
Discussion 5

— Iterators & Iterables —

Administrivia

- MT 1 is Wednesday, 8-10pm! (material up to 2/15)
 - Material is up to and including 2/15
- Note that you will be allowed to bring **one letter size page of handwritten notes** (front and back) to the first midterm
- Check out @1605 for MT1 Resources

Good luck! We're proud of all of your hard work, and we're here to talk if you're feeling particularly stressed this week. (reach out to your mentor GSI!)

What are iterators/iterables?

- Remember them from 61A?
 - These concepts in Java are conceptually similar to those in Python
- An iterator is to a bookmark as an iterable is to a book
 - An **iterable** is what you're iterating through (the book)
 - An **iterator** is the actual machine doing the iterating
 - An iterator's hasNext() tells you whether or not there are any pages left, and its next() method keeps track of where you are in the book

Key Components: Iterator

```
import java.util.Iterator;

public class BookmarkIterator implements Iterator<String> {
    public BookmarkIterator(String book) {...} // constructor
    public boolean hasNext() {...}
    public String next() { // returns a letter on each iteration}
}
```

```
String book = "ab";  
BookmarkIterator bIterator = new BookmarkIterator(book);  
int spareTime = 1;  
while (spareTime > 0) {  
    if (bIterator.hasNext()) {  
        System.out.println(bIterator.next());  
    }  
    spareTime--;  
}  
  
System.out.println("I didn't have time to read"  
    + bIterator.next());
```

Output:

```
String book = "ab";  
BookmarkIterator bIterator = new BookmarkIterator(book);  
int spareTime = 1;  
while (spareTime > 0) {  
    if (bIterator.hasNext()) {  
        System.out.println(bIterator.next());  
    }  
    spareTime--;  
}  
  
System.out.println("I didn't have time to read"  
    + bIterator.next());
```

← True

Output:


```
String book = "ab";
BookmarkIterator bIterator = new BookmarkIterator(book);
int spareTime = 1;
while (spareTime > 0) {
    if (bIterator.hasNext()) {
        System.out.println(bIterator.next());
    }
    spareTime--;
}

System.out.println("I didn't have time to read"
    + bIterator.next());
```

Output:

a

```
String book = "ab";  
BookmarkIterator bIterator = new BookmarkIterator(book);  
int spareTime = 1;  
while (spareTime > 0) {  
    if (bIterator.hasNext()) {  
        System.out.println(bIterator.next());  
    }  
    spareTime--;  
}
```




```
System.out.println("I didn't have time to read"  
    + bIterator.next());
```

Output:

a



```
String book = "ab";  
BookmarkIterator bIterator = new BookmarkIterator(book);  
int spareTime = 1;  
while (spareTime > 0) {  
    if (bIterator.hasNext()) {  
        System.out.println(bIterator.next());  
    }  
    spareTime--;  
}  
  
System.out.println("I didn't have time to read"  
    + bIterator.next());
```



Output:

a

```
String book = "ab";  
BookmarkIterator bIterator = new BookmarkIterator(book);  
int spareTime = 1;  
while (spareTime > 0) {  
    if (bIterator.hasNext()) {  
        System.out.println(bIterator.next());  
    }  
    spareTime--;  
}  
  
System.out.println("I didn't have time to read"  
    + bIterator.next());
```



Output:

```
a  
I didn't have time  
to read b
```

When/why do we use iterators/iterables?

- If you want to use an enhanced for loop, you must make sure that you are looping through an iterable
 - If you build your own iterable, make sure that it `implements Iterable<T>`, and that it defines an `iterator()` method
- Instead of using an enhanced for loop (as shown in lecture), you can do “ugly iteration” using an iterator (example shown previously)

Agenda

- Questions 1 & 2
- If there's time: Question 3

Q1: Iterators Warmup

1. If we were to define a class that implements the interface `Iterable<Integer>`, what method(s) would this class need to define? Write the function signature(s) below.
2. If we were to define a class that implements the interface `Iterator<Integer>`, what method(s) would this class need to define? Write the function signature(s) below.
3. What's one difference between `Iterator` and `Iterable`?

Q1: Iterators Warmup

1. If we were to define a class that implements the interface `Iterable<Integer>`, what method(s) would this class need to define? Write the function signature(s) below.
`public Iterator<Integer> iterator()`

2. If we were to define a class that implements the interface `Iterator<Integer>`, what method(s) would this class need to define? Write the function signature(s) below.
`public boolean hasNext()`
`public Integer next()`

3. What's one difference between `Iterator` and `Iterable`?

We use `Iterable` to support the enhanced for loop, and `Iterator` to define custom iterators that maintain some state about iteration.

Q2: OHQueue

```
1  public class OHRequest {
2      public String description;
3      public String name;
4      public OHRequest next;
5
6      public OHRequest(String description, String name, OHRequest next) {
7          this.description = description;
8          this.name = name;
9          this.next = next;
10     }
11 }
```

Q2: OHQueue

First, let's define an iterator. Create a class `OHIterator` that implements an iterator over `OHRequest` objects that only returns requests with good descriptions. Our `OHIterator`'s constructor will take in an `OHRequest` object that represents the first `OHRequest` object on the queue. We've provided a function, `isGood`, that accepts a description and says if the description is good or not.

```
1  import java.util.Iterator;
2  public class OHIterator _____ {
3      OHRequest curr;
4
5      public OHIterator(OHRequest queue) {
6
7      }
8
9      public boolean isGood(String description) {
10         return description != null && !description.equals("") && description.length() > 5;
11     }
12
13
14
15
16
17
18
19
20
21 }
```


Q2: OHQueue

First, let's define an iterator. Create a class `OHIterator` that implements an iterator over `OHRequest` objects that only returns requests with good descriptions. Our `OHIterator`'s constructor will take in an `OHRequest` object that represents the first `OHRequest` object on the queue. We've provided a function, `isGood`, that accepts a description and says if the description is good or not.

```
1  import java.util.Iterator;
2  import java.util.NoSuchElementException;
3
4  public class OHIterator implements Iterator<OHRequest> {
5      OHRequest curr;
6
7      public OHIterator(OHRequest original) {
8          curr = original;
9      }
10
11
12     public boolean isGood(String description) {
13         return description != null && description.length() > 5;
14     }
```

Q2: OHQueue

```
15
16     @Override
17     public boolean hasNext() {
18         while (curr != null && !isGood(curr.description)) {
19             curr = curr.next;
20         }
21         return curr != null;
22     }
23
24     @Override
25     public OHRequest next() {
26         if (!hasNext()) {
27             throw new NoSuchElementException();
28         }
29         OHRequest currRequest = curr;
30         curr = curr.next;
31         return currRequest;
32     }
33 }
```

Now, define a class `OfficeHoursQueue`. We want our `OfficeHoursQueue` to be iterable, so that we can process `OHRequest` objects with good descriptions. Our constructor will take in an `OHRequest` object representing the first request on the queue.

Q2: OHQueue

```
1  import java.util.Iterator;
2
3  public class OfficeHoursQueue ----- {
4
5
6      public OfficeHoursQueue (OHRequest queue) {
7
8      }
9
10
11
12
13
14
15 }
```

Now, define a class `OfficeHoursQueue`. We want our `OfficeHoursQueue` to be iterable, so that we can process `OHRequest` objects with good descriptions. Our constructor will take in an `OHRequest` object representing the first request on the queue.

Q2: OHQueue

```
1  import java.util.Iterator;
2  import java.util.NoSuchElementException;
3
4  public class OfficeHoursQueue implements Iterable<OHRequest> {
5      OHRequest queue;
6
7      public OfficeHoursQueue (OHRequest queue) {
8          this.queue = queue;
9      }
10
11      @Override
12      public Iterator<OHRequest> iterator() {
13          return new OHIterator(queue);
14      }
15  }
```

Fill in the main method below so that you make a new `OfficeHoursQueue` object and print the names of people with good descriptions. Note : the main method is part of the `OfficeHoursQueue` class.

Q2: OHQueue

```
1  public class OfficeHoursQueue ... {
2      ....
3
4      public static void main(String [] args) {
5          OHRequest s1 = new OHRequest("Failing my test for get in arrayDeque, NPE", "Pam", null);
6          OHRequest s2 = new OHRequest("conceptual: what is dynamic method selection", "Michael", s1);
7          OHRequest s3 = new OHRequest("git: what does checkout do.", "Jim", s2);
8          OHRequest s4 = new OHRequest("help", "Dwight", s3);
9          OHRequest s5 = new OHRequest("debugging get(i)", "Creed", s4);
10
11
12         for (_____ : _____) {
13
14         }
15
16     }
```

Fill in the main method below so that you make a new OfficeHoursQueue object and print the names of people with good descriptions. Note : the main method is part of the OfficeHoursQueue class.

Q2: OHQueue

```
1  public class OfficeHoursQueue ... {
2      ....
3
4      public static void main(String[] args) {
5          OHRequest s1 = new OHRequest("Failing my test for get in arrayDeque, NPE", "Pam", null);
6          OHRequest s2 = new OHRequest("conceptual: what is dynamic method selection", "Michael", s1);
7          OHRequest s3 = new OHRequest("git: what does checkout do.", "Jim", s2);
8          OHRequest s4 = new OHRequest("help", "Dwight", s3);
9          OHRequest s5 = new OHRequest("debugging get(i)", "Creed", s4);
10         OfficeHoursQueue q = new OfficeHoursQueue(s5);
11         for (OHRequest o : q) {
12             System.out.println(o.name);
13         }
14     }
```

Q3: Thank you, next

Now that we have our `OfficeHoursQueue` from problem 2, we'd like to add some functionality. We've noticed that occasionally in office hours, the system will put someone on the queue twice. It seems that this happens whenever the description contains the words "thank u." To combat this, we'd like to define a new iterator, `TYIterator`.

If the current item's description contains the words "thank u," it should skip the next item on the queue. As an example, if there were 4 `OHRequest` objects on the queue with descriptions ["thank u", "thank u", "im bored", "help me"], calls to `next()` should return the 0th, 2nd, and 3rd `OHRequest` objects on the queue. Note: we are still enforcing good descriptions on the queue as well!

hint : to check if a description contains the words "thank u", you can use
`curr.description.contains("thank u")`

Q3: Thank you, next

```
1 public class TYIterator ----- {  
2  
3     public TYIterator(OHRequest queue) {  
4  
5     }
```


Q3: Thank you, next

```
1
2 public class TYIterator extends OHIterator {
3     public TYIterator(OHRequest original) {
4         super(original);
5     }
6
7     @Override
8     public OHRequest next() {
9         OHRequest result = super.next();
10        if (curr != null && curr.description.contains("thank u")) {
11            curr = curr.next;
12        }
13        return result;
14    }
15 }
```

Any questions?
Good luck on the midterm!

Check out @1605 for MT 1 Resources!

The following slides are old (didn't want to delete them, but they aren't relevant for Sp19)

List

- An ordered collection or sequence
- `add(element);` // adds element to the end of the list
- `add(index, element);` // adds element at the given index
- `get(index);` // returns element at the given index
- `size();` // the number of elements in the list

Set

- A (usually unordered) collection of **unique** elements
 - Calling `add("hi")` twice doesn't change anything about the set
- `add(element);` // adds element to the collection
- `contains(object);` // checks if set contains object
- `size();` // number of elements in the set
- `remove(object);` // removes specified object from set

Map

- A collection of key-value mappings
 - The keys of a map are unique
 - Analogous to a dictionary in python
- `put(key, value);` // adds key-value pair to the map
- `get(key);` // returns value for the corresponding key
- `containsKey(key);` // checks if map contains the specified key
- `keySet();` // returns set of all keys in map

Stacks and Queues and Deques

- Linear collections with rules about item addition/removal
- Stacks are Last In First Out (LIFO)
- Queues are First In First Out (FIFO)
- Deques combine both



Priority Queue

- Like a queue but each element has a priority associated with it that determines the ordering of removal
- `add(e);` // adds element `e` to the priority queue
- `peek();` // looks at the highest priority element, but does not remove it from the PQ
- `poll();` // pops the highest priority element from the PQ

Solving problems with ADTs

- Don't worry about implementation, assume the ADTs correctly implement their given set of methods
- Writing pseudocode might help you
 - It also might not, depends on you
- Try out a few of the ADTs and compare their usefulness in the situation
- Think about correctness first, and then efficiency

(a) Given a news article, find the frequency of each word used in the article.

Use a map. When you encounter a word for the first time, put the key into the map with a value of 1. For every subsequent time you encounter a word, get the value, and put the key back into the map with its value you just got, plus 1.

(b) Given an unsorted array of integers, return the array sorted from least to greatest.

Use a priority queue. For each integer in the unsorted array, enqueue the integer with a priority equal to its value. Calling dequeue will return the largest integer; therefore, we can insert these values from index length-1 to 0 into our array to sort from least to greatest.

(c) Implement the forward and back buttons for a web browser

Use two stacks, one for each button. Each time you visit a new web page, add the previous page to the back button's stack. When you click the back button, add the current page to the forward button stack, and pop a page from the back button stack. When you click the forward button, add the current page to the back button stack, and pop a page from the forward button stack. Finally, when you visit a new page, clear the forward button stack.

BiDividerMap

- `put(k, V);` // put a key, value pair
- `getByKey(K);` // get the value corresponding to a key
- `getByValue(V);` // get the key corresponding to a value
- `numLessThan(K);` // return number of keys in map less than K

BiDividerMap

- Create two maps, $\text{Map}\langle K, V \rangle$ and $\text{Map}\langle V, K \rangle$.
 - How does this change insertion into the data structure compared to a normal map?
- numLessThan?
 - Get the list of keys (remember keySet?), sort it ($N \log N$) and iterate or binary search
 - Or keep a sorted list of keys in addition to the two maps, keeping it ordered as insertions occur

MedianFinder

- `add(x);` // adds x to the collection of numbers
- `median();` // returns the median from a collection of numbers

MedianFinder

- Use a list to store numbers, then sort it and return the middle index element when median() is called
- Two priority queues: lessThanMedian (max PQ) and greaterThanMedian (minPQ)
 - Always keep track of the current median and store the rest of the numbers in the appropriate priority queue
 - Rebalance as necessary when adding. i.e. if you lessThanMedian has size 4 and greaterThanMedian has size 5 and you the number you're adding will go in greaterThanMedian, move the current median to lessThanMedian and the min of greaterThanMedian becomes your new median.

Define a Queue using Stacks

- Brainstorm first before writing any code
- Check your work with a small example

```
public class Queue {  
    private Stack stack = new Stack();  
    public void push(E element) {  
        Stack temp = new Stack();  
        while (!stack.isEmpty()) {  
            temp.push(stack.poll());  
        }  
        stack.push(element);  
        while (!temp.isEmpty()) {  
            stack.push(temp.poll());  
        }  
    }  
    public E poll() {return stack.poll(); }  
}
```

```
public class Queue {  
    private Stack stack = new Stack();  
    public void push(E element) { stack.push(element); }  
    public E poll() { return poll(stack.pop()); }  
    private E poll(E previous) {  
        if (stack.isEmpty()) {  
            return previous;  
        }  
        E current = stack.poll();  
        E toReturn = poll(current);  
        push(previous);  
        return toReturn;  
    }  
}
```