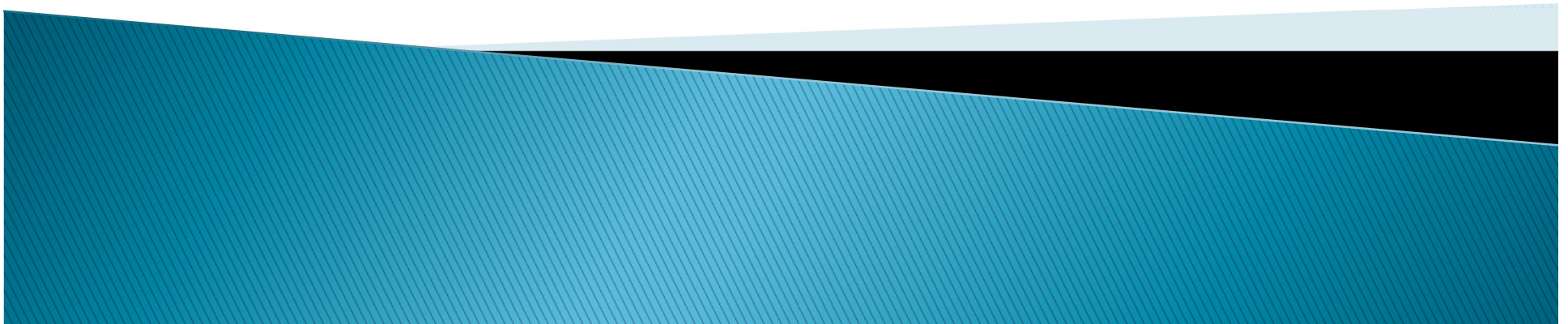


# Java Collections

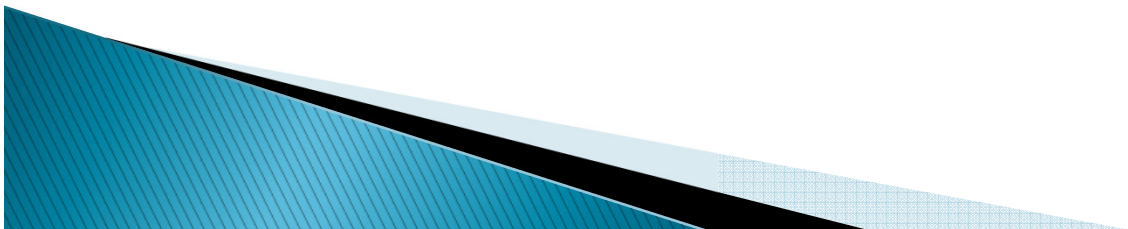
CS 15-121

Ananda Gunawardena



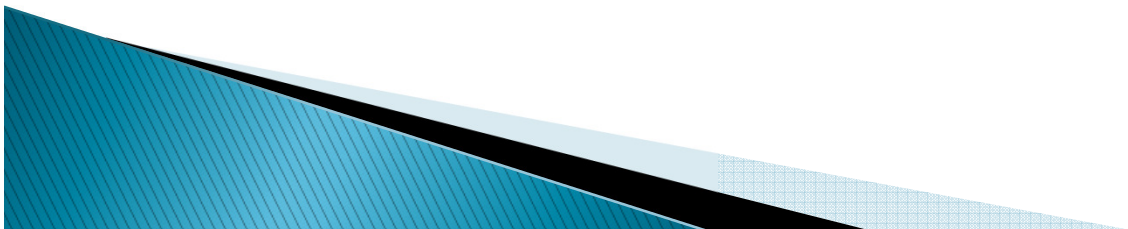
# What is a collection?

- ▶ A collection (sometimes called a *container*) is simply an object that groups multiple elements into a single unit.
- ▶ Collections are used to store, retrieve and manipulate data, and to transmit data from one method to another.
- ▶ Examples:
  - Arrays, hashtables, vector, set, map, tree



# ArrayList Collection

- ▶ ArrayList is a java collection that allows management of a dynamic list collection
- ▶ Eg:
  - `ArrayList A = new ArrayList();`
  - `A.add(new Integer(10));`
  - `System.out.println(A.get(0));`
  - `If (A.contains(new Integer(10))) { ....}`
  - `A.remove(0);`



# ArrayList API (some methods)

## Constructor Summary

[ArrayList\(\)](#)

Constructs an empty list with an initial capacity of ten.

[ArrayList\(Collection c\)](#)

Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

[ArrayList\(int initialCapacity\)](#)

Constructs an empty list with the specified initial capacity.

## Method Summary

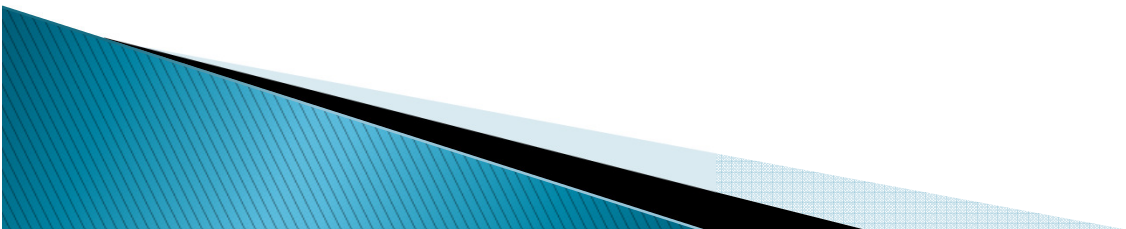
void	<u><a href="#">add(int index, Object element)</a></u> Inserts the specified element at the specified position in this list.
boolean	<u><a href="#">add(Object o)</a></u> Appends the specified element to the end of this list.
boolean	<u><a href="#">addAll(Collection c)</a></u> Appends all of the elements in the specified Collection to the end of this list, in the order that they are returned by the specified Collection Iterator.
boolean	<u><a href="#">addAll(int index, Collection c)</a></u> Inserts all of the elements in the specified Collection into this list, starting at the specified position.
void	<u><a href="#">clear()</a></u> Removes all of the elements from this list.
<u><a href="#">Object</a></u>	<u><a href="#">clone()</a></u> Returns a shallow copy of this ArrayList instance.
boolean	<u><a href="#">contains(Object elem)</a></u> Returns true if this list contains the specified element.

See others at

<http://java.sun.com/j2se/1.4.2/docs/api/java/util/ArrayList.html>

# Exercise

- ▶ Given an ArrayList A of objects write a method removeDups that removes all duplicate elements from the list

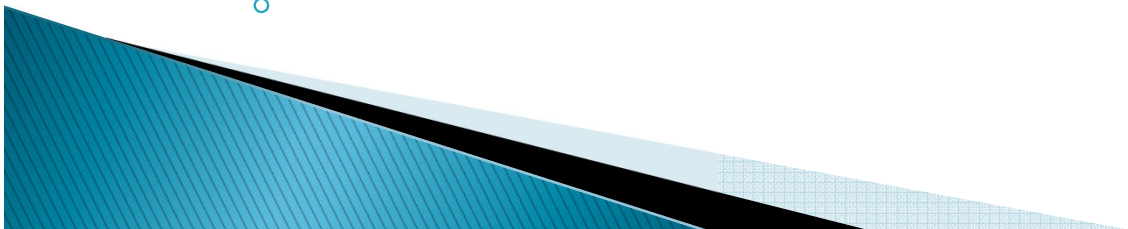


# Introduction

## ▶ The Collections Framework

- A unified architecture for representing and manipulating collections, allowing them to be manipulated independently of the details of their representation.
- Reduces programming effort while increasing performance.
- Allows for interoperability among unrelated APIs, reduces effort in designing and learning new APIs, and fosters software reuse.

◦



# Collections Framework

- ▶ **Interfaces:** abstract data types representing collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages like Java, these interfaces generally form a hierarchy.
- ▶ **Implementations:** concrete implementations of the collection interfaces. In essence, these are *reusable data structures*.
- ▶ **Algorithms:** methods that perform useful computations, like searching and sorting, on objects that implement collection interfaces. These algorithms are said to be *polymorphic* because the same method can be used on many different implementations of the appropriate collections interface. In essence, algorithms are *reusable functionality*.

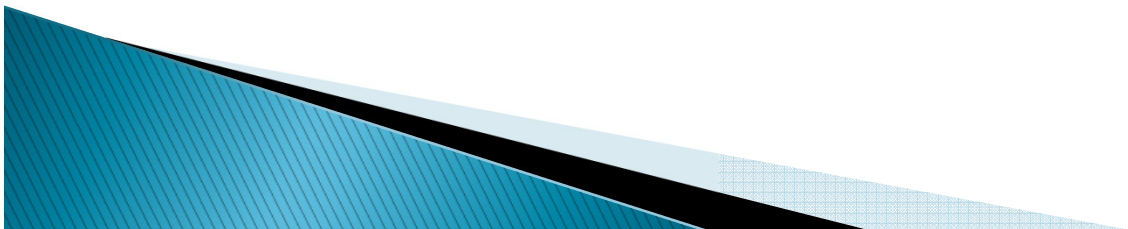




# Interfaces

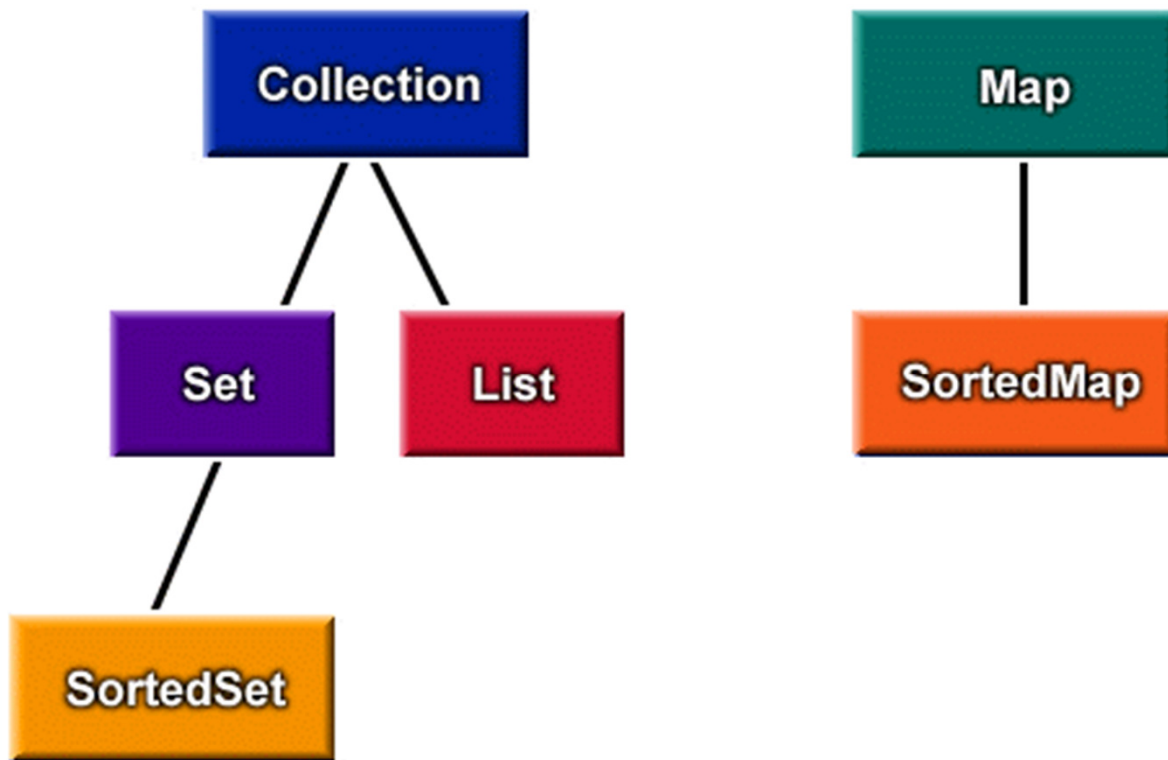
- ▶ A Java interface contains specifications for a known set of functions.
- ▶ These functions are to be implemented by classes
  - Eg: public class foo implements List { }
- ▶ Interfaces from java collections framework

Interfaces	Implementations
List	LinkedList, ArrayList
Set	HashSet, TreeSet
Map	HashMap, TreeMap, Hashtable





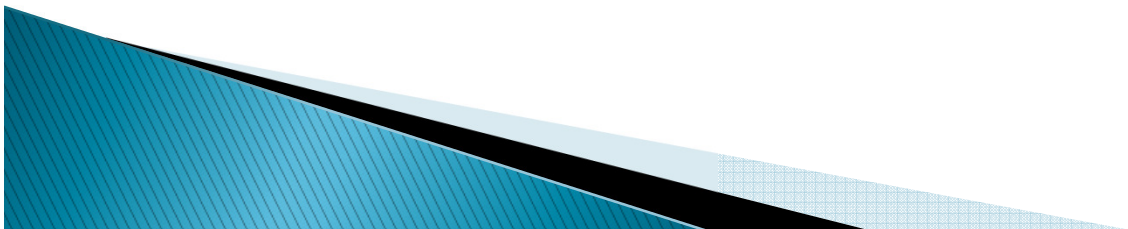
# Core Collection Interfaces



# Examples of using Collections

```
ArrayList<Integer> nums = new ArrayList<Integer> ();  
nums.add(1);  
Integer x = nums.get(0);
```

```
ArrayList nums = new ArrayList ();  
nums.add(1);  
Integer x = (Integer) nums.get(0);
```



# Creating your own generic type

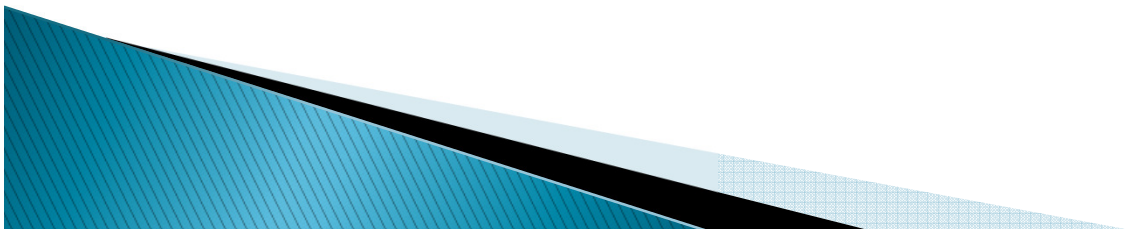
```
public class Box<AnyType>
{
    private ArrayList<AnyType> data;
    public Box()
    {
        data = new ArrayList<AnyType>();
    }
    public void add(AnyType x)
    {
        data.add(x);
    }
    public AnyType remove()
    {
        if (data.size() > 0)
            return data.remove(0);
        else
            return null;
    }
}
```

```
public class Box<AnyType>
{
    private AnyType[] contents = new AnyType[5];
    ...
}
```

Java does not allow above

```
public class Box<AnyType>
{
    private AnyType[] stack = (AnyType[]) new Object[5];
    ...
}
```

Here is a hack



# Benefits of Collections

- ▶ Reduces programming effort
- ▶ Increases program speed and quality
- ▶ Allows interoperability among unrelated APIs
- ▶ Reduces the effort to learn and use new APIs:
- ▶ Reduces effort to design new APIs
- ▶ Supports software reuse

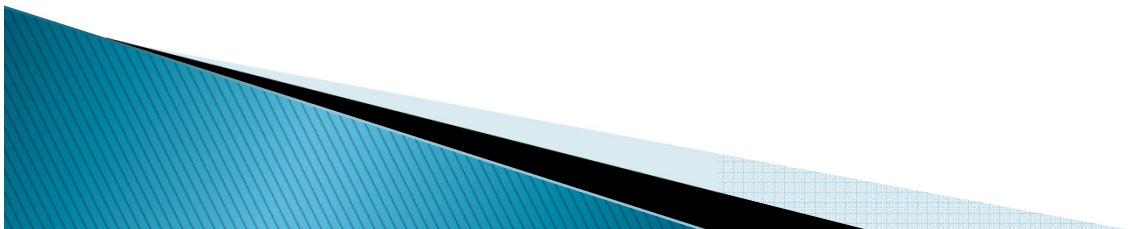


# The Set interface

- ▶ Set is a collection with NO order
- ▶ Similar to the Mathematical Abstraction of Set

```
public interface Set {  
    // Basic Operations  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(Object element);  
    // Optional  
    boolean remove(Object element);  
    Iterator iterator();  
    boolean removeAll(Collection c);  
    boolean retainAll(Collection c);  
    void clear();  
    // Bulk Operations  
    boolean containsAll(Collection c);  
    boolean addAll(Collection c);  
    //Array Operations  
    Object[] toArray(); Object[] toArray(Object a[]);  
}
```

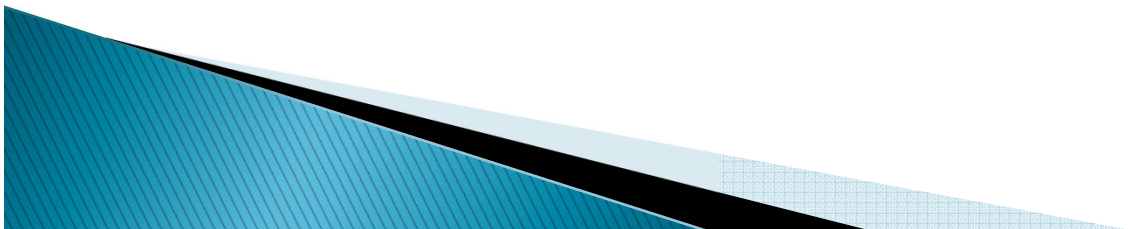
- ▶ Example: `Collections noDups = new HashSet(c );`



# Examples of using Set

```
import java.util.*;
public class FindDups {
    public static void main(String args[]) {
        Set s = new HashSet();
        for (int i=0; i<args.length; i++)
            if (!s.add(args[i]))
                System.out.println("Duplicate detected: "+args[i]);
        System.out.println(s.size()+" distinct words detected: "+s);
    }
}
```

- ▶ What is the output of : `java FindDups I am sam sam I am`



# More Set Operations

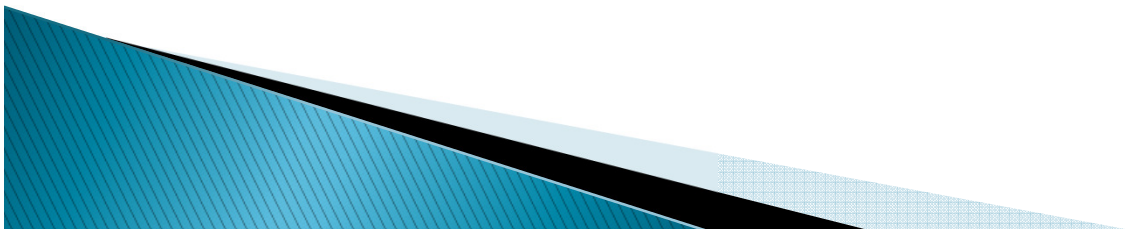
- ▶ `s1.containsAll(s2)`: Returns true if `s2` is a subset of `s1`
- ▶ `s1.addAll(s2)`: Transforms `s1` into the union of `s1` and `s2`
- ▶ `s1.retainAll(s2)`: Transforms `s1` into the intersection of `s1` and `s2`
- ▶ `s1.removeAll(s2)`: Transforms `s1` into the (asymmetric) set difference of `s1` and `s2`
- ▶ Exercise: Write a program that will read a line of words and list those ones that occur once and those that occur more than once.
  - Example: `java FindDups i am Sam Sam I am`
  - Unique words: `[i, I]`
  - Duplicate words: `[Sam, am]`
- ▶ More info at:
  - <http://java.sun.com/docs/books/tutorial/collections/interfaces/set.html>





# The List Interface

- ▶ List is an ordered Collection (sometimes called a *sequence*).
- ▶ Lists may contain duplicate elements. In addition to the operations inherited from Collection, the List interface includes operations for:
  - Index based Access: manipulate elements based on their numerical position in the list.
  - Search: search for a specified object in the list and return its index position.
  - List Iteration: extend Iterator semantics to take advantage of the list's sequential nature.
  - Range-view: perform arbitrary *range operations* on the list.



# The List Interface

public interface List extends Collection

{ **// index based Access**

Object get(int index);

Object set(int index, Object element);

**// Optional**

void add(int index, Object element);

Object remove(int index);

abstract boolean addAll(int index, Collection c); //

Search int indexOf(Object o);

int lastIndexOf(Object o);

ListIterator listIterator();

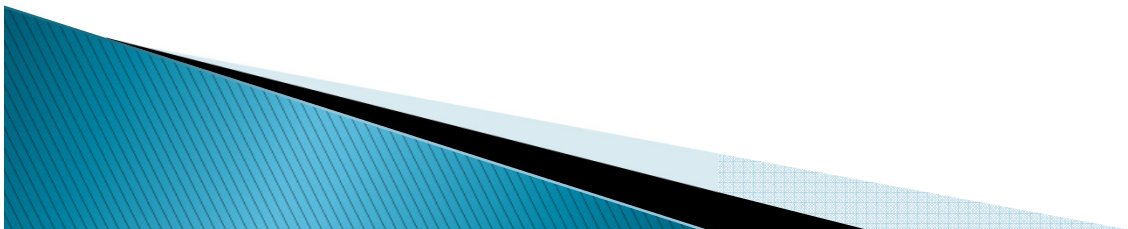
ListIterator listIterator(int index);

**// Range-view**

List subList(int from, int to); }

# List Implementation

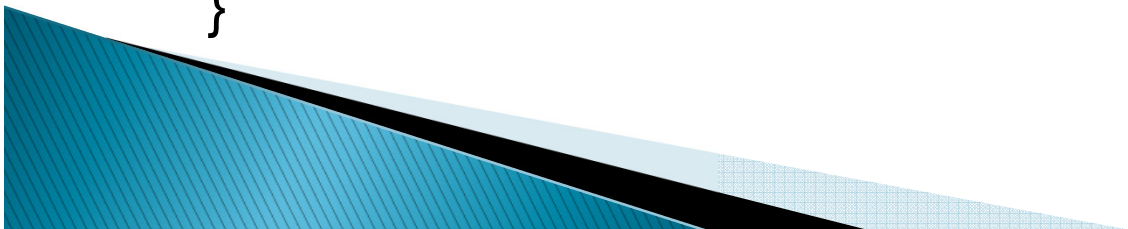
- ▶ Two important List implementations
  - ArrayList
  - LinkedList



# Iterators

- ▶ Iterators allow collection to be accessed using a pre-defined “pointer”

```
public interface ListIterator extends Iterator {  
    boolean hasNext();  
    Object next();  
    boolean hasPrevious();  
    Object previous();  
    int nextIndex();  
    int previousIndex();  
    void remove();  
    // Optional  
    void set(Object o);  
    void add(Object o);  
}
```

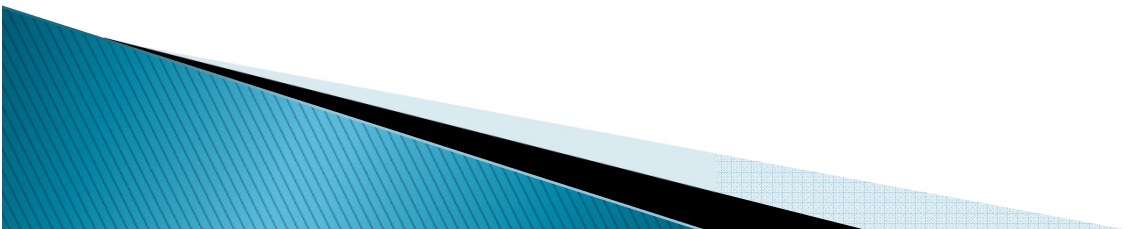


# Iterators Examples

- ▶ What is the purpose of the following code?

```
for (ListIterator i=L.listIterator(l.size()); i.hasPrevious(); )  
    { Foo f = (Foo) i.previous(); ... }
```


- Steps through a list backwards



# Map Interface

- ▶ Maps keys to values
- ▶ Each Key can map to a unique value

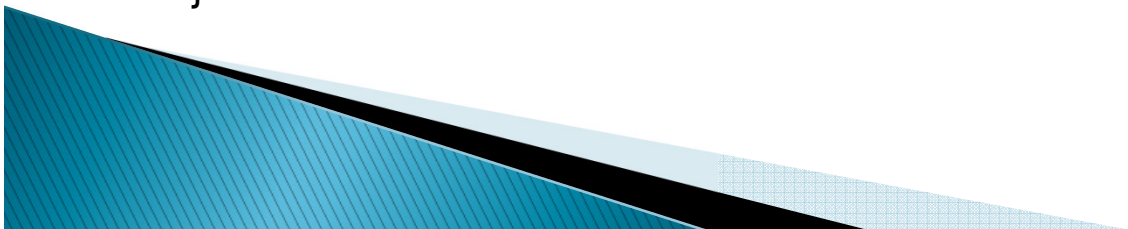
```
public interface Map {  
    // Basic Operations  
    Object put(Object key, Object value);  
    Object get(Object key);  
    Object remove(Object key);  
    boolean containsKey(Object key);  
    boolean containsValue(Object value);  
    int size();  
    boolean isEmpty();  
    // Bulk Operations  
    void putAll(Map t);  
    void clear();  
    // Collection Views  
    public Set keySet();  
    public Collection values();  
    public Set entrySet();  
    // Interface for entrySet elements  
    public interface Entry { Object getKey(); Object getValue(); Object setValue(Object value); }  
}
```



# HashTable

- ▶ HashTable is an implementation of the Map interface
- ▶ An example of generating a frequency table

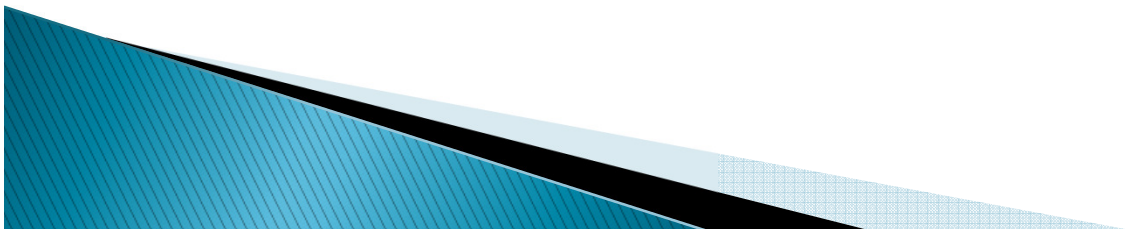
```
import java.util.*;
public class Freq {
    private static final Integer ONE = new Integer(1);
    public static void main(String args[]) {
        Map m = new HashMap();
        // Initialize frequency table from command line
        for (int i=0; i<args.length; i++) {
            Integer freq = (Integer) m.get(args[i]);
            m.put(args[i],
                (freq==null ? ONE : new Integer(freq.intValue() + 1)));
        }
        System.out.println(m.size()+" distinct words detected:");
        System.out.println(m);
    }
}
```





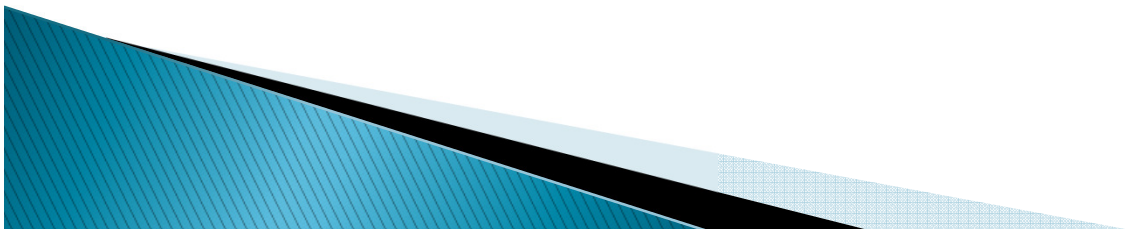
# Algorithms

- ▶ The Java provides a reusable algorithms that can operate on a List. The following algorithms are provided.
  - Sorting
  - Shuffling
  - Routine Data Manipulation
  - Searching
  - Composition
  - Finding Extreme Values



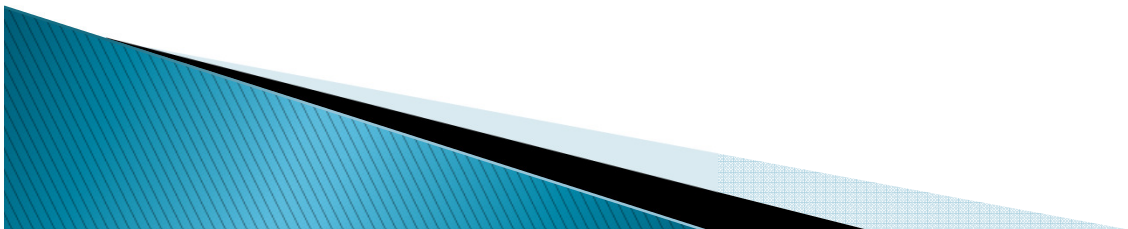
# Sorting

- ▶ A List can be sorted according to its natural ordering
- ▶ Java uses an optimized merge sort
  - Fast: runs in  $n \log(n)$  time
  - Runs faster on nearly sorted lists.
  - Stable: It doesn't reorder equal elements.



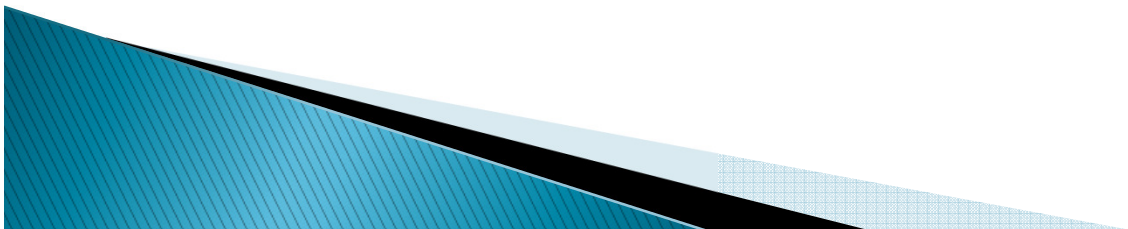
# Example

```
import java.util.*;  
public class Sort {  
    public static void main(String[] args) {  
        List list = new ArrayList();  
        list.add("guna"); list.add("bob"); //etc...  
        Collections.sort(list);  
        System.out.println(list);  
    }  
}
```



# Shuffling

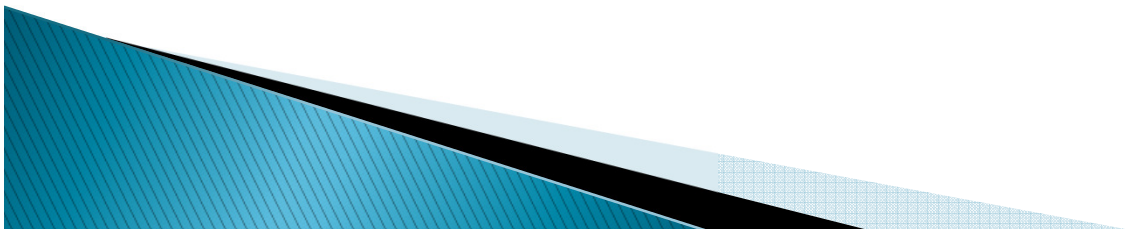
- ▶ Shuffle algorithm does the opposite of sort
- ▶ It puts elements in the list in a random order so that each element is equally likely
- ▶ Applications
  - Shuffle a card deck
- ▶ Example:
  - `Collections.shuffle(list);`



# Useful Algorithms

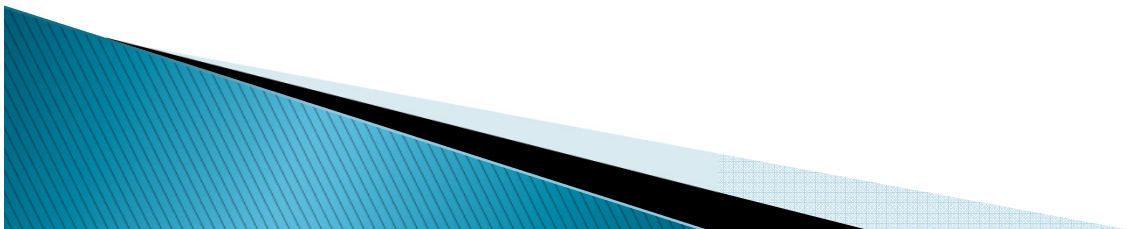
- ▶ Reverse
  - reverses the order of the elements in a List.
- ▶ Fill
  - overwrites every element in a List with the specified value. This operation is useful for reinitializing a List.
- ▶ Copy
  - takes two arguments, a destination List and a source List, and copies the elements of the source into the destination, overwriting its contents. The destination List must be at least as long as the source. If it is longer, the remaining elements in the destination List are unaffected.
- ▶ Swap
  - swaps the elements at the specified positions in a List.
- ▶ addAll
  - adds all the specified elements to a Collection. The elements to be added may be specified individually or as an array.

Source: [java.sun.com](http://java.sun.com)



# Search

- ▶ The binary search algorithm provides a way to search for a key.
- ▶ Example
  - `int pos = Collections.binarySearch(list, key);`
  - `if (pos < 0) l.add(-pos-1);`



# Composition

- ▶ Frequency
  - counts the number of times the specified element occurs in the specified collection
- ▶ Disjoint
  - determines whether two Collections are disjoint; that is, whether they contain no elements in common

