Session 6

Collections

Contents

- the java.util package
- Collections Overview

Introduction

- Java collections framework
  - Provides reusable component
  - Existing data structures
    - Example of code reuse

Collections Overview

- Collection
  - Data structure (object) that can hold other objects

- Collections framework
  - Interfaces that define operations for various collection types
  - Belong to package **java.util**
    - **Collection**
    - **Set**
    - **List**
    - **Map**

Class **Arrays**

- Class **Arrays**
  - Provides **static** methods for manipulating arrays
  - Provides “high-level” methods
    - Method **binarySearch** for searching sorted arrays
    - Method **equals** for comparing arrays
    - Method **fill** for placing values into arrays
    - Method **sort** for sorting arrays

```
1  // Fig. 21.1: UsingArrays.java
2  // Using Java arrays.
3
4  // Java core packages
5  import java.util.*;
6
7  public class UsingArrays {
8     private int intValues[] = { 1, 2, 3, 4, 5, 6 };
9     private double doubleValues[] = { 8.4, 9.3, 0.2, 7.9, 3.4 };
10    private int filledInt[], intValuesCopy[];
11
12    // initialize arrays
13    public UsingArrays()
14    {
15       filledInt = new int[ 10 ];
16       intValuesCopy = new int[ intValues.length ];
17
18       Arrays.fill( filledInt, 7 );   // fill with 7s
19
20       Arrays.sort( doubleValues ); // sort doubleValues
21
22       System.arraycopy( intValues, 0, intValuesCopy,
23          0, intValues.length );
24    }
25
26    // output values in each array
27    public void printArrays()
28    {
29       System.out.print( "doubleValues: " );
30
31       for ( int count = 0; count < doubleValues.length; count++ )
32          System.out.print( doubleValues[ count ] + " " );
33
34       System.out.print( "\nintValues: " );
35
```

Fig. 21.1 Using methods of class `Arrays`.

Line 18

Line 20

Use **`static`** method **`fill`** of class **`Arrays`** to populate array with 7s

Use **`static`** method **`sort`** of class **`Arrays`** to sort array’s elements in ascending order

Use **`static`** method **`arraycopy`** of class **`System`** to copy array **`intValues`** into array **`intValuesCopy`**

```
36        for ( int count = 0; count < intValues.length; count++ )
37           System.out.print( intValues[ count ] + " " );
38
39        System.out.print( "\nfilledInt: " );
40
41        for ( int count = 0; count < filledInt.length; count++ )
42           System.out.print( filledInt[ count ] + " " );
43
44        System.out.print( "\nintValuesCopy: " );
45
46        for ( int count = 0; count < intValuesCopy.length; count++ )
47           System.out.print( intValuesCopy[ count ] + " " );
48
49        System.out.println();
50     }
51
52     // find value in array intValues
53     public int searchForInt( int value )
54     {
55        return Arrays.binarySearch( intValues, value );
56     }
57
58     // compare array contents
59     public void printEquality()
60     {
61        boolean b = Arrays.equals( intValues, intValuesCopy );
62
63        System.out.println( "intValues " + ( b ? "==" : "!=" )
64                            + " intValuesCopy" );
65
66        b = Arrays.equals( intValues, filledInt );
67
68        System.out.println( "intValues " + ( b ? "==" : "!=" )
69                            + " filledInt" );
70     }
```

Fig. 21.1 Using methods of class `Arrays`. (Part 2)

Line 55

Lines 61 and 66

Use **`static`** method **`binarySearch`** of class **`Arrays`** to perform binary search on array

Use **`static`** method **`equals`** of class **`Arrays`** to determine whether values of the two arrays are equivalent

```java

   // execute application
   public static void main( String args[] )
   {
      UsingArrays usingArrays = new UsingArrays();

      usingArrays.printArrays();
      usingArrays.printEquality();

      int location = usingArrays.searchForInt( 5 );
      System.out.println( ( location >= 0 ?
         "Found 5 at element " + location : "5 not found" ) +
         " in intValues" );

      location = usingArrays.searchForInt( 8763 );
      System.out.println( ( location >= 0 ?
         "Found 8763 at element " + location :
         "8763 not found" ) + " in intValues" );
   }

}  // end class UsingArrays
```

```
doubleValues: 0.2 3.4 7.9 8.4 9.3
      intValues: 1 2 3 4 5 6
  filledInt: 7 7 7 7 7 7 7 7 7 7
    intValuesCopy: 1 2 3 4 5 6
    intValues == intValuesCopy
      intValues != filledInt
Found 5 at element 4 in intValues
   8763 not found in intValues
```

Fig. 21.1 Using methods of class `Arrays`. (Part 3)

```java
// Fig. 21.2: UsingAsList.java
// Using method asList

// Java core packages
import java.util.*;

public class UsingAsList {
   private String values[] = { "red", "white", "blue" };
   private List list;

   // initialize List and set value at location 1
   public UsingAsList()
   {
      list = Arrays.asList( values );   // get List
      list.set( 1, "green" );           // change a value
   }

   // output List and array
   public void printElements()
   {
      System.out.print( "List elements : " );

      for ( int count = 0; count < list.size(); count++ )
         System.out.print( list.get( count ) + " " );

      System.out.print( "\nArray elements: " );

      for ( int count = 0; count < values.length; count++ )
         System.out.print( values[ count ] + " " );

      System.out.println();
   }

```

Fig. 21.2 Using `static` method `asList`.

Use **`static`** method **`asList`** of class **`Arrays`** to return **`List`** *view* of array **`values`**

Use method **`set`** of **`List`** object to change the contents of element **`1`** to **`"green"`**

`List` method **`size`** returns number of elements in **`List`**

`List` method **`get`** returns individual element in **`List`**

```
   // execute application
   public static void main( String args[] )
   {
      new UsingAsList().printElements();
   }

}  // end class UsingAsList
```

```
List elements : red green blue
Array elements: red green blue
```

Interface Collection and Class Collections

- Interface **Collection**
  - Contains *bulk operations*
    - Adding, clearing, comparing and retaining objects
  - Interfaces **Set** and **List** extend interface **Collection**

- Class **Collections**
  - Provides **static** methods that manipulate collections
  - Collections can be manipulated polymorphically

Lists

- List
  - Ordered **`Collection`** that can contain duplicate elements
  - Sometimes called a *sequence*
  - Implemented via interface **`List`**
    - **`ArrayList`**
    - **`LinkedList`**
    - **`Vector`**

```
1  // Fig. 21.3: CollectionTest.java
2  // Using the Collection interface
3
4  // Java core packages
5  import java.awt.Color;
6  import java.util.*;
7
8  public class CollectionTest {
9     private String colors[] = { "red", "white", "blue" };
10
11    // create ArrayList, add objects to it and manipulate it
12    public CollectionTest()
13    {
14       ArrayList list = new ArrayList();
15
16       // add objects to list
17       list.add( Color.magenta );    // add a color object
18
19       for ( int count = 0; count < colors.length; count++ )
20          list.add( colors[ count ] );
21
22       list.add( Color.cyan );       // add a color object
23
24       // output list contents
25       System.out.println( "\nArrayList: " );
26
27       for ( int count = 0; count < list.size(); count++ )
28          System.out.print( list.get( count ) + " " );
29
30       // remove all String objects
31       removeStrings( list );
32
33       // output list contents
34       System.out.println( "\n\nArrayList after calling" +
35          " removeStrings: " );
```

Fig. 21.3 Using an `ArrayList` to demonstrate interface `Collection`.

Lines 17-22

Line 28

Line 31

Use **`List`** method **`add`** to add objects to **`ArrayList`**

`List` method **`get`** returns individual element in **`List`**

Method **`removeStrings`** takes a **`Collection`** as an argument; Line 31 passes **`List`**, which extends **`Collection`**, to this method

```
36
37       for ( int count = 0; count < list.size(); count++ )
38          System.out.print( list.get( count ) + " " );
39    }
40
41    // remove String objects from Collection
42    public void removeStrings( Collection collection )
43    {
44       // get iterator
45       Iterator iterator = collection.iterator();
46
47       // loop while collection has items
48       while ( iterator.hasNext() )
49
50          if ( iterator.next() instanceof String )
51             iterator.remove();  // remove String object
52    }
53
54    // execute application
55    public static void main( String args[] )
56    {
57       new CollectionTest();
58    }
59
60 }  // end class CollectionTest
```

```
                          ArrayList:
java.awt.Color[r=255,g=0,b=255] red white blue java.awt.Color
                       [r=0,g=255,b=255]

              ArrayList after calling removeStrings:
java.awt.Color[r=255,g=0,b=255] java.awt.Color[r=0,g=255,b=255]
```

Fig. 21.3 Using an `ArrayList` to demonstrate interface `Collection`.

Obtain **`Collection`** iterator

`Iterator` method **`hasNext`** determines whether the **`Iterator`** contains more elements

`Iterator` method **`next`** returns next **`Object`** in **`Iterator`**

Line 51

Use **`Iterator`** method **`remove`** to remove **`String`** from **`Iterator`**

```
1  // Fig. 21.4: ListTest.java
2  // Using LinkLists
3
4  // Java core packages
5  import java.util.*;
6
7  public class ListTest {
8     private String colors[] = { "black", "yellow", "green",
9        "blue", "violet", "silver" };
10    private String colors2[] = { "gold", "white", "brown",
11       "blue", "gray", "silver" };
12
13    // set up and manipulate LinkedList objects
14    public ListTest()
15    {
16       LinkedList link = new LinkedList();
17       LinkedList link2 = new LinkedList();
18
19       // add elements to each list
20       for ( int count = 0; count < colors.length; count++ ) {
21          link.add( colors[ count ] );
22          link2.add( colors2[ count ] );
23       }
24
25       link.addAll( link2 );           // concatenate lists
26       link2 = null;                   // release resources
27
28       printList( link );
29
30       uppercaseStrings( link );
31
32       printList( link );
33
34       System.out.print( "\nDeleting elements 4 to 6..." );
35       removeItems( link, 4, 7 );
```

Fig. 21.4 Using `Lists` and `ListIterators`.

Lines 16-17

Line 25

Line 26

Create two **`LinkedList`** objects

Use **`LinkedList`** method **`addAll`** to append **`link2`** elements to **`link`**

Nullify **`link2`**, so it can be garbage collected

```
36
37          printList( link );
38       }
39
40       // output List contents
41       public void printList( List list )
42       {
43          System.out.println( "\nlist: " );
44
45          for ( int count = 0; count < list.size(); count++ )
46             System.out.print( list.get( count ) + " " );
47
48          System.out.println();
49       }
50
51       // locate String objects and convert to upper
52       public void uppercaseStrings( List list )
53       {
54          ListIterator iterator = list.listIterator();
55
56          while ( iterator.hasNext() ) {
57             Object object = iterator.next();  // get item
58
59             if ( object instanceof String )   // check for String
60                iterator.set(
61                   ( ( String ) object ).toUpperCase() );
62          }
63       }
64
65       // obtain sublist and use clear method to delete sublist items
66       public void removeItems( List list, int start, int end )
67       {
68          list.subList( start, end ).clear();  // remove items
69       }
70
```

(Part 2)

Lines 41-49

Use **List** method get to obtain object in **LinkedList**, then print its value

Use **ListIterator** to traverse **LinkedList** elements and convert them to upper case (if elements are **String**s)

Use **List** method **subList** and **clear** methods to remove **LinkedList** elements

```java
   // execute application
   public static void main( String args[] )
   {
      new ListTest();
   }

}  // end class ListTest
```

```
                                  list:
black yellow green blue violet silver gold white brown blue gray silver

                                  list:
BLACK YELLOW GREEN BLUE VIOLET SILVER GOLD WHITE BROWN BLUE GRAY SILVER

                       Deleting elements 4 to 6...
                                  list:
         BLACK YELLOW GREEN BLUE WHITE BROWN BLUE GRAY SILVER
```

```
1  // Fig. 21.5: UsingToArray.java
2  // Using method toArray
3
4  // Java core packages
5  import java.util.*;
6
7  public class UsingToArray {
8
9     // create LinkedList, add elements and convert to array
10    public UsingToArray()
11    {
12       LinkedList links;
13       String colors[] = { "black", "blue", "yellow" };
14
15       links = new LinkedList( Arrays.asList( colors ) );
16
17       links.addLast( "red" );   // add as last item
18       links.add( "pink" );      // add to the end
19       links.add( 3, "green" );  // add at 3rd index
20       links.addFirst( "cyan" ); // add as first item
21
22       // get LinkedList elements as an array
23       colors = ( String [] ) links.toArray(
24          new String[ links.size() ] );
25
26       System.out.println( "colors: " );
27
28       for ( int count = 0; count < colors.length; count++ )
29          System.out.println( colors[ count ] );
30    }
31
```

Use **List** method **toArray** to obtain array representation of **LinkedList**

Fig. 21.5 Using method toArray.

Lines 23-24

```
   // execute application
   public static void main( String args[] )
   {
      new UsingToArray();
   }

}  // end class UsingToArray
```

```
colors:
 cyan
black
 blue
yellow
green
 red
 pink
```

Collections Class

- Collections Framework provides set of algorithms
  - Implemented as **static** methods
    - **List** algorithms
      - **sort**
      - **binarySearch**
      - **reverse**
      - **shuffle**
      - **fill**
      - **copy**
    - **Collection** algorithms
      - **min**
      - **max**

Algorithm sort

- **sort**
  - Sorts **List** elements
    - Order is determined by natural order of elements' type
    - Relatively fast

```
1  // Fig. 21.6: Sort1.java
2  // Using algorithm sort
3
4  // Java core packages
5  import java.util.*;
6
7  public class Sort1 {
8     private static String suits[] =
9        { "Hearts", "Diamonds", "Clubs", "Spades" };
10
11    // display array elements
12    public void printElements()
13    {
14       // create ArrayList
15       ArrayList list = new ArrayList( Arrays.asList( suits ) );
16
17       // output list
18       System.out.println( "Unsorted array elements:\n" + list );
19
20       // sort ArrayList
21       Collections.sort( list );
22
23       // output list
24       System.out.println( "Sorted array elements:\n" + list );
25    }
26
27    // execute application
28    public static void main( String args[] )
29    {
30       new Sort1().printElements();
31    }
32
33 }  // end class Sort1
```

Fig. 21.6 Using algorithm `sort`.

Line 15

Line 21

Create **`ArrayList`**

Use **`Collections`** method **`sort`** to sort **`ArrayList`**

Fig. 21.6 Using algorithm `sort`. (Part 2)

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

```
1  // Fig. 21.7: Sort2.java
2  // Using a Comparator object with algorithm sort
3
4  // Java core packages
5  import java.util.*;
6
7  public class Sort2 {
8     private static String suits[] =
9        { "Hearts", "Diamonds", "Clubs", "Spades" };
10
11    // output List elements
12    public void printElements()
13    {
14       // create List
15       List list = Arrays.asList( suits );
16
17       // output List elements
18       System.out.println( "Unsorted array elements:\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.println( "Sorted list elements:\n" + list );
25    }
26
27    // execute application
28    public static void main( String args[] )
29    {
30       new Sort2().printElements();
31    }
32
33 }  // end class Sort2
```

Fig. 21.7 Using a `Comparator` object in `sort`.

Line 21

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`**

Fig. 21.7 Using a `Comparator` object in `sort`. (Part 2)

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

Algorithm shuffle

- **shuffle**
  - Randomly orders **List** elements

```java
// Fig. 21.8: Cards.java
// Using algorithm shuffle

// Java core packages
import java.util.*;

// class to represent a Card in a deck of cards
class Card {
   private String face;
   private String suit;

   // initialize a Card
   public Card( String initialface, String initialSuit )
   {
      face = initialface;
      suit = initialSuit;
   }

   // return face of Card
   public String getFace()
   {
      return face;
   }

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

   // return String representation of Card
   public String toString()
   {
      StringBuffer buffer =
         new StringBuffer( face + " of " + suit );
```

Fig. 21.8 Card shuffling and dealing example.

```
36
37       buffer.setLength( 20 );
38
39       return buffer.toString();
40    }
41
42 }  // end class Card
43
44 // class Cards definition
45 public class Cards {
46    private static String suits[] =
47       { "Hearts", "Clubs", "Diamonds", "Spades" };
48    private static String faces[] = { "Ace", "Deuce", "Three",
49       "Four", "Five", "Six", "Seven", "Eight", "Nine", "Ten",
50       "Jack", "Queen", "King" };
51    private List list;
52
53    // set up deck of Cards and shuffle
54    public Cards()
55    {
56       Card deck[] = new Card[ 52 ];
57
58       for ( int count = 0; count < deck.length; count++ )
59          deck[ count ] = new Card( faces[ count % 13 ],
60             suits[ count / 13 ] );
61
62       list = Arrays.asList( deck );   // get List
63       Collections.shuffle( list );    // shuffle deck
64    }
65
66    // output deck
67    public void printCards()
68    {
69       int half = list.size() / 2 - 1;
70
```

Fig. 21.8 Card shuffling and dealing example. (Part 2)

Line 63

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

Fig. 21.8 Card shuffling and dealing example. (Part 3)

```
      for ( int i = 0, j = half; i <= half; i++, j++ )
         System.out.println(
            list.get( i ).toString() + list.get( j ) );
   }

   // execute application
   public static void main( String args[] )
   {
      new Cards().printCards();
   }

}  // end class Cards
```

Fig. 21.8 Card shuffling and dealing example. (Part 4)

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

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. 21.9: Algorithms1.java
2  // Using algorithms reverse, fill, copy, min and max
3
4  // Java core packages
5  import java.util.*;
6
7  public class Algorithms1 {
8     private String letters[] = { "P", "C", "M" }, lettersCopy[];
9     private List list, copyList;
10
11    // create a List and manipulate it with algorithms from
12    // class Collections
13    public Algorithms1()
14    {
15       list = Arrays.asList( letters );     // get List
16       lettersCopy = new String[ 3 ];
17       copyList = Arrays.asList( lettersCopy );
18
19       System.out.println( "Printing initial statistics: " );
20       printStatistics( list );
21
22       Collections.reverse( list );         // reverse order
23       System.out.println( "\nPrinting statistics afte
24          "calling reverse: " );
25       printStatistics( list );
26
27       Collections.copy( copyList, list );  // copy List
28       System.out.println( "\nPrinting statistics after " +
29          "copying: " );
30       printStatistics( copyList );
31
32       System.out.println( "\nPrinting statistics af
33          "calling fill: " );
34       Collections.fill( list, "R" );
35       printStatistics( list );
```

Fig. 21.9 Using algorithms `reverse`, `fill`, `copy`, `max` and `min`.

Line 22

Line 27

Line 34

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

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

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

```
36     }
37
38     // output List information
39     private void printStatistics( List listRef )
40     {
41        System.out.print( "The list is: " );
42
43        for ( int k = 0; k < listRef.size(); k++ )
44           System.out.print( listRef.get( k ) + " " );
45
46        System.out.print( "\nMax: " + Collections.max( listRef ) );
47        System.out.println(
48           "  Min: " + Collections.min( listRef ) );
49     }
50
51     // execute application
52     public static void main( String args[] )
53     {
54        new Algorithms1();
55     }
56
57  }  // end class Algorithms1
```

```
Printing initial statistics:
The list is: P C M
Max: P  Min: C
Printing statistics after calling reverse:
The list is: M C P
Max: P  Min: C
Printing statistics after copying:
The list is: M C P
Max: P  Min: C
Printing statistics after calling fill:
The list is: R R R
Max: R  Min: R
```

Fig. 21.9 Using algorithms reverse,

Obtain maximum value in **List**

Line 46

Line 48

Obtain minimum value in **List**

Algorithm binarySearch

- **binarySearch**
  - Locates **Object** in **List**
    - Returns index of **Object** in **List** if **Object** exists
    - Returns negative value if **Object** does not exist

```java
1  // Fig. 21.10: BinarySearchTest.java
2  // Using algorithm binarySearch
3
4  // Java core packages
5  import java.util.*;
6
7  public class BinarySearchTest {
8     private String colors[] = { "red", "white", "blue", "black",
9        "yellow", "purple", "tan", "pink" };
10    private ArrayList list;         // ArrayList reference
11
12    // create, sort and output list
13    public BinarySearchTest()
14    {
15       list = new ArrayList( Arrays.asList( colors ) );
16       Collections.sort( list );   // sort the ArrayList
17       System.out.println( "Sorted ArrayList: " + list );
18    }
19
20    // search list for various values
21    public void printSearchResults()
22    {
23       printSearchResultsHelper( colors[ 3 ] ); // first item
24       printSearchResultsHelper( colors[ 0 ] ); // middle item
25       printSearchResultsHelper( colors[ 7 ] ); // last item
26       printSearchResultsHelper( "aardvark" );  // below lowest
27       printSearchResultsHelper( "goat" );      // does not exist
28       printSearchResultsHelper( "zebra" );     // does not exist
29    }
30
31    // helper method to perform searches
32    private void printSearchResultsHelper( String key )
33    {
34       int result = 0;
35
```

Sort **List** in ascending order

Fig. 21.10 Using algorithm `binarySearch`.

Line 16

```
36         System.out.println( "\nSearching for: " + key );
37         result = Collections.binarySearch( list, key );
38         System.out.println(
39            ( result >= 0 ? "Found at index " + result :
40            "Not Found (" + result + ")" ) );
41      }
42
43      // execute application
44      public static void main( String args[] )
45      {
46         new BinarySearchTest().printSearchResults();
47      }
48
49   }  // end class BinarySearchTest
```

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

Line 37

```
     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: aardvark
                            Not Found (-1)

                         Searching for: goat
                            Not Found (-3)

                        Searching for: zebra
                            Not Found (-9)
```

Sets

- **Set**
  - **Collection** that contains unique elements
  - **HashSet**
    - Stores elements in hash table
  - **TreeSet**
    - Stores elements in tree

```java
1  // Fig. 21.11: SetTest.java
2  // Using a HashSet to remove duplicates
3
4  // Java core packages
5  import java.util.*;
6
7  public class SetTest {
8     private String colors[] = { "red", "white", "blue",
9        "green", "gray", "orange", "tan", "white", "cyan",
10       "peach", "gray", "orange" };
11
12    // create and output ArrayList
13    public SetTest()
14    {
15       ArrayList list;
16
17       list = new ArrayList( Arrays.asList( colors ) );
18       System.out.println( "ArrayList:  " + list );
19       printNonDuplicates( list );
20    }
21
22    // create set from array to eliminate duplicates
23    public void printNonDuplicates( Collection collection )
24    {
25       // create a HashSet and obtain its iterator
26       HashSet set = new HashSet( collection );
27       Iterator iterator = set.iterator();
28
29       System.out.println( "\nNonduplicates are:  " );
30
31       while ( iterator.hasNext() )
32          System.out.print( iterator.next() + "  " );
33
34       System.out.println();
35    }
```

Fig. 21.11 Using a HashSet to remove duplicates.

Line 26

Lines 31-32

Create HashSet from Collection object

Use Iterator to traverse HashSet and print nonduplicates

```

   // execute application
   public static void main( String args[] )
   {
      new SetTest();
   }

}  // end class SetTest
```

```
ArrayList: [red, white, blue, green, gray, orange, tan, white, cyan,
                        peach, gray, orange]

                         Nonduplicates are:
          orange cyan green tan white blue peach red gray
```

```java
1  // Fig. 21.12: SortedSetTest.java
2  // Using TreeSet and SortedSet
3
4  // Java core packages
5  import java.util.*;
6
7  public class SortedSetTest {
8     private static String names[] = { "yellow", "green", "black",
9        "tan", "grey", "white", "orange", "red", "green" };
10
11    // create a sorted set with TreeSet, then manipulate it
12    public SortedSetTest()
13    {
14       TreeSet tree = new TreeSet( Arrays.asList( names ) );
15
16       System.out.println( "set: " );
17       printSet( tree );
18
19       // get headSet based upon "orange"
20       System.out.print( "\nheadSet (\"orange\"):  " );
21       printSet( tree.headSet( "orange" ) );
22
23       // get tailSet based upon "orange"
24       System.out.print( "tailSet (\"orange\"):  " );
25       printSet( tree.tailSet( "orange" ) );
26
27       // get first and last elements
28       System.out.println( "first: " + tree.first() );
29       System.out.println( "last : " + tree.last() );
30    }
31
32    // output set
33    public void printSet( SortedSet set )
34    {
35       Iterator iterator = set.iterator();
```

Fig. 21.12 Using SortedSets and TreeSets.

Line 14

Line 21

Line 25

Lines 28-29

Create **TreeSet** from **names** array

Use **TreeSet** method **headSet** to get **TreeSet** subset less than **"orange"**

Use **TreeSet** method **tailSet** to get **TreeSet** subset greater than **"orange"**

Methods **first** and **last** obtain smallest and largest **TreeSet** elements, respectively

```
      while ( iterator.hasNext() )
         System.out.print( iterator.next() + " " );

      System.out.println();
   }

   // execute application
   public static void main( String args[] )
   {
      new SortedSetTest();
   }

}  // end class SortedSetTest
```

Use **`Iterator`** to traverse **`HashSet`** and print values

```
                      set:
  black green grey orange red tan white yellow

     headSet ("orange"):  black green grey
tailSet ("orange"):  orange red tan white yellow
                  first: black
                 last : yellow
```

Maps

- **Map**
  - Associates keys to values
  - Cannot contain duplicate keys
    - Called *one-to-one mapping*

```java
1  // Fig. 21.13: MapTest.java
2  // Using a HashMap to store the number of words that
3  // begin with a given letter
4
5  // Java core packages
6  import java.util.*;
7
8  public class MapTest {
9     private static String names[] = { "one", "two", "three",
10       "four", "five", "six", "seven", "two", "ten", "four" };
11
12    // build a HashMap and output contents
13    public MapTest()
14    {
15       HashMap map = new HashMap();
16       Integer i;
17
18       for ( int count = 0; count < names.length; count++ ) {
19          i = ( Integer ) map.get(
20             new Character( names[ count ].charAt( 0 ) ) );
21
22          // if key is not in map then give it value one
23          // otherwise increment its value by 1
24          if ( i == null )
25             map.put(
26                new Character( names[ count ].charAt( 0 ) ),
27                new Integer( 1 ) );
28          else
29             map.put(
30                new Character( names[ count ].charAt( 0 ) ),
31                new Integer( i.intValue() + 1 ) );
32       }
33
34       System.out.println(
35          "\nnumber of words beginning with each letter:    " );
```

Create **HashMap**

Use method **get** to retrieve a **Character** from **HashMap**

Use method **put** to store a **Character** with an **Integer** key in **HashMap**

Fig. 21.13 Using HashMaps and Maps.

Line 15

Lines 19-20

Lines 25-31

Fig. 21.13 Using HashMaps and Maps. (Part 2)

```
         printMap( map );
      }

   // output map contents
   public void printMap( Map mapRef )
   {
      System.out.println( mapRef.toString() );
      System.out.println( "size: " + mapRef.size() );
      System.out.println( "isEmpty: " + mapRef.isEmpty() );
   }

   // execute application
   public static void main( String args[] )
   {
      new MapTest();
   }

}  // end class MapTest
```

```
number of words beginning with each letter:
            {t=4, s=2, o=1, f=3}
                  size: 4
               isEmpty: false
```

Thank you!