

JAVA PROGRAMMERS GUIDE LESSON 1

| | |
|---------------|-----------------|
| File: | JGuiGuideL1.doc |
| Date Started: | July 10, 2000 |
| Last Update: | Jan 2, 2002 |
| Status: | proof |

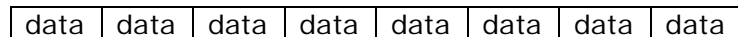
DICTIONARIES, MAPS AND COLLECTIONS

We have classes for **Sets**, **Lists** and **Maps** and **Dictionaries**. We have interfaces to represent the constructed objects and to state what methods the implementation class must implement. We need to know how data is stored Data may be stored what **arrays**, **lists**, **hash tables** and **trees**. Once we know how data is stored then will introduce all the Collection API interfaces.

DATA STRUCTURES

arrays

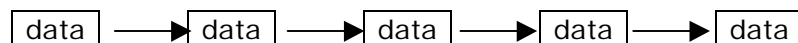
Arrays organize each data element as sequential memory cells each accessed by an index.



Arrays have inefficient memory, because you do not know how many items you have. Unsorted arrays have fast access and insertion times because data elements are easily accessed by an index. Unsorted arrays have slow search times because you need to look at every item to find the element you want. Arrays that are sorted have slower insertion time because data may need to be moved around to maintain a sorted order but fast search times.

lists

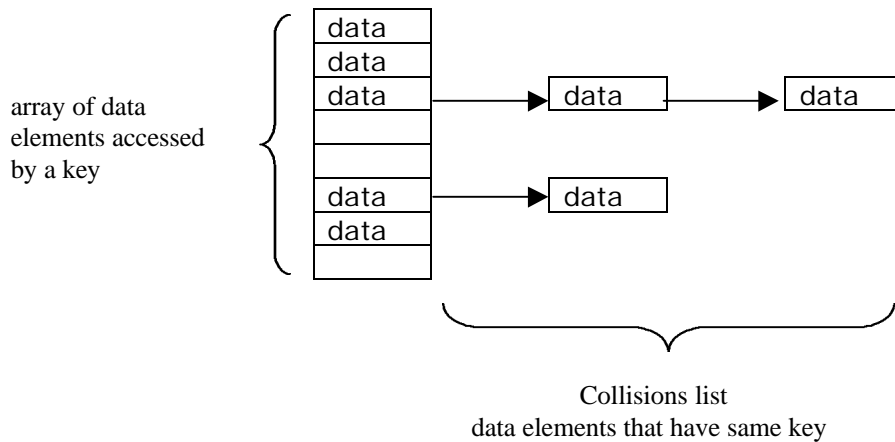
Each data element has its own memory cell. Each memory cell has a pointer to the next memory cell.



Lists have efficient memory because you just use memory as you need it. Sorted lists have slow access and insertion times because you have to search the list to find where to insert the data element. Unsorted lists have fast access and insertion times because you are just adding to the end of the list. Both unsorted and sorted lists have slow search times because you have to look at every data element to find the one you want.

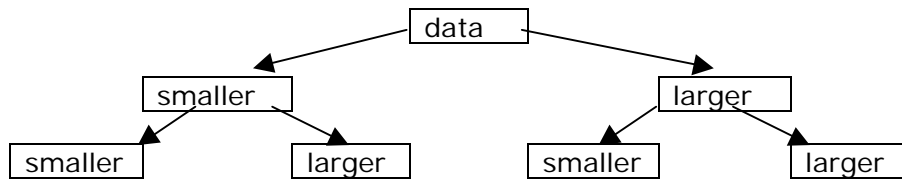
hash

A hash table uses an array and a list. An array index or key is generated from a data value. This can be just as simple as adding up all its characters and taking the **modulus** of the table size. Memory is still inefficient because we do not know how many items we have. The search time now is very fast. You just go to the array index represented by the key. Sometime two data elements have the same key. In this situation a link list is used to store data elements having the same key. Data values with the same key is known as **collisions**



trees

Each data element has its own memory cell. Each memory cell has a pointer to two other memory cells. A memory cell that has a lower value (left side) a memory cells that has a higher value (right cell). The left memory cell is called the left child. The right memory cell is called the right child. A tree cannot have duplicate keys.



Trees have efficient memory because you just use memory as you need it. Access, insertion, search times are fast because you do not need to look at every data element to find the element you want. Since a tree is ordered the left side is smaller than the right side you can now find data elements fast. The timing is log base 2 n because it only takes 4 comparisons to find an item in a tree with 16 items. log 16 is 4.

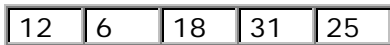
The following data structures are not part of the Collections interface but part of java.util package.

vectors

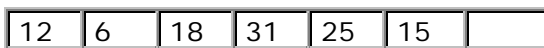
A vector is a dynamically resizing array. The major problem when you declare an array, it is of fixed size. If you need to add more items, and your array is filled you are stuck. If you need to delete a lot of items in your array permanently then you are left with a large amount of computer memory unused. Vectors come to the rescue. In a vector ADT you allocate array memory to a reasonable size. If the vector gets filled it will allocate some more memory at an agreed reasonable amount called the **step**. When items are deleted the vector will shrink the allocated memory to a reasonable size accordingly to the step. We use a step because we want new items are to be added so no new memory needs to be allocated continuously.

Vector operation:

Create a vector of size 5 and step 2 and fill with 5 data items

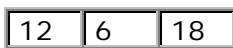


Add 1 more data item, the vector grows to 7



increase size by step(2)

Delete 3 data items the vector shrinks to 3

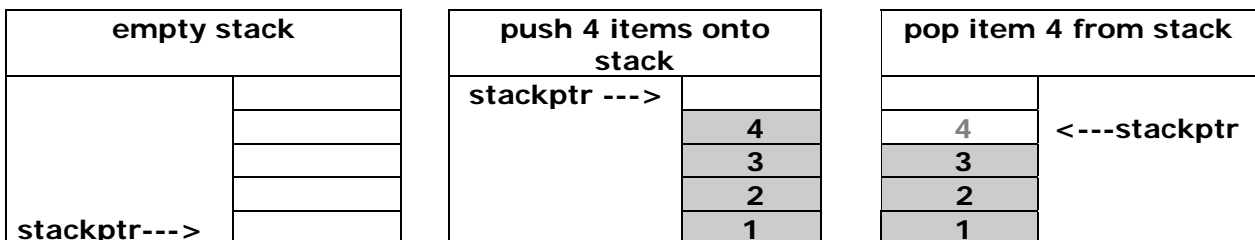


decrease size by 2 steps (4)

Vectors are represented by the Vector class in package java.util.

stacks

A Stack allows you to insert and retrieve items as **last in first out** (LIFO) into a list. This means if you insert the number 1, 2 then 3 you will get back 3, 2 and 1 in the reverse order. Stack operations are known as **push** and **pop**. To insert an item onto the stack a push operation is done. To remove items from the stack a pop operation is done. The location to insert the data item into the stack is pointed to by the **stack pointer**. The stack pointer is initially set to zero to the beginning of the stack. The beginning of the stack is known as the stack bottom. The end of the stack is known as the stack top. When an item is pushed onto the stack the stackptr is incremented after the operation. When a item is popped of the stack the stackptr is decremented before the operation.

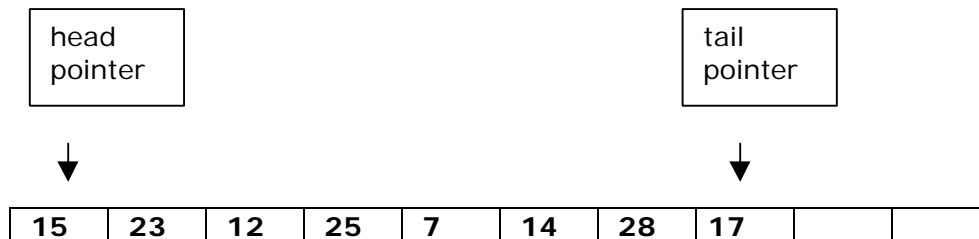


Think of a stack as a **stack of books** each being placed on top of each other as they are added.

Stacks are represented by the Stack class in package java.util.

queues

A Queue let you add items to the end of a list and to remove from the start of a list. A Queue implements **first in first out** (FIFO). This means if you insert 1, 2, and 3 you will get back 1,2 and 3. A Queue has both a head pointer and a tail pointer. Think as a Queue as a line in a bank. People enter the bank and stand in line. People get served in the bank at the start of the line. As each person is served they are removed from the Queue. Newcomers must start lining up at the end of the line. The first people who enter the bank are the first to be served and the first to leave. When you insert an item on the Queue it is known as **enqueue** and the tail pointer increments. When the tail pointer comes to the end of the array it wraps around to the start of the array. When you remove an item from the Queue it is known as **dequeue**. . When the tail pointer comes to the end of the array it wraps around to the start of the array.



Presently there is no Queue class in java.util.

DATA STRUCTURE COMPARISON CHART

We can make a chart listing all the characteristics of the data structure. With arrays the search time is faster if the list is sorted because we can start searching from a mid point. Searching sorted lists or unsorted lists there is not too much time difference because we always have to start at the beginning or end of the list. Hash tables and trees provide fast search time since they do their own sorting.

| data structure | memory | unsorted | | sorted | |
|----------------|-------------|-------------|-------------|-------------|-------------|
| | | access time | search time | access time | search time |
| array | inefficient | fast | slow | fast | fast |
| list | efficient | fast | slow | fast | slow |

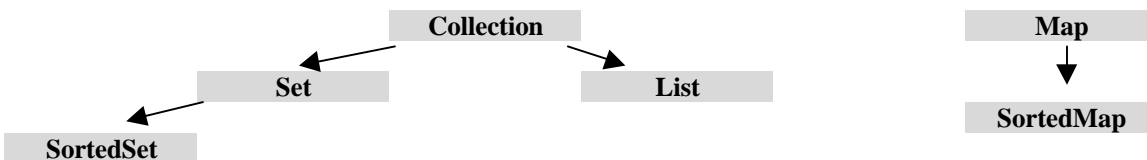
| data structure | memory | access time | search time |
|----------------|-------------|-------------|-------------|
| hashtable | inefficient | fast | fast |
| tree | efficient | fast | fast |

COLLECTIONS INTERFACES

All the collection classes implement the Collection interface. The Collection interface sub interfaces is quite extensive hierarchy. The following chart shows which class implements, which interface, and the supporting classes.

| interface | Implementation | | | | support |
|-------------------|----------------|-----------|---------|------------|------------|
| Collection | | | | | |
| Set | HashSet | | | | |
| SortedSet | | | TreeSet | | Comparable |
| List | | ArrayList | | LinkedList | |
| Map | HashMap | HashTable | | | |
| SortedMap | | | TreeMap | | Comparable |

The **Collection** interface is the super interface for the **Set** and **List** interfaces. The **Map** interface is the super interface for the **SortedMap** interface. Interfaces can extend other interfaces.



Collection

A collection represents a group of objects where each object is known as an element. The collection elements may or may not allow duplicates and may be ordered or unordered. The **Collection** interface has no implementation classes.

Set

A set is a collection that cannot contain duplicate elements. There are two **Set** implementations **HashSets** and **TreeSets**. **HashSets** have much faster access but are not ordered. **TreeSets** offer in-ordered insertions.

SortedSet

A sorted set is a set that keeps its elements in sorted order. The **Comparable** interface is used to provide ordering. Most Java data objects now implement the **Comparable** interface. If not, your data object must implement the **Comparable** interface.

List

A list is an ordered collection and may contain duplicate elements. There are two **List** implementations, **ArrayLists** and **LinkedLists**. **ArrayLists** are much faster than **LinkedLists**. **ArrayLists** take up more memory where as **LinkedLists** just uses memory when needed and are made up of nodes containing the data linked together.

Map

A map maps keys to values. Maps cannot contain duplicate keys. Each key can only map to one value. There is only one map implementation the **HashMap**. The order is unpredictable and allows duplicate keys. The HashTable is the same as a HashMap is an older version. The key is used to generate a location where the data is stored.

SortedMap

A sorted map is a map that maintains its mapping in ascending key order. You can have only one key. Sorted maps are used for dictionaries or telephone directories. Where the key is the word and the value is the meaning description. Or the key is the name of the person and the value is the telephone number. There is only one map implementation the **TreeMap**. A **TreeMap** cannot have duplicate keys. If the same key exists the old value is overwritten. The **TreeMap** offers sorted in-ordered insertions using data objects that implement the comparable interface. Most Java data objects now implement the **Comparable** interface. If not, then your data object must implement the **Comparable** interface.

SUPPORT

Comparable Interface

Objects used in lists and array sorted maps implement the **Comparable** interface. The `compareTo()` method compares this object with the specified object for order. Returns a negative integer if less than, zero if equal to or a positive integer if greater than the specified object.

```
public interface Comparable
{
    public int compareTo(Object o);
}
```

Comparator Interface

The **Comparable** interface is implemented by an object that wants to compare two objects arguments for order. The compare method returns a negative number if less, zero if equal and a positive integer if the first argument is greater than the second. The equals method returns true if the objects data contents are equal.

```
public interface Comparator
{
    int compare(Object o1, Object o2);
    boolean equals(Object obj);
}
```

Iterator Interface

An **Iterator** iterates over a collection. An **Iterator** takes the place of an **Enumeration** in the Java collections framework. Iterators differ from enumerations. Iterators allow you to remove elements from the collection as you iterate through the collection.

```
public interface Iterator
{
    // Returns true if the iteration has more elements
    boolean hasNext();
    // Returns the next element in the iteration.
    Object next();

    /* Removes the last element returned by the iterator . This method can be called only once per call to next. The
    behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any
    way other than by calling this method. The UnsupportedOperationException is thrown if the remove operation is not
    supported by this Iterator. The exception IllegalStateException is thrown if the next method has not yet been called, or
    the remove method has already been called after the last call to the next method. */
    void remove();
}
```

ListIterator interface

The List iterator is used for traversing a list. The ListIterator extends Iterator.

```
public interface ListIterator extends Iterator
{
    // Inserts the specified element into the list
    // The UnsupportedOperationException is thrown if the add() method is
    // not supported by this list iterator.
    // The ClassCastException is thrown if the class of the specified element
    // prevents it from being added to this Set.
    // The IllegalArgumentException is thrown if some aspect of this element
    // prevents it from being added to this Collection.
    void add(Object o);

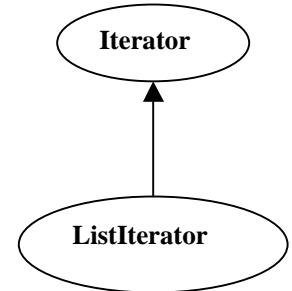
    // Returns true if this list iterator has more elements when
    // traversing the list in the forward direction.
    boolean hasNext();

    // Returns true if this list iterator has more elements when traversing the
    // list in the reverse direction.
    boolean hasPrevious();

    // Returns the next element in the list.
    // The NoSuchElementException is thrown if the iteration has no next element.
    Object next();

    // Returns the index of the element that would be returned by a
    // subsequent call to next()
    int nextIndex();

    // Returns the previous element in the list
    // NoSuchElementException thrown if the iteration has no previous element.
    Object previous();
}
```



```

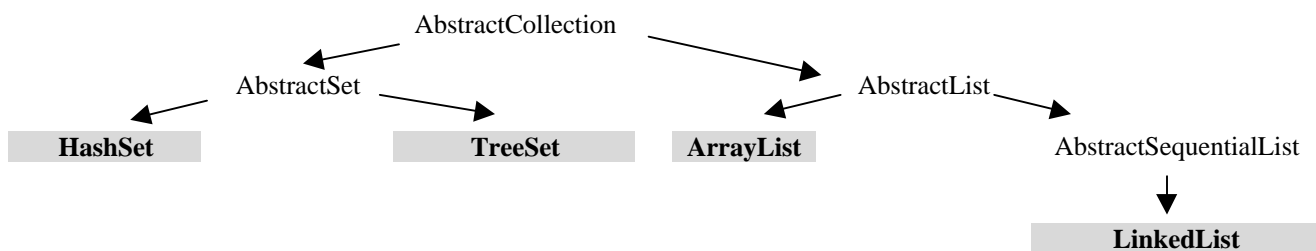
// Returns the index of the element that would be returned by a
// subsequent call to previous()
int previousIndex();
// Removes from the list the element that was returned by
// next() or previous ()
// UnsupportedOperationException thrown if the remove operation is not
// supported by this list iterator.
// IllegalStateException is thrown if neither next() nor previous() have
// been called, or remove() or add() have been called.
void remove();

// Replaces the element returned by next() or previous() with the
// specified element
// The UnsupportedOperationException is thrown if the set operation
// is not supported by this list iterator.
// The ClassCastException is thrown if the class of the specified element
// prevents it from being added to this list.
// The IllegalArgumentException is thrown if some aspect of the specified
// element prevents it from being added to this list.
// The IllegalStateException is thrown if neither next() nor previous() have
// been called() or remove() or add() have been called after the last call to
// next() or previous().
void set(Object o);
}

```

IMPLEMENTATION CLASSES

Each class implementing the collection interface has a super class. It may seem kind of confusing having a interface hierarchy and a implementation hierarchy. The Set implementation classes extends **AbstractSet**. The List implementation classes extend **AbstractList**. Both **AbstractSet** and **AbstractList** classes extends **AbstractCollection**.



Abstract Set

The **AbstractSet** class contains all the common methods for the **HashSet** and **TreeSet** classes.

| | |
|---------------------------------------|---|
| protected <code>AbstractSet()</code> | constructor. |
| boolean <code>equals(Object o)</code> | Compares the specified object with this set for equality. |
| int <code>hashCode()</code> | Returns the hash code value for this set, the sum of the hash codes of the elements in the set. |

Methods inherited from class **AbstractCollection** are `add`, `addAll`, `clear`, `contains`, `containsAll`, `isEmpty`, `iterator`, `remove`, `removeAll`, `retainAll`, `size`, `toArray`, `toArray`, `toString`

AbstractSequentialList class

The **AbstractSequentialList** class contains all the common methods for the **LinkedList** class.

| | |
|--|---|
| protected <code>AbstractSequentialList()</code> | Sole constructor. |
| void <code>add(int index, Object element)</code> | Inserts the specified element at the specified position in this list. |
| boolean <code>addAll(int index, Collection c)</code> | Inserts all of the elements in the specified collection into this list at the specified position. |
| Object <code>get(int index)</code> | Returns the element at the specified position in this list. |
| Iterator <code>iterator()</code> | Returns an iterator over the elements in this list (in proper sequence). |
| abstract ListIterator <code>listIterator(int index)</code> | Returns a list iterator over the elements in this list (in proper sequence). |
| Object <code>remove(int index)</code> | Removes the element at the specified position in this list. |
| Object <code>set(int index, Object element)</code> | Replaces the element at the specified position in this list with the specified element. |

Methods inherited from class **AbstractList** are `add`, `clear`, `equals`, `hashCode`, `indexOf`, `lastIndexOf`, `listIterator`, `removeRange`, `subList`

Methods inherited from class **AbstractCollection** are `addAll`, `contains`, `containsAll`, `isEmpty`, `remove`, `removeAll`, `retainAll`, `size`, `toArray`, `toArray`, `toString`

AbstractList

The **AbstractList** contains all the common methods for the **ArrayList** and **ArraySequentialList** classes.

| | |
|--|--|
| protected <code>AbstractList()</code> | Sole constructor. |
| void <code>add(int index, Object element)</code> | Inserts the specified element at the specified position in this list (optional operation). |
| boolean <code>add(Object o)</code> | Appends the specified element to the end of this List (optional operation). |
| boolean <code>addAll(int index, Collection c)</code> | Inserts all of the elements in the specified collection into this list at the specified position (optional operation). |
| void <code>clear()</code> | Removes all of the elements from this collection (optional operation). |
| boolean <code>equals(Object o)</code> | Compares the specified object with this list for equality. |
| abstract Object <code>get(int index)</code> | Returns the element at the specified position in this list. |
| int <code>hashCode()</code> | Returns the hash code value for this list. |

| | |
|---|---|
| <code>int indexOf(Object o)</code> | Returns the index in this list of the first occurrence of the specified element, or -1 if the list does not contain this element. |
| <code>Iterator iterator()</code> | Returns an iterator over the elements in this list in proper sequence. |
| <code>int lastIndexOf(Object o)</code> | Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element. |
| <code>ListIterator listIterator()</code> | Returns an iterator of the elements in this list (in proper sequence). |
| <code>ListIterator listIterator(int index)</code> | Returns a list iterator of the elements in this list (in proper sequence), starting at the specified position in the list. |
| <code>Object remove(int index)</code> | Removes the element at the specified position in this list (optional operation). |
| <code>protected void removeRange(int fromIndex, int toIndex)</code> | Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive. |
| <code>Object set(int index, Object element)</code> | Replaces the element at the specified position in this list with the specified element (optional operation). |
| <code>List subList(int fromIndex, int toIndex)</code> | Returns a view of the portion of this list between fromIndex, inclusive, and toIndex, exclusive. |

Methods inherited from class **AbstractCollection** are `addAll`, `contains`, `containsAll`, `isEmpty`, `remove`, `removeAll`, `retainAll`, `size`, `toArray`, `toArray`, `toString`

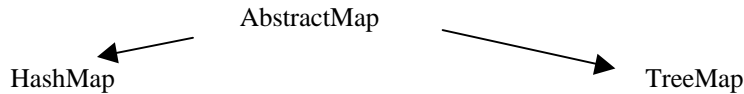
AbstractCollection class

The **AbstractCollection** class contains all the common methods for the **AbstractSet** and **AbstractList** classes.

| | |
|--|---|
| <code>protected AbstractCollection()</code> | Sole constructor. |
| <code>boolean add(Object o)</code> | Ensures that this collection contains the specified element (optional operation). |
| <code>boolean addAll(Collection c)</code> | Adds all of the elements in the specified collection to this collection (optional operation). |
| <code>void clear()</code> | Removes all of the elements from this collection (optional operation). |
| <code>boolean contains(Object o)</code> | Returns true if this collection contains the specified element. |
| <code>boolean containsAll(Collection c)</code> | Returns true if this collection contains all of the elements in the specified collection. |
| <code>boolean isEmpty()</code> | Returns true if this collection contains no elements. |
| <code>abstract Iterator iterator()</code> | Returns an iterator over the elements contained in this collection. |
| <code>boolean removeAll(Collection c)</code> | Removes from this collection all of its elements that are contained in the specified collection (optional operation). |
| <code>boolean retainAll(Collection c)</code> | Retains only the elements in this collection that are contained in the specified collection (optional operation). |
| <code>abstract int size()</code> | Returns the number of elements in this collection. |
| <code>Object[] toArray()</code> | Returns an array containing all of the elements in this collection. |
| <code>Object[] toArray(Object[] a)</code> | Returns an array with a runtime type is that of the specified array and that contains all of the elements in this collection. |
| <code>String toString()</code> | Returns a string representation of this collection. |

MAP IMPLEMENTATION CLASSES

The **Map** implementation classes extend **AbstractMap**.



Abstract Map

The **AbstractMap** class contains all the common methods for the **HashMap** and **TreeMap** classes.

| | |
|--|---|
| <code>protected AbstractMap()</code> | Sole constructor. |
| <code>void clear()</code> | Removes all mappings from this map (optional operation). |
| <code>boolean containsKey (Object key)</code> | Returns true if this map contains a mapping for the specified key. |
| <code>boolean containsValue (Object value)</code> | Returns true if this map maps one or more keys to this value. abstract |
| <code>Set entrySet()</code> | Returns a set view of the mappings contained in this map |
| <code>boolean equals(Object o)</code> | Compares the specified object with this map for equality. |
| <code>Object get(Object key)</code> | Returns the value to which this map maps the specified key. |
| <code>int hashCode()</code> | Returns the hash code value for this map. |
| <code>boolean isEmpty()</code> | Returns true if this map contains no key-value mappings. |
| <code>Set keySet()</code> | Returns a Set view of the keys contained in this map. |
| <code>Object put (Object key, Object value)</code> | Associates the specified value with the specified key in this map (optional operation). |
| <code>void putAll(Map t)</code> | Copies all of the mappings from the specified map to this map (optional operation). |
| <code>Object remove(Object key)</code> | Removes the mappings for this key from this map if present (optional operation) |
| <code>int size()</code> | returns the number of key-values mappings in this map |
| <code>String toString()</code> | returns a string representation of this map |
| <code>Collection values()</code> | returns a collection view of the values contained in this map |

HASHSET

The **HashSet** implements the **Set** interface using a hash table. There is no order to the insertion. The Hash set is not synchronized. If multiple threads are to access the hash set then you must synchronize the hash set by using the `Collections.synchronizedSet` wrapper method:

```
Sorted set s = Collections.synchronizedSet(new HashSet());
```

The iterator generates a **ConcurrentModificationException** if the **HashSet** is modified after the iterator is created. A hash set has a default size of 100 elements and a load factor of .75. The capacity states how many elements the hash table will contain. The load factor is how many actual elements the hash table will accept before being re-hashed. Rehashing is when the hash table size is increased and the elements are re-inserted. Constructors are used to create a empty **HashSet**, initialize with elements from an existing **Collection**, a empty **HashSet** with initial **Capacity** or **initialCapacity** and **load factor**.

Constructors

| | |
|---|---|
| <code>HashSet()</code> | Constructs a new, empty set; the backing <code>HashMap</code> instance has default capacity and load factor, which is 0.75. |
| <code>HashSet(Collection c)</code> | Constructs a new set containing the elements in the specified collection. |
| <code>HashSet(int initialCapacity)</code> | Constructs a new, empty set; the backing <code>HashMap</code> instance has the specified initial capacity and default load factor, which is 0.75. |
| <code>HashSet(int initialCapacity, float loadFactor)</code> | Constructs a new, empty set; the backing <code>HashMap</code> instance has the specified initial capacity and the specified load factor. |

Methods

| | |
|---|---|
| <code>boolean add(Object o)</code> | Adds the specified element to this set if it is not already present. |
| <code>void clear()</code> | Removes all of the elements from this set. |
| <code>Object clone()</code> | Returns a shallow copy of this <code>HashSet</code> instance; the elements themselves are not cloned. |
| <code>boolean contains(Object o)</code> | Returns true if this set contains the specified element. |
| <code>boolean isEmpty()</code> | Returns true if this set contains no elements. |
| <code>Iterator iterator()</code> | Returns an iterator over the elements in this set. |
| <code>boolean remove(Object o)</code> | Removes the given element from this set if it is present. |
| <code>int size()</code> | Returns the number of elements in this set (its cardinality). |

Methods inherited from class **AbstractSet** are [equals](#), [hashCode](#).

Methods inherited from class **AbstractCollection** are [addAll](#), [containsAll](#), [removeAll](#), [retainAll](#), [toArray](#), [toArray](#), [toString](#)

sample program using HashSet

```
import java.util.*;

class HashSetTest
{
    public static void main(String[] args)
    {
        // create hash set
        HashSet hs = new HashSet();
        // add entries
        hs.add("tom");
        hs.add("bill");
        hs.add("jane");
        hs.add("jane");
        // print out table
        System.out.println(hs);
        // contains
        System.out.println
        ("contains jane: " + hs.contains("jane"));
        // print out current hash code
        System.out.println("hash code: " + hs.hashCode());
        // print out current size of table
        System.out.println("size: " + hs.size());
    }
}
```

```

// remove key
System.out.println("removing jane");
hs.remove("jane");
// print out table
System.out.println(hs);
// print out current hash code
System.out.println("hash code: " + hs.hashCode());
}

```

program output: (notice only 1 entry for jane)

```

[tom, jane, bill]
contains jane: true
hash code: 6393479
size: 3
removing jane
[tom, bill]
hash code: 3138905

```

TREESET

This class implements the Set interface sorted in ascending order using the comparable interface or your own comparator implementing the comparable interface. The **TreeSet** is not synchronized. If multiple threads are to access the TreeSet then you must synchronize the TreeSet by using the Collections.synchronizedSet wrapper method.

Sorted set s = Collections.synchronizedSet(new TreeSet())

Constructors

| | |
|-----------------------|--|
| TreeSet() | Constructs a new, empty set, sorted according to the elements' natural order. |
| TreeSet(Collection c) | Constructs a new set containing the elements in the specified collection, sorted according to the elements' natural order. |
| TreeSet(Comparator c) | Constructs a new, empty set, sorted according to the given comparator. |
| TreeSet(SortedSet s) | Constructs a new set containing the same elements as the given sorted set, sorted according to the same ordering. |

methods

| | |
|-------------------------------------|--|
| boolean add(Object o) | Adds the specified element to this set if it is not already present. |
| boolean addAll(Collection c) | Adds all of the elements in the specified collection to this set. |
| void clear() | Removes all of the elements from this set. |
| Object clone() | Returns a shallow copy of this TreeSet instance. |
| Comparator comparator() | Returns the comparator used to order this sorted set, or null if this tree map uses its keys natural ordering. |
| boolean contains(Object o) | Returns true if this set contains the specified element. |
| Object first() | Returns the first (lowest) element currently in this sorted set. |
| SortedSet headSet(Object toElement) | Returns a view of the portion of this set whose elements are strictly less than toElement. |
| boolean isEmpty() | Returns true if this set contains no elements. |
| Iterator iterator() | Returns an iterator over the elements in this set. |
| Object last() | Returns the last (highest) element currently in this sorted set. |

| | |
|--|--|
| <code>boolean remove(Object o)</code> | Removes the given element from this set if it is present. |
| <code>int size()</code> | Returns the number of elements in this set (its cardinality). |
| <code>SortedSet subSet (Object fromElement, Object toElement)</code> | Returns a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive. |
| <code>SortedSet tailSet (Object fromElement)</code> | Returns a view of the portion of this set whose elements are greater than or equal to fromElement. |

Methods inherited from class **AbstractSet** are [equals](#), [hashCode](#).

Methods inherited from class **AbstractCollection** are [containsAll](#), [removeAll](#), [retainAll](#), [toArray](#), [toArray](#), [toString](#)

sample program using TreeSet

```
import java.util.*;

class TreeSetTest
{
    public static void main(String[] args)
    {
        // create tree set
        TreeSet ts = new TreeSet();
        // add entries
        ts.add("tom");
        ts.add("bill");
        ts.add("jane");
        ts.add("jane");
        // print out set
        System.out.println(ts);
        // first
        System.out.println("first: " + ts.first());
        // last
        System.out.println("last: " + ts.last());
        // contains
        System.out.println
        ("contains jane: " + ts.contains("jane"));
        // print out current hash code
        System.out.println("hash code: " + ts.hashCode());
        // print out current size of table
        System.out.println("size: " + ts.size());
        // remove key
        System.out.println("removing jane");
        ts.remove("jane");
        // print out set
        System.out.println(ts);
        // print out current hash code
        System.out.println("hash code: " + ts.hashCode());
    }
}
```

program output: (notice only 1 entry for jane)

```
[bill, jane, tom]
first: bill
last: tom
contains jane: true
hash code: 6393479
size: 3
removing jane
[bill, tom]
hash code: 3138905
```

ARRAYLIST

The ArrayList implements the List interface and is an expandable array. The ArrayList class is basically a Vector class without the synchronization overhead. If multiple threads are to access the hash set then you must synchronize the HashSet by using the Collections.synchronizedSet wrapper method.

```
List list = Collections.synchronizedSet(new ArrayList());
```

The iterator generates a ConcurrentModificationException if the HashSet is modified after the iterator is created.

Constructors

| | |
|--|---|
| <code>ArrayList()</code> | Constructs an empty list. |
| <code>ArrayList(Collection c)</code> | Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator. |
| <code>ArrayList (int initialCapacity)</code> | Constructs an empty list with the specified initial capacity. |

Methods

| | |
|---|---|
| <code>void add (int index, Object element)</code> | Inserts the specified element at the specified position in this list. |
| <code>boolean add(Object o)</code> | Appends the specified element to the end of this list. |
| <code>boolean addAll (Collection c)</code> | Appends all of the elements in the specified Collection to the end of this list, in the order that they are returned by the specified Collection's Iterator. |
| <code>boolean addAll (int index, Collection c)</code> | Inserts all of the elements in the specified Collection into this list, starting at the specified position. |
| <code>void clear()</code> | Removes all of the elements from this list. |
| <code>Object clone()</code> | Returns a shallow copy of this ArrayList instance. |
| <code>boolean contains (Object elem)</code> | Returns true if this list contains the specified element. |
| <code>void ensureCapacity (int minCapacity)</code> | Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument. |
| <code>Object get(int index)</code> | Returns the element at the specified position in this list. |
| <code>int indexOf (Object elem)</code> | Searches for the first occurrence of the given argument, testing for |

| | |
|--|--|
| | equality using the equals method. |
| boolean isEmpty() | Tests if this list has no elements. |
| int lastIndexOf (Object elem) | Returns the index of the last occurrence of the specified object in this list. |
| Object remove (int index) | Removes the element at the specified position in this list. |
| protected void removeRange (int fromIndex, int toIndex) | Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive. |
| Object set (int index, Object element) | Replaces the element at the specified position in this list with the specified element. |
| int size() | Returns the number of elements in this list. |
| Object[] toArray() | Returns an array containing all of the elements in this list in the correct order. |
| Object[] toArray (Object[] a) | Returns an array containing all of the elements in this list in the correct order. |
| void trimToSize() | Trims the capacity of this ArrayList instance to be the list's current size. |

Methods inherited from class **AbstractList** are equals, hashCode, iterator, listIterator, listIterator, subList

Methods inherited from class **AbstractCollection** are containsAll, remove, removeAll, retainAll, toString

sample program using ArrayList

```
import java.util.*;

class ArrayListTest
{
    public static void main(String[] args)
    {
        // create array list
        ArrayList al = new ArrayList();
        // add entries
        al.add("tom");
        al.add("bill");
        al.add("jane");
        al.add("jane");
        // print out array list
        System.out.println(al);
        // contains
        System.out.println
        ("contains key jane: " + al.contains("jane"));
        // index of
        System.out.println("index of jane: " + al.indexOf("jane"));
        // last index of
        System.out.println
        ("last index of jane: " + al.lastIndexOf("jane"));
        // get value for specified index
        System.out.println("get value for index 2: " + al.get(2));
    }
}
```

```

// print out current hash code
System.out.println("hash code: " + al.hashCode());
// print out current size of list
System.out.println("size: " + al.size());
// remove
System.out.println("removing jane");
al.remove("jane");
// print out array list
System.out.println(al);
// print out current hash code
System.out.println("hash code: " + al.hashCode());
}
}

```

program output: (notice multiple entries for jane)

```

[tom, bill, jane, jane]
contains key jane: true
index of jane: 2
last index of jane: 3
get value for index 2: jane
hash code: 2142789878
size: 4
removing jane
[tom, bill, jane]
hash code: 207564600

```

Try the Set ([int index](#), [Object element](#)) method.

LINKED LIST

The LinkedList implements the List interface. The LinkedList has methods to add and remove nodes at the beginning and end of list making it ideal for stacks, queues and deques. If multiple threads are to access the hash set then you must synchronize the HashSet by using the Collections.synchronizedSet wrapper method.

```
List list = Collections.synchronizedSet(new LinkedList());
```

The iterator generates a **ConcurrentModificationException** if the HashSet is modified after the iterator is created.

Constructors

| | |
|--|---|
| LinkedList() | Constructs an empty list. |
| LinkedList(Collection c) | Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator. |

Methods

| | |
|--|---|
| void add (int index, Object element) | Inserts the specified element at the specified position in this list. |
| boolean add(Object o) | Appends the specified element to the end of this list. |
| boolean addAll | Appends all of the elements in the specified collection to the end of |

| | |
|--|--|
| (Collection c) | this list, in the order that they are returned by the specified collection's iterator. boolean |
| addAll (int index, Collection c) | Inserts all of the elements in the specified collection into this list, starting at the specified position. |
| void addFirst(Object o) | Inserts the given element at the beginning of this list. |
| void addLast(Object o) | Appends the given element to the end of this list. |
| void clear() | Removes all of the elements from this list. |
| Object clone() | Returns a shallow copy of this LinkedList. |
| boolean contains(Object o) | Returns true if this list contains the specified element. |
| Object get(int index) | Returns the element at the specified position in this list. |
| Object getFirst() | Returns the first element in this list. |
| Object getLast() | Returns the last element in this list. |
| int indexOf(Object o) | Returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain |
| int lastIndexOf(Object o) | Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain |
| ListIterator listIterator (int index) | Returns a list-iterator of the elements in this list (in proper sequence), starting at the specified position in the list. |
| Object remove(int index) | Removes the element at the specified position in this list. |
| boolean remove(Object o) | Removes the first occurrence of the specified element in this list. |
| Object removeFirst() | Removes and returns the first element from this list. |
| Object removeLast() | Removes and returns the last element from this list. |
| int size() | Returns the number of elements in this list. |
| Object[] toArray() | Returns an array containing all of the elements in this list in the correct order. |
| Object[]toArray (Object[] a) | Returns an array containing all of the elements in this list in the correct order. |

Methods inherited from class **AbstractSequentialList** are [iterator](#)

Methods inherited from class **AbstractList** are [equals](#), [hashCode](#), [listIterator](#), [removeRange](#), [subList](#)

Methods inherited from class **AbstractCollection** are [containsAll](#), [isEmpty](#), [removeAll](#), [retainAll](#), [toString](#)

sample program using LinkedList

```
import java.util.*;

class LinkedListTest
{
public static void main(String[] args)
    {
        // create linked list
        LinkedList ll = new LinkedList();
        // add entries
        ll.add("tom");
        ll.add("bill");
        ll.add("jane");
        ll.add("jane");
        // print out linked list
        System.out.println(ll);
    }
}
```

```

// first key
System.out.println("first key: " + ll.getFirst());
// last key
System.out.println("last key: " + ll.getLast());
// contains
System.out.println("contains key jane: " + ll.contains("jane"));
// get value for specified index
System.out.println("get value for index 2: " + ll.get(2));
// print out current hash code
System.out.println("hash code: " + ll.hashCode());
// print out current size of list
System.out.println("size: " + ll.size());
// remove
System.out.println("removing jane");
ll.remove("jane");
// print out linked list
System.out.println(ll);
// print out current hash code
System.out.println("hash code: " + ll.hashCode());
}
}

```

program output: (notice multiple entries for jane)

```

[tom, bill, jane, jane]
first key: tom
last key: jane
contains key jane: true
get value for index 2: jane
hash code: 2142789878
size: 4
removing jane
[tom, bill, jane]
hash code: 207564600

```

DICTIONARY

The **Dictionary** class is the super class of the **HashTable**. A dictionary maps keys to values. This class is abstract and is now deprecated.

| | |
|---|---|
| Dictionary() | Sole constructor. |
| abstract Enumeration elements() | Returns an enumeration of the values in this dictionary. |
| Abstract Object get(Object key) | Returns the value to which the key is mapped in this dictionary. |
| abstract boolean isEmpty() | Tests if this dictionary maps no keys to value. |
| abstract Enumeration keys() | Returns an enumeration of the keys in this dictionary. |
| abstract Object put(Object key, Object value) | Maps the specified key to the specified value in this dictionary. |
| abstract Object remove (Object key) | Removes the key (and its corresponding value) from this dictionary. |
| abstract int size() | Returns the number of entries (distinct keys) in this dictionary |

HASHTABLE

A hash table maps keys to values. The objects used as keys must implement the `hashCode` and `equals` methods. A hash table has a capacity and a load factor. The capacity is the number of buckets in the hash table. The default capacity is 100. In case of collisions where two objects have the same hash key a bucket stores multiple entries. Each bucket then has to be searched sequentially for entries. The load factor indicates how full the hash table may get before the size of the hash table is increased. When a hash table size is increased this is known as **rehashing**. A hash table size is increased by calling the `rehash` method. The default load factor is 0,75. In JDK 1.2 the **Hashtable** class implements the **Map** interface and does not extend the abstract **Dictionary** class.

Constructors

| | |
|---|--|
| <code>Hashtable()</code> | Constructs a new, empty hash table with a default capacity and load factor, which is 0.75. |
| <code>Hashtable(int initialCapacity)</code> | Constructs a new, empty hash table with the specified initial capacity and default load factor, which is 0.75. |
| <code>Hashtable(int initialCapacity, float loadFactor)</code> | Constructs a new, empty hashtable with the specified initial capacity and the specified load factor. |
| <code>Hashtable(Map t)</code> | Constructs a new hashtable with the same mappings as the given Map. |

Methods

| | |
|---|--|
| <code>void clear()</code> | Clears this hashtable so that it contains no keys. |
| <code>Object clone()</code> | Creates a shallow copy of this hashtable. |
| <code>Boolean contains(Object value)</code> | Tests if some key maps into the specified value in this hashtable. |
| <code>Boolean containsKey(Object key)</code> | Tests if the specified object is a key in this hashtable. |
| <code>Boolean containsValue(Object value)</code> | Returns true if this Hashtable maps one or more keys to this value. |
| <code>Enumeration elements()</code> | Returns an enumeration of the values in this hashtable. |
| <code>Set entrySet()</code> | Returns a Set view of the entries contained in this Hashtable. |
| <code>boolean equals(Object o)</code> | Compares the specified Object with this Map for equality, as per the definition in the Map interface. |
| <code>Object get(Object key)</code> | Returns the value to which the specified key is mapped in this hashtable. |
| <code>int hashCode()</code> | Returns the hash code value for this Map as per the definition in the Map interface. |
| <code>boolean isEmpty()</code> | Tests if this hashtable maps no keys to values. |
| <code>Enumeration keys()</code> | Returns an enumeration of the keys in this hashtable. |
| <code>Set keySet()</code> | Returns a Set view of the keys contained in this Hashtable. |
| <code>Object put(Object key, Object value)</code> | Maps the specified key to the specified value in this hashtable. |
| <code>void putAll(Map t)</code> | Copies all of the mappings from the specified Map to this Hashtable These mappings will replace any mappings that this Hashtable had for any of the keys currently in the specified Map. |
| <code>protected void rehash()</code> | Increases the capacity of and internally reorganizes this hashtable, in order to accommodate and access its entries more efficiently. |
| <code>Object remove(Object key)</code> | Removes the key (and its corresponding value) from this hashtable. |
| <code>int size()</code> | Returns the number of keys in this hashtable. |

| | |
|---------------------|--|
| String toString() | Returns a string representation of this Hashtable object in the form of a set of entries, enclosed in braces and separated by the ASCII characters ", " (comma and space). |
| Collection values() | Returns a Collection view of the values contained in this Hashtable. |

sample program using Hashtable

```
import java.util.*;

class HashtableTest
{
public static void main(String[] args)
    {
    // create hash table
    Hashtable ht = new Hashtable();
    // add entries
    ht.put("tom", "345-8765");
    ht.put("bill", "654-9785");
    ht.put("jane", "067-9545");
    ht.put("jane", "876-0456");
    // print out table
    System.out.println(ht);
    // contains key
    System.out.println
    ("contains key jane: " + ht.containsKey("jane"));
    // contains value
    System.out.println
    ("contains value 876-0456: " + ht.containsValue("876-0456"));
    // get value for specified key
    System.out.println("get value for key jane: " + ht.get("jane"));
    // print out current hash code
    System.out.println("hash code: " + ht.hashCode());
    // print out current size of table
    System.out.println("size: " + ht.size());
    // remove key
    System.out.println("removing jane");
    ht.remove("jane");
    // print out table
    System.out.println(ht);
    // print out current hash code
    System.out.println("hash code: " + ht.hashCode());
    }
}
```

program output:

```
{tom=345-8765, jane=876-0456, bill=654-9785}
contains key jane: true
contains value 876-0456: true
get value for key jane: 876-0456
hash code: 2069433134
size: 3
removing jane
{tom=345-8765, bill=654-9785}
hash code: 100251513
```

HASHMAP

A HashMap implements the Map interface and is like a HashTable but is unsynchronized. There is no order to the map. Performance is affected by initial capacity and load factor. The capacity is the number of buckets in the hash table where the initial capacity is the number of buckets created when the HashMap is constructed. The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased. In this situation the capacity is doubled by calling the rehash method. A load factor of .75 is usually used. If multiple threads are to access the hash set then you must synchronize the hash set by using the Collections.synchronizedSet wrapper method.

```
Map m = Collections.synchronizedSet(new HashMap());
```

Constructor

| | |
|---|---|
| HashMap() | Constructs a new, empty map with a default capacity and load factor, which is 0.75. |
| HashMap (int initialCapacity) | Constructs a new, empty map with the specified initial capacity and default load factor, which is 0.75. |
| HashMap (int initialCapacity, float loadFactor) | Constructs a new, empty map with the specified initial capacity and the specified load factor. |
| HashMap(Map t) | Constructs a new map with the same mappings as the given map. |

Methods

| | |
|---------------------------------------|---|
| void clear() | Removes all mappings from this map. |
| Object clone() | Returns a shallow copy of this HashMap instance: the keys and values themselves are not cloned. |
| boolean containsKey (Object key) | Returns true if this map contains a mapping for the specified key. |
| boolean containsValue (Object value) | Returns true if this map maps one or more keys to the specified value. |
| Set entrySet() | Returns a collection view of the mappings contained in this map. |
| Object get(Object key) | Returns the value to which this map maps the specified key. |
| boolean isEmpty() | Returns true if this map contains no key-value mappings. |
| Set keySet() | Returns a set view of the keys contained in this map. |
| Object put (Object key, Object value) | Associates the specified value with the specified key in this map. |
| void putAll(Map t) | Copies all of the mappings from the specified map to this one. |
| Object remove (Object key) | Removes the mapping for this key from this map if present. |
| int size() | Returns the number of key-value mappings in this map. |
| Collection values() | Returns a collection view of the values contained in this map. |

Methods inherited from class **AbstractMap** are [equals](#), [hashCode](#), [toString](#)

sample program using HashMap

```
import java.util.*;

class HashTableTest
{
public static void main(String[] args)
{
    // create hash map
    HashMap hm = new HashMap();
    // add entries
    hm.put("tom", "345-8765");
    hm.put("bill", "654-9785");
    hm.put("jane", "067-9545");
    hm.put("jane", "876-0456"); // duplicate
    // print out table
    System.out.println(hm);
    // contains key
    System.out.println
    ("contains key jane: " + hm.containsKey("jane"));
    // contains value
    System.out.println
    ("contains value 876-0456: " + hm.containsValue("876-0456"));
    // get value for specified key
    System.out.println("get value for key jane: " + hm.get("jane"));
    // print out current hash code
    System.out.println("hash code: " + hm.hashCode());
    // print out current size of table
    System.out.println("size: " + hm.size());
    // remove key
    System.out.println("removing jane");
    hm.remove("jane");
    // print out table
    System.out.println(hm);
    // print out current hash code
    System.out.println("hash code: " + hm.hashCode());
}
}
```

program output: (notice only 1 jane entry old value overwritten)

```
{tom=345-8765, jane=876-0456, bill=654-9785}
contains key jane: true
contains value 876-0456: true
get value for key jane: 876-0456
hash code: 2069433134
size: 3
removing jane
{tom=345-8765, bill=654-9785}
hash code: 100251513
```

TREEMAP

This is a tree based implementation of the sorted map interface. The map will be sorted in ascending key order. using the supplied comparator or by a compurgator implementing the comparable interface. If multiple threads are to access the hash set then you must synchronize the hash set by using the Collections.synchronizedset wrapper method.

```
Map m = Collections.synchronizedSet(new TreeMap());
```

Constructor

| | |
|---------------------------|--|
| TreeMap() | Constructs a new, empty map, sorted according to the keys' natural order. |
| TreeMap (Comparator c) | Constructs a new, empty map, sorted according to the given comparator. |
| TreeMap (Map m) | Constructs a new map containing the same mappings as the given map, sorted according to the keys' natural order. |
| TreeMap (SortedMap m) | Constructs a new map containing the same mappings as the given SortedMap, sorted according to the same ordering. |

Methods

| | |
|---|--|
| void clear() | Removes all mappings from this TreeMap. |
| Object clone() | Returns a shallow copy of this TreeMap instance. |
| Comparator comparator() | Returns the comparator used to order this map, or null if this map uses its keys' natural order. |
| boolean containsKey (Object key) | Returns true if this map contains a mapping for the specified key. |
| boolean containsValue (Object value) | Returns true if this map maps one or more keys to the specified value. |
| Set entrySet() | Returns a set view of the mappings contained in this map. |
| Object firstKey() | Returns the first (lowest) key currently in this sorted map. |
| Object get(Object key) | Returns the value to which this map maps the specified key. |
| SortedMap headMap(Object toKey) | Returns a view of the portion of this map whose keys are strictly less than toKey. |
| Set keySet() | Returns a Set view of the keys contained in this map. |
| Object lastKey() | Returns the last (highest) key currently in this sorted map. |
| Object put (Object key, Object value) | Associates the specified value with the specified key in this map. |
| void putAll(Map map) | Copies all of the mappings from the specified map to this map. |
| Object remove (Object key) | Removes the mapping for this key from this TreeMap if present. |
| int size() | Returns the number of key-value mappings in this map. |
| SortedMap subMap (Object fromKey, Object toKey) | Returns a view of the portion of this map whose keys range from fromKey, inclusive, to toKey, exclusive. |
| SortedMap tailMap (Object fromKey) | Returns a view of the portion of this map whose keys are greater than or equal to fromKey. |
| Collection values() | Returns a collection view of the values contained in this map. |

Methods inherited from class **AbstractMap** are [equals](#), [hashCode](#), [isEmpty](#), [toString](#)

sample program using TreeMap

```
public static void main(String[] args)
{
    // create tree map
    TreeMap tm = new TreeMap();
    // add entries
    tm.put("tom", "345-8765");
    tm.put("bill", "654-9785");
    tm.put("jane", "067-9545");
    tm.put("jane", "876-0456");
    // print out table
    System.out.println(tm);
    // first key
    System.out.println("first key: " + tm.firstKey());
    // last key
    System.out.println("first key: " + tm.lastKey());
    // contains key
    System.out.println
    ("contains key jane: " + tm.containsKey("jane"));
    // contains value
    System.out.println
    ("contains value 876-0456: " + tm.containsValue("876-0456"));
    // get value for specified key
    System.out.println("get value for key jane: " + tm.get("jane"));
    // print out current hash code
    System.out.println("hash code: " + tm.hashCode());
    // print out current size of table
    System.out.println("size: " + tm.size());
    // remove key
    System.out.println("removing jane");
    tm.remove("jane");
    // print out table
    System.out.println(tm);
    // print out current hash code
    System.out.println("hash code: " + tm.hashCode());
}
```

program output: (notice only 1 jane entry old value overwritten)

```
{bill=654-9785, jane=876-0456, tom=345-8765}
first key: bill
first key: tom
contains key jane: true
contains value 876-0456: true
get value for key jane: 876-0456
hash code: 2069433134
size: 3
removing jane
{bill=654-9785, tom=345-8765}
hash code: 100251513
```

USING ITERATOR

An iterator just goes through each item in the set, list or map one by one. An iterator is like an enumeration except the iterator lets you remove an element.

A iterator object implements the iterator interface having the following methods.

| | |
|---------------------------------|--|
| <code>boolean hasNext();</code> | Returns true if the iteration has more elements |
| <code>Object next();</code> | Returns the next element in the iteration. |
| <code>void remove();</code> | Removes from the underlying collection the last element returned by the iterator . |

This remove method can be called only once per call to next. The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method. The **UnsupportedOperationException** is thrown if the remove operation is not supported by this Iterator. The exception **IllegalStateException** is thrown if the next method has not yet been called, or the remove method has already been called after the last call to the next method.

```
import java.util.*;

class IteratorTest
{

public static void main(String[] args)
    {
        // create hash map
        HashSet hs = new HashSet();
        // add entries
        hs.add("tom");
        hs.add("bill");
        hs.add("jane");

        // print out set
        System.out.println(hs);

        // get iterator
        Iterator itr = hs.iterator();

        // get elements from iterator and print out
        while(itr.hasNext())
            {
                String s = (String)itr.next();
                System.out.println(s);
            }

        // remove last element
        itr.remove();

        // print out set
        System.out.println(hs);
    }
}
```

program output: (notice the order of insertion is different)

```
[tom, jane, bill]
tom
jane
bill
[tom, jane]
```

LESSON 1 EXERCISE 1

Make a tree map that uses a linked list as its data elements. Insert many data elements using the same key. You need to first check if the key is in the tree. If it is not just insert the key and the hash set as the data element. If the key is present just insert the data into the linked list for that key. Print out the keys and all the data contents for each key in the tree using a iterator.

LESSON 1 EXERCISE 2

Make a ADT that gets a key and a value. Store all the keys in order of insertion. Basically you would need an ArrayList or Link List that stores a node with a key and data fields. Print out the keys and all the data contents for each key in the tree using an iterator.

IMPORTANT

You should use all the material in all the lessons to do the questions and exercises. If you do not know how to do something or have to use additional books or references to do the questions or exercises please let us know immediately. We want to have all the required information in our lessons. By letting us know we can add the required information to the lessons. The lessons are updated on a daily bases. We call our lessons the "living lessons". Please let us keep our lessons alive.

E-Mail all typos, unclear test, and additional information required to:

courses@cstutoring.com

E-Mail all attached files of your completed exercises to:

students@cstutoring.com

Order your next Lesson from:

<http://www.cstutoring.com/jgui.htm>

This lesson is copyright (C) 1998-2001 by The Computer Science Tutoring Center "cstutoring"
This document is not to be copied or reproduced in any form. For use of student only