Queues and Iterators

Lecture 5

Menu

- Examples of using Map
- Queues and Priority Queues
- Classes/Interfaces that accompany collections
  - Iterator
  - Iterable

Example of using Map

- Find the highest frequency word in a file

  ⇒ must count frequency of every word.

  ie, need to associate a count (int) with each word (String)

  ⇒ use a Map of word–count pairs:

- Two Steps:
  - construct the counts of each word: countWords(file) → map
  - find the highest count findMaxCount(map) → word

```
System.out.println( findMaxCount( countWords(file) ) );
```

Example of using Map

```
/** Construct histogram of counts of all words in a file */
public Map<String, Integer> countWords(Scanner sc){
    // construct new map
    // for each word in file
    //     if word is in the map, increment its count
    //     else, put it in map with a count of 1
    // return map
}

/** Find word in histogram with highest count */
public String findMaxCount(Map<String, Integer> counts){
    // for each word in map
    //    if has higher count than current max, record it
    // return current max word
}
```

Example of using Map

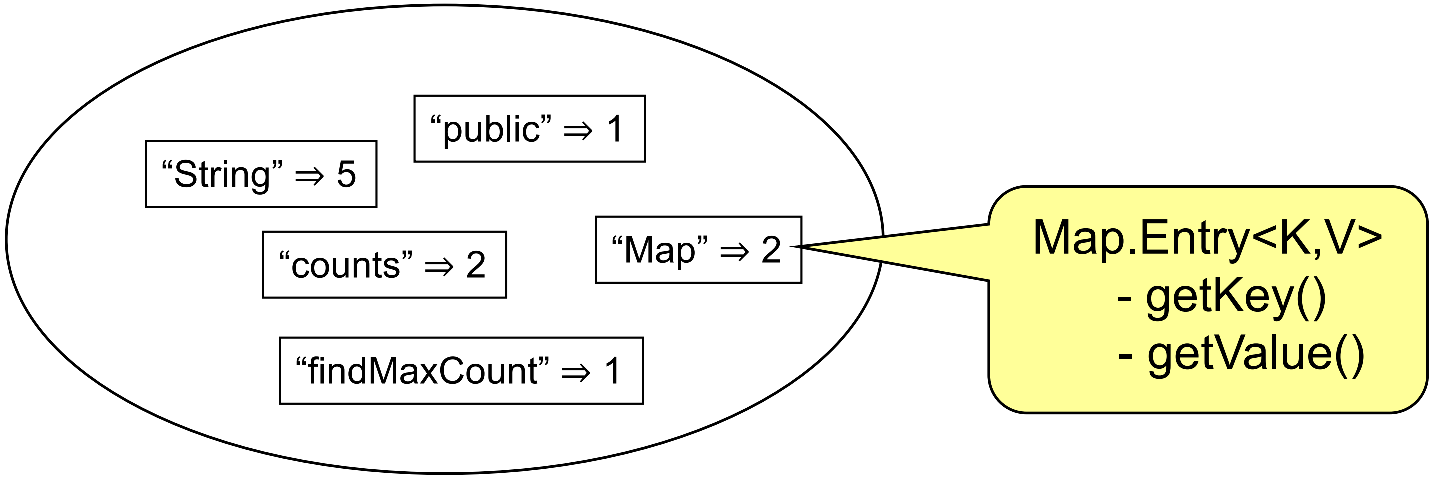
```
/** Construct histogram of counts of all words in a file */
public Map<String, Integer> countWords(Scanner scan){
    Map<String, Integer>  counts = new HashMap<String, Integer> ();
    while (scan.hasNext()){
      String word = scan.next();
      if ( counts.containsKey(word) )
          counts.put(word, counts.get(word)+1);
      else
          counts.put(word, 1);
    }
    return counts;
}
/** Find word in histogram with highest count */
public String findMaxCount(Map<String, Integer> counts){
    // for each word in map
    //    if has higher count than current max, record it
    // return current max word
}
```

Iterating through Map: keySet

```
/** Find word in histogram with highest count */
public String findMaxCount(Map<String, Integer> counts){
    String maxWord = null;
    int maxCount = -1;
    for (String word : counts.keySet() ){
        int count = counts.get(word);
        if (count > maxCount){
            maxCount = count;
            maxWord = word;
        }
    }
    return maxWord;
}
```

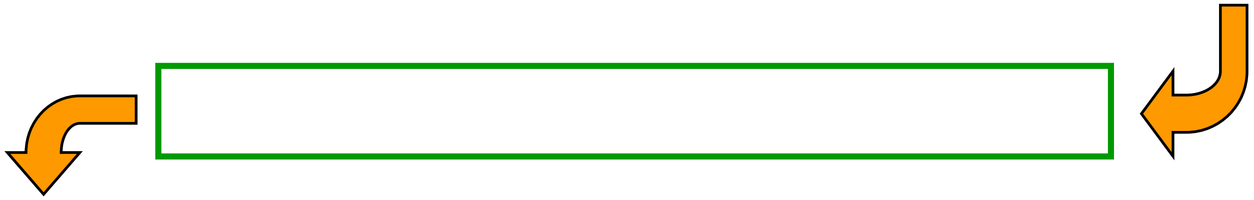
Iterating through Map: entrySet

```
public String findMaxCount(Map<String, Integer> counts){
    String maxWord = null;
    int maxCount = -1;
    for (Map.Entry<String, Integer> entry : counts.entrySet() ){
      if (entry.getValue() > maxCount){
          maxCount = entry.getValue();
          maxWord = entry.getKey();
      }
    }
    return maxWord;
}
```

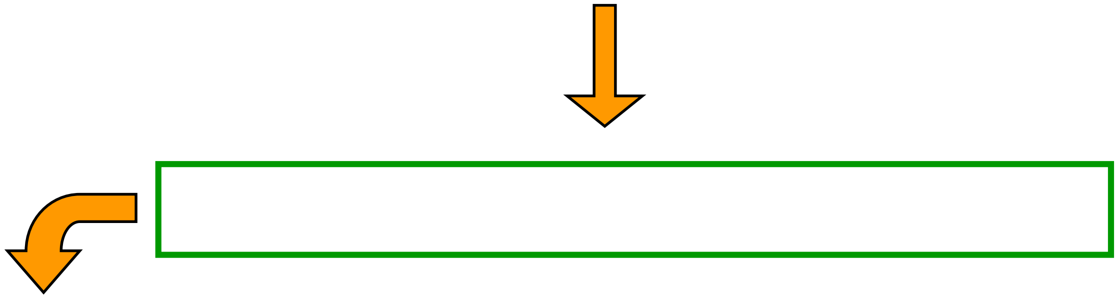


Queues

- Queues are like/unlike Stacks
  - Collection of values with an order
  - Constrained access:
    - Only remove from the front
  - Two varieties:
    - **Ordinary queues**: only add at the back



    - **Priority queues**: add or remove with a given priority



Queues

- Used for
  - Operating Systems, Network Applications, Multi-user Systems
    - Handling requests/events/jobs that must be done in order
    - (memory pool holding such requests are often called a “buffer” in this context)
  - Simulation programs
    - Representing queues in the real world (traffic, customers, deliveries, ….)
    - Managing events that must happen in the future
  - Search Algorithms
    - Computer Games
    - Artificial Intelligence
- Java provides
  - a Queue interface
  - several classes: **LinkedList, PriorityQueue**

Queue Operations

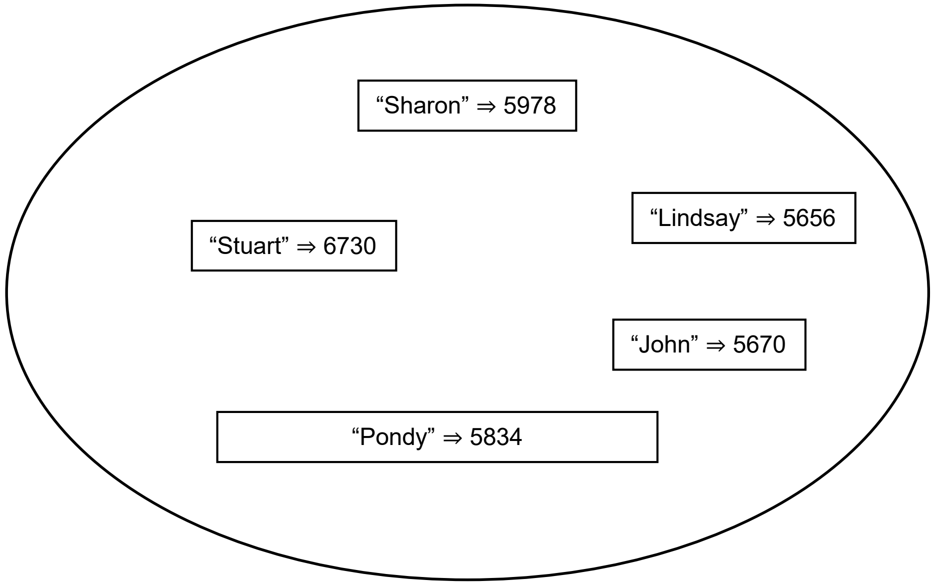
- **offer**(value) ⇒ boolean
  - add a value to the queue
  - (sometimes called “enqueue”)
- **poll**() ⇒ *value*
  - remove and return value at front/head of queue or null if the queue is empty
  - (sometimes called “dequeue”, like “pop”)
- **peek**() ⇒ *value*
  - return value at head of queue, or null if queue is empty (doesn’t remove from queue)
- **remove**() and **element**()
  - like poll() and peek(), but throw exception if queue is empty.

Iteration and "for each" loop

- Standard "for each" loop with collections:

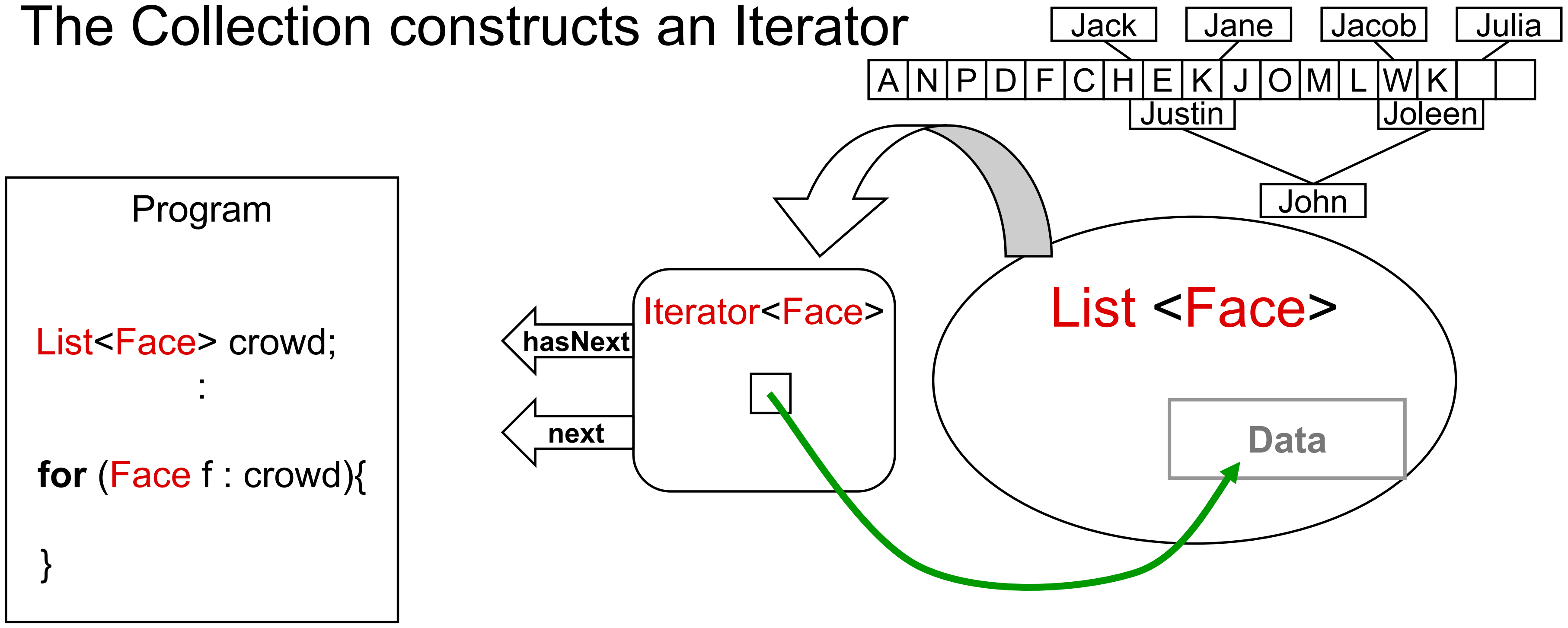
```
for (Face face : crowd) {
        face.render(canvas);
    }
for (Map.Entry<String,Integer> entry : phonebook.entrySet()){
        textArea.append(entry.getKey()+" : "+entry.getValue());
    }
```

- Uses Iterators.



Why Iterators?

- Program cannot get inside the Collection object
- The Collection constructs an Iterator



- Iterator may access inside of collection
- Iterator provides elements one at a time.
- Each Collection class needs an associated Iterator class

Iterator Interface

- Operations on Iterators:
  - **hasNext**() : *returns* true *iff there is another value to get*
  - **next**() : *returns the next value*
- Standard pattern of use:

```
Iterator<type > itr  = construct iterator
while (itr.hasNext() ){
    type var = itr.next();
    … var …
}
```

- Almost same as the “for each” loop:

```
for (type var :  collection ){
    … var …
```

Iterators and Iterable

- But, **the <u>"for each" loop requires an Iterable</u>**:

```
for (type var :  Iterable <type> ){
   … var …
}
```

eg, all Collections

```
Iterable <T>

  public Iterator<T> iterator();
```

```
Iterator <T>
        public boolean hasNext();
        public T next();
```

```
Iterator<type > itr  = construct iterator

while (itr.hasNext() ){
   type var = itr.next();
   … var …
}
```

Creating Iterators

- Iterators are not just for Collection objects:
  - Anything that generates a sequence of values
  - Scanner
  - Pseudo Random Number generator :

```
public class NumCreator implements Iterator<Integer>{
        private int num  = 1,

        public boolean hasNext(){
            return true;
        }
        public Integer next(){
            num = (num * 92863) % 104729 + 1;
            return num;
        }
   :
  Iterator<Integer> lottery = new NumCreator();
  for (int i = 1; i<1000; i++)
        textArea.append(lottery.next()+ "\n");
```

Creating an Iterable

- Class that provides an Iterator:
  - eg: A NumberSequence representing an infinite arithmetic sequence of numbers, with a starting number and a step size,
    eg 5, 8, 11, 14, 17,….

```
public class NumberSequence implements Iterable<Integer>{
    private int start;
    private int step;
    public NumberSequence(int start, int step){
        this.start = start;
        this.step = step;
    }
    public Iterator<Integer> iterator(){
        return new NumberSequenceIterator(this);
    }
```

Creating an Iterator for an Iterable

```
    private class NumberSequenceIterator implements Iterator<Integer>{
        private int nextNum;
        private NumberSequence source;
        public NumberSequenceIterator(NumberSequence ns){
            source = ns;
            nextNum = ns.start;
        }
        public boolean hasNext(){
            return true;
        }
        public Integer next(){
            int ans = nextNum;
            nextNum += ns.step;
            return ans;
        }
    }   // end of NumberSequenceIterator class
}   // end of Number Sequence class
```

Using the Iterable

- Can use the iterable object in the for each loop:

```
for (int n : new NumberSequence(15, 8)){
     System.out.printf(“next number is %d \n“, n);
}
```

- Can use the iterator of the iterable object directly.

```
Iterator<Integer> iter = new NumberSequence(15, 8).iterator();
processFirstPage(iter);
for (int p=2; p<maxPages; p++)
    processNextPage(p, iter);
```

(passing iterator to different methods to deal with)

Q&A

- Java has specified a “Queue” interface. (T or F)
- Java does not have any class support for “Priority Queue”. (T or F)
- peek() operation under the Queue interface will throw an exception if the queue is empty. (T or F)
- poll() operation under the Queue interface will throw an exception if the queue is empty. (T or F)
- There is an element() method under the Queue interface. (T or F)
- Iterable is an interface specification for a class that is equipped with an Iterator.
- Iterator is an interface specification for a class that can generate iterative elements.

Summary

- Queues and Priority Queues
- Classes/Interfaces that accompany collections
  - Iterator
  - Iterable

Readings

- [Mar07] Read 3.7, 3.4
- [Mar13] Read 3.7, 3.4