Week 12 - Friday

COMP 2000

Last time

- What did we talk about last time?
- Java Collections Framework
  - `Map`
    - `HashMap`
    - `TreeMap`
  - `Set`
    - `HashSet`
    - `TreeSet`

Questions?

Project 4

Sorting

Sorting

- Computer scientists only have a few tricks
  - Searching through lists
  - Making decisions based on data
  - Doing stuff really fast
  - **And sorting!**
- We can sort huge lists of data so well that people don't even realize that it's a fascinating problem
- Hundreds of papers and many full books have been written on the problem of sorting efficiently

Why do we care?

- We need sorting directly
  - To rank things
  - To find the median
  - To find the mode
  - Before doing binary search
- Sorting is also the first preprocessing step in many fascinating algorithms
  - Maximum weighted interval selection
  - Convex hull
  - Closest points in a plane
  - ...countless others

But how do you do it?

- In COMP 1600, we learned bubble sort
- Earlier this semester, we learned merge sort
- In COMP 2100, we discuss many other interesting algorithms
- You should understand the ideas behind them…
- **But you should almost never write a sort yourself in industrial-strength code**
- There are libraries for that
  - They've been heavily tested, so you're confident they work
  - They've been tuned to perform incredibly well in a variety of circumstances

Sorting arrays

- Every language has its own libraries for sorting
- Let's start with sorting arrays
- It would be nice if every array just had a **`sort()`** method
  - But it doesn't!
- Instead, there's an **`Arrays`** (note the **`s`**) class with a number of useful static methods (with versions for arrays of every primitive type as well as **`Object`**):
  - **`sort()`**
  - **`binarySearch()`**
  - **`toString()`**
- To use it, import **`java.util.Arrays`**

Array sorting example

- Calling **`Arrays.sort()`** will sort arrays of **`byte`**, **`char`**, **`short`**, **`int`**, **`long`**, **`float`**, **`double`**, and **`String`**, always in ascending order
- Calling **`Arrays.toString()`** also produces a nice printable version of an array

```
// Obviously, data could also be input from the user or file
int[] numbers = {98, 50, 25, 30, 10, 56, 79, 86, 18, 92};
Arrays.sort(numbers);
// Output: [10, 18, 25, 30, 50, 56, 79, 86, 92, 98]
System.out.println(Arrays.toString(numbers));

String[] words = {"The", "quick", "brown", "fox", "jumps", "over",
"the", "lazy", "dog"};
Arrays.sort(words);
// Output: [The, brown, dog, fox, jumps, lazy, over, quick, the]
// Don't forget that uppercase letters have lower ASCII values
System.out.println(Arrays.toString(words));
```

Sorting other collections

- If you're sorting a collection (meaning **List**, **LinkedList**, **ArrayList**, **Vector**, etc.), you can use **Collections.sort()**
- When a collection has its own **sort()** method (as **ArrayList** does), use that, since it's tuned for performance on that collection

```java
Scanner file = new Scanner(new File(fileName));
LinkedList<String> words = new LinkedList<>();
while(file.hasNext())
     words.add(file.next());
file.close();
// Print out all the words in the file, sorted
Collections.sort(words);
for(String word : words)
     System.out.println(word);
```

Example

- The **mode** is the element that occurs in a list more than any other
- Example:
  - List: 7, 4, 6, 2, 1, 7, 6, 3, 7, 9
  - Mode: 7
- Find the mode of an array of **`int`** values
- Algorithm:
  - Sort the array (obviously)
  - Count neighboring items that are the same
    - If it's the same as the previous largest count of items, mark a **`noMode`** value **`true`**
    - If the count is bigger than previous counts, update the count and mark **`noMode`** **`false`**

Comparable Interface

What if you want to sort something else?

- Sorting numbers and **`String`** values makes sense
- But what if you want to sort a bunch of **`Wombat`** values?
- How do you order wombats?
  - Age?
  - Weight?
  - Name?
- What about breaking ties?

`Comparable<T>`

- If you want to sort an array or a list of some object, it must implement the **`Comparable<T>`** interface, where **`T`** is usually the type of the object itself
- The **`Comparable<T>`** interface has one method in it:

```
int compareTo(T other);
```

- An object that implements **`Comparable<T>`** will return:
  - A negative number if it comes before **`other`** in order
  - A positive number if it comes after **`other`** in order
  - Zero if it is equivalent to **`other`**
- It's usually not important what the values are, just whether they are positive or negative

Wombat example

- Wombat's are prized for their cuddliness, so we want to compare them by how fat they are
- By subtracting the other **Wombat** object's weight from our own, we get negative if we're smaller, positive if we're bigger, and zero if we weigh the same

```
public class Wombat implements Comparable<Wombat> {
        private int weight;
        private String name;
        public Wombat(String name, int weight) {
                this.name = name;
                this.weight = weight;
        }
        public int compareTo(Wombat other) {
                return weight - other.weight;
        }
        public int String getName() {
                return name;
        }
}
```

Person example

- In Western countries, people are usually alphabetized by their last name, with ties broken by their last name
- We can use the fact that **`String`** implements **`Comparable<String>`** to help us compare **`Person`** objects

```
public class Person implements Comparable<Person> {
        private String firstName;
        private String lastName;
        public Person(String firstName, String lastName) {
                this.firstName = firstName;
                this.lastName = lastName;
        }
        public int compareTo(Person other) {
                int difference = lastName.compareTo(other.lastName);
                if(difference == 0)
                        return firstName.compareTo(other.firstName);
                else
                        return difference;
        }
        public String getFirstName() {
                return firstName;
        }
}
```

Sorting Comparable objects

- As long as objects implement the **`Comparable`** interface, they can be sorted with **`Arrays.sort()`** or **`Collections.sort()`**
- If you try to sort things that don't implement **`Comparable`**, those methods will throw an **`IllegalArgumentException`**

```
Wombat[] wombats = new Wombats[4];
wombats[0] = new Wombat("Angelica", 31);
wombats[1] = new Wombat("Grover", 63);
wombats[2] = new Wombat("Hubert", 22);
wombats[3] = new Wombat("Beauregard", 28);
Arrays.sort(wombats);  //Sorted: Hubert, Beauregard, Angelica, Grover
List<Person> people = new ArrayList<>();

people.add(new Person("Martha", "Stewart"));
people.add(new Person("John", "Comerford"));
people.add(new Person("Kristen", "Stewart"));
Collections.sort(people); //Sorted: John Comerford, Kristen Stewart, Martha Stewart
```

Custom Comparators

What if things weren't designed to be sorted?

- What if the objects you're working with don't implement the **Comparable** interface?
- Or if you want to sort them in some other way?
- You can supply a custom **Comparator<T>** object to the **sort()** methods that will say how they should be compared
- The **Comparator<T>** interface contains one method you have to implement:

```
int compare(T a, T, b);
```

- It should return negative if **a** comes before **b**, positive if **a** comes after **b**, and zero if **a** and **b** are equivalent

Planet example

- Here's a simple class for **`Planet`**, a class they didn't expect to sort

```
public class Planet {
        private String name;
        private double radius;
        public Planet(String name, double radius) {
                this.name = name;
                this.radius = radius;
        }
        public double getRadius() {
                return radius;
        }
        public String getName() {
                return name;
        }
}
```

Sorting `Planet` objects by radius

- Since the **`Planet`** class doesn't implement **`Comparable`**, we have to make a **`Comparator`** to pass to the **`sort()`** method
- We have to make an anonymous inner **`Comparable`** class, using the **`Double.compare()`** method to help use order by radius

```
List<Planet> planets = new ArrayList<>();

planets.add(new Planet("Venus", 6051.8));
planets.add(new Planet("Earth", 6371.0));
planets.add(new Planet("Mars", 3389.5));
Comparator<Planet> comparator = new Comparator<Planet>() {
     int compare(Planet a, Planet b) {
          return Double.compare(a.getRadius(), b.getRadius());
     }
};
Collections.sort(planets, comparator);
// Order: Mars, Venus, Earth
```

Sorting `Planet` objects in Java 8

- Using Java 8 style, we could also create the **`Comparator`** object with the quicker (but slightly more confusing) **`->`** syntax

```
List<Planet> planets = new ArrayList<>();

planets.add(new Planet("Venus", 6051.8));
planets.add(new Planet("Earth", 6371.0));
planets.add(new Planet("Mars", 3389.5));

// Order: Mars, Venus, Earth
Collections.sort(planets, (a, b) ->
Double.compare(a.getRadius(), b.getRadius()));
```

Using Comparator for a different order

- If a class already implements the **Comparable** interface, you can still create a custom **Comparator** to sort in some different way
- Here, we sort **wombats** by name instead of weight
- You can also use a custom **Comparator** to sort objects in descending (reverse) order

```
Wombat[] wombats = new Wombats[4];
wombats[0] = new Wombat("Angelica", 31);
wombats[1] = new Wombat("Grover", 63);
wombats[2] = new Wombat("Hubert", 22);
wombats[3] = new Wombat("Beauregard", 28);

//Sorted: Angelica, Beauregard, Grover, Hubert
Arrays.sort(wombats, (a,b) -> a.getName().compareTo(b.getName()));
```

Rules for sorting

- If it's an array, use **`Arrays.sort()`**
- If it's a collection
  - Always use the **`sort()`** built into the collection if it has one
  - Otherwise, use **`Collections.sort()`**
- If you're sorting types with a natural ordering like **`int`** or **`String`**, that's all you need
- If you're sorting objects that don't have a natural ordering or you want to sort them in some unusual way, supply a custom **`Comparator`**

Quiz

Upcoming

Next time…

- UML diagrams
- Program design

Reminders

- Start Project 4
  - Get your teams figured out immediately!