Java Collections Framework
Sets & Maps

Instructor

Mehrnaz Zhian

Mehrnaz.zhian@senecacollege.ca

Introduction

- A *data structure* is a collection of data organized in some fashion.
- The structure not only stores data but also supports operations for accessing and manipulating 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.

Java Collections Framework

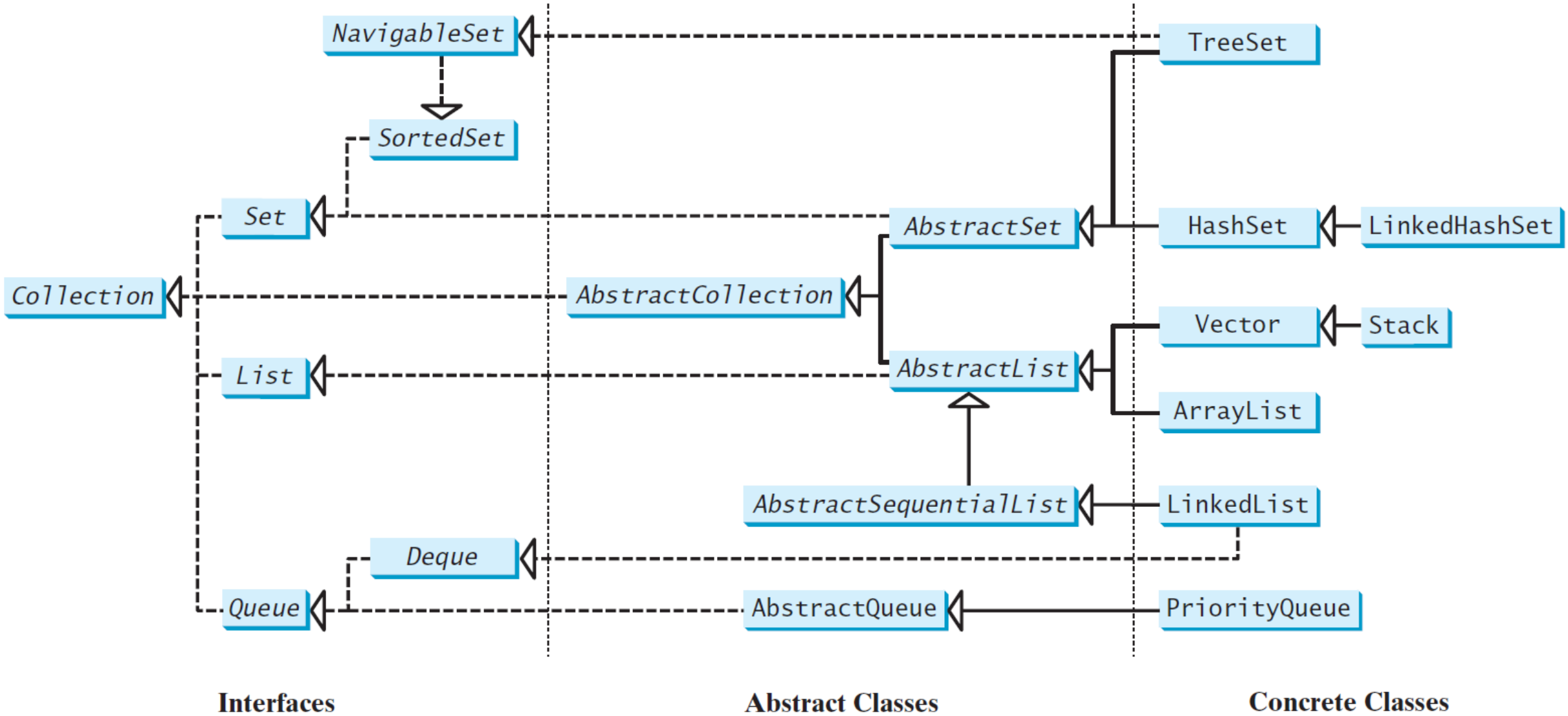
- For instance, the **ArrayList** class is a data structure to store elements in a list.

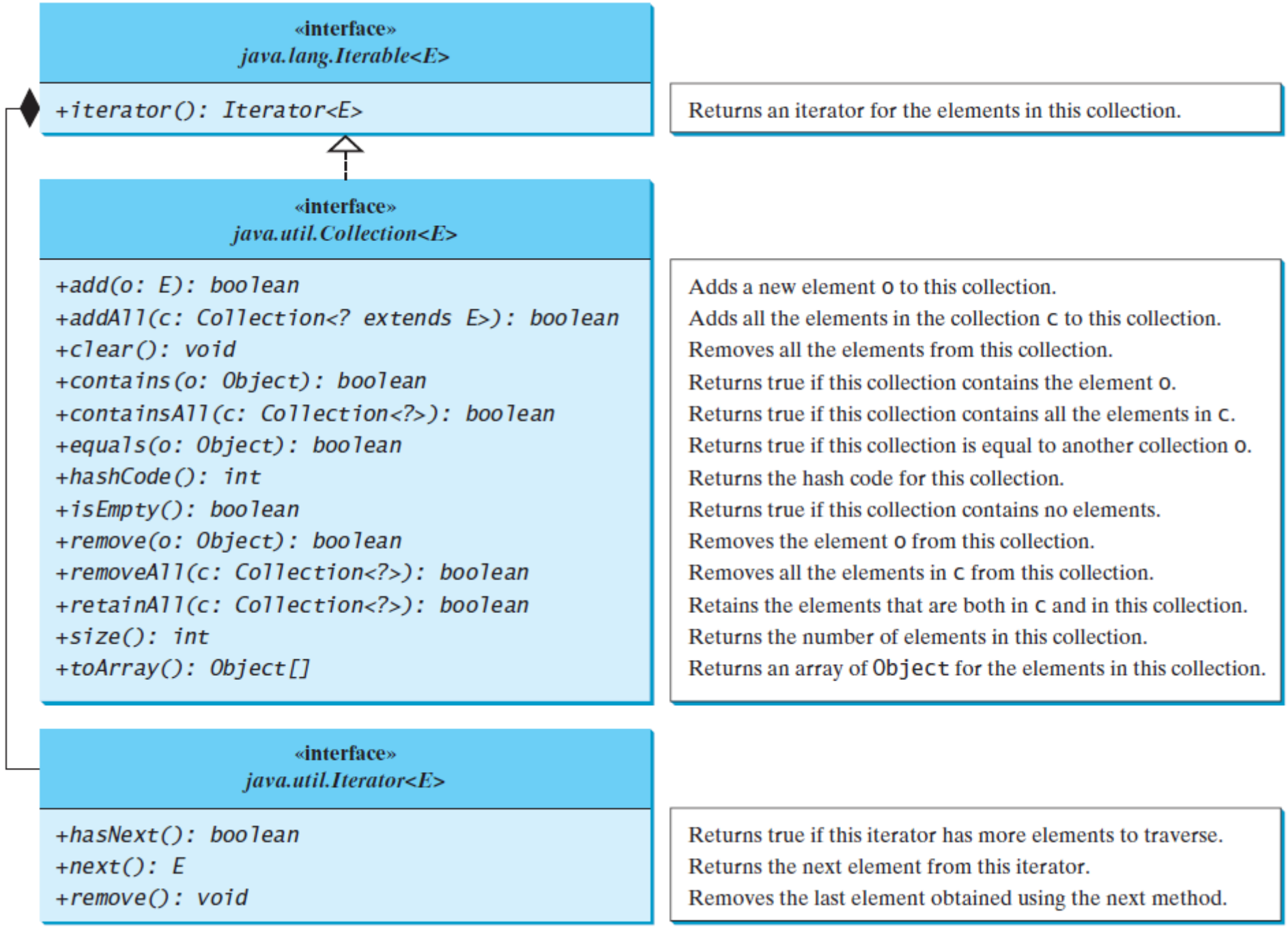
- Java provides several more data structures (such as lists, vectors, stacks, queues, priority queues, sets, and maps) that can be used to organize and manipulate data efficiently.

- These are commonly known as *Java Collections Framework*.

Collections

- 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*.
- There are different kinds of collections.
  - **List**s store an ordered collection of elements.
  - **Set**s store a group of non-duplicate elements.
  - **Stack**s store objects that are processed in a last-in, first-out fashion.
  - **Queue**s store objects that are processed in first-in, first-out fashion.


NavigableSet
SortedSet
Set
Collection
List
Deque
Queue
AbstractSet
AbstractCollection
AbstractList
AbstractSequentialList
AbstractQueue
TreeSet
HashSet
LinkedHashSet
Vector
Stack
ArrayList
LinkedList
PriorityQueue
Interfaces
Abstract Classes
Concrete Classes


«interface»
java.lang.Iterable<E>
+iterator(): Iterator<E>
Returns an iterator for the elements in this collection.
«interface»
java.util.Collection<E>
+add(o: E): boolean
+addAll(c: Collection<? extends E>): boolean
+clear(): void
+contains(o: Object): boolean
+containsAll(c: Collection<?>): boolean
+equals(o: Object): boolean
+hashCode(): int
+isEmpty(): boolean
+remove(o: Object): boolean
+removeAll(c: Collection<?>): boolean
+retainAll(c: Collection<?>): boolean
+size(): int
+toArray(): Object[]
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.
«interface»
java.util.Iterator<E>
+hasNext(): boolean
+next(): E
+remove(): void
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.

TestCollection.java

- It gives an example to use the methods defined in the C**ollection** interface.

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.

Iterators (Cont.)

- The **iterator** method in the **Collection** interface returns an instance of the **Iterator** interface, 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.

TestIterator.java

- It gives an example that uses the iterator to traverse all the elements in an array list.
- You can simplify the code using a for-each loop without using an iterator, as follows:

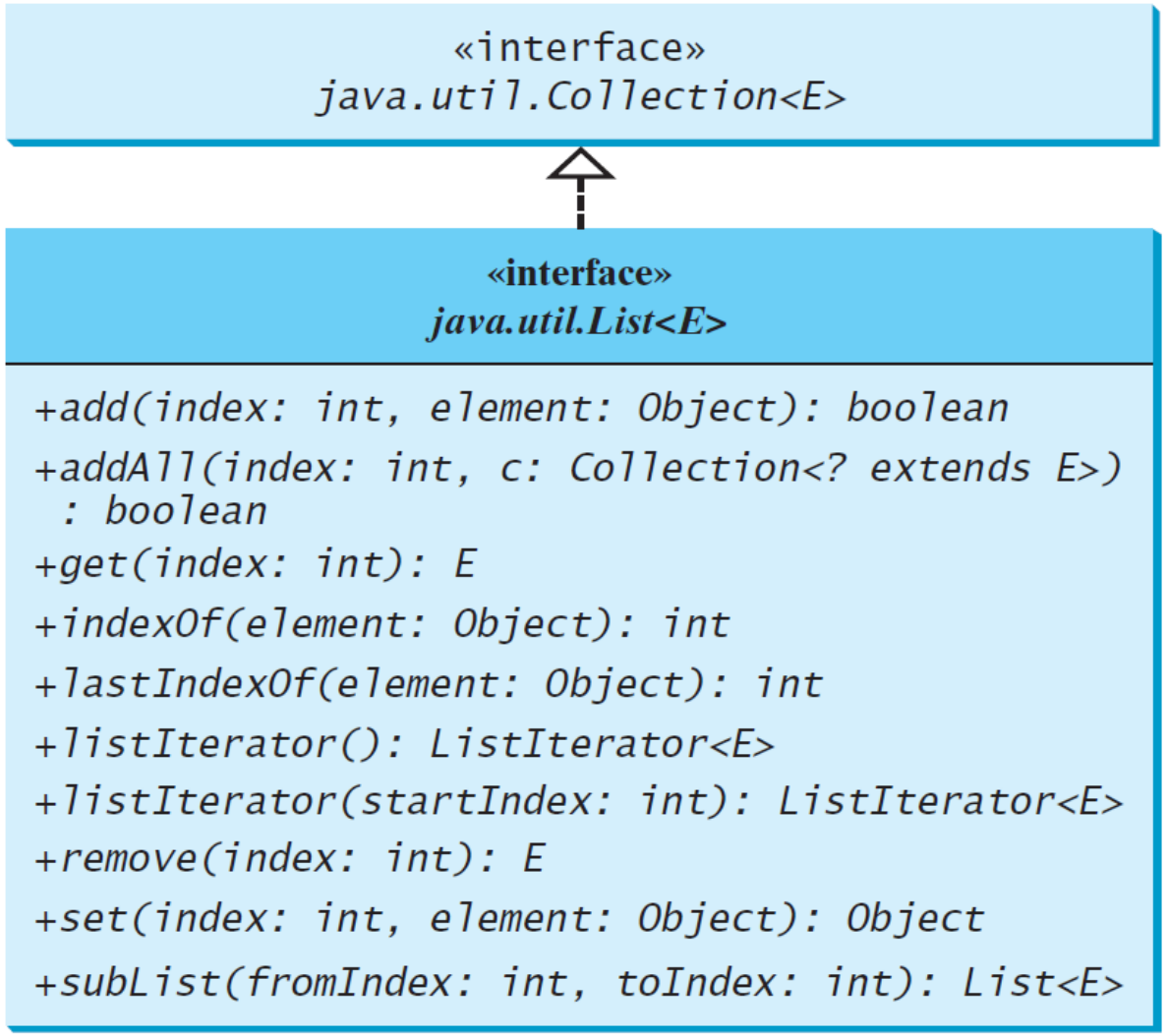
```
for (String element: collection)
  System.out.print(element.toUpperCase() + " ");
```

- This loop is read as “for each element in the collection, do the following.”
- The for-each loop can be used for arrays as well as any instance of **Iterable**.

The **List** interface stores elements in sequence and permits duplicates.

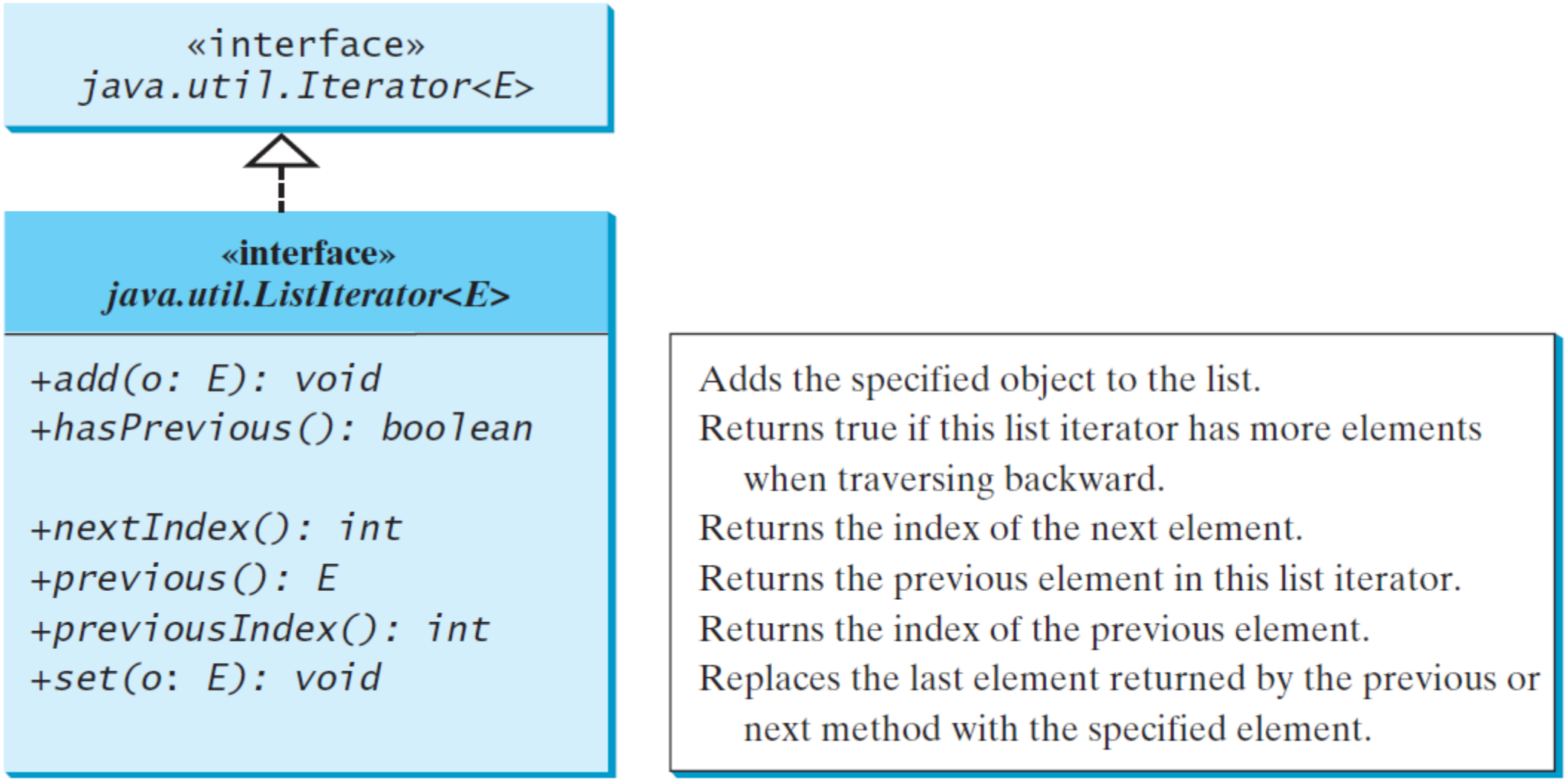
- **ArrayList** and **LinkedList** are defined under the **List** interface.
- The **List** interface extends **Collection** to define an ordered collection with duplicates allowed.
- The **List** interface adds position-oriented operations, as well as a new list iterator that enables the user to traverse the list bi-directionally.

The Common Methods in the **List** Interface



Adds a new element at the specified index.	
Adds all the elements in `c` to this list at the specified index.	
Returns the element in this list at the specified index.	
Returns the index of the first matching element.	
Returns the index of the last matching element.	
Returns the list iterator for the elements in this list.	
Returns the iterator for the elements from `startIndex`.	
Removes the element at the specified index.	
Sets the element at the specified index.	
Returns a sublist from `fromIndex` to `toIndex-1`.	

ListIterator enables traversal of a list bi-directionally



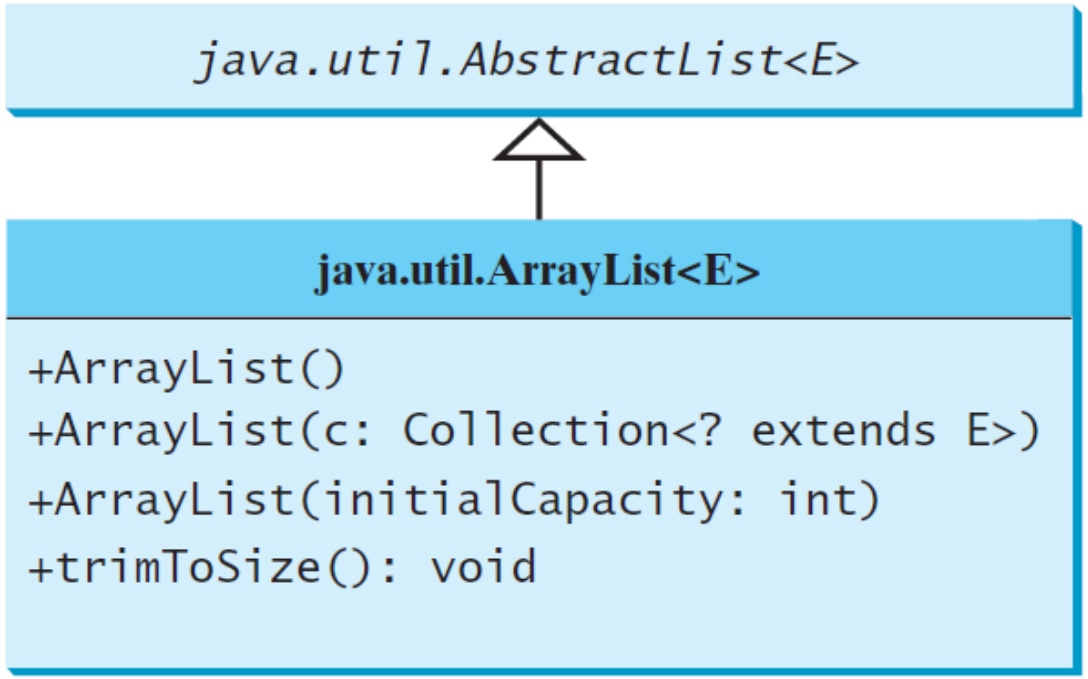
The **ArrayList** and **LinkedList** Class

- The **ArrayList** class and the **LinkedList** class are two concrete implementations of the **List** interface.

- If you need to support random access through an index without inserting or removing elements at the beginning of the list, **ArrayList** offers the most efficient collection.

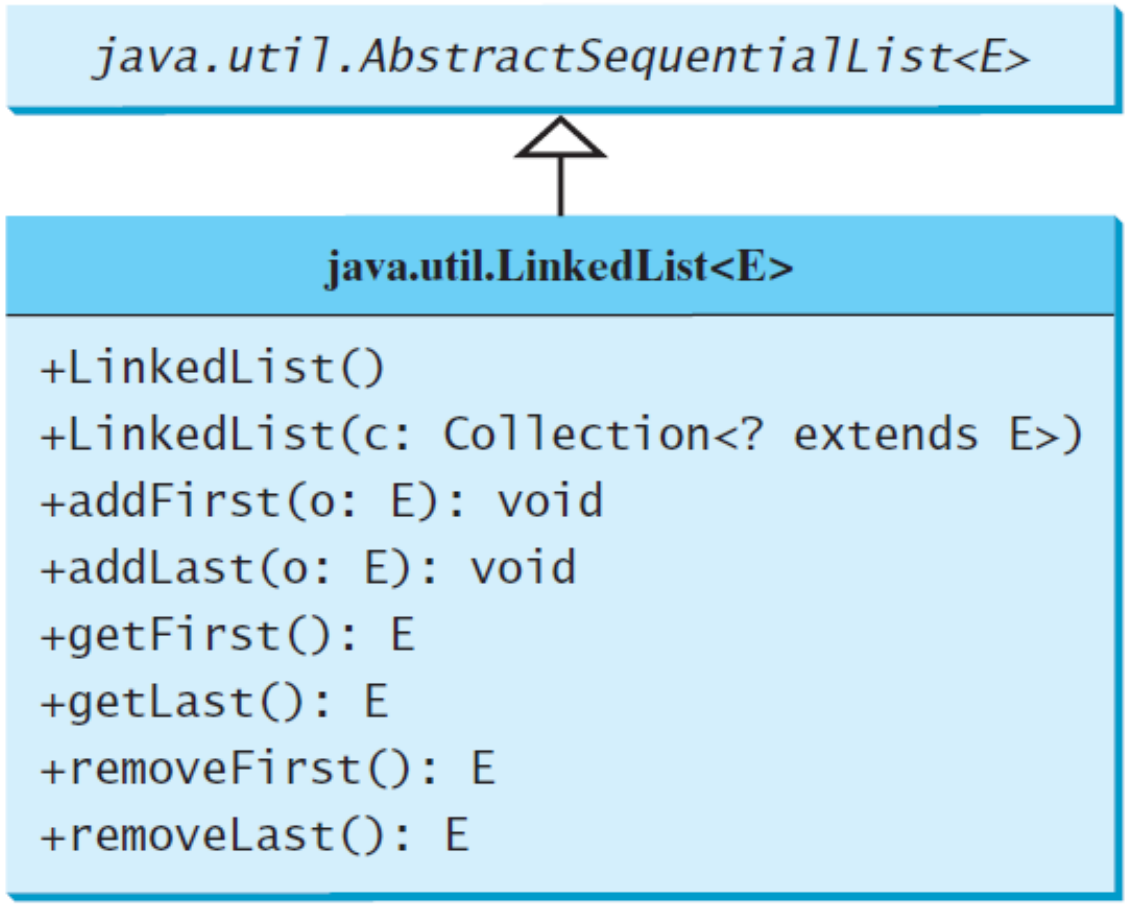
- If, however, your application requires the insertion or deletion of elements at the beginning of the list, you should choose **LinkedList**.

ArrayList implements **List** using an array.



Creates an empty list with the default initial capacity.	
Creates an array list from an existing collection.	
Creates an empty list with the specified initial capacity.	
Trims the capacity of this `ArrayList` instance to be the list's current size.	

LinkedList provides methods for adding and inserting elements at both ends of the list.



Creates a default empty linked list.	
Creates a linked list from an existing collection.	
Adds the object to the head of this list.	
Adds the object 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.	

TestArrayAndLinkedList.java

- It gives a program that creates an array list filled with numbers and inserts new elements into specified locations in the list.
- The example also creates a linked list from the array list and inserts and removes elements from the list.
- Finally, the example traverses the list forward and backward.

TestArrayAndLinkedList.java

- A list can hold identical elements. Integer **1** is stored twice in the list.
- **ArrayList** and **LinkedList** operate similarly. The critical difference between them pertains to internal implementation, which affects their performance.
- **ArrayList** is efficient for retrieving elements and **LinkedList** is efficient for inserting and removing elements at the beginning of the list.

TestArrayAndLinkedList.java

- The **get(i)** method is available for a linked list, but it is a time-consuming operation. Do not use it to traverse all the elements in a list as shown in (a).
- Instead you should use an iterator as shown in (b). Note that a for-each loop uses an iterator implicitly.

```
for (int i = 0; i < linkedList.size(); i++) {
  process linkedList.get(i);
}
```

(a) Very inefficient

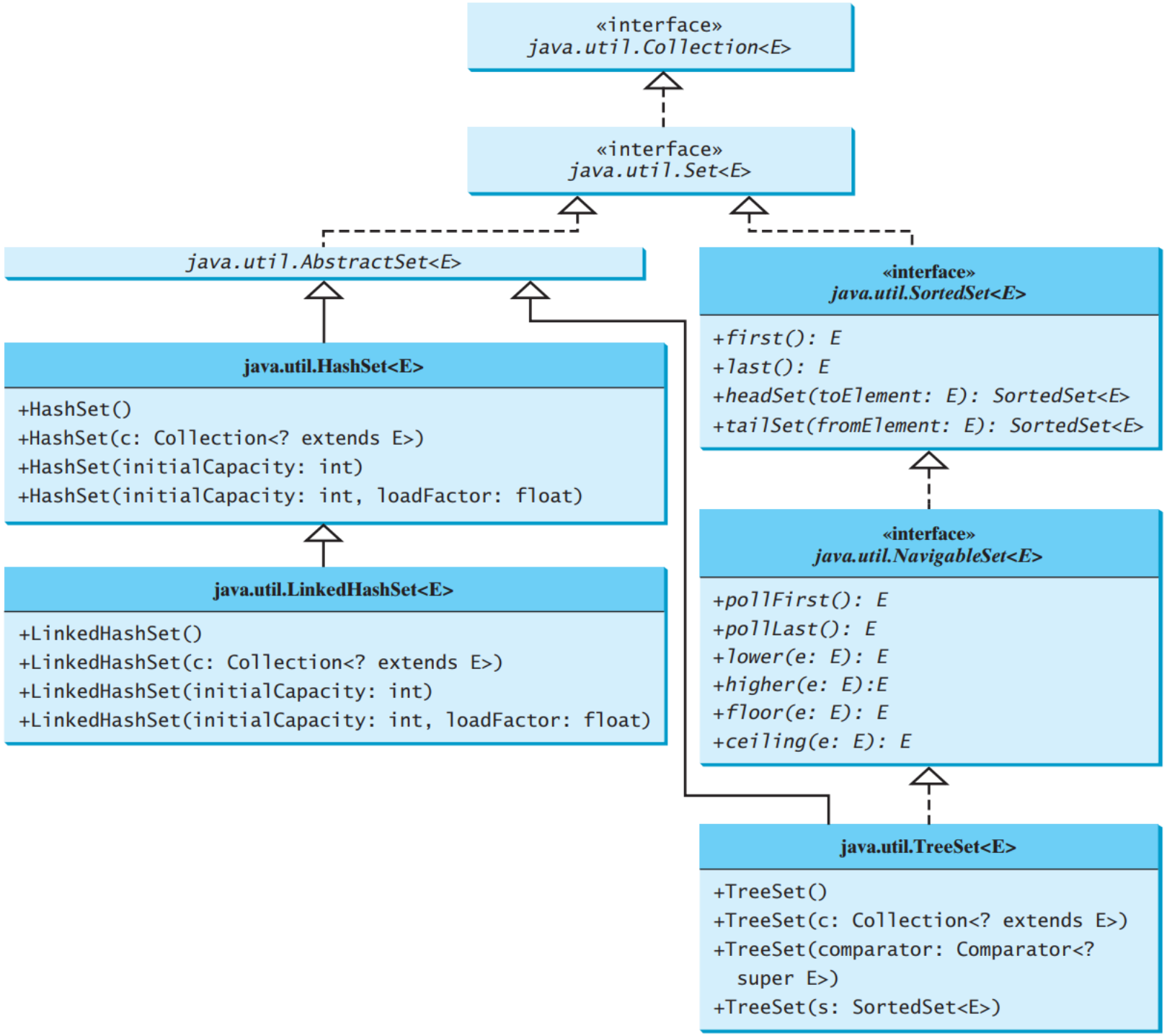
```
for (listElementType s: linkedList) {
  process s;
}
```

(b) Efficient

Set

- A set is an efficient data structure for storing and processing nonduplicate elements.

- You can create a set using one of its three concrete classes: **HashSet**, **LinkedHashSet**, or TreeSet.


«interface»
java.util.Collection<E>
«interface»
java.util.Set<E>
java.util.AbstractSet<E>
«interface»
java.util.SortedSet<E>
+first(): E
+last(): E
+headSet(toElement: E): SortedSet<E>
+tailSet(fromElement: E): SortedSet<E>
java.util.HashSet<E>
+HashSet()
+HashSet(c: Collection<? extends E>)
+HashSet(initialCapacity: int)
+HashSet(initialCapacity: int, loadFactor: float)
«interface»
java.util.NavigableSet<E>
+pollFirst(): E
+pollLast(): E
+lower(e: E): E
+higher(e: E):E
+floor(e: E): E
+ceiling(e: E): E
java.util.LinkedHashSet<E>
+LinkedHashSet()
+LinkedHashSet(c: Collection<? extends E>)
+LinkedHashSet(initialCapacity: int)
+LinkedHashSet(initialCapacity: int, loadFactor: float)
java.util.TreeSet<E>
+TreeSet()
+TreeSet(c: Collection<? extends E>)
+TreeSet(comparator: Comparator<?
super E>)
+TreeSet(s: SortedSet<E>)

Examples

- TestHashSet.java
- TestMethodsInCollection.java

LinkedHashSet

- **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.

Example

- TestLinkedHashSet.java

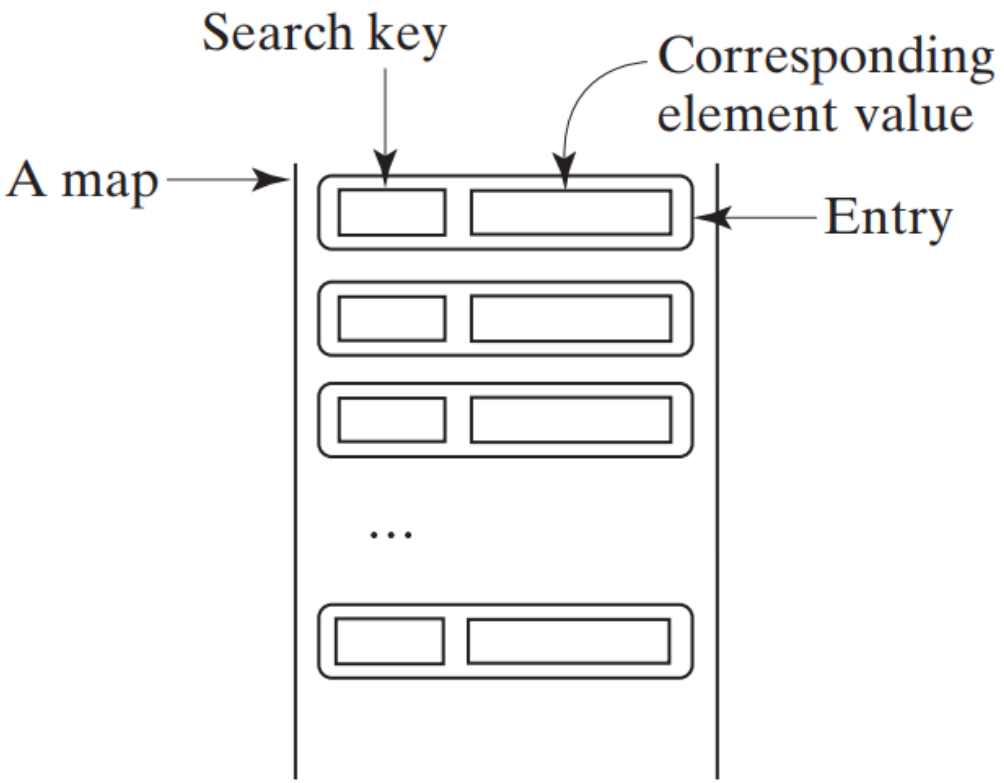
Map

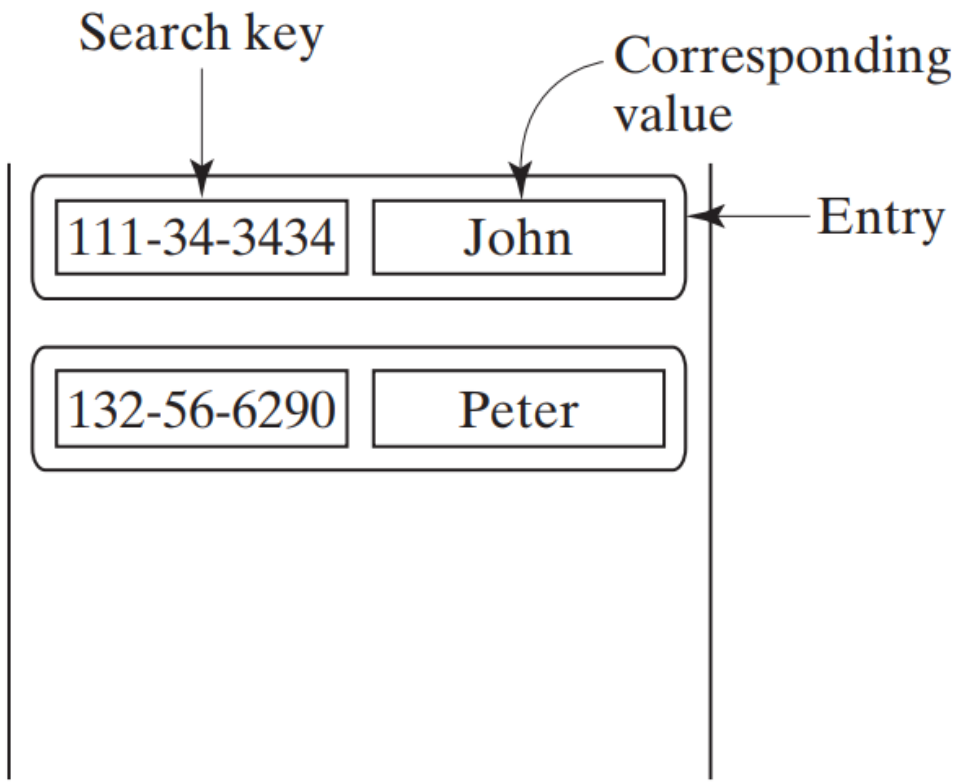
- A map is like a dictionary that provides a quick lookup to retrieve a value using a key.

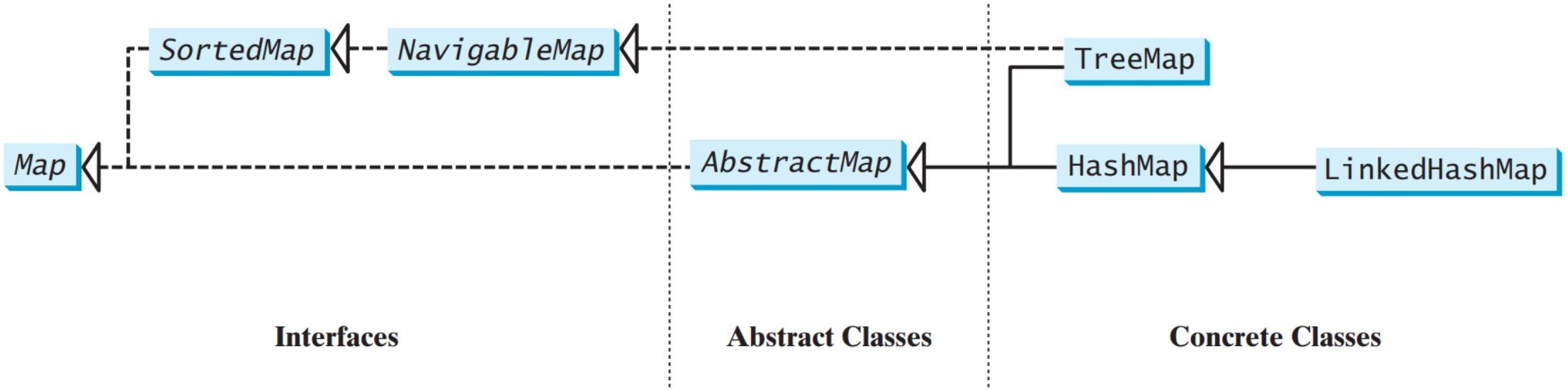
- You can create a map using one of its three concrete classes: **HashMap**, **LinkedHashMap**, or TreeMap.

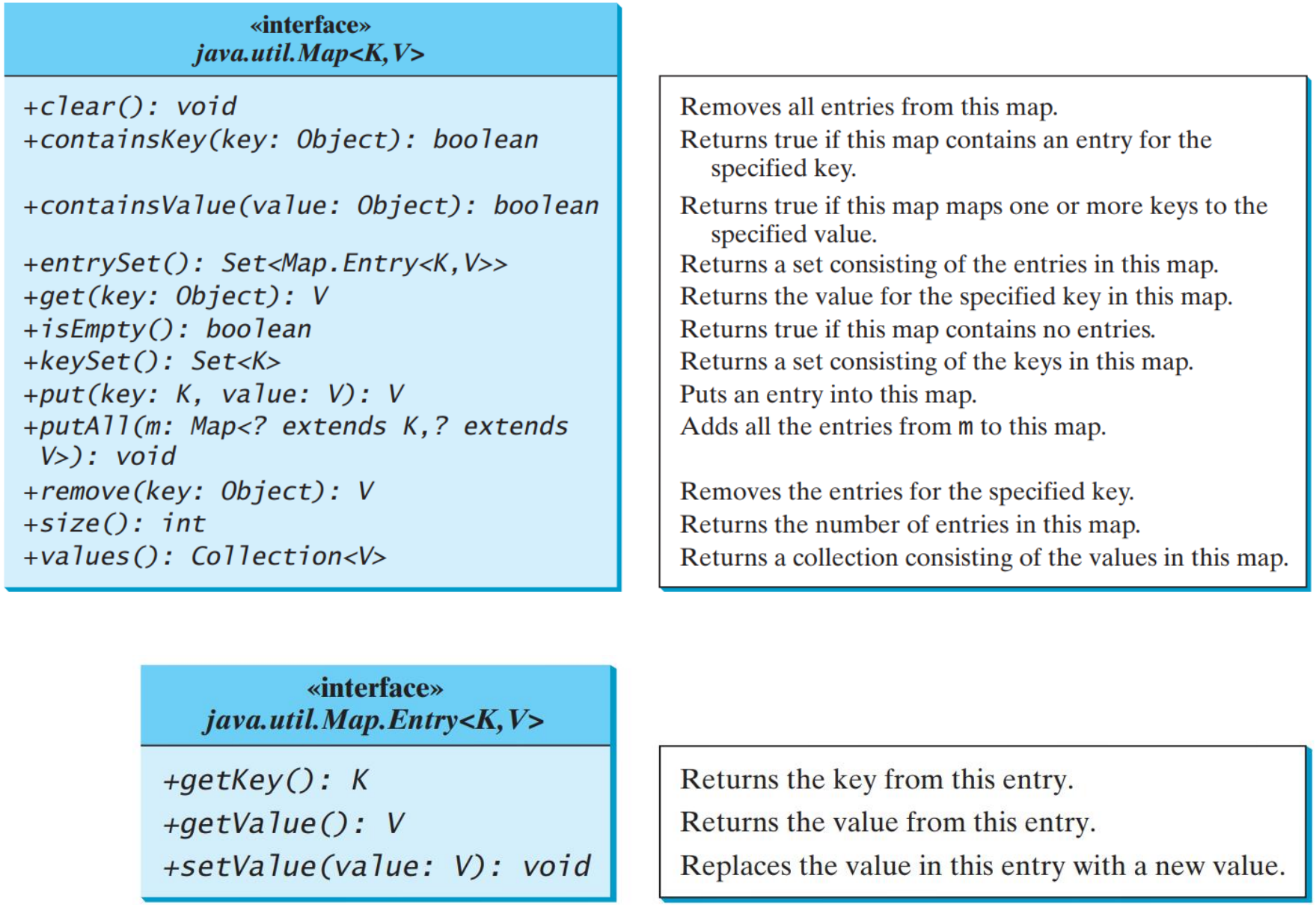
Map

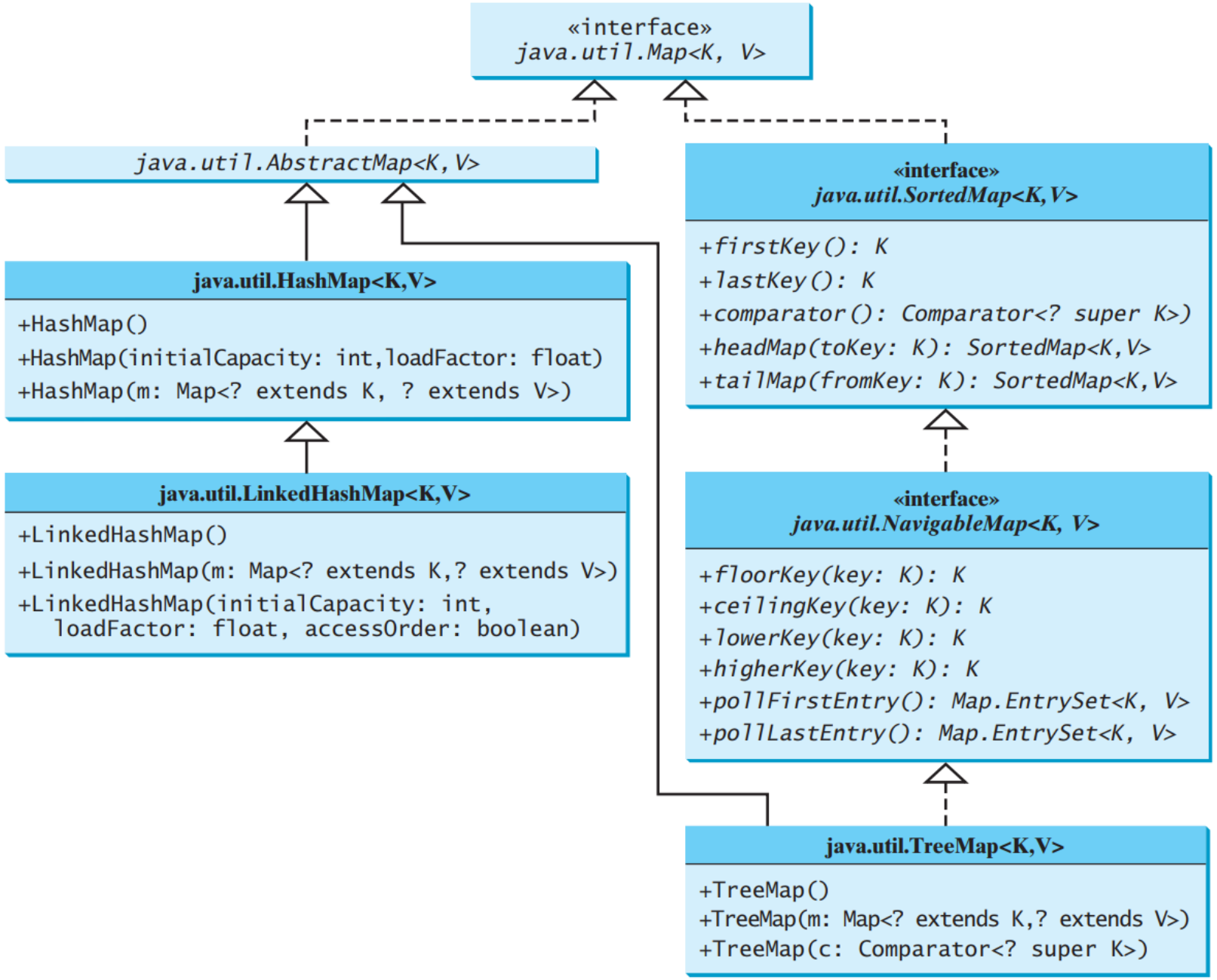
- A *map* is a container object that stores a collection of key/value pairs.
- It enables fast retrieval, deletion, and updating of the pair through the key.
- A map stores the values along with the keys. The keys are like indexes.
- In **List**, the indexes are integers. In **Map**, the keys can be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


Search key
Corresponding element value
A map
Entry
...


Search key
Corresponding value
Entry
111-34-3434
John
132-56-6290
Peter


SortedMap
NavigableMap
TreeMap
Map
AbstractMap
HashMap
LinkedHashMap
Interfaces
Abstract Classes
Concrete Classes


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


«interface»
java.util.Map<K, V>
java.util.AbstractMap<K,V>
«interface»
java.util.SortedMap<K,V>
+firstKey(): K
+lastKey(): K
+comparator(): Comparator<? super K>)
+headMap(toKey: K): SortedMap<K,V>
+tailMap(fromKey: K): SortedMap<K,V>
java.util.HashMap<K,V>
+HashMap()
+HashMap(initialCapacity: int,loadFactor: float)
+HashMap(m: Map<? extends K, ? extends V>)
«interface»
java.util.NavigableMap<K, V>
+floorKey(key: K): K
+ceilingKey(key: K): K
+lowerKey(key: K): K
+higherKey(key: K): K
+pollFirstEntry(): Map.EntrySet<K, V>
+pollLastEntry(): Map.EntrySet<K, V>
java.util.LinkedHashMap<K,V>
+LinkedHashMap()
+LinkedHashMap(m: Map<? extends K,? extends V>)
+LinkedHashMap(initialCapacity: int,
loadFactor: float, accessOrder: boolean)
java.util.TreeMap<K,V>
+TreeMap()
+TreeMap(m: Map<? extends K,? extends V>)
+TreeMap(c: Comparator<? super K>)

Map Concrete Classes

- The **HashMap** class is efficient for locating a value, inserting an entry, and deleting an entry.

- **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 in the order in which they were inserted into the map (known as the *insertion order*).

Examples

- TestMap.java
- CountOccurrenceOfWords.java