ECS414
Object Oriented Programming
Sorting

Matthew Huntbach
matthew.huntbach@qmul.ac.uk

Sorting

- Java provides built-in code to sort collections such as `ArrayList<E>`s, and for real programming it is better to use that than to write your own
- However, looking at code that does sorting, and practicing coding by writing it yourself, is a good way to help learn about basic aspects of programming that can be used more generally
- It is easy to understand the general principle of sorting a collection
- For example, looking at a method that sorts an `ArrayList<E>` will give you practice in how to manipulate `ArrayList<E>`s

Bubble Sort of an Array of Integers (1)

- Here is code that you saw in the module ECS401U Procedural Programming:

```
static void sort(int[] array){
  boolean sorted = false;
  while (!sorted) {
     sorted = true;
     for(int i=0; i<array.length-1; i++)   {
         if(array[i]>array[i+1]) {
            int tmp = array[i+1];
            array[i+1] = array[i];
            array[i] = tmp;
            sorted = false;
           }
        }
     }
 }
```

Bubble Sort of an Array of Integers with Comments

- Here is a version with comments explaining it:

```
static void sort(int[] array) {
  boolean sorted = false;
  while (!sorted) {
     // array potentially sorted
     sorted = true;
     // traverse array to end switching ill-ordered pairs
     for(int i=0; i<array.length-1; i++){
         if(array[i]>array [i+1]) {
            //swap them
            int tmp = array[i+1];
            array[i+1] = array[i];
            array[i] = tmp;
            // array wasn't sorted
            sorted = false;
           }
        }
     }
 }
```

Explanation of Bubble Sort

- Go through each position of the array up to the last but one
- At each position, if the element there is greater than the element in the next position, swap them
- So, if you have done a swap, the element you are then checking at the next position will be the same element you checked at the previous position
- If you have gone through the whole array and not done any swaps, then the array must be completely sorted
- Otherwise, it is not completely sorted, so go through it doing the same things again
- Each time you go through it, it will get closer to being completely sorted, so you will eventually finish

Bubble Sort of an `ArrayList` of Integers (1)

- Here is the exact equivalent for an `ArrayList` of integers:

```
static void sort(ArrayList<Integer> array) {
  boolean sorted = false;
  while (!sorted) {
     sorted = true;
     for(int i=0; i < array.size()-1; i++) {
         if(array.get(i) > array.get(i+1)) {
            int tmp = array.get(i+1);
            array.set(i+1,array.get(i));
            array.set(i,tmp);
            sorted = false;
           }
        }
     }
 }
```

Bubble Sort of an Array of Integers (2)

- A better version that recognises that each time round the loop means the largest will be put to the end is:

```
static void sort(int[] array){
  boolean sorted=false;
  int sortLength = array.length-1;
  while (!sorted) {
     sorted = true;
     for(int i=0; i < sortLength; i++){
         if(array[i] > array [i+1]){
            int tmp = array[i+1];
            array[i+1] = array[i];
            array[i] = tmp;
            sorted = false;
           }
        }
      sortLength--;
     }
 }
```

Bubble Sort of an `ArrayList` of Integers (2)

- It is always best not to make a method call more than once if you know it will return the same result, so that is also dealt with in this improved version for `ArrayList`:

```
static void sort(ArrayList<Integer> list){
  boolean sorted = false;
  int sortLength = list.size()-1;
  while (!sorted) {
     sorted = true;
     for(int i=0; i < sortLength; i++){
         int first = list.get(i);
         int second = list.get(i+1)
         if(first > second) {
            list.set(i+1,first);
            list.set(i,second);
            sorted = false;
           }
        }
      sortLength--;
     }
 }
```

Constructive v. Destructive

- The methods give for sorting are destructive, that is:

```
sort(list);
```

  sorts the actual `ArrayList` that the variable `list` refers to. So it does not need to return anything.

- If you do:

```
list2 = list1;
sort(list2);
```

  that will also sort what `list1` refers to as `list2` is an alias to what `list1` refers to

- One way to get a sorted version and keep the original is:

```
list2 = new ArrayList<>(list1);
sort(list2);
```

  making use of the constructor for `ArrayList<E>` that creates a copy

Insertion Sort of an `ArrayList` of Integers

- Here is a sorting method that works constructively:

```
static ArrayList<Integer> sort(ArrayList<Integer> list){
   ArrayList<Integer> sortedList = new ArrayList<>();
   for(Integer next : list)
      insert(sortedList,next);
   return sortedList;
}

static void insert(ArrayList<Integer> list, Integer num){
  for(int i=list.size()-1; i>=0; i--)
     if(num>list.get(i)) {
        list.add(i+1,num);
        return;
       }
  list.add(0,num);
 }
```

- With this, `list2 = sort(list1)` will set `list2` to refer to a sorted version of what `list1` refers to but leave what `list1` refers to unchanged

Explanation of Insertion Sort and `add` compared to `set`

- Insertion sort works as following:
  - Create a new `ArrayList`
  - Go through each element of the `ArrayList` being sorted and add it to the new `ArrayList`
  - When you add the new element, put it in the position that keeps the new `ArrayList` sorted
- Note that the code for insertion sort uses `list.add(i+1,num)` whereas bubble sort uses `list.set(i+1,first)`
- The call `list.add(p,val)` puts `val` at position `p` and moves what was at position `p` and everything after it up by one position, so it increases the size of what `list` refers to
- The call `list.set(p,val)` puts `val` at position `p` and leaves everything else in the same position, so what was at position `p` is removed and replaced by what `val` refers to, and what `list` refers to remains the same size

Selection Sort of an Array of Integers (1)

- Here is another sorting algorithm for an array of integers:

```
static void sort(int[] array) {
  int sortLength = array.length;
  while(sortLength>1) {
      int maxPos = 0;
      for(int i=1; i<sortLength; i++)
          if(array[i]>array[maxPos])
              maxPos = i;
      sortLength--;
      int tmp = array[maxPos];
      array[maxPos] = array[sortLength];
      array[sortLength] = tmp;
     }
}
```

Selection Sort of an Array of Integers (2)

- It can be easier to understand if divided into separate methods:

```
static void sort(int[] array) {
 int sortLength = array.length;
 while(sortLength>1) {
     int maxPos = maxPosIn(0, sortLength, array);
     sortLength--;
     swap(maxPos, sortLength, array);
    }
}

static int maxPosIn(int from, int to, int[] array) {
// Return position of highest integer in section between from and to
 int maxPos = from;
 for(int i=from+1; i<to; i++)
     if(array[i]>array[maxPos])
        maxPos = i;
 return maxPos;
}

static void swap(int pos1, int pos2, int[] array) {
// Swap what is in position pos1 with what is in position pos2
  int tmp = array[pos1];
  array[pos1] = array[pos2];
  array[pos2] = tmp;
 }
```

Selection Sort of an Array of Integers (3)

- Here is how the same algorithm could be expressed recursively:

```
static void sort(int[] array)
{
  sortSection(0,array.length,array);
}

static void sortSection(int from, int to, int[] array)
{
  if(to-from>1)
     {
      int maxPos = maxPosIn(from,to,array);
      swap(maxPos,to-1,array);
      sortSection(from,to-1,array);
     }
 }
```

- The methods `maxPosIn` and `swap` are as before

Explanation of Selection Sort

- To sort a section of an array, start at the first position and go to the last position, finding the maximum element in the section given by those positions
- Swap the maximum element found from the position it is in to the last position
- Then do the same for the section of the array from the first position to the position one less than the last position
- To find the maximum in a section, set a variable to the first position, and check what is in the position given by that variable with each element in the other positions – change the value of the variable any position where the element at that position is greater than the element at the position given by the variable
- Sorting the whole array is done by sorting the section starting at position 0 and finishing at the last position.

Selection Sort of an `ArrayList` of Integers

```
static void sort(ArrayList<Integer> list) {
 int sortLength = list.size();
 while(sortLength>1) {
    int maxPos = 0;
    for(int i=1; i<sortLength; i++)
       if(list.get(i)>list.get(maxPos))
           maxPos = i;
    sortLength--;
    int tmp = list.get(maxPos);
    list.set(maxPos,list.get(sortLength));
    list.set(sortLength,tmp);
   }
 }
```

Selection Sort of an `ArrayList` of `Strings` by length

```
static void sortByLength(ArrayList<String> list) {
 int sortLength = list.size();
 while(sortLength>1) {
    int maxPos = 0;
    for(int i=1; i<sortLength; i++)
      if(list.get(i).length()>list.get(maxPos).length())
         maxPos = i;
    sortLength--;
    String tmp = list.get(maxPos);
    list.set(maxPos,list.get(sortLength));
    list.set(sortLength,tmp);
   }
}
```

Selection Sort of an `ArrayList` of `Strings` Alphabetically

```
static void sortAlphabetically(ArrayList<String> list) {
 int sortLength = list.size();
 while(sortLength>1) {
    int maxPos = 0;
    for(int i=1; i<sortLength; i++)
       if(list.get(i).compareTo(list.get(maxPos))>0)
          maxPos = i;
    sortLength--;
    String tmp = list.get(maxPos);
    list.set(maxPos,list.get(sortLength));
    list.set(sortLength,tmp);
   }
}
```

Problems with Sorting as Given So Far

- The algorithms you have been shown for sorting so far are very inefficient
- It is also very poor practice to write lots of pieces of code that are almost identical to each other
- So instead of having lots of sort methods for sorting collections of different types of objects, you should have one generalised sort method that can sort a collection of any type of object
- Also a method that allows objects to be sorted in different ways, for example `Strings` sorted by length, or sorted by alphabetic order, or sorted by Scrabble score
- If you just have one generalised method, then if you realise there is a better algorithm it can use, you just need to change the one method
- If you had lots of methods all using the same algorithm, then you would have to change every one of those methods

Selection Sort of an `ArrayList` of `Strings` by Scrabble Score

```
static void sortByScrabbleScore(ArrayList<String> list) {
 int sortLength = list.size();
 while(sortLength>1) {
    int maxPos = 0;
    for(int i=1; i<sortLength; i++)
       if(sscore(list.get(i))>sscore(list.get(maxPos)))
          maxPos = i;
    sortLength--;
    String tmp = list.get(maxPos);
    list.set(maxPos,list.get(sortLength));
    list.set(sortLength,tmp);
   }
}
```

Scrabble Score

```
static int sscore(String word) {
   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};
   word = word.toLowerCase();
   int sum = 0;
   for(int i=0; i<word.length(); i++)
     sum+=scores[word.charAt(i)-'a'];
   return sum;
}
```

- This is given as an example just to illustrate the general idea that you can sort objects using different comparisons
- Just using this is inefficient, as it means the Scrabble score of a word would need to be reconsidered each time it is needed
- Better ways of doing this require further techniques not yet covered

Code to Test the Timing of Sorting

Here code you could use to test the time taken to sort an `ArrayList` of integers:

```
Random rand = new Random();
long time1,time2;
Scanner input = new Scanner(System.in);
System.out.print("Enter the number of numbers: ");
int num = input.nextInt();
ArrayList<Integer> list = new ArrayList<>();
System.out.print("Enter the highest number: ");
int high = input.nextInt();
for(int i=0; i<num; i++)
   list.add(rand.nextInt(high));
time1 = new Date().getTime();
sort(list);
time2 = new Date().getTime();
System.out.print("Time taken to sort:");
System.out.println(" "+(time2-time1)+"ms");
```

Time Taken by the Code Given

- To sort an ArrayList of 1000 integers:
  - Bubble Sort: 33ms
  - Selection Sort: 23ms
  - Insertion Sort: 12ms
- To sort an ArrayList of 10000 integers:
  - Bubble Sort: 830ms
  - Selection Sort: 153ms
  - Insertion Sort: 55ms
- To sort an ArrayList of 50000 integers:
  - Bubble Sort: 21652ms
  - Selection Sort: 4683ms
  - Insertion Sort: 1314ms

Better Sorting Algorithms

- More efficient sorting algorithms work as follows: if the list has a length more than 1, divide it into two parts, sort the two parts, then join them together
- The Quicksort algorithm works by picking one element, called the “pivot”, and dividing the collection into one collection of all elements less than the pivot and one of all greater than or equal. Then the two sorted collections are joined just by putting the elements second sorted one after those of the first sorted one with the pivot in between
- The Merge Sort algorithm works by just dividing the collection into half, it does not matter which element goes into which half. Then merge the two sorted half collections by going through both, adding the lowest element not yet added to the sorted collection

Merge Sort Code (1)

```
static void sort(ArrayList<Integer> list) {
  if(list.size()>1) {
      ArrayList<Integer> list1 = new ArrayList<>();
      ArrayList<Integer> list2 = new ArrayList<>();
      for(int i=0; i<list.size(); i+=2) {
          list1.add(list.get(i));
          if(i+1<list.size())  list2.add(list.get(i+1));
         }
      sort(list1);
      sort(list2);
      for(int i=0, j=0, k=0; i<list.size(); i++) {
           if(j==list1.size())  list.set(i,list2.get(k++));
           else if(k==list2.size())  list.set(i,list1.get(j++));
           else {
                 int n1=list1.get(j);
                 int n2=list2.get(k);
                 if(n1<n2) {
                     list.set(i,n1);
                     j++;
                    }
                 else {
                    list.set(i,n2);
                    k++;
                   }
              }
           }
     }
```

Merge Sort Code (2)

```
static void sort(ArrayList<Integer> list) {
  if(list.size()>1) {
      int mid = list.size()/2;
      ArrayList<Integer> list1 =
          new ArrayList<>(list.subList(0,mid));
      ArrayList<Integer> list2 =
          new ArrayList<>(list.subList(mid,list.size());
      sort(list1);
      sort(list2);
      for(int i=0, j=0, k=0; i<list.size(); i++) {
          if(j==list1.size())  list.set(i,list2.get(k++));
          else if(k==list2.size())  list.set(i,list1.get(j++));
          else {
                int n1=list1.get(j);
                int n2=list2.get(k);
                if(n1<n2) {
                    list.set(i,n1);
                    j++;
                   }
                else {
                    list.set(i,n2);
                    k++;
                   }
               }
          }
     }
```

Time Taken by the Code Given

- To sort an ArrayList of 1000 integers:
  - Selection Sort: 23ms
  - Insertion Sort: 12ms
  - Merge Sort: 13ms
- To sort an ArrayList of 10000 integers:
  - Selection Sort: 153ms
  - Insertion Sort: 55ms
  - Merge Sort: 28ms
- To sort an ArrayList of 50000 integers:
  - Selection Sort: 4683ms
  - Insertion Sort: 1314ms
  - Merge Sort: 113ms

Constructive Quicksort Code

```
static ArrayList<Integer> sort(ArrayList<Integer> list) {
 if(list.size()>1) {
    ArrayList<Integer> list1 = new ArrayList<>();
    ArrayList<Integer> list2 = new ArrayList<>();
    int pivot = list.get(0);
    for(int i=1; i<list.size(); i++) {
        int next = list.get(i);
        if(next<pivot)
           list1.add(next);
        else
           list2.add(next);
       }
    list1=sort(list1);
    list2=sort(list2);
    list1.add(pivot);
    list1.addAll(list2);
    return list1;
   }
 else
    return list;
}
```

A Version of Destructive Quicksort Code

```
static void sort(ArrayList<Integer> list) {
 sortSection(list,0,list.size());
}

static void sortSection(ArrayList<Integer> list, int from, int to) {
 if(to-from>1) {
     int mid = (from+to)/2;
     int i=from, j=to-1;
     while(true) {
         int midval=list.get(mid);
         while(i<mid && list.get(i)<=midval) i++;
         while(j>mid && list.get(j)>=midval) j--;
         if(i==j) break;
         int tmp = list.get(i);
         list.set(i,list.get(j));
         list.set(j,tmp);
         if(i==mid) { mid = j; i++; }
         else if(j==mid) { mid = i; j--; }
         else { i++; j--; }
        }
     sortSection(list,from,mid);
     sortSection(list,mid,to);
    }
}
```

Writing Sorting Code

- For the purpose of this module, all that is required is that you write code that gives the correct results, it does not matter whether it is efficient or not
- In the exam, what the code does will be tested by running it and checking the result, but the time taken is not tested
- So, if you are asked to write code that sorts something, just write what you find easiest to do
- Although you could instead use Java's built-in sorting methods, that could require an understanding of generalisation techniques not yet fully covered
- The main point of going through lots of variations of sorting code here is just to give you examples of code that uses `ArrayList<E>`s
- Also to emphasise the difference between code that works by creating and returning a new `ArrayList<E>` and code that changes the actual `ArrayList<E>` passed to it

Built-in Sorting Code (1)

- If `list` refers to an `ArrayList<E>`, then `Collections.sort(list)` will sort the `ArrayList<E>` using the `static` method in Java's class `Collections`
- It will only work if `E` is set to a type where if `obj1` and `obj2` refer to two objects of that type then `obj1.compareTo(obj2)` returns a negative number if what `obj1` refers to is less than what `obj2` refers to, a positive number if it is greater, and 0 if they are equal
- If you want a collection of your own class of objects to be able to be sorted in that way, it must be declared as

```
implements Comparable<T>
```

with `T` set to itself, as Java has a built in interface:

```
interface Comparable<T>
{
 int compareTo(T obj);
}
```

Comparable<T> example

```
class Rectangle implements Comparable<Rectangle>
{
 private int length, width;

 public Rectangle(int llength, int wwidth) {
  length=llength;
  width=width;
 }

 public int area() {
  return length*width;
 }

 public int compareTo(Rectangle that) {
  return this.area()-that.area();
 }
}
```

- This will give `Rectangle` objects a natural order based on their area

Arrays and `ArrayLists`

- An array is of a fixed size, if you write code that creates and returns a new array, you would have to start by creating an array of the required size, and then fill its contents
- Arrays are not immutable, because although their size cannot change, their contents can
- `ArrayLists` can have their size changed, if you write code that creates and returns a new `ArrayList`, it starts by creating one of size 0 and then increases its size by calling the add method on on it
- An `ArrayList<E>` actually works by having an array inside it, and a count saying how much of the array is currently in use, that is how it gives the effect of an array that changes its size

Generalised Selection Sort of an `ArrayList` with natural order

```
static <E extends Comparable<E>> void sort(ArrayList<E> list)
{
 int sortLength = list.size();
 while(sortLength>1)
   {
    int maxPos = 0;
    for(int i=1; i<sortLength; i++)
       if(list.get(i).compareTo(list.get(maxPos)>0))
           maxPos = i;
    sortLength--;
    int tmp = list.get(maxPos);
    list.set(maxPos,list.get(sortLength));
    list.set(sortLength,tmp);
   }
 }
```

Built-in Sorting Code (2)

- As we have seen, we may want to sort an `ArrayList` in an order other than that given by `obj1.compareTo(obj2)`, which is **natural order**
- If `list` refers to an `ArrayList<E>`, then `Collections.sort(list,comp)` will sort the `ArrayList<E>` using another `static` method in Java's class `Collections` where objects are compared by `comp.compare(obj1,obj2)`
- Then `comp.compareTo(obj1,obj2)` returns a negative number if what `obj1` refers to is less than what `obj2` refers to, a positive number if it is greater, and 0 if they are equal according to how the `Comparator<T>` object referred to by `comp` does the comparison
- Java has a built in interface:

```
interface Comparator<T>
{
  int compare(T obj1, T obj2);
}
```

Generalised Selection Sort of an `ArrayList<E>` with a `Comparator<E>`

```
static <E> void sort(ArrayList<E> list, Comparator<E> comp)
{
 int sortLength = list.size();
 while(sortLength>1)
   {
    int maxPos = 0;
    for(int i=1; i<sortLength; i++)
       if(comp.compare(list.get(i),list.get(maxPos))>0))
           maxPos = i;
    sortLength--;
    int tmp = list.get(maxPos);
    list.set(maxPos,list.get(sortLength));
    list.set(sortLength,tmp);
   }
 }
```

Example of `Comparator<T>` (1)

```
import java.util.Comparator;

class ScrabbleComp implements Comparator<String> {

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

 private int sscore(String word) {
   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};
   word = word.toLowerCase();
   int sum = 0;
   for(int i=0; i<word.length(); i++)
      sum+=scores[word.charAt(i)-'a'];
   return sum;
 }
}
```

Example of `Comparator<T>` (2)

```
import java.util.Comparator;

class ClosestTo implements Comparator<Rectangle>
{
 private Rectangle rec;

 public ClosestTo(Rectangle rrec) {
  rec=rrec;
 }

 public int compare(Rectangle rec1, Rectangle rec2) {
  int recArea = rec.area();
  int diff1 = Math.abs(recArea-rec1.area());
  int diff2 = Math.abs(recArea-rec2.area());
  return diff2-diff1;
 }
}
```

Sorting in ECS414U

- For the exam purpose of this module, you don't need to know the details of sorting algorithms, or the generalisation techniques making use of type variables
- If you were asked in the ECS414U exam to write code that sorts a collection of a particular type of object, code that just sorts a collection of that object is all that is required
- Marking is done automatically by running the code you write and checking it gives the correct result for the test data that the checking code uses
- So you can use whatever algorithm you find easiest to understand, there are no extra marks for writing more efficient code
- The main point of going through different algorithms for sorting here is to give you some more examples of the general practice of writing code that uses `ArrayList<E>`s