

ECS510

Algorithms and Data Structures in an
Object Oriented Framework

“ADSOOF”

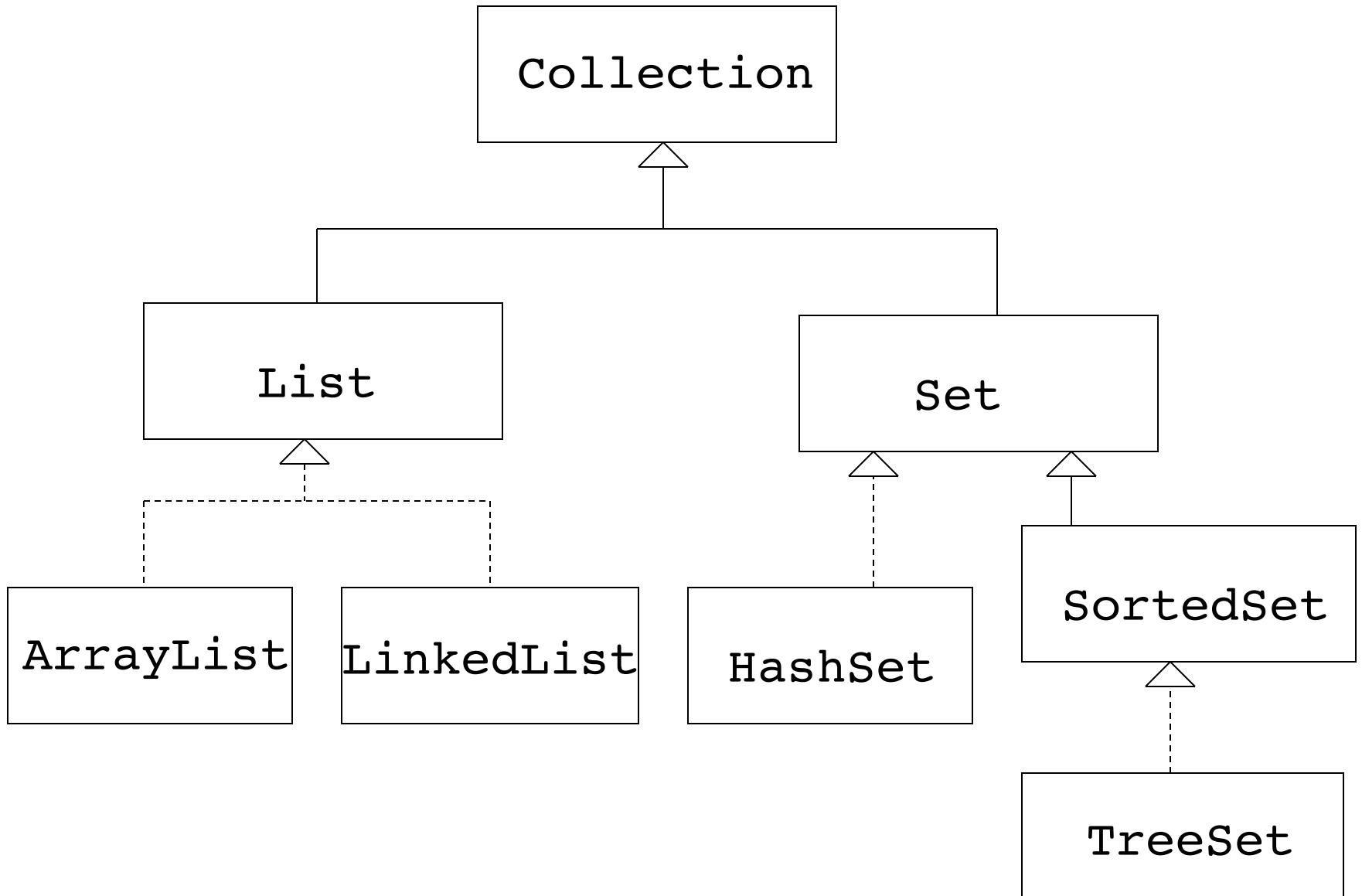
Java’s Collections Framework

Java's Collections Framework

- A “framework” is a group of classes and interfaces designed to work together
- Java's Collections Framework provides library code for collections of data
- Using library code saves you programming effort
- It means you are using a common standard, others will be familiar with that standard, and will more easily understand your code
- Your code will more easily link with other code which uses the same standard

ArrayList

- `ArrayList<E>` is just one class in the Collections Framework
- It is an implementation of the interface `List<E>`
- `List<E>` is an extension of another interface, `Collection<E>`
- `List<E>` is intended to define array-like behaviour
- Unless you have particular need for “hands-on” control, it’s best to program using these rather than arrays directly



Collection<E>

Has methods which include:

```
boolean add(E e)
```

```
boolean remove(Object o)
```

```
boolean contains(Object o)
```

```
int size()
```

Doesn't fully define how they work

Set<E>

- Defines `s.add(e)` to work as:
 - If `s.contains(e)` was `false`, it becomes `true`, method returns `true`, `s.size()` increased by 1
 - If `s.contains(e)` was `true`, it remains `true`, method returns `false`, `s.size()` unchanged
- Defines `s.remove(e)` to work as:
 - If `s.contains(e)` was `true`, it becomes `false`, method returns `true`, `s.size()` decreased by 1
 - If `s.contains(e)` was `false`, it remains `false`, method returns `false`, `s.size()` unchanged
- This is mathematical set behaviour

List<E>

- Adds methods which work on items being in a position in the collection:
 - `E get(i)`
 - `E set(i, e)` (returns replaced element)
 - `int indexOf(e)`
- Defines `ls.add(e)` to work by adding `e` to end of list, returns `true`
- Defines `ls.remove(e)` to work by removing lowest indexed occurrence of `e`, returns `false` if no occurrence of `e`, `true` otherwise

Overloaded `List<E>` methods

Overloading is when there are two methods of the same name in a class or interface, `List<E>` has two examples:

- `ls.add(i, e)` adds `e` at position `i` in list, following elements move up one position (different from `list.add(e)` as that has one argument, this has two)
- `ls.remove(i)` removes the element at position `i` in list, following elements down one position (different from the general `remove` method only by argument type, `int`, rather than `Object`)

Copy constructors

Collection classes do not have a constructor which works as constructors for new arrays do, that is by creating a collection of a particular size. But they do have constructors which take another collection object as their argument and create a new collection containing that collection's elements

- `ArrayList(Collection<? extends E> c)`
- `HashSet(Collection<? extends E> c)`
- `TreeSet(Collection<? extends E> c)`

Iterator<E>

- Class `Collection<E>` has method
`Iterator<E> iterator()`
- Class `Iterator<E>` has methods
 - `E next()` returns an item from the collection that hasn't been returned by a previous `next()` call, throws `NoSuchElementException` if there are no more
 - `boolean hasNext()` returns `true` if there are further items to be returned by a `next()` call, `false` otherwise
- You can't use `for(int i=0; i<s.size(); i++)` to go through a `Set`, but you can use `Iterator`

For-each loop

```
for(Iterator<E> it=a.iterator(); it.hasNext(); )  
    {  
        E element = it.next();  
        ...  
    }
```

is a common pattern, Java (since Java 5) allows a shorthand equivalent:

```
for(E element: coll)  
    {  
        ...  
    }
```

This works for `coll` of type `Collection<E>` or any extension, and also for `coll` of type `E[]`

Order of return for `Iterator<E>`

- When produced from an object which implements `List<E>`, elements are returned in their position order
- When produced from a `HashSet<E>` object, elements are not returned in any obvious order
- When produced from a `TreeSet<E>` object, elements are returned in their natural order (but see later for variation of this)

Other Collection<E> methods

- `boolean addAll(Collection<? extends E> c)`
- `boolean containsAll(Collection<?> c)`
- `boolean removeAll(Collection<?> c)`
- `boolean retainAll(Collection<?> c)`
- `Object[] toArray()`
- `<T> T[] toArray(T[] a)`
- `boolean isEmpty()`

Other List<E> methods

- `boolean addAll(int index,
Collection<? extends E> c)`
- `int lastIndexOf(Object o)`
- `ListIterator<E> listIterator()`
- `List<E> subList(int fromIndex, int toIndex)`

SortedSet<E>

- SortedSet<E> introduces some extra methods to Set<E>
 - SortedSet<E> headSet(e) where s.headSet(e) returns the set of all elements from s less than e
 - SortedSet<E> tailSet(e) where s.tailSet(e) returns the set of all elements from s greater than or equal to e
- TreeSet<E> implements SortedSet<E>

HashSet<E> and TreeSet<E>

- HashSet<E> is implemented using a “hash table”, elements stored in array, “hash function” gives their position, $O(1)$ access
- TreeSet<E> is implemented using an “ordered binary tree”, linked structure, cells have two links, follow one or other in binary search to find element, $O(\log N)$ access
- There is not time to discuss these data structures in more detail in this module
- See data structures text books and web sites

Comparator<T>

- Provided with Java, it is:

```
interface Comparator<T>
{
    int compare(T o1, T o2)
}
```

- If `c` is of type `Comparator<T>` and `t1` and `t2` are of type `T`, then `c.compare(t1, t2)` returns
 - A negative integer if `t1` less than `t2`
 - A positive integer if `t1` greater than `t2`
 - 0 if they are equalin some ordering

Writing your own Comparator

- You can write your own class which implements `Comparator<T>` for `T` some type, but which gives an ordering different from `T`'s natural order, example:

```
class LengthComparer implements Comparator<String>
{
    public int compare(String str1,String str2)
    {
        return str1.length()-str2.length();
    }
}
```

This orders `Strings` by length rather than alphabetically

Scrabble score Comparator

```
class Scrabble implements Comparator<String>
{
    public static final int[] scores =
        {1,3,3,2,1,4,2,4,1,8,5,1,3,1,1,3,10,1,1,1,1,4,4,8,4,10};

    public static int score(String str) {
        String str1 = str.toUpperCase();
        int score = 0;
        for(int i=0; i<str1.length(); i++)
            score+=scores[str1.charAt(i)-'A'];
        return score;
    }

    public int compare(String str1,String str2) {
        return score(str1)-score(str2);
    }
}
```

Finding the biggest using a Comparator

```
public static <T> T
    biggest(Collection<T> coll, Comparator<? super T> comp)
{
    Iterator<T> it = coll.iterator();
    T biggest = it.next();
    while(it.hasNext())
    {
        T nextItem = it.next();
        if(comp.compare(nextItem, biggest) > 0)
            biggest = nextItem;
    }
    return biggest;
}
```

You can use this to find the biggest item in a collection according to the ordering given by the `Comparator` argument

- If we have `ArrayList<String> words`
 - `biggest(words, new LengthComparer())`
returns the longest string from words
 - `biggest(words, new Scrabble())`
returns the string with the highest Scrabble score
- But rather than write our own `biggest`, we can use Java's method `max` in class `Collections`
 - `Collections.max(words, new LengthComparer())`
returns the longest string from words
 - `Collections.max(words, new Scrabble())`
returns the string with the highest Scrabble score
- `Comparator<? super T>` allows us to use e.g. a `Comparator<Fruit>` to find the biggest from a `Basket<Banana>`

Sorting using a Comparator

- We have already seen Java's built-in methods which sort by “natural order”, so if `ls` is of type `List<E>` then the method call `Collections.sort(ls)` will sort the list to which `ls` refers. The sorting is done “in place”.
- This means the elements of `ls` must have their own `compareTo` method, so `E` should be declared as `<E extends Comparable<? super E>>`
- As an alternative, `Collections.sort(ls, comp)` will sort the list to which `ls` refers by the order given by a comparator referred to by `comp`.
- With this, `E` can be any type, so does not need a bound, and `comp` should be of type `Comparator<? super E>`

Anonymous Classes

- ```
Comparator<String> c = new Comparator<String>(){
 public int compare(String str1,String str2)
 {
 return str1.length()-str2.length();
 }
};
```

is an alternative to

- ```
Comparator<String> c = new LengthComparer();
```

It avoids the need to have a separate class
LengthComparer

Constructing a TreeSet using a Comparator

- `new TreeSet<Thing>()` will construct a new `TreeSet<Thing>` object with its contents ordered by the natural order of `Thing` (as given by its `compareTo` method)
- If `c` is of type `Comparator<? super Thing>` then `new TreeSet<Thing>(c)` will construct a new `TreeSet<Thing>` object with its contents ordered by the `compare` method in `c`.
- The variant ordering will be observed in the `Iterator` produced from the `TreeSet`, and also in the way `headSet` etc works

Map<K, V>

- Map<K, V> is a generic interface with two type arguments
- It represents a collection of objects of type V indexed by keys of type K
- K is often but not always String
- HashMap<K, V> and TreeMap<K, V> are two classes which implement it
- TreeMap<K, V> has elements stored in order of K
- HashMap<K, V> more efficient but no ordering of elements

Map methods

- With `m` of type `Map<K, V>`, `tag` of type `K` and `item` of type `V`:
 - `m.put(tag, item)` puts `item` into the map with key `tag`, returns previous item with key `tag`
 - `m.get(tag)` returns the last item put in with key `tag` so long as it hasn't been removed
 - `m.remove(tag)` removes the item with key `tag`
 - `m.keySet()` returns an object of type `Set<K>` giving all the keys of items in the map
 - `m.values()` return an object of type `Collection<V>` giving all the items in the map

Map uses

- A collection whose elements are indexed by an identifier (key) rather than a position
- Unlike lists, adding or deleting an item does not change the index of other items
- Could be used as a simple database - K is student no., NI no., car reg, V is class representing students, citizens, cars
- Or anywhere else where we have data as a collection of pairs, with one part of the pair not duplicated in other pairs e.g. count of number of occurrences of each word in a document, K is `String` for word, V is `Integer` for the count

SubLists

- If `str` is of type `String` then `str.substring(from,to)` is a `String` of the characters in `str` starting at position `from` up to but not including position `to`
- Similarly, if `a` is of type `List<E>` (so `ArrayList<E>` or `LinkedList<E>`) then `a.subList(from,to)` is a `List<E>` of the elements in `a` starting at position `from` up to but not including position `to`
- Example, `a` is `[10, 20, 30, 40, 50, 60, 70]`, so `b=a.subList(2,5)` sets `b` to `[30, 40, 50]`

Views

- A view of a collection is a collection which shares (some of) the data of the collection
- Changing the view changes the collection, and vice-versa
- `b=a.subList(p1,p2)` creates a view of `a`
- If `s` is of type `TreeSet<E>` and `e` is of type `E`, then `s.headSet(e)` and `s.tailSet(e)` create views of `s`
- `b=Collections.unmodifiableList(a)` creates a view of `a`

SubLists are views

- Example, a is [10 , 20 , 30 , 40 , 50 , 60 , 70], so `b=a.subList(2 , 5)` sets b to [30 , 40 , 50]
- This means `b.add(1 , 35)` changes b to [30 , 35 , 40 , 50] and changes a to [10 , 20 , 30 , 35 , 40 , 50 , 60 , 70]
- After this `a.add(4 , 45)` changes a to [10 , 20 , 30 , 35 , 40 , 45 , 50 , 60 , 70] and changes b to [30 , 35 , 40 , 45 , 50]

Unmodifiable views

- `b=Collections.unmodifiableList(a)`
creates a view of `a`
- If you attempt to call any method on `b` which changes it, e.g. `add` or `remove`, you will get an `UnsupportedOperationException`
- This is useful if you want another part of a program to access your data but not to change it
- But if a call on `a` changes `a`, `b` will get changed in the same way

Java's Collections Framework

- Consists of interfaces, implementations and algorithms
- Interfaces specify an Abstract Data Type, and are arranged in a hierarchy (an interface may specialise another interface)
- Implementations use data structures to provide working code - different implementations have different benefits
- Some algorithms, e.g. sort, provided as static methods
- There are various extra interfaces to aid writing generalised code

Algorithms and Data Structures

- Seeing how algorithms and data structures translate to code helps develop your coding ability
- It give you and understanding of the basic techniques of algorithm development, and efficiency issues
- It shows how careful attention to the application and implementation divide gives well structured programs
- In practice, however, you wouldn't need to write your own code for the simple algorithms and data structures presented here - you would use library code

Code Re-use

- Code re-use means seeing if there is an existing class which provides what you want (not “cut and paste”)
- An interface with carefully chosen and well specified methods makes code open for re-use
- Using interface types which have just the necessary methods is one way of generalising code, so it can be re-used in a variety of situations
- Considering how aspects of the code could be provided as parameters rather than fixed is another way of generalising

Java™ 's APIs

- API – “Application Programming Interface”
- The formal name for a “code library”
- Provided to give code for common operations, and for interaction with databases, graphics, web etc
- Once you understand the basics of OOP, a lot of programming is knowing what and how to use from the API
- Oracle’s Java™ APIs may be supplemented by APIs from other suppliers
- You may define and implement your own APIs