Outline

- equality and comparisons in Java
- sorting
- selection sort
- bubble sort
- insertion sort
- introduction to algorithm analysis

Java Equality details for Objects

- **four kinds of equality:**
- **== is true if and only if two references are to the exact same object**
  - for basic types, if the values are the same, e.g. x == 3 is true if x is an int with value 3
- **`object1.equals(object2)` is `true` if object1's `equals` method decides (*in its wisdom*) that they are the same**
- **`object1.compareTo(object2)` is `0` if the `compareTo` method decides (*in its wisdom*) that they are the same**
  - the result is negative if object1 < object2, and greater than 0 if object1 > object2)
- **`object0.compare(object1, object2)` is similar to `compareTo`, comparing object1 to object2**
- **the behavior of the last three depends on the implementation:**
  - ideally, they compare the contents of the object
  - ideally they are all consistent with each other

  but:
  - they may not do what you think
  - they may have bugs

properties of the equals method

- **the equals method should be:**
  - reflexive: `a.equals(a)` should always be true
  - symmetric: `a.equals(b) == b.equals(a)`
    - both true or both false
  - transitive: `(a.equals(b) && b.equals(c))` only if `a.equals(c)`
  - consistent: successive identical calls should return the same result
  - if a is not null, `a.equals(null)` should always be `false`
- **the equals method can be overridden for any class**
- **if equals is *not* overridden, `Object.equals` is the same as ==**

ordering objects in Java

- **two Java methods to find out which object is greater and which object is less:**
- **Comparable<T> interface, specifies the int compareTo(T x) method**
- **Comparator<T> interface, specifies the int compare(T x1, T x2) method (which could be static, but isn't – interfaces don’t list static methods)**
- **if comparing a to b:**
  - that is, if we are calling x.compareTo(y) or compare(x, y)
  - both methods return:
    - a value that is < 0 if x < y
    - 0 if x equals y
    - a value that is > 0 if x > y
  - these methods are not part of the Object class, so are not provided by all objects

compare and compareTo

- **given that `signum` is the sign function, returning -1 for negative numbers, 0 for zero, and 1 for positive numbers:**
- **both comparison methods should have the following properties**
  - symmetric:

    `Integer.signum(compare(a, b)) == - Integer.signum(compare(b, a))`
  - transitive: if `(((compare(a, b) > 0) && ((compare(b, c) > 0)),`

    then `((compare(a, c) > 0)`
  - consistent: if `(compare(a, b) == 0)`, then the sign of `(compare(a, c))` should be the same as the sign of `(compare(b, c))`
- **when sorting, can use either interface**
- **e.g. class Collections has two different methods for sorting:**
  - `static <T extends Comparable<? super T>> void sort(List<T> list)` works when elements are `Comparable`
  - `static <T> void sort(List<T> list, Comparator<? super T> c)` takes as parameter a `Comparator`
- **compare and compareTo are essentially the same, but some classes only provide one of the two**

Applications of sorting

- **finding duplicate elements in a list or an array**
  - or eliminating duplicates by making a list of elements, where each element occurs at most once
- **preparing to make future searches more efficient**
  - dictionary, phone book
- **presenting data in an appropriate format, e.g. for printing**
  - transactions in a bank statement are sorted by date
- **comparing two lists to find out which elements are in one, the other, or both**
- **merging multiple sorted collections into a new sorted collection, with or without duplicates**

Sorting with the Java Standard Library

- Java class `Arrays` has

```
public static void sort(x[] items);
public static void
     sort(x[] items, int fromIndex, int toIndex);
```

- for type x being any one of:
- `int` and all of the basic types except `boolean`
- `Object` – the objects must be `Comparable`
- `T` – if the last parameter is a `Comparator<T>`

Selection Sort

- **start with an unsorted array a**
- **find the smallest element in the array, at index *i***
- **swap the element at index *i* with the element at index 0**
- **now, find the smallest element in 1..a.length-1**
- **swap the elements at index *I'* and at index 1**
- **now, the sub-array with elements 0..1 is sorted**
- **now, find the smallest element in 2..a.length – 1**
  - see animations at https://en.wikipedia.org/wiki/Selection_sort
- **in-class exercise (in groups of 2 or 3): write code to implement selection sort**

Bubble Sort

- start with an unsorted array a
- loop through all the elements of array a, swapping any that are not sorted
- repeat until the array is sorted
- see animations at https://en.wikipedia.org/wiki/Bubble_sort
- trivia: bubble sort of an array of size n can be done in parallel on n processors

Insertion Sort

- in selection sort, find the smallest element, and put it in the next position
- in insertion sort, take the next element, and put it in the right place
- trivia: card players often use insertion sort to arrange their decks
- the sub-array at the beginning of the array is already sorted, just as in selection sort
- in selection sort, the next element e is always taken from the next index in the array
- elements in the sorted part of the array that are greater than e are shifted up one position, to free the position where element e will be stored

  see animations at https://en.wikipedia.org/wiki/Insertion_sort
- in-class exercise (in groups of 2 or 3): write code to implement insertion sort

Introduction to Algorithm Analysis: motivation

- **selection sort, bubble sort, and insertion sort all sort an array**
- **which one of these is better?**
  - assuming the implementations already exist
- **all of these algorithms sort in place, so they all use the same amount of memory space**
- **so the only difference may be in the time they take**
- **but the time depends on both the computing system and the data**
  - for example, bubble sort is very fast on data that is already sorted, or if sorting data stored on magnetic tape

Introduction to Algorithm Analysis: analysis

- **for each algorithm, we will have a best-case time, a worst-case time, and sometimes an average-case time**
- **the time taken for sorting depends on the size of the array: sorting 1000 elements takes longer than sorting 10 elements**
- **because computing systems vary, and to keep things simple, all constant-time operations are assumed to take the same amount of time**
  - so this analysis assumes that addition, multiplication, and comparisons, each take the same, constant time

Introduction to Algorithm Analysis: notation

- **If, even in the worst case, an algorithm takes constant time, we say it is O(1)**
  - for example, the time to access an element of an array is constant, no matter the size of the array
- **If the worst case of an algorithm takes time proportional to the size of the input, it is O(n), with n the size of the input**
  - in the worst case, finding out whether an unsorted array of integers contains a specific number (called linear search) takes time proportional to the size of the array
  - linear search in an array of 1,000,000 elements should be twice as fast as in an array of 2,000,000 elements
- **If an algorithm in the worst case takes time proportional to the square of n, we say it is $O(n^2)$**
- **We will also see O(log n), O(n log n), O(√n), and so on**
- **O() always looks at the worst case as n gets very large**
  - unless we are specifically looking for the best case or average case