Collections part 2

Summary

- You will know Collections
- You will recognize Map
- Also, you will know Lambda and streams

Stream Boost
Traversing collection

- (1) For-each
- (2) Iterator
- (3) Streams

Traversing collection

- (1) For-each

```
for (Object o : collection) {
        System.out.println(o);
}
```

Traversing collection

- (2) Iterator
  - an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired.

```
public interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove(); //optional
}
```

```
static void filter(Collection<?> c) {
  for (Iterator<?> it = c.iterator(); it.hasNext(); ) {
        if (!cond(it.next())) {
            it.remove();
        }
  }
}
```

Traversing collection

1. Aggregate operations are often used in conjunction with lambda expressions to make programming more expressive, using less lines of code. The following code sequentially iterates through a collection of shapes and prints out the red objects:

- (3) Aggregate Operation via Streams

```
for(Shape s : myShapesCollection.) {
    if(s.getColor == Color. RED) {
        System.out.println(e.getName());
    }
}
```

Traversing collection

1. Aggregate operations are often used in conjunction with lambda expressions to make programming more expressive, using less lines of code. The following code sequentially iterates through a collection of shapes and prints out the red objects:

- (3) Aggregate Operation via **Streams**

```
for(Shape s : myShapesCollection.) {
    if(s.getColor == Color. RED) {
            System.out.println(e.getName());
    }
}
```

```
myShapesCollection.stream()
 ....
```

Just pipelining here

Traversing collection

1. Aggregate operations are often used in conjunction with lambda expressions to make programming more expressive, using less lines of code. The following code sequentially iterates through a collection of shapes and prints out the red objects:

- (3) Aggregate Operation via Streams

```
for(Shape s : myShapesCollection.) {
    if(s.getColor == Color. RED) {
        System.out.println(e.getName());
    }
}
```

```
myShapesCollection.stream()
 .filter(s -> s.getColor() == Color.RED)
 ....
```

Intermediate state

Traversing collection

1. Aggregate operations are often used in conjunction with lambda expressions to make programming more expressive, using less lines of code. The following code sequentially iterates through a collection of shapes and prints out the red objects:

- (3) Aggregate Operation via Streams

```
for(Shape s : myShapesCollection.) {
    if(s.getColor == Color. RED) {
        System.out.println(e.getName());
    }
}
```

```
myShapesCollection.stream()
 .filter(s -> s.getColor() == Color.RED)
 .forEach(s -> System.out.println(s.getName()));
```

Terminated

Traversing collection

1. Aggregate operations are often used in conjunction with lambda expressions to make programming more expressive, using less lines of code. The following code sequentially iterates through a collection of shapes and prints out the red objects:

- (3) Aggregate Operation via Streams

```
for(Shape s : myShapesCollection.) {
   if(s.getColor == Color. RED) {
         System.out.println(e.getName());
   }
}
```

```
myShapesCollection.stream()
 .filter(s -> s.getColor() == Color.RED)
 .forEach(s -> System.out.println(s.getName()));
```

Synchronous

Traversing collection

1. Aggregate operations are often used in conjunction with lambda expressions to make programming more expressive, using less lines of code. The following code sequentially iterates through a collection of shapes and prints out the red objects:

2. parallel stream, which might make sense if the collection is large enough and your computer has enough cores:

- (3) Aggregate Operation via Streams

```
for(Shape s : myShapesCollection.) {
    if(s.getColor == Color. RED) {
        System.out.println(e.getName());
    }
}
```

```
myShapesCollection.stream()
 .filter(s -> s.getColor() == Color.RED)
 .forEach(s -> System.out.println(s.getName()));
```

Parallel

```
myShapesCollection.parallelStream()
 .filter(e -> e.getColor() == Color.RED)
 .forEach(e -> System.out.println(e.getN
```

Traversing collection

- (3) Aggregate Operation via Streams

```
myShapesCollection.stream()
 .filter(s -> s.getColor() == Color.RED)
 .forEach(s -> System.out.println(s.getName()));
```

1. Aggregate operations are often used in conjunction with lambda expressions to make programming more expressive, using less lines of code. The following code sequentially iterates through a collection of shapes and prints out the red objects:

2. parallel stream, which might make sense if the collection is large enough and your computer has enough cores:

3. Convert the elements of a Collection to String objects, then join them, separated by commas:

```
String joined = elements.stream()
   .map(Object::toString)
   .collect(Collectors.joining(", "));
```

Intermediate state

Terminated

Traversing collection

- (3) Aggregate Operation via Streams

1. Aggregate operations are often used in conjunction with lambda expressions to make programming more expressive, using less lines of code. The following code sequentially iterates through a collection of shapes and prints out the red objects:

2. parallel stream, which might make sense if the collection is large enough and your computer has enough cores:

3. Convert the elements of a Collection to String objects, then join them, separated by commas:

4. Sum the salaries of all employees:

Not

```
String joined = elements.stream()
   .map(Object::toString)
   .collect(Collectors.joining(", "));
```

```
int total = employees.stream()
   .collect(Collectors
                  .summingInt(Employee.
                                                            ));
```

Map

- A Map is an object that maps keys to values.
- A map cannot contain duplicate keys: Each key can map to at most one value.
- It models the **mathematical *function*** abstraction.
- The Map interface includes methods for basic operations (such as put, get, remove, containsKey, containsValue, size, and empty), bulk operations (such as putAll and clear), and collection views (such as keySet, entrySet, and values).

```
Map<String, Integer> m = new Ha
m.put("myPin", 1234);
int pin = m.get("myPin");
```

Iterate over a map

Key	Value

```
Map<..> m = ..
for (KeyType key : m.keySet()) {
    System.out.println(key);
}


for (Map.Entry<KeyType, ValType> e : m.entrySet()) {
     System.out.println(e.getKey() + ": " + e.getValue());
}
```

Map Example

generates a frequency table of the words found in its argument list. The frequency table maps each word to the number of times it occurs in the argument list.

```
public class FrequncyTable {
  public static void main(String[] args) {
    Map<String, Integer> m = new HashMap<String, Integer>();
     // Initialize frequency table from command line
    for (String a : args) {
         Integer freq = m.get(a);
         m.put(a, (freq == null) ? 1 : freq + 1);
    }
    System.out.println(m.size() + " distinct words:");
    System.out.println(m);
} }
```

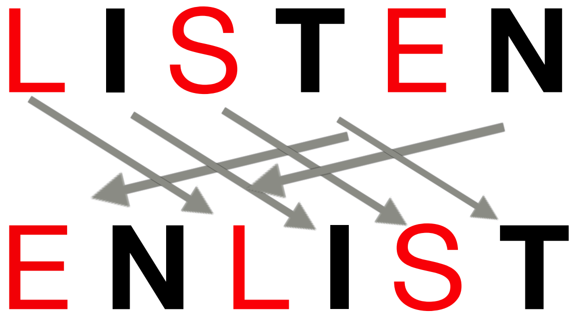
Map Implementations See the impact!

- Input
  - *java Freq if it is to be it is up to me to delegate*
- Output when using HashMap
  - *8 distinct words: {to=3, delegate=1, be=1, it=2, up=1, if=1, me=1, is=2}*
- change the implementation type of the Map from HashMap to TreeMap.
  - *8 distinct words: {be=1, delegate=1, if=1, is=2, it=2, me=1, to=3, up=1}*
  - ***ordered***
- change the implementation type of .. HashMap to LinkedHashMap
  - *8 distinct words: {if=1, it=2, is=2, to=3, be=1, up=1, me=1, d*
  - ***first to occur***

Example Multimap

Input: text file

-> dictionary of words containing the same letters



Input: Large book, e.g., War and piece
And identify all anagrams

```
private static String alphabetize(Stri
    char[] a = s.toCharArray();
    Arrays.sort(a);
    return new String(a);
}
```

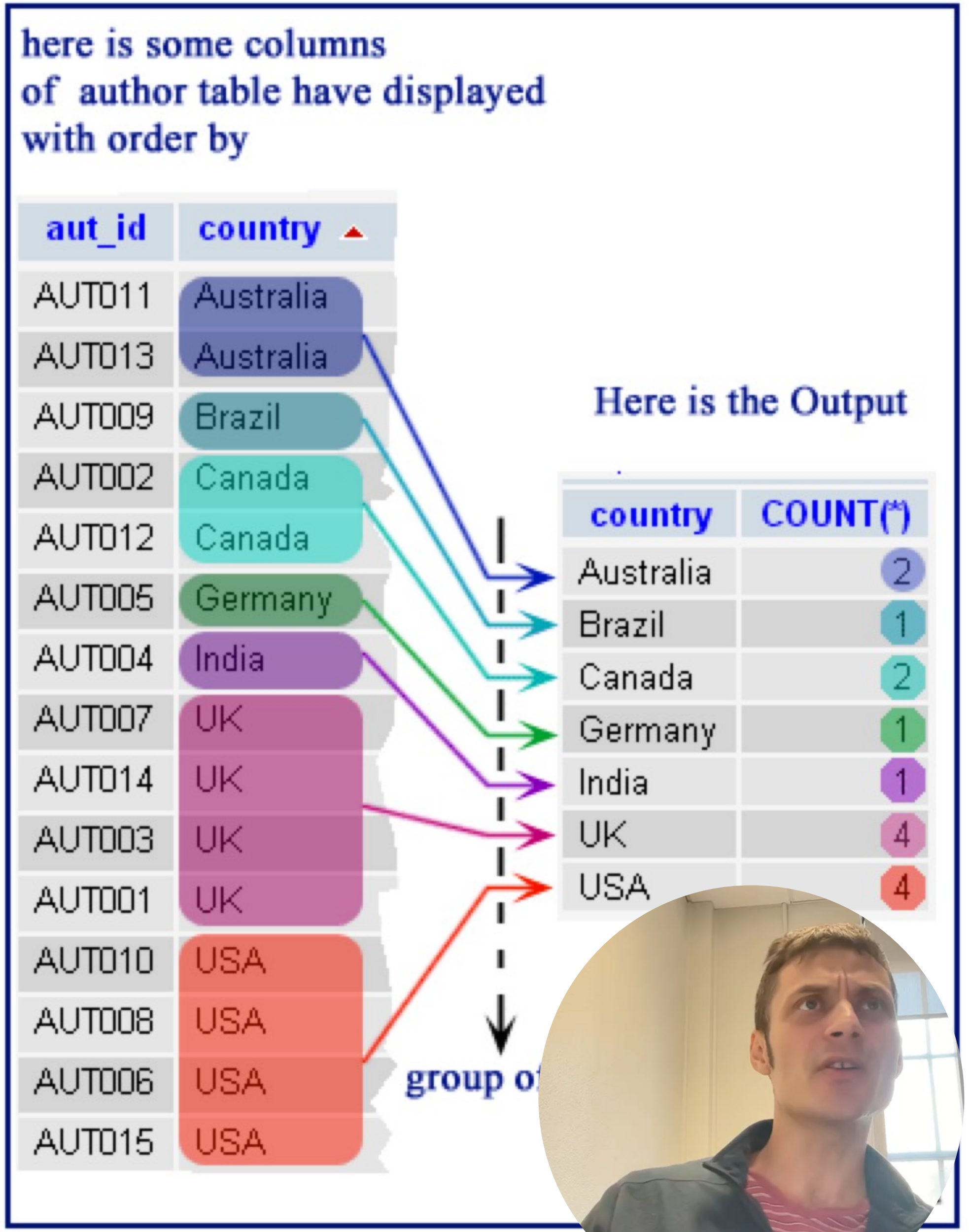
```
public class Anagrams {
   public static void main(String[] args) {
        int minGroupSize = Integer.parseInt(args[1]);
       // Read words from file & put into a multimap
       Map<String, List<String>> map = new HashMap<String, List
       try {
          Scanner s = new Scanner(new File(args[0]));
          while (s.hasNext()) {
             String word = s.next();
             String alpha = alphabetize(word);
             List<String> l = map.get(alpha);
             if (l == null)
                map.put(alpha, l=new ArrayList<String>());
                l.add(word);
            }
       } catch (IOException e) { System.err.println(e); System.exit(1
       // Print all permutation groups above size threshold
       for (List<String> l : map.values())
             if (l.size() >= minGroupSize)
                   System.out.println(l.size() + ": " + l);
}
```

Example Multimap

- Running this program on a 173,000-word dictionary file with a minimum anagram group size of eight produces the following output.
- 9: [estrin, inerts, insert, inters, niters, nitres, sinter, triens, trines]
- 8: [lapse, leaps, pales, peals, pleas, salep, sepal, spale]
- 8: [aspers, parses, passer, prases, repass, spares, sparse, spears]
- 10: [least, setal, slate, stale, steal, stela, taels, tales, teals, tesla]
- 8: [enters, nester, renest, rentes, resent, tenser, ternes, treens]
- 8: [arles, earls, lares, laser, lears, rales, reals, seral]
- 8: [earings, erasing, gainers, reagins, regains, reginas, searing, s
- 8: [peris, piers, pries, prise, ripes, speir, spier, spire]
- ...

The Group by operation

- Quite common for data summaries
    - Excel Pivot Table
- Let us check out Streams..



Map + streams : Group By operations

- // Group employees by department

```
- Map<Department, List<Employee>> byDept = employees.stream()
  .collect(Collectors.groupingBy(Employee::getDepartment));
```

- // Compute sum of salaries by department

```
- Map<Department, Integer> totalByDept = employees.stream()
  .collect(Collectors.groupingBy(Employee::getDepartment,
  Collectors.summingInt(Employee::getSalary)));
```

- // Partition students into passing and failing

```
- Map<Boolean, List<Student>> passingFailing = students.stream()
  .collect(Collectors.partitioningBy(s -> s.getGrade()>=
  PASS_THRESHOLD));
```

- // Classify Person objects by city

```
- Map<String, List<Person>> peopleByCity =
  personStream.collect(Collectors.groupingBy(Person::get
```

Tools

- A List may be sorted as follows.

```
- Collections.sort(l)
```

// basic types

- My type?

```
- Collections.sort(list, comparator)
```

- Comparator

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

```
public class Name implements Comparable<Name> {
   private final String firstName, lastName;
   public Name(String firstName, String lastName) {
      if (firstName == null || lastName == null) throw new NullPointerException();
      this.firstName = firstName;
      this.lastName = lastName;
  }
  public String firstName() { return firstName; }
  public String lastName() { return lastName; }

  public int compareTo(Name n) {
     int lastCmp = lastName.compareTo(n.lastName);
     return (lastCmp != 0 ? lastCmp : firstName.compareTo(n.firstNa
} }
```

```java
public class Name implements Comparable<Name> {
    private final String firstName, lastName;
    public Name(String firstName, String lastName) {
        if (firstName == null || lastName == null) throw new NullPointerException();
        this.firstName = firstName;
        this.lastName = lastName;
   }
   public String firstName() { return firstName; }
   public String lastName() { return lastName; }
   public boolean equals(Object o) {
        if (!(o instanceof Name)) return false;
        Name n = (Name) o;
        return n.firstName.equals(firstName) && n.lastName.equals(lastName);
   }
   public int hashCode() {
        return 31*firstName.hashCode() + lastName.hashCode();
   }
   public String toString() { return firstName + " " + lastName; }
   public int compareTo(Name n) {
      int lastCmp = lastName.compareTo(n.lastName);
      return (lastCmp != 0 ? lastCmp : firstName.compareTo(n.firstNa
} }
```

Previous Name

```
public class NameSort {
   public static void main(String[] args) {
      Name nameArray[] = {
          new Name("John", "Smith"),
          new Name("Karl", "Ng"),
          new Name("Jeff", "Smith"),
          new Name("Tom", "Rich") };

     List<Name> names = Arrays.asList(nameArray);
     Collections.sort(names);
     System.out.println(names);
  }
}

//[Karl Ng, Tom Rich, Jeff Smith, John Smith]
```

Most common use of comparator (appart)

```
public class EmpSort {
    static final Comparator<Employee> SENIORITY_ORDER
                        = new Comparator<Employee>() {
        public int compare(Employee e1, Employee e2) {
            return e2.hireDate().compareTo(e1.hireDate());
        }
    };
     // Employee database
    static final Collection<Employee> employees = ... ;

   public static void main(String[] args) {
      List<Employee> e = new ArrayList<Employee>(employees
      Collections.sort(e, SENIORITY_ORDER);
      System.out.println(e);
   }
}
```

More

- Deque
- Interface SortedSet
  - *impl* TreeSet *(needs a comparator)*
- Interfaces SortedMap
  - *impl TreeMap (needs a comparator)*