

1 Equals (Lecture 10)

from Fall 2025 Midterm 1

Fill in the `equals` method below. Two `GoatMeasurement` objects are considered equal if and only if they have the same `name`, and their `sizes` differ by at most 0.1. Assume `name` is never null.

Hint: You may find `Math.abs` useful.

Syntax hint: `if (o instanceof Dog uddaDog)`

```
public class GoatMeasurement {
    public String name;
    public double size;

    @Override
    public boolean equals(Object o) {

        if (o instanceof GoatMeasurement other) {

            return (name.equals(other.name) &&
                    (Math.abs(size - other.size) < 0.1));

        }
        return false;
    }
}
```

2 DefaultQueue (Lecture 9)

When we write **default** methods, we have no choice but to use our existing abstractions. This is because we don't have any knowledge of or access to the underlying base reality. Suppose we have a **MyQueue** interface that we want to implement. We want to include three default methods in the interface: **clear**, **remove** and **appendAll**. Fill in these methods in the code below.

```
public interface MyQueue<E> {
    void enqueue(E element); // adds an element to the end of the queue
    E dequeue();             // removes and returns the front element of the queue
    boolean isEmpty();       // returns true if the queue is empty
    int size();              // returns the number of elements in the queue
    // removes all items from the queue
    default void clear() {

        while (!isEmpty()) {
            dequeue();
        }

    }

    // removes all items equal to item from the queue
    // the remaining items should be in the same order as they were before
    // use .equals to compare items rather than ==
    default void remove(E item) {

        int i = 0;
        int currSize = size();
        while (i < currSize) {
            E currItem = dequeue();
            if (!currItem.equals(item)) {
                enqueue(currItem);
            }
            i += 1;
        }

    }

    // appends all items from the other queue into this queue
    // this method should be non-destructive on the otherQueue!
    default void appendAll(MyQueue<E> otherQueue) {

        int n = otherQueue.size();
        for (int i = 0; i < n; i += 1) {
            E item = otherQueue.dequeue();
            enqueue(item);
            otherQueue.enqueue(item);
        }

    }
}
```

3 Every Other Character (Lecture 10)

Implement an `EveryOtherCharacterIterator` that iterates over every other character of a `String`, starting with the character at index 0.

```
import java.util.Iterator;
import java.util.NoSuchElementException;

public class EveryOtherCharacterIterator implements Iterator<Character> {
    private String str;
    private int index;

    public EveryOtherCharacterIterator(String s) {
        str = s;
        index = 0;
    }

    @Override
    public boolean hasNext() {
        return index < str.length();
    }

    @Override
    public Character next() {
        if (!hasNext()) { throw new NoSuchElementException(); }
        char result = str.charAt(index);
        index += 2;
        return result;
    }
}
```

As an example, the code below creates an `EveryOtherCharacterIterator` and uses it to print every other character of "ghooautt":

```
EveryOtherCharacterIterator eoci = new EveryOtherCharacterIterator("ghooautt");
while (eoci.hasNext()) {
    IO.print(eoci.next()); // prints goat
}
```

- (a) Why does our class implement `Iterator<Character>` and not `Iterator<String>`?

Each call to `next()` produces a single character, not a whole `String`. Generic type parameters must be reference types, and the reference type corresponding to the primitive `char` is `Character` — so an iterator that yields one character at a time is an `Iterator<Character>`. (If we implemented `Iterator<String>`, `next()` would have to return a `String`, i.e. a sequence of characters.)

- (b) Why can't we write the following? What would we need to change about the `EveryOtherCharacterIterator` to make this work?

```
for (char c : eoci) {
    IO.print(c);
}
```

The enhanced for loop only works on objects whose class implements **Iterable<Character>**. Java calls the object's **iterator()** method to get an iterator. **EveryOtherCharacterIterator** is an **Iterator**, but it does not implement **Iterable<Character>**, so it cannot be used with the enhanced for loop. We'd have to change the implements line so that it says **implements Iterator<Character>, Iterable<Character>**, and add the **iterator()** method that **Iterable** requires.

4 Default (Lecture 9)

Suppose we have a **MyStack** interface that we want to implement. We want to add two default methods to the interface: **insertAtBottom** and **flip**. Fill in these methods in the code below.

```
public interface MyStack<E> {
    void push(E element); // adds an element to the top of the stack
    E pop();               // removes and returns the top element of the stack
    boolean isEmpty();     // returns true if the stack is empty
    int size();            // returns the number of elements in the stack

    // inserts the item at the bottom of the stack using push, pop, isEmpty, and size
    private void insertAtBottom(E item) {

        if (isEmpty()) {
            push(item);
            return;
        }

        E topElem = pop();
        insertAtBottom(item);
        push(topElem);
    }

    // flips the stack upside down (hint: use insertAtBottom)
    default void flip() {

        // Base case
        if (isEmpty()) {
            return;
        }
        // Pop top
        E topElem = pop();
        // Recursively reverse the remainder
        flip();
        // Insert the popped element at the bottom
        insertAtBottom(topElem);
    }
}
```