Chapter 16
Generic Collections

Java How to Program, 10/e

OBJECTIVES

In this chapter you'll:

- Learn what collections are.
- Use class `Arrays` for array manipulations.
- Learn the type-wrapper classes that enable programs to process primitive data values as objects.
- Use prebuilt generic data structures from the collections framework.
- Use iterators to "walk through" a collection.
- Use persistent hash tables manipulated with objects of class `Properties`.
- Learn about synchronization and modifiability wrappers.

16.1 Introduction

16.2 Collections Overview

16.3 Type-Wrapper Classes

16.4 Autoboxing and Auto-Unboxing

16.5 Interface **Collection** and Class **Collections**

16.6 Lists

16.6.1 **ArrayList** and **Iterator**

16.6.2 **LinkedList**

16.7 Collections Methods

16.7.1 Method **sort**

16.7.2 Method **shuffle**

16.7.3 Methods **reverse**, **fill**, **copy**, **max** and **min**

16.7.4 Method **binarySearch**

16.7.5 Methods **addAll**, **frequency** and **disjoint**

16.8 **Stack** Class of Package **java.util**

16.9 Class **PriorityQueue** and Interface **Queue**

16.10 Sets

16.11 Maps
16.12 `Properties` Class
16.13 Synchronized Collections
16.14 Unmodifiable Collections
16.15 Abstract Implementations
16.16 Wrap-Up

16.1 Introduction

- **What is a Framework?**
  - A framework is a set of classes and interfaces which provide a ready-made architecture.
  - Java collections framework
    - Contains *prebuilt* generic data structures

16.2 Collections Overview

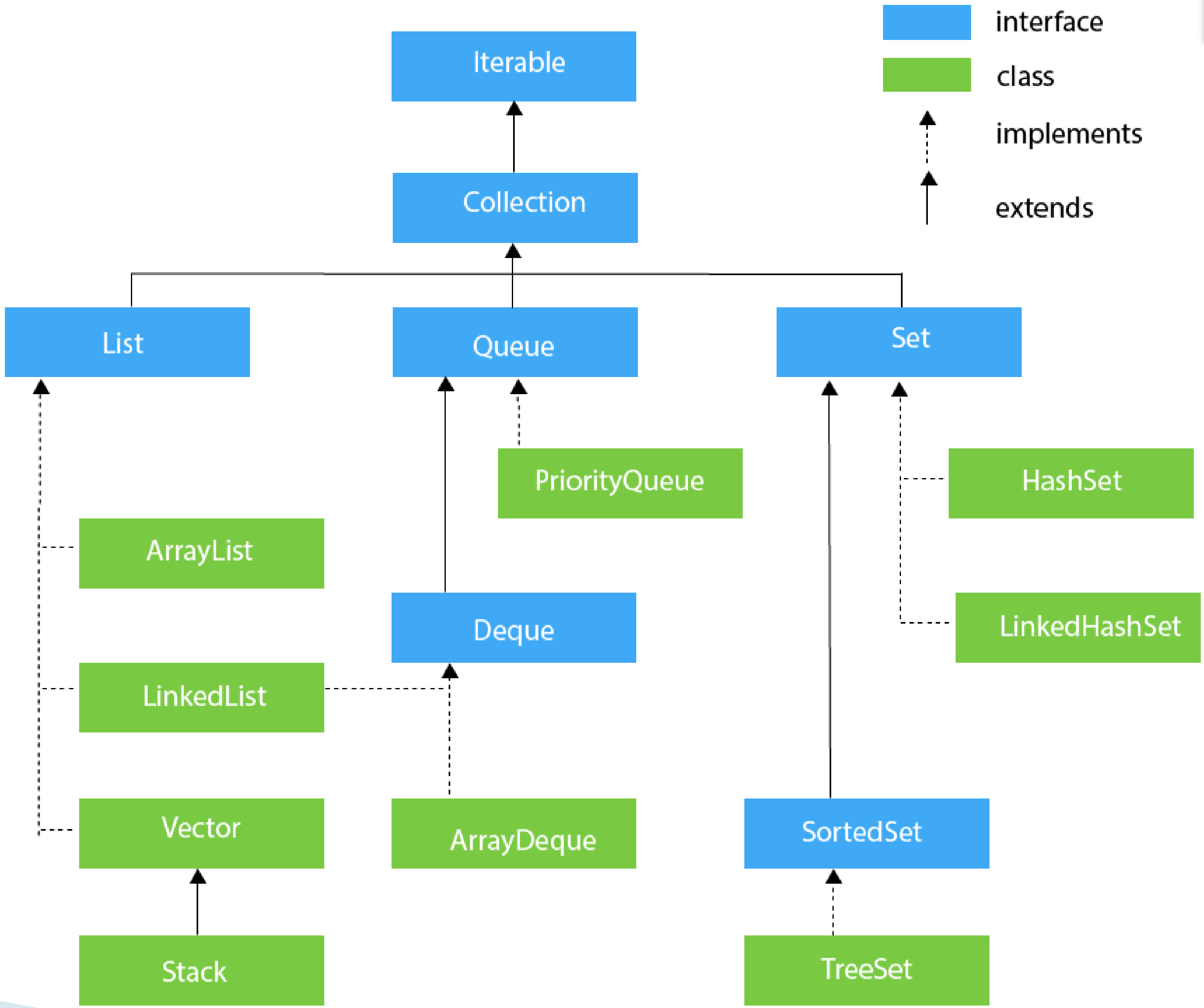
- A collection is a data structure—actually, an object—that can hold references to other objects.
  - Usually, collections contain references to objects of any type that has the *is-a* relationship with the type stored in the collection.
- Figure 16.1 lists some of the collections framework interfaces.
- Package `java.util`.

- A *collections framework* is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:
  - **Interfaces**
  - **Implementations**
  - **Algorithms**

Using Collections

- import java.util.*
  or import java.util.Collection;

- There is a sister class, java.util.Collection**s**; that provides a number of algorithms for use with collections: sort, binarySearch, copy, shuffle, reverse, max, min, etc.


interface
class
implements
extends
Iterable
Collection
List
Queue
Set
PriorityQueue
HashSet
ArrayList
Deque
LinkedHashSet
LinkedList
Vector
ArrayDeque
SortedSet
Stack
TreeSet

Collection interface

- https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/Collection.html
- Don’t confuse with Collections class

List Interface

- A List is an ordered Collection. Lists may contain duplicate elements.
- In addition to the operations inherited from Collection, the List interface includes operations for the following:
  - Positional access — manipulates elements based on their numerical position in the list. This includes methods such as get, set, add, addAll, and remove.
  - Search — searches for a specified object in the list and returns its numerical position. Search methods include indexOf and lastIndexOf.
  - Iteration — extends Iterator semantics to take advantage of the list's sequential nature. The listIterator methods provide this behavior.
  - Range-view — The sublist method performs arbitrary range operations on the list.

List Interface

- This list interface is implemented by various classes like ArrayList, Vector, Stack, etc. Since all the subclasses implement the list, we can instantiate a list object with any of these classes. For example,

- List <T> al = new ArrayList<> ();
- List <T> ll = new LinkedList<> ();
- List <T> v = new Vector<> ();
- Where T is the type of the object

- Type Inference with the <> Notation
  - Java SE 7 introduced *type inferencing* with the <> notation—known as the diamond notation—in statements that declare and create generic type variables and objects. For example:

```
List<String> list = new ArrayList<>();
```

  - Java uses the type in angle brackets on the left of the declaration (that is, `String`) as the type stored in the `ArrayList` created on the right side of the declaration.

ArrayList

- ArrayList provides us with dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed.
- The size of an ArrayList is increased automatically if the collection grows or shrinks if the objects are removed from the collection.
- Java ArrayList allows us to randomly access the list.

- ArrayList can not be used for primitive types, like int, char, etc. We will need a wrapper class for such cases.

16.3 Type-Wrapper Classes

- Each primitive type has a corresponding type-wrapper class (in package `java.lang`).
  - Boolean, Byte, Character, Double, Float, Integer, Long and Short.
- Each type-wrapper class enables you to manipulate primitive-type values as objects.
- Collections cannot manipulate variables of primitive types.
  - They can manipulate objects of the type-wrapper classes, because every class ultimately derives from `Object`.

16.3 Type-Wrapper Classes (cont.)

- Each of the numeric type-wrapper classes—`Byte`, `Short`, `Integer`, `Long`, `Float` and `Double`—extends class `Number`.
- The type-wrapper classes are `final` classes, so you cannot extend them.
- Primitive types do not have methods, so the methods related to a primitive type are located in the corresponding type-wrapper class.

16.4 Autoboxing and Auto-Unboxing

- A boxing conversion converts a value of a primitive type to an object of the corresponding type-wrapper class.
- An unboxing conversion converts an object of a type-wrapper class to a value of the corresponding primitive type.
- These conversions are performed automatically—called autoboxing and auto-unboxing.
- Example:
  - ```
    // create integerArray
    Integer[] integerArray = new Integer[5];

    // assign Integer 10 to integerArray[ 0 ]
    integerArray[0] = 10;

    // get int value of Integer
    int value = integerArray[0];
    ```
    ```

16.5 Interface `Collection` and Class `Collections`

- Interface Collection contains bulk operations for *adding*, *clearing* and *comparing* objects in a collection.
- Interface `Collection` provides a method that returns an Iterator object, which allows a program to walk through the collection and remove elements from the collection during the iteration.
- A `Collection` can be converted to an array.
- Class Collections provides `static` methods that *search*, *sort* and perform other operations on collections.

Collection Interface Bulk Operations

- Bulk operations perform an operation on an entire Collection.

- `containsAll` — returns `true` if the target `Collection` contains all of the elements in the specified `Collection`.
- `addAll` — adds all of the elements in the specified `Collection` to the target `Collection`.
- `removeAll` — removes from the target `Collection` all of its elements that are also contained in the specified `Collection`.
- `retainAll` — removes from the target `Collection` all its elements that are *not* also contained in the specified `Collection`.
  those elements in the target `Collection` that are also contained in the specified `Collection`.
- `clear` — removes all elements from the `Collection`.

Software Engineering Observation 16.1

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

16.6 Lists

- In addition to the methods inherited from `Collection`, `List` provides methods for manipulating elements via their indices, manipulating a specified range of elements, searching for elements and obtaining a ListIterator to access the elements.
- Interface `List` is implemented by several classes, including ArrayList, LinkedList and Vector.
- Autoboxing occurs when you add primitive-type values to objects of these classes, because they store only references to objects.

16.6 Lists (cont.)

- Class `ArrayList` and `Vector` are resizable-array implementations of `List`.
- Inserting an element between existing elements of an `ArrayList` or `Vector` is an *inefficient* operation.
- A `LinkedList` enables *efficient* insertion (or removal) of elements in the middle of a collection, but is much less efficient than an `ArrayList` for jumping to a specific element in the collection.
- The primary difference between `ArrayList` and `Vector` is that operations on `Vector`s are *synchronized* by default, whereas those on `ArrayList`s are not.
- *Unsynchronized collections provide better performance than synchronized ones.*
- For this reason, `ArrayList` is typically preferred over `Vector` in programs that do not share a collection among threads.

16.6.1 ArrayList and Iterator

- List method add adds an item to the end of a list.
- List method size returns the number of elements.
- List method get retrieves an individual element's value from the specified index.
- Collection method iterator gets an Iterator for a Collection.
- Iterator- method hasNext determines whether there are more elements to iterate through.
  - Returns true if another element exists and false otherwise.
- Iterator method next obtains a reference to the next element.
- Collection method contains determine whether a Collection contains a specified element.
- Iterator method remove removes the current element from a Collection.

Iteration options

- There are three ways to traverse collections: (1) using aggregate operations (Using Streams and Lambdas)
- (2) with the for-each construct and
- (3) by using Iterators.

for–each

- The for–each construct allows you to concisely traverse a collection or array using a for loop. The following code uses the for–each construct to print out each element of a collection on a separate line.

```
for (Object o : collection)
    System.out.println(o);
```

Iterators

- You get an Iterator for a collection by calling its iterator method. The following is the Iterator interface.

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

Iterator

- The following method shows you how to use an Iterator to filter an arbitrary Collection — that is, traverse the collection removing specific elements.

```
void filter(Collection<?> c) {
    for (Iterator<?> it = c.iterator();
it.hasNext(); )
        if (!cond(it.next()))
            it.remove();
}
```

Iterator

- The hasNext method returns true if the iteration has more elements, and the next method returns the next element in the iteration.
- The remove method removes the last element that was returned by next from the underlying Collection.
- The remove method may be called only once per call to next and throws an exception if this rule is violated.

Iterator

- Note that Iterator.remove is the only safe way to modify a collection during iteration; the behavior is unspecified if the underlying collection is modified in any other way while the iteration is in progress.
- Use Iterator instead of the for-each construct when you need to:
  - Remove the current element. The for-each construct hides the iterator, so you cannot call remove. Therefore, the for-each construct is not usable for filtering.

Common Programming Error 16.1

If a collection is modified by one of its methods after an iterator is created for that collection, the iterator immediately becomes invalid—any operation performed with the iterator fails immediate and throws a `ConcurrentModificationException`*. For this reason, iterators are said to be "fail fast." Fail-fast iterators help ensure that a modifiable collection is not manipulated by two or more threads at the same time, which could corrupt the collection. In Chapter 23, Concurrency, you'll learn about concurrent collections (package* `java.util.concurrent`*) that can be safely manipulated by multiple concurrent threads.*

Collection Interface Array Operations

- The toArray methods are provided as a bridge between collections and older APIs that expect arrays on input.
- The array operations allow the contents of a Collection to be translated into an array.

Collection Interface Array Operations

- For example, suppose that c is a Collection. To dump the contents of c into a newly allocated array of Object
  - `Object[] a = c.toArray();`
- Suppose that c is known to contain only strings, dump the contents of c into a newly allocated array of String whose length is identical to the number of elements in c.

```
String[] a = c.toArray(new String[10]);
```

```java
// Fig. 16.2: CollectionTest.java
// Collection interface demonstrated via an ArrayList object.
import java.util.List;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

public class CollectionTest
{
   public static void main(String[] args)
   {
      // add elements in colors array to list
      String[] colors = {"MAGENTA", "RED", "WHITE", "BLUE", "CYAN"};
      List<String> list = new ArrayList<String>();

      for (String color : colors)
         list.add(color); // adds color to end of list

      // add elements in removeColors array to removeList
      String[] removeColors = {"RED", "WHITE", "BLUE"};
      List<String> removeList = new ArrayList<String>();

      for (String color : removeColors)
         removeList.add(color);
```

Fig. 16.2 | `Collection` interface demonstrated via an `ArrayList` object. (Part 1 of 3.)

```java

      // output list contents
      System.out.println("ArrayList: ");

      for (int count = 0; count < list.size(); count++)
         System.out.printf("%s ", list.get(count));

      // remove from list the colors contained in removeList
      removeColors(list, removeList);

      // output list contents
      System.out.printf("%n%nArrayList after calling removeColors:%n");

      for (String color : list)
         System.out.printf("%s ", color);
   }

```

Fig. 16.2 | `Collection` interface demonstrated via an `ArrayList` object. (Part 2 of 3.)

```
    // remove colors specified in collection2 from collection1
    private static void removeColors(Collection<String> collection1,
       Collection<String> collection2)
    {
       // get iterator
       Iterator<String> iterator = collection1.iterator();

       // loop while collection has items
       while (iterator.hasNext())
       {
          if (collection2.contains(iterator.next()))
             iterator.remove(); // remove current element
       }
    }
 } // end class CollectionTest
```

```
ArrayList:
MAGENTA RED WHITE BLUE CYAN

ArrayList after calling removeColors:
MAGENTA CYAN
```

Fig. 16.2 | `Collection` interface demonstrated via an `ArrayList` object. (Part 3 of 3.)

16.6.2 LinkedList

- List method addAll *appends all elements* of a collection to the end of a `List`.
- List method clear remove the elements of a `List`.
- List method size returns the number of items in the `List`.
- List method subList obtains a portion of a `List`.
  - This is a so-called range-view method, which enables the program to view a portion of the list.

16.6.2 `LinkedList` (cont.)

- List method listIterator gets A `List`’s bidirectional iterator.
- List-Iterator method set replaces the current element to which the iterator refers with the specified object.
- ListIterator method hasPrevious determines whether there are more elements while traversing the list backward.
- ListIterator method previous gets the previous element from the list.

```java
// Fig. 16.3: ListTest.java
// Lists, LinkedLists and ListIterators.
import java.util.List;
import java.util.LinkedList;
import java.util.ListIterator;

public class ListTest
{
   public static void main(String[] args)
   {
      // add colors elements to list1
      String[] colors =
         {"black", "yellow", "green", "blue", "violet", "silver"};
      List<String> list1 = new LinkedList<>();

      for (String color : colors)
         list1.add(color);

      // add colors2 elements to list2
      String[] colors2 =
         {"gold", "white", "brown", "blue", "gray", "silver"};
      List<String> list2 = new LinkedList<>();

```

Fig. 16.3 | Lists, LinkedLists and ListIterators. (Part 1 of 5.)

```java
      for (String color : colors2)
         list2.add(color);

      list1.addAll(list2); // concatenate lists
      list2 = null; // release resources
      printList(list1); // print list1 elements

      convertToUppercaseStrings(list1); // convert to uppercase string
      printList(list1); // print list1 elements

      System.out.printf("%nDeleting elements 4 to 6...");
      removeItems(list1, 4, 7); // remove items 4-6 from list
      printList(list1); // print list1 elements
      printReversedList(list1); // print list in reverse order
   }

```

Fig. 16.3 | Lists, LinkedLists and ListIterators. (Part 2 of 5.)

```java
   // output List contents
   private static void printList(List<String> list)
   {
      System.out.printf("%nlist:%n");

      for (String color : list)
         System.out.printf("%s ", color);

      System.out.println();
   }

   // locate String objects and convert to uppercase
   private static void convertToUppercaseStrings(List<String> list)
   {
      ListIterator<String> iterator = list.listIterator();

      while (iterator.hasNext())
      {
         String color = iterator.next(); // get item
         iterator.set(color.toUpperCase()); // convert to upper case
      }
   }

```

Fig. 16.3 | `List`s, `LinkedList`s and `ListIterator`s. (Part 3 of 5.)

```java
    // obtain sublist and use clear method to delete sublist items
    private static void removeItems(List<String> list,
       int start, int end)
    {
       list.subList(start, end).clear(); // remove items
    }

    // print reversed list
    private static void printReversedList(List<String> list)
    {
       ListIterator<String> iterator = list.listIterator(list.size());

       System.out.printf("%nReversed List:%n");

       // print list in reverse order
       while (iterator.hasPrevious())
          System.out.printf("%s ", iterator.previous());
    }
} // end class ListTest
```

Fig. 16.3 | Lists, LinkedLists and ListIterators. (Part 4 of 5.)

```
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
```

Fig. 16.3 | Lists, LinkedLists and ListIterators. (Part 5 of 5.)

16.6.2 `LinkedList` (cont.)

- Class `Arrays` provides `static` method asList to view an array as a List collection.
  - A `List` view allows you to manipulate the array as if it were a list.
  - This is useful for adding the elements in an array to a collection and for sorting array elements.
- Any modifications made through the `List` view change the array, and any modifications made to the array change the `List` view.
- The only operation permitted on the view returned by `asList` is `set`, which changes the value of the view and the backing array.
  - Any other attempts to change the view result in an UnsupportedOperationException.
- List method toArray gets an array from a `List` collection.

```java
// Fig. 16.4: UsingToArray.java
// Viewing arrays as Lists and converting Lists to arrays.
import java.util.LinkedList;
import java.util.Arrays;

public class UsingToArray
{
   // creates a LinkedList, adds elements and converts to array
   public static void main(String[] args)
   {
      String[] colors = {"black", "blue", "yellow"};
      LinkedList<String> links = new LinkedList<>(Arrays.asList(colors));

      links.addLast("red"); // add as last item
      links.add("pink"); // add to the end
      links.add(3, "green"); // add at 3rd index
      links.addFirst("cyan"); // add as first item

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

```

Fig. 16.4 | Viewing arrays as `Lists` and converting `Lists` to arrays. (Part 1 of 2.)

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

      for (String color : colors)
         System.out.println(color);
   }
} // end class UsingToArray
```

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

Fig. 16.4 | Viewing arrays as `Lists` and converting `Lists` to arrays. (Part 2 of 2.)

Common Programming Error 16.2

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.*

16.6.2 `LinkedList` (cont.)

- LinkedList method addLast adds an element to the end of a `List`.
- LinkedList method add also adds an element to the end of a `List`.
- LinkedList method addFirst adds an element to the beginning of a `List`.

16.7 Collections Methods

- Class `Collections` provides several high-performance algorithms for manipulating collection elements.
- The algorithms (Fig. 16.5) are implemented as `static` methods.

Method	Description
`sort`	Sorts the elements of a `List`.
`binarySearch`	Locates an object in a `List`, using the high-performance binary search algorithm which we introduced in Section 7.15 and discuss in detail in Section 19.4.
`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.

Fig. 16.5 | `Collections` methods.

16.7.1 Method `sort`

- Method sort sorts the elements of a `List`
  - The elements must implement the Comparable interface. **(How is this made sure? How do you specify the argument to a method should implement an interface?)**
  - The order is determined by the natural order of the elements' type as implemented by a `compareTo` method.
  - Method `compareTo` is declared in interface `Comparable` and is sometimes called the natural comparison method.
  - The `sort` call may specify as a second argument a Comparator object that determines an alternative ordering of the elements.

```java
// Fig. 16.6: Sort1.java
// Collections method sort.
import java.util.List;
import java.util.Arrays;
import java.util.Collections;

public class Sort1
{
   public static void main(String[] args)
   {
      String[] suits = {"Hearts", "Diamonds", "Clubs", "Spades"};

      // Create and display a list containing the suits array elements
      List<String> list = Arrays.asList(suits);
      System.out.printf("Unsorted array elements: %s%n", list);

      Collections.sort(list); // sort ArrayList
      System.out.printf("Sorted array elements: %s%n", list);
   }
} // end class Sort1
```

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

Fig. 16.6 | Collections method sort.

16.7.1 Method `sort` (cont.)

- The `Comparator` interface is used for sorting a `Collection`'s elements in a **different order**.
- The `static` Collections method reverseOrder returns a `Comparator` object that orders the collection's elements in reverse order.

```java
// Fig. 16.7: Sort2.java
// Using a Comparator object with method sort.
import java.util.List;
import java.util.Arrays;
import java.util.Collections;

public class Sort2
{
   public static void main(String[] args)
   {
      String[] suits = {"Hearts", "Diamonds", "Clubs", "Spades"};

      // Create and display a list containing the suits array elements
      List<String> list = Arrays.asList(suits); // create List
      System.out.printf("Unsorted array elements: %s%n", list);

      // sort in descending order using a comparator
      Collections.sort(list, Collections.reverseOrder());
      System.out.printf("Sorted list elements: %s%n", list);
   }
} // end class Sort2
```

Fig. 16.7 | Collections method sort with a Comparator object. (Part 1 of 2.)

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

Fig. 16.7 | `Collections` method `sort` with a `Comparator` object. (Part 2 of 2.)

16.7.1 Method `sort` (cont.)

- Figure 16.8 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. 16.8: TimeComparator.java
// Custom Comparator class that compares two Time2 objects.
import java.util.Comparator;

public class TimeComparator implements Comparator<Time2>
{
   @Override
   public int compare(Time2 time1, Time2 time2)
   {
      int hourDifference = time1.getHour() - time2.getHour();

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

      int minuteDifference = time1.getMinute() - time2.getMinute();

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

      int secondDifference = time1.getSecond() - time2.getSecond();
      return secondDifference;
   }
} // end class TimeComparator
```

Fig. 16.8 | Custom `Comparator` class that compares two `Time2` objects.

```java
// Fig. 16.9: 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<>(); // 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. 16.9 | `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 class Sort3
```

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

16.7.2 Method `shuffle`

- Method shuffle randomly orders a `List`'s elements.

```java
// Fig. 16.10: DeckOfCards.java
// Card shuffling and dealing with Collections method shuffle.
import java.util.List;
import java.util.Arrays;
import java.util.Collections;

// class to represent a Card in a deck of cards
class Card
{
   public static enum Face {Ace, Deuce, Three, Four, Five, Six,
      Seven, Eight, Nine, Ten, Jack, Queen, King };
   public static enum Suit {Clubs, Diamonds, Hearts, Spades};

   private final Face face;
   private final Suit suit;

   // constructor
   public Card(Face face, Suit suit)
   {
       this.face = face;
       this.suit = suit;
   }

```

Fig. 16.10 | Card shuffling and dealing with `Collections` method `shuffle`. (Part 1 of 5.)

```java
    // return face of the card
    public Face getFace()
    {
       return face;
    }

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

    // return String representation of Card
    public String toString()
    {
       return String.format("%s of %s", face, suit);
    }
 } // end class Card

```

Fig. 16.10 | Card shuffling and dealing with `Collections` method `shuffle`. (Part 2 of 5.)

```
// class DeckOfCards declaration
public class DeckOfCards
{
   private List<Card> list; // declare List that will store Cards

   // set up deck of Cards and shuffle
   public DeckOfCards()
   {
      Card[] deck = new Card[52];
      int count = 0; // number of cards

      // populate deck with Card objects
      for (Card.Suit suit: Card.Suit.values())
      {
         for (Card.Face face: Card.Face.values())
         {
            deck[count] = new Card(face, suit);
            ++count;
         }
      }

```

Fig. 16.10 | Card shuffling and dealing with `Collections` method `shuffle`. (Part 3 of 5.)

```
        list = Arrays.asList(deck); // get List
        Collections.shuffle(list);  // shuffle deck
    } // end DeckOfCards constructor

    // output deck
    public void printCards()
    {
        // display 52 cards in two columns
        for (int i = 0; i < list.size(); i++)
            System.out.printf("%-19s%s", list.get(i),
                ((i + 1) % 4 == 0) ? "%n" : "");
    }

    public static void main(String[] args)
    {
        DeckOfCards cards = new DeckOfCards();
        cards.printCards();
    }
} // end class DeckOfCards
```

Fig. 16.10 | Card shuffling and dealing with `Collections` method `shuffle`. (Part 4 of 5.)

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

Fig. 16.10 | Card shuffling and dealing with `Collections` method `shuffle`. (Part 5 of 5.)

16.7.3 Methods `reverse`, `fill`, `copy`, `max` and `min`

- Collections method reverse reverses the order of the elements in a `List`
- Method fill overwrites elements in a `List` with a specified value.
- Method copy takes two arguments—a destination `List` and a source `List`.
  - Each source `List` element is copied to the destination `List`.
  - The destination `List` must be at least as long as the source `List`; otherwise, an `IndexOutOfBoundsException` occurs.
  - If the destination `List` is longer, the elements not overwritten are unchanged.
- Methods min and max each operate on any `Collection`.
  - Method `min` returns the smallest element in a `Collection`, and method `max` returns the largest element in a `Collection`.

```java
// Fig. 16.11: Algorithms1.java
// Collections methods reverse, fill, copy, max and min.
import java.util.List;
import java.util.Arrays;
import java.util.Collections;

public class Algorithms1
{
   public static void main(String[] args)
   {
      // create and display a List<Character>
      Character[] letters = {'P', 'C', 'M'};
      List<Character> list = Arrays.asList(letters); // get List
      System.out.println("list contains: ");
      output(list);

      // reverse and display the List<Character>
      Collections.reverse(list); // reverse order the elements
      System.out.printf("%nAfter calling reverse, list contains:%n");
      output(list);

```

Fig. 16.11 | Collections methods reverse, fill, copy, max and min. (Part 1 of 3.)

```java
      // create copyList from an array of 3 Characters
      Character[] lettersCopy = new Character[3];
      List<Character> copyList = Arrays.asList(lettersCopy);

      // copy the contents of list into copyList
      Collections.copy(copyList, list);
      System.out.printf("%nAfter copying, copyList contains:%n");
      output(copyList);

      // fill list with Rs
      Collections.fill(list, 'R');
      System.out.printf("%nAfter calling fill, list contains:%n");
      output(list);
   }

   // output List information
   private static void output(List<Character> listRef)
   {
      System.out.print("The list is: ");

      for (Character element : listRef)
         System.out.printf("%s ", element);

```

Fig. 16.11 | `Collections` methods `reverse`, `fill`, `copy`, `max` and `min`. (Part 2 of 3.)

```java
         System.out.printf("%nMax: %s", Collections.max(listRef));
         System.out.printf("  Min: %s%n", Collections.min(listRef));
      }
   } // end class Algorithms1
```

```
list contains:
The list is: P C M
Max: P  Min: C

After calling reverse, list contains:
The list is: M C P
Max: P  Min: C

After copying, copyList contains:
The list is: M C P
Max: P  Min: C

After calling fill, list contains:
The list is: R R R
Max: R  Min: R
```

Fig. 16.11 | `Collections` methods `reverse`, `fill`, `copy`, `max` and `min`. (Part 3 of 3.)

16.7.4 Method `binarySearch`

- `static` Collections method binarySearch locates an object in a `List`.
  - If the object is found, its index is returned.
  - If the object is not found, `binarySearch` returns a negative value.

```java
// Fig. 16.12: BinarySearchTest.java
// Collections method binarySearch.
import java.util.List;
import java.util.Arrays;
import java.util.Collections;
import java.util.ArrayList;

public class BinarySearchTest
{
   public static void main(String[] args)
   {
      // create an ArrayList<String> from the contents of colors array
      String[] colors = {"red", "white", "blue", "black", "yellow",
         "purple", "tan", "pink"};
      List<String> list =
         new ArrayList<>(Arrays.asList(colors));

      Collections.sort(list); // sort the ArrayList
      System.out.printf("Sorted ArrayList: %s%n", list);

```

Fig. 16.12 | Collections method binarySearch. (Part 1 of 3.)

```java
      // search list for various values
      printSearchResults(list, "black"); // first item
      printSearchResults(list, "red"); // middle item
      printSearchResults(list, "pink"); // last item
      printSearchResults(list, "aqua"); // below lowest
      printSearchResults(list, "gray"); // does not exist
      printSearchResults(list, "teal"); // does not exist
   }

   // perform search and display result
   private static void printSearchResults(
      List<String> list, String key)
   {
      int result = 0;

      System.out.printf("%nSearching for: %s%n", key);
      result = Collections.binarySearch(list, key);

      if (result >= 0)
         System.out.printf("Found at index %d%n", result);
      else
         System.out.printf("Not Found (%d)%n",result);
   }
} // end class BinarySearchTest
```

Fig. 16.12 | Collections method binarySearch. (Part 2 of 3.)

```
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)
```

Fig. 16.12 | `Collections` method `binarySearch`. (Part 3 of 3.)

16.7.5 Methods `addAll`, `frequency` and `disjoint`

- Collections method addAll takes two arguments—a `Collection` into which to *insert* the new element(s) and an array that provides elements to be inserted.
- Collections method frequency takes two arguments—a `Collection` to be searched and an `Object` to be searched for in the collection.
  - Method `frequency` returns the number of times that the second argument appears in the collection.
- Collections method disjoint takes two `Collection`s and returns `true` if they have *no elements in common*.

```java
// Fig. Fig. 16.13: Algorithms2.java
// Collections methods addAll, frequency and disjoint.
import java.util.ArrayList;
import java.util.List;
import java.util.Arrays;
import java.util.Collections;

public class Algorithms2
{
   public static void main(String[] args)
   {
      // initialize list1 and list2
      String[] colors = {"red", "white", "yellow", "blue"};
      List<String> list1 = Arrays.asList(colors);
      ArrayList<String> list2 = new ArrayList<>();

      list2.add("black"); // add "black" to the end of list2
      list2.add("red"); // add "red" to the end of list2
      list2.add("green"); // add "green" to the end of list2

      System.out.print("Before addAll, list2 contains: ");

```

Fig. 16.13 | `Collections` methods `addAll`, `frequency` and `disjoint`. (Part 1 of 3.)

```java
        // display elements in list2
        for (String s : list2)
           System.out.printf("%s ", s);

        Collections.addAll(list2, colors); // add colors Strings to list2

        System.out.printf("%nAfter addAll, list2 contains: ");

        // display elements in list2
        for (String s : list2)
           System.out.printf("%s ", s);

        // get frequency of "red"
        int frequency = Collections.frequency(list2, "red");
        System.out.printf(
           "%nFrequency of red in list2: %d%n", frequency);

        // check whether list1 and list2 have elements in common
        boolean disjoint = Collections.disjoint(list1, list2);

        System.out.printf("list1 and list2 %s elements in common%n",
           (disjoint ? "do not have" : "have"));
     }
  } // end class Algorithms2
```

Fig. 16.13 | `Collections` methods `addAll`, `frequency` and `disjoint`. (Part 2 of 3.)

```
Before addAll, list2 contains: black red green
After addAll, list2 contains: black red green red white yellow blue
Frequency of red in list2: 2
list1 and list2 have elements in common
```

Fig. 16.13 | `Collections` methods `addAll`, `frequency` and `disjoint`. (Part 3 of 3.)

16.8 `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.
- 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.

```java
// Fig. 16.14: 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<>(); // 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. 16.14 | 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);
         }
      }
      catch (EmptyStackException emptyStackException)
      {
         emptyStackException.printStackTrace();
      }
   }

```

Fig. 16.14 | Stack class of package `java.util`. (Part 2 of 4.)

```java
    // display Stack contents
    private static void printStack(Stack<Number> stack)
    {
       if (stack.isEmpty())
          System.out.printf("stack is empty%n%n"); // the stack is empty
       else // stack is not empty
          System.out.printf("stack contains: %s (top)%n", stack);
    }
} // end class StackTest
```

Fig. 16.14 | 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. 16.14 | Stack class of package `java.util`. (Part 4 of 4.)

Error-Prevention Tip 16.1

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`*,*

16.9 Class `PriorityQueue` and Interface `Queue`

Documentation

- 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. 16.15: 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<>();

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

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

```

Fig. 16.15 | PriorityQueue test program. (Part 1 of 2.)

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

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

Fig. 16.15 | PriorityQueue test program. (Part 2 of 2.)

16.10 Sets

- A Set is an *unordered* `Collection` of unique elements (i.e., *no duplicates*).
- 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. 16.16: 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);
   }

```

Fig. 16.16 | 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<>(values);

       System.out.printf("%nNonduplicates are: ");

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

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

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

16.10 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. Documentation

```java
// Fig. 16.17: 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<>(Arrays.asList(colors));

      System.out.print("sorted set: ");
      printSet(tree);

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

```

Fig. 16.17 | Using SortedSets and TreeSets. (Part 1 of 3.)

```
      // 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());
   }

   // 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 class SortedSetTest
```

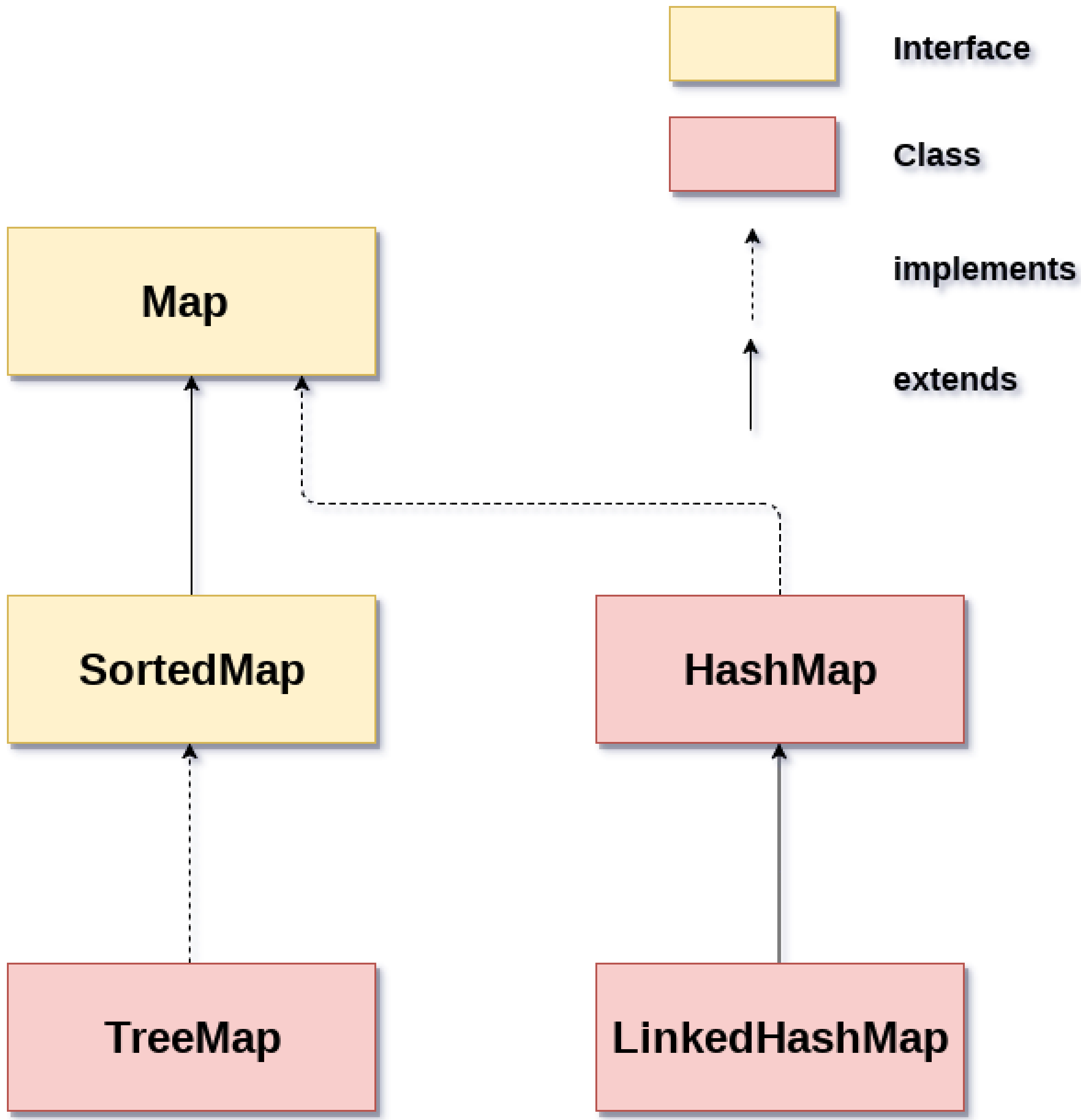
Fig. 16.17 | Using SortedSets and TreeSets. (Part 2 of 3.)

```
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. 16.17 | Using SortedSets and TreeSets. (Part 3 of 3.)

16.11 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.


Interface
Class
implements
extends
Map
SortedMap
HashMap
TreeMap
LinkedHashMap

- A Map doesn't allow duplicate keys, but you can have duplicate values. HashMap and LinkedHashMap allow null keys and values, but TreeMap doesn't allow any null key or value.
- A Map can't be traversed, so you need to convert it into Set using *keySet()* or *entrySet()* method.

Map – Applications

- Key-value association mapping such as dictionaries. Some examples are:
- A map of error codes and their descriptions.
- A map of zip codes and cities.
- A map of managers and employees. Each manager (key) is associated with a list of employees (value) he manages.
- A map of classes and students. Each class (key) is associated with a list of students (value).

16.11 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.
- One might ask, why not simply use a *List* and get rid of the keys all together? Especially since *HashMap* consumes more memory for saving keys and its entries are not ordered. The answer lies in the performance benefits for searching elements.
- ***HashMap* is very efficient at checking if a key exists or retrieving a value based on a key. Those operations take *O(1)* on average.**
- Adding and removing elements from a *HashMap* based on a key takes *O(1)* constant-time. Checking for an element without knowing the key takes linear time *O(n),* as it's necessary to loop over all the elements.

List Vs. Map

- A common antipattern we sometimes encounter is trying to maintain order using a map.
- When we iterate through the map values, we're not guaranteed to retrieve them in the same order we put them in.
- That is simply because a map wasn't designed for maintaining the order of elements.
- *Maps* are designed for quick access and search based on unique keys.
- When we want to maintain order or work with position-based indexes, lists are a natural choice.

Class	Description
HashMap	HashMap is the implementation of Map, but it doesn't maintain any order.
LinkedHashMap	LinkedHashMap is the implementation of Map. It inherits HashMap class. It maintains insertion order.
TreeMap	TreeMap is the implementation of Map and SortedMap. It maintains ascending order.

```java
// Fig. 16.18: 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<>();

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

```

Fig. 16.18 | Program counts the number of occurrences of each word in a `String`. (Part 1 of 5.)

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

```

Fig. 16.18 | Program counts the number of occurrences of each word in a `String`. (Part 2 of 5.)

```java
         // 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
         }
         else
            map.put(word, 1); // add new word with a count of 1 to map
      }
   }

   // 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<>(keys);

      System.out.printf("%nMap contains:%nKey\t\tValue%n");

```

Fig. 16.18 | Program counts the number of occurrences of each word in a `String`. (Part 3 of 5.)

```
        // 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 class WordTypeCount
```

Fig. 16.18 | Program counts the number of occurrences of each word in a `String`. (Part 4 of 5.)

```
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. 16.18 | Program counts the number of occurrences of each word in a `String`. (Part 5 of 5.)

16.11 Maps (Cont.)

- Map method containsKey determines whether a key is in a map.
- Map method put creates a new entry 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.