19

Collections

I think this is the most extraordinary collection of talent, of human knowledge, that has ever been gathered together at the White House—with the possible exception of when Thomas Jefferson dined alone.

— **John F. Kennedy**

The shapes a bright container can contain!

— **Theodore Roethke**

Journey over all the universe in a map.

— **Miguel de Cervantes**

Not by age but by capacity is wisdom acquired.

— **Titus Maccius Plautus**

It is a riddle wrapped in a mystery inside an enigma.

— **Winston Churchill**

OBJECTIVES

In this chapter you will learn:

- What collections are.
- To use class `Arrays` for array manipulations.
- To use the collections framework (prepackaged data structure) implementations.
- To use collections framework algorithms to manipulate (such as `search`, `sort` and `fill`) collections.
- To use the collections framework interfaces to program with collections polymorphically.
- To use iterators to "walk through" a collection.
- To use persistent hash tables manipulated with objects of class `Properties`.
- To use synchronization and modifiability wrappers.

Outline

19.1 Introduction

- **Java collections framework**
  - **Contain prepackaged data structures, interfaces, algorithms**
  - **Use generics**
  - **Use existing data structures**
    - **Example of code reuse**
  - **Provides reusable componentry**

19.2 Collections Overview

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

- **Collections framework**
  - **Interfaces declare operations for various collection types**
  - **Provide high-performance, high-quality implementations of common data structures**
  - **Enable software reuse**
  - **Enhanced with generics capabilities in J2SE 5.0**
    - **Compile-time type checking**

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

Fig. 19.1 | Some collection framework interfaces.

19.3 Class Arrays

- Class `Arrays`
  - <mark>Provides `static` methods for manipulating arrays</mark>
  - 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

```java
1  // Fig. 19.2: UsingArrays.java
2  // Using Java arrays.
3  import java.util.Arrays;
4
5  public class UsingArrays
6  {
7     private int intArray[] = { 1, 2, 3, 4, 5, 6 };
8     private double doubleArray[] = { 8.4, 9.3, 0.2, 7.9, 3.4 };
9     private int filledIntArray[], intArrayCopy[];
10
11    // constructor initializes arrays
12    public UsingArrays()
13    {
14       filledIntArray = new int[ 10 ]; // create int array with 10 elements
15       intArrayCopy = new int[ intArray.length ];
16
17       Arrays.fill( filledIntArray, 7 ); // fill with 7s
18       Arrays.sort( doubleArray ); // sort doubleArray
19
20       // copy array intArray into array intArrayCopy
21       System.arraycopy( intArray, 0, intArrayCopy,
22          0, intArray.length );
23    } // end UsingArrays constructor
24
```

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 intArray into array intArrayCopy
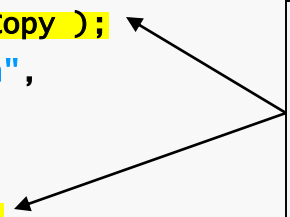
```java
25    // output values in each array
26    public void printArrays()
27    {
28        System.out.print( "doubleArray: " );
29        for ( double doubleValue : doubleArray )
30            System.out.printf( "%.1f ", doubleValue );
31
32        System.out.print( "\nintArray: " );
33        for ( int intValue : intArray )
34            System.out.printf( "%d ", intValue );
35
36        System.out.print( "\nfilledIntArray: " );
37        for ( int intValue : filledIntArray )
38            System.out.printf( "%d ", intValue );
39
40        System.out.print( "\nintArrayCopy: " );
41        for ( int intValue : intArrayCopy )
42            System.out.printf( "%d ", intValue );
43
44        System.out.println( "\n" );
45    } // end method printArrays
46
47    // find value in array intArray
48    public int searchForInt( int value )
49    {
50        return Arrays.binarySearch( intArray, value );
51    } // end method searchForInt
52
```

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

```
53    // compare array contents
54    public void printEquality()
55    {
56       boolean b = Arrays.equals( intArray, intArrayCopy );
57       System.out.printf( "intArray %s intArrayCopy\n",
58          ( b ? "==" : "!=" ) );
59
60       b = Arrays.equals( intArray, filledIntArray );
61       System.out.printf( "intArray %s filledIntArray\n",
62          ( b ? "==" : "!=" ) );
63    } // end method printEquality
64
65    public static void main( String args[] )
66    {
67       UsingArrays usingArrays = new UsingArrays();
68
69       usingArrays.printArrays();
70       usingArrays.printEquality();
71
```

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

```
72        int location = usingArrays.searchForInt( 5 );
73        if ( location >= 0 )
74            System.out.printf(
75                "Found 5 at element %d in intArray\n", location );
76        else
77            System.out.println( "5 not found in intArray" );
78
79        location = usingArrays.searchForInt( 8763 );
80        if ( location >= 0 )
81            System.out.printf(
82                "Found 8763 at element %d in intArray\n", location );
83        else
84            System.out.println( "8763 not found in intArray" );
85    } // end main
86 } // end class UsingArrays
```

```
doubleArray: 0.2 3.4 7.9 8.4 9.3
intArray: 1 2 3 4 5 6
filledIntArray: 7 7 7 7 7 7 7 7 7 7
intArrayCopy: 1 2 3 4 5 6

intArray == intArrayCopy
intArray != filledIntArray
Found 5 at element 4 in intArray
8763 not found in intArray
```

Common Programming Error 19.1

Passing an unsorted array to `binarySearch` is a logic error—the value returned is undefined.

19.4 Interface Collection and Class Collections

- **Interface Collection**
  - **Root interface in the collection hierarchy**
  - **Interfaces Set, Queue, List extend interface Collection**
    - **Set** – collection does not contain duplicates
    - **Queue** – collection represents a waiting line
    - **List** – ordered collection can contain duplicate elements
  - **Contains *bulk operations***
    - **Adding, clearing, comparing and retaining objects**
  - **Provide method to return an Iterator object**
    - **Walk through collection and remove elements from collection**

Software Engineering Observation 19.1

`Collection` is used commonly as a method parameter type to allow polymorphic processing of all objects that implement interface `Collection`.

Software Engineering Observation 19.2

Most collection implementations provide a constructor that takes a `Collection` argument, thereby allowing a new collection to be constructed containing the elements of the specified collection.

19.4 Interface Collection and Class Collections (Cont.)

- **Class Collections**
  - Provides `static` methods that manipulate collections
    - Implement algorithms for searching, sorting and so on
  - Collections can be manipulated polymorphically

- **Synchronized collection**

- **Unmodifiable collection**

19.5 Lists

- **List**

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

Performance Tip 19.1

`ArrayLists` behave like `Vectors` without synchronization and therefore execute faster than `Vectors` because `ArrayLists` do not have the overhead of thread synchronization.

Software Engineering Observation 19.3

LinkedLists can be used to create stacks, queues, trees and deques (double-ended queues, pronounced "decks"). The collections framework provides implementations of some of these data structures.

19.5.1 ArrayList and Iterator

- ## ArrayList example

  - Demonstrate Collection interface capabilities
  - Place two String arrays in ArrayLists
  - Use Iterator to remove elements in ArrayList

```java
1  // Fig. 19.3: CollectionTest.java
2  // Using the Collection interface.
3  import java.util.List;
4  import java.util.ArrayList;
5  import java.util.Collection;
6  import java.util.Iterator;
7
8  public class CollectionTest
9  {
10    private static final String[] colors =
11       { "MAGENTA", "RED", "WHITE", "BLUE", "CYAN" };
12    private static final String[] removeColors =
13       { "RED", "WHITE", "BLUE" };
14
15    // create ArrayList, add Colors to it and manipulate it
16    public CollectionTest()
17    {
18       List< String > list = new ArrayList< String >();
19       List< String > removeList = new ArrayList< String >();
20
```

Create `ArrayList` objects and assign their references to variable `list` and `removeList`, respectively

```
21        // add elements in colors array to list
22        for ( String color : colors )
23           list.add( color );

24
25        // add elements in removeColors to removeList
26        for ( String color : removeColors )
27           removeList.add( color );

28
29        System.out.println( "ArrayList: " );

30
31        // output list contents
32        for ( int count = 0; count < list.size(); count++ )
33           System.out.printf( "%s ", list.get( count ) );

34
35        // remove colors contained in removeList
36        removeColors( list, removeList );

37
38        System.out.println( "\n\nArrayList after calling removeColors: " );

39
40        // output list contents
41        for ( String color : list )
42           System.out.printf( "%s ", color );
43     } // end CollectionTest constructor
44
```

Use **List** method **add** to add objects to **list** and **removeList**, respectively

Use **List** method **size** to get the number of **ArrayList** elements

Use **List** method **get** to retrieve individual element values

Method **removeColors** takes two **Collection**s as arguments; Line 36 passes two **List**s, which extends **Collection**, to this method

```
45    // remove colors specified in collection2 from collection1
46    private void removeColors(
47       Collection< String > collection1, Collection< String > collection2 )
48    {
49       // get iterator
50       Iterator< String > iterator = collection1.iterator();
51
52       // loop while collection has items
53       while ( iterator.hasNext() )
54
55          if ( collection2.contains( iterator.next() ) )
56             iterator.remove(); // remove current Color
57    } // end method removeColors
58
59    public static void main( String args[] )
60    {
61       new CollectionTest();
62    } // end main
63 } // end class CollectionTest
```

```
ArrayList:
MAGENTA RED WHITE BLUE CYAN

ArrayList after calling removeColors:
MAGENTA CYAN
```

Method `removeColors` allows any `Collection`s containing strings to be passed as arguments to this method

Obtain `Collection` iterator

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

`Iterator` method `next` returns a reference to the next element

Collection method `contains` determines whether `collection2` contains the element returned by `next`

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

Common Programming Error 19.2

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

19.5.2 LinkedList

- `LinkedList` example
  - Add elements of one `List` to the other
  - Convert `String`s to uppercase
  - Delete a range of elements

```java
1  // Fig. 19.4: ListTest.java
2  // Using LinkLists.
3  import java.util.List;
4  import java.util.LinkedList;
5  import java.util.ListIterator;
6
7  public class ListTest
8  {
9     private static final String colors[] = { "black", "yellow",
10       "green", "blue", "violet", "silver" };
11    private static final String colors2[] = { "gold", "white",
12       "brown", "blue", "gray", "silver" };
13
14    // set up and manipulate LinkedList objects
15    public ListTest()
16    {
17       List< String > list1 = new LinkedList< String >();
18       List< String > list2 = new LinkedList< String >();
19
20       // add elements to list link
21       for ( String color : colors )
22          list1.add( color );
23
```

Create two
`LinkedList` objects

Use `List` method `add` to append elements from
array `colors` to the end of `list1`

```
24        // add elements to list link2
25        for ( String color : colors2 )
26          list2.add( color );
27
28        list1.addAll( list2 ); // concatenate lists
29        list2 = null; // release resources
30        printList( list1 ); // print list1 elements
31
32        convertToUppercaseStrings( list1 ); // conve
33        printList( list1 ); // print list1 elements
34
35        System.out.print( "\nDeleting elements 4 to 6..." );
36        removeItems( list1, 4, 7 ); // remove items 4-7 from list
37        printList( list1 ); // print list1 elements
38        printReversedList( list1 ); // print list in reverse order
39     } // end ListTest constructor
40
41     // output List contents
42     public void printList( List< String > list )
43     {
44        System.out.println( "\nlist: " );
45
46        for ( String color : list )
47           System.out.printf( "%s ", color );
48
49        System.out.println();
50     } // end method printList
51
```

Use `List` method `add` to append elements from array `colors2` to the end of `list2`

Use `List` method `addAll` to append all elements of `list2` to the end of `list1`

Method `printList` allows any `List`s containing strings to be passed as arguments to this method

```java
52    // locate String objects and convert to uppercase
53    private void convertToUppercaseStrings( List< String > list )
54    {
55       ListIterator< String > iterator = list.listIterator();
56
57       while ( iterator.hasNext() )
58       {
59          String color = iterator.next(); // get item
60          iterator.set( color.t
61       } // end while
62    } // end method convertToUp
63
64    // obtain sublist and use clear method t
65    private void removeItems( List< String >
66    {
67       list.subList( start, end ).clear(); // remove items
68    } // end method removeItems
69
70    // print reversed list
71    private void printReversedList( List
72    {
73       ListIterator< String > iterator = list.listIterator( list.size() );
74
```

Method `convertToUppercaseStrings`

Invoke `List` method `listIterator` to get a bidirectional iterator for the List

Invoke `ListIterator` method `hasNext` to determine whether the List

Invoke `ListIterator` method `next`

Invoke `ListIterator` method `set` to replace the current `String` to which `iterator` refers with the `String` returned by method `toUpperCase`

Method `removeItems` allows any `List`s containing to this method

Invoke `List` method `subList` to obtain a portion of the `List`

Method `printReversedList` allows any `List`s co ... passed as argu...

Invoke `List` method `listIterator` with one argument that specifies the starting position to get a bidirectional iterator for the list

◀ ▶

```
75        System.out.println( "\nReversed List:" );
76
77        // print list in reverse order
78        while ( iterator.hasPrevious() )
79            System.out.printf( "%s ", iterator.previous() );
80    } // end method printReversedList
81
82    public static void main( String args[] )
83    {
84        new ListTest();
85    } // end main
86 } // end class ListTest
```

The `while` condition calls method `hasPrevious` to determine whether there are more elements while traversing the list backward

Invoke `ListIterator` method `previous` to get the previous element from the list

```
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

Reversed List:
SILVER GRAY BLUE BROWN WHITE BLUE GREEN YELLOW BLACK
```

19.5.2 Linkedlist (Cont.)

- **static method asList of class Arrays**
  - View an array as a **List** collection
  - Allow programmer to manipulate the array as if it were a list
  - Any modification made through the **List** view change the array
  - Any modification made to the array change the **List** view

```
1  // Fig. 19.5: UsingToArray.java
2  // Using method toArray.
3  import java.util.LinkedList;
4  import java.util.Arrays;
5
6  public class UsingToArray
7  {
8     // constructor creates LinkedList, adds elements and converts to array
9     public UsingToArray()
10    {
11       String colors[] = { "black", "blue", "yellow" };
12
13       LinkedList< String > links =
14          new LinkedList< String >( Arrays.asList( colors ) );
15
16       links.addLast( "red" );      // add as last item
17       links.add( "pink" );         // add to the end
18       links.add( 3, "green" );     // add at 3rd index
19       links.addFirst( "cyan" );    // add as first item
20
```

Call method asList to create a List view of array colors, which is then used for creating a LinkedList

Call LinkedList method addLast to add "red" to the end of links

Call LinkedList method add to add "pink" as the last element and "green" as the element at index 3

Call LinkedList method addFirst to add "cyan" as the new first item in the LinkedList

```
21        // get LinkedList elements as an array
22        colors = links.toArray( new String[ links.size() ] );

24        System.out.println( "colors: " );

26        for ( String color : colors )
27            System.out.println( color );
28     } // end UsingToArray constructor

30     public static void main( String args[] )
31     {
32         new UsingToArray();
33     } // end main

34  } // end class UsingToArray
```

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

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

Common Programming Error 19.3

Passing an array that contains data to `toArray` can cause logic errors. If the number of elements in the array is smaller than the number of elements in the list on which `toArray` is called, a new array is allocated to store the list's elements—without preserving the array argument's elements. If the number of elements in the array is greater than the number of elements in the list, the elements of the array (starting at index zero) are overwritten with the list's elements. Array elements that are not overwritten retain their values.

19.5.3 `Vector`

- **Class `Vector`**

    – **Array-like data structures that can resize themselves dynamically**

    – **Contains a *capacity***

    – **Grows by *capacity increment* if it requires additional space**

Performance Tip 19.2

Inserting an element into a `Vector` whose current size is less than its capacity is a relatively fast operation.

Performance Tip 19.3

Inserting an element into a `Vector` that needs to grow larger to accommodate the new element is a relatively slow operation.

Performance Tip 19.4

The default capacity increment doubles the size of the `Vector`. This may seem a waste of storage, but it is actually an efficient way for many `Vector`s to grow quickly to be "about the right size." This operation is much more efficient than growing the `Vector` each time by only as much space as it takes to hold a single element. The disadvantage is that the `Vector` might occupy more space than it requires. This is a classic example of the space–time trade-off.

Performance Tip 19.5

If storage is at a premium, use `Vector` method `trimToSize` to trim a `Vector`'s capacity to the `Vector`'s exact size. This operation optimizes a `Vector`'s use of storage. However, adding another element to the `Vector` will force the `Vector` to grow dynamically (again, a relatively slow operation)—trimming leaves no room for growth.

```java
1  // Fig. 19.6: VectorTest.java
2  // Using the Vector class.
3  import java.util.Vector;
4  import java.util.NoSuchElementException;
5
6  public class VectorTest
7  {
8     private static final String colors[] = { "red", "white", "blue" };
9
10    public VectorTest()
11    {
12       Vector< String > vector = new Vector< String >();
13       printVector( vector ); // print vector
14
15       // add elements to the vector
16       for ( String color : colors )
17          vector.add( color );
18
19       printVector( vector ); // print
20
```

Create **Vector** of type **String** with initial capacity of 10 element and capacity increment of zero

Call **Vector** method **add** to add objects (**String**s in this example) to the end of the **Vector**

```java
21    // output the first and last elements
22    try
23    {
24       System.out.printf( "First element: %s\n", vector.firstElement());
25       System.out.printf( "Last element: %s\n", vector.lastElement() );
26    } // end try
27    // catch exception if vector is empty
28    catch ( NoSuchElementException exception )
29    {
30       exception.printStackTrace();
31    } // end catch
32
33    // does vector contain "red"?
34    if ( vector.contains( "red" ) )
35       System.out.printf( "\n\"red\" found at
36          vector.indexOf( "red" ) );
37    else
38       System.out.println( "\n\"red\" n
39
40    vector.remove( "red" );// remove the string "red"
41    System.out.println( "\"red\" has been removed" );
42    printVector( vector ); // print vector
43
```

Call `Vector` method `firstElement` to return a reference to the first element in the `Vector`

Call `Vector` method `lastElement` to return a reference to the last element in the `Vector`

`Vector` method `contains` returns `boolean` that indicates whether `Vector` contains a specific `Object`

`Vector` method `remove` removes the first occurrence of its argument `Object` from `Vector`

`Vector` method `indexOf` returns index of first location in `Vector` containing the argument

```java
44        // does vector contain "red" after remove operation?
45        if ( vector.contains( "red" ) )
46           System.out.printf(
47              "\"red\" found at index %d\n", vector.indexOf( "red" ) );
48        else
49           System.out.println( "\"red\" not found" );
50
51        // print the size and capacity of vector
52        System.out.printf( "\nSize: %d\nCapacity: %d\n", vector.size(),
53           vector.capacity() );
54     } // end Vector constructor
55
56     private void printVector( Vector< String > vectorToOutput )
57     {
58        if ( vectorToOutput.isEmpty() )
59           System.out.print( "vector is empty
60        else  // iterate through the elements
61        {
62           System.out.print( "vector contains
63
64           // output elements
65           for ( String element : vectorToOutput )
66              System.out.printf( "%s ", element );
67        } // end else
68
```

Vector methods size and capacity return number of elements in Vector and Vector capacity, respectively

Method printVector allows any Vectors containing strings to be passed as arguments to this method

Vector method isEmpty returns true if there are no elements in the Vector

```
69        System.out.println( "\n" );
70    } // end method printVector
71
72    public static void main( String args[] )
73    {
74        new VectorTest(); // create object and call its constructor
75    } // end main
76 } // end class VectorTest
```

```
vector is empty

vector contains: red white blue

First element: red
Last element: blue

"red" found at index 0

"red" has been removed
vector contains: white blue

"red" not found

Size: 2
Capacity: 10
```

Common Programming Error 19.4

Without overriding method `equals`, the program performs comparisons using operator == to determine whether two references refer to the same object in memory.

Performance Tip 19.6

`Vector` methods `contains` and `indexOf` perform linear searches of a `Vector`'s contents. These searches are inefficient for large `Vector`s. If a program frequently searches for elements in a collection, consider using one of the Java Collection API's `Map` implementations (Section 19.10), which provide high-speed searching capabilities.

19.6 Collections Algorithms

- **Collections framework provides set of algorithms**
  - **Implemented as `static` methods**
    - `List` algorithms
      - `sort`
      - `binarySearch`
      - `reverse`
      - `shuffle`
      - `fill`
      - `copy`

19.6 Collections Algorithms

- **Collection algorithms**
  - min
  - max
  - addAll
  - frequency
  - disjoint

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.

Fig. 19.7 | Collections algorithms.

Software Engineering Observation 19.4

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

```java
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
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]
```

◄ ►

```java
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]
```

```java
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();
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
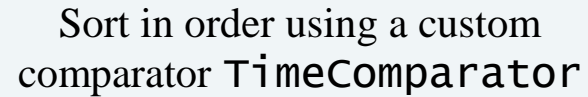
```
1  // Fig. 19.11: Sort3.java
2  // Sort a list using the custom Comparator class TimeComparator.
3  import java.util.List;
4  import java.util.ArrayList;
5  import java.util.Collections;
6
7  public class Sort3
8  {
9     public void printElements()
10    {
11       List< Time2 > list = new ArrayList< 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"
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

```java
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.value
58           {
59              deck[ count ] = new Card( face, sui
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
```

19.6.3 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`

```java
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
```

19.6.4 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**
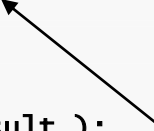   - **Subtract 1 from insertion point**

```java
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)
```

19.6.5 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

```java
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
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    System.out.printf(
42        "\n\nFrequency of red in vector: %d\n", frequ
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
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
```

19.7 Stack Class of Package `java.util`

- **`Stack`**
  - Implements stack data structure
  - Extends class **`Vector`**
  - Stores references to objects

```java
1   // Fig. 19.16: StackTest.java
2   // Program to test java.util.Stack.
3   import java.util.Stack;
4   import java.util.EmptyStackException;
5
6   public class StackTest
7   {
8      public StackTest()
9      {
10        Stack< Number > stack = new Stack< Number >();
11
12        // create numbers to store in the stack
13        Long longNumber = 12L;
14        Integer intNumber = 34567;
15        Float floatNumber = 1.0F;
16        Double doubleNumber = 1234.5678;
17
18        // use push method
19        stack.push( longNumber ); // push a long
20        printStack( stack );
21        stack.push( intNumber ); // push an int
22        printStack( stack );
23        stack.push( floatNumber ); // push a float
24        printStack( stack );
25        stack.push( doubleNumber ); // push a double
26        printStack( stack );
27
```

Create an empty `Stack` of type `Number`

`Stack` method `push` adds object to top of `Stack`

```
28      // remove items from stack
29      try
30      {
31         Number removedObject = null;
32
33         // pop elements from stack
34         while ( true )
35         {
36            removedObject = stack.pop(); // use pop method
37            System.out.printf( "%s popped\n", removedObject );
38            printStack( stack );
39         } // end while
40      } // end try
41      catch ( EmptyStackException emptyStackException )
42      {
43         emptyStackException.printStackTrace();
44      } // end catch
45   } // end StackTest constructor
46
47   private void printStack( Stack< Number > stack )
48   {
49      if ( stack.isEmpty() )
50         System.out.print( "stack is empty\n\n" ); // the stack is empty
51      else  // stack is not empty
52      {
53         System.out.print( "stack contains: " );
54
```

Stack method pop removes element from top of Stack

Stack method isEmpty returns true if Stack is empty

```java
        // iterate through the elements
        for ( Number number : stack )
            System.out.printf( "%s ", number );

        System.out.print( "(top) \n\n" ); // indicates top of the stack
      } // end else
   } // end method printStack

   public static void main( String args[] )
   {
      new StackTest();
   } // end main
} // end class StackTest
```

```
stack contains: 12 (top)

stack contains: 12 34567 (top)

stack contains: 12 34567 1.0 (top)

stack contains: 12 34567 1.0 1234.5678 (top)

1234.5678 popped
stack contains: 12 34567 1.0 (top)

1.0 popped
stack contains: 12 34567 (top)

34567 popped
stack contains: 12 (top)

12 popped
stack is empty

java.util.EmptyStackException
        at java.util.Stack.peek(Unknown Source)
        at java.util.Stack.pop(Unknown Source)
        at StackTest.<init>(StackTest.java:36)
        at StackTest.main(StackTest.java:65)
```

Error-Prevention Tip 19.1

Because `Stack` extends `Vector`, all `public` `Vector` methods can be called on `Stack` objects, even if the methods do not represent conventional stack operations. For example, `Vector` method `add` can be used to insert an element anywhere in a stack—an operation that could "corrupt" the stack. When manipulating a `Stack`, only methods `push` and `pop` should be used to add elements to and remove elements from the `Stack`, respectively.

19.8 Class `PriorityQueue` and Interface `Queue`

- **Interface `Queue`**
  - New collection interface introduced in J2SE 5.0
  - Extends interface `Collection`
  - Provides additional operations for inserting, removing and inspecting elements in a queue

- Class `PriorityQueue`
  - Implements the `Queue` interface
  - Orders elements by their natural ordering
    - Specified by `Comparable` elements' `compareTo` method
    - `Comparator` object supplied through constructor

```
1   // Fig. 19.17: PriorityQueueTest.java
2   // Standard library class PriorityQueue test program.
3   import java.util.PriorityQueue;
4
5   public class PriorityQueueTest
6   {
7      public static void main( String args[] )
8      {
9         // queue of capacity 11
10        PriorityQueue< Double > queue = new PriorityQueue< Double >();
11
12        // insert elements to queue
13        queue.offer( 3.2 );
14        queue.offer( 9.8 );
15        queue.offer( 5.4 );
16
17        System.out.print( "Polling fro
18
19        // display elements in queue
20        while ( queue.size() > 0 )
21        {
22           System.out.printf( "%.1f ", queue.peek() ); //
23           queue.poll(); // remove top element
24        } // end while
25     } // end main
26  } // end class PriorityQueueTe
```

Create a `PriorityQueue` that stores `Double`s with an initial capacity of 11 elements and orders the object's natural ordering

Use method `offer` to add elements to the priority queue

Use method `size` to determine whether the priority queue is empty

Use method `peek` to retrieve the highest-priority element in the queue

Use method `pool` to remove the highest-priority element from the queue

```
Polling from queue: 3.2 5.4 9.8
```

19.9 Sets

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

```java
1  // Fig. 19.18: SetTest.java
2  // Using a HashSet to remove duplicates.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.HashSet;
6  import java.util.Set;
7  import java.util.Collection;
8
9  public class SetTest
10 {
11    private static final String colors[] = { "red", "white", "blue",
12       "green", "gray", "orange", "tan", "white", "cyan",
13       "peach", "gray", "orange" };
14
15    // create and output ArrayList
16    public SetTest()
17    {
18       List< String > list = Arrays.asList( colors );
19       System.out.printf( "ArrayList: %s\n", list );
20       printNonDuplicates( list );
21    } // end SetTest constructor
22
```

Create a List that contains String objects

```
23      // create set from array to eliminate duplicates
24      private void printNonDuplicates( Collection< String > collection )
25      {
26         // create a HashSet
27         Set< String > set = new HashSet< String >( col
28
29         System.out.println( "\nNonduplicates are: " ).
30
31         for ( String s : set )
32            System.out.printf( "%s ", s );
33
34         System.out.println();
35      } // end method printNonDuplicates
36
37      public static void main( String args[] )
38      {
39         new SetTest();
40      } // end main
41   } // end class SetTest
```

Method printNonDuplicates accepts
a Collection of type String

Construct a HashSet from
the Collection argument

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

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

```java
1   // Fig. 19.19: SortedSetTest.java
2   // Using TreeSet and SortedSet.
3   import java.util.Arrays;
4   import java.util.SortedSet;
5   import java.util.TreeSet;
6
7   public class SortedSetTest
8   {
9      private static final String names[] = { "yellow", "green",
10         "black", "tan", "grey", "white", "orange", "red", "green" };
11
12     // create a sorted set with TreeSet, then manipulate it
13     public SortedSetTest()
14     {
15        // create TreeSet
16        SortedSet< String > tree =
17           new TreeSet< String >( Arrays.asList( names ) );
18
19        System.out.println( "sorted set: " );
20        printSet( tree ); // output contents of tree
21
```

Create `TreeSet` from `names` array

```
22        // get headSet based on "orange"
23        System.out.print( "\nheadSet (\"orange\"):  " );
24        printSet( tree.headSet( "orange" ) );
25
26        // get tailSet based upon "orange"
27        System.out.print( "tailSet (\"orange\"):  " );
28        printSet( tree.tailSet( "orange" ) );
29
30        // get first and last elements
31        System.out.printf( "first: %s\n", tree.first() );
32        System.out.printf( "last : %s\n", tree.last() );
33     } // end SortedSetTest constructor
34
35     // output set
36     private void printSet( SortedSet< String > set )
37     {
38        for ( String s : set )
39           System.out.printf( "%s ",  s );
40
```

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

```
41        System.out.println();
42     } // end method printSet
43
44     public static void main( String args[] )
45     {
46        new SortedSetTest();
47     } // end main
48 } // end class SortedSetTest
```

```
sorted 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
```

19.10 Maps

- **`Map`**
  - **Associates keys to values**
  - **Cannot contain duplicate keys**
    - **Called *one-to-one mapping***
  - **Implementation classes**
    - **`Hashtable`, `HashMap`**
      - **Store elements in hash tables**
    - **`TreeMap`**
      - **Store elements in trees**
  - **Interface `SortedMap`**
    - **Extends `Map`**
    - **Maintains its keys in sorted order**

19.10 Maps (Cont.)

- **Map implementation with hash tables**
  - **Hash tables**
    - **Data structure that use *hashing***
      - **Algorithm for determining a *key* in table**
        - **Keys in tables have associated values (data)**
    - **Each table cell is a hash "bucket"**
      - **Linked list of all *key-value pairs* that hash to that cell**
      - **Minimizes *collisions***

Performance Tip 19.7

The load factor in a hash table is a classic example of a memory-space/execution-time trade-off: By increasing the load factor, we get better memory utilization, but the program runs slower, due to increased hashing collisions. By decreasing the load factor, we get better program speed, because of reduced hashing collisions, but we get poorer memory utilization, because a larger portion of the hash table remains empty.

```java
1  // Fig. 19.20: WordTypeCount.java
2  // Program counts the number of occurrences of each word in a string
3  import java.util.StringTokenizer;
4  import java.util.Map;
5  import java.util.HashMap;
6  import java.util.Set;
7  import java.util.TreeSet;
8  import java.util.Scanner;
9
10 public class WordTypeCount
11 {
12    private Map< String, Integer > map;
13    private Scanner scanner;
14
15    public WordTypeCount()
16    {
17       map = new HashMap< String, Integer >(); // create HashMap
18       scanner = new Scanner( System.in ); // create scanner
19       createMap(); // create map based on user input
20       displayMap(); // display map content
21    } // end WordTypeCount constructor
22
```

Create an empty HashMap with a default capacity 16 and a default load factor 0.75. The keys are of type String and the values are of type Integer

```java
23    // create map from user input
24    private void createMap()
25    {
26        System.out.println( "Enter a string:" ); // prompt for user input
27        String input
28
29        // create St
30        StringTokenizer toke
31
32        // processi
33        while ( tok
34        {
35            String word = tokenizer.nextToken().toLowerCase(); // get word
36
37            // if the map contains the word
38            if ( map.containsKey( word ) ) // is word in map
39            {
40                int count = map.get( word ); // get
41                map.put( word, count + 1 ); // in
42            } // end if
43            else
44                map.put( word, 1 ); // add ne
45        } // end while
46    } // end method createMap
47
```

Create a `StringTokenizer` to break the input string argument into its component individual words

Use `StringTokenizer` method `hasMoreTokens` to determine whether there are more tokens in the string

Use `StringTokenizer` method `nextToken` to obtain the next token

Map method `containsKey` determines whether the key specified as an argument is in the hash table

Use method `get` to obtain the key's

Increment the value and use method `put` to replace the key's associated value

Create a new entry in the map, with the word as the key and an `Integer` object containing 1 as the value

```
48    // display map content
49    private void displayMap()
50    {
51       Set< String > keys = map.keySet(); // get
52
53       // sort keys
54       TreeSet< String > sortedKeys = new TreeSet< String >( keys );
55
56       System.out.println( "Map contains:\nK
57
58       // generate output fo
59       for ( String key : so
60          System.out.printf( "%-10s%10s\n", key, map.get( key ) );
61
62       System.out.printf(
63          "\nsize:%d\nisEmpty:%b\n", map.size(), map.isEmpty() );
64    } // end method displayMap
65
```

Use `HashMap` method `keySet` to obtain a set of the keys

Access each key and its value in the map

Call `Map` method `size` to get the number of key-value pairs in the `Map`

Call `Map` method `isEmpty` to determine whether the `Map` is empty

```
66    public static void main( String args[] )
67    {
68       new WordTypeCount();
69    } // end main
70 } // end class WordTypeCount
```

```
Enter a string:
To be or not to be: that is the question Whether 'tis nobler to suffer
Map contains:
Key                 Value
'tis                  1
be                    1
be:                   1
is                    1
nobler                1
not                   1
or                    1
question              1
suffer                1
that                  1
the                   1
to                    3
whether               1

size:13
isEmpty:false
```

19.11 Properties Class

- ## Properties

  - **Persistent `Hashtable`**

    - **Can be written to output stream**

    - **Can be read from input stream**

  - **Provides methods `setProperty` and `getProperty`**

    - **Store/obtain key-value pairs of `String`s**

- ## Preferences API

  - **Replace Properties**

  - **More robust mechanism**

```java
1  // Fig. 19.21: PropertiesTest.java
2  // Demonstrates class Properties of the java.util package.
3  import java.io.FileOutputStream;
4  import java.io.FileInputStream;
5  import java.io.IOException;
6  import java.util.Properties;
7  import java.util.Set;
8
9  public class PropertiesTest
10 {
11     private Properties table;
12
13     // set up GUI to test Properties table
14     public PropertiesTest()
15     {
16         table = new Properties(); // create Properties table
17
18         // set properties
19         table.setProperty( "color", "blue" );
20         table.setProperty( "width", "200" );
21
22         System.out.println( "After setting properties" );
23         listProperties(); // display property values
24
25         // replace property value
26         table.setProperty( "color", "red" );
27
```

Create empty `Properties`

`Properties` method `setProperty` stores value for the specified key

```java
      System.out.println( "After replacing properties" );
      listProperties(); // display property values

      saveProperties(); // save properties

      table.clear(); // empty table

      System.out.println( "After clearing properties" );
      listProperties(); // display property values

      loadProperties(); // load properties

      // get value of property color
      Object value = table.getProperty( "color" );

      // check if value is in table
      if ( value != null )
         System.out.printf( "Property color's value is %s\n", value );
      else
         System.out.println( "Property color is not in table" );
   } // end PropertiesTest constructor
```

Line numbers: 28–49

Use **Properties** method **clear** to empty the hash table

Use **Properties** method **getProperty** to locate the value associated with the specified key

```
50    // save properties to a file
51    public void saveProperties()
52    {
53        // save contents of table
54        try
55        {
56            FileOutputStream output = new FileOutputStream( "props.dat" );
57            table.store( output, "Sample Properties" ); // save properties
58            output.close();
59            System.out.println( "After saving
60            listProperties();
61        } // end try
62        catch ( IOException ioException )
63        {
64            ioException.printStackTrace();
65        } // end catch
66    } // end method saveProperties
67
```

Properties method store saves Properties contents to FileOutputStream

```java
68    // load properties from a file
69    public void loadProperties()
70    {
71       // load contents of table
72       try
73       {
74          FileInputStream input = new FileInputStream( "props.dat" );
75          table.load( input ); // load properties
76          input.close();
77          System.out.println( "After loading prope
78          listProperties(); // display property va
79       } // end try
80       catch ( IOException ioException )
81       {
82          ioException.printStackTrace();
83       } // end catch
84    } // end method loadProperties
85
86    // output property values
87    public void listProperties()
88    {
89       Set< Object > keys = table.keySet(); // get
90
91       // output name/value pairs
92       for ( Object key : keys )
93       {
94          System.out.printf(
95             "%s\t%s\n", key, table.getProperty( ( String ) key ) );
96       } // end for
97
```

Properties method load restores Properties contents from FileInputStream

Use Properties method keySet to obtain a Set of the property names

Obtain the value of a property by passing a key to method getProperty

```
98        System.out.println();
99     } // end method listProperties
100
101    public static void main( String args[] )
102    {
103       new PropertiesTest();
104    } // end main
105} // end class PropertiesTest
```

```
After setting properties
color    blue
width    200

After replacing properties
color    red
width    200

After saving properties
color    red
width    200

After clearing properties

After loading properties
color    red
width    200

Property color's value is red
```

19.12 Synchronized Collections

- **Built-in collections are unsynchronized**
  - **Concurrent access to a `Collection` can cause errors**
  - **Java provides *synchronization wrappers* to avoid this**
    - **Via set of `public static` methods**

public static method headers
< T > Collection< T > synchronizedCollection(Collection< T > c)
< T > List< T > synchronizedList(List< T > aList)
< T > Set< T > synchronizedSet(Set< T > s)
< T > SortedSet< T > synchronizedSortedSet(SortedSet< T > s)
< K, V > Map< K, V > synchronizedMap(Map< K, V > m)
< K, V > SortedMap< K, V > synchronizedSortedMap(SortedMap< K, V > m)

Fig. 19.22 | Synchronization wrapper methods.

19.13 Unmodifiable Collections

- **Unmodifiable wrapper**
  - **Converting collections to unmodifiable collections**
  - **Throw `UnsorrtedOperationException` if attempts are made to modify the collection**

Software Engineering Observation 19.5

You can use an unmodifiable wrapper to create a collection that offers read-only access to others, while allowing read–write access to yourself. You do this simply by giving others a reference to the unmodifiable wrapper while retaining for yourself a reference to the original collection.

public static method headers
< T > Collection< T > unmodifiableCollection(Collection< T > c)
< T > List< T > unmodifiableList(List< T > aList)
< T > Set< T > unmodifiableSet(Set< T > s)
< T > SortedSet< T > unmodifiableSortedSet(SortedSet< T > s)
< K, V > Map< K, V > unmodifiableMap(Map< K, V > m)
< K, V > SortedMap< K, V > unmodifiableSortedMap(SortedMap< K, V > m)

Fig. 19.23 | Unmodifiable wrapper methods.

19.14 Abstract Implementations

- **Abstract implementations**
  - **Offer "bare bones" implementation of collection interfaces**
    - **Programmers can "flesh out" customizable implementations**
  - `AbstractCollection`
  - `AbstractList`
  - `AbstractMap`
  - `AbstractSequentialList`
  - `AbstractSet`
  - `AbstractQueue`