to declare the control variables of the `for` statements. Recall that method `values` of an `enum` type returns an array that contains all the constants of the `enum` type. Lines 44–49 use enhanced `for` statements to construct 52 new `Card`s.

The shuffling occurs in line 52, which calls `static Collections` method `shuffle` to shuffle the `Card`s. Method `shuffle` requires a `List` argument, so we must obtain a `List` view of the array before we can shuffle it. Line 51 invokes `static` method `asList` of class `Arrays` to get a `List` view of the `deck` array.

Method `printCards` (lines 56–62) displays the deck of cards in four columns. In each iteration of the loop, lines 59–60 output a card left-justified in a 19-character field followed by either a newline or an empty string based on the number of cards output so far. If the number of cards is divisible by 4, a newline is output; otherwise, the empty string is output. Note that line 60 uses `System` method `lineSeparator` to get the platform-independent newline character to output after every four cards.

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

Class `Collections` provides methods for *reversing*, *filling* and *copying* `List`s. `Collections` **method** `reverse` reverses the order of the elements in a `List`, and **method** `fill` *overwrites* elements in a `List` with a specified value. The `fill` operation is useful for reinitializing a `List`. **Method** `copy` takes two arguments—a destination `List` and a source `List`. Each element in the source `List` 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.

Each method we’ve seen so far operates on `List`s. 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`. Both of these methods can be called with a `Comparator` object as a second argument to perform *custom comparisons* of objects, such as the `TimeComparator` in Fig. 16.9. Figure 16.11 demonstrates methods `reverse`, `fill`, `copy`, `max` and `min`.

```java
// Fig. 16.11: Algorithms1.java
// Collections methods reverse, fill, copy, max
import java.util.List;
import java.util.Arrays;
import java.util.Collections;

public class Algorithms1 {
   public static void main(String[] args) {
      // create and display a List<Character>
      Character[] letters = {'P', 'C', 'M'};
      List<Character> list = Arrays.asList(lette
      System.out.println("list contains: ");
      output(list);

      // reverse and display the List<Character>
      Collections.reverse(list); // reverse orde
      System.out.printf("%nAfter calling reverse
      output(list);

      // create copyList from an array of 3 Char
      Character[] lettersCopy = new Character[3]
      List<Character> copyList = Arrays.asList(l

      // copy the contents of list into copyList
      Collections.copy(copyList, list);
      System.out.printf("%nAfter copying, copyLi
      output(copyList);

      // fill list with Rs
      Collections.fill(list, 'R');
      System.out.printf("%nAfter calling fill, l
      output(list);
   }

   // output List information
   private static void output(List<Character> li
      System.out.print("The list is: ");

      for (Character element : listRef) {
         System.out.printf("%s ", element);
      }

      System.out.printf("%nMax: %s", Collections
      System.out.printf(" Min: %s%n", Collection
   }
}
```

```
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. 16.11

`Collections` methods `reverse`, `fill`, `copy`, `max` and `min`.

Line 11 creates `List<Character>` variable `list` and initializes it with a `List` view of the `Character` array `letters`. Lines 12–13 output the current contents of the `List`. Line 16 calls `Collections` method `reverse` to reverse the order of `list`. Method `reverse` takes one `List` argument. Since `list` is a `List` view of the array `letters`, the array's elements are now in reverse order. The reversed contents are output in lines 17–18. Line 25 uses `Collections` method `copy` to copy `list`'s elements into `copyList`. Changes to `copyList` do not change `letters`, because `copyList` is a separate `List` that's not a `List` view of the array `letters`. Method `copy` requires two `List` arguments—the destination `List` and the source `List`. Line 30 calls `Collections` method `fill` to place the character `'R'` in each `list` element. Because `list` is a `List` view of the array `letters`, this operation changes each element in `letters` to `'R'`. Method `fill` requires a `List` for the first argument and an object of the `List`'s element type for the second argument—in this case, the object is the *boxed* `Character` version of `'R'`. Lines 43–44 in method `output` call `Collections` methods `max` and `min` to find the largest and the smallest element of a `Collection`, respectively. Recall that interface `List` extends interface `Collection`, so a `List` *is a*

Collection.

16.7.4 Method binarySearch

The high-speed binary search algorithm—which we discuss in detail in Section 19.4—is built into the Java collections framework as a static Collections **method binarySearch**. This method locates an object in a List (e.g., a LinkedList or an ArrayList). If the object is found, its index is returned. If the object is not found, binarySearch returns a negative value. Method binarySearch determines this negative value by first calculating the insertion point and making its sign negative. Then, binarySearch subtracts 1 from the insertion point to obtain the return value, which guarantees that method binarySearch returns positive numbers (>= 0) if and only if the object is found. If multiple elements in the list match the search key, there’s no guarantee which one will be located first. Figure 16.12 uses method binarySearch to search for a series of strings in an ArrayList.

```
// Fig. 16.12: BinarySearchTest.java
// Collections method binarySearch.
import java.util.List;
import java.util.Arrays;
import java.util.Collections;
import java.util.ArrayList;

public class BinarySearchTest {
```

```java
    public static void main(String[] args) {
        // create an ArrayList<String> from the co
        String[] colors = {"red", "white", "blue",
            "purple", "tan", "pink"};
        List<String> list = new ArrayList<>(Arrays

        Collections.sort(list); // sort the ArrayL
        System.out.printf("Sorted ArrayList: %s%n"

        // search list for various values
        printSearchResults(list, "black");
        printSearchResults(list, "red");
        printSearchResults(list, "pink");
        printSearchResults(list, "aqua"); // below
        printSearchResults(list, "gray"); // does
        printSearchResults(list, "teal"); // does
    }

    // perform search and display result
    private static void printSearchResults(
        List<String> list, String key) {

        System.out.printf("%nSearching for: %s%n",
        int result = Collections.binarySearch(list

        if (result >= 0) {
            System.out.printf("Found at index %d%n"
        }
        else {
            System.out.printf("Not Found (%d)%n",re
        }
    }
}
```

```
Sorted ArrayList: [black, blue, pink, purple, red, ta

Searching for: black
Found at index 0
```

```
Searching for: red
Found at index 4

Searching for: pink
Found at index 2

Searching for: aqua
Not Found (-1)

Searching for: gray
Not Found (-3)

Searching for: teal
Not Found (-7)
```

Fig. 16.12

`Collections` method `binarySearch`.

Line 13 initializes `list` with an `ArrayList` containing a copy of the elements in array `colors`. `Collections` method `binarySearch` expects its `List` argument's elements to be sorted in *ascending* order, so line 15 uses `Collections` method `sort` to sort the list. If the `List` argument's elements are *not* sorted, `binarySearch`'s result is *undefined*. Line 16 outputs the sorted list. Lines 19–24 call method `printSearchResults` (lines 28–41) to perform searches and output the results. Line 32 calls `Collections` method `binarySearch` to search `list` for the specified `key`. Method `binarySearch` takes a `List` as the first argument and the search key as the second argument. Lines 34–39 output the results of the search. An overloaded version of `binarySearch` takes a `Comparator` object as its third argument, which specifies how `binarySearch` should compare the search key to the `List`’s elements.

16.7.5 Methods `addAll`, `frequency` and `disjoint`

Class `Collections` also provides the methods `addAll`, `frequency` and `disjoint`. `Collections` **method** `addAll` takes two arguments—a `Collection` into which to *insert* the new element(s) and an array (or variable-length argument list) 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*. Figure 16.13 demonstrates the use of methods `addAll`, `frequency` and `disjoint`.

```java
// Fig. 16.13: Algorithms2.java
// Collections methods addAll, frequency and dis
import java.util.ArrayList;
import java.util.List;
import java.util.Arrays;
import java.util.Collections;
```

```java

public class Algorithms2 {
   public static void main(String[] args) {
      // initialize list1 and list2
      String[] colors = {"red", "white", "yellow
      List<String> list1 = Arrays.asList(colors)
      ArrayList<String> list2 = new ArrayList<>(

      list2.add("black"); // add "black" to the
      list2.add("red"); // add "red" to the end
      list2.add("green"); // add "green" to the

      System.out.print("Before addAll, list2 con

      // display elements in list2
      for (String s : list2) {
         System.out.printf("%s ", s);
      }

      Collections.addAll(list2, colors); // add

      System.out.printf("%nAfter addAll, list2 c

      // display elements in list2
      for (String s : list2) {
         System.out.printf("%s ", s);
      }

      // get frequency of "red"
      int frequency = Collections.frequency(list
      System.out.printf("%nFrequency of red in l

      // check whether list1 and list2 have elem
      boolean disjoint = Collections.disjoint(li

      System.out.printf("list1 and list2 %s elem
         (disjoint ? "do not have" : "have"));
   }
}
```

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

Fig. 16.13

`Collections` methods `addAll`, `frequency` and `disjoint`.

Line 12 initializes `list1` with elements in array `colors`, and lines 15–17 add `String`s `"black"`, `"red"` and `"green"` to `list2`. Line 26 invokes method `addAll` to add elements in array `colors` to `list2`. Line 36 gets the frequency of `String "red"` in `list2` using method `frequency`. Line 40 invokes method `disjoint` to test whether `Collection`s `list1` and `list2` have elements in common, which they do in this example.