Cape Peninsula University of Technology

Software Design 2
SDN260S

Generic Collections

H. Mataifa
Department of Electronic, Electrical and Computer Engineering

Outline

- Interface Collection and Class Collections
- Lists
- Collections methods
- Class Stack
- Class PriorityQueue
- Sets
- Maps
- Class Properties

Collections

- Java Collections framework: prebuilt data structures, along with interfaces and methods for manipulating them
  - Data structure: a collection of information organized to enable efficient processing (e.g. list, set, map, stack, queue, tree, etc.)
- A Collection is a data structure (an object) that can hold references to other objects

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`	A collection that 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.

Some Collection frameworks in Java

Type-Wrapper Classes

- **Collections** can only manipulate references to objects, thus cannot be used to manipulate primitive-type variables (boolean, byte, character, double, float, integer, long, short)
- Type-wrapper classes: classes allowing primitive types to be manipulated as objects (Boolean, Byte, Character, Double, Float, Integer, Long, Short)
  - Type-wrapper classes enable primitive-type variables to be manipulated by Collections
- Auto-boxing: an automatic conversion from primitive type to corresponding type-wrapper class (e.g. when primitive type variables are assigned to a Collections object)
- Auto-unboxing: automatic conversion from type-wrapper class to corresponding primitive type

```
Integer[] integerArray = new Integer[ 5 ]; // create integerArray
integerArray[ 0 ] = 10; // assign Integer 10 to integerArray[ 0 ]
int value = integerArray[ 0 ]; // get int value of Integer
```

Auto-boxing and auto-unboxing

Interface Collection and Class Collections

- Interface vs. Class: an interface contains the behaviours (abstract methods) that a class implements (cannot be instantiated)
- Interface Collection: the root interface from which interfaces Set, Queue and List are derived
  - Contains bulk operations for adding, clearing, comparing objects in a collection, etc.
- **Set**: defines a collection that contains no duplicates
- **Queue**: defines a collection that represents a waiting line
- **List**: a collection that can contain duplicates
- Class **Collections**: provides static methods for searching, sorting, and performing other operations on collections

Lists

- **List**: a collection that can contain duplicate elements
  - Uses zero-based indexing
- **ListIterator**: an object by means of which List elements can be accessed and manipulated
- Classes inheriting from Interface List (ArrayList, Vector, LinkedList):
  - ArrayList and Vector are resizable-array implementations of List; element insertion is inefficient
  - LinkedList: enables efficient insertion/removal of elements anywhere in the collection
- Auto-boxing occurs when assigning primitive-type values to Lists
- Main difference between Vector and ArrayList:
  - Vector is synchronized by default, ArrayList is not; important for performance and synchronization considerations
- LinkedLists are typically used to create Stacks, Queues

Collections Methods

- List methods:
  - add: adds an item to the end of a List
  - size: returns the number of elements in a List
  - get: retrieves the element from a List for the specified index
- Collection methods:
  - Iterator: gets an Iterator for a Collection
  - contains: determines whether a Collection contains a specified element
- Iterator methods:
  - hasNext: determines whether a Collection contains more elements; returns true if another element exists
  - next: obtains a reference to the next element
  - remove: removes the current element from a Collection

CollectionTest (ArrayList)

```
1  // Fig. 20.2: CollectionTest.java
2  // Collection interface demonstrated via an ArrayList object.
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    public static void main( String[] args )
11    {
12       // add elements in colors array to list
13       String[] colors = { "MAGENTA", "RED", "WHITE", "BLUE", "CYAN" };
14       List< String > list = new ArrayList< String >();
15
16       for ( String color : colors )
17          list.add( color ); // adds color to end of list
18
19       // add elements in removeColors array to removeList
20       String[] removeColors = { "RED", "WHITE", "BLUE" };
21       List< String > removeList = new ArrayList< String >();
22
23       for ( String color : removeColors )
24          removeList.add( color );
25
26       // output list contents
27       System.out.println( "ArrayList: " );
28
29       for ( int count = 0; count < list.size(); count++ )
30          System.out.printf( "%s ", list.get( count ) );
31
32       // remove from list the colors contained in removeList
33       removeColors( list, removeList );
34
35       // output list contents
36       System.out.println( "\n\nArrayList after calling removeColors: " );
37
38       for ( String color : list )
39          System.out.printf( "%s ", color );
40    } // end main
41
42    // remove colors specified in collection2 from collection1
43    private static void removeColors( Collection< String > collection1,
44       Collection< String > collection2 )
45    {
46       // get iterator
47       Iterator< String > iterator = collection1.iterator();
48
49       // loop while collection has items
50       while ( iterator.hasNext() )
51       {
52          if ( collection2.contains( iterator.next() ) )
53             iterator.remove(); // remove current Color
54       } // end while
55    } // end method removeColors
56 } // end class CollectionTest
```

LinkedList

- **LinkedList**: enables efficient insertion/removal of elements
- **List methods**:
  - addAll:
  - ListIterator:
  - subList:
  - clear:
  - toArray:
- **ListIterator methods**:
  - set:
  - hasPrevious:
  - previous:
  - List view:
- Class Arrays method asList:

LinkedList

```
1  // Fig. 20.3: ListTest.java
2  // Lists, LinkedLists and ListIterators.
3  import java.util.List;
4  import java.util.LinkedList;
5  import java.util.ListIterator;
6
7  public class ListTest
8  {
9     public static void main( String[] args )
10    {
11       // add colors elements to list1
12       String[] colors =
13          { "black", "yellow", "green", "blue", "violet", "silver" };
14       List< String > list1 = new LinkedList< String >();
15
16       for ( String color : colors )
17          list1.add( color );
18
19       // add colors2 elements to list2
20       String[] colors2 =
21          { "gold", "white", "brown", "blue", "gray", "silver" };
22       List< String > list2 = new LinkedList< String >();
23
24       for ( String color : colors2 )
25          list2.add( color );
26
27       list1.addAll( list2 ); // concatenate lists
28       list2 = null; // release resources
29       printList( list1 ); // print list1 elements
30
31       convertToUppercaseStrings( list1 ); // convert to uppercase string
32       printList( list1 ); // print list1 elements
33
34       System.out.print( "\nDeleting elements 4 to 6..." );
35       removeItems( list1, 4, 7 ); // remove items 4-6 from list
36       printList( list1 ); // print list1 elements
37       printReversedList( list1 ); // print list in reverse order
38    } // end main
39
```

LinkedList

```
40    // output List contents
41    private static void printList( List< String > list )
42    {
43       System.out.println( "\nlist: " );
44
45       for ( String color : list )
46          System.out.printf( "%s ", color );
47
48       System.out.println();
49    } // end method printList
50
51    // locate String objects and convert to uppercase
52    private static void convertToUppercaseStrings( List< String > list )
53    {
54       ListIterator< String > iterator = list.listIterator();
55
56       while ( iterator.hasNext() )
57       {
58          String color = iterator.next(); // get item
59          iterator.set( color.toUpperCase() ); // convert to upper case
60       } // end while
61    } // end method convertToUppercaseStrings
62
63    // obtain sublist and use clear method to delete sublist items
64    private static void removeItems( List< String > list,
65       int start, int end )
66    {
67       list.subList( start, end ).clear(); // remove items
68    } // end method removeItems
69
70    // print reversed list
71    private static void printReversedList( List< String > list )
72    {
73       ListIterator< String > iterator = list.listIterator( list.size() );
74
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 } // end class ListTest
```

List View of Arrays

```
1  // Fig. 20.4: UsingToArray.java
2  // Viewing arrays as Lists and converting Lists to arrays.
3  import java.util.LinkedList;
4  import java.util.Arrays;
5
6  public class UsingToArray
7  {
8     // creates a LinkedList, adds elements and converts to array
9     public static void main( String[] args )
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
21       // get LinkedList elements as an array
22       colors = links.toArray( new String[ links.size() ] );
23
24       System.out.println( "colors: " );
25
26       for ( String color : colors )
27          System.out.println( color );
28    } // end main
29 } // end class UsingToArray
```

Collections Methods

- Class **Collections** provides several high-performance algorithms (implemented as static methods) for manipulating collection elements
- The **Collections** framework methods are polymorphic; each can operate on objects that implement specific interfaces, regardless of the underlying implementations

Method	Description
`sort`	Sorts the elements of a `List`.
`binarySearch`	Locates an object in a `List`.
`reverse`	Reverses the elements of a `List`.
`shuffle`	Randomly orders a `List`'s elements.
`fill`	Sets every `List` element to refer to a specified object.
`copy`	Copies references from one `List` into another.
`min`	Returns the smallest element in a `Collection`.
`max`	Returns the largest element in a `Collection`.
`addAll`	Appends all elements in an array to a `Collection`.
`frequency`	Calculates how many collection elements are equal to the specified element.
`disjoint`	Determines whether two collections have no elements in common.

Collections methods

Collections Methods

- Method sort:
  - sorts the elements of a list
  - The elements must implement the **Comparable** interface
  - Order is determined by the natural order of the elements' type as implemented by a CompareTo method (declared in interface Comparable)
  - An alternative ordering of elements may be specified in the call to method sort (a **Comparator** object is passed as a second argument)

Collections Methods: sort (ascending)

```
1  // Fig. 20.6: Sort1.java
2  // Collections method sort.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6
7  public class Sort1
8  {
9     public static void main( String[] args )
10    {
11       String[] suits = { "Hearts", "Diamonds", "Clubs", "Spades" };
12
13       // Create and display a list containing the suits array elements
14       List< String > list = Arrays.asList( suits ); // create List
15       System.out.printf( "Unsorted array elements: %s\n", list );
16
17       Collections.sort( list ); // sort ArrayList
18
19       // output list
20       System.out.printf( "Sorted array elements: %s\n", list );
21    } // end main
22 } // end class Sort1
```

Collections Methods: sort (descending)

```
1  // Fig. 20.7: Sort2.java
2  // Using a Comparator object with method sort.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6
7  public class Sort2
8  {
9     public static void main( String[] args )
10    {
11       String[] suits = { "Hearts", "Diamonds", "Clubs", "Spades" };
12
13       // Create and display a list containing the suits array elements
14       List< String > list = Arrays.asList( suits ); // create List
15       System.out.printf( "Unsorted array elements: %s\n", list );
16
17       // sort in descending order using a comparator
18       Collections.sort( list, Collections.reverseOrder() );
19
20       // output List elements
21       System.out.printf( "Sorted list elements: %s\n", list );
22    } // end main
23 } // end class Sort2
```

Custom Comparator

```
1  // Fig. 20.8: 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 time1, Time2 time2 )
8     {
9        int hourCompare = time1.getHour() - time2.getHour(); // compare hour
10
11       // test the hour first
12       if ( hourCompare != 0 )
13          return hourCompare;
14
15       int minuteCompare =
16          time1.getMinute() - time2.getMinute(); // compare minute
17
18       // then test the minute
19       if ( minuteCompare != 0 )
20          return minuteCompare;
21
22       int secondCompare =
23          time1.getSecond() - time2.getSecond(); // compare second
24
25       return secondCompare; // return result of comparing seconds
26    } // end method compare
27 } // end class TimeComparator
```

Collections Methods: sort with custom Comparator

```
1  // Fig. 20.9: Sort3.java
2  // Collections method sort with a custom Comparator object.
3  import java.util.List;
4  import java.util.ArrayList;
5  import java.util.Collections;
6
7  public class Sort3
8  {
9     public static void main( String[] args )
10    {
11       List< Time2 > list = new ArrayList< Time2 >(); // create List
12
13       list.add( new Time2(  6, 24, 34 ) );
14       list.add( new Time2( 18, 14, 58 ) );
15       list.add( new Time2(  6, 05, 34 ) );
16       list.add( new Time2( 12, 14, 58 ) );
17       list.add( new Time2(  6, 24, 22 ) );
18
19       // output List elements
20       System.out.printf( "Unsorted array elements:\n%s\n", list );
21
22       // sort in order using a comparator
23       Collections.sort( list, new TimeComparator() );
24
25       // output List elements
26       System.out.printf( "Sorted list elements:\n%s\n", list );
27    } // end main
28 } // end class Sort3
```

Collections Methods: binarySearch

- Method binarySearch:
  - static method built into the Collections framework for searching through a List (LinkedList, ArrayList)
  - Returns the index of searchObject if found, else returns a negative value

```
1  // Fig. 20.12: BinarySearchTest.java
2  // Collections method 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    public static void main( String[] args )
11    {
12       // create an ArrayList< String > from the contents of colors array
13       String[] colors = { "red", "white", "blue", "black", "yellow",
14          "purple", "tan", "pink" };
15       List< String > list =
16          new ArrayList< String >( Arrays.asList( colors ) );
17
18       Collections.sort( list ); // sort the ArrayList
19       System.out.printf( "Sorted ArrayList: %s\n", list );
20
21       // search list for various values
22       printSearchResults( list, colors[ 3 ] ); // first item
23       printSearchResults( list, colors[ 0 ] ); // middle item
24       printSearchResults( list, colors[ 7 ] ); // last item
25       printSearchResults( list, "aqua" ); // below lowest
26       printSearchResults( list, "gray" ); // does not exist
27       printSearchResults( list, "teal" ); // does not exist
28    } // end main
29
30    // perform search and display result
31    private static void printSearchResults(
32       List< String > list, String key )
33    {
34       int result = 0;
35
36       System.out.printf( "\nSearching for: %s\n", key );
37       result = Collections.binarySearch( list, key );
38
39       if ( result >= 0 )
40          System.out.printf( "Found at index %d\n", result );
41       else
42          System.out.printf( "Not Found (%d)\n",result );
43    } // end method printSearchResults
44 } // end class BinarySearchTest
```

Other Collections Methods

- **Static methods**:
  - shuffle (program 20.10): randomly orders the elements of a List
  - reverse (program 20.11): reverses the order of elements in a List
  - copy (program 20.11): takes two List arguments, source and destination Lists; copies elements from source List to destination List (destination List must be at least as long as source List)
  - fill (program 20.11): overwrites elements in a List with a specified value
  - min/max (program 20.11): returns smallest/largest element in a Collection (can be called with a Comparator object to implement custom comparison)
  - addAll (program 20.13): takes two arguments, inserts elements of second (array) argument into first (Collection) argument
  - frequency (program 20.13): takes two arguments, the Collection to be searched (1st argument) and the Object to search for (2nd argument)
  - disjoint (program 20.13): takes two Collections arguments, checks if they have no element in common, returns true if so

Class Stack

- Class **Stack** (java.util.Stack) extends Class **Vector** to implement a stack data structure
  - A stack data structure is a Last-In, First-Out (**LIFO**) data structure
  - Can be thought of as a pile of dishes; a dish is normally placed on top of the pile; last dish to go onto the pile will also be the first to be taken off
  - A dish is pushed onto the top of the pile, and also popped off the top of the pile
  - Program-execution stack (which a program uses to handle method-calls) is a typical example of the application of the stack data structure

- When manipulating a **Stack**, only methods *push* and *pop* should be used to add elements to and remove elements from the **Stack** respectively (although many other methods of the Class **Vector** can possibly be used)

Class Stack

```
1  // Fig. 20.14: StackTest.java
2  // Stack class of package java.util.
3  import java.util.Stack;
4  import java.util.EmptyStackException;
5
6  public class StackTest
7  {
8     public static void main( String[] args )
9     {
10       Stack< Number > stack = new Stack< Number >(); // create a Stack
11
12       // use push method
13       stack.push( 12L ); // push long value 12L
14       System.out.println( "Pushed 12L" );
15       printStack( stack );
16       stack.push( 34567 ); // push int value 34567
17       System.out.println( "Pushed 34567" );
18       printStack( stack );
19       stack.push( 1.0F ); // push float value 1.0F
20       System.out.println( "Pushed 1.0F" );
21       printStack( stack );
22       stack.push( 1234.5678 ); // push double value 1234.5678
23       System.out.println( "Pushed 1234.5678 " );
24       printStack( stack );
25
26       // remove items from stack
27       try
28       {
29          Number removedObject = null;
30
31          // pop elements from stack
32          while ( true )
33          {
34             removedObject = stack.pop(); // use pop method
35             System.out.printf( "Popped %s\n", removedObject );
36             printStack( stack );
37          } // end while
38       } // end try
39       catch ( EmptyStackException emptyStackException )
40       {
41          emptyStackException.printStackTrace();
42       } // end catch
43    } // end main
44
45    // display Stack contents
46    private static void printStack( Stack< Number > stack )
47    {
48       if ( stack.isEmpty() )
49          System.out.println( "stack is empty\n" ); // the stack is empty
50       else // stack is not empty
51          System.out.printf( "stack contains: %s (top)\n", stack );
52    } // end method printStack
53 } // end class StackTest
```

Class PriorityQueue and Interface Queue

- **Queue**: a collection that represents a waiting line; typically, insertions are made the back of a queue, deletions are made in front (**FIFO** data structure)
- Interface **Queue** extends interface **Collection** and provides additional operations specific to a queue data structure (insertion, removal, inspection of queue)
- Class **PriorityQueue** implements interface **Queue**, and orders elements by priority
  - Can be natural ordering, for elements that implement interface **Comparable**, or custom ordering, using a custom **Comparator** object
  - Provides functionality that enables sorted insertion/removal; elements are always ordered such that highest-priority element will be first to be removed
- Main **PriorityQueue** methods:
  - offer: insert an element appropriately according to priority
  - poll: remove highest-priority element from the queue
  - peek: get a reference to the highest-priority element of the queue
  - size: get number of elements in the queue
  - clear: remove all elements from the queue

Class PriorityQueue and Interface Queue

```
1  // Fig. 20.15: PriorityQueueTest.java
2  // 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 from queue: " );
18
19       // display elements in queue
20       while ( queue.size() > 0 )
21       {
22          System.out.printf( "%.1f ", queue.peek() ); // view top element
23          queue.poll(); // remove top element
24       } // end while
25    } // end main
26 } // end class PriorityQueueTest
```

Sets

- **Set**: an unordered **Collection** of elements, with no duplicates (i.e. unique elements)
- Examples of **Set** implementations in the collections framework:
  - **HashSet**: stores elements in a hash table (a data structure that stores key/value pairs)
  - **TreeSet**: stores elements in a tree
- Interface **SortedSet** extends **Set**, to enable set elements to be sorted
- Class **TreeSet** implements interface **SortedSet**

Sets (HashSet)

```
1  // Fig. 20.16: SetTest.java
2  // HashSet used to remove duplicate values from array of strings.
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    public static void main( String[] args )
12    {
13       // create and display a List< String >
14       String[] colors = { "red", "white", "blue", "green", "gray",
15          "orange", "tan", "white", "cyan", "peach", "gray", "orange" };
16       List< String > list = Arrays.asList( colors );
17       System.out.printf( "List: %s\n", list );
18
19       // eliminate duplicates then print the unique values
20       printNonDuplicates( list );
21    } // end main
22
23    // create a Set from a Collection to eliminate duplicates
24    private static void printNonDuplicates( Collection< String > values )
25    {
26       // create a HashSet
27       Set< String > set = new HashSet< String >( values );
28
29       System.out.print( "\nNonduplicates are: " );
30
31       for ( String value : set )
32          System.out.printf( "%s ", value );
33
34       System.out.println();
35    } // end method printNonDuplicates
36 } // end class SetTest
```

Sets (TreeSet)

```
1  // Fig. 20.17: SortedSetTest.java
2  // Using SortedSets and TreeSets.
3  import java.util.Arrays;
4  import java.util.SortedSet;
5  import java.util.TreeSet;
6
7  public class SortedSetTest
8  {
9     public static void main( String[] args )
10    {
11       // create TreeSet from array colors
12       String[] colors = { "yellow", "green", "black", "tan", "grey",
13          "white", "orange", "red", "green" };
14       SortedSet< String > tree =
15          new TreeSet< String >( Arrays.asList( colors ) );
16
17       System.out.print( "sorted set: " );
18       printSet( tree ); // output contents of tree
19
20       // get headSet based on "orange"
21       System.out.print( "headSet (\"orange\"):   " );
22       printSet( tree.headSet( "orange" ) );
23
24       // get tailSet based upon "orange"
25       System.out.print( "tailSet (\"orange\"):   " );
26       printSet( tree.tailSet( "orange" ) );
27
28       // get first and last elements
29       System.out.printf( "first: %s\n", tree.first() );
30       System.out.printf( "last : %s\n", tree.last() );
31    } // end main
32
33    // output SortedSet using enhanced for statement
34    private static void printSet( SortedSet< String > set )
35    {
36       for ( String s : set )
37          System.out.printf( "%s ", s );
38
39       System.out.println();
40    } // end method printSet
41 } // end class SortedSetTest
```

Maps

- **Maps** associate keys to values; keys must be unique, values need not be
  - A one-to-one mapping has unique key/value pairs; a many-to-one mapping has only keys being unique
- Examples of **Map** implementations in the collections framework:
  - **Hashtable**
  - **HashMap**
  - **TreeMap**
- Interface **SortedMap** extends **Map**, to maintain keys in a sorted order
- **Hashtable**s/**HashMap**s are very useful for efficient storage/retrieval of information when dealing with large datasets (where array indexing may be inefficient)
  - **Hashing**: converting keys to unique array indices that can be used to access the data stored in a **Hashtable**

Maps

```
1  // Fig. 20.18: WordTypeCount.java
2  // Program counts the number of occurrences of each word in a String.
3  import java.util.Map;
4  import java.util.HashMap;
5  import java.util.Set;
6  import java.util.TreeSet;
7  import java.util.Scanner;
8
9  public class WordTypeCount
10 {
11    public static void main( String[] args )
12    {
13       // create HashMap to store String keys and Integer values
14       Map< String, Integer > myMap = new HashMap< String, Integer >();
15
16       createMap( myMap ); // create map based on user input
17       displayMap( myMap ); // display map content
18    } // end main
19
20    // create map from user input
21    private static void createMap( Map< String, Integer > map )
22    {
23       Scanner scanner = new Scanner( System.in ); // create scanner
24       System.out.println( "Enter a string:" ); // prompt for user input
25       String input = scanner.nextLine();
26
27       // tokenize the input
28       String[] tokens = input.split( " " );
29
30       // processing input text
31       for ( String token : tokens )
32       {
33          String word = token.toLowerCase(); // get lowercase word
34
35          // if the map contains the word
36          if ( map.containsKey( word ) ) // is word in map
37          {
38             int count = map.get( word ); // get current count
39             map.put( word, count + 1 ); // increment count
40          } // end if
41          else
42             map.put( word, 1 ); // add new word with a count of 1 to map
43       } // end for
44    } // end method createMap
45
```

Maps

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

Class Properties

- **Properties** object: a persistent **Hashtable** that normally stores key/value pairs of strings
  - Persistent because it can be written to/read from a file
  - Extends Class **Hashtable**<Object, Object>
- Common use of **Properties** object in Java has been to maintain *application-configuration data* or *user preferences* for applications (**Preferences API** does this in newer versions of Java)

Class Properties

```
1  // Fig. 20.19: 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    public static void main( String[] args )
12    {
13       Properties table = new Properties(); // create Properties table
14
15       // set properties
16       table.setProperty( "color", "blue" );
17       table.setProperty( "width", "200" );
18
19       System.out.println( "After setting properties" );
20       listProperties( table ); // display property values
21
22       // replace property value
23       table.setProperty( "color", "red" );
24
25       System.out.println( "After replacing properties" );
26       listProperties( table ); // display property values
27
28       saveProperties( table ); // save properties
29
30       table.clear(); // empty table
31
32       System.out.println( "After clearing properties" );
33       listProperties( table ); // display property values
34
35       loadProperties( table ); // load properties
36
37       // get value of property color
38       Object value = table.getProperty( "color" );
39
40       // check if value is in table
41       if ( value != null )
42          System.out.printf( "Property color's value is %s\n", value );
43       else
44          System.out.println( "Property color is not in table" );
45    } // end main
46
```

Class Properties

```
47    // save properties to a file
48    private static void saveProperties( Properties props )
49    {
50       // save contents of table
51       try
52       {
53          FileOutputStream output = new FileOutputStream( "props.dat" );
54          props.store( output, "Sample Properties" ); // save properties
55          output.close();
56          System.out.println( "After saving properties" );
57          listProperties( props ); // display property values
58       } // end try
59       catch ( IOException ioException )
60       {
61          ioException.printStackTrace();
62       } // end catch
63    } // end method saveProperties
64
65    // load properties from a file
66    private static void loadProperties( Properties props )
67    {
68       // load contents of table
69       try
70       {
71          FileInputStream input = new FileInputStream( "props.dat" );
72          props.load( input ); // load properties
73          input.close();
74          System.out.println( "After loading properties" );
75          listProperties( props ); // display property values
76       } // end try
77       catch ( IOException ioException )
78       {
79          ioException.printStackTrace();
80       } // end catch
81    } // end method loadProperties
82
83    // output property values
84    private static void listProperties( Properties props )
85    {
86       Set< Object > keys = props.keySet(); // get property names
87
88       // output name/value pairs
89       for ( Object key : keys )
90          System.out.printf(
91             "%s\t%s\n", key, props.getProperty( ( String ) key ) );
92
93       System.out.println();
94    } // end method listProperties
95 } // end class PropertiesTest
```