Data Structures

06 – Java Collections Algorithms

David Drohan

Introduction

- Collections Framework recap.
- Motivation for Collection Algorithms
- Comparing & Sorting Custom/User ADTs
- How to use the Collections Framework interfaces to program with collections polymorphically.
- How to use Collection Algorithms (such as `search`, `sort` and `fill etc.`) to manipulate collections from class **`Collections`**.

Introduction

- Remember, the Java Collections Framework contain the following:
  - **Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
  - **Implementations:** These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
  - **Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be ***polymorphic***: that is, the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.

Comparing & Sorting Custom/User ADTs

- You may (should!) have noticed that some classes (class String, class Double etc.) provide the ability to be sorted.
  - How is this possible when the collection is supposed to be de-coupled from the data?
- Java defines two ways of comparing objects:
  - The objects implement the **Comparable** interface **OR**
  - A **Comparator** object is used to compare the two objects
- If the objects in question are Comparable, they are said to be sorted by their *"natural "* order.
- Comparable objects can only offer **one** form of sorting. To provide multiple forms of sorting, Comparators must be used.

The `Comparable` Interface

- You may recall a method from the String class:
  - `int compareTo(Object)`
- This method returns:
  - 0 if the Strings are equal
  - <0 if this object is less than the specified object
  - >0 if this object is greater than the specified object.
- The Comparable interface contains the `compareTo` method.
- If you wish to provide a natural ordering for your objects, you must implement the Comparable Interface
- Any object which is "Comparable" can be compared to another object of the same type.
- There is only ***one method*** defined within this interface. Therefore, there is only one natural ordering of objects of a given type/class.

The `Comparator` Interface

- The Comparator interface defines two methods:
  - `int compare(Object, Object)`
- This method returns:
  - 0 if the Objects are equal
  - <0 if the first object is less than the second object
  - >0 if the first object is greater than the second object.

  - `boolean equals(Object)`
  - returns true if the specified object is equal to this comparator. i.e. the specified object provides the same type of comparison that this object does.
  - this method is optional

Using Comparators

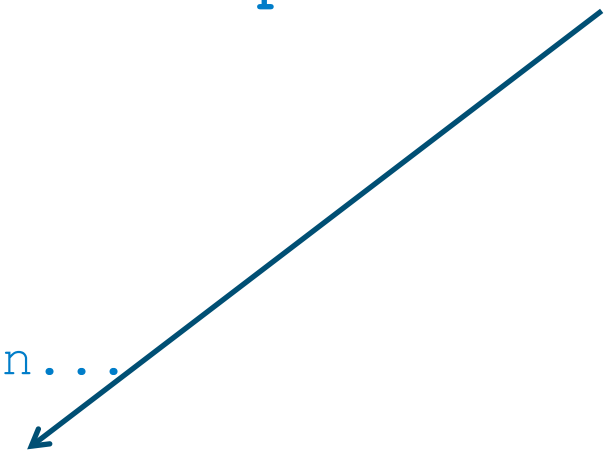
- Comparators are useful when objects must be sorted in different ways.
- For example
  - If Employees need to be sorted by first name, last name, start date, termination date and salary
    - A Comparator could be provided for each case
    - The Comparator interrogates the objects for the required values and returns the appropriate integer based on those values.
    - The appropriate Comparator is then provided as a parameter to the sorting algorithm.

Our refactored **Device** class – Option 1

```
public class Device implements Comparable<Device>
{
  private int id;
  private String title;
  private double price;

 // Existing implementation...

public int compareTo(Device other) {
            return this.getID() - other.getID();
        } // End of compareTo() method
} // End of class Device
```



```
Collections.sort(deviceList);
```

Our refactored **`Device`** class – Option 2


```
public class DeviceIDComparator implements

Comparator<Device>
{
public int compare(Device current, Device other) {
            return current.getID() - other.getID();
        } // End of compare() method
} // End of class Device
```

```
Collections.sort(deviceList, new DeviceIDComparator());
```

Collection Algorithms

- Java provides a series of pre-written algorithms based on the Collection interface
- These algorithms are accessible through the Collections class.
  - They are made available as static methods
  - Some methods are overloaded to provide natural ordering or ordering using a Comparator
- For Example - The method max has two implementations

```
Object max(Collection)
```

  - returns the maximum object based on the natural ordering of the objects (via its Comparable interface)

```
Object max(Collection, Comparator)
```

  - returns the maximum object based on the order induced by the Comparator

Collection Algorithms

Algorithm	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 elements in the collection are equal to the specified element.
`disjoint`	Determines whether two collections have no elements in common.

Collection Algorithms

- Here are some of the common Collection Algorithms provided by Java

```
int binarySearch(List, Object key)
int binarySearch(List, Object key, Comparator)
void copy(List dest, List src)
void fill(List, Object)
Object max(Collection)
Object max(Collection, Comparator)
Object min(Collection)
Object min(Collection, Comparator)
void reverse(List)
void shuffle(List)
void sort(List)
void sort(List, Comparator)
void synchronizedCollection(Collection)
void unmodifiableCollection(Collection)
```

Software Engineering Observation

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

19.6.1 Algorithm `sort`

- `sort`
  - Sorts `List` elements
    - Order is determined by **natural order** of elements' type
    - `List` elements must implement the `Comparable` interface **OR**
    - Pass a `Comparator` to method `sort`
- Sorting in ascending order
  - Collections method `sort`
- Sorting in descending order
  - Collections static method `reverseOrder`
- Sorting with a `Comparator`
  - Create a custom **`Comparator`** class

```
1  // Fig. 19.8: Sort1.java
2  // Using algorithm sort.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6
7  public class Sort1
8  {
9     private static final String suits[] =
10       { "Hearts", "Diamonds", "Clubs", "Spades" };
11
12    // display array elements
13    public void printElements()
14    {
15       List< String > list = Arrays.asList( suits ); // create List
16
```

Create `List` of `String`s

```
17       // output list
18       System.out.printf( "Unsorted array elements:\n%s\n", list );
19
20       Collections.sort( list ); // sort ArrayList
21
22       // output list
23       System.out.printf( "Sorted array elements:\n%s\n", list );
24    } // end method printElements
25
26    public static void main( String args[] )
27    {
28       Sort1 sort1 = new Sort1();
29       sort1.printElements();
30    } // end main
31 } // end class Sort1
```

Implicit call to the `list`'s `toString` method to output the list contents

Use algorithm `sort` to order the elements of `list` in ascending order

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


```
1  // Fig. 19.9: Sort2.java
2  // Using a Comparator object with algorithm sort.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6
7  public class Sort2
8  {
9     private static final String suits[] =
10       { "Hearts", "Diamonds", "Clubs", "Spades" };
11
12    // output List elements
13    public void printElements()
14    {
15       List< String > list = Arrays.asList( suits ); // create List
16
```

```
17        // output List elements
18        System.out.printf( "Unsorted array elements:\n%s\n", list );
19
20        // sort in descending order using a comparator
21        Collections.sort( list, Collections.reverseOrder() );
22
23        // output List elements
24        System.out.printf( "Sorted list elements:\n%s\
25     } // end method printElements
26
27     public static void main( String args[] )
28     {
29        Sort2 sort2 = new Sort2();
30        sort2.printElements();
31     } // end main
32  } // end class Sort2
```

Method `reverseOrder` of class `Collections` returns a **`Comparator`** object that represents the collection's reverse order

Method `sort` of class `Collections` can use a **`Comparator`** object to sort a `List`

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

```
1  // Fig. 19.10: 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 tim1, 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
24
25       return secondCompare; // return result of comparing seconds
26    } // end method compare
27 } // end class TimeComparator
```

Custom comparator `TimeComparator` implements `Comparator` interface and compares `Time2` object

Implement method `compare` to determine the order of two `Time2` objects

```
1  // Fig. 19.11: Sort3.java
2  // Sort a list using the custom Comparator class TimeComparator.
3  import java.util.List;
4  import java.util.ArrayList;
5  import java.util.Collections;
6
7  public class Sort3
8  {
9     public void printElements()
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
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 method printElements
28
29     public static void main( String args[] )
30     {
31        Sort3 sort3 = new Sort3();
32        sort3.printElements();
33     } // end main
34  } // end class Sort3
```

Sort in order using a custom comparator `TimeComparator`

```
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]
```

19.6.2 Algorithm `shuffle`

- `shuffle`
  - Randomly orders `List` elements

```
1  // Fig. 19.12: DeckOfCards.java
2  // Using algorithm 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
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
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
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( "%-20s%s", list.get( i ),
74             ( ( i + 1 ) % 2 == 0 ) ? "\n" : "\t" );
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
```

Use `enum` type `Suit` outside of class `Card`, qualify the `enum`'s type name (`Suit`) with the class name `Card` and a dot (`.`) separator

Use `enum` type `Face` outside of class `Card`, qualify the `enum`'s type name (`Face`) with the class name `Card` and a dot (`.`) separator

Invoke `static` method `asList` of class `Arrays` to get a `List` view of the `deck` array

Use method `shuffle` of class `Collections` to shuffle `List`

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

Algorithm `reverse`, `fill`, `copy`, `max` and `min`

- `reverse`
  - Reverses the order of `List` elements
- `fill`
  - Populates `List` elements with values
- `copy`
  - Creates copy of a `List`
- `max`
  - Returns largest element in `List`
- `min`
  - Returns smallest element in `List`

```
1  // Fig. 19.13: Algorithms1.java
2  // Using algorithms reverse, fill, copy, min and max.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6
7  public class Algorithms1
8  {
9     private Character[] letters = { 'P', 'C', 'M' };
10    private Character[] lettersCopy;
11    private List< Character > list;
12    private List< Character > copyList;
13
14    // create a List and manipulate it with methods from Collections
15    public Algorithms1()
16    {
17       list = Arrays.asList( letters ); // get List
18       lettersCopy = new Character[ 3 ];
19       copyList = Arrays.asList( lettersCopy ); // list view of lettersCopy
20
21       System.out.println( "Initial list: " );
22       output( list );
23
24       Collections.reverse( list ); // reverse order
25       System.out.println( "\nAfter calling reverse: " );
26       output( list );
27
```

Use method `reverse` of class `Collections` to obtain `List` in reverse order

```
28       Collections.copy( copyList, list ); // copy List
29       System.out.println( "\nAfter copying: " );
30       output( copyList );
31
32       Collections.fill( list, 'R' ); // fill list with Rs
33       System.out.println( "\nAfter calling fill: " );
34       output( list );
35    } // end Algorithms1 constructor
36
37    // output List information
38    private 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 );
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
```

Use method `copy` of class `Collections` to obtain copy of `List`

Use method `fill` of class `Collections` to populate `List` with the letter 'R'

Obtain maximum value in `List`

Obtain minimum value in `List`

```
49    public static void main( String args[] )
50    {
51       new Algorithms1();
52    } // end main
53 } // end class Algorithms1
```

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

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

After copying:
The list is: M C P
Max: P  Min: C

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

Algorithm `binarySearch`

- `binarySearch`
  - Locates object in `List`
    - Returns index of object in `List` if object exists
    - Returns negative value if Object does not exist
      - Calculate insertion point
      - Make the insertion point sign negative
      - Subtract 1 from insertion point

```
1  // Fig. 19.14: BinarySearchTest.java
2  // Using algorithm binarySearch.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6  import java.util.ArrayList;
7
8  public class BinarySearchTest
9  {
10    private static final String colors[] = { "red", "white",
11       "blue", "black", "yellow", "purple", "tan", "pink" };
12    private List< String > list; // ArrayList reference
13
14    // create, sort and output list
15    public BinarySearchTest()
16    {
17       list = new ArrayList< String >( Arrays.asList( colors ) );
18       Collections.sort( list ); // sort the ArrayList
19       System.out.printf( "Sorted ArrayList: %s\n", list );
20    } // end BinarySearchTest constructor
21
```

Sort `List` in ascending order

```
22    // search list for various values
23    private void search()
24    {
25       printSearchResults( colors[ 3 ] ); // first item
26       printSearchResults( colors[ 0 ] ); // middle item
27       printSearchResults( colors[ 7 ] ); // last item
28       printSearchResults( "aqua" ); // below lowest
29       printSearchResults( "gray" ); // does not exist
30       printSearchResults( "teal" ); // does not exist
31    } // end method search
32
33    // perform searches and display search result
34    private void printSearchResults( String key )
35    {
36       int result = 0;
37
38       System.out.printf( "\nSearching for: %s\n", key );
39       result = Collections.binarySearch( list, key );
40
41       if ( result >= 0 )
42          System.out.printf( "Found at index %d\n", result );
43       else
44          System.out.printf( "Not Found (%d)\n",result );
45    } // end method printSearchResults
46
```

Use method `binarySearch` of class `Collections` to search `list` for specified `key`

```
47    public static void main( String args[] )
48    {
49       BinarySearchTest binarySearchTest = new BinarySearchTest();
50       binarySearchTest.search();
51    } // end main
52 } // end class BinarySearchTest
```

```
Sorted ArrayList: [black, blue, pink, purple, red, tan, white, yellow]

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)
```

Algorithms `addAll`, `frequency` and `disjoint`

- `addAll`
  - Insert all elements of an array into a Collection
- `frequency`
  - Calculate the number of times a specific element appear in the Collection
- `Disjoint`
  - Determine whether two Collections have elements in common

```
1  // Fig. 19.15: Algorithms2.java
2  // Using algorithms addAll, frequency and disjoint.
3  import java.util.List;
4  import java.util.Vector;
5  import java.util.Arrays;
6  import java.util.Collections;
7
8  public class Algorithms2
9  {
10    private String[] colors = { "red", "white", "yellow", "blue" };
11    private List< String > list;
12    private Vector< String > vector = new Vector< String >();
13
14    // create List and Vector
15    // and manipulate them with methods from Collections
16    public Algorithms2()
17    {
18       // initialize list and vector
19       list = Arrays.asList( colors );
20       vector.add( "black" );
21       vector.add( "red" );
22       vector.add( "green" );
23
24       System.out.println( "Before addAll, vector contains: " );
25
```

```
26        // display elements in vector
27        for ( String s : vector )
28           System.out.printf( "%s ", s );
29
30        // add elements in colors to list
31        Collections.addAll( vector, colors );
32
33        System.out.println( "\n\nAfter addAll, vector contains: " );
34
35        // display elements in vector
36        for ( String s : vector )
37           System.out.printf( "%s ", s );
38
39        // get frequency of "red"
40        int frequency = Collections.frequency( vector, "red" );
41
42        System.out.printf("\n\nFrequency of red in vector: %d\n", frequency );
43
```

Invoke method `addAll` to add elements in array `colors` to `vector`

Get the frequency of `String` "`red`" in `Collection vector` using method `frequency`

```
44          // check whether list and vector have elements in common
45          boolean disjoint = Collections.disjoint( list, vector );
46
47          System.out.printf( "\nlist and vector %s elements in common\n",
48             ( disjoint ? "do not have" : "have"  ) );
49       } // end Algorithms2 constructor
50
51       public static void main( String args[] )
52       {
53          new Algorithms2();
54       } // end main
55    } // end class Algorithms2
```

Invoke method `disjoint` to test whether `Collections` `list` and `vector` have elements in common

```
Before addAll, vector contains:
black red green

After addAll, vector contains:
black red green red white yellow blue

Frequency of red in vector: 2

list and vector have elements in common
```

Questions?