```
1  // Fig. 24.1 : UsingArrays.java
2  // Using Java arrays
3  import java.util.*;
4
5  public class UsingArrays {
6     private int intValues[] = { 1, 2, 3, 4, 5, 6 };
7     private double doubleValues[] = { 8.4, 9.3, 0.2, 7.9, 3.4 };
8     private int filledInt[], intValuesCopy[];
9
10    public UsingArrays()
11    {
12       filledInt = new int[ 10 ];
13       intValuesCopy = new int[ intValues.length ];
14       Arrays.fill( filledInt, 7 );   // fill with 7s
15       Arrays.sort( doubleValues );   // sort doubleValues
16       System.arraycopy( intValues, 0, intValuesCopy,
17                         0, intValues.length );
18    }
19
20    public void printArrays()
21    {
22       System.out.print( "doubleValues: " );
23       for ( int k = 0; k < doubleValues.length; k++ )
24          System.out.print( doubleValues[ k ] + " " );
25
26       System.out.print("\nintValues: " );
27       for ( int k = 0; k < intValues.length; k++ )
28          System.out.print( intValues[ k ] + " " );
29
30       System.out.print("\nfilledInt: " );
31       for ( int k = 0; k < filledInt.length; k++ )
32          System.out.print( filledInt[ k ] + " " );
33
34       System.out.print("\nintValuesCopy: " );
35       for ( int k = 0; k < intValuesCopy.length; k++ )
36          System.out.print( intValuesCopy[ k ] + " " );
37
38       System.out.println();
39    }
40
41    public int searchForInt( int value )
42    {
43       return Arrays.binarySearch( intValues, value );
44    }
45
46    public void printEquality()
47    {
48       boolean b = Arrays.equals( intValues, intValuesCopy );
49
50       System.out.println( "intValues " + ( b ? "==" : "!=" )
51                           + " intValuesCopy" );
52
53       b = Arrays.equals( intValues, filledInt );
```

Fig. 24.1 Using methods of class **Arrays** (part 1 of 2).

```
54          System.out.println( "intValues " + ( b ? "==" : "!=" )
55                              + " filledInt" );
56       }
57
58       public static void main( String args[] )
59       {
60          UsingArrays u = new UsingArrays();
61
62          u.printArrays();
63          u.printEquality();
64
65          int n = u.searchForInt( 5 );
66          System.out.println( ( n >= 0 ? "Found 5 at element " + n :
67                              "5 not found" ) + " in intValues" );
68          n = u.searchForInt( 8763 );
69          System.out.println( ( n >= 0 ? "Found 8763 at element "
70                              + n : "8763 not found" )
71                              + " in intValues" );
72       }
73    }
```

```
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. 24.1 Using methods of class **`Arrays`** (part 2 of 2).

```
1  // Fig. 24.2 : UsingAsList.java
2  // Using method asList
3  import java.util.*;
4
5  public class UsingAsList {
6     private String values[] = { "red", "white", "blue" };
7     private List theList;
8
9     public UsingAsList()
10    {
11       theList = Arrays.asList( values );   // get List
12       theList.set( 1, "green" );           // change a value
13    }
14
15    public void printElements()
16    {
17       System.out.print( "List elements : " );
18       for ( int k = 0; k < theList.size(); k++ )
19          System.out.print( theList.get( k ) + " " );
20
21       System.out.print( "\nArray elements: " );
22       for ( int k = 0; k < values.length; k++ )
23          System.out.print( values[ k ] + " " );
24
25       System.out.println();
26    }
27
28    public static void main( String args[] )
29    {
30       new UsingAsList().printElements();
31    }
32 }
```

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

Fig. 24.2 Using `static` method `asList`.

```
1  // Fig. 24.3 : CollectionTest.java
2  // Using the Collection interface
3  import java.util.*;
4  import java.awt.Color;
5
6  public class CollectionTest {
7     private String colors[] = { "red", "white", "blue" };
8
```

Fig. 24.3 Using an **`ArrayList`** to demonstrate **`Collection`** interface features (part 1 of 2).

```
 9     public CollectionTest()
10     {
11        ArrayList aList = new ArrayList();
12
13        aList.add( Color.magenta );     // add a color object
14
15        for ( int k = 0; k < colors.length; k++ )
16           aList.add( colors[ k ] );
17
18        aList.add( Color.cyan );        // add a color object
19
20        System.out.println( "\nArrayList: " );
21        for ( int k = 0; k < aList.size(); k++ )
22           System.out.print( aList.get( k ) + " " );
23
24        removeStrings( aList );
25
26        System.out.println( "\n\nArrayList after calling" +
27                            " removeStrings: " );
28        for ( int k = 0; k < aList.size(); k++ )
29           System.out.print( aList.get( k ) + " " );
30     }
31
32     public void removeStrings( Collection c )
33     {
34        Iterator i = c.iterator();    // get iterator
35
36        while ( i.hasNext() ) // loop while collection has items
37
38           if ( i.next() instanceof String )
39              i.remove();             // remove String object
40     }
41
42     public static void main( String args[] )
43     {
44        new CollectionTest();
45     }
46  }
```

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

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

Fig. 24.3 Using an **`ArrayList`** to demonstrate **`Collection`** interface features (part 2 of 2).

```
1  // Fig. 24.4 : ListTest.java
2  // Using LinkLists
3  import java.util.*;
4
5  public class ListTest {
6     private String colors[] = { "black", "yellow", "green",
7                                 "blue", "violet", "silver" };
8     private String colors2[] = { "gold", "white", "brown",
9                                  "blue", "gray", "silver" };
10
11    public ListTest()
12    {
13       LinkedList link = new LinkedList();
14       LinkedList link2 = new LinkedList();
```

Fig. 24.4 Using **`List`**s and **`ListIterator`**s (part 1 of 3).

```
15
16          for ( int k = 0; k < colors.length; k++ ) {
17             link.add( colors[ k ] );
18             link2.add( colors2[ k ] );   // same length as colors
19          }
20
21          link.addAll( link2 );           // concatenate lists
22          link2 = null;                   // release resources
23
24          printList( link );
25          uppercaseStrings( link );
26          printList( link );
27          System.out.print( "\nDeleting elements 4 to 6..." );
28          removeItems( link, 4, 7 );
29          printList( link );
30       }
31
32       public void printList( List listRef )
33       {
34          System.out.println( "\nlist: " );
35          for ( int k = 0; k < listRef.size(); k++ )
36             System.out.print( listRef.get( k ) + " " );
37
38          System.out.println();
39       }
40
41       public void uppercaseStrings( List listRef2 )
42       {
43          ListIterator listIt = listRef2.listIterator();
44
45          while ( listIt.hasNext() ) {
46             Object o = listIt.next();       // get item
47
48             if ( o instanceof String )      // check for String
49                listIt.set( ( ( String ) o ).toUpperCase() );
50          }
51       }
52
53       public void removeItems( List listRef3, int start, int end )
54       {
55          listRef3.subList( start, end ).clear();  // remove items
56       }
57
58       public static void main( String args[] )
59       {
60          new ListTest();
61       }
62    }
```

Fig. 24.4 Using **`List`**s and **`ListIterator`**s (part 2 of 3).

```
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
```

Fig. 24.4 Using **`List`**s and **`ListIterator`**s (part 3 of 3).

```
1  // Fig. 24.5 : UsingToArray.java
2  // Using method toArray
3  import java.util.*;
4
5  public class UsingToArray {
6
7     public UsingToArray()
8     {
9        LinkedList links;
10       String colors[] = { "black", "blue", "yellow" };
11
12       links = new LinkedList( Arrays.asList( colors ) );
13
14       links.addLast( "red" );   // add as last item
15       links.add( "pink" );      // add to the end
16       links.add( 3, "green" );  // add at 3rd index
17       links.addFirst( "cyan" ); // add as first item
18
19       // get the LinkedList elements as an array
20       colors = ( String [] ) links.toArray( new String[ 0 ] );
21
22       System.out.println( "colors: " );
23       for ( int k = 0; k < colors.length; k++ )
24          System.out.println( colors[ k ] );
25    }
26
27    public static void main( String args[] )
28    {
29       new UsingToArray();
30    }
31 }
```

Fig. 24.5 Using method **`toArray`** (part 1 of 2).

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

Fig. 24.5 Using method **`toArray`** (part 2 of 2).

```
1  // Fig. 24.6 : Sort1.java
2  // Using algorithm sort
3  import java.util.*;
4
5  public class Sort1 {
6     private static String suits[] = { "Hearts", "Diamonds",
7                                       "Clubs", "Spades" };
8
9     public void printElements()
10    {
11       ArrayList theList =
12                   new ArrayList( Arrays.asList( suits ) );
13
14       System.out.println( "Unsorted array elements:\n" +
15                           theList );
16
17       Collections.sort( theList );    // sort the List
18
19       System.out.println( "Sorted array elements:\n" +
20                           theList );
21    }
```

Fig. 24.6 Using algorithm **sort** (part 1 of 2).

```
22
23    public static void main( String args[] )
24    {
25       new Sort1().printElements();
26    }
27 }
```

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

Fig. 24.6 Using algorithm `sort` (part 2 of 2).

```
1  // Fig. 24.7 : Sort2.java
2  // Using a Comparator object with algorithm sort
3  import java.util.*;
4
5  public class Sort2 {
6     private static String suits[] = { "Hearts", "Diamonds",
7                                       "Clubs", "Spades" };
8
9     public void printElements()
10    {
11       List theList = Arrays.asList( suits );   // get List
12       System.out.println( "Unsorted array elements:\n" +
13                           theList );
14
15       // sort in descending order
16       Collections.sort( theList, Collections.reverseOrder() );
17
18       System.out.println( "Sorted list elements:\n" +
19                           theList );
20    }
21
22    public static void main( String args[] )
23    {
24       new Sort2().printElements();
25    }
26 }
```

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

Fig. 24.7 Using a `Comparator` object in `sort`.

```
1  // Fig. 24.8 : Cards.java
2  // Using algorithm shuffle
3  import java.util.*;
4
5  class Card {
6     private String face;
7     private String suit;
8
9     public Card( String face, String suit )
10    {
11       this.face = face;
12       this.suit = suit;
13    }
14
15    public String getFace() { return face; }
16
17    public String getSuit() { return suit; }
18
19    public String toString()
20    {
21      StringBuffer buf = new StringBuffer( face + " of " + suit );
22
23       buf.setLength( 20 );
24       return ( buf.toString() );
25    }
26 }
27
```

Fig. 24.8 Card shuffling and dealing example (part 1 of 3).

```
28 // class Cards definition
29 public class Cards {
30    private static String suits[] = { "Hearts", "Clubs",
31                                      "Diamonds", "Spades" };
32    private static String faces[] = { "Ace", "Deuce", "Three",
33                                      "Four", "Five", "Six",
34                                      "Seven", "Eight", "Nine",
35                                      "Ten", "Jack", "Queen",
36                                      "King" };
37    private List theList;
38
39    public Cards()
40    {
41       Card deck[] = new Card[ 52 ];
42
43       for ( int k = 0; k < deck.length; k++ )
44         deck[ k ] = new Card( faces[ k % 13 ], suits[ k / 13 ] );
45
46       theList = Arrays.asList( deck );   // get List
47       Collections.shuffle( theList );    // shuffle deck
48    }
49
50    public void printCards()
51    {
52       int half = theList.size() / 2 - 1;
53
54       for ( int k = 0, k2 = half; k <= half; k++, k2++ )
55          System.out.println( theList.get( k ).toString() +
56                              theList.get( k2 ) );
57    }
58
59    public static void main( String args[] )
60    {
61       new Cards().printCards();
62    }
63 }
```

Fig. 24.8 Card shuffling and dealing example (part 2 of 3).

```
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
```

Fig. 24.8 Card shuffling and dealing example (part 3 of 3).

```
1  // Fig. 24.9 : Algorithms1.java
2  // Using algorithms reverse, fill, copy, min and max
3  import java.util.*;
4
5  public class Algorithms1 {
6     private String letters[] = { "P", "C", "M" }, lettersCopy[];
7     private List theList, copyList;
8
9     public Algorithms1()
10    {
11       theList = Arrays.asList( letters );    // get List
12       lettersCopy = new String[ 3 ];
13       copyList = Arrays.asList( lettersCopy );
14
```

Fig. 24.9 Using algorithms **`reverse`**, **`fill`**, **`copy`**, **`max`** and **`min`** (part 1 of 2).

```
15         System.out.println( "Printing initial statistics: " );
16         printStatistics( theList );
17
18         Collections.reverse( theList );        // reverse order
19         System.out.println( "\nPrinting statistics after " +
20                             "calling reverse: " );
21         printStatistics( theList );
22
23         Collections.copy( copyList, theList );  // copy List
24         System.out.println( "\nPrinting statistics after " +
25                             "copying: " );
26         printStatistics( copyList );
27
28         System.out.println( "\nPrinting statistics after " +
29                             "calling fill: " );
30         Collections.fill( theList, "R" );
31         printStatistics( theList );
32      }
33
34      private void printStatistics( List listRef )
35      {
36         System.out.print( "The list is: " );
37         for ( int k = 0; k < listRef.size(); k++ )
38            System.out.print( listRef.get( k ) + " " );
39
40        System.out.print( "\nMax: " + Collections.max( listRef ) );
41         System.out.println( "  Min: " +
42                             Collections.min( listRef ) );
43      }
44
45      public static void main( String args[] )
46      {
47         new Algorithms1();
48      }
49   }
```

```
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. 24.9 Using algorithms `reverse`, `fill`, `copy`, `max` and `min` (part 2 of 2).

```
1  // Fig. 24.10 : BinarySearchTest.java
2  // Using algorithm binarySearch
3  import java.util.*;
4
5  public class BinarySearchTest {
6     private String colors[] = { "red", "white", "blue",
7                                 "black", "yellow",
8                                 "purple", "tan", "pink" };
9     private ArrayList aList;        // ArrayList reference
10
11    public BinarySearchTest()
12    {
13       aList = new ArrayList( Arrays.asList( colors ) );
14       Collections.sort( aList );   // sort the ArrayList
15       System.out.println( "Sorted ArrayList: " + aList );
16    }
17
```

Fig. 24.10 Using algorithm **`binarySearch`** (part 1 of 2).

```
18     public void printSearchResults()
19     {
20        printSearchResultsHelper( colors[ 3 ] ); // first item
21        printSearchResultsHelper( colors[ 0 ] ); // middle item
22        printSearchResultsHelper( colors[ 7 ] ); // last item
23        printSearchResultsHelper( "aardvark" );  // below lowest
24        printSearchResultsHelper( "goat" );      // doesnt exist
25        printSearchResultsHelper( "zebra" );     // doesnt exist
26     }
27
28     private void printSearchResultsHelper( String key )
29     {
30        int result = 0;
31
32        System.out.println( "\nSearching for: " + key );
33        result = Collections.binarySearch( aList, key );
34        System.out.println( ( result >= 0 ? "Found at index "
35                              + result
36                              : "Not Found (" + result + ")" ) );
37     }
38
39     public static void main( String args[] )
40     {
41        new BinarySearchTest().printSearchResults();
42     }
43  }
```

```
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)
```

Fig. 24.10 Using algorithm **`binarySearch`** (part 2 of 2).

```
1  // Fig. 24.11 : SetTest.java
2  // Using a HashSet to remove duplicates
3  import java.util.*;
4
5  public class SetTest {
6     private String colors[] = { "red", "white", "blue",
7                                 "green", "gray", "orange",
8                                 "tan", "white", "cyan",
9                                 "peach", "gray", "orange" };
10
11    public SetTest()
12    {
13       ArrayList aList;
14
15       aList = new ArrayList( Arrays.asList( colors ) );
16       System.out.println( "ArrayList: " + aList );
17       printNonDuplicates( aList );
18    }
19
20    public void printNonDuplicates( Collection c )
21    {
22       HashSet ref = new HashSet( c );   // create a HashSet
23       Iterator i = ref.iterator();      // get iterator
24
25       System.out.println( "\nNonduplicates are: " );
26       while ( i.hasNext() )
27          System.out.print( i.next() + " " );
28
29       System.out.println();
30    }
31
```

Fig. 24.11 Using a **HashSet** to remove duplicates (part 1 of 2).

```
32    public static void main( String args[] )
33    {
34       new SetTest();
35    }
36 }
```

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

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

Fig. 24.11 Using a **HashSet** to remove duplicates (part 2 of 2).

```
1  // Fig. 24.12 : SortedSetTest.java
2  // Using TreeSet and SortedSet
3  import java.util.*;
4
5  public class SortedSetTest {
6     private static String names[] = { "yellow", "green", "black",
7                                       "tan", "grey", "white",
8                                      "orange", "red", "green" };
9
10    public SortedSetTest()
11    {
12       TreeSet m = new TreeSet( Arrays.asList( names ) );
13
14       System.out.println( "set: " );
15       printSet( m );
16
```

Fig. 24.12 Using **SortedSet**s and **TreeSet**s (part 1 of 2).

```
17          // get headSet based upon "orange"
18          System.out.print( "\nheadSet (\"orange\"):  " );
19          printSet( m.headSet( "orange" ) );
20
21          // get tailSet based upon "orange"
22          System.out.print( "tailSet (\"orange\"):  " );
23          printSet( m.tailSet( "orange" ) );
24
25          // get first and last elements
26          System.out.println( "first: " + m.first() );
27          System.out.println( "last : " + m.last() );
28       }
29
30       public void printSet( SortedSet setRef )
31       {
32          Iterator i = setRef.iterator();
33
34          while ( i.hasNext() )
35             System.out.print( i.next() + " " );
36
37          System.out.println();
38       }
39
40       public static void main( String args[] )
41       {
42          new SortedSetTest();
43       }
44    }
```

```
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
```

Fig. 24.12 Using **SortedSet**s and **TreeSet**s (part 2 of 2).

```
1  // Fig. 24.13 : MapTest.java
2  // Using a HashMap to store the number of words that
3  // begin with a given letter
4  import java.util.*;
5
6  public class MapTest {
7     private static String names[] = { "one", "two", "three",
8                                       "four", "five", "six",
9                                    "seven", "two", "ten", "four" };
10
11    public MapTest()
12    {
13       HashMap m = new HashMap();
14       Integer i;
15
16       for ( int k = 0; k < names.length; k++ ) {
17          i = ( Integer ) m.get( new Character(
18                                    names[ k ].charAt( 0 ) ) );
19
20          // if key is not in map then give it value one
21          // otherwise increment its value by 1
22          if ( i == null )
23             m.put( new Character( names[ k ].charAt( 0 ) ),
24                    new Integer( 1 ) );
25          else
26             m.put( new Character( names[ k ].charAt( 0 ) ),
27                    new Integer( i.intValue() + 1 ) );
28       }
29
30       System.out.println( "\nnumber of words beginning with "
31                           + "each letter:    " );
32       printMap( m );
33    }
```

Fig. 24.13 Using **HashMap**s and **Map**s (part 1 of 2).

```
34
35     public void printMap( Map mapRef )
36     {
37        System.out.println( mapRef.toString() );
38        System.out.println( "size: " + mapRef.size() );
39        System.out.println( "isEmpty: " + mapRef.isEmpty() );
40     }
41
42     public static void main( String args[] )
43     {
44        new MapTest();
45     }
46  }
```

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

Fig. 24.13 Using **HashMap**s and **Map**s (part 2 of 2).

public static method header
`Collection synchronizedCollection( Collection c )`
`List synchronizedList( List aList )`
`Set synchronizedSet( Set s )`
`SortedSet synchronizedSortedSet( SortedSet s )`
`Map synchronizedMap( Map m )`
`SortedMap synchronizedSortedMap( SortedMap m )`

Fig. 24.14 Synchronization wrapper methods.

public static method header
`Collection unmodifiableCollection( Collection c )`
`List unmodifiableList( List aList )`
`Set unmodifiableSet( Set s )`
`SortedSet unmodifiableSortedSet( SortedSet s )`
`Map unmodifiableMap( Map m )`
`SortedMap unmodifiableSortedMap( SortedMap m )`

Fig. 24.15 Unmodifiable wrapper methods.