CS 61B Spring 2024

Comparators, Iterators

Discussion 04: February 12, 2024

1 OHQueue

Meshan is designing the new 61B Office Hours Queue. The code below for `OHRequest` represents a single request. It has a reference to the `next` request. `description` and `name` contain the description of the bug and name of the person on the queue, and `isSetup` marks the ticket as being a setup issue or not.

```
public class OHRequest {
    public String description;
    public String name;
    public boolean isSetup;
    public OHRequest next;

    public OHRequest(String description, String name, boolean isSetup, OHRequest next) {
        this.description = description;
        this.name = name;
        this.isSetup = isSetup;
        this.next = next;
    }
}
```

(a) Create a class `OHIterator` that implements an `Iterator` over `OHRequest`s and only returns requests with good descriptions (using the `isGood` function). Our `OHIterator`'s constructor takes in an `OHRequest` that represents the first `OHRequest` on the queue. If we run out of office hour requests, we should throw a `NoSuchElementException` when our iterator tries to get another request, like so:

```
throw new NoSuchElementException();
```

Solution:

```
public class OHIterator implements Iterator<OHRequest> {
    private OHRequest curr;

    public OHIterator(OHRequest request) {
        curr = request;
    }

    public static boolean isGood(String description) { return description.length() >= 5; }

    @Override
    public boolean hasNext() {
        while (curr != null && !isGood(curr.description)) {
            curr = curr.next;
        }
        return curr != null;
    }

    @Override
    public OHRequest next() {
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        OHRequest temp = curr;
        curr = curr.next;
        return temp;
    }
}
```

Explanation: The `OHRequest` object `queue` passed into `OHIterator`'s constructor represents the first `OHRequest` on the queue. Initializing `curr` to `queue` in the constructor allows our `OHIterator` to start at this first request. Since `OHIterator` implements an Iterator over `OHRequests`, we must provide implementations for the interface methods `hasNext()` and `next()`. The `hasNext()` method handles checking whether there are more `OHRequests`. However, we only want requests with good (as defined by `isGood`) descriptions, so we must check the descriptions of each `OHRequest` and skip over the ones with bad descriptions before determining whether there are `OHRequests` left.

(b) Define a class `OHQueue` below: we want our `OHQueue` to be `Iterable` so that we can process `OHRequest` objects with good descriptions. Our constructor takes in the first `OHRequest` object on the queue.

Solution:

```
public class OHQueue implements Iterable<OHRequest> {
    private OHRequest request;

    public OHQueue(OHRequest request) {
        this.request = request;
    }

    @Override
    public Iterator<OHRequest> iterator() {
        return new OHIterator(request);
    }
}
```

Explanation: If we want our `OHQueue` to be `Iterable`, `OHQueue` has to implement the interface `Iterable`. A condition of this is implementing the methods of the interface (which in the case of `Iterable`, is the `iterator()` method). As our `OHQueue` processes `OHRequest` objects, `iterator()` in `OHQueue` should return an `OHIterator` over `OHRequest` objects.

(c) Suppose we notice a bug in our office hours system: if a ticket's description contains the words "thank u", it is put on the queue twice. 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, because we know the next item is an accidental duplicate from our buggy system. 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.

To check if a `String s` contains the substring "thank u", you can use: `s.contains("thank u")`

Hint - we've already enforced good descriptions with our regular `OHIterator`*. Using inheritance, how can we reuse that functionality without repeating ourselves? Also, notice that* `OHIterator`*'s instance variables are* **`private`***, so we can't access them from subclasses of* `OHIterator`*.*

Solution:

```
public class TYIterator extends OHIterator {
    public TYIterator(OHRequest queue) {
        super(queue);
    }

    @Override
    public OHRequest next() {
        OHRequest result = super.next();
        if (result.description.contains("thank u")) {
            super.next();
        }
        return result;
    }
}
```

(d) Now assume the `OHQueue` uses a `TYIterator` as its iterator. Fill in the blanks to print only the names of tickets from the queue beginning at `s1` with good descriptions, skipping over duplicate descriptions that contain "thank u". What would be printed after we run the `main` method?

Solution:

```
public static void main(String[] args) {
    OHRequest s5 = new OHRequest("I deleted all of my files, thank u", "Elana", true, null);
    OHRequest s4 = new OHRequest("conceptual: what is Java", "Mihir", false, s5);
    OHRequest s3 = new OHRequest("git: I never did lab 1", "Kevin", true, s4);
    OHRequest s2 = new OHRequest("help", "Angel", false, s3);
    OHRequest s1 = new OHRequest("no I haven't tried stepping through", "Ashley", false, s2);

    OHQueue q = new OHQueue(s1);
    for (OHRequest r: q) {
        System.out.println(r.name);
    }
}
```

Overall, we print:

```
Ashley
Kevin
Mihir
Elana
```

(e) Meshan would like to find a way to prioritize setup tickets on the queue so that they appear at the top. He wants to implement this based on the `isSetup` field of each `OHRequest`, but sometimes students forget to set it to `true`, so he decides to use `description` as backup to break ties.

Fill in the `compare` method of `OHRequestComparator` below. First, if one but not both of the `OHRequest`s have their `isSetup` set to **`true`**, the one with `isSetup` set to **`true`** should take priority (ie. earlier on the queue). If both or neither of the `OHRequest`s have their `isSetup` set to **`true`**, tiebreak with the `description`: the `description` has to **exactly match "setup"** in order to be counted as a setup issue. If both requests have such descriptions, it's a true tie and return 0.

*Hint - if o1 is prioritized over o2, then o1 is considered **less** than o2*

Solution:

```
public class OHRequestComparator implements Comparator<OHRequest> {
    @Override
    public int compare(OHRequest o1, OHRequest o2) {
        boolean isO1DescSetup = o1.description.equals("setup");
        boolean isO2DescSetup = o2.description.equals("setup");
        if (o1.isSetup && !o2.isSetup) {
            return -1;
        } else if (!o1.isSetup && o2.isSetup) {
            return 1;
        } else if (isO1DescSetup && !isO2DescSetup) {
            return -1;
        } else if (!isO1DescSetup && isO2DescSetup) {
            return 1;
        }
        return 0;
    }
}
```

Alternate:

```
public class OHRequestComparator implements Comparator<OHRequest> {
    @Override
    public int compare(OHRequest o1, OHRequest o2) {
        if (o1.isSetup == o2.isSetup) {
            boolean isO1DescSetup = o1.description.equals("setup");
            boolean isO2DescSetup = o2.description.equals("setup");
            return Boolean.compare(isO1DescSetup, isO2DescSetup);
        }
        return Boolean.compare(o1.isSetup, o2.isSetup);
    }
```

```
}
```

Explanation: The `compare` method should return a negative integer if `s1` has higher priority than `s2`, zero if the two requests are of equal priority, and a positive integer if `s1` has lower priority than `s2`. We are given that if either (but not both) request has their `isSetUp` set to true, the request with the true value receives higher priority. The two possible combinations of this scenario are covered in the first two if statements. We only check `description` if both or neither requests have `isSetUp` set to true, so the `description` checks must come after those of `isSetUp`.

The alternate uses the same idea, but exploits the fact that `Boolean.compare` compares two `booleans` for us. This allows us to condense the if-statements. Note that we use `==` to check for equality on primitives and `.equals` to check for equality on reference types (e.g. `String`).