Sorting: Java way

Lecture 13

Java collections framework

- Adding as many elements as you want
- Finding items by name
- Automatically remove all duplicates
- Sorting
- Searching

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

ArrayList: automatically resizable array

boolean	**add**(E e) Appends the specified element to the end of this list.
void	**add**(int index, E element) Inserts the specified element at the specified position in this list.
boolean	**addAll**(Collection<? extends E> c) Appends all of the elements in the specified collection to the end of this list, in the order that they a
boolean	**addAll**(int index, Collection<? extends E> c) Inserts all of the elements in the specified collection into this list, starting at the specified position.
void	**clear**() Removes all of the elements from this list.
Object	**clone**() Returns a shallow copy of this `ArrayList` instance.
boolean	**contains**(Object o) Returns `true` if this list contains the specified element.
void	**ensureCapacity**(int minCapacity) Increases the capacity of this `ArrayList` instance, if necessary, to ensure that it can hold at least th
E	**get**(int index) Returns the element at the specified position in this list.
int	**indexOf**(Object o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does n
boolean	**isEmpty**() Returns `true` if this list contains no elements.

ArrayList: automatically resizable array

int	lastIndexOf(Object o) Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not
E	remove(int index) Removes the element at the specified position in this list.
boolean	remove(Object o) Removes the first occurrence of the specified element from this list, if it is present.
protected void	removeRange(int fromIndex, int toIndex) Removes from this list all of the elements whose index is between fromIndex, inclusive, and toInd
E	set(int index, E element) Replaces the element at the specified position in this list with the specified element.
int	size() Returns the number of elements in this list.
Object[]	toArray() Returns an array containing all of the elements in this list in proper sequence (from first to last elem
<T> T[]	toArray(T[] a) Returns an array containing all of the elements in this list in proper sequence (from first to last elem
void	trimToSize() Trims the capacity of this ArrayList instance to be the list's current size.

SuppressWarnings annotation

all to suppress all warnings

boxing to suppress warnings relative to boxing/unboxing operations

cast to suppress warnings relative to cast operations

fallthrough to suppress warnings relative to missing breaks in switch statements

null to suppress warnings relative to null analysis

rawtypes to suppress warnings relative to un-specific types when using generics on class params

unchecked to suppress warnings relative to unchecked operations

unused to suppress warnings relative to unused code

Raw Array List: contains Object elements

```
package sorting;
import java.util.*;

public class SortingStrings
{
	@SuppressWarnings(value={"unchecked","rawtypes"})

	public static void main (String [] args)
	{
		String [] stuff={"apple","orange","bagel","monster","youtube"};

		List s=new LinkedList(Arrays.asList(stuff));
		Collections.sort(s);
		System.out.println(s);
		Collections.sort(s,Collections.reverseOrder());
		System.out.println(s);
	}
}
```

Sorting song names: reader

```
@SuppressWarnings(value={"unchecked","rawtypes"})

public class SongsNameReader {
    List songList = new ArrayList();
    public void go() {readSongs();}

    void readSongs()
    {
        try{
            File file = new File("songs.txt");
            BufferedReader reader = new BufferedReader(new FileReader(file));
            String line = null;
            while ((line= reader.readLine()) != null) { addSong(line);}
            reader.close();
        } catch(Exception ex) { ex.printStackTrace();}
        }

    void addSong(String lineToParse) {
            String[] tokens = lineToParse.split(",");
            songList.add(tokens[0]);
            }
}
```

Sorting song names: sort

```
package sorting;
import java.util.*;

@SuppressWarnings(value={"unchecked","rawtypes"})
public class SortingSongNames
{
		public static void main(String[] args)
		{

				SongsNameReader reader=new SongsNameReader();
				reader.go();
				List songNames=reader.songList;
				System.out.println(songNames);

				System.out.println("Sorted songs:");
				Collections.sort(songNames);
				System.out.println(songNames);
		}
}
```

Sorting songs: Song class

```
public class Song {
		private String name;
		private String artist;
		private int rank;

		public Song(String name,String artist, int rank)
		{
				this.name =name;
				this.artist =artist;
				this.rank=rank;
		}
		public String getName() {
				return name;
		}
		public void setName(String name) {
				this.name = name;
		}
		public String getArtist() {
				return artist;
		}
		public void setArtist(String artist) {
				this.artist = artist;
		}
		public int getRank() {
				return rank;
		}
		public void setRank(int rank) {
				this.rank = rank;
		}

}
```

Sorting songs: SongsReader

```
void addSong(String lineToParse) {
        String[] tokens = lineToParse.split(",");
        songList.add(new Song(tokens[0],tokens[1],Integer.parseInt(tokens[2])));
    }
```

Sorting songs: printing songs

```
SongsReader reader=new SongsReader();
reader.go();
List songs=reader.songList;
System.out.println(songs);
```

```
[sorting.Song@15b7986, sorting.Song@87816d, sorting.Song@422ede,
sorting.Song@112f614]
```

Sorting songs: override toString

```
@Override
public String toString() {
        return "Song [name=" + name + ", artist=" + artist + ", rank=" + rank+ "]";
}
```

```
[Song [name=Whistle, artist=Flo Rida, rank=7], Song [name=Too Close,
artist=Alex Clare, rank=9], Song [name=Good Time, artist=Owl City, rank=6],
Song [name=Home, artist=Phillip Phillips, rank=1]]
```

Sorting songs: sort

```
System.out.println("Sorted songs:");
Collections.sort(songs);
System.out.println(songs);
```

```
Exception in thread "main" java.lang.ClassCastException: sorting.Song cannot be cast to
java.lang.Comparable
at java.util.ComparableTimSort.countRunAndMakeAscending(Unknown Source)
at java.util.ComparableTimSort.sort(Unknown Source)
at java.util.ComparableTimSort.sort(Unknown Source)
at java.util.Arrays.sort(Unknown Source)
at java.util.Collections.sort(Unknown Source)
at sorting.SortingSongs.main(SortingSongs.java:16)
```

Bubble Sort

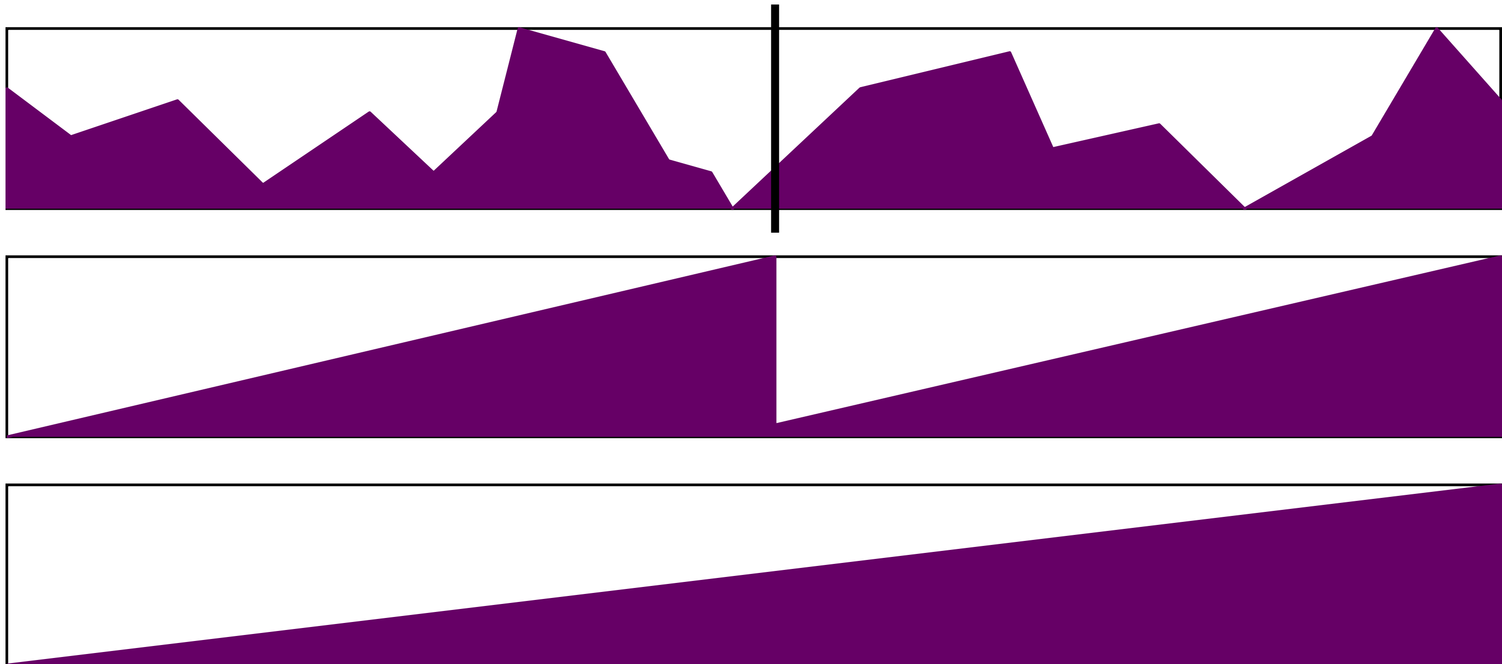
- Iterate through sequence, **compare** each element to right neighbor.
- Exchange adjacent elements if necessary.
- Keep passing through sequence until no exchanges are required (up to N times).
- Each pass causes largest element to bubble into place: 1st pass, largest; 2nd pass, 2nd largest, ...
- Therefore get a sorted sub-array on the right and can stop one position sooner each pass (more efficient than brute force bubbling through entire array each pass…)

Insertion Sort

- Like inserting a new card into a partially sorted hand by bubbling to the left into sorted subarray on left; little less brute-force than bubble sort
  - add one element a[i] at a time
  - find proper position, j+1, to the left by shifting to the right a[i-1], a[i-2], ..., a[j+1] left neighbors, until **a[j] < a[i]**
  - move a[i] into vacated a[j+1]
- After iteration i<n, the original a[0] ... a[i] are in sorted order, but not necessarily in final position

Java Collections use Merge sort

- *Partition* sequence into two sub-sequences of N/2 elements.
- Recursively *partition* and *sort* each sub-sequence.
- *Merge* the sorted sub-sequences.



Main part of every sorting algorithm:

- Comparing two elements

Collections.sort API

sort

public static void sort(List list)

Sorts the specified list into ascending order, according to the *natural ordering* of its elements. All elements in the list must implement the **Comparable** interface. Furthermore, all elements in the list must be mutually comparable (that is, e1.compareTo(e2) must not throw a ClassCastException for any elements e1 and e2 in the list).

This sort is guaranteed to be stable: equal elements will not be reordered as a result of the sort.

The specified list must be modifiable, but need not be resizable.

The sorting algorithm is a modified mergesort (in which the merge is omitted if the highest element in the low sublist is less than the lowest element in the high sublist). This algorithm offers guaranteed n log(n) performance. This implementation dumps the specified list into an array, sorts the array, and iterates over the list resetting each element from the corresponding position in the array. This avoids the n2 log(n) performance that would result from attempting to sort a linked list in place.

Interface Comparable - API: defines a single method

compareTo

int compareTo(T o)

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

The implementor must ensure sgn(x.compareTo(y)) == -sgn(y.compareTo(x)) for all x and y. (This implies that x.compareTo(y) must throw an exception iff y.compareTo(x) throws an exception.)

The implementor must also ensure that the relation is transitive: (x.compareTo(y)>0 && y.compareTo(z)>0) implies x.compareTo(z)>0.

Finally, the implementor must ensure that x.compareTo(y)==0 implies that sgn(x.compareTo(z)) == sgn(y.compareTo(z)), for all z.

It is strongly recommended, but not strictly required that (x.compareTo(y)==0) == (x.equals(y)). Generally speaking, any class that implements the Comparable interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

In the foregoing description, the notation sgn(expression) designates the mathematical signum function, which is defined to return one of -1, 0, or 1 according to whether the value of expression is negative, zero or positive.

Parameters:

o - the object to be compared.

Returns:

a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Throws:

How do we compare songs? Natural order

```
public class Song implements Comparable {
    public int compareTo(Object another)
        {
            Song song=(Song)another;
            return this.name.compareTo(song.name);
        }
    }
```

Now the sorting works

```
Sorted songs:
[Song [name=Good Time, artist=Owl City, rank=6], Song [name=Home, artist=Phillip Phillips, rank=1], Song [name=Too Close, artist=Alex Clare, rank=9], Song [name=Whistle, artist=Flo Rida, rank=7]]
```

New challenge: sorting by rank

- But when you make a collection comparable, you get only one chance to implement the compareTo() method.

Invoking Collections.sort(List o, Comparator c)

- Invoking the one-argument sort(List 0) method means the list element's compareTo() method determines the order. So the elements in the list MUST implement the Comparable interface
- Invoking sort(List o, Comparator c) means the lit element's compareTo() method will NOT be called, and the Comparator's compare() method will be used instead
- If you do not have a source code of an element's class, you can still put things in order by creating a Comparator

Implementing RankComparator

```
import java.util.Comparator;

@SuppressWarnings(value={"rawtypes"})
public class RankComparator implements Comparator
{
        public int compare(Object one, Object two)
        {
                Song song1=(Song)one;
                Song song2=(Song)two;

                if(song1.getRank()>song2.getRank())
                                return 1;
                if(song1.getRank()<song2.getRank())
                        return -1;
                return 0;
        }
}
```

Sorting songs by rank

```
System.out.println("Sorted by rank:");
RankComparator rankC=new RankComparator();
Collections.sort(songs, rankC);
System.out.println(songs);
```

```
Sorted by rank:
[Song [name=Home, artist=Phillip Phillips, rank=1], Song [name=Good Time, artist=Owl City, rank=6], Song [name=Whistle, artist=Flo Rida, rank=7], Song [name=Too Close, artist=Alex Clare, rank=9]]
```

How to sort by artist?

Tomorrow in the lab

- Write all possible sorting methods for books collection
- This has the only purpose: to make you remember how the sorting in Java works
- You can use parametrized types for your book list, if you know what it is. (We have used Raw types here in order to present a concept of sorting)

- **Given the following compilable statement:**

Collections.sort(myArrayList);

1. What must the class of objects stored in myArrayList implement?
2. What method must it implement?
3. Can the class of objects stored in myArrayList implement both Comparator and Comparable?

- Given the following compilable statement:

Collections.sort (myList, myComparator)

1. Can the class of the objects stored in myList implement Comparable?
2. Can it implement Comparator?
3. Must the class of the objects stored in myList implement Comparable?
4. Must it implement Comparator?
5. What must the class of the myCompare object implement?
6. What method must the class of the myComparator object implement?