CS 61B
Fall 2025

Comparators, Iterators, and Iterables

Discussion 04: September 22, 2025

1 Inheritance Syntax

Suppose we have the classes below:

```
public class ComparatorTester {
    public static void main(String[] args) {
        String[] strings = new String[] {"horse", "cat", "dogs"};
        System.out.println(Maximizer.max(strings, new LengthComparator()));
    }
}

public class LengthComparator implements Comparator<String> {
    @Override
    public int compare(String a, String b) {
        return a.length() - b.length();
    }
}

public class Maximizer {
    /**
     * Returns the maximum element in items, according to the given Comparator.
     */
    // public static <T> T max(T[] items, Comparator<T> c) {
    public static <String> String max(String[] items, Comparator<String> c) {
        ...
        int cmp = c.compare(items[i], items[maxDex]);
        ...
    }
}
```

(a) Suppose we omit the `compare` method from `LengthComparator`. Which of the following will fail to compile?

- ◯ `ComparatorTester.java`
- ◯ `LengthComparator.java`
- ◯ `Maximizer.java`
- ◯ `Comparator.java`

`LengthComparator`, because it is claiming to be a `Comparator`, but it is missing a `compare` method.

(b) Suppose we omit `implements Comparator<String>` in `LengthComparator`. Which file will fail to compile?

- ☐ `ComparatorTester.java`
- ☐ `LengthComparator.java`
- ☐ `Maximizer.java`

☐ `Comparator.java`

`ComparatorTester`, because we are trying to provide a `LengthComparator` (which isn't a `Comparator`) to the method `max`, which expects a `Comparator`.

`LengthComparator`, because `compare` is no longer overriding anything, thus causing the `@Override` to trigger a compiler error.

(c) Suppose we removed `@Override`. What are the implications?

The code will work fine, but it's best practice to say "Override" to prevent typos and make our code more clear.

2 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();
```

```
public class OHIterator ____________________________________________________ {
    private OHRequest curr;

    public OHIterator(OHRequest request) {

        ________________________________________;

    }

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

    @Override
    ________________ _______________ ______________________________ {

        while (_____________________________________________________________) {

            ___________________________________;

        }

        ___________________________________;

    }

    @Override
    ________________ _______________ ______________________________ {

        if (___________________________________) {

            throw __________ ________________________________________________;

        }

        ___________________________________;

        ___________________________________;

        ___________________________________;

    }
}
```

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.

```
public class OHQueue ____________________________________________________ {
    private OHRequest request;
    public OHQueue(OHRequest request) {

        ___________________________________________;
    }

    @Override
    ________________ _________________ ______________________________ {

        ___________________________________________;
    }
}
```

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 adjust our implementation of `OHIterator`'s `next()`.

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")`

```
@Override
_____________ ___________ ______________ {

    if (___________________________________) {

            throw __________ ________________________________________________;

    }
    ____________________________________;

    ____________________________________;

    if (_____________________________________________) {

        ______________________;
    }
    return ______________________;
}
```

Solution:

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

(d) Now assume the **OHQueue** uses the modified **OHIterator** 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?

```
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", "Ethan", false, s3);
    OHRequest s1 = new OHRequest("no I haven't tried the debugger", "Ashley", false,
s2);

    OHQueue q = ____________________________________;

    for (_____________________________________________) {

        ____________________________________________;
    }
}
```

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", "Ethan", false, s3);
    OHRequest s1 = new OHRequest("no I haven't tried the debugger", "Ashley", false,
s2);

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

Overall, we print:

```
Ashley
Kevin
Mihir
Elana
```