Java Collection Framework

Objectives

- To describe the Java Collections Framework hierarchy
- To use the common methods defined in the `Collection` interface for operating *sets* and *lists*
- To use the `Iterator` interface to traverse a `collection`
- To use the `for-each` loop to simplify traversing a `collection`
- To explore how and when to use `HashSet`, `LinkedHashSet` or `TreeSet` to store elements
- To compare elements using the `Comparable` interface and the `Comparator` interface
- To explore how and when to use `ArrayList` or `LinkedList` to store elements
- To use *the static utility methods* in the `Collections` class for sorting, searching, shuffling lists, and finding the largest and smallest element in collections
- To compare performance of *sets* and *lists*
- 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.

Java Collection Framework

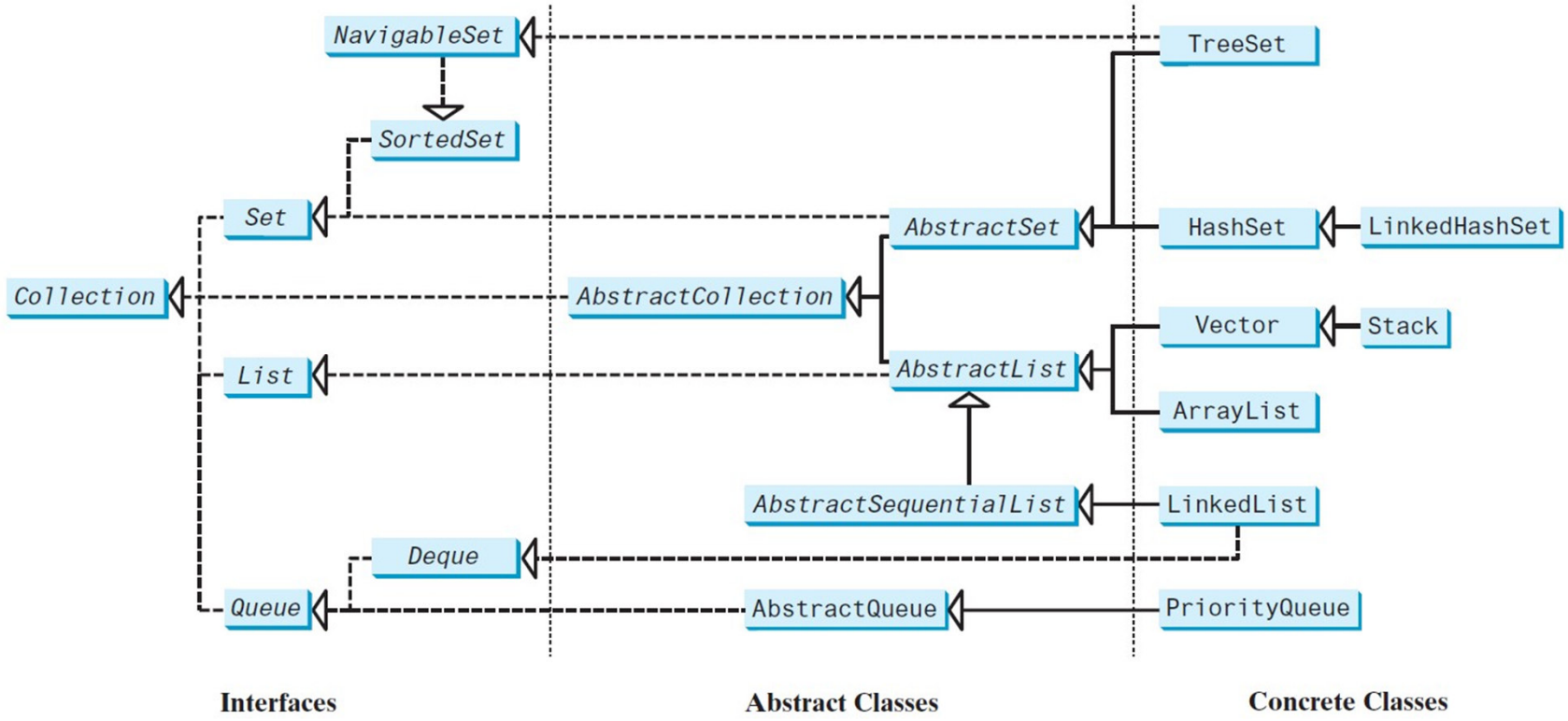
- A *data structure* is a collection of data organized in some fashion and not only stores data but also supports operations for manipulating and accessing the data.
- In object-oriented thinking, a data structure, also known as a *container* or *container object*, is an object that stores other objects, referred to as data or elements.
- To create a data structure is therefore to create an instance from the class in Java.
- Java provides several datastructures (lists, vectors, stacks, queues, priority queues, sets, and maps) that can be used to organize and manipulate data efficiently. They are commonly known as ***Java Collections Framework***.

Java Collection Framework hierarchy

- The Java Collections Framework supports two types of containers:
  - One for storing a collection of elements is simply called a ***collection***.
  - The other, for storing key/value pairs, is called a ***map***
- Maps are efficient data structures for quickly searching an element using a key.
- We will introduce the following collections:
  - `Set`s store a group of nonduplicate elements.
  - `List`s store an ordered collection of elements.
- The common operations of these collections are defined in the interfaces, and implementations are provided in concrete classes

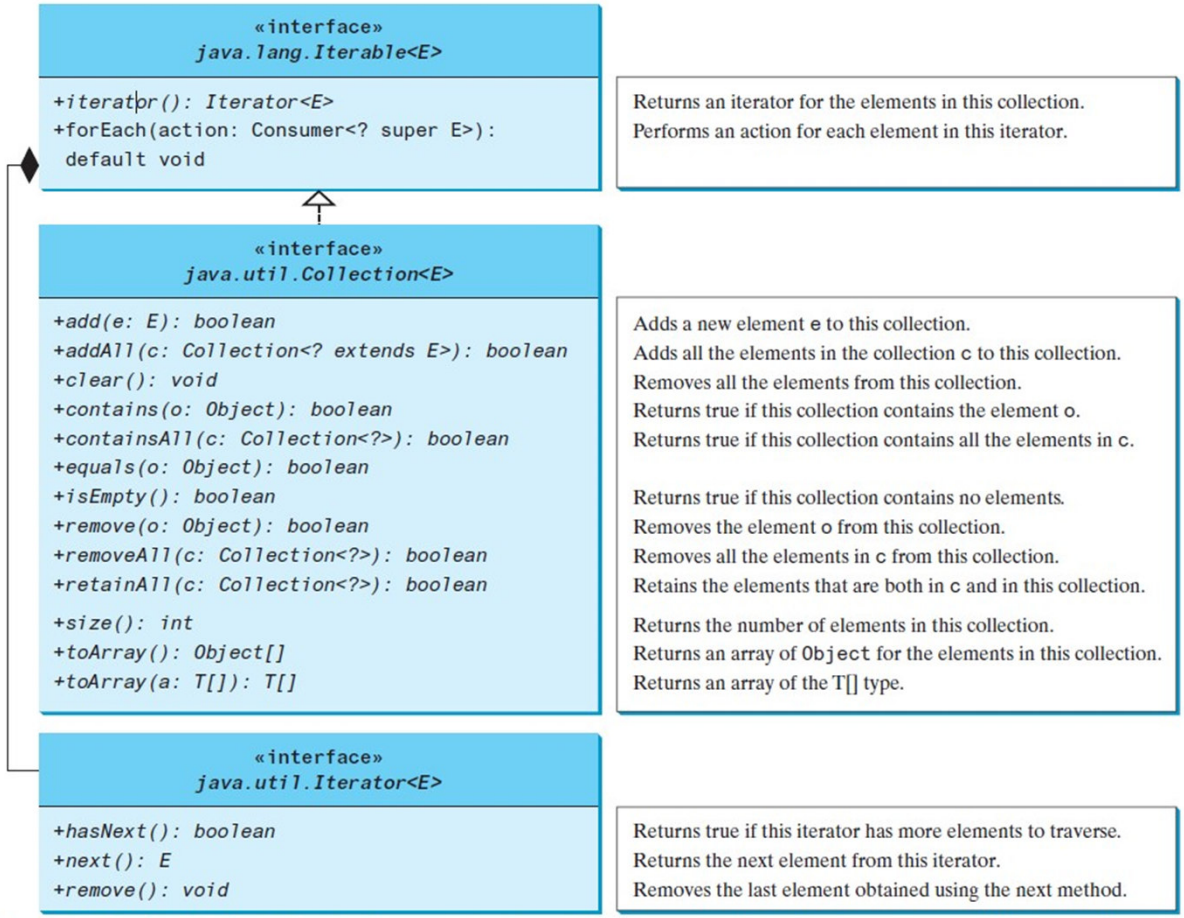
Java Collection Framework hierarchy, cont.

- `Set` and `List` are sub interfaces of `Collection`



The `Collection` Interface

- The **Collection** interface is the root interface for manipulating a collection of objects.
- The **AbstractCollection** class provides partial implementation for the **Collection** interface. It implements all the methods in **Collection** except the **add**, **size**, and **iterator** methods. These are implemented in the concrete subclasses.



Collection Example

```
import java.util.ArrayList;
import java.util.Collection;

public class TestCollection {

     public static void main(String[] args) {

          // create an ArrayList object
          ArrayList<String> collection1 = new ArrayList<>();

          // Add some elements in to colection1
          collection1.add("New York");
          collection1.add("Atlanta");
          collection1.add("Dallas");
          collection1.add("Madison");

          // display collection1
          System.out.println("A list of cities in collection1:");
          System.out.println(collection1);

          // using contain method to check whether the 'Dallas' in the collection or not
          System.out.println("\nIs Dallas in collection1? "
          + collection1.contains("Dallas"));

          collection1.remove("Dallas");
          System.out.println("\n" + collection1.size()
          + " cities are in collectionn1 now");
```

Collection Example

```
        // create new an new ArrayList object
        Collection<String> collection2 = new ArrayList<>();
        collection2.add("Seattle");
        collection2.add("Portland");
        collection2.add("Los Angeles");
        collection2.add("Atlanta");

        // show collection2
        System.out.println("\nA list of cities in collection2:");
        System.out.println(collection2);

        // make a copy of collection2
        ArrayList<String> c1 = (ArrayList<String>)(collection1.clone());
        c1.addAll(collection2);
        System.out.println("\nCities in collection1 or collection2: ");
        System.out.println(c1);

        c1 = (ArrayList<String>)(collection1.clone());
        c1.removeAll(collection2);
        System.out.print("\nCities in collection1, but not in 2: ");
        System.out.println(c1);
    }

}
```

Run

Iterators

- **Iterator** is a classic design pattern for walking through a data structure without having to expose the details of how data is stored in the data structure.
- The **Collection** interface extends the **Iterable** interface. The **Iterable** interface defines the **iterator** method, which returns an iterator. The **Iterator** interface provides a uniform way for traversing elements in various types of collections.
- Each collection is **Iterable**. You can obtain its **Iterator** object to traverse all the elements in the collection.

```
public interface Iterable<T> {
    /**
     * Returns an iterator over
     * elements of type {@code T}.
     *
     * @return an Iterator.
     */
    Iterator<T> iterator();
}
```

Iterators (cont.)

- The **iterator()** method in the **Iterable** interface returns an instance of **Iterator**, which provides sequential access to the elements in the collection using the **next()** method. You can also use the **hasNext()** method to check whether there are more elements in the iterator, and the **remove()** method to remove the last element returned by the iterator.

```
public interface Interator<E> {
boolean hasNext ();
E next();
void remove();
...

}
```

Example: Using Iterator

```
import java.util.ArrayList; ...

public class TestIterator {

    public static void main(String[] args) {
        Collection<String> collection = new ArrayList<>();
        collection.add("New York");
        collection.add("Atlanta");
        collection.add("Dallas");
        collection.add("Madison");

        Iterator<String> iterator = collection.iterator();
        while (iterator.hasNext()) {
            System.out.print(iterator.next().toUpperCase() + " ");
        }
        System.out.println();
    }
}
```

TIP: for-each loop

- You can simplify the code from previous example using a JDK 1.5 enhanced for loop without using an iterator, as follows:

```
for (String element: collection)
    System.out.print(element.toUpperCase() + " ");
```

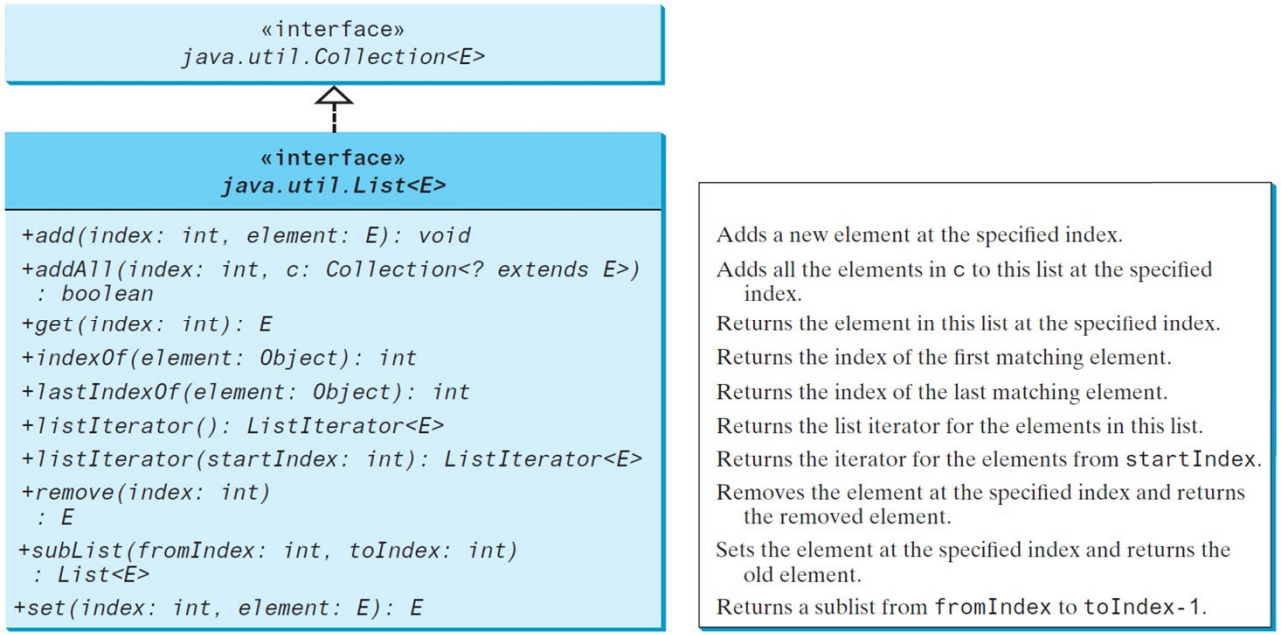
- Using Iterator :

```
Iterator<String> iterator = collection.iterator();
while (iterator.hasNext()) {
    System.out.print(iterator.next().toUpperCase() + " ");
}
```

- This simplification works for any instance of Iterable and arrays.

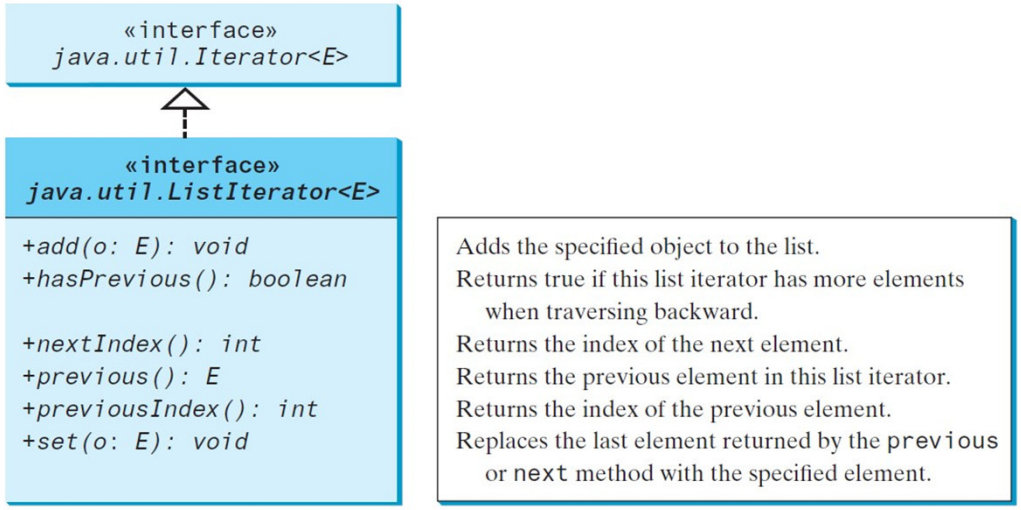
Lists

- The **List** interface extends the **Collection** interface and defines a collection for storing elements in a sequential order. To create a list, use one of its two concrete classes: **ArrayList** or **LinkedList**.



Lists

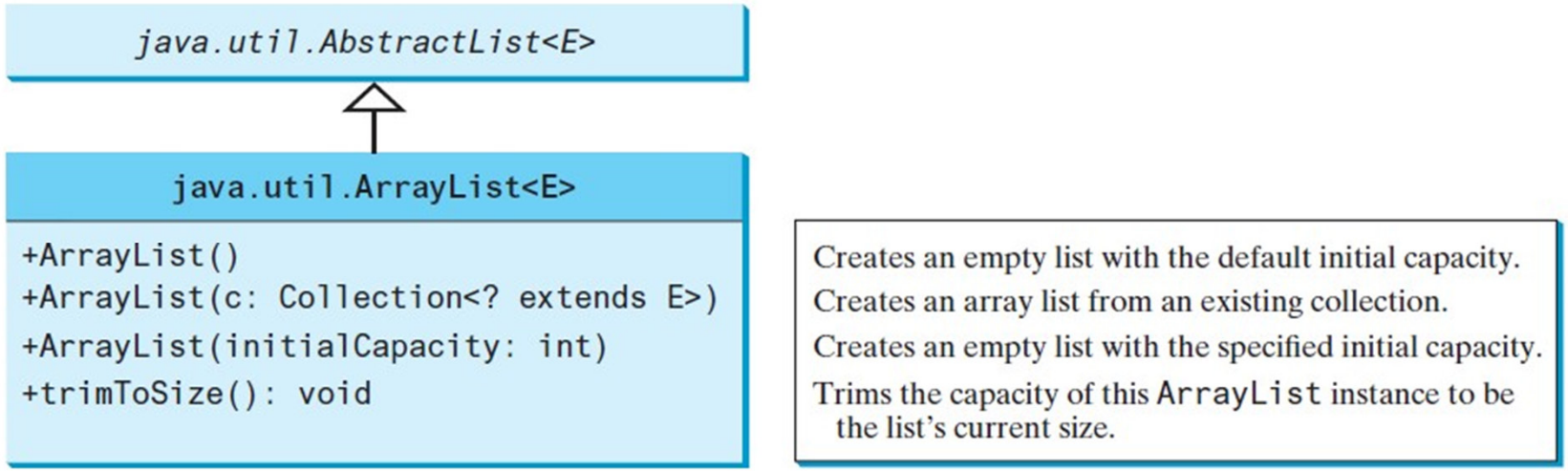
- The **listIterator()** or **listIterator(startIndex)** method returns an instance of **ListIterator**. The **ListIterator** interface extends the **Iterator** interface to add bidirectional traversal of the list



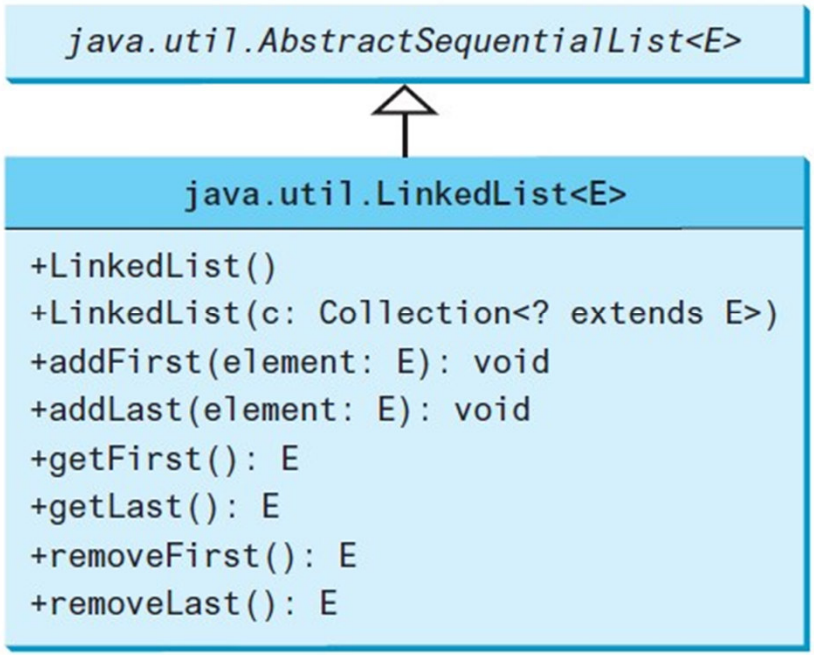
The `ArrayList` and `LinkedList` Classes

- The **ArrayList** class and the **LinkedList** class are two concrete implementations of the **List** interface.
- **ArrayList** stores elements in an array. The array is dynamically created. If the capacity of the array is exceeded, a larger new array is created and all the elements from the current array are copied to the new array.
- **LinkedList** stores elements in a *linked list*.
- If you need to support random access through an index without inserting or removing elements at the beginning of the list, **ArrayList** is the most efficient.
- If, however, your application requires the insertion or deletion of elements at the beginning of the list, you should choose **LinkedList**.
- A list can grow or shrink dynamically. Once it is created, an array is fixed. If your application does not require the insertion or deletion of elements, an array is the most efficient data structure.

Java.util.ArrayList



Java.util.LinkedList



Creates a default empty linked list.	
Creates a linked list from an existing collection.	
Adds the element to the head of this list.	
Adds the element to the tail of this list.	
Returns the first element from this list.	
Returns the last element from this list.	
Returns and removes the first element from this list.	
Returns and removes the last element from this list.	

Example: Array and LinkedList

```
import java.util.*;

public class TestArrayAndLinkList {
    public static void main(String[] args) {
        List<Integer> arrayList = new ArrayList<>();
        arrayList.add(1); // 1 is autoboxed to new Integer(1)
        arrayList.add(2);
        arrayList.add(3);
        arrayList.add(1);
        arrayList.add(4);
        arrayList.add(0, 10);
        arrayList.add(3, 30);

        System.out.println("A list of integers in the array list:");
        System.out.println(arrayList);
```

Example: Array and LinkedList

```
        LinkedList<Object> linkedList = new
        LinkedList<Object>(arrayList);
        linkedList.add(1, "red");
        linkedList.removeLast();
        linkedList.addFirst("green");

        System.out.println("Display the linked list forward:");
        ListIterator<Object> listIterator = linkedList.listIterator();
        while (listIterator.hasNext()) {
            System.out.print(listIterator.next() + " ");
        }
        System.out.println();

        System.out.println("Display the linked list backward:");
        listIterator = linkedList.listIterator(linkedList.size());
        while (listIterator.hasPrevious()) {
            System.out.print(listIterator.previous() + " ");
        }
    }
}
```

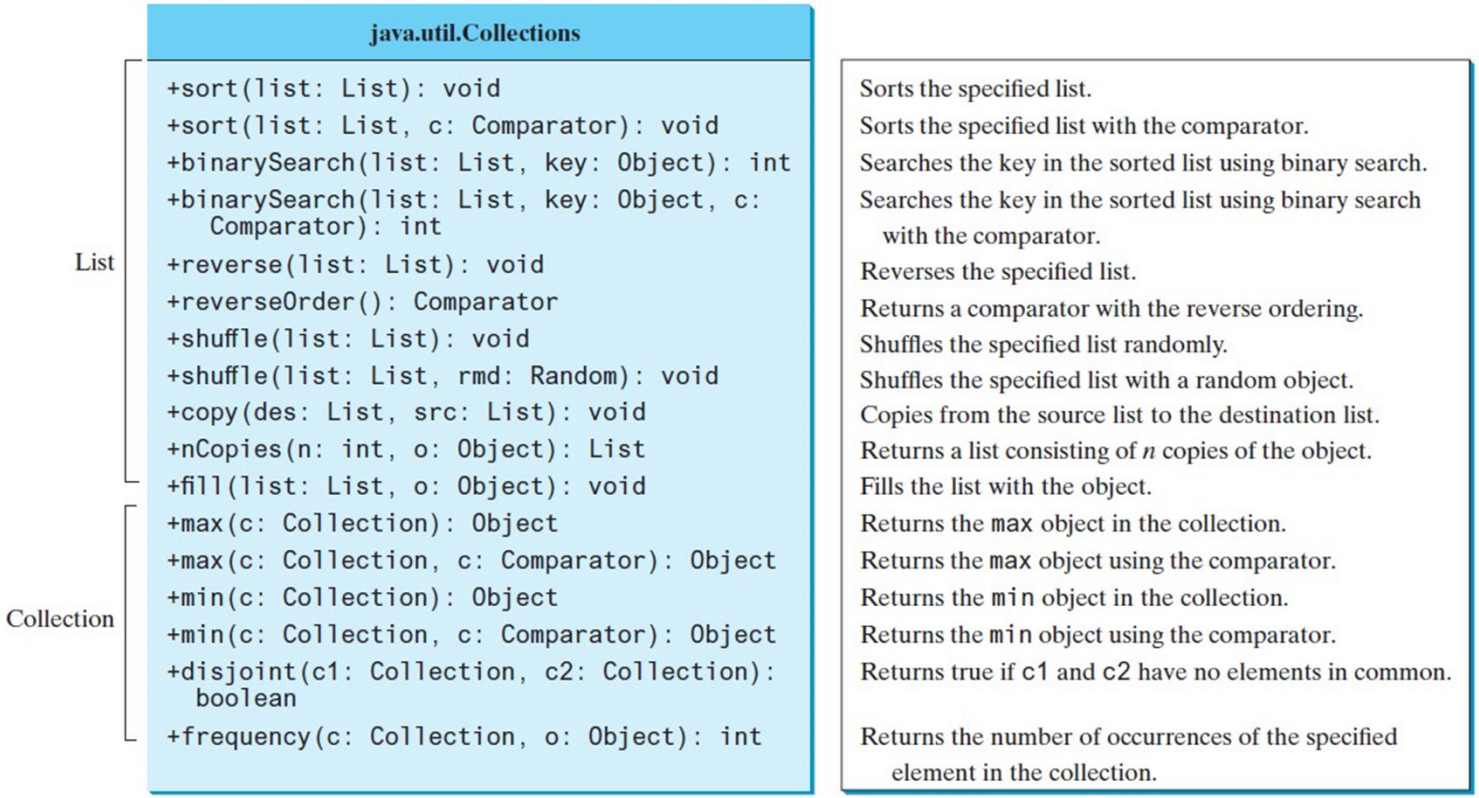
Tip

- Java provides the static `asList` method for creating a list from an array of a generic type. That is, it creates a list view of an array. This method acts as bridge between array-based and collection-based API

```
List<String> list1 = Arrays.asList("red", "green", "blue");
List<Integer> list2 = Arrays.asList(10, 20, 30, 40, 50);
```

The Collections Class

- *The* **Collections** *class contains static methods to perform common operations in a collection and a list.*



Using the Collections Class

```
// sort list ascending order
List<String> list = Arrays.asList("yellow", "red", "green", "blue");
Collections.sort(list);
System.out.println(list);

// sort list descending order
Collections.sort(list, Collections.reverseOrder());
System.out.println(list);
```

```
// use binarySearch method to search a key in a list
List<Integer> list1 = Arrays.asList(2, 4, 7, 10, 11, 45, 50, 59, 60, 66);
System.out.println("(1) Index: " + Collections.binarySearch(list1, 7));
System.out.println("(2) Index: " + Collections.binarySearch(list1, 9));

List<String> list2 = Arrays.asList("blue", "green", "red");
System.out.println("(3) Index: " + Collections.binarySearch(list2, "red"));
System.out.println("(4) Index: " + Collections.binarySearch(list2, "cyan"));
```

Using the Collections Class (cont.)

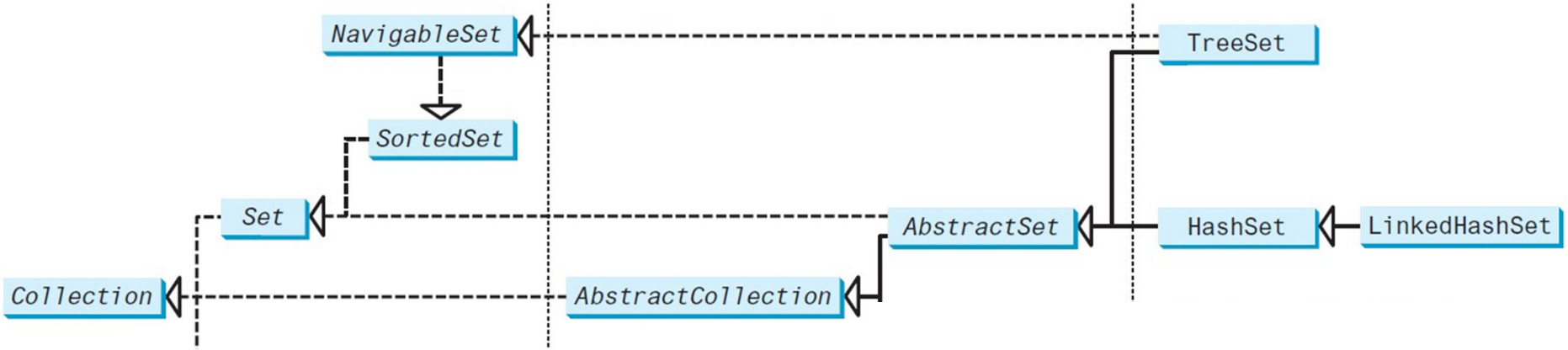
```
// use shuffle method to randomly reorder the elements in a list.
Collections.shuffle(list);
System.out.println(list);
```

```
// use the max and min methods for finding
// the maximum and minimum elements in a collection
Collection<String> collection = Arrays.asList("red", "green", "blue");
System.out.println(Collections.max(collection)); // Use Comparable
System.out.println(Collections.min(collection)); // Use Comparator
System.out.println(Collections.min(collection,
        Comparator.comparing(String::length))); // Use Comparator
```

Run

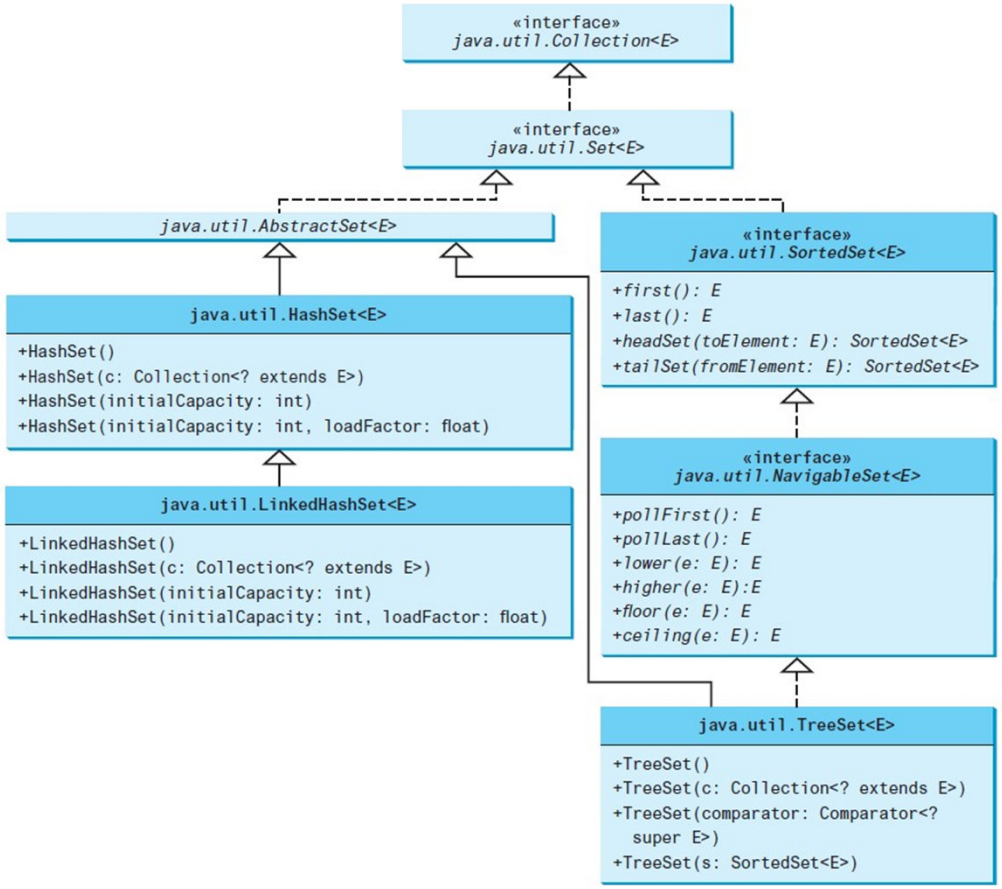
The Set Interface

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



- Three concrete classes of Set are HashSet, LinkedHashSet, and TreeSet, as shown in Figure。

The Set Interface Hierarchy



The AbstractSet Class

- The **AbstractSet** class extends **AbstractCollection** and partially implements **Set**. The **AbstractSet** class provides concrete implementations for the **equals** method and the **hashCode** method. The hash code of a set is the sum of the hash codes of all the elements in the set. Since the **size** method and **iterator** method are not implemented in the **AbstractSet** class, **AbstractSet** is an abstract class.

HashSet

- 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
- A **HashSet** can be used to store *duplicate-free* elements. For efficiency, objects added to a hash set need to implement the **hashCode** method in a manner that properly disperses the hash code.

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.

Run

```
1  import java.util.HashSet;
2  import java.util.Iterator;
3
4  public class TestHashSet {
5      public static void main(String[] args) {
6
7          // Create a hash set
8          HashSet<String> hset = new HashSet<>();
9
10         //Add strings to the set
11         hset.add("London");
12         hset.add("Paris");
13         hset.add("New York");
14         hset.add("San Francisco");
15         hset.add("Beijing");
16         hset.add("New York");
17
18         System.out.println(hset);
19         System.out.println("-----------------------------------------");
20         // Display the elements in the hash set
21         // Using for-each loop
22         for (String s: hset) {
23             System.out.print(s.toUpperCase() + "; ");
24         }
25         System.out.println();
26         System.out.println("-----------------------------------------");
27         // Using Iterator
28         Iterator<String> iter = hset.iterator();
29         while(iter.hasNext()) {
30             System.out.print(iter.next().toLowerCase() + "; ");
31         }
32
33     }
34 }
```

Example: TestMethodsInCollection

```
// Create set1
HashSet<String>  set1 = new HashSet<>();

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

// Get size
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
HashSet<String> set2 = new java.util.HashSet<>();

// 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 retaining common elements in set2 "
+ "and set2, set1 is " + set1);
```

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 **HashSet** are not ordered, but the elements in a **LinkedHashSet** can be retrieved in the order in which they were inserted into the set.
- **Tip**: If you don't need to maintain the order in which the elements are inserted, use HashSet, which is more efficient than LinkedHashSet.

```
// Create a linked hash set
LinkedHashSet<String> set = new LinkedHashSet<>();
// Add strings to the set
set.add("London");
set.add("Paris");
set.add("New York");
set.add("San Francisco");
System.out.println(set);

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

Run

The SortedSet Interface and the TreeSet Class

- **SortedSet** is a subinterface 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.
- **TreeSet** implements the **SortedSet, NavigableSet** interfaces. You can use an iterator to traverse the elements in the sorted order. The elements can be compared in two ways: using the **Comparable** interface or the **Comparator** interface.

The Comparator Interface

- Comarable Interface:

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

- Comparator Interface:

```
public interface Comarator<T> {
    public int compare(T o1, T o2);
    public boolean equals(Object obj);
}
```

- **Comparator** *can be used to compare the objects of a class that doesn't implement* **Comparable** *or define a new criteria for comparing objects.*

The Comparator Interface

- What if the elements' classes do not implement the **Comparable** interface? Can these elements be compared? You can define a *comparator* to compare the elements of different classes. To do so, define a class that implements the **java.util.Comparator<T>** interface and overrides its **compare** method.

```
public int compare(T o1, T o2);
```

- Returns a negative value if **o1** is less than **o2**, a positive value if **o1** is greater than **o2**, and zero if they are equal.

```
boolean equals(Object obj);
```

- Returns true if the specified object is also a comparator and imposes the same ordering as this comparator

Example: GeometricObjectComparator

```
public class GeometricObjectComparator
    implements Comparator<GeometricObject>{

    @Override
    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 -1;
        return 0;
    }
}
```

Example: TestComparator

- Write a program that demonstrates how to sort elements in a tree set using the Comparator interface. The example creates a tree set of geometric objects. The geometric objects are sorted using the compare method in the Comparator interface:

Example: TestComparator

```
public class TestComparator {
    public static void main(String[] args) {
    GeometricObject g1 = new Rectangle(5,5);
    GeometricObject g2 = new Circle(5);
    GeometricObject g = max (g1, g2, new GeometricObjectComparator());

    System.out.println("The area of the larger object is "
        + g.getArea());
    }

    public static GeometricObject max(GeometricObject g1,
        GeometricObject g2, Comparator<GeometricObject> c) {
    if (c.compare(g1, g2) > 0)
        return g1;
    else
        return g2;
    }
}
```

Example: Using TreeSet to Sort Elements in a Set

- This 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 for the same strings. The strings are sorted in the tree set using the compareTo method in the Comparable interface. The example also creates a tree set of geometric objects. The geometric objects are store using the compare method in the Comparator interface.

```java
import java.util.HashSet;
import java.util.TreeSet;

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

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

        System.out.println("HashSet: " + hset);

        // create a tree set
        TreeSet<String> set = new TreeSet<>(hset);
        System.out.println("Sorted tree set: " + set);
```

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

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

```
        // Create a tree set for geometric objects using a
        comparator
        TreeSet<GeometricObject> geoSet =
                new TreeSet<>(new GeometricObjectComparator());

        geoSet.add(new Rectangle(4,5));
        geoSet.add(new Circle(40));
        geoSet.add(new Circle(40));
        geoSet.add(new Rectangle(4,1));

        // Display geometric objects in the tree set
        System.out.println("A sorted set of geometric objects");
        for (GeometricObject element: geoSet)
            System.out.println("area = " + element.getArea());

    }
}
```

Run

Performance of Sets and Lists

- **Key Point**: `Sets` are more efficient than `lists` for storing nonduplicate elements. Lists are useful for accessing elements through the index.
- We now conduct an interesting experiment to test the performance of sets and lists.
  1. Testing whether an element is in a **hash set**, **linked hash set**, **tree set**, **array list**, or **linked list.**
  2. removing elements from a **hash set**, **linked hash set**, **tree set**, **array list**, and **linked list**.

```
public class SetListPerformanceTest {
    static final int N = 50000;

    public static void main(String[] args) {
    // Add numbers 0, 1, 2, ..., N – 1 to the array list
        ArrayList<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");
```

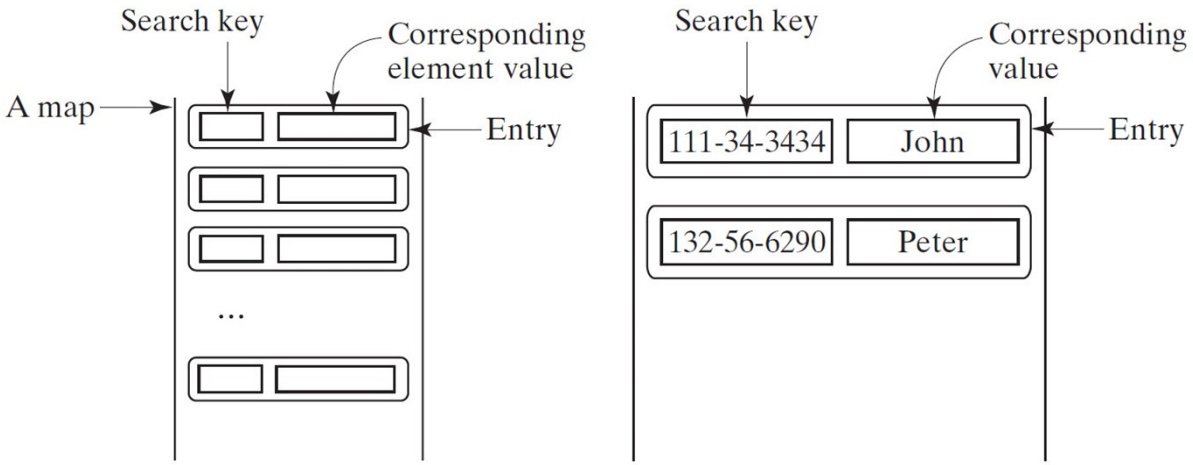
```
// Create an array list, and test its performance
Collection<Integer> list1 = new ArrayList<>(list);
System.out.println("Member test time for array list is " +
getTestTime(list1) + " milliseconds");
System.out.println("Remove element time for array list is " +
getRemoveTime(list1) + " milliseconds");

// 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");
}
```

```
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;
}
```

Run

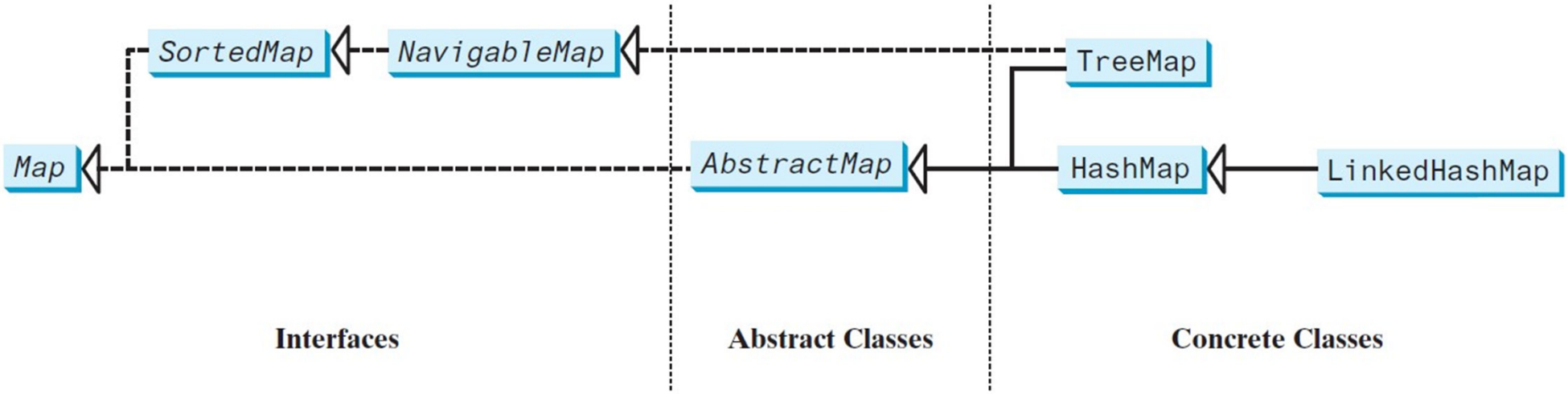
The Map Interface

- A *map* is a container object that stores a collection of key/value pairs. It enables fast retrieval, deletion, and updating of the pair through the key. A map stores the values along with the keys. The keys are like indexes. In **List**, the indexes are integers. In **Map**, the keys can be any objects.
- A map cannot contain duplicate keys. Each key maps to one value. A key and its corresponding value form an entry stored in a map,

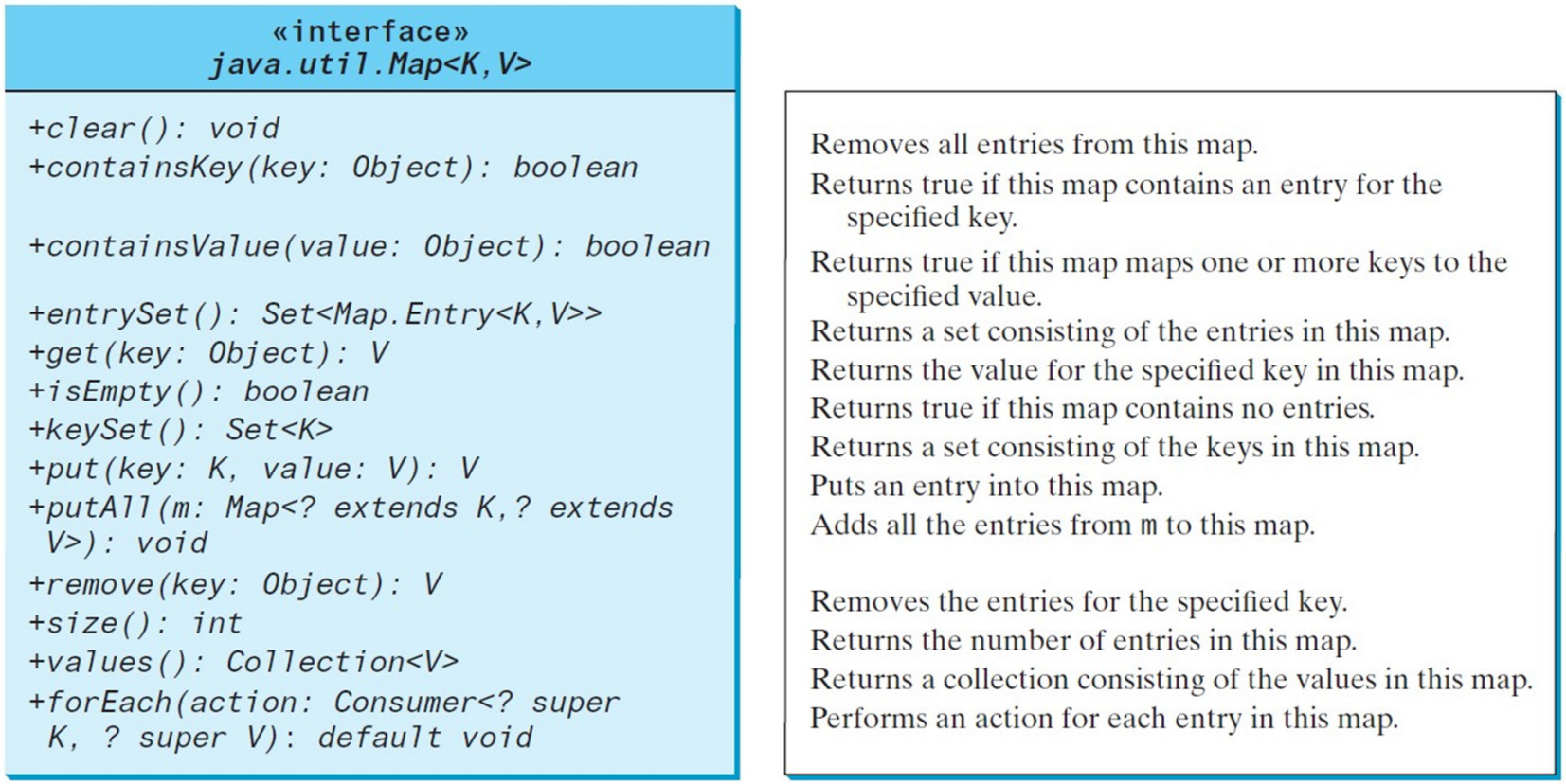


The Map Interface

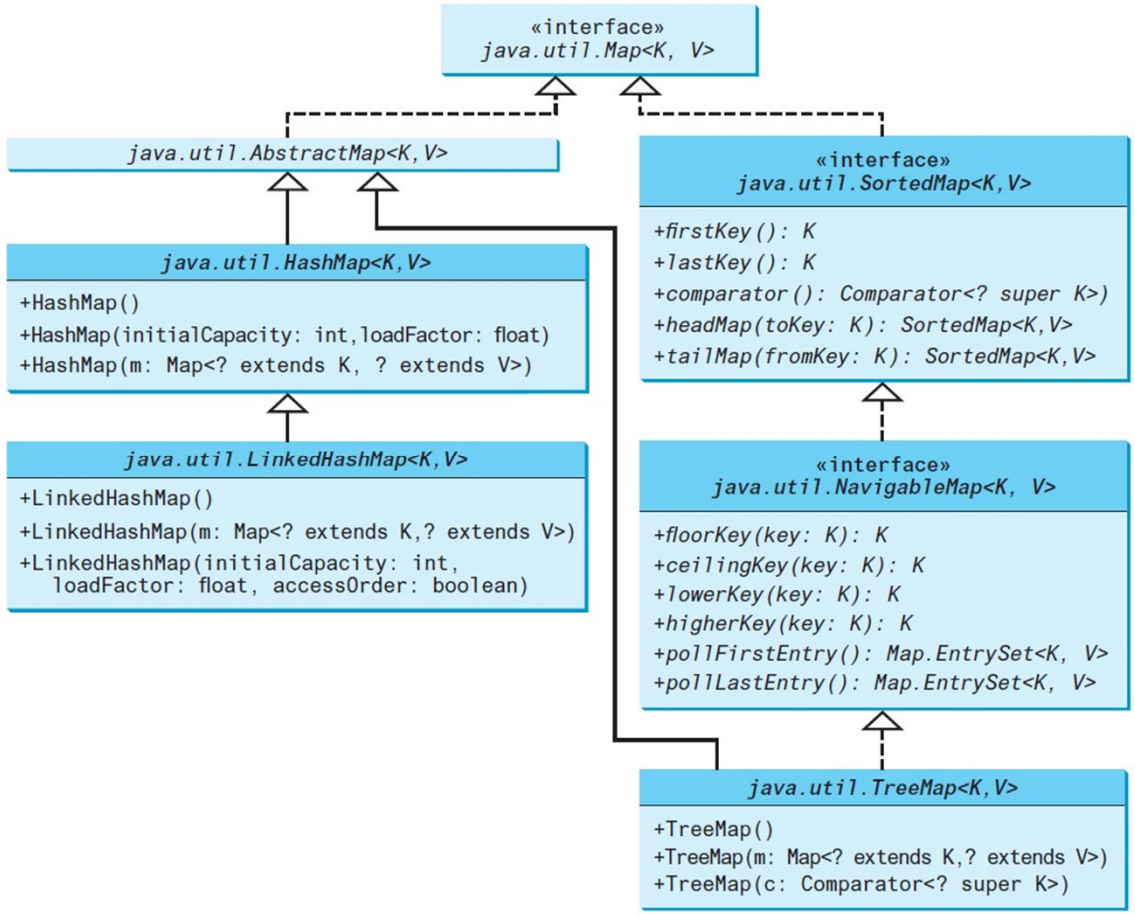
- There are three types of maps: **HashMap**, **LinkedHashMap**, and **TreeMap**. The common features of these maps are defined in the **Map** interface.



The Map Interface UML Diagram



Concrete Map Classes



Hash Map and TreeMap

- The **HashMap** class is efficient for locating a value, inserting an entry, and deleting an entry.
- The **TreeMap** class is efficient for traversing the keys in a sorted order. The keys can be sorted using the **Comparable** interface or the **Comparator** interface
- If you create a **TreeMap** using its no-arg constructor, the **compareTo** method in the **Comparable** interface is used to compare the keys in the map, assuming the class for the keys implements the **Comparable** interface. To use a comparator, you have to use the **TreeMap (Comparator comparator)** constructor to create a sorted map that uses the **compare** method in the comparator to order the entries in the map based on the keys.

LinkedHashMap

- **LinkedHashMap** extends **HashMap** with a linked-list implementation that supports an ordering of the entries in the map.
- The entries in a **HashMap** are not ordered, but 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 no-arg constructor constructs a **LinkedHashMap** with the insertion order. To construct a **LinkedHashMap** with the access order, use **LinkedHashMap(initialCapacity, loadFactor, true)**.

Example: Using HashMap and TreeMap

- This example creates a hash map, a linked hash map, and a tree map for mapping students to ages. The program first creates a hash map with the student's name as its key and the age 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, adds the same entries to the map, and displays the entries.

```java
public class TestMap {
    public static void main(String[] args) {
        // Create a HashMap
        HashMap<String, Integer> hashMap = new HashMap<>();
        hashMap.put("Smith", 30);
        hashMap.put("Anderson", 31);
        hashMap.put("Lewis", 29);
        hashMap.put("Cook", 29);
        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
        LinkedHashMap<String, Integer> linkedHashMap =
            new LinkedHashMap<>(16, 0.75f, true);
        linkedHashMap.put("Smith", 30);
        linkedHashMap.put("Anderson", 31);
        linkedHashMap.put("Lewis", 29);
        linkedHashMap.put("Cook", 29);
```

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

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

    // Display each entry with name and age
    System.out.println("\nNames and ages are ");
    treeMap.forEach(
      (name, age) -> System.out.println(name + ": " + age + " "));
  }

}
```

Run