A simple ordered Hashtable in Java ME

This article illustrates how to implement an ordered hashtable, which maps keys to values. Any non-null object can be used as a key or as a value.

As with typical Hashtables, to successfully store and retrieve objects from a hashtable, the objects used as keys must implement the hashCode method and the equals method.

There are many instances where you would like to use an ordered Hashtable, for example, to keep your user interface elements ordered, or to keep ordered items from a database or backend while keeping rapid access via Hashtable keys, or to store and access any value you want to access using a key.

This ordered Hashtable is called simple because internally it uses the Legacy collection classes, a Vector to maintain the element's order and a Hashtable to provide hashing capabilities. Because Hashtable and Vector grow differently, the implementation of

A Hashtable implementation

SimpleOrderedHashtable is not the most efficient one, but may be good enough for your needs.

```
/* ---------------------------------------------------------------------------
 * SimpleOrderedHashtable.java
 * Author: C. Enrique Ortiz
 * Copyright (c) 2004-2009 C. Enrique Ortiz <eortiz@j2medeveloper.com>
 *
 * SimpleOrderedHashtable.java implements a simple Hashtable that is
 * chronologically ordered.
 *
 * This is free software; you can redistribute it and/or modify it under
 * the terms of the GNU Lesser General Public License as published by the Free
 * Software Foundation; either version 2 of the License, or (at your option) any
 * later version.
 *
 * Usage & redistributions of source code must retain the above copyright notice.
 *
 * This software is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should get a copy of the GNU Lesser General Public License from
 * the Free Software Foundation, Inc., 59 Temple Place, Suite 330,
 * Boston, MA  02111-1307  USA
 * ---------------------------------------------------------------------------
 */

import java.util.Vector;
import java.util.Enumeration;
import java.util.Hashtable;

/**
 * Implements an Ordered Hashtable, with elements in
 * chronological order (i.e. insertion order)
 */
public class SimpleOrderedHashtable {

    private Vector orderedKeys;
    private Hashtable hashTable;

    /**
     * Constructor, creates an SimpleOrderedHashtable.
     */
    public SimpleOrderedHashtable() {
        orderedKeys = new Vector();
        hashTable = new Hashtable();
    }

    /**
     * Constructor, creates an SimpleOrderedHashtable.
     * @param initialCapacity is the initial size for the container.
     */
    public SimpleOrderedHashtable(int initialCapacity) {
        orderedKeys = new Vector(initialCapacity);
        hashTable = new Hashtable(initialCapacity);
    }

    /**
     * Maps the specified key to the specified value in this SimpleOrderedHashtable.
     * The value can be retrieved by calling the get method with a key that is
     * equal to the original key.
     * @param key is the hashtable key.
     * @param value is the value.
     * @return the previous value of the specified key in this
     * SimpleOrderedHashtable, or null if it did not have one.
     */
    synchronized public Object put(Object key, Object value) {
        int i = orderedKeys.indexOf(key);
        if (i == -1) {
            // Add new name/value pair.
            orderedKeys.addElement(key); // insert (append) to the end of the list
        } else {
            // Replace name/value pair.
            orderedKeys.setElementAt(key, i);
        }
        return hashTable.put(key, value);
    }

    /**
     * Returns the value to which the specified key is mapped in this
     * hashtable.
     * @param key is a key in the SimpleOrderedHashtable.
     * @return the value to which the key is mapped in this hashtable; null if
     * the key is not mapped to any value in this hashtable.
     */
```


(C) Copyright Nokia 2013. All rights reserved.

```
    synchronized public Object get(Object key) {
        return hashTable.get(key);
    }

    /**
     * Returns an enumeration of the keys in this SimpleOrderedHashtable.
     * @return an enumeration of the keys in this SimpleOrderedHashtable.
     */
    synchronized public Enumeration keys() {
        return orderedKeys.elements();
    }

    /**
     * Returns an enumeration of the elements in this SimpleOrderedHashtable.
     * @return an enumeration of the elements in this SimpleOrderedHashtable.
     */
    synchronized public Enumeration elements() {
        int s = hashTable.size();
        Vector elements = new Vector(s);
        for (int i=0; i&lt;s; i++) {
            elements.addElement(elementAt(i));
        }
        return elements.elements();
    }

    /**
     * Returns the component at the specified index.
     * @param index is an index into this SimpleOrderedHashtable.
     * @return the &lt;code&gt;Object&lt;/code&gt; component at the specified index.
     * @throws ArrayIndexOutOfBoundsException if index is out of bounds.
     */
    synchronized public Object elementAt(int index)
    throws ArrayIndexOutOfBoundsException {
        Object key = orderedKeys.elementAt(index);
        return hashTable.get(key);
    }

    /**
     * Returns the key at the specified index.
     * @param index is an index into this SimpleOrderedHashtable.
     * @return the &lt;code&gt;Object&lt;/code&gt; key at the specified index.
     * @throws ArrayIndexOutOfBoundsException if index is out of bounds.
     */
    synchronized public Object keyAt(int index)
    throws ArrayIndexOutOfBoundsException {
        return orderedKeys.elementAt(index);
    }

    /**
     * Returns the index of the specified &lt;code&gt;Object&lt;/code&gt;.
     * @param key is a key in the SimpleOrderedHashtable.
     * @return the index of the specified &lt;code&gt;Object&lt;/code&gt;.
     */
    synchronized public int getIndex(Object key) {
        return orderedKeys.indexOf(key);
    }

    /**
     * Removes the key (and its corresponding value) from this hashtable. This
     * method does nothing if the key is not in the hashtable.
     * @param key is the key that needs to be removed.
     */
    synchronized public void remove(Object key) {
        orderedKeys.removeElement(key);
        hashTable.remove(key);
    }

    /**
     * Removes an element at the specified index.
     * @param i is the index of the element to remove.
     */
    synchronized public void removeElementAt(int i) {
        Object key = orderedKeys.elementAt(i);
        orderedKeys.removeElementAt(i);
        hashTable.remove(key);
    }

    /**
     * Clears this SimpleOrderedHashtable so that it contains no keys.
     */
    synchronized public void clear() {
        orderedKeys.removeAllElements();
        hashTable.clear();
    }

    /**
     * Returns the number of components in this SimpleOrderedHashtable.
     * @return the number of components in this vector.
     */
    synchronized public int size() {
        return orderedKeys.size();
    }

    /**
     * Recomputes the SimpleOrderedHashtable capacity.
     * @param capacity is the capacity to ensure.
     */
    synchronized public void ensureCapacity(int capacity) {
        orderedKeys.ensureCapacity(capacity);
    }
}
```


(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.