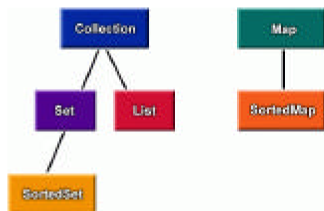
Collections

- A collection (sometimes called a *container*) is 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.
- Collections typically represent data items that form a natural group, a card hand, a mail folder, a telephone directory…

The Java Collections Framework

- The Java collections framework is made up of a set of interfaces and classes for working with groups of objects
- The Java Collections Framework provides
  - **Interfaces**: abstract data types representing collections. **Implementations** : concrete implementations of the collection interfaces.
  - **Algorithms**: methods that perform useful computations, like searching and sorting, on objects that implement collection interfaces.

The Interfaces



Note: Some of the material on these slides was taken from the Java Tutorial at http://www.java.sun.com/docs/books/tutorial

Sets

- A group of unique items, meaning that the group contains no duplicates
- Some examples
  - The set of uppercase letters ‘A’ through ‘Z’
  - The set of nonnegative integers { 0, 1, 2, … }
  - The empty set { }
- The basic properties of sets
  - Contain only one instance of each item
  - May be finite or infinite
  - Can define abstract concepts

Maps

- A map is a special kind of set.
- A map is a set of pairs, each pair representing a one-directional “mapping” from one set to another
  - An object that maps keys to values
- Some examples
  - A map of keys to database records
  - A dictionary (words mapped to meanings)
  - The conversion from base 2 to base 10

What Is The Real Difference?

- Collections
  - You can add, remove, lookup *isolated* items in the collection
- Maps
  - The collection operations are available but they work with a *key-value* pair instead of an isolated element
  - The typical use of a `Map` is to provide access to values stored by key

Another Way to Look At It

- The `Collection` interface is a group of objects, with duplicates allowed
- `Set` extends `Collection` but forbids duplicates
- `List` extends `Collection` and allows duplicates and positional indexing
- `Map` extends neither `Set` nor `Collection`

The `Collection` Interface

Found in the `java.util` package

Optional methods throw `UnsupportedOperationException` if the implementing class does not support the operation.

Bulk operations perform some operation on an entire `Collection` in a single shot

The `toArray` methods allow the contents of a `Collection` to be translated into An array.

Collection

```
// Basic Operations
size():int;
isEmpty():boolean;
contains(Object):boolean;
add(Object):boolean;           // Optional
remove(Object):boolean;        // Optional
iterator():Iterator;

// Bulk Operations
containsAll(Collection):boolean ;
addAll(Collection):boolean;    // Optional
removeAll (Collection):boolean;// Optional
retainAll (Collecton):boolean; // Optional
clear():void;                  // Optional

// Array Operations
toArray():Object[];
toArray(Object[]):Object[];
```


`Set` Interface

- A `Set` is a `Collection` that cannot contain duplicate elements.
  - Set models the mathematical *set* abstraction.
- The `Set` interface extends `Collection` and contains *no* methods other than those inherited from `Collection`
- It adds the restriction that duplicate elements are prohibited.
- Two `Set` objects are equal if they contain the same elements.

Set Bulk Operations

- The bulk operations perform standard set - algebraic operations. Suppose s1 and s2 are Sets.
  - s1.containsAll(s2): Returns true if s2 is a **subset** of s1.
  - s1.addAll(s2): Transforms s1 into the **union** of s1 and s2. (The union of two sets is the set containing all the elements contained in either set.)
  - s1.retainAll(s2): Transforms s1 into the **intersection** of s1 and s2. (The intersection of two sets is the set containing only the elements that are common in both sets.)

Java Lists

- A 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:
  - Positional Access
  - Search
  - List Iteration
  - Range-view

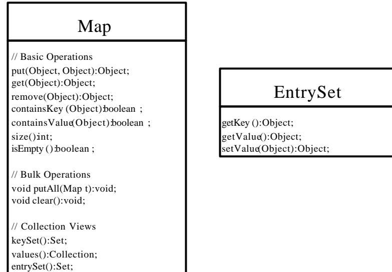
List Interface

Think of the Vector class

List

```
// Positional Access
get(int):Object;
set(int,Object):Object;             // Optional
add(int, Object):void;              // Optional
remove(int index):Object;           // Optional
addAll(int, Collection):boolean;    // Optional

// Search
int indexOf(Object);
int lastIndexOf(Object);

// Iteration
listIterator():ListIterator;
listIterator(int):ListIterator;

// Range-view List
subList(int, int):List;
```

`Map` Interface



Implementation Classes

Interface	Implementation				Historical
Set	HashSet		TreeSet		
List		ArrayList		LinkedList	Vector Stack
Map	HashMap		TreeMap		HashTable Properties

Note: When writing programs think about interfaces and not implementations. This way the program does not become dependent on any added methods in a given implementation, leaving the programmer with the freedom to change implementations.

UniqueWords

```
import java.io.*;
import java.util.*;

public class UniqueWords {
    public static void main( String args[] ) {
        // Usage check & open file
        if ( args.length != 1 ) {
            System.err.println( "Usage:  java UniqueWords word-file" );
            System.exit( 1 );
        }

        StreamTokenizer in = null;
        try {
            in = new StreamTokenizer(
                    new BufferedReader ( new FileReader( args[ 0 ] ) ) );
            in.ordinaryChar( '.' );
        }
        catch ( FileNotFoundException e ) {
            System.err.println( "UniqueWords:  " + e.getMessage() );
            System.exit( 1 );
        }
```

UniqueWords

```
        try {
            Set set = new HashSet();

            while ( ( in.nextToken() != in.TT_EOF ) ) {
                if ( in.ttype == in.TT_WORD )
                    set.add( in.sval );
            }

            System.out.println( "There are " + set.size() + " unique words" );
            System.out.println( set );
        }
        catch ( IOException e ) {
            System.err.println( "UniqueWords:  " + e.getMessage() );
            System.exit( 1 );
        }
    }
}
```

3/14/01 Java Collection Framework 16

You Want Them Sorted?

```
        try {
            SortedSet set = new TreeSet();

            while ( ( in.nextToken() != in.TT_EOF ) ) {
                if ( in.ttype == in.TT_WORD )
                    set.add( in.sval );
            }

            System.out.println( "There are " + set.size() + " unique words" );
            System.out.println( set );
        }
        catch ( IOException e ) {
            System.err.println( "UniqueWords:  " + e.getMessage() );
            System.exit( 1 );
        }
    }
}
```

3/14/01 Java Collection Framework 17

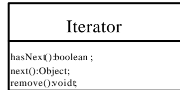
Or

```
        try {
            Set set = new HashSet();

            while ( ( in.nextToken() != in.TT_EOF ) ) {
                if ( in.ttype == in.TT_WORD )
                    set.add( in.sval );
            }

            System.out.println( "There are " + set.size() + " unique words" );
            System.out.println( new TreeSet(set) );
        }
        catch ( IOException e ) {
            System.err.println( "UniqueWords:  " + e.getMessage() );
            System.exit( 1 );
        }
    }
}
```

3/14/01 Java Collection Framework 18

Iterator

- An object that implements the `Iterator` interface generates a series of elements, one at a time
  - Successive calls to the `next()` method return successive elements of the series.
- The `remove()` method removes from the underlying Collection the last element that was returned by next.



Pretty Output

```
        try {
            SortedSet set = new TreeSet();

            while ( ( in.nextToken() != in.TT_EOF ) ) {
                if ( in.ttype == in.TT_WORD )
                    set.add( in.sval );
            }

            System.out.println( "There are " + set.size() + " unique words" );

            Iterator elements = set.iterator();

            System.out.println();
            while ( elements.hasNext() )
                System.out.println( elements.next() );
        }
        catch ( IOException e ) {
            System.err.println( "UniqueWords:  " + e.getMessage() );
            System.exit( 1 );
        }
    }
}
```


CountWords

```
        try {
            HashMap map = new HashMap();
            Integer one = new Integer( 1 );

            while ( ( in.nextToken() != in.TT_EOF ) ) {
                if ( in.ttype == in.TT_WORD ) {
                    Integer freq = ( Integer )map.get( in.sval );

                    if ( freq == null )
                        freq = one;
                    else
                        freq = new Integer( freq.intValue() + 1 );

                    map.put( in.sval, freq );
                }
            }
```

CountWords

```
        System.out.println( "There are " + map.size() + " unique words" );

        SortedMap sorted = new TreeMap( map );
        Iterator elements = sorted.entrySet().iterator();

        while ( elements.hasNext() ) {
            Map.Entry cur = ( Map.Entry )elements.next();
            System.out.println( cur.getValue() + "\t" + cur.getKey() );
        }
    }
    catch ( IOException e ) {
        System.err.println( "UniqueWords:  " + e.getMessage() );
        System.exit( 1 );
    }
```


`Collections` Class

- This class consists exclusively of static methods that operate on or return collections
- It contains
  - Polymorphic algorithms that operate on collections
  - Wrappers which return a new collection backed by a specified collection
  - Plus a a few other odds and ends.

Algorithms

- All of the algorithms, provided by the `Collections` class, take the form of static methods
  - Most of the algorithms operate on `List` objects, but a couple of them (`max` and `min`) operate on arbitrary `Collection` objects

Sorting

- The sort operation uses a slightly optimized merge sort algorithm
  - Fast: This algorithm is guaranteed to run in n log(n) time, and runs substantially faster on nearly sorted lists.
  - Stable: That is to say, it doesn't reorder equal elements.

SortExample

```
import java.util.*;

public class SortExample {
    public static void main( String args[] ) {
      List l = new ArrayList();

      for (  int i = 0; i < args.length; i++ )
          l.add( args[ i ] );

      Collections.sort( l );

      System.out.println( l );
    }
}
```


Arrays

- It is too bad that arrays are not collections
  - You loose all of the power provided by the collection framework
- The class `Arrays` contains
  - various methods for manipulating arrays (such as sorting and searching)
  - It also contains methods that allows arrays to be viewed as lists.

SortExample

```
import java.util.*;

public class SortExample {
    public static void main( String args[] ) {
      Arrays.sort( args );

      List l = Arrays.asList( args );

      System.out.println( l );
    }
}
```


Other Algorithms

- Other algorithms provided by the `Collections` class include
  - Shuffling
  - Data manipulation
    - `reverse()`
    - `fill()`
    - `copy()`
  - Searching
  - Finding extreme values
    - `max()`
    - `min()`

Wrapper Implementations

- Wrapper implementations add some functionality on top of what a collection offer
  - Synchronization
  - Unmodifiable
- Wrappers simply delegate all of their real work to a specified collection

Unmodifiable wrappers

- The unmodifiable wrappers, rather than adding functionality to the wrapped collection, take functionality away.
  - Any attempt to modify the collection generates an UnsupportedOperationException
- The unmodifiable wrappers have two main uses:
  - To make a collection immutable once it has been built.
  - To allow "second-class citizens" read-only access to your data structures. You keep a reference to the backing collection, but hand out a reference to the wrapper.

Unmodifiable wrappers

```
public static Collection unmodifiableCollection(Collection c);
public static Set unmodifiableSet(Set s);
public static List unmodifiableList(List list);
public static Map unmodifiableMap(Map m);
public static SortedSet unmodifiableSortedSet(SortedSet s);
public static SortedMap unmodifiableSortedMap(SortedMap m);
```


What About User Objects?

- The Collections framework will work with any Java class
- You need to be sure you have defined
  - `equals()`
  - `hashCode()`
  - `compareTo()`
- Don’t use mutable objects for keys in a Map

`hashCode()`

- `hashCode()` returns distinct integers for distinct objects.
  - If two objects are equal according to the `equals()` method, then the `hashCode()` method on each of the two objects must produce the same integer result.
  - When `hashCode()` is invoked on the same object more than once, it must return the same integer, provided no information used in equals comparisons has been modified.
  - It is *not* required that if two objects are unequal according to `equals()` that `hashCode()` must return distinct integer values.

Interface `Comparable`

- This ordering is referred to as the class's *natural ordering*, and the class's `compareTo()` method is referred to as its *natural comparison method*.
- A class's natural ordering is said to be *consistent with equals* if and only if `(e1.compareTo((Object)e2)==0)` has the same boolean value as: `e1.equals((Object)e2)` for every `e1` and `e2` of class C.

`Name`

```
import java.util.*;
import java.util.*;

public class Name implements Comparable {

    private String first;
    private String last;

    public Name( String firstName , String lastName ) {
         first = firstName ;
         last  = lastName;
    }

    public String getFirst() {
         return first;
    }

    public String getLast () {
         return last;
    }
```

Name

```
public boolean equals( Object o ) {
     boolean retval = false;

     if (o !=null && o instanceof Name ) {
         Name n = ( Name )o;
         retval = n.getFirst().equals( first ) &&
                    n.getLast().equals( last );
     }

     return retval;
}

public int hashCode() {
     return first.hashCode() + last.hashCode();
}

public String toString() {
     return first + " " + last;
}
```


Name

```
    public int compareTo( Object o ) throws ClassCastException {
         int retval;

         Name n = ( Name ) o;

         retval = last.compareTo( n.getLast() );

         if ( retval == 0 )
             retval = first.compareTo( n.getFirst() );

         return retval;
    }

} //Name
```