Collections – Part II
Lesson Overview

- Topics covered:

  (1) Collection interfaces:
  - List
  - Queue
  - Map

  (2) Ordering objects – compareTo and Comparable

  (3) Ordered Collection interfaces:
  - OrderedSet
  - OrderedMap
The List Interface

- A List — an ordered Collection (sometimes called a sequence).
- 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
  - **Search** - searches for a specified object in the list and returns its numerical position
  - **Iteration** - extends Iterator semantics to take advantage of the list's sequential nature
  - **Range-view** - performs arbitrary range operations on the list.
The List Interface

```java
public interface List<E> extends Collection<E> {

    // Positional access

    E get(int index);
    E set(int index, E element); // optional
    boolean add(E element); // optional
    void add(int index, E element); // optional
    E remove(int index); // optional
    abstract boolean addAll(int index, Collection<? extends E> c); // optional
```
The List Interface (cont.)

//Search
int indexOf(Object o);
int lastIndexOf(Object o);

//Iteration
ListIterator<E> listIterator();
ListIterator<E> listIterator(int index);

//Range-view
List<E> subList(int from, int to);
List Implementations

The Java platform contains two general-purpose List implementations:

1. **ArrayList** - which is usually the better-performing implementation

2. **LinkedList** - which offers better performance under certain circumstances.

- The Vector has also been retrofitted to implement List.
List Implementations - Collection operations

- The `remove` operation always removes the first occurrence of the specified element from the list.

- The `add` and `addAll` operations always append the new element(s) to the end of the list. Thus, the following idiom concatenates one list to another:

  ```java
  list1.addAll(list2);
  ```

- Here's a nondestructive form of this snippet, which produces a third List consisting of the second list appended to the first:

  ```java
  List<Type> list3 = new ArrayList<Type>(list1);
  list3.addAll(list2);
  ```
List Implementations - Collection operations

- The basic positional access operations (get, set, add and remove) behave just like their longer-named counterparts in Vector (elementAt, setElementAt, insertElementAt, and removeElementAt)

- The search operations indexOf and lastIndexOf behave exactly like the identically named operations in Vector.

- The addAll operation inserts all the elements of the specified Collection starting at the specified position. The elements are inserted in the order they are returned by the specified Collection's iterator.
Here's a little method to swap two indexed values in a List:

```java
public static <E> void swap(List<E> a, int i, int j) {
    E tmp = a.get(i);
    a.set(i, a.get(j));
    a.set(j, tmp);
}
```

This is a polymorphic algorithm: It swaps two elements in any List, regardless of its implementation type.
**List Algorithm example: Shuffling**

```
public static void shuffle(List<?> list, Random rnd) {  
    for (int i = list.size(); i > 1; i--)  
        swap(list, i - 1, rnd.nextInt(i));  
}
```

- This algorithm, is included in the Java platform's Collections API.
- It randomly permutes the specified list using the specified source of randomness.
- It runs up the list from the bottom, repeatedly swapping a randomly selected element into the current position. Unlike most naive attempts at shuffling, it's fair and fast (requiring exactly list.size()-1 swaps).
Code Example: Using the Shuffle Algorithm

```java
import java.util.*;

public class Shuffle {

    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        for (String a : args)
            list.add(a);
        Collections.shuffle(list, new Random());
        System.out.println(list);
    }
}
```
Code Example: Using the Shuffle Algorithm – A short solution

```java
import java.util.*;

public class Shuffle {
    public static void main(String args[]) {
        List<String> list = Arrays.asList(args);
        Collections.shuffle(list);
        System.out.println(list);
    }
}
```

The `Arrays` class has a static factory method called `asList`, which allows an array to be viewed as a `List`. 
The Arrays asList method

- The asList method does not copy the array.
- Changes in the List write through to the array and vice-versa.
- The resulting List is not a general-purpose List implementation, in that it doesn't implement the (optional) add and remove operations: Arrays are not resizable.
List Iterators

- The Iterator returned by List's iterator operation returns the elements of the list in proper sequence.
- List also provides a richer iterator, called a ListIterator, which allows you to traverse the list in either direction, modify the list during iteration, and obtain the current position of the iterator.
- Here is an example of how you traverse a list backwards:

```java
for (ListIterator<Type> i = list.listIterator(list.size()); i.hasPrevious(); ) {
    Type t = i.previous();
    ...
}
```
List Iterators

Here is a polymorphic algorithm to replace all occurrences of a specified value with the sequence of values contained in the specified list:

```java
public static <E> void replace(List<E> s, E val, List<E> newVals) {
    for (ListIterator<E> i = s.listIterator(); i.hasNext(); ) {
        if (val == null ? i.next() == null : val.equals(i.next())) {
            i.remove();
            for (E e : newVals) i.add(e);
        }
    }
}
```
List Range View Operations

- The range-view operation, `subList(int fromIndex, int toIndex)`, returns a List view of the portion of this list whose indices range from `fromIndex`, inclusive, to `toIndex`, exclusive.

- This half-open range mirrors the typical for loop:
  ```java
  for (int i = fromIndex; i < toIndex; i++) {
    ...
  }
  ```

- As the term `view` implies, the returned List is backed up by the List on which `subList` was called, so changes in the former are reflected in the latter.
List Range View Operations

- This method eliminates the need for explicit range operations (of the sort that commonly exist for arrays).

- Any operation that expects a List can be used as a range operation by passing a subList view instead of a whole List.

- For example, the following idiom removes a range of elements from a List:

  ```java
  list.subList(fromIndex, toIndex).clear();
  ```

- Similarly we can search for an element in a range:

  ```java
  int i = list.subList(fromIndex, toIndex).indexOf(o);
  int j = list.subList(fromIndex, toIndex).lastIndexOf(o);
  ```
List Algorithms

- Most polymorphic algorithms in the Collections class apply specifically to List. Having all these algorithms at your disposal makes it very easy to manipulate lists.

- A summary of List algorithms:
  - sort - sorts a List using a merge sort algorithm, which provides a fast, stable sort. (A stable sort is one that does not reorder equal elements.)
  - shuffle - randomly permutes the elements in a List.
  - reverse - reverses the order of the elements in a List.
List Algorithms

- summary of List algorithms (cont.):
  - rotate - rotates all the elements in a List by a specified distance.
  - swap - swaps the elements at specified positions in a List.
  - replaceAll - replaces all occurrences of one specified value with another.
  - fill - overwrites every element in a List with the specified value.
  - copy - copies the source List into the destination List.
List Algorithms

- summary of List algorithms (cont.):
  
  - binarySearch - searches for an element in an ordered List using the binary search algorithm.
  
  - indexOfSubList - returns the index of the first sublist of one List that is equal to another.
  
  - lastIndexOfSubList - returns the index of the last sublist of one List that is equal to another.
The Queue interface

- A Queue is a collection for holding elements prior to processing.
- Besides basic Collection operations, queues provide additional insertion, removal, and inspection operations.
- The Queue interface:

```java
public interface Queue<E> extends Collection<E> {
    E element();
    boolean offer(E o);
    E peek();
    E poll();
    E remove();
}
```
The Queue interface

- Each Queue method exists in two forms:
  
  (1) One throws an exception if the operation fails,
  
  (2) The other returns a special value (either null or false, depending on the operation).

<table>
<thead>
<tr>
<th>Method</th>
<th>Throws</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>add(e)</td>
<td>offer(e)</td>
</tr>
<tr>
<td>Remove</td>
<td>remove()</td>
<td>poll()</td>
</tr>
<tr>
<td>Examine</td>
<td>element()</td>
<td>peek()</td>
</tr>
</tbody>
</table>
The Queue interface

- Queue implementations generally do not allow insertion of null elements (The LinkedList implementation, which was retrofitted to implement Queue, is an exception).

- Queue implementations generally do not define element-based versions of the equals and hashCode methods but instead inherit the identity-based versions from Object.
Heap Sort using a Priority Queue

- The following example sorts a collection using a priority queue:

```java
static <E> List<E> heapSort(Collection<E> c) {
    Queue<E> queue = new PriorityQueue<E>(c);
    List<E> result = new ArrayList<E>();
    while (!queue.isEmpty())
        result.add(queue.remove());
    return result;
}
```
The Map interface

- A Map is an object that maps keys to values.
- A map cannot contain duplicate keys: Each key can map to at most one value.
- The Java platform contains three general-purpose Map implementations: HashMap, TreeMap, and LinkedHashMap. Their behavior and performance are precisely analogous to HashSet, TreeSet, and LinkedHashSet.
The Map interface

public interface Map {
    // Basic operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();
}
The Map interface

// interface Map continued...

// Bulk operations
void putAll(Map<? extends K, ? extends V> t);
void clear();

// Collection Views
public Set<K> keySet();
public Collection<V> values();
public Set<Map.Entry<K, V>> entrySet();
The Map interface

//interface Map continued...

// Interface for entrySet elements
public interface Entry {
    K getKey();
    V getValue();
    V setValue(V value);
}
}
Maps vs. Hash-Tables

- A Map is an interface while a HashTable is a concrete implementation.

- The following are the major differences:

  (1) Map provides Collection views instead of direct support for iteration via Enumeration objects. Collection views greatly enhance the expressiveness of the interface.

  (2) Map allows you to iterate over keys, values, or key-value pairs; Hashtable does not provide the third option.

  (3) Map provides a safe way to remove entries in the midst of iteration; Hashtable did not.
Map Constructors

- All Map implementations provide constructors that take a Map object and initialize the new Map to contain all the key-value mappings in the specified Map. (Convention)

- This standard Map conversion constructor is entirely analogous to the standard Collection constructor: It allows the caller to create a Map of a desired implementation type that initially contains all of the mappings in another Map, regardless of the other Map's implementation type.

- For example, suppose you have a Map, named m. The following one-liner creates a new HashMap initially containing all of the same key-value mappings as m:

  ```java
  Map<K, V> copy = new HashMap<K, V>(m);
  ```
Map Bulk-Operations

- The clear operation removes all the mappings from the Map.
- The putAll operation is the Map analogue of the Collection interface's addAll operation.
- In addition to its obvious use of dumping one Map into another, it has a second, more subtle use:
  - The putAll operation, (+ the conversion const.) provides a neat way to implement attribute map creation with default values from an existing Map.
Map Bulk-Operations

- The following is a static factory method that demonstrates this technique (m is a Map):

```java
static <K, V> Map<K, V> newAttributeMap(
    Map<K, V> defaults, Map<K, V> overrides) {
    Map<K, V> result = new HashMap<K, V>(defaults);
    result.putAll(overrides);
    return result;
}
```
Collection views

The Collection view methods allow a Map to be viewed as a Collection in these three ways:

1. keySet - the Set of keys contained in the Map.
2. values - The Collection of values contained in the Map. This Collection is not a Set, because multiple keys can map to the same value.
3. entrySet - the Set of key-value pairs contained in the Map.
   - The Map interface provides a small nested interface called Map.Entry, the type of the elements in this Set.
Collection views

- The Collection views provide the only means to iterate over a Map.

- The standard idiom for iterating over the keys in a Map with a for-each construct:

```java
for (KeyType key : m.keySet())
    System.out.println(key);
```

- and with an iterator:

```java
// Filter a map based on some property of its keys.
for (Iterator<Type> i = m.keySet().iterator();
     i.hasNext(); )
    if (i.next().isBogus())
        i.remove();
```
Collection views

- The idiom for iterating over values is analogous.

- Following is the idiom for iterating over key-value pairs:

```java
for (Map.Entry<KeyType, ValType> e : m.entrySet())
    System.out.println(e.getKey() + " : " + e.getValue());
```
Collection views – and efficiency

- These idioms may be slow because the `Map` has to create a new `Collection` instance each time a Collection view operation is called. This is solved by making sure that a `Map` always returns the same object each time it is asked for a given Collection view.

- This is precisely what all the Map implementations in `java.util` do.

- With all three Collection views, calling an Iterator's `remove` operation removes the associated entry from the backing `Map`, assuming that the backing Map supports element removal to begin with.
Collection views - and efficiency

- With the entrySet view, it is also possible to change the value associated with a key by calling a Map.Entry's setValue method during iteration (again, assuming the Map supports value modification to begin with).

- Note that these are the only safe ways to modify a Map during iteration; the behavior is unspecified if the underlying Map is modified in any other way while the iteration is in progress.

- The Collection views support element removal in all its many forms - remove, removeAll, retainAll, and clear operations, as well as the Iterator.remove operation.
Collection views – and efficiency

- The Collection views do not support element addition under any circumstances.
- It would make no sense for the keySet and values views, and it's unnecessary for the entrySet view, because the backing Map's put and putAll methods provide the same functionality.
Multi-Map example

- A **multimap** is like a Map but it can map each key to multiple values.

- The Java Collections Framework doesn't include an interface for multimaps because they aren't used all that commonly.

- It's a fairly simple matter to use a Map whose values are List instances as a multimap.

- This technique is demonstrated in the next code example, which reads a wordlist containing one word per line (all lowercase) and prints out all the **anagram groups** that meet a size criterion.

- An **anagram group** - is a bunch of words, all of which contain exactly the same letters but in a different order.
import java.util.*;
import java.io.*;

public class Anagrams {

    public static void main(String[] args) {
        int minGroupSize = Integer.parseInt(args[1]);

        // Further code here...
    }
}
Multi-Map example

//Read words from file and put into a multimap

Map<String, List<String>> m = new HashMap<String, List<String>>() {
    List<String> l;
};

try {
    Scanner s = new Scanner(new File(args[0]));
    String word;
    while(s.hasNext()) {
        String alpha = alphabetize(word = s.next());
        List<String> l = m.get(alpha);
        if (l==null)
            m.put(alpha, l=new ArrayList<String>());
        l.add(word);
    }
}
Multi-Map example

//after try...

} catch (IOException e) {
    System.err.println(e);
    System.exit(1);
}

//Print all permutation groups above size threshold
for (List<String> l : m.values()) {
    if (l.size() >= minGroupSize)
        System.out.println(l.size() + " : " + l);
}
} //main
private static String alphabetize(String s) {
    int count[] = new int[256];
    int len = s.length();
    for (int i=0; i<len; i++)
        count[s.charAt(i)]++;
    StringBuffer result = new StringBuffer(len);
    for (char c='a'; c<='z'; c++)
        for (int i=0; i<count[c]; i++)
            result.append(c);
    return result.toString();
}
Multi-Map example

- Running this program on a 173,000-word dictionary file takes about four seconds on a 1.8 GHz Pentium 4.

- With a minimum anagram group size of eight, it produces the following (partial) output:

  9: [estrin, inerts, insert, inters, niters, nitres, sinter, triens, trines]

  8: [lapse, leaps, pales, peals, pleas, salep, sepal, spale]

  8: [aspers, parses, passer, prases, repass, spares, sparse, spears]

  10: [least, setal, slate, stale, steal, stela, taels, tales, teals, tesla]

  8: [enters, nester, renest, rentes, resent, tenser, ternes, treens]

  8: [arles, earls, lares, laser, lears, rales, reals, seral]

  8: [earings, erasing, gainers, reagins, regains, reginas, searing, seringa]
Ordering Objects

- We will now implement a simple Name class, which will have several important ingredients that will allow it to fit nicely into any type of collection.
- Specifically - we will define a natural-ordering on this class, to allow using it in ordered collections:
import java.util.*;

public final class Name implements Comparable<Name> {
    private final String firstName, lastName;

    public Name(String firstName, String lastName) {
        if (firstName == null || lastName == null)
            throw new NullPointerException();
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public int compareTo(Name anotherName) {
        String thisName = this.firstName + this.lastName;
        String anotherNameName = anotherName.firstName + anotherName.lastName;
        return thisName.compareTo(anotherNameName);
    }

    @Override
    public String toString() {
        return firstName + lastName;
    }
}

Slide 46
Ordering Objects

//Name class cont.

    public String firstName() { return firstName; }

    public String lastName() { return lastName; }

    public boolean equals(Object o) {
        if (!(o instanceof Name))
            return false;

        Name n = (Name) o;

        return n.firstName.equals(firstName) &&
               n.lastName.equals(lastName);
    }

Ordering Objects

//Name class cont.

    public int hashCode() {
        return 31 * firstName.hashCode() + 
            lastName.hashCode();
    }

    public String toString() {
        return firstName + " " + lastName;
    }

    public int compareTo(Name n) {
        int lastCmp = lastName.compareTo(n.lastName);
        return (lastCmp != 0 ? lastCmp :
                  firstName.compareTo(n.firstName));
    }

} }//of class
Writing classes for use with Collections

- Name class example: important implementation issues:
  - Name objects are **immutable**. All other things being equal, immutable types are the way to go, especially for objects that will be used as elements in Sets or as keys in Maps. These collections will break if you modify their elements or keys while they're in the collection.
  - The constructor checks its arguments for null. This ensures that all Name objects are well formed so that none of the other methods will ever throw a NullPointerException.
  - The hashCode method is redefined. This is essential for any class that redefines the equals method. (Equal objects must have equal hash codes.)
Writing classes for use with Collections - Ordering

- Name class example: important implementation issues:
  (cont.)
  - The \texttt{equals} method returns false if the specified object is null or of an inappropriate type. The \texttt{compareTo} method throws a runtime exception under these circumstances. Both of these behaviors are required by the general contracts of the respective methods.
  
  - The \texttt{toString} method has been redefined so it prints the \texttt{Name} in human-readable form. This is always a good idea, especially for objects that are going to get put into collections. The various collection types' \texttt{toString} methods depend on the \texttt{toString} methods of their elements, keys, and values.
Writing classes for use with Collections - Ordering

- Name classes example: `compareTo` implementation
  - implementation provided is quite typical: First, you compare the most significant part of the object (in this case, the last name).
  - Often, you can just use the natural ordering of the part's type. In this case, the part is a String and the natural (lexicographic) ordering is exactly what's called for.
  - If the comparison results in anything other than zero, which represents equality, you're done: You just return the result.
  - If the most significant parts are equal, you go on to compare the next most-significant parts. In this case, there are only two parts - first name and last name.
Writing classes for use with Collections - Ordering

- Name classes example: `compareTo` implementation (cont.)
  - If there were more parts, you'd proceed in the obvious fashion, comparing parts until you found two that weren't equal or you were comparing the least-significant parts, at which point you'd return the result of the comparison.
  - There are four restrictions on the behavior of the `compareTo` method, which must be obeyed. Attempting to sort a list of objects that violate the restrictions has undefined behavior. Technically speaking, these restrictions ensure that the natural ordering is a **total order** on the objects of a class that implements it; this is necessary to ensure that sorting is well defined.
The Comparator Interface

- What if you want to sort some objects in an order other than their natural order? Or what if you want to sort some objects that don't implement Comparable?

- To do either of these things, you'll need to provide a Comparator - an object that encapsulates an ordering.

- Like the Comparable interface, the Comparator interface consists of a single method:

```java
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```
The Comparator Interface

- The `compare` method compares its two arguments, returning a negative integer, 0, or a positive integer depending on whether the first argument is less than, equal to, or greater than the second. If either of the arguments has an inappropriate type for the Comparator, the compare method throws a `ClassCastException`.

- Much of what was said about `Comparable` applies to `Comparator` as well - Writing a `compare` method is nearly identical to writing a `compareTo` method, except that the former gets both objects passed in as arguments.

- The `compare` method has to obey the same four technical restrictions as `Comparable`'s `compareTo` method.
Comparator example – Version I

- Let's assume that Employee's natural ordering is by their names. Here is a comparator that compares employees by their hire dates:

```java
static final Comparator<Employee> SENIORITY_ORDER =
    new Comparator<Employee>() {
        public int compare(Employee e1, Employee e2) {
            return e2.hireDate().compareTo(e1.hireDate());
        }
    };
```

- This Comparator works fine for sorting a List, but it cannot be used to order a sorted collection, such as TreeSet, because it generates an ordering that is not compatible with equals.
Comparator example

- This means that this Comparator equates objects that the equals method does not. In particular, any two employees who were hired on the same date will compare as equal.
- When you're sorting a List, this doesn't matter; but when you're using the Comparator to order a sorted collection, it's fatal.
- If you use this Comparator to insert multiple employees hired on the same date into a TreeSet, only the first one will be added to the set; the second will be seen as a duplicate element and will be ignored.
- A possible fix is to define a Comparator that produces an ordering that is compatible with equals.
Comparator example – Version II

static final Comparator<Employee> SENIORITY_ORDER =
    new Comparator<Employee>() {

    public int compare(Employee e1, Employee e2) {

        int dateCmp =
            e2.hireDate().compareTo(e1.hireDate());

        if (dateCmp != 0)
            return dateCmp;

        return (e1.number() < e2.number() ? -1 :
                (e1.number() == e2.number() ? 0 : 1));
    }
};
**Comparator example**

- One last note: You might be tempted to replace the final return statement in the Comparator with the simpler:

  ```java
  return r1.empNumber() - r2.empNumber();
  ```

- Don't do it unless you're absolutely sure no one will ever have a negative employee number!

- This trick does not work in general because the signed integer type is not big enough to represent the difference of two arbitrary signed integers. If i is a large positive integer and j is a large negative integer, i - j will overflow and will return a negative integer!
Comparator example

- The resulting comparator violates one of the four technical restrictions we keep talking about (transitivity) and produces horrible, subtle bugs.

- This is not a purely theoretical concern; people get burned by it!
The SortedSet Interface

- A SortedSet is a Set that maintains its elements in ascending order, sorted according to the elements' natural order or according to a Comparator provided at SortedSet creation time.

- In addition to the normal Set operations, the SortedSet interface provides operations for the following:

  - **Range view** - allows arbitrary range operations on the sorted set.
  - **Endpoints** - returns the first or last element in the sorted set.
  - **Comparator access** - returns the Comparator, if any, used to sort the set.
The SortedSet Interface

```java
public interface SortedSet<E> extends Set<E> {
    // Range-view
    SortedSet<E> subSet(E fromElement, E toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);
    // Endpoint
    E first();
    E last();
    // Comparator access
    Comparator<? super E> comparator();
}
```
The SortedSet Interface

- The operations that SortedSet inherits from Set behave identically on sorted sets and normal sets with two exceptions:

  1. The Iterator returned by the iterator operation traverses the sorted set in order.
  2. The array returned by toArray contains the sorted set's elements in order.

- Although the interface doesn't guarantee it, the toString method of the Java platform's SortedSet implementations returns a string containing all the elements of the sorted set, in order.
**RangeView operations**

- The range-view operations are somewhat analogous to those provided by the List interface, but there is one big difference: Range views of a sorted set remain valid even if the backing sorted set is modified directly.

- This is feasible because the endpoints of a range view of a sorted set are absolute points in the element space rather than specific elements in the backing collection, as is the case for lists.

- A range-view of a sorted set is really just a window onto whatever portion of the set lies in the designated part of the element space. Changes to the range-view write back to the backing sorted set and vice versa. Thus, it's okay to use range-views on sorted sets for long periods of time, unlike range-views on lists.
RangeView operations

- Sorted sets provide three range-view operations:

  1. `subSet` - takes two endpoints, like `subList`. Rather than indices, the endpoints are objects and must be comparable to the elements in the sorted set, using the Set's Comparator or the natural ordering of its elements, whichever the Set uses to order itself. Like `subList`, the range is half open, including its low endpoint but excluding the high one.

- The following one line of code tells you how many words between "doorbell" and "pickle", including doorbell but excluding pickle, are contained in a `SortedSet` of strings called `dictionary`:

```java
int count =
dictionary.subSet("doorbell", "pickle").size();
```
RangeView operations

- In like manner, the following one-liner removes all the elements beginning with the letter f:

```
dictionary.subSet("f", "g").clear();
```

- A similar trick can be used to print a table telling you how many words begin with each letter:

```
for (char ch = 'a'; ch <= 'z'; ) {
    String from = String.valueOf(ch++);
    String to = String.valueOf(ch);
    System.out.println(from + " : " +
                       dictionary.subSet(from, to).size());
}
```
RangeView operations

- Sorted sets provide three range-view operations:
  
  (2-3) headSet and tailSet – both of which take a single Object argument. The former returns a view of the initial portion of the backing SortedSet, up to but not including the specified object. The latter returns a view of the final portion of the backing SortedSet, beginning with the specified object and continuing to the end of the backing SortedSet.

Thus, the following code allows you to view the dictionary as two disjoint volumes (a-m and n-z).

```java
SortedSet<String> volume1=dictionary.headSet("n");
SortedSet<String> volume2=dictionary.tailSet("n");
```
Endpoint operations

- The `SortedSet` interface contains operations to return the first and last elements in the sorted set, not surprisingly called `first` and `last`.

- In addition to their obvious uses, last allows a workaround for a deficiency in the `SortedSet` interface. One thing you'd like to do with a `SortedSet` is to go into the interior of the Set and iterate forward or backward. It's easy enough to go forward from the interior: Just get a `tailSet` and iterate over it. Unfortunately, there's no easy way to go backward.

- The following idiom obtains the first element that is less than a specified object `o` in the element space:

  ```java
  Object predecessor = s.headSet(o).last();
  ```
Endpoint operations

Object predecessor = ss.headSet(o).last();

- This is a fine way to go one element backward from a point in the interior of a sorted set. It could be applied repeatedly to iterate backward, but this is very inefficient, requiring a lookup for each element returned.
Collection Summary

- The core collection interfaces are the foundation of the Java Collections Framework.
- The Java Collections Framework hierarchy consists of two distinct interface trees:
  - The first tree starts with the Collection interface, which provides for the basic functionality used by all collections, such as add and remove methods. Its subinterfaces - Set, List, and Queue, provide for more specialized collections:
    - The Set interface does not allow duplicate elements. This can be useful for storing collections such as a deck of cards or student records. The Set interface has a subinterface, SortedSet, that provides for ordering of elements in the set.
Collection Summary

- The core collection interfaces are the foundation of the Java Collections Framework.
- The first tree starts with the Collection interface, which provides for the basic functionality used by all collections, such as add and remove methods.
  - The List interface provides for an ordered collection, for situations in which you need precise control over where each element is inserted. You can retrieve elements from a List by their exact position.
  - The Queue interface enables additional insertion, extraction, and inspection operations. Elements in a Queue are typically ordered in a FIFO basis.
Collection Summary

- The core collection interfaces are the foundation of the Java Collections Framework.

- The second tree starts with the Map interface, which maps keys and values similarly to a Hashtable.

- Map's subinterface, SortedMap, maintains its key-value pairs in ascending order or in an order specified by a Comparator.

- All of These interfaces allow collections to be manipulated independently of the details of their representation.