**The Java Collections Framework**

**Definition**
Set of **interfaces**, abstract and concrete classes that define common abstract data types in Java
- e.g. list, stack, queue, set, map

*Part of the java.util package*

**Implementation**
Extensive use of generic types, hash codes (Object.hashCode()), and Comparable interface (compareTo(), e.g. for sorting)

**Collection Interface**
Defines common operations for sets and lists (‘unordered’ ops.)

**Maps**
Represented by separate interfaces from list/set
(due to key/value relationship vs. a group of elements)
Java Collections Interfaces

*(slide: Carl Reynolds)*

Note: Some of the material on these slides was taken from the Java Tutorial at http://www.java.sun.com/docs/books/tutorial
Figure 22.1  Set and List are subinterfaces of Collection.

Figure 22.2  An instance of Map stores a group of objects and their associated keys.
### Implementation Classes

*(slide derived from: Carl Reynolds)*

<table>
<thead>
<tr>
<th>Interface</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Set</strong></td>
<td><strong>HashSet</strong></td>
</tr>
<tr>
<td></td>
<td><strong>TreeSet</strong></td>
</tr>
<tr>
<td></td>
<td><strong>LinkedHashSet</strong></td>
</tr>
<tr>
<td><strong>List</strong></td>
<td><strong>ArrayList</strong></td>
</tr>
<tr>
<td></td>
<td><strong>LinkedList</strong></td>
</tr>
<tr>
<td><strong>Map</strong></td>
<td><strong>HashMap</strong></td>
</tr>
<tr>
<td></td>
<td><strong>TreeMap</strong></td>
</tr>
<tr>
<td></td>
<td><strong>LinkedHashMap</strong></td>
</tr>
</tbody>
</table>

Note: When writing programs use the interfaces rather than the implementation classes where you can: this makes it easier to change implementations of an ADT.
Notes on ‘Unordered’ Collections
(Set, Map Implementations)

HashMap, HashSet
Hash table implementation of set/map
Use hash codes (integer values) to determine where set elements or
(key,value) pairs are stored in the hash table (array)

LinkedHashMap, LinkedHashSet
Provide support for arranging set elements or (key,value) pairs by
order of insertion by adding a linked list within the hash table elements

TreeMap, TreeSet
Use binary search tree implementations to order set elements by
value, or (key,value) pairs by key value
Sets in the Collections Framework

- `java.util.Set<E>`
  - `java.util.AbstractSet<E>`
    - `java.util.HashSet<E>`
      - `HashSet()`
      - `HashSet(c: Collection<? extends E>)`
    - `java.util.LinkedHashSet<E>`
      - `LinkedHashSet()`
      - `LinkedHashSet(c: Collection<? extends E>)`
  - `java.util.TreeSet<E>`
    - `TreeSet()`
    - `TreeSet(c: Collection<? extends E>)`
    - `TreeSet(c: Comparator<? super E>)`

E: a generic type parameter

- `java.util.SortedSet<E>`
  - `first(): E`
  - `last(): E`
  - `headSet(toElement: E): SortedSet<E>`
  - `tailSet(fromElement: E): SortedSet<E>`

Returns the first in this set.
Returns the last in this set.
headSet/tailSet returns a portion of the set less than toElement/greater than fromElement.

Creates a tree set with the specified comparator.

Figure 22.4 The Java Collections Framework provides three concrete set classes.
The Collection interface contains the methods for manipulating the elements in a collection, and each collection object contains an iterator for traversing elements in the collection.
HashSet
(Example: TestHashSet.java, Liang)

**Methods:**
Except for constructors, defined methods identical to Collection

**Element Storage:**
‘Unordered,’ but stored in a hash table according to their hash codes
**All elements are unique**
*Do not* expect to see elements in the order you add them when you output them using `toString()`.

**Hash Codes**
– Most classes in Java API override the `hashCode()` method in the Object class
– Need to be defined to properly disperse set elements in storage (i.e. throughout locations of the hash table)
– For two equivalent objects, hash codes must be the same
LinkedHashSet
(example: TestLinkedHashSet.java)

Methods
Again, same as Collection Interface except for constructors

Addition to HashSet
– Elements in hash table contain an extra field defining order in which elements are added (as a linked list)
– List maintained by the class

Hash Codes
Notes from previous slide still apply (e.g. equivalent objects, equivalent hash codes)
Ordered Sets: TreeSet
(example: TestTreeSet.java)

Methods
Add methods from SortedSet interface:
first(), last(), headSet(toElement: E), tailSet(fromElement: E)

Implementation
A binary search tree, such that either:
1. Objects (elements) implement the Comparable interface (compareTo())
   (“natural order” of objects in a class), or
2. TreeSet is constructed using an object implementing the Comparator
   interface (compare()) to determine the ordering (permits comparing
   objects of the same or different classes, create different orderings)

One of these will determine the ordering of elements.

Notes
– It is faster to use a hash set to retrieve elements, as TreeSet keeps
  elements in a sorted order (making search necessary)
– Can construct a tree set using an existing collection (e.g. a hash set)
Iterator Interface

**Purpose**
Provides uniform way to traverse sets and lists
Instance of Iterator given by iterator() method in Collection

**Operations**
- Similar behaviour to operations used in Scanner to obtain a sequence of tokens
- Check if all elements have been visited (hasNext())
- Get next element in order imposed by the iterator (next())
- remove() the last element returned by next()
# List Interface

*(modified slide from Carl Reynolds)*

```
List<E>

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

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

// Iteration
listIterator():ListIterator<E>;
listIterator(int):ListIterator<E>;

// Range-view List
subList(int, int):List<E>;
```
the **ListIterator** interface extends **Iterator**

Forward and reverse directions are possible

**ListIterator** is available for Java Lists, such as the **LinkedList** implementation

<table>
<thead>
<tr>
<th>ListIterator &lt;E&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>hasNext():boolean;</td>
</tr>
<tr>
<td>next():E;</td>
</tr>
<tr>
<td>hasPrevious():boolean;</td>
</tr>
<tr>
<td>previous(): E;</td>
</tr>
<tr>
<td>nextIndex(): int;</td>
</tr>
<tr>
<td>previousIndex(): int;</td>
</tr>
<tr>
<td>remove():void;</td>
</tr>
<tr>
<td>set(E o): void;</td>
</tr>
<tr>
<td>add(E o): void;</td>
</tr>
</tbody>
</table>
The Collections Class

Operations for Manipulating Collections

Includes static operations for sorting, searching, replacing elements, finding max/min element, and to copy and alter collections in various ways.

(using this in lab5)

Note!

*Collection* is an interface for an abstract data type, *Collections* is a separate class for methods operating on collections.
List: Example

TestArrayAndLinkedList.java
(course web page)
Map \(<K, V>\) Interface

(modified slide from Carl Reynolds)

### Map \(<K, V>\>

// Basic Operations
- put(K, V): V;
- get(K): V;
- remove(K): V;
- containsKey(K): boolean;
- containsValue(V): boolean;
- size(): int;
- isEmpty(): boolean;

// Bulk Operations
- void putAll(Map t): void;
- void clear(): void;

// Collection Views
- keySet(): Set<K>;
- values(): Collection<V>;
- entrySet(): Set<Entry<K, V>>;

### Entry \(<K, V>\>

- getKey(): K;
- getValue(): V;
- setValue(V): V;
Map Examples

CountOccurrenceOfWords.java
(course web page)
TestMap.java (from text)
You may define an alternate ordering for objects of a class using objects implementing the Comparator Interface (i.e. rather than using compareTo())

Sort people by age instead of name
Sort cars by year instead of Make and Model
Sort clients by city instead of name
Sort words alphabetically regardless of case
Comparator\(<T>\) Interface

One method:

\[
\text{compare}( T \, o1, \, T \, o2 )
\]

Returns:

- negative if \( o1 < o2 \)
- Zero if \( o1 == o2 \)
- positive if \( o1 > o2 \)
import java.util.*;
public class CaseInsensitiveComparator implements Comparator<String> {
    public int compare( String stringOne, String stringTwo ) {

        // Shift both strings to lower case, and then use the
        // usual String instance method compareTo()
        return stringOne.toLowerCase().compareTo( stringTwo.toLowerCase() );
    }
}
Using a Comparator...

```java
import java.util.*;
public class SortExample2B {
    public static void main( String args[] ) {

        List aList = new ArrayList<String>();

        for ( int i = 0; i < args.length; i++ ) {
            aList.add( args[ i ] );
        }
        Collections.sort( aList , new CaseInsensitiveComparator() );
        System.out.println( aList );
    }
}
```