**Collections**

**What is a collection?**

A collection is an object that holds a group of objects. Some collections

* allow duplicate elements (and others do not)
* are ordered (and others are unordered)
* restrict access to elements according to certain rules (and others allow unrestricted access)

The primary kinds of collections are:

|  |  |
| --- | --- |
| Set | a group of unordered objects without duplicates |
| Bag (Multiset) | a group of unordered objects allowing duplicates |
| List (Sequence) | a sequence of objects, with a distinguished first object, and in which each object has a unique successor. Various kinds of restricted sequences can be defined, such as stacks, queues, priority queues and deques |
| Map (Dictionary) | essentially a set of (key,value) pairs with unique keys, supporting a lookup operation to find the value associated with a given key |
| Multimap | essentially a bag of pairs |
| Hierarchy (Oriented Tree) | a group of object in which every object except a single distinguished **root** node has a single parent. You can distinguish between ordered and unordered hierarchies by considering the children of an object to be lists or sets, respectively |
| Graph (Network) | a group of objects in which each object is "related to," somehow or other, to zero or more members of the group |

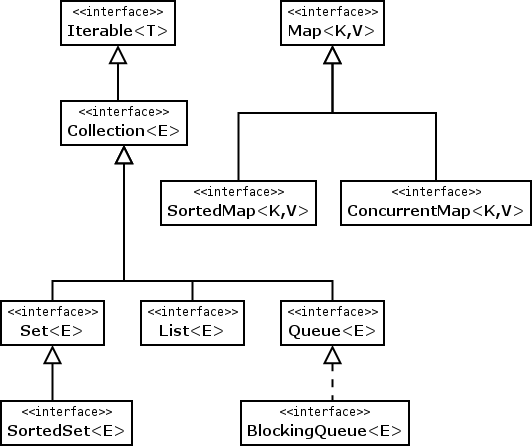
**Collections in Java**

The Java Core API comes loaded with collection interfaces, general purpose implementations, legacy implementations (Vector and Hashtable), wrapper implementations, convenience implementations, abstract implementations, algorithms, infrastructure and array utilities.

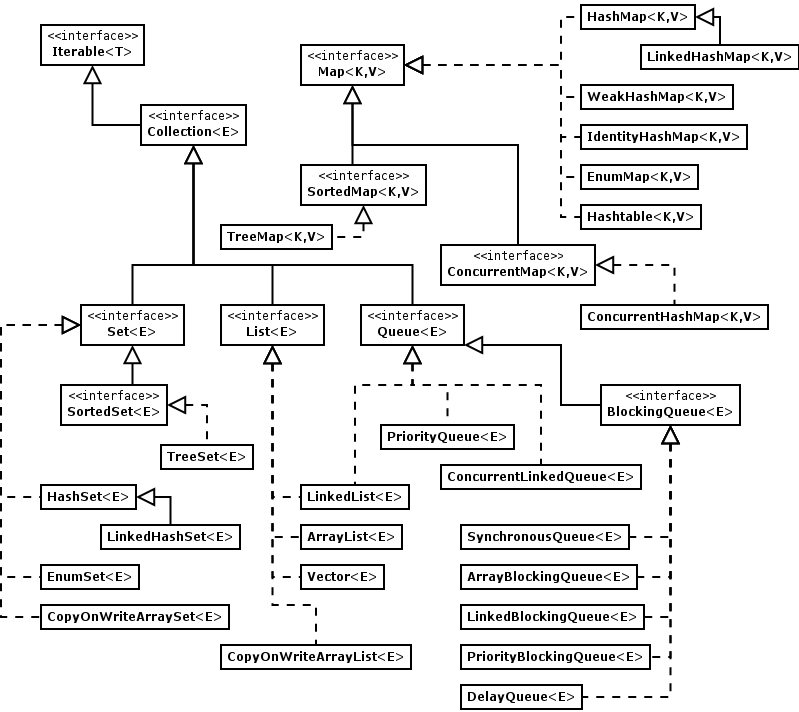
Sun provides some good documentation on the Java Collections Framework:

* [Overview](http://java.sun.com/j2se/1.5.0/docs/guide/collections/overview.html)
* [Annotated Outline](http://java.sun.com/j2se/1.5.0/docs/guide/collections/reference.html)
* [Tutorial](http://java.sun.com/docs/books/tutorial/collections/index.html)

Start by learning the interfaces:



The collection interfaces and a few of the implementing classes are summarized below for convenience:



**The Java Collection Interfaces**

**Interface java.util.Iterable**

If something is iterable, you can obtain an iterator for it, and use it as the target of a foreach statement.

public interface Iterable<T> {

Iterator<T> iterator();

}

**Interface java.util.Collection**

A collection is something that can be added to and removed from; you can test membership of items; you can ask for its size and whether it is empty. You can dump its items into an array.

public interface Collection<E> extends Iterable<E> {

Iterator<E> iterator();

int size();

boolean isEmpty();

boolean contains(Object o);

boolean add(E e);

boolean remove(Object element);

boolean containsAll(Collection<?> c);

boolean addAll(Collection<? extends E> c);

boolean removeAll(Collection<?> c);

boolean retainAll(Collection<?> c);

void clear();

Object[] toArray();

<T> T[] toArray(T[] a);

}

**Interface java.util.Set**

A set is a collection that contains no duplicate elements. More formally, sets contain no pair of elements e1 and e2 such that e1.equals(e2), and at most one null element.

public interface Set<E> extends Collection<E> {

}

**Interface java.util.SortedSet**

A sorted set is a set that further guarantees that its iterator will traverse the set in ascending element order, sorted according to the natural ordering of its elements, or by a comparator provided at sorted set creation time.

public interface SortedSet<E> extends Set<E> {

Comparator<? super E> comparator();

E first();

E last();

SortedSet<E> subSet(E fromElement, E toElement);

SortedSet<E> headSet(E toElement);

SortedSet<E> tailSet(E fromElement);

}

**Interface java.util.List**

A list is an ordered collection (also known as a sequence). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

public interface List<E> extends Collection<E> {

E get(int index);

E set(int index, E e);

void add(int index, E e);

E remove(int index);

boolean addAll(int index, Collection<? extends E> c);

int indexOf(Object o);

int lastIndexOf(Object o);

ListIterator<E> listIterator();

ListIterator<E> listIterator(int index);

List<E> subList(int from, int to);

}

**Interface java.util.Map**

A map is an object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value. The Map interface provides three collection views, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. The order of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the TreeMap class, make specific guarantees as to their order; others, like the HashMap class, do not.

public interface Map<K,V> {

boolean containsKey(Object key);

boolean containsValue(Object value);

V put(K key, V value);

V get(Object key);

V remove(Object key);

void putAll(Map<? extends K, ? extends V> t);

void clear();

public Set<K> keySet();

public Collection<V> values();

Set<Map.Entry<K, V>> entrySet();

public interface Entry<K,V> {

K getKey();

V getValue();

V setValue(V value);

}

}

**Interface java.util.SortedMap**

A sorted map is a map that further guarantees that it will be in ascending key order, sorted according to the natural ordering of its keys, or by a comparator provided at sorted map creation time. This order is reflected when iterating over the sorted map's collection views (returned by the entrySet, keySet and values methods).

public interface SortedMap<K, V> extends Map<K, V> {

Comparator<? super E> comparator();

K firstKey();

K lastKey();

SortedMap<K,V> subMap(K fromKey, K toKey);

SortedMap<K,V> headMap(K toKey);

SortedMap<K,V> tailMap(K fromKey);

}

**Interface java.util.concurrent.ConcurrentMap**

This interface includes four methods that are supposed to execute atomically.

public interface ConcurrentMap<K, V> extends Map<K, V> {

V putIfAbsent(K key, V value);

boolean remove(Object key, Object value);

boolean replace(K key, V oldValue, V newValue);

V replace(K key, V value);

}

**Interface java.util.Queue**

A queue is a place to store things until you need them later. Different kinds of queues will impose different ordering restrictions, like FIFO, LIFO, or priority-first.

public interface Queue<E> extends Collection<E> {

boolean offer(E o); // adds if possible

E poll(); // retrieve and remove head (return null if empty)

E remove(); // retrieve and remove head (throw if empty)

E peek(); // retrieve but do not remove head (return null if empty)

E element(); // retrieve but do not remove head (throw if empty)

}

**Interface java.util.concurrent.BlockingQueue**

A blocking queue is a queue with operations that wait for the queue to become non-empty when retrieving an element, and that wait for space to become available in the queue when storing an element.

public interface BlockingQueue<E> extends Queue<E> {

boolean offer(E o, long timeout, TimeUnit unit) throws InterruptedException;

void put(E o) throws InterruptedException;

boolean add(E e);

E poll(long timeout, TimeUnit unit) throws InterruptedException;

E take() throws InterruptedException;

int drainTo(Collection<? super E> c);

int drainTo(Collection<? super E> c, int maxElements);

int remainingCapacity();

}

**The Java Collection Classes**

Here's a description of most of them, anyway:

|  |  |  |  |
| --- | --- | --- | --- |
| **Class** | **Impl's** | **Thread- safe?** | **Description** |
| HashSet | Set | No | A simple set that stores elements based on their hash codes. If the hashCode() function is good, the add,remove, and contains methods should run in near constant time. |
| LinkedHashSet | Set | No | A HashSet whose values, when you iterate over them, come out in the same order you put them in. |
| EnumSet | Set | No | A set that can only contain enum values of a given type. This is a very high-performance set (backed by a bit-vector). |
| TreeSet | SortedSet | No | A set whose values, when you iterate over them, come out in sorted order. A red-black tree is used behind the scenes. |
| ArrayList | List | No | List implemented with a resizable array. |
| LinkedList | List Queue | No | Doubly-linked list implementation. Good when insertions and deletions are frequent. Access via the Queue interface gives you a FIFO queue. |
| Vector | List | Yes | The ancient threadsafe list from Java 1.0. Has some old "legacy methods" you don't need. Not much different from Collections.synchronizedList(new ArrayList()) except that (as of Java 1.5 anyway) Vectors double their size when they need more room and ArrayLists grow their size by 50% when they need more room. |
| HashMap | Map | No | Good general-purpose map that hashes on element keys. Nulls are supported for keys and values. |
| LinkedHashMap | Map | No | A HashMap whose entries are iterated over in the order they were inserted. |
| WeakHashMap | Map | No | A hash map with weak keys — when a key object isn't referenced by anything except the key reference in the map, the entry will get removed. |
| IdentityHashMap | Map | No | A map that uses ==, rather than Object.equals() on its keys. Runs fast. |
| EnumMap | Map | No | A map whose keys can only be enum values of a given type. This is a very high-performance map (backed by an array). |
| Hashtable | Map | Yes | The ancient thread-safe map from Java 1.0. No need to use it anymore; use a ConcurrentHashMap if you need thread safety, or HashMap if you do not. |
| TreeMap | SortedMap | No | A map whose keys, when you iterate over them, come out in sorted order. A red-black tree is used behind the scenes. |
| PriorityQueue | Queue | No | An unbounded priority queue implemented with a heap. It uses the element's natural ordering or a comparator passed in at construction time. The head element is the smallest. offer(), poll(), remove() and add run in logarithmic time. remove(Object) runs in linear time. peek, element, and size run in constant time. |
| ConcurrentLinkedQueue | Queue | Yes | Thread-safe, unbounded FIFO queue, optimized for use in situations where several threads are inserting and deleting items. Null elements are not allowed. |
| ArrayBlockingQueue | Queue | Yes | A bounded, non-extensible, blocking FIFO queue. |
| LinkedBlockingQueue | Queue | Yes | A blocking FIFO queue of fixed size. The size can be set at creation time; it defaults to Integer.MAX\_SIZE. Super large bounds are okay because the elements are dynamically created as needed. |
| PriorityBlockingQueue | Queue | Yes | Essentially an unbounded priority queue with blocking semantics. Here unbounded means that there will be blocking when trying to remove from an empty queue, but the queue really only fills up when system resources are exhausted and an OutOfMemoryError is thrown. Nulls are not allowed. |
| DelayQueue | Queue | Yes | An unbounded, blocking queue, whose elements must all implement the Delayed interface. An element can only be taken from the queue if its delay has expired. The head of the queue is the one whose delay expired furthest in the past. |
| SynchronousQueue | Queue | Yes | Here's the Javadoc description from Sun: [A Synchronous queue is a] blocking queue in which each put must wait for a take, and vice versa. A synchronous queue does not have any internal capacity, not even a capacity of one. You cannot peek at a synchronous queue because an element is only present when you try to take it; you cannot add an element (using any method) unless another thread is trying to remove it; you cannot iterate as there is nothing to iterate. The head of the queue is the element that the first queued thread is trying to add to the queue; if there are no queued threads then no element is being added and the head is null. For purposes of other Collection methods (for example contains), a SynchronousQueue acts as an empty collection. This queue does not permit null elements. |

**Iteration**

There are basically two ways two iterate over a collection: use a foreach statement or use an iterator object.

Set<Dog> kennel = new HashSet<Dog>();

// The foreach way

for (Dog d : kennel) {

d.bark();

}

// The explicit iterator way

Iterator<Dog> it = kennel.iterator();

while (it.hasNext()) {

Dog d = it.next();

d.bark();

}

// or if you like for loops and compact code

for (Iterator<Dog> i = kennel.iterator(); i.hasNext();) {

i.next().bark();

}

// or without type safety...

for (Iterator i = kennel.iterator(); i.hasNext();) {

((Dog)i.next()).bark();

}

What's that iterator thingy? There's an interface in java.util:

public interface Iterator<E> {

boolean hasNext();

E next();

void remove();

}

The remove() operation is in there to make it safe to update a collection while an iteration is in progress. If you remove an element using a method on the collection, the next use of the iterator will throw a ConcurrentModificationException:

for (Iterator<Dog> i = kennel.iterator(); i.hasNext();) {

Dog d = i.next();

if (d.getBreed() == Dog.POODLE) {

kennel.remove(d); // **NOOOOOOOOOOOOOOO!**

i.remove(); // Yes, do it this way

}

}

*If you use a foreach statement, an iterator is still created for you, but you cannot access it through your code! (So you'll not be able to call remove, among other things.) Therefore if you need to mutate the list during traversal (or if you need to iterate over two or more collections at a time, use iterators explicitly)*

If you know you are using a list, you can get a list iterator instead — it's got more features

package java.util;

public interface ListIterator<E> extends Iterator<E> {

boolean hasNext();

E next();

boolean hasPrevious();

E previous();

int nextIndex();

int previousIndex();

void remove();

void set(E o);

void add(E o);

}

**Ordering**

If you tried to sort a list of strings with

Collections.sort(myStringList);

the list would indeed get sorted. But if you created an Employee class list this:

class Employee {

private String id;

private Date hireDate;

private Date birthDate;

private BigDecimal salary;

private Department department;

.

.

.

}

and wrote

Collections.sort(myEmployeeList);

you'd get a compile-time error. To use the one-argument version of sort your class has to implementjava.util.Comparable. This interface has a method compareTo which defines the **natural order** on elements of that type. The invocation

x.compareTo(y)

returns a negative integer if x is "less than" y, zero if they are "equal", and a positive integer if x is "greater than" y. Many classes in the Java Core API already implement Comparable, such as String, Date, Integer, BigInteger, Time, TimeUnit, Enum, Long, Calendar, Charset, and many others.

But for employees, you may want to sort them by name, or by age, or by salary, or, well several other dimensions. So the kind of calls you would make would look like

Collections.sort(myEmployeeList, ageComparator);

where ageComparator is an object implementing the Comparator interface. Comparator has one method, called compare; the invocation

c.compare(x, y)

returns a negative integer if x is "less than" y, zero if they are "equal", and a positive integer if x is "greater than" y.

Here is a simple application illustrating how you make a class that both implements Comparable and defines additional comparators.

OrderingDemo.java

package edu.lmu.cs.collections;  
  
import java.text.DateFormat;  
import java.text.SimpleDateFormat;  
import java.util.Arrays;  
import java.util.Collections;  
import java.util.Comparator;  
import java.util.Date;  
import java.util.List;  
  
*/\*\*  
 \* A short application showing how to use Comparable and Comparator.  
 \*/*  
public class OrderingDemo {  
  
    */\*\*  
     \* A dog has a name, birthday, and breed.  Dogs are ordered naturally  
     \* by name.  
     \*/*  
    private static class Dog implements Comparable<Dog> {  
        private static DateFormat ISO\_FORMAT =  
            new SimpleDateFormat("yyyy-MM-dd");  
  
        private String name;  
        private Date birthday;  
        private String breed;  
  
        public Dog(String name, Date birthday, String breed) {  
            this.name = name;  
            this.birthday = birthday;  
            this.breed = breed;  
        }  
  
        public Date getBirthday() {  
            return birthday;  
        }  
  
        public String getBreed() {  
            return breed;  
        }  
  
        public String getName() {  
            return name;  
        }  
  
        public int compareTo(Dog that) {  
            return this.name.compareTo(that.name);  
        }  
  
        public String toString() {  
            String birthdayString = ISO\_FORMAT.format(birthday);  
            return name + "/" + breed + "(b. " + birthdayString + ")";  
        }  
    }  
  
    private static Comparator<Dog> breedComparator = new Comparator<Dog>() {  
        public int compare(Dog x, Dog y) {  
            return x.getBreed().compareTo(y.getBreed());  
        }  
    };  
  
    private static List<Dog> dogs =  
        Arrays.asList(new Dog[] {  
                new Dog("Spot", new Date(), "Chihuahua"),  
                new Dog("Sugar", new Date(), "Pit Bull"),  
                new Dog("Killer", new Date(), "Poodle"),  
                new Dog("Jack", new Date(), "Bull Massive")  
        });  
  
    public static void main(String[] args) {  
        System.out.println("Sorted naturally");  
        Collections.sort(dogs);  
        System.out.println(dogs);  
  
        System.out.println("Sorted naturally in reverse");  
        Collections.sort(dogs, Collections.reverseOrder());  
        System.out.println(dogs);  
  
        System.out.println("Sorted by breed");  
        Collections.sort(dogs, breedComparator);  
        System.out.println(dogs);  
  
        System.out.println("Sorted by breed in reverse");  
        Collections.sort(dogs, Collections.reverseOrder(breedComparator));  
        System.out.println(dogs);  
    }  
}

There are good notes on this topic in the [Java Tutorial page on Object Ordering](http://java.sun.com/docs/books/tutorial/collections/interfaces/order.html).

**Read-Only Collections**

You can get a read-only view of a collection, for example:

List<Dog> theGang = Collections.unmodifiableList(kennel);

After this, calls such as

theGang.add(bijou);

throw UnsupportedOperationException.

**More Java Stuff**

**Collections.EMPTY\_SET**

An immutable empty set.

**Collections.EMPTY\_LIST**

An immutable empty list.

**Collections.EMPTY\_MAP**

An immutable empty map.

**Collections.emptySet()**

A typesafe immutable empty set.

**Collections.emptyList()**

A typesafe immutable empty list.

**Collections.emptyMap()**

A typesafe immutable empty map.

**Collections.singleton(object)**

Returns an immutable set containing only the given object.

**Collections.singletonList(object)**

Returns an immutable list containing only the given object.

**Collections.singletonMap(key, value)**

Returns an immutable map mapping only the given key to the given value.

**Collections.nCopies(size, value)**

Returns an immutable list with the given size in which every element is the given object.

**Collections.sort(list)**

Sorts the list into ascending order according to its natural order.

**Collections.sort(list, comparator)**

Sorts the list into ascending order according to the given comparator.

**Collections.binarySearch(list, key)**

Returns the integer index of (one of) the occurrences in the list equal to key, or the negative of the position at which it would be inserted if no such object is in the list. The list must be sorted by natural order before this method is called, otherwise the result is undefined.

**Collections.binarySearch(list, key, comparator)**

Returns the integer index of (one of) the occurrences in the list equal to key, or the negative of the position at which it would be inserted if no such object is in the list. The list must be sorted according to the given comparator before this method is called, otherwise the result is undefined.

**Collections.reverse(list)**

Reverses a list.

**Collections.shuffle(list)**

Randomly permutes the list.

**Collections.shuffle(list, randomizer)**

Randomly permutes the list using the given java.util.Random object.

**Collections.swap(list, i, j)**

Swaps the elements at the given positions within the given list.

**Collections.fill(list, object)**

Replaces all of the elements in the list with the given object.

**Collections.copy(destinationList, sourceList)**

Copies the elements of the source to the corresponding positions in the destination. Throws an exception if the destination list isn't long enough. Ignores further elements if it is strictly longer.

**Collections.min(collection)**

Returns the minimum element of the collection using the natural ordering of its elements.

**Collections.min(collection, comparator)**

Returns the minimum element of the collection using the given comparator.

**Collections.max(collection)**

Returns the maximum element of the collection using the natural ordering of its elements.

**Collections.max(collection, comparator)**

Returns the maximum element of the collection using the given comparator.

**Collections.rotate(list, distance)**

Rotates elements to the right by the given distance (to rotate left make the distance negative).

**Collections.replaceAll(list, oldValue, newValue)**

Replace all occurrences of oldValue with newValue in the list.

**Collections.indexOfSubList(sourceList, targetList)**

Returns the index position of the source within the target, or -1 if there's no such occurrence.

**Collections.lastIndexOfSubList(sourceList, targetList)**

Returns the index position of the source within the target, or -1 if there's no such occurrence.

**Collections.unmodifiableCollection(collection)**

**Collections.unmodifiableSet(set)**

**Collections.unmodifiableSortedSet(sortedSet)**

**Collections.unmodifiableList(list)**

**Collections.unmodifiableMap(map)**

**Collections.unmodifiableSortedMap(sortedMap)**

Returns an unmodifiable view of the given object. Attempts to modify the returned object throw UnsupportedOperationException.

**Collections.synchronizedCollection(collection)**

**Collections.synchronizedSet(set)**

**Collections.synchronizedSortedSet(sortedSet)**

**Collections.synchronizedList(list)**

**Collections.synchronizedMap(map)**

**Collections.synchronizedSortedMap(sortedMap)**

Returns an thread-safe object backed by the given object.

**Collections.reverseOrder()**

Returns a comparator that orders in the reverse of the natural order of the elements in the collection to which it ever gets applied.

**Collections.reverseOrder(comparator)**

Returns a comparator that orders in the reverse of the given comparator.

**Collections.enumeration(collection)**

Returns an enumeration over the given collection. Useful when you have ancient Java code that expects Enumerations instead of Iterators.

**Collections.list(enumeration)**

Returns the list of elements returned by the given enumeration. Useful when you have ancient Java code that produce enumerations and you want to use this data with modern code expecting collections.

**Collections.frequency(collection, object)**

Returns the number of elements in the specified collection equal to the specified object. More formally, returns the number of elements e in the collection such that (o == null ? e == null : o.equals(e))

**Collections.disjoint(c1, c2)**

Returns whether the two collections have no elements in common.

**Collections.addAll(collection, object1, ...)**

Adds the second through last arguments to the given collection.

**Arrays.asList(a)**

Returns a fixed-size list backed by the specified array

**Arrays.fill(array, value)**

**Arrays.fill(array, fromIndex, toIndex, value)**

Assigns the specified value to each element of the specified range of the specified array of doubles.

**Arrays.sort(array)**

**Arrays.sort(array, fromIndex, toIndex)**

**Arrays.sort(array, comparator)**

**Arrays.sort(array, fromIndex, toIndex, comparator)**

Sorts the specified range of the specified array into ascending order, according to the given comparator (if present) or the natural ordering of its elements (if no comparator present).

**Arrays.binarySearch(array, key)**

**Arrays.binarySearch(array, key, comparator)**

Searches the specified array for the specified value using the binary search algorithm.

**Arrays.toString(array)**

Returns a string representation of the contents of the specified array.

**Arrays.equals(array1, array2)**

Returns whether the two specified arrays are equal to one another (having the same number of elements and all corresponding pairs of elements are equal)

**Arrays.hashCode(array)**

Returns a hash code based on the contents of the specified array

**Arrays.deepToString(array)**

Returns a string representation of the "deep contents" of the specified array

**Arrays.deepEquals(array1, array2)**

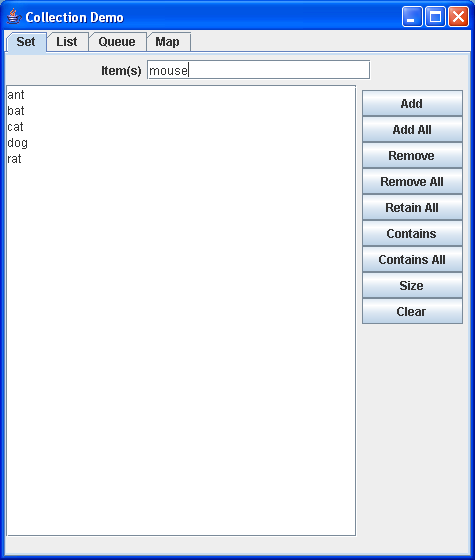
Returns true if the two specified arrays are deeply equal to one another

**Arrays.deepHashCode(array)**

Returns a hash code based on the "deep contents" of the specified array.

**A Demo**

Here is an interactive demonstration of many of the available collections. First a screenshot:



and here's the source

CollectionDemo.java

package edu.lmu.cs.collections;  
  
import java.awt.BorderLayout;  
import java.awt.Color;  
import java.awt.GridLayout;  
import java.awt.event.ActionEvent;  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.Collection;  
import java.util.Collections;  
import java.util.LinkedList;  
import java.util.List;  
import java.util.Map;  
import java.util.Queue;  
import java.util.Set;  
import java.util.TreeMap;  
import java.util.TreeSet;  
  
import javax.swing.AbstractAction;  
import javax.swing.JButton;  
import javax.swing.JFrame;  
import javax.swing.JLabel;  
import javax.swing.JPanel;  
import javax.swing.JScrollPane;  
import javax.swing.JTabbedPane;  
import javax.swing.JTextArea;  
import javax.swing.JTextField;  
import javax.swing.text.AttributeSet;  
import javax.swing.text.BadLocationException;  
import javax.swing.text.Document;  
import javax.swing.text.PlainDocument;  
  
*/\*\*  
 \* A Swing-based application hosting three demos of some of the  
 \* interfaces from the Java Collections Framework.  The application  
 \* consists of a main frame containing a tabbed pane for selecting  
 \* between a Set demo, a List demo, a Queue demo, and a Map demo.  
 \* Each of these demos are panels described separately.  
 \*/*  
@SuppressWarnings("serial")  
public class CollectionDemo extends JPanel {  
  
    protected JPanel buttonPanel = new JPanel();  
    protected JTextArea display = new JTextArea(10, 30);  
    protected JLabel message = new JLabel(" ");  
  
    */\*\*  
     \* Constructs the common part of each demo panel.  
     \*/*  
    protected CollectionDemo() {  
        setLayout(new BorderLayout());  
        display.setEditable(false);  
        add(new JScrollPane(display), BorderLayout.CENTER);  
        message.setBackground(Color.lightGray);  
        add(message, BorderLayout.SOUTH);  
        buttonPanel.setLayout(new GridLayout(0, 1));  
        JPanel buttonPanelWrapper = new JPanel();  
        buttonPanelWrapper.add(buttonPanel);  
        add(buttonPanelWrapper, BorderLayout.EAST);  
    }  
  
    */\*\*  
     \* Returns a list of the non-space substrings of a string.  
     \*/*  
    protected static Collection<String> tokensOf(String input) {  
        return Arrays.asList(input.split("\\s+"));  
    }  
  
    */\*\*  
     \* Runs the application.  
     \*/*  
    public static void main(String[] args) {  
        JTabbedPane pane = new JTabbedPane();  
        pane.addTab("Set", null, new SetDemo(), "Demo sets");  
        pane.addTab("List", null, new ListDemo(), "Demo lists");  
        pane.addTab("Queue", null, new QueueDemo(), "Demo queues");  
        pane.addTab("Map", null, new MapDemo(), "Demo maps");  
  
        JFrame frame = new JFrame("Collection Demo");  
        frame.getContentPane().add(pane, BorderLayout.CENTER);  
        frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);  
        frame.pack();  
        frame.setVisible(true);  
    }  
}  
  
*/\*\*  
 \* A panel with (1) a textarea that displays the contents of a field  
 \* which is a set, (2) a panel with buttons that invoke methods on the  
 \* set and cause the display to be updated if necessary, (3) a textfield  
 \* in which to supply arguments to the methods invoked by the buttons,  
 \* and (4) a message area in which responses to these methods may be  
 \* displayed.  
 \*/*  
class SetDemo extends CollectionDemo {  
  
    Set<String> theSet = new TreeSet<String>();  
    JTextField input = new JTextField(20);  
  
    *// We will have a lot of buttons that each run a test.  We have a command*  
    *// object which encapsulates a button and its listener.   he idea is that*  
    *// every time you make a command, it creates a button and puts it in a panel*  
    *// and attaches a listener which first reads the inputs, then calls a*  
    *// button specific method with those inputs, then displays the resulting*  
    *// set in the display area.  If an exception is thrown, the exception will*  
    *// be written to the message area.*  
  
    private abstract class Command {  
        public Command(String label) {  
            buttonPanel.add(new JButton(new AbstractAction(label) {  
                public void actionPerformed(ActionEvent event) {  
                    try {  
                        String item = input.getText();  
                        message.setText(" ");  
                        test(item);  
                        display.setText("");  
                        for (String s : theSet) {  
                            display.append(s + "\n");  
                        }  
                    } catch (Exception e) {  
                        message.setText(e.toString());  
                    }  
                }  
            }));  
        }  
        protected abstract void test(String item);  
    }  
  
    */\*\*  
     \* Do layout and other initialization, like coloring some components and  
     \* making the display area non-editable. It also creates all the "commands."  
     \*/*  
    public SetDemo() {  
        JPanel inputPanel = new JPanel();  
        inputPanel.add(new JLabel("Item(s)"));  
        inputPanel.add(input);  
        add(inputPanel, "North");  
  
        new Command("Add") {  
            protected void test(String item) {  
                theSet.add(item);}};  
        new Command("Add All") {  
            protected void test(String item) {  
                theSet.addAll(tokensOf(item));}};  
        new Command("Remove") {  
            protected void test(String item) {  
                theSet.remove(item);}};  
        new Command("Remove All") {  
            protected void test(String item) {  
                theSet.removeAll(tokensOf(item));}};  
        new Command("Retain All") {  
            protected void test(String item) {  
                theSet.retainAll(tokensOf(item));}};  
        new Command("Contains") {  
            protected void test(String item) {  
                message.setText(theSet.contains(item) + "");}};  
        new Command("Contains All") {  
            protected void test(String item) {  
                message.setText(theSet.containsAll(tokensOf(item)) + "");}};  
        new Command("Size") {  
            protected void test(String item) {  
                message.setText(theSet.size() + " item(s) in the set");}};  
        new Command("Clear") {  
            protected void test(String item) {  
                theSet.clear();}};  
    }  
}  
  
*/\*\*  
 \* A panel with (1) a textarea that displays the contents of a field  
 \* which is a queue, (2) a panel with buttons that invoke methods on the  
 \* queue and cause the display to be updated if necessary, (3) a textfield  
 \* in which to supply arguments to the methods invoked by the buttons,  
 \* and (4) a message area in which responses to these methods may be  
 \* displayed.  
 \*/*  
@SuppressWarnings("serial")  
class QueueDemo extends CollectionDemo {  
  
    Queue<String> theQueue = new LinkedList<String>();  
    JTextField input = new JTextField(20);  
  
    *// We will have a lot of buttons that each run a test.  We have a command*  
    *// object which encapsulates a button and its listener.   he idea is that*  
    *// every time you make a command, it creates a button and puts it in a panel*  
    *// and attaches a listener which first reads the inputs, then calls a*  
    *// button specific method with those inputs, then displays the resulting*  
    *// set in the display area.  If an exception is thrown, the exception will*  
    *// be written to the message area.*  
  
    private abstract class Command {  
        public Command(String label) {  
            buttonPanel.add(new JButton(new AbstractAction(label) {  
                public void actionPerformed(ActionEvent event) {  
                    try {  
                        String item = input.getText();  
                        message.setText(" ");  
                        test(item);  
                        display.setText("");  
                        for (String s : theQueue) {  
                            display.append(s + "\n");  
                        }  
                    } catch (Exception e) {  
                        message.setText(e.toString());  
                    }  
                }  
            }));  
        }  
        protected abstract void test(String item);  
    }  
  
    */\*\*  
     \* Do layout and other initialization, like coloring some components and  
     \* making the display area non-editable. It also creates all the "commands."  
     \*/*  
    public QueueDemo() {  
        JPanel inputPanel = new JPanel();  
        inputPanel.add(new JLabel("Item(s)"));  
        inputPanel.add(input);  
        add(inputPanel, "North");  
  
        new Command("Add") {  
            protected void test(String item) {  
                theQueue.add(item);}};  
        new Command("Add All") {  
            protected void test(String item) {  
                theQueue.addAll(tokensOf(item));}};  
        new Command("Offer") {  
            protected void test(String item) {  
                theQueue.offer(item);}};  
        new Command("Poll") {  
            protected void test(String item) {  
                message.setText(theQueue.poll() + " returned");}};  
        new Command("Remove()") {  
            protected void test(String item) {  
                message.setText(theQueue.remove() + " returned");}};  
        new Command("Peek") {  
            protected void test(String item) {  
                message.setText(theQueue.peek() + " returned");}};  
        new Command("Element") {  
            protected void test(String item) {  
                message.setText(theQueue.element() + " returned");}};  
        new Command("Remove(x)") {  
            protected void test(String item) {  
                theQueue.remove(item);}};  
        new Command("Remove All") {  
            protected void test(String item) {  
                theQueue.removeAll(tokensOf(item));}};  
        new Command("Retain All") {  
            protected void test(String item) {  
                theQueue.retainAll(tokensOf(item));}};  
        new Command("Contains") {  
            protected void test(String item) {  
                message.setText(theQueue.contains(item) + "");}};  
        new Command("Contains All") {  
            protected void test(String item) {  
                message.setText(theQueue.containsAll(tokensOf(item)) + "");}};  
        new Command("Size") {  
            protected void test(String item) {  
                message.setText(theQueue.size() + " item(s) in the queue");}};  
        new Command("Clear") {  
            protected void test(String item) {  
                theQueue.clear();}};  
    }  
}  
  
*/\*\*  
 \* A panel with (1) a textarea that displays the contents of a field  
 \* which is a list, (2) a panel with buttons that invoke methods on the  
 \* list and cause the display to be updated if necessary, (3) textfields  
 \* in which to supply arguments to the methods invoked by the buttons,  
 \* and (4) a message area in which responses may be displayed.  
 \*/*  
class ListDemo extends CollectionDemo {  
  
    List<String> theList = new ArrayList<String>();  
    JTextField indexInput = new NumericTextField(5);  
    JTextField itemInput = new JTextField(20);  
  
    *// We will have a lot of buttons that each run a test.  We have a command*  
    *// object which encapsulates a button and its listener.  The idea is that*  
    *// every time you make a command, it creates a button and puts it in a panel*  
    *// and attaches a listener which first reads the inputs, then calls a*  
    *// button specific method with those inputs, then displays the resulting*  
    *// list in the display area.  If an exception is thrown, the exception will*  
    *// be written to the message area.*  
  
    private abstract class Command {  
        public Command(String label) {  
            buttonPanel.add(new JButton(new AbstractAction(label) {  
                public void actionPerformed(ActionEvent event) {  
                    try {  
                        String index = indexInput.getText();  
                        String item = itemInput.getText();  
                        message.setText(" ");  
                        test(index, item);  
                        display.setText("");  
                        for (int i = 0; i < theList.size(); i++) {  
                            display.append("[" + i + "]    "  
                                    + theList.get(i) + "\n");  
                        }  
                    } catch (Exception e) {  
                        message.setText(e.toString());  
                    }  
                }  
            }));  
        }  
        protected abstract void test(String index, String item);  
    }  
  
    */\*\*  
     \* Do layout and other initialization, like coloring some  
     \* components and making the display area non-editable.    It  
     \* also creates all the "testers."  
     \*/*  
    public ListDemo() {  
        JPanel inputPanel = new JPanel();  
        inputPanel.add(new JLabel("Index"));  
        inputPanel.add(indexInput);  
        inputPanel.add(new JLabel("Item(s)"));  
        inputPanel.add(itemInput);  
        add(inputPanel, BorderLayout.NORTH);  
  
        new Command("Add") {  
            protected void test(String index, String item) {  
                theList.add(item);}};  
        new Command("Add at Index") {  
            protected void test(String index, String item) {  
                theList.add(Integer.parseInt(index), item);}};  
        new Command("Add All") {  
            protected void test(String index, String item) {  
                theList.addAll(tokensOf(item));}};  
        new Command("Set") {  
            protected void test(String index, String item) {  
                theList.set(Integer.parseInt(index), item);}};  
        new Command("Get") {  
            protected void test(String index, String item) {  
                message.setText(theList.get(Integer.parseInt(index)));}};  
        new Command("IndexOf") {  
            protected void test(String index, String item) {  
                message.setText(theList.indexOf(item) + "");}};  
        new Command("Remove") {  
            protected void test(String index, String item) {  
                theList.remove(item);}};  
        new Command("Remove All") {  
            protected void test(String index, String item) {  
                theList.removeAll(tokensOf(item));}};  
        new Command("Retain All") {  
            protected void test(String index, String item) {  
                theList.retainAll(tokensOf(item));}};  
        new Command("Contains") {  
            protected void test(String index, String item) {  
                message.setText(theList.contains(item) + "");}};  
        new Command("Contains All") {  
            protected void test(String index, String item) {  
                message.setText(theList.containsAll(tokensOf(item)) + "");}};  
        new Command("Reverse") {  
            protected void test(String index, String item) {  
                Collections.reverse(theList);}};  
        new Command("Fill") {  
            protected void test(String index, String item) {  
                Collections.fill(theList, item);}};  
        new Command("Shuffle") {  
            protected void test(String index, String item) {  
                Collections.shuffle(theList);}};  
        new Command("Sort") {  
            protected void test(String index, String item) {  
                Collections.sort(theList);}};  
        new Command("Size") {  
            protected void test(String index, String item) {  
                message.setText(theList.size() + " item(s) in the list");}};  
        new Command("Clear") {  
            protected void test(String index, String item) {  
                theList.clear();}};  
    }  
}  
  
*/\*\*  
 \* A panel with (1) a textarea that displays the contents of a field  
 \* which is a map, (2) a panel with buttons that invoke methods on the  
 \* map and cause the display to be updated if necessary, (3) textfields  
 \* in which to supply arguments to the methods invoked by the buttons,  
 \* and (4) a message area in which responses may be displayed.  
 \*/*  
@SuppressWarnings("serial")  
class MapDemo extends CollectionDemo {  
  
    Map<String, String> theMap = new TreeMap<String,String>();  
    JTextField keyInput = new JTextField(10);  
    JTextField valueInput = new JTextField(10);  
  
    *// We will have a lot of buttons that each run a test.  We have a command*  
    *// object which encapsulates a button and its listener.  The idea is that*  
    *// every time you make a command, it creates a button and puts it in a panel*  
    *// and attaches a listener which first reads the inputs, then calls a*  
    *// button specific method with those inputs, then displays the resulting*  
    *// list in the display area.  If an exception is thrown, the exception will*  
    *// be written to the message area.*  
  
    private abstract class Command {  
        public Command(String label) {  
            buttonPanel.add(new JButton(new AbstractAction(label) {  
                public void actionPerformed(ActionEvent event) {  
                    try {  
                        String key = keyInput.getText();  
                        String value = valueInput.getText();  
                        message.setText(" ");  
                        test(key, value);  
                        display.setText("");  
                        for (Map.Entry e : theMap.entrySet()) {  
                            display.append(e + "\n");  
                        }  
                    } catch (Exception e) {  
                        message.setText(e.toString());  
                    }  
                }  
            }));  
        }  
        protected abstract void test(String key, String value);  
    }  
  
    */\*\*  
     \* Do layout and other initialization, like coloring some  
     \* components and making the display area non-editable.    It  
     \* also creates all the "commands."  
     \*/*  
    public MapDemo() {  
        JPanel inputPanel = new JPanel();  
        inputPanel.add(new JLabel("Key"));  
        inputPanel.add(keyInput);  
        inputPanel.add(new JLabel("Value"));  
        inputPanel.add(valueInput);  
        add(inputPanel, "North");  
  
        new Command("Put") {  
            protected void test(String key, String value) {  
                theMap.put(key, value);}};  
        new Command("Get") {  
            protected void test(String key, String value) {  
                String answer = theMap.get(key);  
                message.setText(answer == null ? "NO\_SUCH\_KEY" : answer);}};  
        new Command("Remove") {  
            protected void test(String key, String value) {  
                theMap.remove(key);}};  
        new Command("Contains Key") {  
            protected void test(String key, String value) {  
                message.setText(theMap.containsKey(key) + "");}};  
        new Command("Contains Value") {  
            protected void test(String key, String value) {  
                message.setText(theMap.containsValue(value) + "");}};  
        new Command("Size") {  
            protected void test(String key, String value) {  
                message.setText(theMap.size() + " item(s) in the map");}};  
        new Command("Clear") {  
            protected void test(String key, String value) {  
                theMap.clear();}};  
    }  
}  
  
*/\*\*  
 \* A simple extension of JTextField whose instances only allow  
 \* numeric characters.  Attempts to insert non-numeric characters  
 \* into a NumericTextField quietly fail.  
 \*/*  
@SuppressWarnings("serial")  
class NumericTextField extends JTextField {  
  
    public NumericTextField(int columns) {  
        super(columns);  
    }  
  
    protected Document createDefaultModel() {  
        return new NumericDocument();  
    }  
  
    static class NumericDocument extends PlainDocument {  
        public void insertString(int offset, String string, AttributeSet a)  
                throws BadLocationException {  
            if (string == null) {  
                return;  
            }  
            for (int i = 0; i < string.length(); i++) {  
                if (!Character.isDigit(string.charAt(i))) {  
                    return;  
                }  
            }  
            super.insertString(offset, string, a);  
        }  
    }  
}