17 Java Collections Framework

Objectives

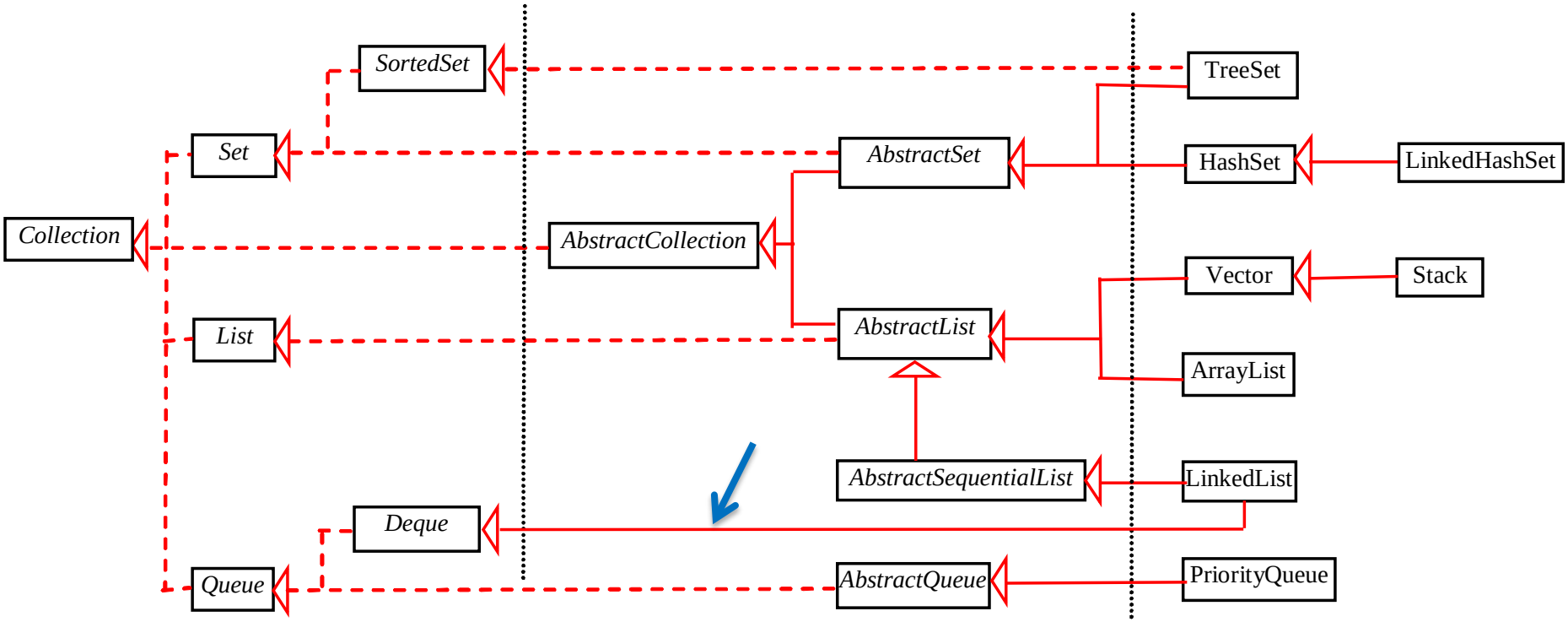
- To describe the Java Collections Framework hierarchy (§ § 22.1-22.2).
- To use the common methods defined in the Collection interface for operating sets and lists (§ 22.3).
- To use the Iterator interface to traverse a collection (§ 22.4).
- To use the for-each loop to simplify traversing a collection (§ 22.4).
- To explore how and when to use HashSet (§ 22.4.1), LinkedHashSet (§ 22.4.2), or TreeSet (§ 22.4.3) to store elements.
- To compare elements using the Comparable interface and the Comparator interface (§ 22.5).
- To explore how and when to use ArrayList or LinkedList to store elements (§ 22.6).
- To use the static utility methods in the Collections class for sorting, searching, shuffling lists, and finding the largest and smallest element in collections (§ 22.7).
- To compare performance of sets and lists (§ 22.8).
- To distinguish between Vector and ArrayList, and to use the Stack class for creating stacks (§ 22.9).
- To explore the relationships among Collection, Queue, LinkedList, and PriorityQueue and to create priority queues using the PriorityQueue class (§ 22.10).
- 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 (§ 22.11).
- To obtain singleton sets, lists, and maps, and unmodifiable sets, lists, and maps, using the static methods in the Collections class (§ 22.12).

Java Collection Framework hierarchy

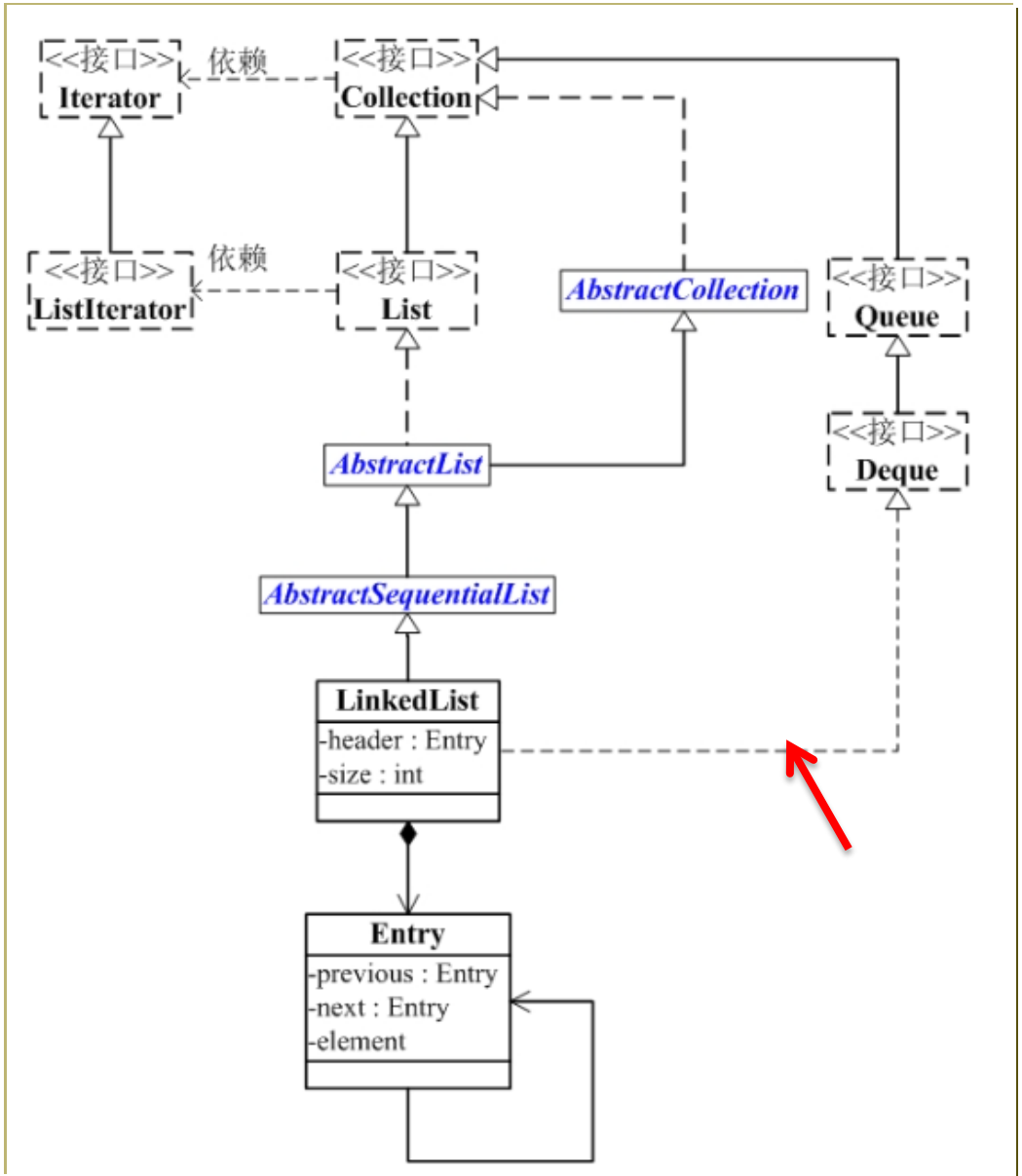
A *collection* is a container object that represents a group of objects, often referred to as *elements*. The Java Collections Framework supports three types of collections, named *sets, lists,* and *maps*.

Java Collection Framework hierarchy, cont.

Set and List are subinterfaces of Collection.

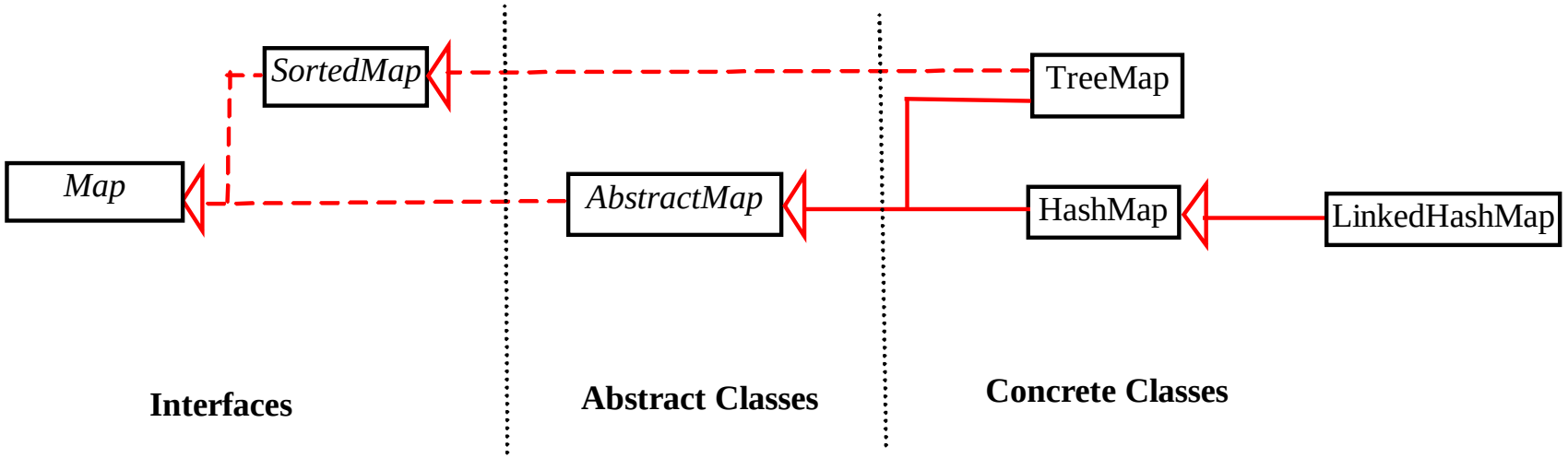


```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable
{
```



Java Collection Framework hierarchy, cont.

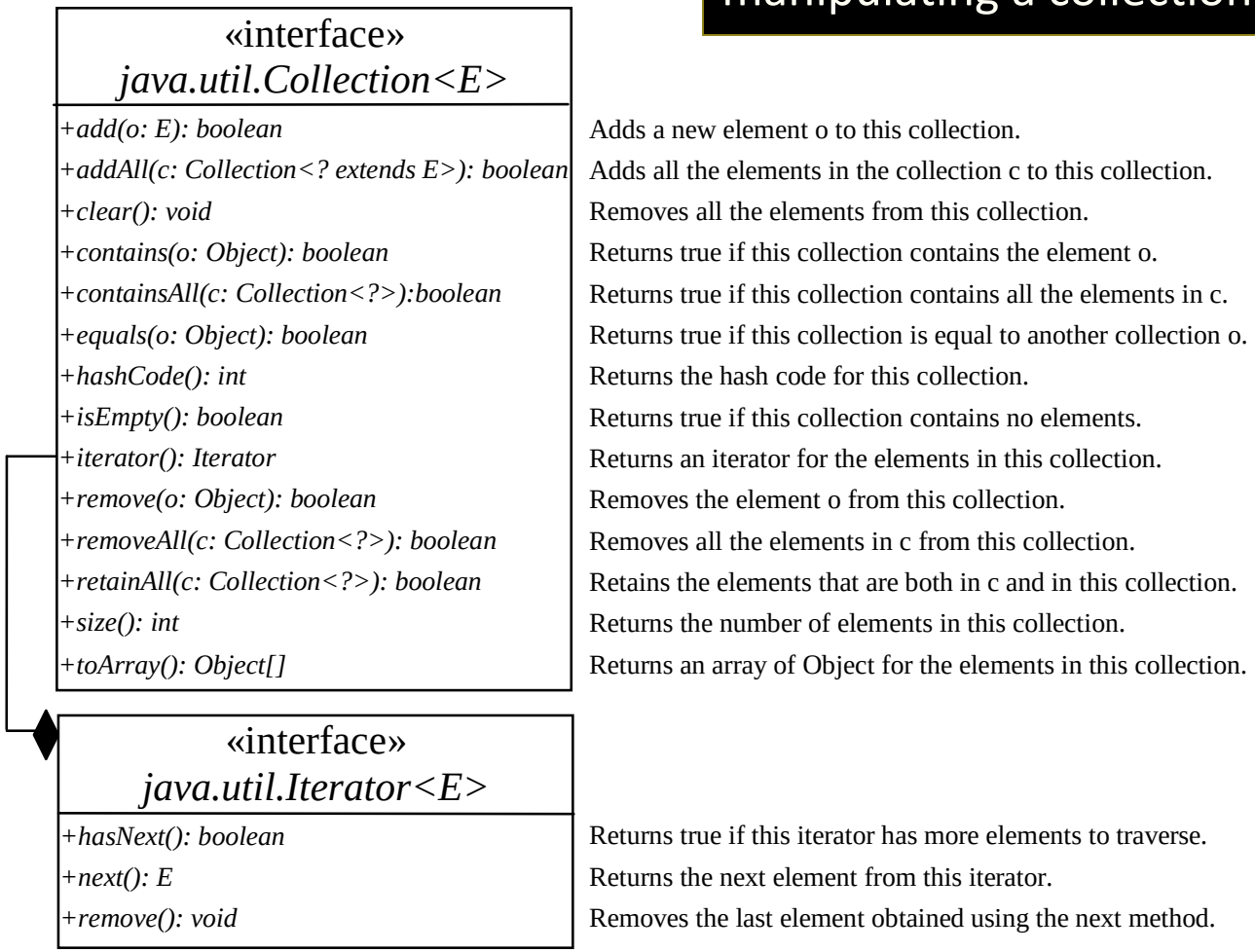
An instance of Map represents a group of objects, each of which is associated with a key. You can get the object from a map using a key, and you have to use a key to put the object into the map.




```
128  */
129 public interface Map<K,V> {
130     // Query Operations
131
```

The Collection Interface

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



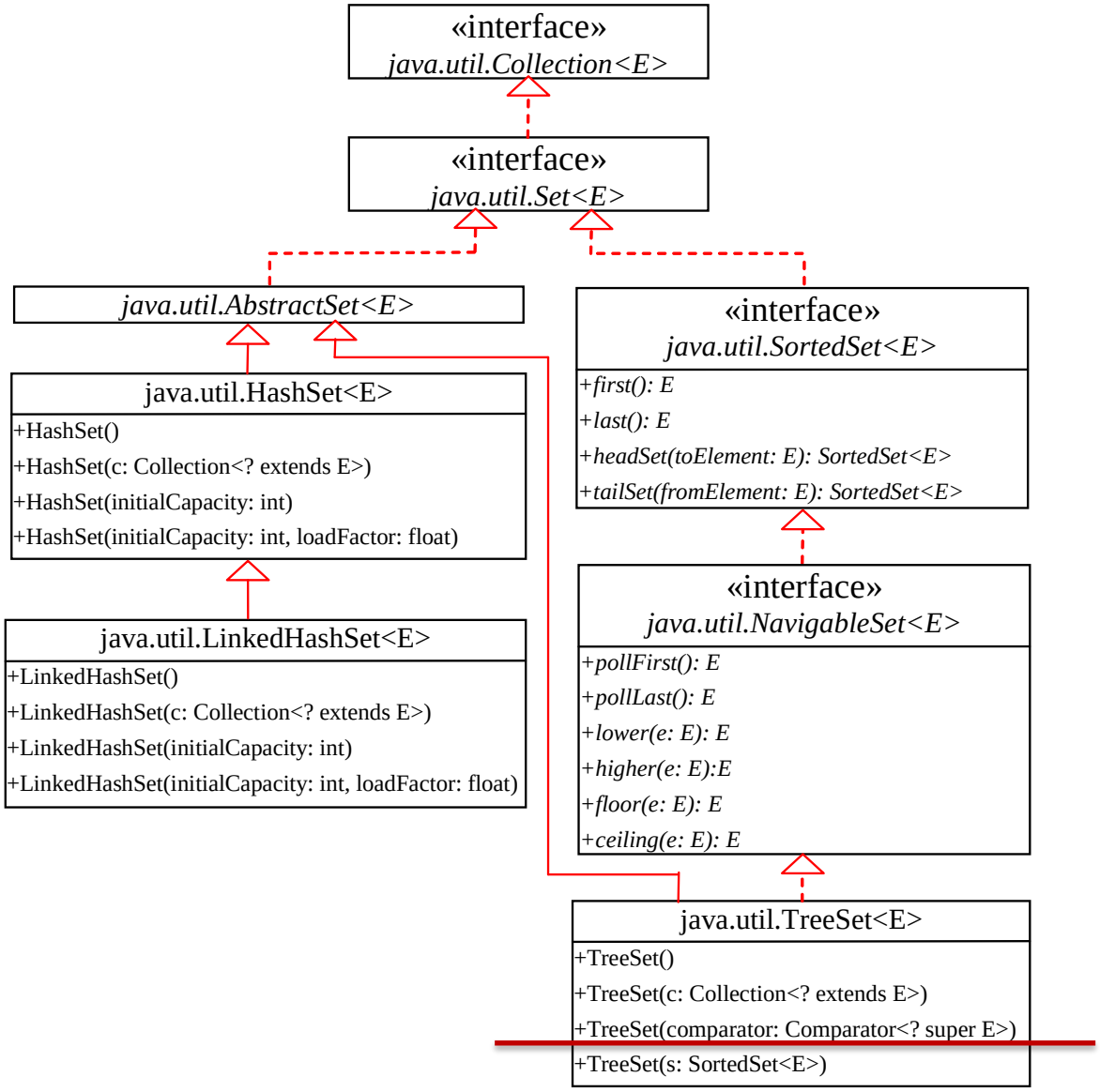
Adds a new element o to this collection.
Adds all the elements in the collection c to this collection.
Removes all the elements from this collection.
Returns true if this collection contains the element o.
Returns true if this collection contains all the elements in c.
Returns true if this collection is equal to another collection o.
Returns the hash code for this collection.
Returns true if this collection contains no elements.
Returns an iterator for the elements in this collection.
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. 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.

The Set Interface Hierarchy



The AbstractSet Class

The AbstractSet class is a convenience class that extends AbstractCollection and 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 code 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.

```
public boolean equals(Object o) {
    if (o == this)
        return true;

    if (!(o instanceof Set))
        return false;
    Collection<?> c = (Collection<?>) o;
    if (c.size() != size())
        return false;
    try {
        return containsAll(c);
    } catch (ClassCastException unused)   {
        return false;
    } catch (NullPointerException unused) {
        return false;
    }
}
```

```
public int hashCode() {
    int h = 0;
    Iterator<E> i = iterator();
    while (i.hasNext()) {
        E obj = i.next();
        if (obj != null)
            h += obj.hashCode();
    }
    return h;
}
```

The HashSet Class

The HashSet class is a concrete class that implements Set. It 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.

TestHashSet

Run

TestMethodsInCollection

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

    // 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() + " ");
    }
  }
}
```

Problems | Javadoc | Declaration | Console

```
<terminated> TestHashSet [Java Application] C:\Program Files\Java\jdk
[San Francisco, Beijing, New York, London, Paris]
SAN FRANCISCO BEIJING NEW YORK LONDON PARIS
```

顺序与插入顺序无关

```
// Create set1
java.util.Set<String> set1 = new java.util.HashSet<String>();

// 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<String>();

// 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);
```

Problems | Javadoc | Declaration | Console

<terminated> TestMethodsInCollection [Java Application] C:\Program Files\Java\jdk1.8.0_141\jre\bin\javaw.exe (2017年12月3日 下午3:18

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

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

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

Is Taipei in set2? false

After adding set2 to set1, set1 is [San Francisco, Beijing, New York, Shanghai, London, Paris]
After removing set2 from set1, set1 is [San Francisco, Beijing, New York]
After removing common elements in set2 from set1, set1 is []
```

```
/*
    可以使用retainAll方法：oldCourses.retainAll(newCoures)。
    如果存在相同元素，oldCourses中仅保留相同的元素。
    如果不存在相同元素，oldCourse会变为空。
 */
```

TIP: for-each loop

You can simplify the code in Lines 21-26 using a JDK 1.5 enhanced for loop without using an iterator, as follows:

```
for (Object element: set)
  System.out.print(element.toString() + " ");
```

Example: Using LinkedHashSet

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

LinkedHashSet用链表实现来扩展HashSet类，可以按照插入顺序提取；而HashSet是没有顺序的。

TestLinkedHashSet

Run

```java
import java.util.*;

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

    // 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() + " ");
  }
}
```

Problems | Javadoc | Declaration | Console

```
<terminated> TestLinkedHashSet [Java Application] C:\Program Files\Java\jdk1.8.0_141\jre\bin\javaw
[London, Paris, New York, San Francisco, Beijing]
london paris new york san francisco beijing
```

HashSet与LinkedHashSet

```
public class HashSet<E>
    extends AbstractSet<E>
    implements Set<E>, Cloneable, java.io.Serializable
{
    static final long serialVersionUID = -5024744406713321676L;

    private transient HashMap<E,Object> map;

    // Dummy value to associate with an Object in the backing Map
    private static final Object PRESENT = new Object();

    public HashSet() {
        map = new HashMap<>();
    }
    public HashSet(Collection<? extends E> c) {
        map = new HashMap<>(Math.max((int) (c.size()/.75f) + 1, 16));
        addAll(c);
    }

    public HashSet(int initialCapacity, float loadFactor) {
        map = new HashMap<>(initialCapacity, loadFactor);
    }
    public HashSet(int initialCapacity) {
        map = new HashMap<>(initialCapacity);
    }

    HashSet(int initialCapacity, float loadFactor, boolean dummy) {
        map = new LinkedHashMap<>(initialCapacity, loadFactor);
    }
```

```
@SuppressWarnings("unchecked")
public Object clone() {
    try {
        HashSet<E> newSet = (HashSet<E>) super.clone();
        newSet.map = (HashMap<E, Object>) map.clone();
        return newSet;
    } catch (CloneNotSupportedException e) {
        throw new InternalError(e);
    }
}

/**
 * Save the state of this <tt>HashSet</tt> instance t
 * serialize it).
 *
 * @serialData The capacity of the backing <tt>HashMa
 *             (int), and its load factor (float) are
 *             the size of the set (the number of ele
 *             (int), followed by all of its elements
 *             no particular order.
 */
private void writeObject(java.io.ObjectOutputStream s
    throws java.io.IOException {
    // Write out any hidden serialization magic
    s.defaultWriteObject();

    // Write out HashMap capacity and load factor
    s.writeInt(map.capacity());
    s.writeFloat(map.loadFactor());

    // Write out size
    s.writeInt(map.size());

    // Write out all elements in the proper order.
    for (E e : map.keySet())
        s.writeObject(e);
}
```

```
private void readObject(java.io.ObjectInputStream s)
    throws java.io.IOException, ClassNotFoundException {
    // Read in any hidden serialization magic
    s.defaultReadObject();

    // Read capacity and verify non-negative.
    int capacity = s.readInt();
    if (capacity < 0) {
        throw new InvalidObjectException("Illegal capacity: " +
                                         capacity);
    }

    // Read load factor and verify positive and non NaN.
    float loadFactor = s.readFloat();
    if (loadFactor <= 0 || Float.isNaN(loadFactor)) {
        throw new InvalidObjectException("Illegal load factor: " +
                                         loadFactor);
    }

    // Read size and verify non-negative.
    int size = s.readInt();
    if (size < 0) {
        throw new InvalidObjectException("Illegal size: " +
                                         size);
    }

    // Set the capacity according to the size and load factor ensurin
    // the HashMap is at least 25% full but clamping to maximum capac
    capacity = (int) Math.min(size * Math.min(1 / loadFactor, 4.0f),
            HashMap.MAXIMUM_CAPACITY);

    // Create backing HashMap
    map = (((HashSet<?>)this) instanceof LinkedHashSet ?
           new LinkedHashMap<E,Object>(capacity, loadFactor) :
           new HashMap<E,Object>(capacity, loadFactor));

    // Read in all elements in the proper order.
    for (int i=0; i<size; i++) {
        @SuppressWarnings("unchecked")
            E e = (E) s.readObject();
        map.put(e, PRESENT);
    }
}
```

HashSet与LinkedHashSet

```
public class HashSet<E>
    extends AbstractSet<E>
    implements Set<E>, Cloneable, java.io.Serializable
{
    static final long serialVersionUID = -5024744406713321676

    private transient HashMap<E,Object> map;

    // Dummy value to associate with an Object in the backing M
    private static final Object PRESENT = new Object();

    public HashSet() {
        map = new HashMap<>();
    }
    public HashSet(Collection<? extends E> c) {
        map = new HashMap<>(Math.max((int) (c.size()/.75f) + 1, 16));
        addAll(c);
    }

    public HashSet(int initialCapacity, float loadFactor) {
        map = new HashMap<>(initialCapacity, loadFactor);
    }
    public HashSet(int initialCapacity) {
        map = new HashMap<>(initialCapacity);
    }
```

底层由HashMap实现

```
public boolean add(E e) {
    return map.put(e, PRESENT)==null;
}
```

专门给LinkedHashSet

```
    HashSet(int initialCapacity, float loadFactor, boolean dummy) {
        map = new LinkedHashMap<>(initialCapacity, loadFactor);
    }
```

HashSet与LinkedHashSet

```
public class LinkedHashSet<E>
    extends HashSet<E>
    implements Set<E>, Cloneable, java.io.Serializable {

    private static final long serialVersionUID = -2851667679971038690L;
```

```
    public LinkedHashSet(int initialCapacity, float loadFactor) {
        super(initialCapacity, loadFactor, true);
    }
    public LinkedHashSet(int initialCapacity) {
        super(initialCapacity, .75f, true);
    }
    public LinkedHashSet() {
        super(16, .75f, true);
    }
    public LinkedHashSet(Collection<? extends E> c) {
        super(Math.max(2*c.size(), 11), .75f, true);
        addAll(c);
    }
```

```
    HashSet(int initialCapacity, float loadFactor, boolean dummy) {
        map = new LinkedHashMap<>(initialCapacity, loadFactor);
    }
```

在父类 HashSet 中，专为 LinkedHashSet 提供了构造方法，该方法为包访问权限，并未对外公开。
由上述源代码可见，LinkedHashSet 通过继承 HashSet，底层使用 LinkedHashMap，以很简单明了的方式来实现了其自身的所有功能。

HashSet与LinkedHashSet

```
/**
 * The table, initialized on first use, and resized as
 * necessary. When allocated, length is always a power of two.
 * (We also tolerate length zero in some operations to allow
 * bootstrapping mechanics that are currently not needed.)
 */
transient Node<K,V>[] table;
```

HashMap

```
public class LinkedHashMap<K,V>
    extends HashMap<K,V>
    implements Map<K,V>
{
    /**
     * The head (eldest) of the doubly linked list.
     */
    transient LinkedHashMap.Entry<K,V> head;

    /**
     * The tail (youngest) of the doubly linked list.
     */
    transient LinkedHashMap.Entry<K,V> tail;
```

LinkedHashMap,额外用双向链表来维护顺序

LinkedHashSet是HashSet的一个子类,也是用HashCode值来决定元素存储位置,但是LinkedHashSet同时用链表来维护元素的次序(元素的顺序总是与添加的顺序一致)。当遍历LinkedHashSet的时候,
LinkedHashSet就会按元素的添加顺序来访问集合里的元素。

因为LinkedHashSet要通过链表来维护集合元素的顺序,所以性能要比HashSet差。

The SortedSet Interface and the TreeSet Class

SortedSet is a subinterface of Set, which guarantees that the elements in the set are sorted. TreeSet is a concrete class that implements the SortedSet interface. You can use an iterator to traverse the elements in the sorted order. The elements can be sorted in two ways.

SortedSet

Public Methods	
abstract Comparator<? super E>	comparator() Returns the comparator used to compare elements in this SortedSet. 返回与此有序集合关联的比较器，如果使用元素的自然顺序，则返回 null。
abstract E	first() Returns the first element in this SortedSet. 返回此有序集合中当前第一个（最小的）元素。
abstract SortedSet<E>	headSet(E end) Returns a SortedSet of the specified portion of this SortedSet which contains elements less than the end element. 用一个SortedSet, 返回此有序集合中小于end的所有元素。
abstract E	last() Returns the last element in this SortedSet. 返回此有序集合中最后一个（最大的）元素
abstract SortedSet<E>	subSet(E start, E end) Returns a SortedSet of the specified portion of this SortedSet which contains elements greater or equal to the start element less than the end element. 返回此有序集合的部分元素，元素范围从 fromElement（包括）到 toElement（不包括）。
abstract SortedSet<E>	tailSet(E start) Returns a SortedSet of the specified portion of this SortedSet which contains elements greater or equal to the start element. 返回此有序集合的部分元素，其元素大于或等于 fromElement。

- SortedSet时Set的一个子接口。first()和last()返回集合中的第一个和最后一个元素；headSet(toElement)和tailSet(fromElement)返回集合中元素小于toElement和大于fromElement的那部分。
- NavigableSet扩展了SortedSet，提供导航方法lower(e), floor(e), ceiling(e)和higher(e)，分别返回小于、小于或等于、大于或等于、大于一个元素的元素，若没这样的元素，则返回null。
- pollFirst()和pollLast()则分别删除和返回TreeSet中的第一个和最后一个元素。

The SortedSet Interface and the TreeSet Class, cont.

One way is to use the Comparable interface.

The other way 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. This approach is referred to as *order by comparator*.

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 using the compareTo method in the Comparable interface.

The example also creates a tree set of geometric objects. The geometric objects are sorted using the compare method in the Comparator interface.

TestTreeSet

Run

TestTreeSetWithComparator

```
// Create a hash set
Set<String> set = new HashSet<String>();

// 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<String>(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);
}
```

```
<terminated> TestTreeSet [Java Application] C:\Program Files\Java\jdk1.8.0_141\jre\bin\javaw
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]
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]
```

The Comparator Interface

Sometimes you want to insert elements of different types into a tree set. The elements may not be instances of Comparable or are not comparable. You can define a comparator to compare these elements. To do so, create a class that implements the java.util.Comparator interface. The Comparator interface has two methods, compare and equals.

The Comparator Interface

public int compare(Object element1, Object element2)

Returns a negative value if element1 is less than element2, a positive value if element1 is greater than element2, and zero if they are equal.

public boolean equals(Object element)

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

GeometricObjectComparator

```java
import java.util.Comparator;

public class GeometricObjectComparator
    implements Comparator<GeometricObject>, java.io.Serializable {
  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;
  }
}
```

Example: The Using Comparator to Sort Elements in a Set

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.

TestTreeSetWithComparator

Run

```
public class TestTreeSetWithComparator {
  public static void main(String[] args) {
    // Create a tree set for geometric objects using a comparator
    Set<GeometricObject> set =
      new TreeSet<GeometricObject>(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());
  }
}
```

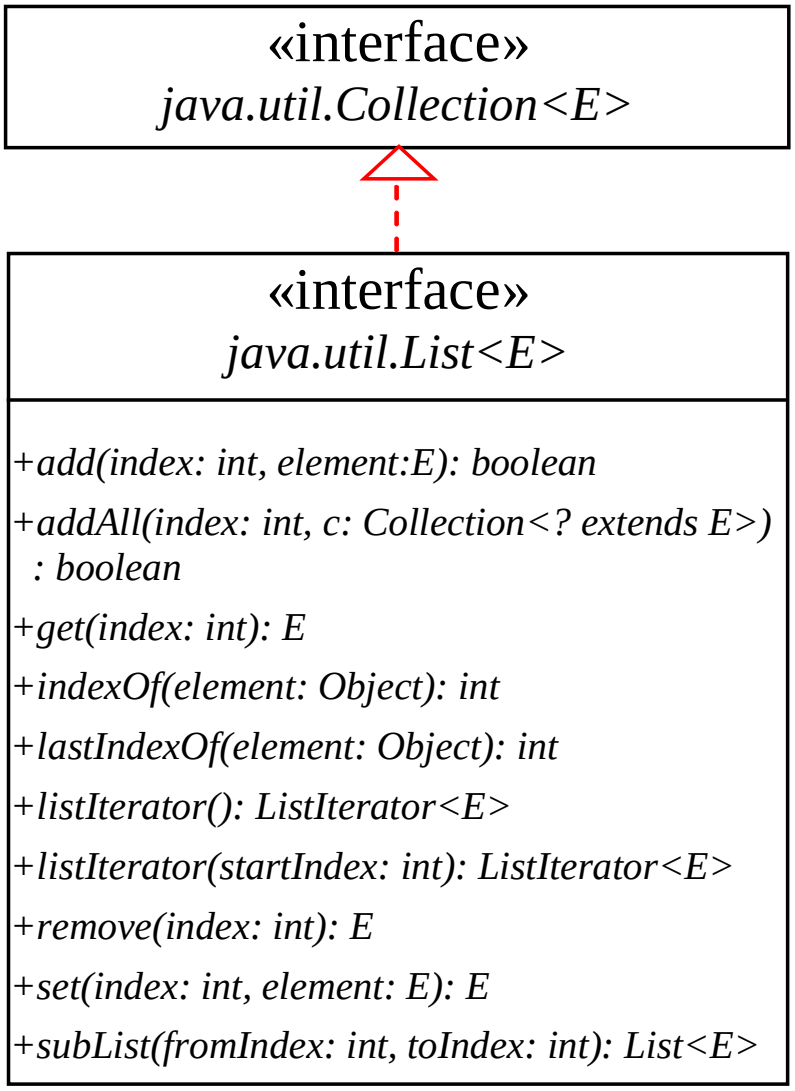
HashSet, LinkedHashSet, TreeSet

- **HashSet**
  - 不能保证元素的排列顺序，顺序有可能发生变化
  - 集合元素可以是null,但只能放入一个null
- **LinkedHashSet**
  - LinkedHashSet集合同样是根据元素的hashCode值来决定元素的存储位置，但是它同时使用链表维护元素的次序。这样使得元素看起 来像是以插入顺序保存的，也就是说，当遍历该集合时候，LinkedHashSet将会以元素的添加顺序访问集合的元素。LinkedHashSet在迭代访问Set中的全部元素时，性能比HashSet好，但是插入时性能稍微逊色于HashSet。
- **TreeSet**
  - TreeSet是SortedSet接口的唯一实现类，TreeSet可以确保集合元素处于排序状态。TreeSet支持两种排序方式，自然排序 和定制排序

The List Interface

A set stores non-duplicate elements. To allow duplicate elements to be stored in a collection, you need to use a list. A list can not only store duplicate elements, but can also allow the user to specify where the element is stored. The user can access the element by index.

The List Interface, cont.



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.

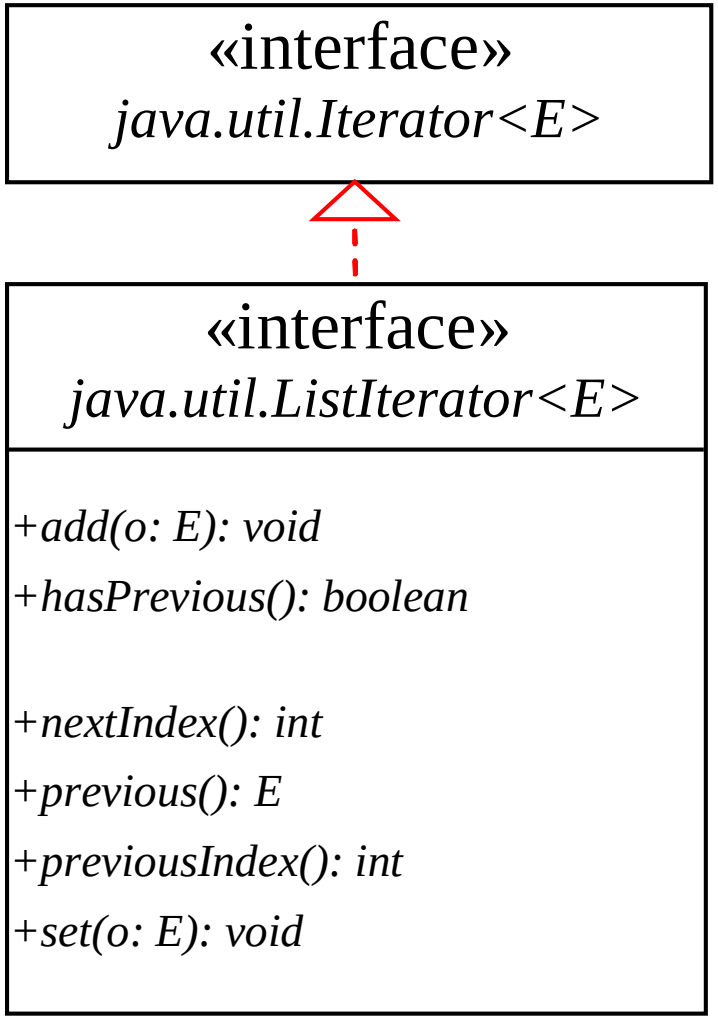
Returns the iterator for the elements from startIndex.

Removes the element at the specified index.

Sets the element at the specified index.

Returns a sublist from fromIndex to toIndex.

The List Iterator



Adds the specified object to the list.

Returns true if this list iterator has more elements when traversing backward.

Returns the index of the next element.

Returns the previous element in this list iterator.

Returns the index of the previous element.

Replaces the last element returned by the previous or next method with the specified element.

ArrayList and LinkedList

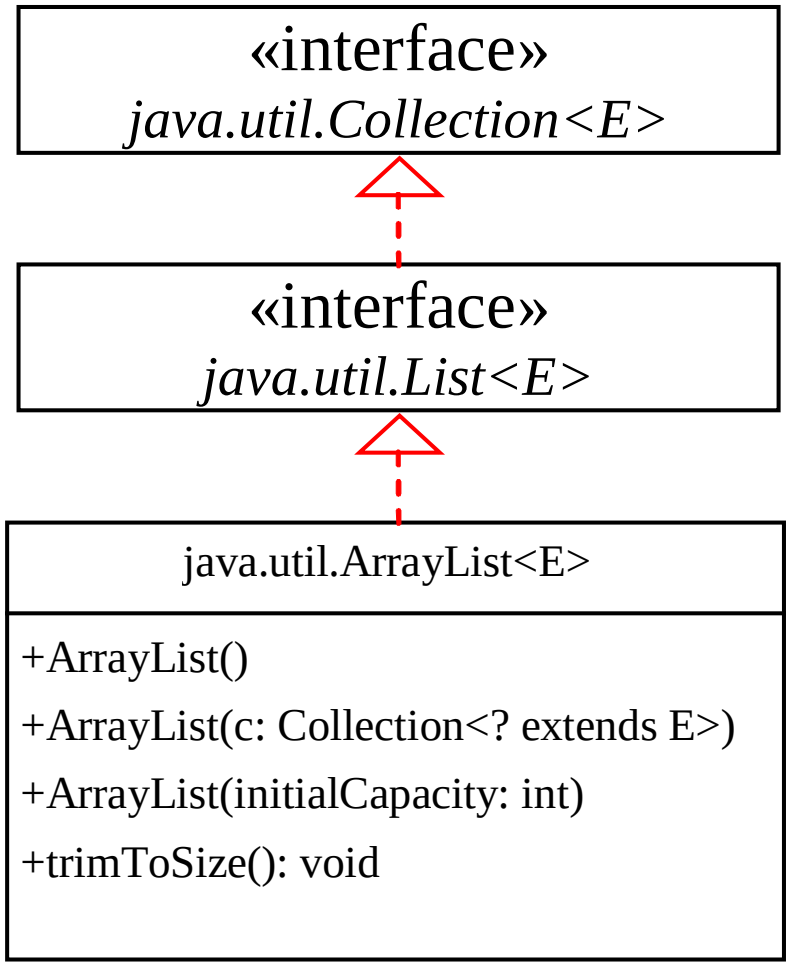
The ArrayList class and the LinkedList class are concrete implementations of the List interface. Which of the two classes you use depends on your specific needs.

If you need to support random access through an index without inserting or removing elements from any place other than the end, ArrayList offers the most efficient collection.

If, however, your application requires the insertion or deletion of elements from any place in the list, you should choose LinkedList.

A list can grow or shrink dynamically. An array is fixed once it is created. If your application does not require insertion or deletion of elements, the most efficient data structure is the array.

java.util.ArrayList



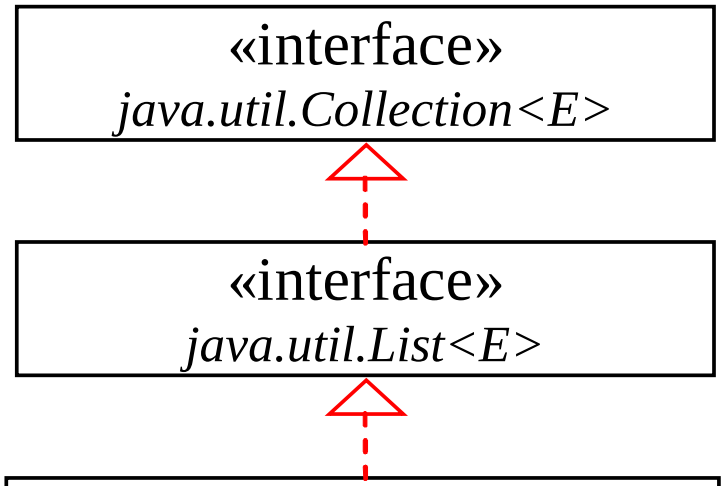
Creates an empty list with the default initial capacity.

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.

java.util.LinkedList



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.

Example: Using ArrayList and LinkedList

This example creates an array list filled with numbers, and inserts new elements into the specified location in the list. The example also creates a linked list from the array list, inserts and removes the elements from the list. Finally, the example traverses the list forward and backward.

TestArrayAndLinkedList

Run

```java
public class TestArrayAndLinkedList {
  public static void main(String[] args) {
    List<Integer> arrayList = new ArrayList<Integer>();
    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);

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

    System.out.println("Display the linked list backward:");
    for (int i = linkedList.size() - 1; i >= 0; i--) {
      System.out.print(linkedList.get(i) + " ");
    }
  }
}
```

hashCode() and equals() used in Collections Framework

- 在Java集合中有两类，一类是List，一类是Set。他们之间的区别就在于List集合中的元素是有序的，且可以重复，而Set集合中元素是无序不可重复的。
- 对于Set而言我们要如何来保证元素不重复呢？通过迭代来equals()是否相等。数据量小还可以接受，当我们的数据量大的时候效率可想而知（当然我们可以利用算法进行优化）。比如我们向HashSet插入1000数据，难道我们真的要迭代1000次，调用1000次equals()方法吗？

hashCode() and equals() used in Collections Framework

- hashCode提供了解决方案，看Object中hashCode的源代码：

```
public native int hashCode();
```

- 它是一个本地方法，它的实现与本地机器有关，这里我们暂且认为他返回的是对象存储的物理位置（实际上不是，这里写是便于理解）。
- 当我们向一个集合中添加某个元素，集合会首先调用hashCode方法，这样就可以直接定位它所存储的位置，若该处没有其他元素，则直接保存。若该处已经有元素存在，就调用equals方法来匹配这两个元素是否相同，相同则不存，不同则散列到其他位置。这样处理，当我们存入大量元素时就可以大大减少调用equals()方法的次数，极大地提高了效率。

hashCode() and equals() used in Collections Framework

- hashCode重要么？不重要，对于List集合、数组而言，就是一个累赘，但是对于HashMap、HashSet、HashTable而言，它变得异常重要。
- 所以在使用HashMap、HashSet、HashTable时一定要注意hashCode。对于一个对象而言，其hashCode过程就是一个简单的Hash算法的实现，其实现过程对你实现对象的存取过程起到非常重要的作用。

如何设计hashCode

- 一个对象势必会存在若干个属性，如何选择属性来进行散列考验着一个人的设计能力。
- 如果我们将所有属性进行散列，这必定会是一个糟糕的设计，因为对象的hashCode方法无时无刻不是在被调用，如果太多的属性参与散列，那么需要的操作数时间将会大大增加，这将严重影响程序的性能。
- 但是如果较少属相参与散列，散列的多样性会削弱，会产生大量的散列“冲突”，除了不能够很好的利用空间外，在某种程度也会影响对象的查询效率。其实这两者是一个矛盾体，散列的多样性会带来性能的降低。

hashCode() and equals() used in Collections Framework

- 在Java中hashCode的实现总是伴随着equals，他们是紧密配合的，你要是自己设计了其中一个，就要设计另外一个。当然在多数情况下，这两个方法是不用我们考虑的，直接使用默认方法就可以帮助我们解决很多问题。但是在有些情况，我们必须要自己动手来实现它，才能确保程序更好的运作。

hashCode() and equals() used in Collections Framework

- 对于equals，我们必须遵循如下规则：
  - 对称性：如果x.equals(y)返回是“true”，那么y.equals(x)也应该返回是“true”。
  - 反射性：x.equals(x)必须返回是“true”。
  - 类推性：如果x.equals(y)返回是“true”，而且y.equals(z)返回是“true”，那么z.equals(x)也应该返回是“true”。
  - 一致性：如果x.equals(y)返回是“true”，只要x和y内容一直不变，不管你重复x.equals(y)多少次，返回都是“true”。
  - 任何情况下，x.equals(null)，永远返回是“false”；x.equals(和x不同类型的对象)永远返回是“false”。

hashCode() and equals() used in Collections Framework

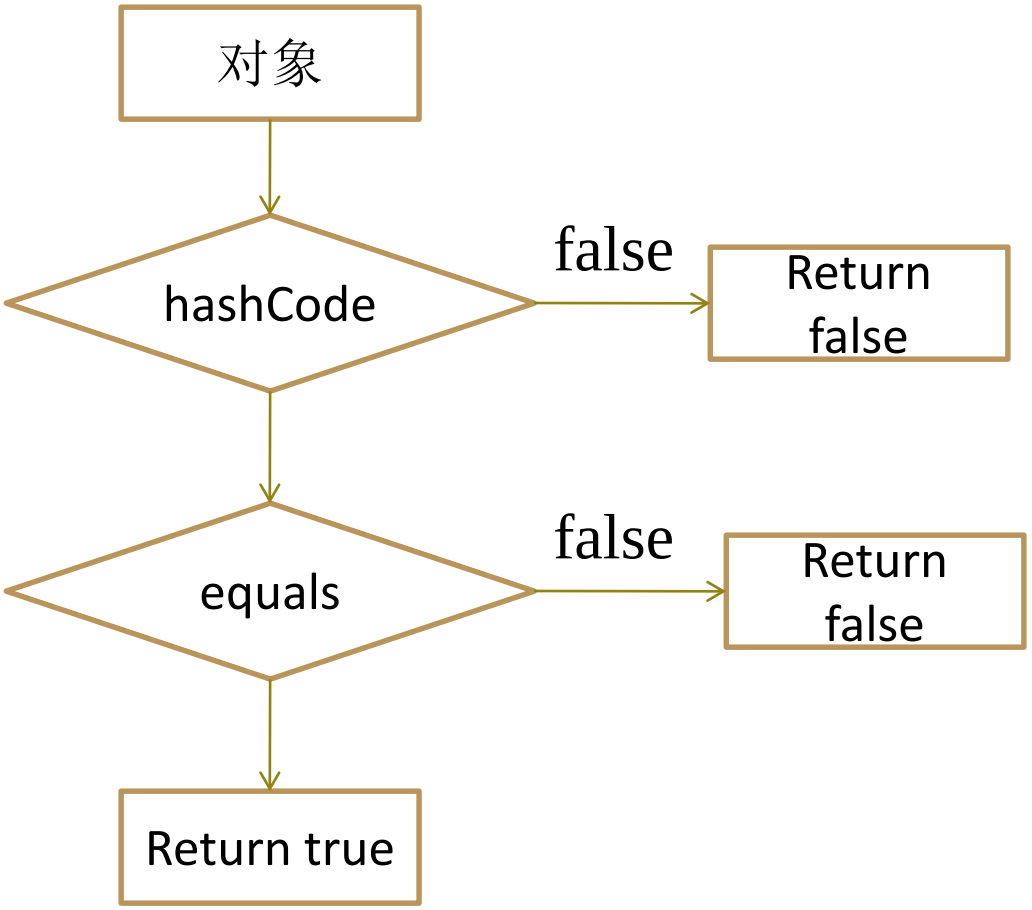
- 对于hashCode，我们应该遵循如下规则：
  - 1. 在一个应用程序执行期间，如果一个对象的equals方法做比较所用到的信息没有被修改的话，则对该对象调用hashCode方法多次，它必须始终如一地返回同一个整数。
  - 2. 如果两个对象根据equals(Object o)方法是相等的，则调用这两个对象中任一对象的hashCode方法必须产生相同的整数结果。
  - 3. 如果两个对象根据equals(Object o)方法是不相等的，则调用这两个对象中任一个对象的hashCode方法，不要求产生不同的整数结果。但如果能不同，则可能提高散列表的性能。

hashCode() and equals() used in Collections Framework

- 至于两者之间的关联关系，我们只需要记住如下即可：
- 如果x.equals(y)返回“true”，那么x和y的hashCode()必须相等。
- 如果x.equals(y)返回“false”，那么x和y的hashCode()有可能相等，也有可能不等。

hashCode() and equals() used in Collections Framework

- 将对象插入Set等时，其流程为：



1、判断两个对象的hashcode是否相等，若不等，则认为两个对象不等，完毕，若相等，则比较equals。
2、若两个对象的equals不等，则可以认为两个对象不等，否则认为他们相等。

```java
public class Person {
    private int age;
    private int sex;    //0: 男, 1: 女
    private String name;
    private final int PRIME = 37;
    Person(int age ,int sex ,String name){
        this.age = age;
        this.sex = sex;
        this.name = name;
    }
   /** 省略getter、setter方法 **/
      @Override
    public int hashCode() {
        System.out.println("调用hashCode方法.....");
       int hashResult = 1;
       hashResult = (hashResult + Integer.valueOf(age).hashCode() + Integer.valueO
        hashResult = PRIME * hashResult + ((name == null) ? 0 : name.hashCode());
         System.out.println("name:"+name +" hashCode:" + hashResult);
               return hashResult;
   }
  /**
    * 重写hashCode()
     */
    public boolean equals(Object obj) {
        System.out.println("调用equals方法.....");
               if(obj == null){
           return false;
       }
       if(obj.getClass() != this.getClass()){
            return false;
        }
       if(this == obj){
           return true;
        }
       Person person = (Person) obj;
                if(getAge() != person.getAge() || getSex()!= person.getSex()){
            return false;
        }
            if(getName() != null){
           if(!getName().equals(person.getName())){
                return false;
           }
     }
     else if(person != null){
          return false;
      }
    return true;
  }
}
```

```java
public class Main extends JPanel {
    public static void main(String[] args) {
        Set<Person> set = new HashSet<Person>();
                Person p1 = new Person(11, 1, "张三");
       Person p2 = new Person(12, 1, "李四");
        Person p3 = new Person(11, 1, "张三");
        Person p4 = new Person(11, 1, "李四");
                //只验证p1、p3
        System.out.println("p1 == p3? :" + (p1 == p3));
        System.out.println("p1.equals(p3)?:"+p1.equals(p3));
        System.out.println("----------------------分割线----------------
        set.add(p1);
        set.add(p2);
       set.add(p3);
       set.add(p4);        System.out.println("set.size()="+set.size());
    }
}
```

```
<terminated> Main [Java Application] D:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (2014年4月
p1 == p3? :false
调用equals方法.....
p1.equals(p3)?:true
----------------------分割线-------------------------
调用hashCode方法.....
name:张三 hashCode:792686
调用hashCode方法.....
name:李四 hashCode:861227
调用hashCode方法.....
name:张三 hashCode:792686
调用equals方法.....
调用hashCode方法.....
name:李四 hashCode:859858
set.size()=3
```

两次hashCode值一样，调用equals方法进行匹配

hashCode和equals方法都返回true，则不添加

The Collections Class

The Collections class contains various static methods for **operating** on collections and maps, for creating **synchronized** collection classes, and for creating **read-only** collection classes.

The Collections Class

```
List<String>list = Arrays.asList(“red”,”green”,”blue”);
Collections.sort(list);
Collections.sort(list,Collections.reverseOrder());

List<Integer> list1 =  Arrays.asList(2,4,7,10,11,45,50,59,60,66);
Collections.binarySearch(list1,7);  //2
Collections.binarySearch(list1,9); //-4

List<String>list2 = Arrays.asList(“blue”,”green”,”red”);
Collections.binarySearch(list2,”red”); //2
Collections.binarySearch(list2, “cyan”);//-2
```

The Collections Class

List<String>list = Arrays.asList(“yellow”, “red”,”green”,”blue”);

Collections.reverse(list);

Collections.suffle(list);

List<String>list1 = Arrays.asList(“yellow”, “red”,”green”,”blue”);

List<String>list2 = Arrays.asList(“yellow”, “red”,”green”,”blue”);

Collections.suffle(list1, new Random(20));

Collections.suffle(list2, new Random(20));

List<String>list1 = Arrays.asList(“yellow”, “red”,”green”,”blue”);

List<String>list2 = Arrays.asList(“white”,”black”);

Collections.copy(list1,list2); //浅复制，只复制表中元素的引用。List2中拷贝到list1中

//list1 = [white, black, green, blue]

The Collections Class

List<String> list = Arrays.asList(“red”,”green”,”blue”);

Collections.fill(list,”black”);

Collections.max(list);

Collections.min(list);

Collection<String>collection1 = Arrays.asList(“red”,”cyan”);

Collection<String>collection2 = Arrays.asList(“red”,”blue”);

Collection<String>collection3 = Arrays.asList(“pink”,”tan”);

Collections.disjoint(collection1, collection2); //两个集合没有相同的元素，则返回true. 这里返回false

Collections.disjoint(collection1, collection3); //返回true

Collections.frequency(collection1, “red”); //1

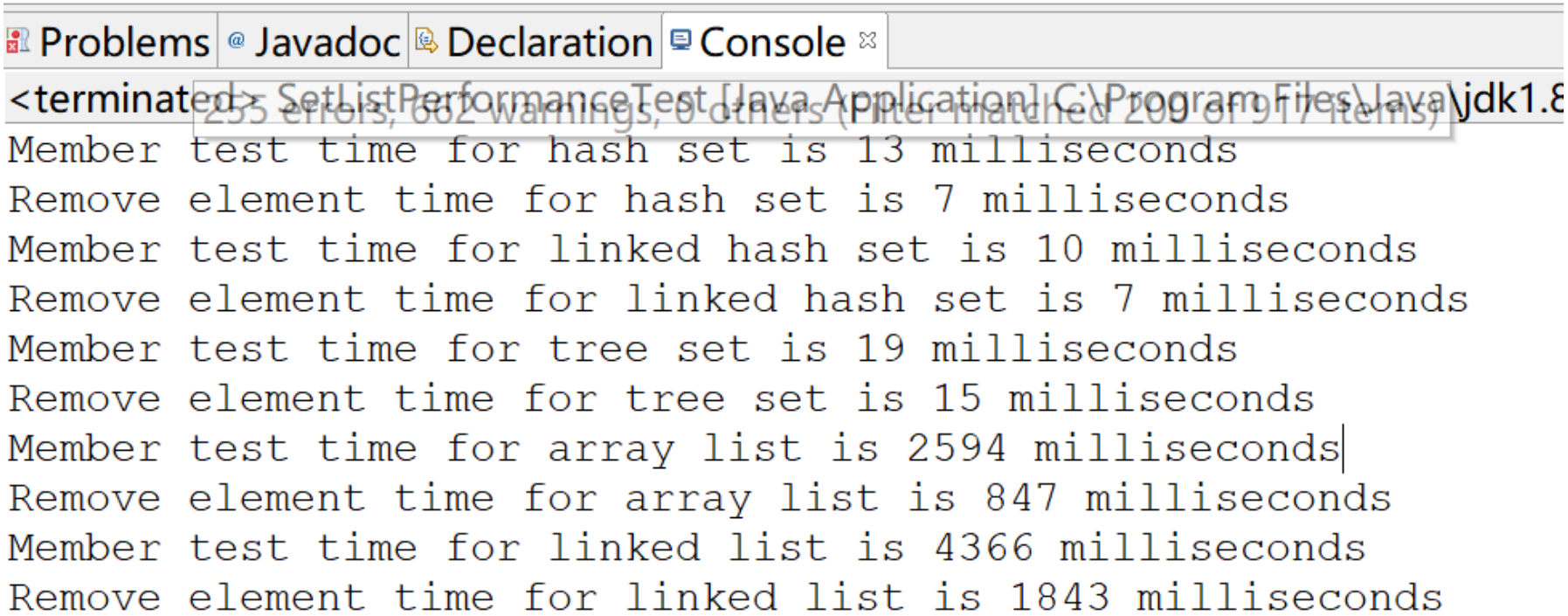
Performance of Sets and Lists

SetListPerformanceTest

Run

```
List<Integer> list = new ArrayList<Integer>();
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<Integer>(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<Integer>(list);
System.out.println("Member test time for linked hash se
  getTestTime(set2) + " milliseconds");
System.out.println("Remove element time for linked hash
  + getRemoveTime(set2) + " milliseconds");

// Create a tree set, and test its performance
Collection<Integer> set3 = new TreeSet<Integer>(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<Integer>(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<Integer>(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;
}
```


Problems
Javadoc
Declaration
Console
<terminated> SetListPerformanceTest [Java Application] C:\Program Files\Java\jdk1.8
Member test time for hash set is 13 milliseconds
Remove element time for hash set is 7 milliseconds
Member test time for linked hash set is 10 milliseconds
Remove element time for linked hash set is 7 milliseconds
Member test time for tree set is 19 milliseconds
Remove element time for tree set is 15 milliseconds
Member test time for array list is 2594 milliseconds
Remove element time for array list is 847 milliseconds
Member test time for linked list is 4366 milliseconds
Remove element time for linked list is 1843 milliseconds

```java
public class SetList {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Set<Integer> set = new TreeSet<Integer>();
        List<Integer> list = new ArrayList<Integer>();

        for(int i=-3;i<3;i++){
            set.add(i);
            list.add(i);
        }
        for(int i=0;i<3;i++){
            set.remove(i);
            list.remove(i);
        }
        System.out.println(set + " " + list);
    }

}
```

[-3, -2, -1] [-2, 0, 2]

boolean java.util.Set.remove(Object o)

Removes the specified element from this set if it is present (optional operation). More formally, removes an element e such that `(o==null ? e==null : o.equals(e))`, if this set contains such an element. Returns `true` if this set contained the element (or equivalently, if this set changed as a result of the call). (This set will not contain the element once the call returns.)

Specified by: remove(...) in Collection

Parameters:

o object to be removed from this set, if present

Returns:

`true` if this set contained the specified element

Integer java.util.List.remove(int index)

Removes the element at the specified position in this list (optional operation). Shifts any subsequent elements to the left (subtracts one from their indices). Returns the element that was removed from the list.

Parameters:

index the index of the element to be removed

Returns:

the element previously at the specified position

Throws:

UnsupportedOperationException - if the `remove` operation is not supported by this list

List

0	1	2	3	4	5
-3	-2	-1	0	1	2

```java
public class SetList {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Set<Integer> set = new TreeSet<Integer>();
        List<Integer> list = new ArrayList<Integer>();

        for(int i=-3;i<3;i++){
            set.add(i);
            list.add(i);
        }
        for(int i=0;i<3;i++){
            set.remove(i);
//          list.remove(i);
            list.remove((Integer)i);
        }
        System.out.println(set + " " + list);
    }

}
```

```
[-3, -2, -1] [-3, -2, -1]
```

The Collections Class UML Diagram

Example: Using the Collections Class

This example demonstrates using the methods in the Collections class. The example creates a list, sorts it, and searches for an element. The example wraps the list into a synchronized and read-only list.

TestCollections

Run

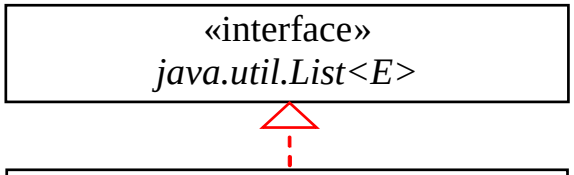
The Vector and Stack Classes

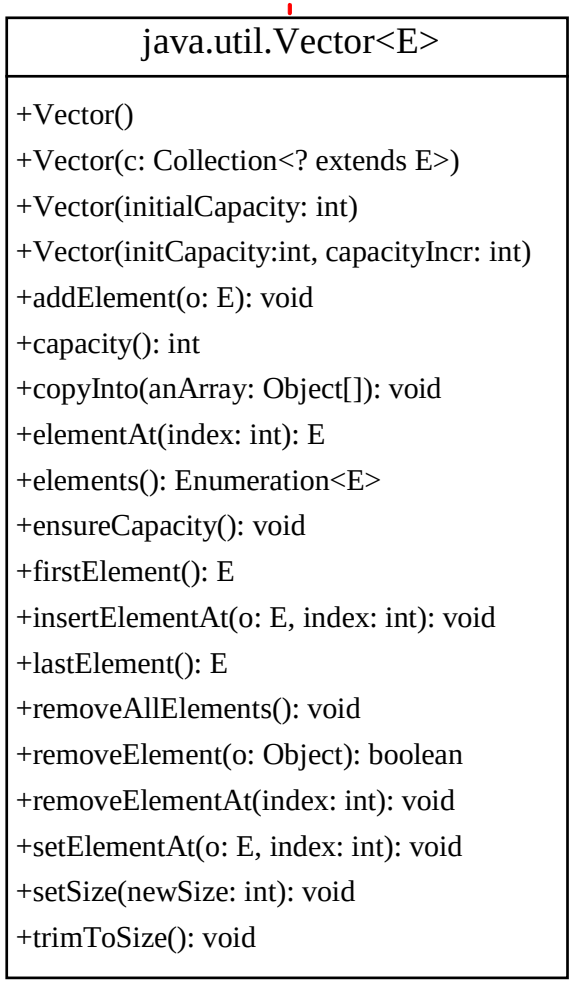
The Java Collections Framework was introduced with Java 2. Several data structures were supported prior to Java 2. Among them are the Vector class and the Stack class. These classes were redesigned to fit into the Java Collections Framework, but their old-style methods are retained for compatibility. This section introduces the Vector class and the Stack class.

The Vector Class

In Java 2, Vector is the same as ArrayList, except that Vector contains the synchronized methods for accessing and modifying the vector. None of the new collection data structures introduced so far are synchronized. If synchronization is required, you can use the synchronized versions of the collection classes. These classes are introduced later in the section, “The Collections Class.”

The Vector Class, cont.





Creates a default empty vector with initial capacity 10.

Creates a vector from an existing collection.

Creates a vector with the specified initial capacity.

Creates a vector with the specified initial capacity and increment.

Appends the element to the end of this vector.

Returns the current capacity of this vector.

Copies the elements in this vector to the array.

Returns the object at the specified index.

Returns an enumeration of this vector.

Increases the capacity of this vector.

Returns the first element in this vector.

Inserts o to this vector at the specified index.

Returns the last element in this vector.

Removes all the elements in this vector.

Removes the first matching element in this vector.

Removes the element at the specified index.

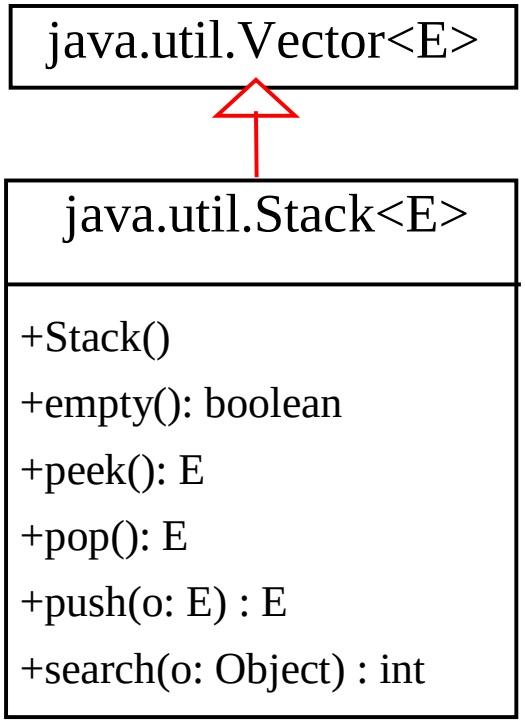
Sets a new element at the specified index.

Sets a new size in this vector.

Trims the capacity of this vector to its size.

The Stack Class

The Stack class represents a last-in-first-out stack of objects. The elements are accessed only from the top of the stack. You can retrieve, insert, or remove an element from the top of the stack.



Creates an empty stack.

Returns true if this stack is empty.

Returns the top element in this stack.

Returns and removes the top element in this stack.

Adds a new element to the top of this stack.

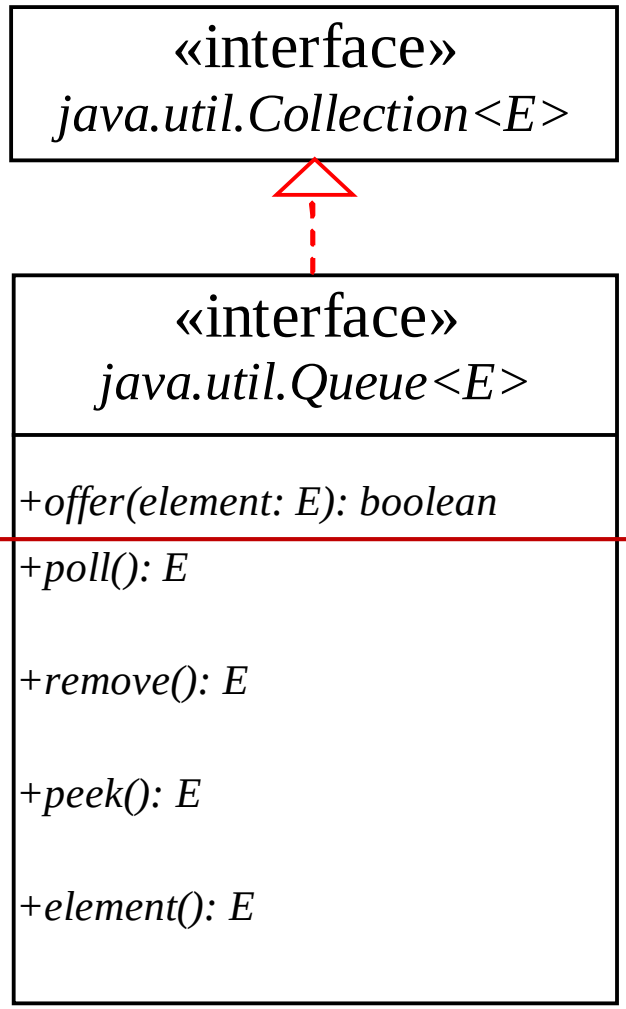
Returns the position of the specified element in this stack.

Queues and Priority Queues

A queue is a first-in/first-out data structure. Elements are appended to the end of the queue and are removed from the beginning of the queue.

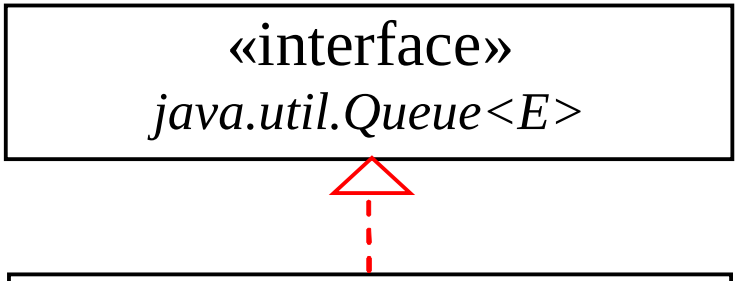
In a priority queue, elements are assigned priorities. When accessing elements, the element with the highest priority is removed first.

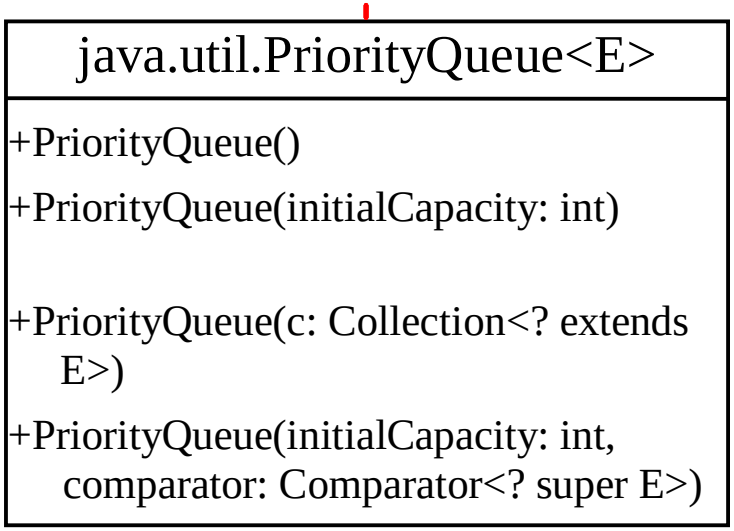
The Queue Interface



+offer(element: E): boolean	Inserts an element to the queue.
+poll(): E	Retrieves and removes the head of this queue, or null if this queue is empty.
+remove(): E	Retrieves and removes the head of this queue and throws an exception if this queue is empty.
+peek(): E	Retrieves, but does not remove, the head of this queue, returning null if this queue is empty.
+element(): E	Retrieves, but does not remove, the head of this queue, throwing an exception if this queue is empty.

The PriorityQueue Class





Creates a default priority queue with initial capacity 11.

Creates a default priority queue with the specified initial capacity.

Creates a priority queue with the specified collection.

Creates a priority queue with the specified initial capacity and the comparator.

PriorityQueueDemo

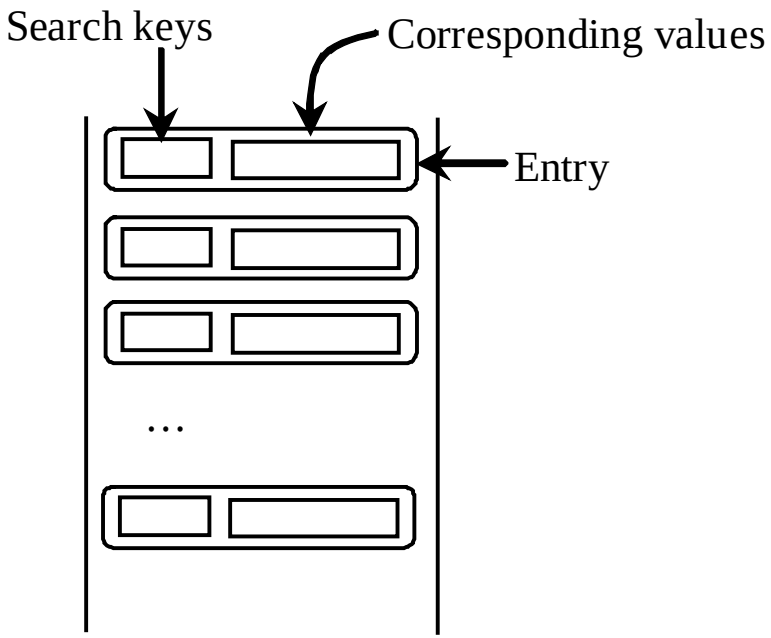
Run

```
public class PriorityQueueDemo {
  public static void main(String[] args) {
    PriorityQueue<String> queue1 = new PriorityQueue<String>();
    queue1.offer("Oklahoma");
    queue1.offer("Indiana");
    queue1.offer("Georgia");
    queue1.offer("Texas");

    System.out.println("Priority queue using Comparable:");
    while (queue1.size() > 0) {
      System.out.print(queue1.remove() + " ");
    }

    PriorityQueue<String> queue2 = new PriorityQueue<String>(
      4, Collections.reverseOrder());
    queue2.offer("Oklahoma");
    queue2.offer("Indiana");
    queue2.offer("Georgia");
    queue2.offer("Texas");

    System.out.println("\nPriority queue using Comparator:");
    while (queue2.size() > 0) {
      System.out.print(queue2.remove() + " ");
    }
  }
}
```

```
Priority queue using Comparable:
Georgia Indiana Oklahoma Texas
Priority queue using Comparator:
Texas Oklahoma Indiana Georgia
```

The Map Interface

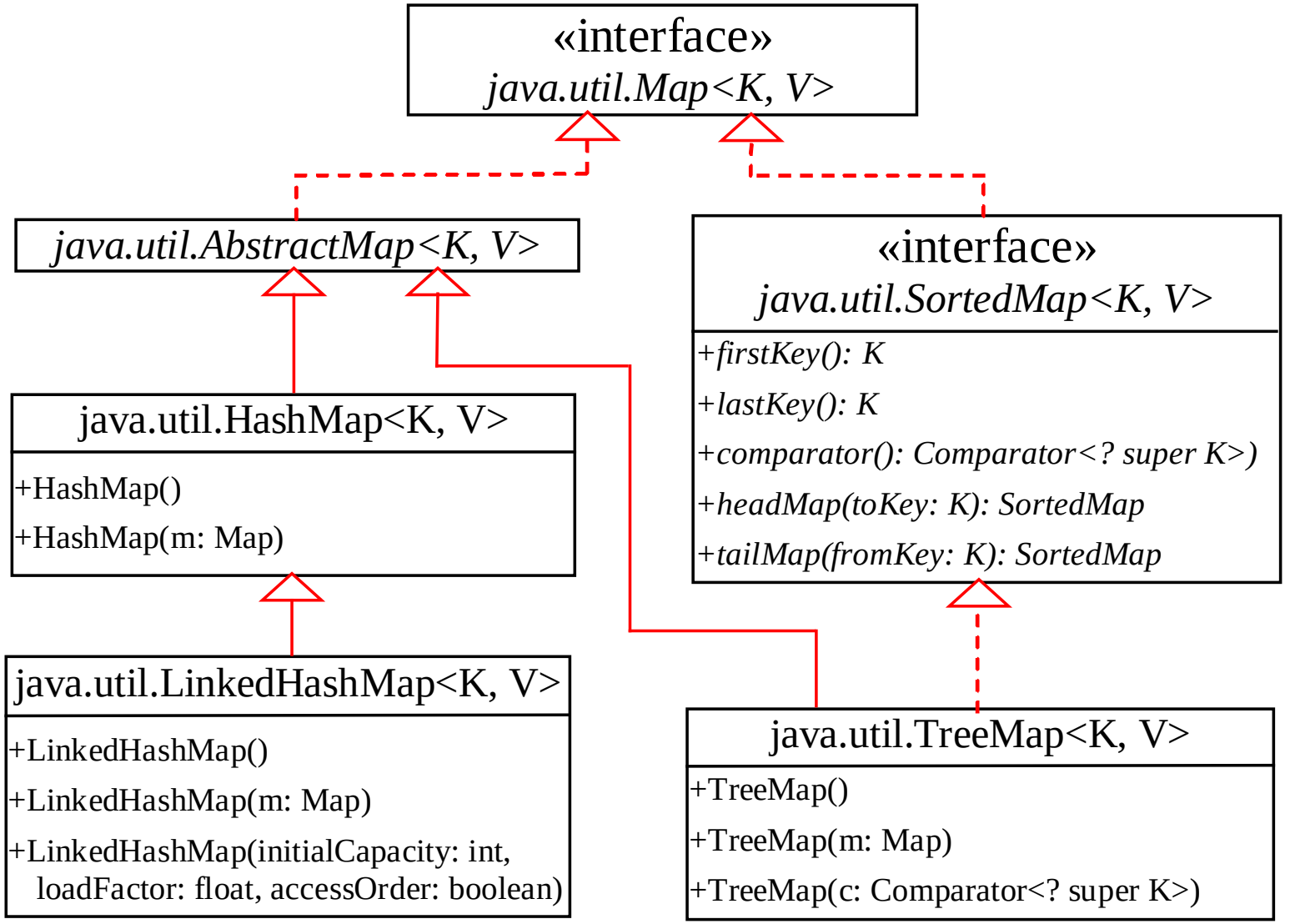
The Map interface maps keys to the elements. The keys are like indexes. In List, the indexes are integer. In Map, the keys can be any objects.



The Map Interface UML Diagram

java.util.Map<K, V>	
+clear(): void	Removes all mappings from this map.
+containsKey(key: Object): boolean	Returns true if this map contains a mapping for the specified key.
+containsValue(value: Object): boolean	Returns true if this map maps one or more keys to the specified value.
+entrySet(): Set	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 mappings.
+keySet(): Set<K>	Returns a set consisting of the keys in this map.
+put(key: K, value: V): V	Puts a mapping in this map.
+putAll(m: Map): void	Adds all the mappings from m to this map.
+remove(key: Object): V	Removes the mapping for the specified key.
+size(): int	Returns the number of mappings in this map.
+values(): Collection<V>	Returns a collection consisting of the values in this map.

Concrete Map Classes



HashMap and TreeMap

The HashMap and TreeMap classes are two concrete implementations of the Map interface.

The HashMap class is efficient for locating a value, inserting a mapping, and deleting a mapping.

The TreeMap class, implementing SortedMap, is efficient for traversing the keys in a sorted order.

LinkedHashMap

LinkedHashMap was introduced in JDK 1.4. It **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), or the order in which they were last accessed, from least recently accessed to most recently (access order).

The no-arg constructor constructs a LinkedHashMap with the insertion order. To construct a LinkedHashMap with the access order, use the LinkedHashMap(initialCapacity, loadFactor, true).

```
public class HashMap<K,V> extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable {
```

```
/**
 * The table, initialized on first use, and resized as
 * necessary. When allocated, length is always a power of two.
 * (We also tolerate length zero in some operations to allow
 * bootstrapping mechanics that are currently not needed.)
 */
transient Node<K,V>[] table;
```

```
/**
 * The number of key-value mappings contained in this map.
 */
transient int size;
```

```
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    V value;
    Node<K,V> next;
```

```java
public V put(K key, V value) {
    return putVal(hash(key), key, value, false, true);
}
```

```java
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    else {
        Node<K,V> e; K k;
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
            e = p;
        else if (p instanceof TreeNode)
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        else {
            for (int binCount = 0; ; ++binCount) {
                if ((e = p.next) == null) {
                    p.next = newNode(hash, key, value, null);
                    if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                        treeifyBin(tab, hash);
                    break;
                }
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    break;
                p = e;
            }
        }
        if (e != null) { // existing mapping for key
            V oldValue = e.value;
            if (!onlyIfAbsent || oldValue == null)
                e.value = value;
            afterNodeAccess(e);
            return oldValue;
        }
    }
    ++modCount;
```

```java
public V get(Object key) {
    Node<K,V> e;
    return (e = getNode(hash(key), key)) == null ? null : e.value;
}
```

```java
final Node<K,V> getNode(int hash, Object key) {
    Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (first = tab[(n - 1) & hash]) != null) {
        if (first.hash == hash && // always check first node
            ((k = first.key) == key || (key != null && key.equals(k))))
            return first;
        if ((e = first.next) != null) {
            if (first instanceof TreeNode)
                return ((TreeNode<K,V>)first).getTreeNode(hash, key);
            do {
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    return e;
            } while ((e = e.next) != null);
        }
    }
    return null;
}
```

- TreeMap, LinkedHashMap 源代码中可以分析其数据是如何组织的

Example: Using HashMap and TreeMap

This example creates a hash map that maps borrowers to mortgages. The program first creates a hash map with the borrower's name as its key and mortgage as its value. The program then creates a tree map from the hash map, and displays the mappings in ascending order of the keys.

TestMap

Run

Example: 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, convert it to a tree map.

CountOccurrenceOfWords

Run

NOTE

```
public final class PhoneNumber {
    private final short areaCode;
    private final short prefix;
    private final short lineNumber;
    public PhoneNumber(int areaCode, int prefix,int lineNumber) {
        this.areaCode = (short) areaCode;
        this.prefix = (short) prefix;
        this.lineNumber = (short) lineNumber;
    }
    @Override
    public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof PhoneNumber))
            return false;
        PhoneNumber pn = (PhoneNumber) o;
        return pn.lineNumber == lineNumber && pn.prefix == prefix
        && pn.areaCode == areaCode;
    }
    public static void main(String[] args) {
        Map<PhoneNumber, String> m = new HashMap<PhoneNumber, String>();
        m.put(new PhoneNumber(707, 867, 5309), "Jenny");
        System.out.println(m.get(new PhoneNumber(707, 867, 5309)));
    }
}
```

输出什么？

打印：null
因为两个new PhoneNumber是不同的对象，有不同的hashcode

```
PhoneNumber p1 = new PhoneNumber(707,867,5309);
PhoneNumber p2 = new PhoneNumber(707,867,5309);
System.out.println(p1.hashCode()+" " + p2.hashCode());
```

```
<terminated> PhoneNumber [Java Applicatic
366712642 1829164700
```

Object的hashCode（）方法，返回的是当前对象的“内存地址”。

- 每个覆盖了equals方法的类中，也必须覆盖hashCode方法；否则会违反Object.hashCode的通用约定，从而导致该类无法结合所有基于散列的集合一起正常运作，这样的集合包括HashMap、HashSet和Hashtable。

The Arrays Class

The Arrays class contains various static methods for sorting and searching arrays, for comparing arrays, and for filling array elements. It also contains a method for converting an array to a list.

The Arrays Class UML Diagram

Arrays	
+asList(a: Object[]): List	Returns a list from an array of objects
Overloaded binarySearch method for byte, char, short, int, long, float, double, and Object. +binarySearch(a: xType[], key: xType): int	Overloaded binary search method to search a key in the array of byte, char, short, int, long, float, double, and Object
Overloaded equals method for boolean, byte, char, short, int, long, float, double, and Object. +equals(a: xType[], a2: xType[]): boolean	Overloaded equals method that returns true if a is equal to a2 for a and a2 of the boolean, byte, char, short, int, long, float, and Object type
Overloaded fill method for boolean char, byte, short, int, long, float, double, and Object. +fill(a: xType[], val: xType): void +fill(a: xType[], fromIndex: int, toIndex: xType, val: xType): void	Overloaded fill method to fill in the specified value into the array of the boolean, byte, char, short, int, long, float, and Object type
Overloaded sort method for char, byte, short, int, long, float, double, and Object. +sort(a: xType[]): void +sort(a: xType[], fromIndex: int, toIndex: int): void	Overloaded sort method to sort the specified array of the char, byte, short, int, long, float, double, and Object type

Example: Using the Arrays Class

This example demonstrates using the methods in the Arrays class. The example creates an array of int values, fills part of the array with 50, sorts it, searches for an element, and compares the array with another array.

TestArrays

Run

EnumSet和EnumMap

- EnumSet提供了非常方便的方法来创建枚举集合。
- EnumSet 是一个与枚举类型一起使用的专用 Set 实现。枚举set中所有元素都必须来自单个枚举类型（即必须是同类型，且该类型是Enum的子类）。

```
public class Text {
    public enum Style {
        BOLD, ITALIC, UNDERLINE, STRIKETHROUGH
    }

    // Any Set could be passed in, but EnumSet is clearly best
    public void applyStyles(Set<Style> styles) {
        // Body goes here
        for(Style style : styles){
            System.out.println(style);
        }
    }

    // Sample use
    public static void main(String[] args) {
        Text text = new Text();
        text.applyStyles(EnumSet.of(Style.BOLD, Style.ITALIC));
    }
}
```

```
public static void main(String[] args) {
    System.out.println("EnumSet.noneOf");
    EnumSet<Student> set = EnumSet.noneOf(Student.class);
    set.add(Student.HARRY);
    set.add(Student.ROBBIE);
    set.add(Student.ROBIN);
    for (Student p : set)
        System.out.println(p);
    set.clear();
    System.out.println("EnumSet.allOf");
    set = EnumSet.allOf(Student.class);
    for (Student p : set)
        System.out.println(p);
    set.clear();
    System.out.println("EnumSet.Of one");
    set = EnumSet.of(Student.ROBIN);
    for (Student p : set)
        System.out.println(p);
    System.out.println("EnumSet.Of two");
    set = EnumSet.of(Student.ROBIN, Student.HARRY);
    for (Student p : set)
        System.out.println(p);
}
```

EnumSet是个虚类，只能通过它提供的静态方法来返回EnumSet的实现类的实例。

- EnumSet的两种不同的实现：如果EnumSet大小小于64，
- 就返回RegularEnumSet实例(当然它继承自EnumSet)，这个EnumSet实际上至用了一个long来存储这个EnumSet。
- 如果 EnumSet大小大于等于64，则返回JumboEnumSet实例，它使用一个long[]来存储。这样做的好处很明显： 大多数情况下返回的RegularEnumSet效率比JumboEnumSet高很多。

```
public static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> elementType) {
    Enum<?>[] universe = getUniverse(elementType);
    if (universe == null)
        throw new ClassCastException(elementType + " not an enum");

    if (universe.length <= 64)
        return new RegularEnumSet<>(elementType, universe);
    else
        return new JumboEnumSet<>(elementType, universe);
}
```

- EnumSet的效率比HashSet等效率高。
- 例如在RegularEnumSet中，对元素的操作是位操作。

```
public boolean add(E e) {
    typeCheck(e);

    long oldElements = elements;
    elements |= (1L << ((Enum<?>)e).ordinal());
    return elements != oldElements;
}
```

```
public boolean remove(Object e) {
    if (e == null)
        return false;
    Class<?> eClass = e.getClass();
    if (eClass != elementType && eClass.getSuperclass() != elementType)
        return false;

    long oldElements = elements;
    elements &= ~(1L << ((Enum<?>)e).ordinal());
    return elements != oldElements;
}
```

- EnumMap也是Map接口的实现，其key-value映射中的key是Enum类型。

```
public class EnumMap<K extends Enum<K>, V> extends AbstractMap<K, V>
    implements java.io.Serializable, Cloneable{
    private final Class<K> keyType;
    private transient K[] keyUniverse;
    private transient Object[] vals;
    private transient int size = 0;
}
```

- 用其它的Map实现（如HashMap）也能完成枚举类型实例到值得映射，但是使用EnumMap会更加高效：它只能接收同一枚举类型的实例作为键值，并且由于枚举类型实例的数量相对固定并且有限，所以EnumMap使用数组来存放与枚举类型对应的值。

添加<key, value>，本质上是添加到vals数组中。

```
public V put(K key, V value) {
    typeCheck(key);

    int index = key.ordinal();
    Object oldValue = vals[index];
    vals[index] = maskNull(value);
    if (oldValue == null)
        size++;
    return unmaskNull(oldValue);
}
```

```
private Object maskNull(Object value) {
    return (value == null ? NULL : value);
}
```

```
public class EnumMapTest
{
    public enum Color
    {
        RED,BLUE,BLACK,YELLOW,GREEN;
    }
    public static void main(String[] args)
    {
        EnumMap<Color,String> map = new EnumMap<>(Color.class);
        EnumMap<Color,String> map = new EnumMap<>(Color.class);
        map.put(Color.YELLOW, "黄色");
        map.put(Color.RED, "红色");
        map.put(Color.BLUE, null);
//        map.put(null, "无");   //会报NullPonitException的错误
        map.put(Color.BLACK, "黑色");
        map.put(Color.GREEN, "绿色");

        for(Map.Entry<Color,String> entry:map.entrySet())
        {
            System.out.println(entry.getKey()+":"+entry.getValue());
```

```
RED:红色
BLUE:null
BLACK:黑色
YELLOW:黄色
GREEN:绿色
{RED=红色, BLUE=null, BLACK=黑色, YELLOW=黄色, GREEN=绿色}
```

Enu
在e

Java Collection Framework的实现利用了很多内嵌类

- 如LinkedHashMap

```
final class LinkedKeySet extends AbstractSet<K> {

final class LinkedValues extends AbstractCollection<V> {

 final class LinkedEntrySet extends AbstractSet<Map.Entry<K,V>> {

final class LinkedKeyIterator extends LinkedHashIterator
    implements Iterator<K> {
    public final K next() { return nextNode().getKey(); }
}

final class LinkedValueIterator extends LinkedHashIterator
    implements Iterator<V> {
    public final V next() { return nextNode().value; }
}

final class LinkedEntryIterator extends LinkedHashIterator
    implements Iterator<Map.Entry<K,V>> {
    public final Map.Entry<K,V> next() { return nextNode(); }
}
```