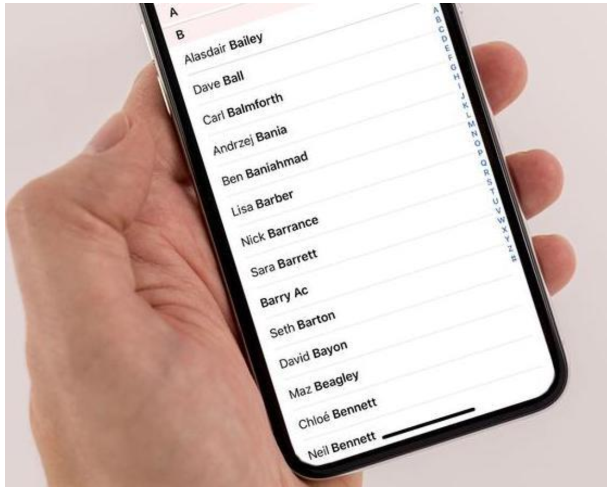
Ulster University

COM410 Programming in Practice

B2.1 An Introduction to Sorting

Sorting

- We are all familiar with arranging objects . . .



- Sorting is the process of arranging a group of items into some defined order based on specific criteria

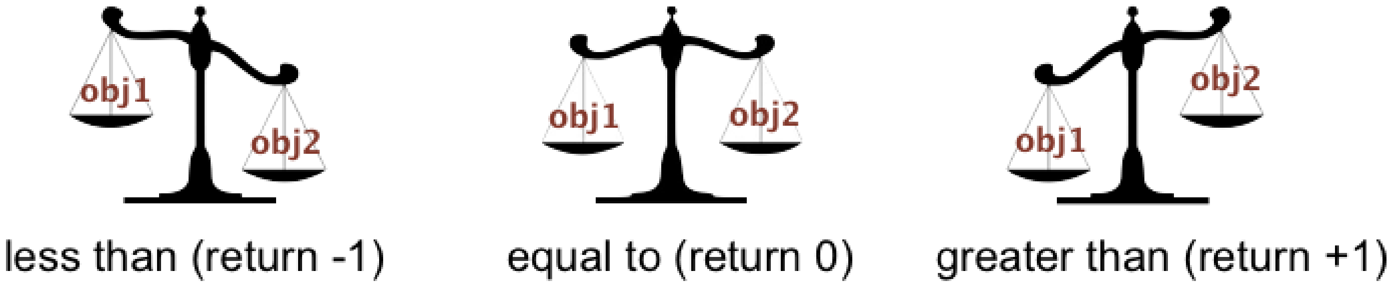
The Java `Comparable` Interface

- We must be able to compare one object to another (in order to define algorithms that can sort any set of objects)
- Although most of our examples will sort integers, the Java implementations given will sort any `Comparable` object (i.e. objects that implement the Comparable interface)
- `Comparable` contains one method `compareTo()` which is designed to return an integer that specifies the relationship between two objects:

```
obj1.compareTo(obj2)
```

The Java `Comparable` Interface

- Implement **`compareTo()`** so that **`obj1.compareTo(obj2)`**
  i. defines a total order between objects
  ii. returns a negative integer (usually `-1`), zero (`0`), or a positive integer (usually `1`) if **`obj1`** is less than, equal to or greater than **`obj2`**, respectively



Scenario – Comparing Cards

- In your `DataStructures` project modify the `Card` class to implement the `Comparable` interface.
  - Add the `compareTo()` method that accepts a card object as a parameter and returns 1 if the rank of the current card is greater than the parameter card, -1 if the rank of the current card is lower and 0 if the ranks are the same.
  - If you do not already have a `getRankValue()` method, you may need to implement another accessor method to return the rank value of a card so that they can be compared.
- Modify the `CardTest` class generate two random cards, compare their rank values, and report the result of the comparison.

Sorting

- Exactly how you compare two objects depends on the nature of the objects and their properties (title, author, size, height, colour, name, etc.)
- Currently, we seek algorithms to arrange the *n* entries in a collection, such that:

$$\textit{entry } 1 \leq \textit{entry } 2 \leq \ldots \leq \textit{entry } n$$

- Sorting an array is usually easier than sorting a chain of linked nodes

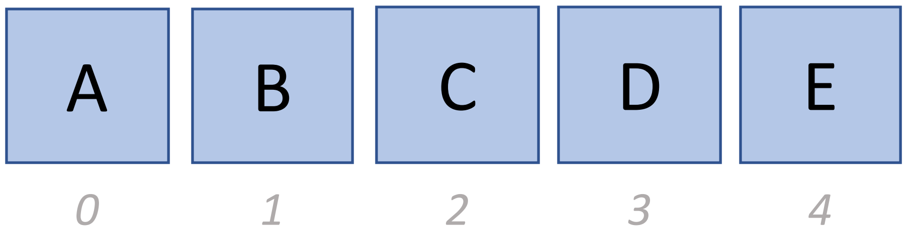
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

- Our algorithms will rearrange the *n* values in an array, such that:

$$a[0] \leq a[1] \leq \ldots \leq a[n - 1]$$

Swapping Elements

- An array of 5 elements `char[] letters = { “A”, “B”, “C”, “D”, “E”};`



Swapping Elements

- An array of 5 elements `char[] letters = { "A", "B", "C", "D", "E"};`
    - We want to swap the contents of elements 1 and 3

A	B	C	D	E
0	*1*	*2*	*3*	*4*

Swapping Elements

- An array of 5 elements `char[] letters = { “A”, “B”, “C”, “D”, “E”};`
  - We want to swap the contents of elements 1 and 3

A	B	C	D	E
0	*1*	*2*	*3*	*4*

  - Copying the value of one cell over another results in loss of data

Swapping Elements

- An array of 5 elements `char[] letters = { “A”, “B”, “C”, “D”, “E”};`
  - We want to swap the contents of elements 1 and 3
  - Solution is to use a temporary variable

A	D	C	B	E
0	*1*	*2*	*3*	*4*

B
temp

Stage 3

- Copy the temp variable back into the new position

```
char temp = letters[1];
letters[1] = letters[3];
letters[3] = temp;
```

Scenario – Manipulating Cards

- In your `DataStructures` project, Implement the class `RankedCards` with functionality as follows…
    - The application should generate 5 random cards and store them in an array
    - When all of the cards have been generated, find the card with the highest rank (if there is a tie, the first instance of the highest rank should be taken).
    - Swap the highest rank card so that it is in the last array position.
    - Print the collection of cards to the terminal window.

Bubble Sort

- Orders a list of values by repetitively comparing neighbouring elements and swapping their positions if necessary

  - Scan the list, exchanging adjacent elements if they are not in relative order (the effect is to “bubble” the highest value to the top)
  - Scan the list again, bubbling up the second highest value
  - Repeat until all elements are in their proper place in the list

- Each swap operation moves an item closer to its final correct position

Bubble Sort

- Consider the first pass through the array that results in the highest value being moved to the last element.

50	20	40	75	35

50	20	40	75	35

20	50	40	75	35

20	50	40	75	35

20	40	50	75	35

20	40	50	75	35

20	40	50	75	35

20	40	50	35	75

Bubble Sort

- Remaining passes...

20	40	50	35	75
20	40	50	35	75
20	40	50	35	75
20	40	50	35	75

20	40	35	50	75
20	40	35	50	75
20	40	35	50	75
20	35	40	50	75

Bubble Sort – version 1

- Pseudocode of algorithm for Bubble Sort that works on the entries in array a:

```
Algorithm bubbleSort(a)
// Sorts the entries of an array a.

set last position to length of the array – 1
set inner last to last position
loop i from 0 to last position - 1
   loop j from 0 to inner last - 1
       if entries at a[j] and a[j+1] are out of order
          swap a[j] with a[j+1]
   set inner last to inner last - 1
```

- Works OK – but potentially continues to check values long after the array has been sorted
- A better approach is to keep track of swaps that have been made and stop when no more are required.

Bubble Sort - improved

- Keep track of the last swap made and exit when none is required

```
Algorithm bubbleSort(a)
// Sorts the entries of an array a.

set first position to 0
set last position to length of the array - 1
while first position is less than the last position
   set last swap position to first position
   loop i from first position to last position - 1
       if entries at a[i] and a[i+1] are out of order
          swap a[i] with a[i+1]
          set last swap to i
   set last position to last swap
```

Scenario

- In a new project called `Sorting`, implement the improved Bubble Sort algorithm in a class called `BubbleSort` and confirm its operation by having it sort an array of 10 integers.
- Add code to show the state of the array at the beginning and end of the process
- Add code to count the number of comparisons and the number of swaps made – and report these values once the sort is complete.
    - Check its performance

      (i) when the array is in a random order

      (ii) when the array is already in the sorted order

      (iii) when the array is in reverse order

Efficiency of Bubble Sort

- The (improved) Bubble Sort maintains a record of the last exchange so redundant passes through the main loop are avoided.

  - Best case: if the array is already in ascending order, the bubble sort makes ***N*–1** comparisons and **0** swaps, so is **O(n)**

  - Worst case: if array is in descending order, process requires ***N*–1** passes; on pass *i*, there are ***N*–*i*** comparisons and ***N*– *i*** swaps, so outer loop and inner loop are both **O(n)**, so is of order **O(n^2)**

  - Average case: more complicated as some of the passes may be skipped, but the average number of passes and swaps are still **O(n)**, hence **O(n^2)**

Scenario

- Prove that the improved Bubble Sort has a time signature of **O(n^2)** by adding the the following functionality to your `BubbleSort` class.
  - Generate random integer arrays of size 100, 200, 400, 800, 1600, 3200 and 6400 elements and measure the time required to sort each.
  - Perform 1000 timed sorts on each size of array (with new random contents each time) and take the total execution time.
  - Present the results as a list of pairs of values showing the array size and total execution time for 1000 sort operations.
  - Note: remember to remove all non-essential code from the Bubble Sort method (i.e. code that generates output to the console or counts comparisons/swaps).