Sets and Maps

Paul Fodor

CSE260, Computer Science B: Honors

Stony Brook University

http://www.cs.stonybrook.edu/~cse260

Objectives

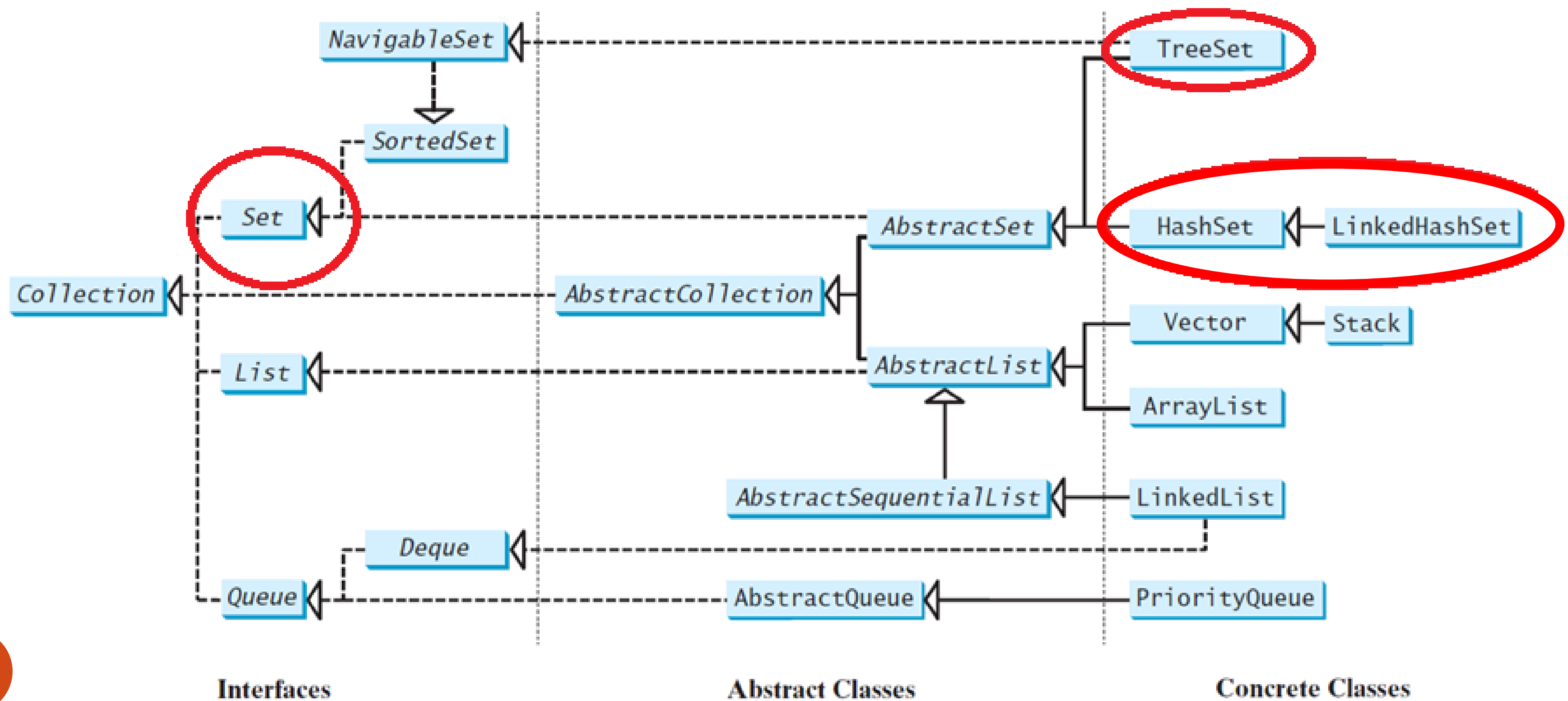
- To store unordered, nonduplicate elements using **sets**
- To explore how and when to use **`HashSet`**, **`LinkedHashSet`** or **`TreeSet`** to store elements
- To compare performance of sets and lists
- To use sets to develop a program that counts the distinct keywords in a Java source file
- To tell the differences between **`Collection`** and **`Map`** and describe when and how to use **`HashMap`**, **`LinkedHashMap`**, and **`TreeMap`** to store values associated with keys
- To use maps to develop a program that counts the occurrence of the words in a text

Motivation

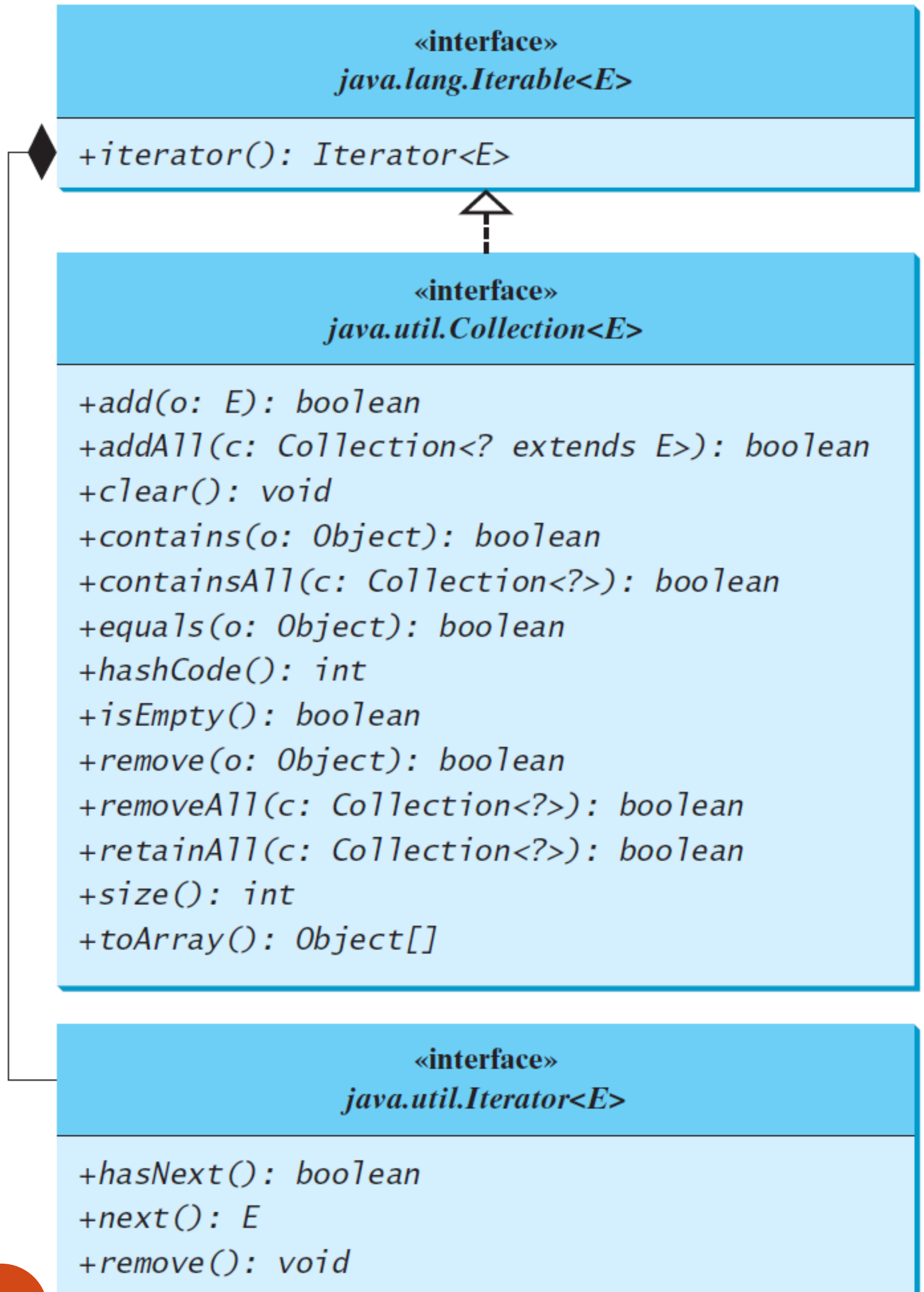
- Suppose we need to write a program that checks whether a student is **<u>in</u>** a class
  - You can use a list to store the names of the students and search for the student with linear search
    - Or sort the list of students and search with binary search
  - However, a more efficient data structure for this application is a ***set*** with efficient methods to search for elements
- Moreover, suppose your program also needs to <u>store detailed information</u> about the students in the class (e.g., grades for labs, homework submissions, submission times, GPA) and <u>all</u> can be retrieved using the name of the student as the ***key***
    - A ***map*** is an efficient data structure for such a task

Review of Java Collection Framework hierarchy

- **`Set`** is a sub-interface of **`Collection`**
- You can create a set using one of its three concrete classes: **`HashSet`**, **`LinkedHashSet`**, or **`TreeSet`**



Reminder Collection



Returns an iterator for the elements in this collection.

- The Collection interface is the root interface for manipulating a collection of objects.

Adds a new element `o` to this collection.
Adds all the elements in the collection `c` to this collection.
Removes all the elements from this collection.
Returns true if this collection contains the element `o`.
Returns true if this collection contains all the elements in `c`.
Returns true if this collection is equal to another collection `o`.
Returns the hash code for this collection.
Returns true if this collection contains no elements.
Removes the element `o` from this collection.
Removes all the elements in `c` from this collection.
Retains the elements that are both in `c` and in this collection.
Returns the number of elements in this collection.
Returns an array of `Object` for the elements in this collection.

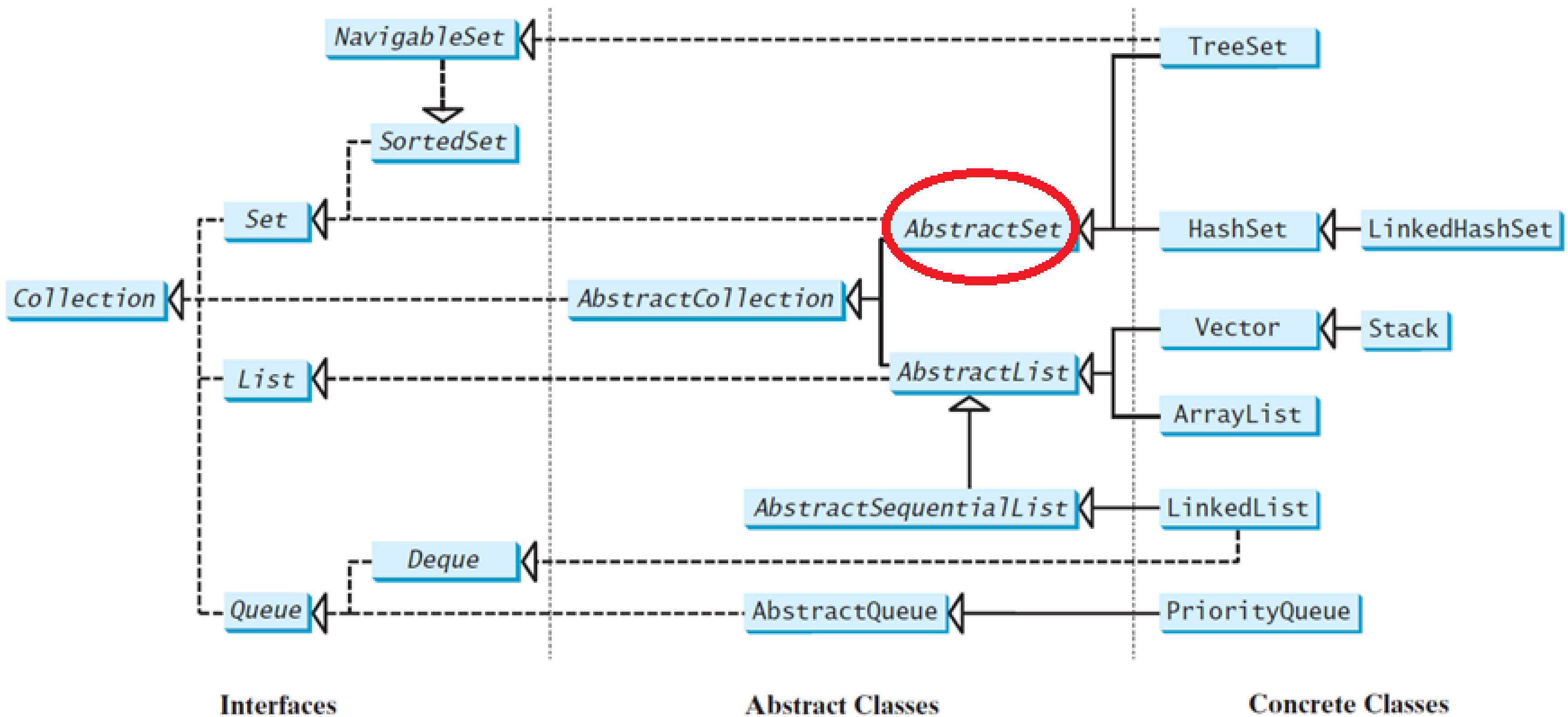
Returns true if this iterator has more elements to traverse.
Returns the next element from this iterator.
Removes the last element obtained using the next method.

The **`Set`** Interface

- The **`Set`** interface extends the **`Collection`** interface, but it does not introduce new methods or constants, but it **stipulates** that an instance of **`Set`** **contains no duplicate elements**
  - That is, **no two elements** **`e1`** and **`e2`** can be in the set such that **`e1.equals(e2)`** is true
    - The concrete classes that implement **`Set`** must ensure that **no duplicate elements can be added** to the set

AbstractSet

- The **AbstractSet** class extends **AbstractCollection** and partially implements **Set**



AbstractSet

- The **`AbstractSet`** class provides concrete implementations for the **`equals`** method and the **`hashCode`** method
  - **<u>The hash code of a set is the sum of the hash codes of all the elements in the set</u>**
  - Since the **`size`** method and **`iterator`** method are not implemented in the **`AbstractSet`** class, **`AbstractSet`** is an abstract class

Hash codes

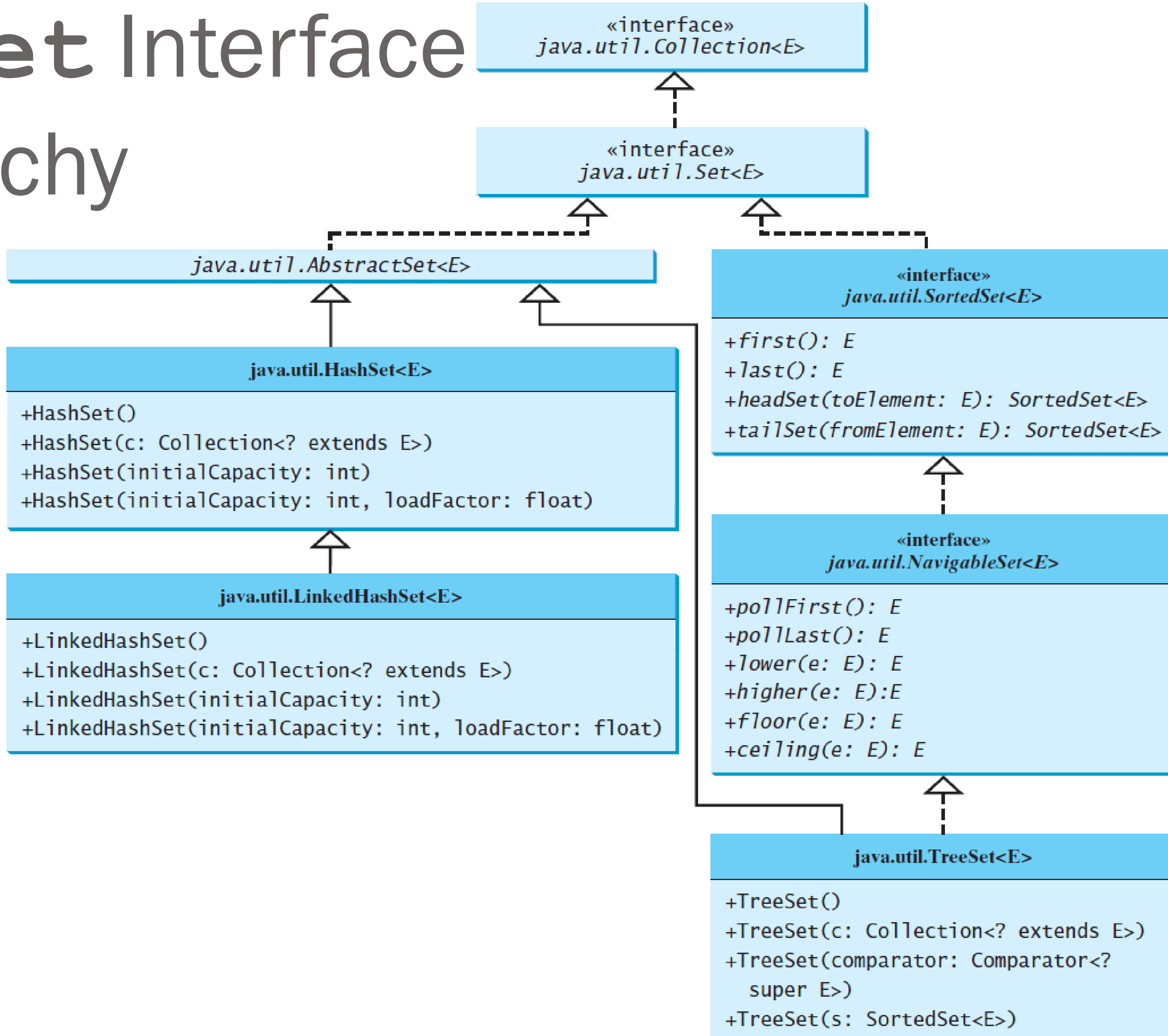
- Hash codes:
  - **`hashCode`** method is defined in the **`Object`** class
  - The hash codes of two objects must be the same if the two objects are **`equal`**
  - Two unequal objects may have the same hash code, but you should implement the **`hashCode`** method to avoid too many such cases
  - API Java hashcode examples:
    - **`hashCode`** in the **`Integer`** class returns its **`int`** value
    - **`hashCode`** in the **`Character`** class returns the Unicode of the character
    - **`hashCode`** in the **`String`** class returns

$$s_0 * 31^{(n - 1)} + s_1 * 31^{(n - 2)} + \ldots + s_{n - 1}$$

where s_i is **`s.charAt(i)`**.

31 is an **odd prime** with a nice property that the multiplication can be replaced by a shift and a subtraction for better performance: **`31*i == (i<<5)-i`**. Modern VMs do this sort of optimization automatically.

The **Set** Interface Hierarchy



The `HashSet` Class

- The **`HashSet`** class is a concrete class that implements **`Set`**
  - You can create an empty hash set using its no-arg constructor or create a hash set from an existing collection
- The elements are not stored in the order in which they are inserted into the set
  - There is no particular order for the elements in a hash set
  - To impose such an order on them, you need to use the **`LinkedHashSet`** class

The `HashSet` Class

- By default, the initial capacity is **`16`** and the load factor is **`0.75`**
  - The *load factor* is the number of elements in the set divided by the capacity
    - It is a value between **`0.0`** (the set is empty) and **`1.0`** (the set is full to capacity)
    - It measures how full the set is allowed to be before its capacity is increased
      - When the number of elements exceeds (greater or equal) the product of the capacity and load factor, the capacity is automatically doubled
    - For example, if the capacity is **`16`** and load factor is **`0.75`**, when the size reaches **`12`** (**`16*0.75 = 12`**) the capacity will be doubled to **`32`**
    - **A higher load factor decreases the space costs but increases the search time**
      - The default load factor **`0.75`** is a good tradeoff between time and space costs – we will see how search works when we implement hashing.
- The position of an element in the **`Set`** is close to the remainder of the division of the **`hashCode`** and the capacity

Example: Using HashSet and Iterator

- This example creates a hash set filled with strings, and uses an iterator to traverse the elements in the list.

```
import java.util.*;
public class TestHashSet {
  public static void main(String[] args) {
    // Create a hash set
    Set<String> set = new HashSet<>();
    // Add strings to the set
    set.add("London");
    set.add("Paris");
    set.add("New York");
    set.add("San Francisco");
    set.add("Beijing");
    set.add("New York");
    System.out.println(set);
    // Display the elements in the hash set
    for (String s: set) {
      System.out.print(s.toUpperCase() + " ");
    }
    System.out.println();
    // Process the elements using the forEach method
    set.forEach(e -> System.out.print(e.toLowerCase() + " "));
  }
}
```

The `HashSet` Class

- Since a set is an instance of **`Collection`**, all methods defined in **`Collection`** can be used for sets
  - Including **`for`**-each loops can be used to traverse all the elements in the set
  - **`Collection`** interface extends the **`Iterable`** interface, so the elements in a set are iterable

Collection methods

```
public class TestMethodsInCollection {
  public static void main(String[] args) {
    // Create set1
    java.util.Set<String> set1 = new java.util.HashSet<>();

    // Add strings to set1
    set1.add("London");
    set1.add("Paris");
    set1.add("New York");
    set1.add("San Francisco");
    set1.add("Beijing");

    System.out.println("set1 is " + set1);
    System.out.println(set1.size() + " elements in set1");

    // Delete a string from set1
    set1.remove("London");
    System.out.println("\nset1 is " + set1);
    System.out.println(set1.size() + " elements in set1");

    // Create set2
    java.util.Set<String> set2 = new java.util.HashSet<>();
```

```java
    // Add strings to set2
    set2.add("London");
    set2.add("Shanghai");
    set2.add("Paris");
    System.out.println("\nset2 is " + set2);
    System.out.println(set2.size() + " elements in set2");

    System.out.println("\nIs Taipei in set2? "
      + set2.contains("Taipei"));

    set1.addAll(set2);
    System.out.println("\nAfter adding set2 to set1, set1 is "
      + set1);

    set1.removeAll(set2);
    System.out.println("After removing set2 from set1, set1 is "
      + set1);

    set1.retainAll(set2);
    System.out.println("After removing common elements in set2 "
      + "from set1, set1 is " + set1);
  }
}
```

Output (cont.):

```
set1 is [San Francisco, New York, Paris, Beijing, London]
5 elements in set1

set1 is [San Francisco, New York, Paris, Beijing]
4 elements in set1

set2 is [Shanghai, Paris, London]
3 elements in set2

Is Taipei in set2? false

After adding set2 to set1, set1 is
 [San Francisco, New York, Shanghai, Paris, Beijing, London]

After removing set2 from set1, set1 is
 [San Francisco, New York, Beijing]

After removing common elements in set2 from set1, set1 is []
```

The `LinkedHashSet` Class

- **`LinkedHashSet`** extends **`HashSet`** with a linked-list implementation that supports an ordering of the elements in the set
  - The elements in a **`LinkedHashSet`** can be retrieved in the order in which they were inserted into the set

- To impose a different order (e.g., increasing or decreasing order), you can use the **`TreeSet`** class

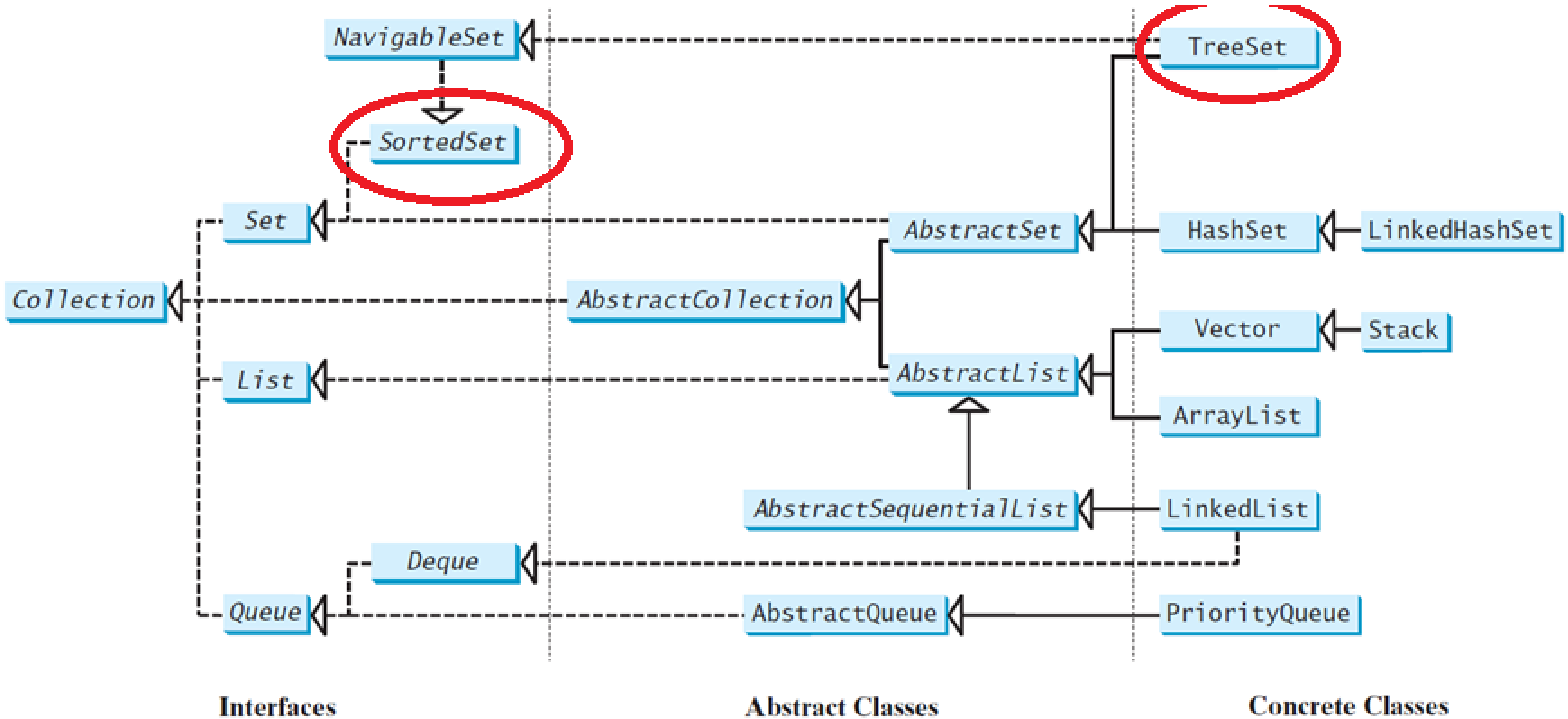
Example: Using LinkedHashSet

- This example creates a hash set filled with strings, and uses an iterator to traverse the elements in the list.

```
import java.util.*;
public class TestLinkedHashSet {
  public static void main(String[] args) {
    // Create a hash set
    Set<String> set = new LinkedHashSet<>();
    // Add strings to the set
    set.add("London");
    set.add("Paris");
    set.add("New York");
    set.add("San Francisco");
    set.add("Beijing");
    set.add("New York");

    System.out.println(set);

    // Display the elements in the hash set
    for (String element: set)
        System.out.print(element.toLowerCase() + " ");
  }
}
```

```
[London, Paris, New York, San Francisco, Beijing]
london paris new york san francisco beijing
```

The **SortedSet** Interface and the **TreeSet** Class



The `SortedSet` Interface and the `TreeSet` Class

- **`SortedSet`** is a sub-interface of **`Set`**, which guarantees that the elements in the set are sorted
- **`NavigableSet`** extends **`SortedSet`** to provide navigation methods **`lower(e)`**, **`floor(e)`**, **`ceiling(e)`**, and **`higher(e)`** that return elements respectively less than, less than or equal, greater than or equal, and greater than a given element and return **`null`** if there is no such element
- The **`pollFirst()`** and **`pollLast()`** methods remove and return the first and last element in the tree set, respectively
- **`headSet(toElement)`** and **`tailSet(fromElement)`** return a portion of the set whose elements are less than **`toElement`** and greater than or equal to **`fromElement`**, respectively

The **`SortedSet`** Interface and the **`TreeSet`** Class

- You can use an iterator to traverse the elements in the sorted order
  - The elements can be sorted in two ways
    - One way is to use the **`Comparable`** interface
    - The other way (*order by comparator*) is to specify a comparator for the elements in the set if the class for the elements does not implement the **`Comparable`** interface, or you don't want to use the **`compareTo`** method in the class that implements the **`Comparable`** interface

The `SortedSet` Interface and the `TreeSet` Class

- **`TreeSet`** is a concrete class that implements the **`SortedSet`** and **`NavigableSet`** interfaces
  - It provides the methods **`first()`** and **`last()`** for returning the first and last elements in the set
  - You can add objects into a tree set as long as they can be compared with each other
- The following example creates a hash set filled with strings, and then creates a tree set for the same strings
  - The strings are sorted in the tree set using the **`compareTo`** method in the **`Comparable`** interface

```
import java.util.*;

public class TestTreeSet {
  public static void main(String[] args) {
    // Create a hash set
    Set<String> set = new HashSet<>();

    // Add strings to the set
    set.add("London");
    set.add("Paris");
    set.add("New York");
    set.add("San Francisco");
    set.add("Beijing");
    set.add("New York");

    TreeSet<String> treeSet = new TreeSet<>(set);
    System.out.println("Sorted tree set: " + treeSet);

    // Use the methods in SortedSet interface
    System.out.println("first(): " + treeSet.first());
    System.out.println("last(): " + treeSet.last());
    System.out.println("headSet(\"New York\"): " +
      treeSet.headSet("New York"));
    System.out.println("tailSet(\"New York\"): " +
      treeSet.tailSet("New York"));
```

```
    // Use the methods in NavigableSet interface
    System.out.println("lower(\"P\"): " + treeSet.lower("P"));
    System.out.println("higher(\"P\"): " + treeSet.higher("P"));
    System.out.println("floor(\"P\"): " + treeSet.floor("P"));
    System.out.println("ceiling(\"P\"): " + treeSet.ceiling("P"));
    System.out.println("pollFirst(): " + treeSet.pollFirst());
    System.out.println("pollLast(): " + treeSet.pollLast());
    System.out.println("New tree set: " + treeSet);
  }
}
```

Output:

```
Sorted tree set: [Beijing, London, New York, Paris, San Francisco]
first(): Beijing
last(): San Francisco
headSet("New York"): [Beijing, London]
tailSet("New York"): [New York, Paris, San Francisco]
Sorted tree set: [Beijing, London, New York, Paris, San Francisco]

lower("P"): New York
higher("P"): Paris
floor("P"): New York
ceiling("P"): Paris

pollFirst(): Beijing
pollLast(): San Francisco
New tree set: [London, New York, Paris]
```

Example: Using `Comparator` to Sort Elements in a Set

- The following example creates a tree set of geometric objects
  - The geometric objects are sorted using the **`compare`** method in the **`Comparator`** interface

```java
import java.util.*;

public class TestTreeSetWithComparator {
  public static void main(String[] args) {
    // Create a tree set for geometric objects using a comparator
    Set<GeometricObject> set =
      new TreeSet<>(new GeometricObjectComparator());
    set.add(new Rectangle(4, 5));
    set.add(new Circle(40));
    set.add(new Circle(40));
    set.add(new Rectangle(4, 1));

    // Display geometric objects in the tree set
    System.out.println("A sorted set of geometric objects");

    for (GeometricObject element: set)
      System.out.println("area = " + element.getArea());
  }
}
```

```
A sorted set of geometric objects
area = 4.0
area = 20.0
area = 5021.548245743669
```

```
import java.util.Comparator;

public class GeometricObjectComparator
        implements Comparator<GeometricObject>,
                   java.io.Serializable {
// It is generally a good idea for comparators to implement
//  Serializable, as they may be used as ordering methods in
//  serializable data structures.
    public int compare(GeometricObject o1,
                  GeometricObject o2) {
        double area1 = o1.getArea();
        double area2 = o2.getArea();
        if (area1 < area2)
            return -1;
        else if (area1 == area2)
            return 0;
        else
            return 1;
    }
}
```

Performance of Sets and Lists

- **Sets are more efficient than lists for storing nonduplicate elements**
- **Lists are useful for accessing elements through the index**
- **Sets do not support indexing** because the elements in a set are unordered
  - To traverse all elements in a set, use a `for`-each loop or iterator

```java
import java.util.*;

public class SetListPerformanceTest {
  static final int N = 50000;

  public static long getTestTime(Collection<Integer> c) {
    long startTime = System.currentTimeMillis();

    // Test if a number is in the collection
    for (int i = 0; i < N; i++)
      c.contains((int)(Math.random() * 2 * N));

    return System.currentTimeMillis() - startTime;
  }

  public static long getRemoveTime(Collection<Integer> c) {
    long startTime = System.currentTimeMillis();

    for (int i = 0; i < N; i++)
      c.remove(i);

    return System.currentTimeMillis() - startTime;
  }
```

```java
public static void main(String[] args) {

  // Add numbers 0, 1, 2, ..., N - 1 to an array list
  //  to populate all data structures
  List<Integer> list = new ArrayList<>();
  for (int i = 0; i < N; i++)
    list.add(i);
  Collections.shuffle(list); // Shuffle the array list

  // Create a hash set, and test its performance
  Collection<Integer> set1 = new HashSet<>(list);
  System.out.println("Member test time for hash set is " +
    getTestTime(set1) + " milliseconds");
  System.out.println("Remove element time for hash set is " +
    getRemoveTime(set1) + " milliseconds");

  // Create a linked hash set, and test its performance
  Collection<Integer> set2 = new LinkedHashSet<>(list);
  System.out.println("Member test time for linked hash set is "
      + getTestTime(set2) + " milliseconds");
  System.out.println("Remove element time for linked hash set is "
      + getRemoveTime(set2) + " milliseconds");
```

```
    // Create a tree set, and test its performance
    Collection<Integer> set3 = new TreeSet<>(list);
    System.out.println("Member test time for tree set is " +
      getTestTime(set3) + " milliseconds");
    System.out.println("Remove element time for tree set is " +
      getRemoveTime(set3) + " milliseconds\n");


    // Create an array list, and test its performance
    Collection<Integer> list1 = new ArrayList<>(list);
    System.out.println("Member test time for array list is " +
      getTestTime(list1) + " milliseconds");
    System.out.println("Remove element time for array list is " +
      getRemoveTime(list1) + " milliseconds");

    // Create a linked list, and test its performance
    Collection<Integer> list2 = new LinkedList<>(list);
    System.out.println("Member test time for linked list is " +
      getTestTime(list2) + " milliseconds");
    System.out.println("Remove element time for linked list is " +
      getRemoveTime(list2) + " milliseconds");
  }
}
```

```
Member test time for hash set is 20 milliseconds
Remove element time for hash set is 27 milliseconds

Member test time for linked hash set is 27 milliseconds
Remove element time for linked hash set is 26
milliseconds

Member test time for tree set is 47 milliseconds
Remove element time for tree set is 34 milliseconds


Member test time for array list is 39802 milliseconds
Remove element time for array list is 16196 milliseconds

Member test time for linked list is 52197 milliseconds
Remove element time for linked list is 14870 milliseconds
```

Case Study: Counting Keywords

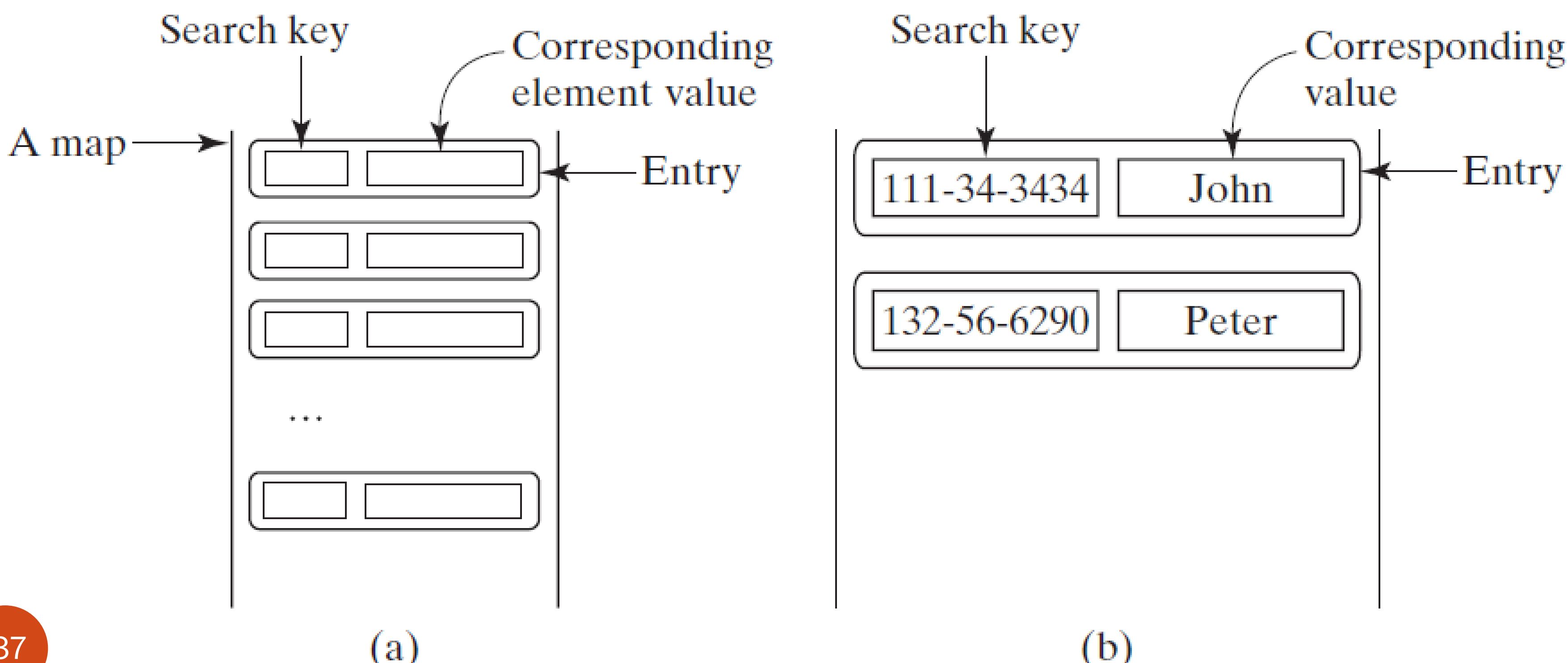
- An application that counts the number of the keywords in a Java source file
  - For each word in a Java source file, we need to determine whether the word is a keyword
  - To handle this efficiently, store all the keywords in a **`HashSet`** and use the **`contains`** method to test if a word is in the keyword set

```java
import java.util.*;
import java.io.*;

public class CountKeywords {
  public static void main(String[] args) throws Exception {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter a Java source file: ");
    String filename = input.nextLine();

    File file = new File(filename);
    if (file.exists()) {
      System.out.println("The number of keywords in " + filename
        + " is " + countKeywords(file));
    }
    else {
      System.out.println("File " + filename + " does not exist");
    }
  }

  public static int countKeywords(File file) throws Exception {
    // Array of all Java keywords + true, false and null
    String[] keywordString = {"abstract", "assert", "boolean",
        "break", "byte", "case", "catch", "char", "class", "const",
        "continue", "default", "do", "double", "else", "enum",
        "extends", "for", "final", "finally", "float", "goto",
        "if", "implements", "import", "instanceof", "int",
```

```
      "interface", "long", "native", "new", "packgrade", "private",
      "protected", "public", "return", "short", "static",
      "strictfp", "super", "switch", "synchronized", "this",
      "throw", "throws", "transient", "try", "void", "volatile",
      "while", "true", "false", "null"};

    Set<String> keywordSet =
      new HashSet<>(Arrays.asList(keywordString));
    int count = 0;

    Scanner input = new Scanner(file);

    while (input.hasNext()) {
      // read the file line by line
      String line = input.next();
      String[] words = line.split("\\W");
      for(String word:words)
        if (keywordSet.contains(word))
          count++;
    }

    return count;
  }
}
```

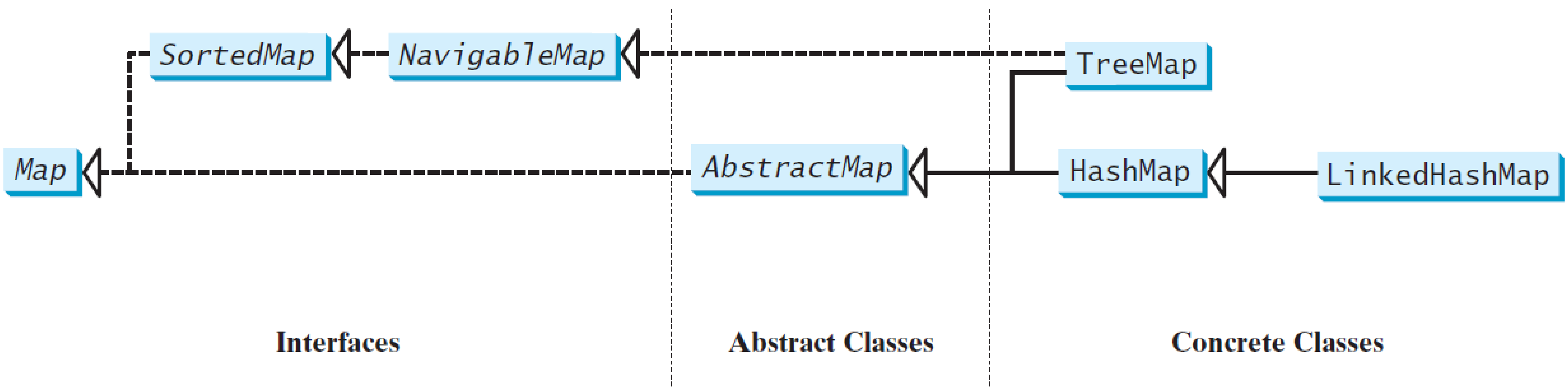
The **Map** Interface

- The **Map** interface maps keys to elements
  - The keys are like indexes, but can be anything (not restricted to integers
    - In **List**, the indexes are integer
    - In **Map**, the keys can be any objects



Map Interface and Class Hierarchy

- There are three types of maps: **HashMap**, **LinkedHashMap**, and **TreeMap**
  - The common features of these maps are defined in the **Map** interface



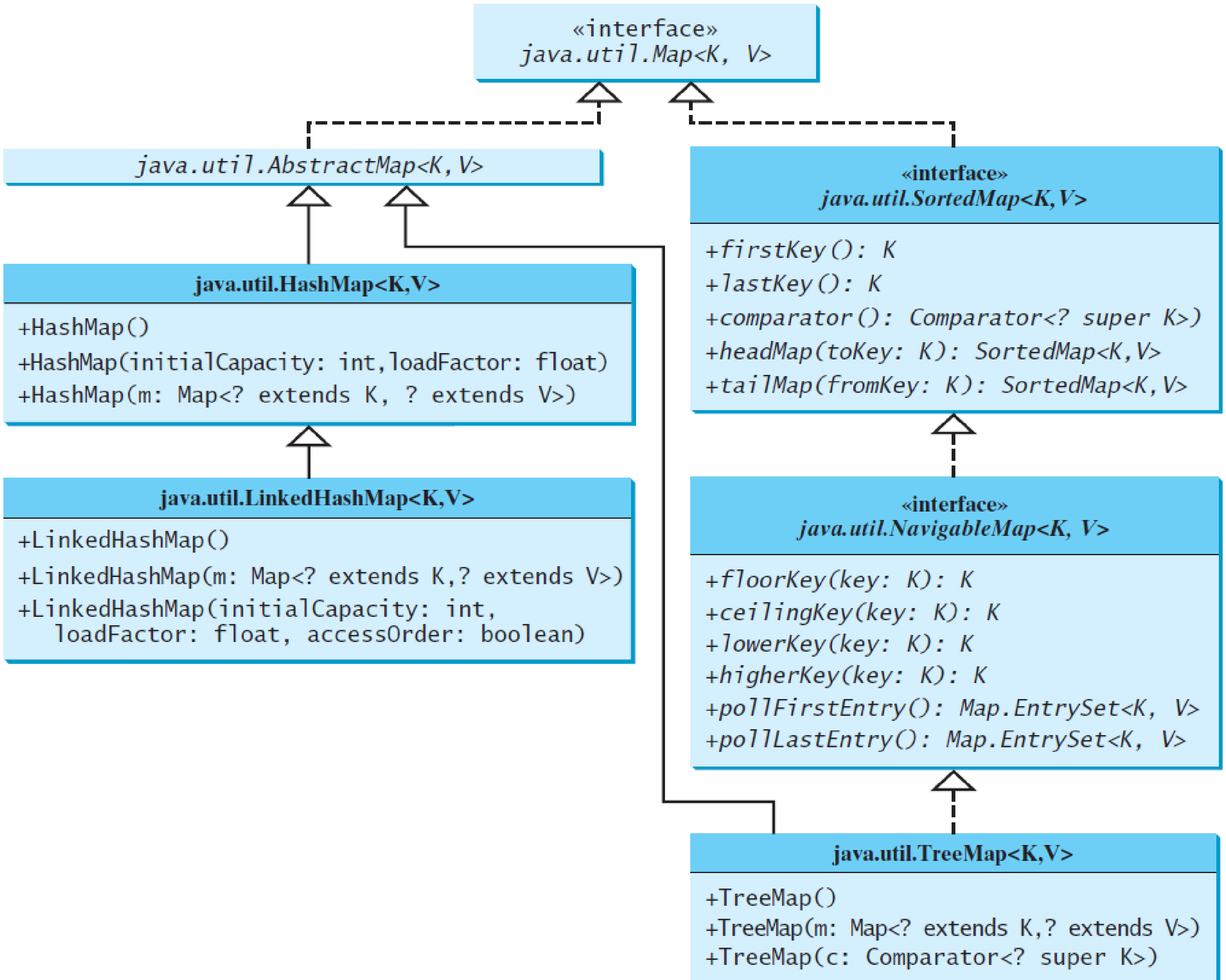
The **Map** Interface UML Diagram

- The **Map** interface provides the methods for querying, updating, and obtaining a collection of values and a set of keys

«interface»
java.util.Map<K,V>

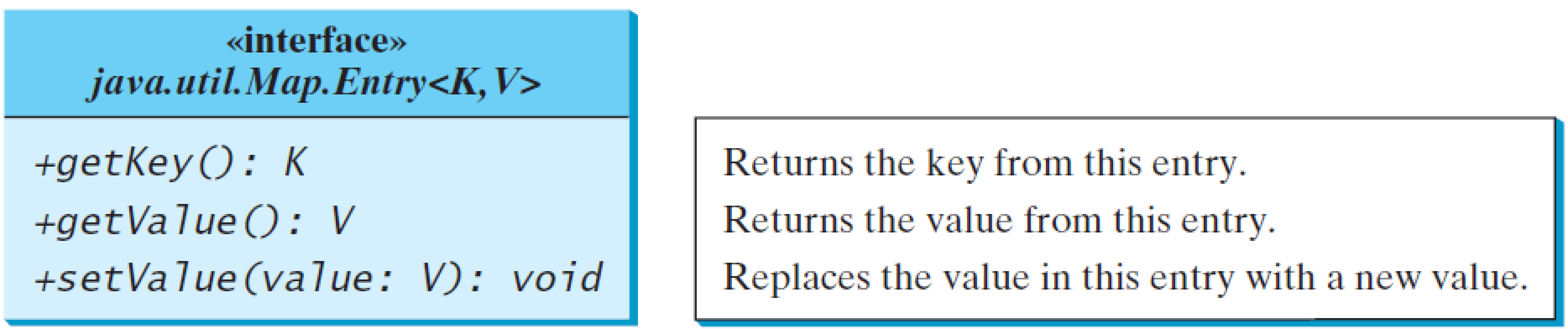
+clear(): void	Removes all entries from this map.
+containsKey(key: Object): boolean	Returns true if this map contains an entry for the specified key.
+containsValue(value: Object): boolean	Returns true if this map maps one or more keys to the specified value.
+entrySet(): Set<Map.Entry<K,V>>	Returns a set consisting of the entries in this map.
+get(key: Object): V	Returns the value for the specified key in this map.
+isEmpty(): boolean	Returns true if this map contains no entries.
+keySet(): Set<K>	Returns a set consisting of the keys in this map.
+put(key: K, value: V): V	Puts an entry into this map.
+putAll(m: Map<? extends K,? extends V>): void	Adds all the entries from m to this map.
+remove(key: Object): V	Removes the entries for the specified key.
+size(): int	Returns the number of entries in this map.
+values(): Collection<V>	Returns a collection consisting of the values in this map.

Concrete `Map` Classes



The **Map** Interface UML Diagram

- You can obtain a set of the keys in the map using the **keySet()** method
- The **entrySet()** method returns a set of entries
  - The entries are instances of the **Map.Entry** interface, where **Entry** is an inner interface for the **Map** interface



HashMap and **TreeMap**

- The **HashMap**, **LinkedHashMap** and **TreeMap** classes are the concrete implementations of the **Map** interface
- The **HashMap** class is efficient for locating a value, inserting a mapping, and deleting a mapping
- **LinkedHashMap** extends **HashMap** with a linked-list implementation that supports an ordering of the entries in the map
  - the entries in a **LinkedHashMap** can be retrieved either in the order in which they were inserted into the map (known as the insertion order) or in the order in which they were last accessed, from least recently to most recently accessed (access order).
- The **TreeMap** class, implementing **SortedMap**, is efficient for traversing the keys in a sorted order using the **Comparable** interface or the **Comparator** interface

Example: Using **`HashMap`** and **`TreeMap`**

- A hash map with the student's name as its key and the grade as its value
- The program then creates a tree map from the hash map and displays the entries in ascending order of the keys
- Finally, the program creates a linked hash map with access order, adds the same entries to the map, and displays the entries
  - E.g., the entry with the key Lewis is last accessed, so it is displayed last

```java
import java.util.*;

public class TestMap {
  public static void main(String[] args) {
    // Create a HashMap
    Map<String, Integer> hashMap = new HashMap<>();
    hashMap.put("Smith", 100);
    hashMap.put("Anderson", 91);
    hashMap.put("Lewis", 99);
    hashMap.put("Cook", 89);

    System.out.println("Display entries in HashMap");
    System.out.println(hashMap + "\n");

    // Create a TreeMap from the preceding HashMap
    Map<String, Integer> treeMap = new TreeMap<>(hashMap);
    System.out.println("Display entries in ascending order of key");
    System.out.println(treeMap);

    // Create a LinkedHashMap
    Map<String, Integer> linkedHashMap =
              new LinkedHashMap<>(16, 0.75f, true);
    linkedHashMap.put("Smith", 100);
    linkedHashMap.put("Anderson", 91);
    linkedHashMap.put("Lewis", 99);
    linkedHashMap.put("Cook", 89);
```

```
    // Display the grade for Lewis
    System.out.println("\nThe grade for " + "Lewis is " +
      linkedHashMap.get("Lewis"));

    System.out.println("Display entries in LinkedHashMap");
    System.out.println(linkedHashMap);

    // Display each entry with name and grade
    System.out.print("\nNames and grades are ");
    treeMap.forEach(
      (name, grade) -> System.out.print(name + ": " + grade + " "));
  }
}
```

```
Output:
  Display entries in HashMap
  {Cook=89, Smith=100, Lewis=99, Anderson=91}

  Display entries in ascending order of key
  {Anderson=91, Cook=89, Lewis=99, Smith=100}

  The grade for Lewis is 99

  Display entries in LinkedHashMap
  {Smith=100, Anderson=91, Cook=89, Lewis=99}
```

Case Study: Counting the Occurrences of Words in a Text

- This program counts the occurrences of words in a text and displays the words and their occurrences in ascending order of the words.
- The program uses a hash map to store a pair consisting of a word and its count.
- For each word, check whether it is already a key in the map.
  - If not, add the key and value 1 to the map.
  - Otherwise, increase the value for the word (key) by 1 in the map.
- To sort the map, we use a tree map.

```java
import java.util.*;

public class CountOccurrenceOfWords {
  public static void main(String[] args) {
    // Set text in a string
    String text = "Good morning. Have a good class. " +
      "Have a good visit. Have fun!";

    // Create a TreeMap to hold words as key and count as value
    Map<String, Integer> map = new TreeMap<>();

    String[] words = text.split("[\\W]+");
    for (int i = 0; i < words.length; i++) {
      String key = words[i].toLowerCase();

      if (key.length() > 0) {
        if (!map.containsKey(key)) {
          map.put(key, 1);
        }
        else {
          int value = map.get(key);
          value++;
          map.put(key, value);
        }
      }
    }
```

```
    // Display key and value for each entry
    map.forEach((k, v) -> System.out.println(k + "\t" + v));
  }
}

Output:
        a 2
        class 1
        fun 1
        good 3
        have 3
        morning 1
        visit 1
```