# Generic Collections – Part II

## Method `sort` (cont.)

- Figure  creates a custom `Comparator` class, named `TimeComparator`, that implements interface `Comparator` to compare two `Time2` objects.
- Class `Time2`, declared in Fig. 8.5, represents times with hours, minutes and seconds.
- Class `TimeComparator` implements interface `Comparator`, a generic type that takes one type argument.
- A class that implements `Comparator` must declare a `compare` method that receives two arguments and returns a negative integer if the first argument is less than the second, 0 if the arguments are equal or a positive integer if the first argument is greater than the second.

```java
// Fig. 20.8: TimeComparator.java
// Custom Comparator class that compares two Time2 objects.
import java.util.Comparator;

public class TimeComparator implements Comparator< Time2 >
{
   public int compare( Time2 time1, Time2 time2 )
   {
      int hourCompare = time1.getHour() - time2.getHour(); // compare hour

      // test the hour first
      if ( hourCompare != 0 )
         return hourCompare;

      int minuteCompare =
         time1.getMinute() - time2.getMinute(); // compare minute

      // then test the minute
      if ( minuteCompare != 0 )
         return minuteCompare;

      int secondCompare =
         time1.getSecond() - time2.getSecond(); // compare second
```

Custom Comparator for Time2 objects

**Fig. 20.8** | Custom Comparator class that compares two Time2 objects. (Part 1 of 2.)



```java

      return secondCompare; // return result of comparing seconds
   } // end method compare
} // end class TimeComparator
```

**Fig. 20.8** | Custom Comparator class that compares two Time2 objects. (Part 2 of 2.)

```java
// Fig. 20.9: Sort3.java
// Collections method sort with a custom Comparator object.
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

public class Sort3
{
   public static void main( String[] args )
   {
      List< Time2 > list = new ArrayList< Time2 >(); // create List

      list.add( new Time2(  6, 24, 34 ) );
      list.add( new Time2( 18, 14, 58 ) );
      list.add( new Time2(  6, 05, 34 ) );
      list.add( new Time2( 12, 14, 58 ) );
      list.add( new Time2(  6, 24, 22 ) );

      // output List elements
      System.out.printf( "Unsorted array elements:\n%s\n", list );

```

**Fig. 20.9** | `Collections` method `sort` with a custom `Comparator` object. (Part 1 of 2.)



```java
      // sort in order using a comparator
      Collections.sort( list, new TimeComparator() );

      // output List elements
      System.out.printf( "Sorted list elements:\n%s\n", list );
   } // end main
} // end class Sort3
```

`Time2` objects could not be sorted before creating the `TimeComparator`; technique can be used to make objects of almost any class sortable

```
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]
```

**Fig. 20.9** | `Collections` method `sort` with a custom `Comparator` object. (Part 2 of 2.)

## Stack Class of Package java.util

- Class `Stack` in the Java utilities package (`java-.util`) extends class `Vector` to implement a stack data structure.
- `Stack` method `push` adds a `Number` object to the top of the stack.
- Any integer literal that has the suffix `L` is a `long` value.
- An integer literal without a suffix is an `int` value.
- Any floating-point literal that has the suffix `F` is a `float` value.
- A floating-point literal without a suffix is a `double` value.
- `Stack` method `pop` removes the top element of the stack.
  - If there are no elements in the `Stack`, method `pop` throws an `EmptyStackException`, which terminates the loop.
- Method `peek` returns the top element of the stack without popping the element off the stack.
- Method `isEmpty` determines whether the stack is empty.



**Error-Prevention Tip 20.1**

*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.*

```java
// Fig. 20.14: StackTest.java
// Stack class of package java.util.
import java.util.Stack;
import java.util.EmptyStackException;

public class StackTest
{
   public static void main( String[] args )
   {
      Stack< Number > stack = new Stack< Number >(); // create a Stack

      // use push method
      stack.push( 12L ); // push long value 12L
      System.out.println( "Pushed 12L" );
      printStack( stack );
      stack.push( 34567 ); // push int value 34567
      System.out.println( "Pushed 34567" );
      printStack( stack );
      stack.push( 1.0F ); // push float value 1.0F
      System.out.println( "Pushed 1.0F" );
      printStack( stack );
      stack.push( 1234.5678 ); // push double value 1234.5678
      System.out.println( "Pushed 1234.5678 " );
      printStack( stack );
```

**Fig. 20.14** | Stack class of package `java.util`. (Part 1 of 4.)



```java

      // remove items from stack
      try
      {
         Number removedObject = null;

         // pop elements from stack
         while ( true )
         {
            removedObject = stack.pop(); // use pop method
            System.out.printf( "Popped %s\n", removedObject );
            printStack( stack );
         } // end while
      } // end try
      catch ( EmptyStackException emptyStackException )
      {
         emptyStackException.printStackTrace();
      } // end catch
   } // end main

   // display Stack contents
   private static void printStack( Stack< Number > stack )
   {
```

**Fig. 20.14** | Stack class of package `java.util`. (Part 2 of 4.)

```java
         if ( stack.isEmpty() )
            System.out.println( "stack is empty\n" ); // the stack is empty
         else // stack is not empty
            System.out.printf( "stack contains: %s (top)\n", stack );
      } // end method printStack
   } // end class StackTest
```

**Fig. 20.14** | Stack class of package `java.util`. (Part 3 of 4.)



```
Pushed 12L
stack contains: [12] (top)
Pushed 34567
stack contains: [12, 34567] (top)
Pushed 1.0F
stack contains: [12, 34567, 1.0] (top)
Pushed 1234.5678
stack contains: [12, 34567, 1.0, 1234.5678] (top)
Popped 1234.5678
stack contains: [12, 34567, 1.0] (top)
Popped 1.0
stack contains: [12, 34567] (top)
Popped 34567
stack contains: [12] (top)
Popped 12
stack is empty

java.util.EmptyStackException
        at java.util.Stack.peek(Unknown Source)
        at java.util.Stack.pop(Unknown Source)
        at StackTest.main(StackTest.java:34)
```

**Fig. 20.14** | Stack class of package `java.util`. (Part 4 of 4.)

# Class `PriorityQueue` and Interface `Queue`

- Interface `Queue` extends interface `Collection` and provides additional operations for inserting, removing and inspecting elements in a queue.
- `PriorityQueue` orders elements by their natural ordering.
  - Elements are inserted in priority order such that the highest-priority element (i.e., the largest value) will be the first element removed from the `PriorityQueue`.
- Common `PriorityQueue` operations are
  - `offer` to insert an element at the appropriate location based on priority order
  - `poll` to remove the highest-priority element of the priority queue
  - `peek` to get a reference to the highest-priority element of the priority queue
  - `clear` to remove all elements in the priority queue
  - `size` to get the number of elements in the queue.



```java
// Fig. 20.15: PriorityQueueTest.java
// PriorityQueue test program.
import java.util.PriorityQueue;

public class PriorityQueueTest
{
   public static void main( String[] args )
   {
      // queue of capacity 11
      PriorityQueue< Double > queue = new PriorityQueue< Double >();

      // insert elements to queue
      queue.offer( 3.2 );
      queue.offer( 9.8 );
      queue.offer( 5.4 );

      System.out.print( "Polling from queue: " );

```

Natural order determines priority

**Fig. 20.15** | PriorityQueue test program. (Part 1 of 2.)

```java
        // display elements in queue
        while ( queue.size() > 0 )
        {
           System.out.printf( "%.1f ", queue.peek() ); // view top element
           queue.poll(); // remove top element
        } // end while
     } // end main
  } // end class PriorityQueueTest
```

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

**Fig. 20.15** | PriorityQueue test program. (Part 2 of 2.)



## Sets

- A `Set` is an unordered `Collection` of unique elements (i.e., no duplicate elements).
- The collections framework contains several `Set` implementations, including `HashSet` and `TreeSet`.
- `HashSet` stores its elements in a hash table, and `TreeSet` stores its elements in a tree.

```java
// Fig. 20.16: SetTest.java
// HashSet used to remove duplicate values from array of strings.
import java.util.List;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;
import java.util.Collection;

public class SetTest
{
   public static void main( String[] args )
   {
      // create and display a List< String >
      String[] colors = { "red", "white", "blue", "green", "gray",
         "orange", "tan", "white", "cyan", "peach", "gray", "orange" };
      List< String > list = Arrays.asList( colors );
      System.out.printf( "List: %s\n", list );

      // eliminate duplicates then print the unique values
      printNonDuplicates( list );
   } // end main

```

**Fig. 20.16** | `HashSet` used to remove duplicate values from an array of strings. (Part 1 of 2.)



```java
   // create a Set from a Collection to eliminate duplicates
   private static void printNonDuplicates( Collection< String > values )
   {
      // create a HashSet
      Set< String > set = new HashSet< String >( values );

      System.out.print( "\nNonduplicates are: " );

      for ( String value : set )
         System.out.printf( "%s ", value );

      System.out.println();
   } // end method printNonDuplicates
} // end class SetTest
```

Sets don't allow duplicates

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

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

**Fig. 20.16** | `HashSet` used to remove duplicate values from an array of strings. (Part 2 of 2.)

# Sets (cont.)

- The collections framework also includes the SortedSet interface (which extends Set) for sets that maintain their elements in sorted order.
- Class TreeSet implements SortedSet.
- TreeSet method headSet gets a subset of the TreeSet in which every element is less than the specified value.
- TreeSet method tailSet gets a subset in which each element is greater than or equal to the specified value.
- SortedSet methods first and last get the smallest and largest elements of the set, respectively.



```java
// Fig. 20.17: SortedSetTest.java
// Using SortedSets and TreeSets.
import java.util.Arrays;
import java.util.SortedSet;
import java.util.TreeSet;

public class SortedSetTest
{
   public static void main( String[] args )
   {
      // create TreeSet from array colors
      String[] colors = { "yellow", "green", "black", "tan", "grey",
         "white", "orange", "red", "green" };
      SortedSet< String > tree =
         new TreeSet< String >( Arrays.asList( colors ) );

      System.out.print( "sorted set: " );
      printSet( tree ); // output contents of tree

      // get headSet based on "orange"
      System.out.print( "headSet (\"orange\"):  " );
      printSet( tree.headSet( "orange" ) );

```

**Fig. 20.17** | Using SortedSets and TreeSets. (Part 1 of 3.)

```java
      // get tailSet based upon "orange"
      System.out.print( "tailSet (\"orange\"):  " );
      printSet( tree.tailSet( "orange" ) );

      // get first and last elements
      System.out.printf( "first: %s\n", tree.first() );
      System.out.printf( "last : %s\n", tree.last() );
   } // end main

   // output SortedSet using enhanced for statement
   private static void printSet( SortedSet< String > set )
   {
      for ( String s : set )
         System.out.printf( "%s ", s );

      System.out.println();
   } // end method printSet
} // end class SortedSetTest
```

**Fig. 20.17** | Using SortedSets and TreeSets. (Part 2 of 3.)



```
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
```

**Fig. 20.17** | Using SortedSets and TreeSets. (Part 3 of 3.)

## Maps

- Maps associate keys to values.
  - The keys in a `Map` must be unique, but the associated values need not be.
  - If a `Map` contains both unique keys and unique values, it is said to implement a one-to-one mapping.
  - If only the keys are unique, the `Map` is said to implement a many-to-one mapping—many keys can map to one value.
- Three of the several classes that implement interface `Map` are `Hashtable`, `HashMap` and `TreeMap`.
- `Hashtable`s and `HashMap`s store elements in hash tables, and `TreeMap`s store elements in trees.



## Maps (Cont.)

- Interface `SortedMap` extends `Map` and maintains its keys in sorted order—either the elements' natural order or an order specified by a `Comparator`.
- Class `TreeMap` implements `SortedMap`.
- Hashing is a high-speed scheme for converting keys into unique array indices.
- A hash table's load factor affects the performance of hashing schemes.
  - The load factor is the ratio of the number of occupied cells in the hash table to the total number of cells in the hash table.
- The closer this ratio gets to 1.0, the greater the chance of collisions.

**Performance Tip 20.2**

*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
// Fig. 20.18: WordTypeCount.java
// Program counts the number of occurrences of each word in a String.
import java.util.Map;
import java.util.HashMap;
import java.util.Set;
import java.util.TreeSet;
import java.util.Scanner;

public class WordTypeCount
{
   public static void main( String[] args )
   {
      // create HashMap to store String keys and Integer values
      Map< String, Integer > myMap = new HashMap< String, Integer >();

      createMap( myMap ); // create map based on user input
      displayMap( myMap ); // display map content
   } // end main

   // create map from user input
   private static void createMap( Map< String, Integer > map )
   {
```

**Fig. 20.18** | Program counts the number of occurrences of each word in a `String`. (Part 1 of 4.)

```java
      Scanner scanner = new Scanner( System.in ); // create scanner
      System.out.println( "Enter a string:" ); // prompt for user input
      String input = scanner.nextLine();

      // tokenize the input
      String[] tokens = input.split( " " );

      // processing input text
      for ( String token : tokens )
      {
         String word = token.toLowerCase(); // get lowercase word

         // if the map contains the word
         if ( map.containsKey( word ) ) // is word in map
         {
            int count = map.get( word ); // get current count
            map.put( word, count + 1 ); // increment count
         } // end if
         else
            map.put( word, 1 ); // add new word with a count of 1 to map
      } // end for
   } // end method createMap
```

**Fig. 20.18** | Program counts the number of occurrences of each word in a `String`. (Part 2 of 4.)



```java

   // display map content
   private static void displayMap( Map< String, Integer > map )
   {
      Set< String > keys = map.keySet(); // get keys

      // sort keys
      TreeSet< String > sortedKeys = new TreeSet< String >( keys );

      System.out.println( "\nMap contains:\nKey\t\tValue" );

      // generate output for each key in map
      for ( String key : sortedKeys )
         System.out.printf( "%-10s%10s\n", key, map.get( key ) );

      System.out.printf(
         "\nsize: %d\nisEmpty: %b\n", map.size(), map.isEmpty() );
   } // end method displayMap
} // end class WordTypeCount
```

**Fig. 20.18** | Program counts the number of occurrences of each word in a `String`. (Part 3 of 4.)

CSE212 Software Development Methodologies Yeditepe University Spring 2023

```
Enter a string:
this is a sample sentence with several words this is another sample
sentence with several different words

Map contains:
Key             Value
a                  1
another            1
different          1
is                 2
sample             2
sentence           2
several            2
this               2
with               2
words              2

size: 10
isEmpty: false
```

**Fig. 20.18** | Program counts the number of occurrences of each word in a `String`. (Part 4 of 4.)



# Maps (Cont.)

- `Map` method `containsKey` determines whether a key is in a map.
- `Map` method `put` creates a new entry in the map or replaces an existing entry's value.
  - Method `put` returns the key's prior associated value, or `null` if the key was not in the map.
- `Map` method `get` obtain the specified key's associated value in the map.
- `HashMap` method `keySet` returns a set of the keys.
- `Map` method `size` returns the number of key/value pairs in the `Map`.
- `Map` method `isEmpty` returns a `boolean` indicating whether the `Map` is empty.

## Properties Class

- A `Properties` object is a persistent `Hashtable` that normally stores key/value pairs of strings—assuming that you use methods `setProperty` and `getProperty` to manipulate the table rather than inherited `Hashtable` methods `put` and `get`.
- The `Properties` object's contents can be written to an output stream (possibly a file) and read back in through an input stream.
- A common use of `Properties` objects in prior versions of Java was to maintain application-configuration data or user preferences for applications.
  - [Note: The Preferences API (package `java.util.prefs`) is meant to replace this use of class `Properties`.]



## Properties Class (cont.)

- `Properties` method `store` saves the object's contents to the `OutputStream` specified as the first argument. The second argument, a `String`, is a description written into the file.
- `Properties` method `list`, which takes a `PrintStream` argument, is useful for displaying the list of properties.
- `Properties` method `load` restores the contents of a `Properties` object from the `InputStream` specified as the first argument (in this case, a `FileInputStream`).

CSE212 Software Development Methodologies Yeditepe University Spring 2023

```java
// Fig. 20.19: PropertiesTest.java
// Demonstrates class Properties of the java.util package.
import java.io.FileOutputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.Properties;
import java.util.Set;

public class PropertiesTest
{
   public static void main( String[] args )
   {
      Properties table = new Properties(); // create Properties table

      // set properties
      table.setProperty( "color", "blue" );
      table.setProperty( "width", "200" );

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

      // replace property value
      table.setProperty( "color", "red" );
```

**Fig. 20.19** | Properties class of package java.util. (Part 1 of 5.)



```java

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

      saveProperties( table ); // save properties

      table.clear(); // empty table

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

      loadProperties( table ); // 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 main

```

**Fig. 20.19** | Properties class of package java.util. (Part 2 of 5.)

```java
   // save properties to a file
   private static void saveProperties( Properties props )
   {
      // save contents of table
      try
      {
         FileOutputStream output = new FileOutputStream( "props.dat" );
         props.store( output, "Sample Properties" ); // save properties
         output.close();
         System.out.println( "After saving properties" );
         listProperties( props ); // display property values
      } // end try
      catch ( IOException ioException )
      {
         ioException.printStackTrace();
      } // end catch
   } // end method saveProperties

   // load properties from a file
   private static void loadProperties( Properties props )
   {
      // load contents of table
      try
      {
```

**Fig. 20.19** | `Properties` class of package `java.util`. (Part 3 of 5.)



```java
         FileInputStream input = new FileInputStream( "props.dat" );
         props.load( input ); // load properties
         input.close();
         System.out.println( "After loading properties" );
         listProperties( props ); // display property values
      } // end try
      catch ( IOException ioException )
      {
         ioException.printStackTrace();
      } // end catch
   } // end method loadProperties

   // output property values
   private static void listProperties( Properties props )
   {
      Set< Object > keys = props.keySet(); // get property names

      // output name/value pairs
      for ( Object key : keys )
         System.out.printf(
            "%s\t%s\n", key, props.getProperty( ( String ) key ) );

      System.out.println();
   } // end method listProperties
} // end class PropertiesTest
```

**Fig. 20.19** | `Properties` class of package `java.util`. (Part 4 of 5.)

CSE212 Software Development Methodologies Yeditepe University Spring 2023

```
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
```

**Fig. 20.19** | `Properties` class of package `java.util`. (Part 5 of 5.)



## Synchronized Collections

- Synchronization wrappers are used for collections that might be accessed by multiple threads.
- A wrapper object receives method calls, adds thread synchronization and delegates the calls to the wrapped collection object.
- The `Collections` API provides a set of `static` methods for wrapping collections as synchronized versions.
- Method headers for the synchronization wrappers are listed in Fig. 20.20.

| 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. 20.20** | Synchronization wrapper methods.



# Unmodifiable Collections

- The `Collections` class provides a set of `static` methods that create unmodifiable wrappers for collections.
- Unmodifiable wrappers throw `UnsupportedOperationException`s if attempts are made to modify the collection.
- Headers for these methods are listed in Fig. 20.21.

CSE212 Software Development Methodologies Yeditepe University Spring 2023

**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. 20.21** | Unmodifiable wrapper methods.



**Software Engineering Observation 20.5**

*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.*

# Abstract Implementations

- The collections framework provides various abstract implementations of `Collection` interfaces from which you can quickly "flesh out" complete customized implementations.
- These include
  - a thin `Collection` implementation called an `AbstractCollection`
  - a `List` implementation that allows random access to its elements called an `AbstractList`
  - a `Map` implementation called an `AbstractMap`
  - a `List` implementation that allows sequential access to its elements called an `AbstractSequentialList`
  - a `Set` implementation called an `AbstractSet`
  - a `Queue` implementation called `AbstractQueue`.