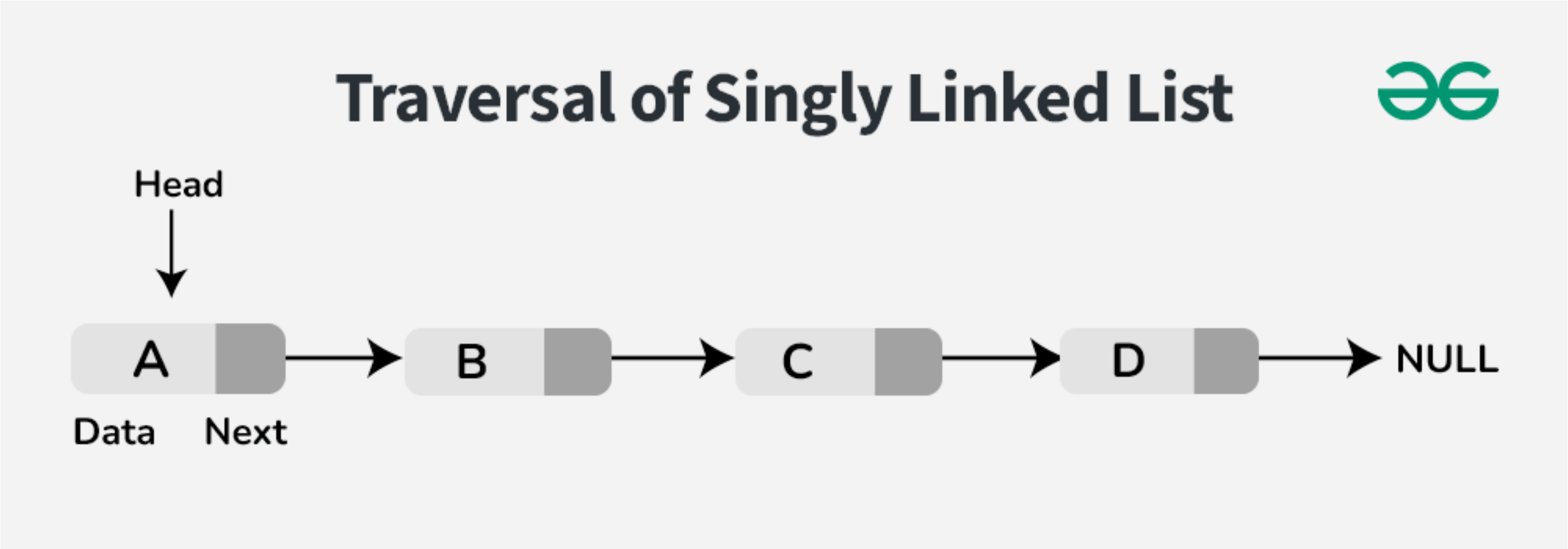
CS62 Class 11: Iterators & Comparators

Sorting



Iterator: an interface that tells us how to get the to the next element (e.g., node.next)

Comparator: an interface that tells us how to compare elements (e.g., node1.data > node2.data?)

Last week review

- Stacks: LIFO (last in, first out). Queues: FIFO (first in, first out). Want to make operations (push/pop, enqueue/dequeue) O(1) time. Ideal implementation for a stack is a singly linked list where we push/pop from the head. Ideal implementation for a queue is a singly linked list with a tail pointer.
- Practice: How would you implement a stack using two queues? What are the time complexities of push and pop?

Last week review

- Approach 1: O(1) push, O(n) pop
  - Push: enqueue to Q1 (which holds the elements of the stack)
  - Pop: transfer all but one element in Q1 to an empty Q2. Dequeue last element in Q1. Make Q1 = Q2 and Q2 empty.
- Approach 2: O(n) push, O(1) pop
  - Push: Enqueue to Q2, which is empty. Transfer all elements in the rest of Q1 to Q2. Make Q1 = Q2 and Q2 empty.
  - Pop: dequeue from Q1 (which holds the elements of the stack).

Agenda

- New chapter: Sorting! Why sorting?
- Iterables & Iterators
- Comparables & Comparators

Why study sorting?

- We’re constantly sorting things: e.g., sorting flights by price, contacts by last name, files by size, emails by day sent, neighborhoods by zipcode, etc.
- Good example of how to compare the performance of different algorithms for the same problem.
- Sorting your data will often be a good starting point when solving other problems (keep that in mind for interviews).
- Sorting definition: the process of arranging n elements of a collection in non-decreasing order (e.g., numerically, lexicographically, etc).
    - Why non decreasing instead of increasing? Each element should be ≥ the one before it (increasing is strictly >).
- To sort data in a data structure, we must first be able to iterate through the data structure…

Iterators

Traversing our own `ArrayList`

- Let's assume we have the following code snippet:

```
ArrayList<String> csClasses = new ArrayList<String>();
myList.add("cs51");
myList.add("cs54");
myList.add("cs62");
```

- The (sometimes unnecessarily verbose) story so far:

```
for (int i = 0; i < csClasses.size(); i++){
     System.out.println(csClasses.get(i);
}
```

- What we would like to do instead:

```
for(String course: csClasses){
  System.out.println(course);
}
```

We need to implement the *Iterable* and *Iterator* interfaces so Java knows how to make our data structures iterable in this loop short hand!

How to make your data structures iterable?

1. Implement `Iterable` interface.
2. Make a private class that implements the `Iterator` interface.
3. Implement `iterator()` method to return an instance of the private class in step 2.

Example: making `ArrayList` iterable

```
public class ArrayList<E> implements List<E>, Iterable<E> {
    //…
  public Iterator<E> iterator() {
    return new ArrayListIterator();
   }

  private class ArrayListIterator implements Iterator<E> {
    private int i = 0;
    public boolean hasNext() {
      return i < size;
    }
    public E next() {
      return data[i++];
    }
  }
```

Step 1

Step 3 (note return type)

Step 2 (nested private class)

Step 4: write public hasNext() and next() methods in your private class

Review question: what does data[i++] do? Why not data[++i]? (Can you remember an earlier in class activity?)

`Iterable<E>` Interface

- Interface that allows an object of a class that implements it to be the target of a for-each loop.

```
interface Iterable<E>{
  //returns an iterator over elements of type E
  Iterator<E> iterator();
}
```

- If the declaration of our class is something like:
- `public class ArrayList<E> implements List<E>, Iterable<E>`
- we promise to have a method `iterator()` that returns an `Iterator<E>` (see step 3 in previous slide)

https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html

Iterator<E> Interface

- Interface that allows us to iterate over a collection (i.e. a data structure) one element at a time.

```
public interface Iterator<E> {
  //returns true if the iterator has more elements
  //that is if next() would return an element instead of throwing an
exception
  boolean hasNext();

  //returns the next element in the iteration
  //post: advances the iterator to the next value
  E next();

}
```

You can also implement this in a different class, it doesn’t have to be your “main” class for the data structure.

https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html

Taking a closer look at `ArrayListIterator`

```
public class ArrayList<E> implements List<E>, Iterable<E> {
    //…
  public Iterator<E> iterator() {
    return new ArrayListIterator();
    }
```

A new ArrayListIterator() is created each time we make a new for loop (so i is reset to 0)

```
  private class ArrayListIterator implements Iterator<E> {
    private int i = 0;
```

i is an instance variable of this new class

```
    public boolean hasNext() {
      return i < size;
    }
    public E next() {
      return data[i++];
    }
  }
```

we increment i every time we call .next()

Worksheet time!

Write an OddIterator class that retrieves only the *odd* values in an ArrayList.

If the ArrayList is [7, 4, 1, 3, 0], the following code should print 7, 1, 3:

```
public static void main(String[] args) {
    ArrayList<Integer> myList = new ArrayList<Integer>(Arrays.asList(7, 4, 1, 3, 0));
    OddIterator oi = new OddIterator(myList);
    while(oi.hasNext()){
        System.out.println(oi.next());
    }
}
```

Worksheet answers

```
public class OddIterator implements Iterator<Integer> {

    // The array whose odd values are to be enumerated
    private ArrayList<Integer> myArrayList;

    //any other instance variables you might need
    int counter;

    //An iterator over the odd values of myArrayList
    public OddIterator(ArrayList<Integer> myArrayList){
        this.myArrayList = myArrayList;
        counter = 0;
    }

    //runs in O(n) time
    public boolean hasNext(){
        for (int i=counter; i<myArrayList.size(); i++){
            if(myArrayList.get(i)%2 == 1){
                counter = i;
                return true;
            }
        }
        return false;
    }

    //runs in O(1) time
    public Integer next(){
        return myArrayList.get(counter++);
    }
}
```

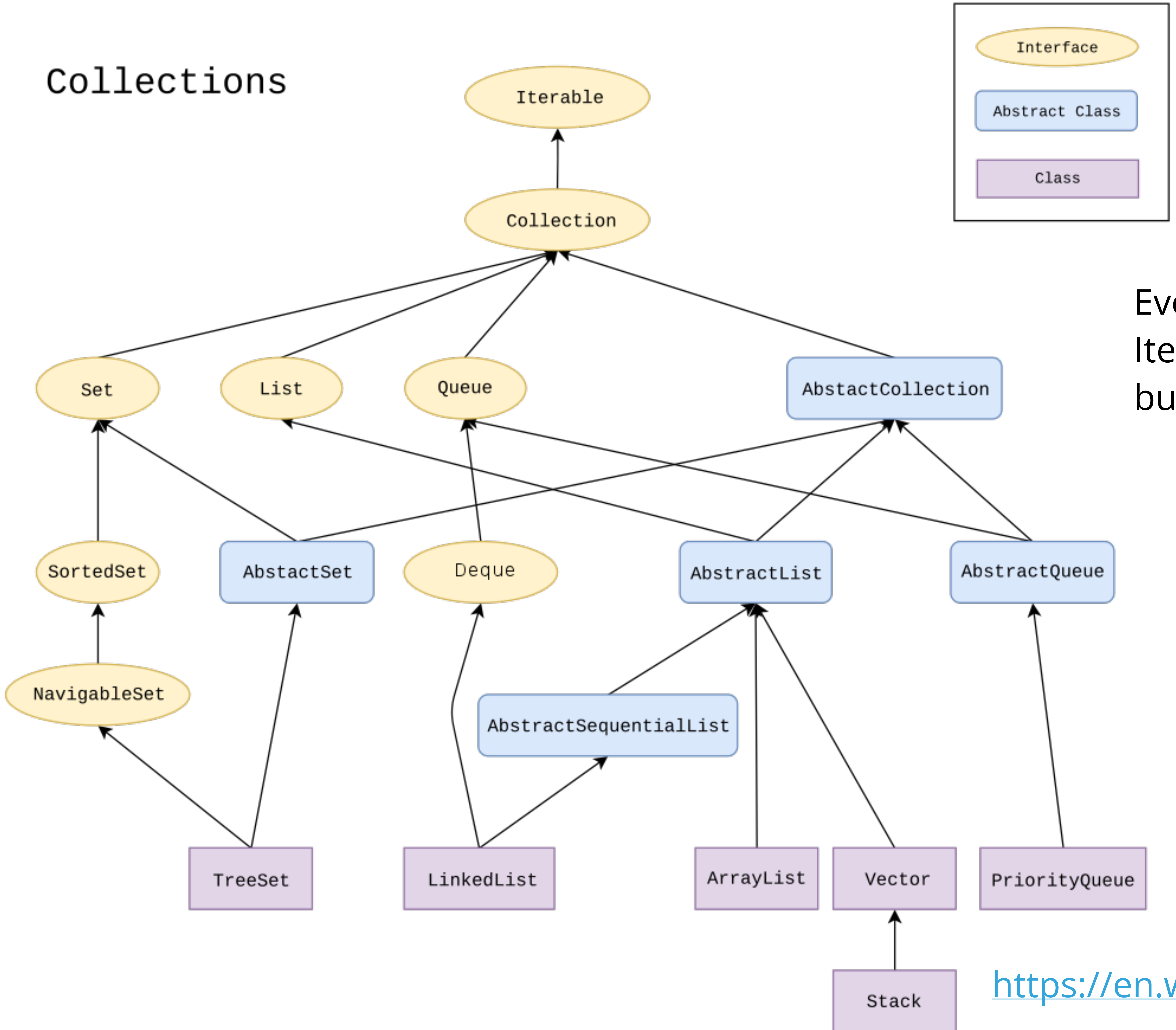
Constructor

Manually iterate through the ArrayList, true if there’s an odd element left

get the element at index “counter”, increment counter

JCF

The Java Collections Framework



Everything in Collection implements Iterable, so you can iterate through with every built-in class in the JCF.

https://en.wikipedia.org/wiki/Java_collections_framework

Comparable & Comparator

Back to sorting...

- Definition of a Key: assuming that an element consists of multiple components, the key is the property based on which we sort elements.
  - Examples: elements could be books and potential keys are the title or the author which can be sorted alphabetically, or the ISBN which can be sorted numerically.
  - Naturally lends itself to OOP where objects have different instance variables that can serve as different keys.
- Let's say we want to sort an array of objects of type E.
- Our class E should implement the `Comparable<E>` interface and we will need to implement the `compareTo(E that)` method.
  - Alternatively, it can also implement the `Comparator<E>` interface and we will need to implement the `compare(E that)` method.

`Comparable<E>`

- Interface with a single method that we need to implement: `public int compareTo(T that)`
- Implement it so that `v.compareTo(w)`:
  - Returns >0 if v is greater than w.
  - Returns <0 if v is smaller than w.
  - Returns 0 if v is equal to w.
- Corresponds to natural ordering.
- Java classes such as `Integer`, `Double`, `String`, `File` all implement `Comparable`.

Example - Employee

```
public class Employee implements Comparable<Employee> {

    private int id;
    private String name;
    private int salary;

    public Employee(int id, String name, int salary) {
        this.id = id;
        this.name = name;
        this.salary = salary;
    }

    public int compareTo(Employee e) {
        if (this.id < e.id) {
            return -1;
        } else if (this.id > e.id) {
            return 1;
        } else
            return 0;
    }
```

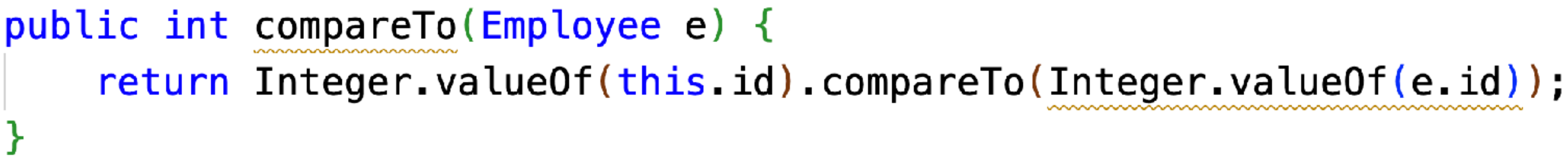
There are 3 instance variables we can sort by here
Let's just start with id for now

If this employee's ID # is smaller than that employee's, return a negative number

If this employee's ID # is bigger than that employee's, return a positive number

Otherwise, they're equal, so return 0

Example - Employee



```
public int compareTo(Employee e) {
    return Integer.valueOf(this.id).compareTo(Integer.valueOf(e.id));
}
```

This method also works - use the built in .compareTo of Integers

Note: Integer is an object, int is a primitive type. Integer.valueOf(int) *unwraps* the primitive int and converts its type to Integer so we can call the .compareTo method.

Integer (object) ≠ int (primitive)!!!

Comparator<E>

- Sometimes the natural ordering is not the type of ordering we want.
- Comparator is an interface which allows us to dictate that kind of ordering we want by implementing the method:
  `public int compare(T this, T that)`
- Implement it so that compare(v, w):
  - Returns >0 if v is greater than w.
  - Returns <0 if v is smaller than w.
  - Returns 0 if v is equal to w.

Basically, kind of the same thing as Comparable<E> and compareTo, but for external controllable ordering

https://stackoverflow.com/questions/2266827/when-to-use-comparable-and-comparator

Example – Employee

```
public class Employee implements Comparable<Employee> {

    private int id;
    private String name;
    private int salary;

    public Employee(int id, String name, int salary) {
        this.id = id;
        this.name = name;
        this.salary = salary;
    }
```

One last method for compareTo: call compare() in the Integer class

```
    public int compareTo(Employee e) {
        return Integer.compare(this.id, e.id);
    }
```

Two Comparator<E>s - different syntax, but both do comparisons

```
    public static Comparator<Employee> nameComparator = new Comparator<Employee>() {
        public int compare(Employee e1, Employee e2) {
            return e1.name.compareTo(e2.name);
        }
    };

    public static Comparator<Employee> salaryComparator() {
        return (Employee e1, Employee e2) -> Integer.compare(e1.salary, e2.salary);
    }
}
```

Example – Employee (syntax explanation)

```
public static Comparator<Employee> nameComparator = new Comparator<Employee>() {
    public int compare(Employee e1, Employee e2) {
        return e1.name.compareTo(e2.name);
    }
};
```

create an object called nameComparator which is of type Comparator<Employee>
nameComparator has access to the compare() method, which returns a call to the built-in .compareTo() method of Strings (e1.name, e2.name)

```
public static Comparator<Employee> salaryComparator() {
    return (Employee e1, Employee e2) -> Integer.compare(e1.salary, e2.salary);
}
```

This is the more "modern" shorthand notation. The -> arrow is a lambda expression, shorthand for

```
public int compare(Employee e1, Employee e2) {
     return Integer.compare(e1.salary, e2.salary);
}
```

Employee e1, Employee e2 are the inputs. The method returns Integer.compare(e1.salary, e2.salary). The -> shorthand is an *anonymous function*: it doesn't need a name, since the Comparator<E> interface only implements one method (compare) by default, and the signatures match.

Note: nameComparator is an object, but salaryComparator() is a method which returns an object! (Changes how you call them)

Sorting with `Collections` with `Comparable`

- As long as our class implements a Comparable interface, we can sort them with the sort() method in the Collections class:
- `Collections.sort(list)`
  - e.g., `Collections.sort(employees)` where employees is an ArrayList of Employee objects
  - If the elements in list do not implement the `Comparable`, throws a `ClassCastException`.

Sorting with `Collections` with `Comparator`

- If we instead choose to use a Comparator interface, we can use
- `Collections.sort(list, someComparator)`
  - e.g., `Collections.sort(employees, Employees.nameComparator)` where employees is an ArrayList of Employee objects
  - If the elements in list can’t be compared with `Comparator`, or do not implement the `Comparable`, throws a `ClassCastException.`

Full Employee Class

```java
public class Employee implements Comparable<Employee> {

    private int id;
    private String name;
    private int salary;

    public Employee(int id, String name, int salary) {
        this.id = id;
        this.name = name;
        this.salary = salary;
    }

    public int compareTo(Employee e) {
        if (this.id < e.id) {
            return -1;
        } else if (this.id > e.id) {
            return 1;
        } else
            return 0;
        // return Integer.valueOf(this.id).compareTo(Integer.valueOf(e.id));
        // return Integer.compare(this.id, e.id);

    }

    public static Comparator<Employee> nameComparator = new Comparator<Employee>() {
        public int compare(Employee e1, Employee e2) {
            return e1.name.compareTo(e2.name);
        }
    };

    public static Comparator<Employee> salaryComparator() {
        return (Employee e1, Employee e2) -> Integer.compare(e1.salary, e2.salary);
    }

    public String toString() {
        return "Name: " + name + " ID: " + id + " Salary: " + salary;
    }
```

Worksheet time!

What does main() print?

```
public static void main(String[] args) {

    Employee e1 = new Employee(5, "Yash", 100000);
    Employee e2 = new Employee(8, "Tharun", 25000);
    Employee e3 = new Employee(4, "Yush", 10000);
    List<Employee> list = new ArrayList<Employee>();
    list.add(e1);
    list.add(e2);
    list.add(e3);

    System.out.println(list);

    Collections.sort(list);
    System.out.println(list);

    Collections.sort(list, Employee.nameComparator);
    System.out.println(list);

    Collections.sort(list, Employee.salaryComparator());
    System.out.println(list);

}
```

Bonus Q: Why is it Employee.nameComparator, but Employee.salaryComparator() (with parentheses?)

Worksheet answers

```
public static void main(String[] args) {

    Employee e1 = new Employee(5, "Yash", 100000);
    Employee e2 = new Employee(8, "Tharun", 25000);
    Employee e3 = new Employee(4, "Yush", 10000);
    List<Employee> list = new ArrayList<Employee>();
    list.add(e1);
    list.add(e2);
    list.add(e3);
```

Unsorted list (order they were added)

```
    System.out.println(list);
    //[Name: Yash ID: 5 Salary: 100000, Name: Tharun ID: 8 Salary: 25000, Name: Yush ID: 4 Salary: 10000]
```

Sorted by ID number (Yush, Yash, Tharun)

```
    Collections.sort(list);
    System.out.println(list);
    //[Name: Yush ID: 4 Salary: 10000, Name: Yash ID: 5 Salary: 100000, Name: Tharun ID: 8 Salary: 25000]
```

Sorted by alphabetical name (Tharun, Yash, Yush)

```
    Collections.sort(list, Employee.nameComparator);
    System.out.println(list);
    //[Name: Tharun ID: 8 Salary: 25000, Name: Yash ID: 5 Salary: 100000, Name: Yush ID: 4 Salary: 10000]
```

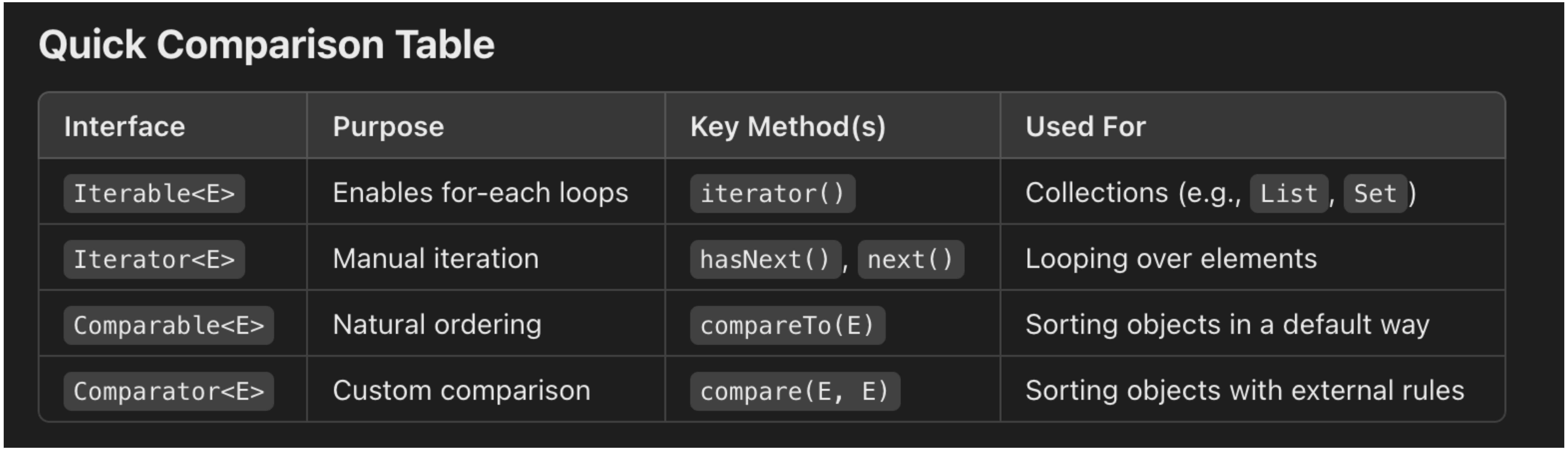
Sorted by lowest->highest salary (Yush, Tharun, Yash)

```
    Collections.sort(list, Employee.salaryComparator());
    System.out.println(list);
    //[Name: Yush ID: 4 Salary: 10000, Name: Tharun ID: 8 Salary: 25000, Name: Yash ID: 5 Salary: 100000]

}
```

Summary

- **Iterable<E> vs Iterator<E>** - Iterable<E> is automatically called in a for each loop. Iterator<E> is a class that specifies hasNext() and next() methods. The iterator() method of an Iterable<E> must return an object of a class that implements Iterator<E>.
- **Comparable<E> vs Comparator<E>** - Comparable<E> defines the “natural ordering” of how comparisons should go. Just like how Iterator<E> defined the control for looping, Comparator<E> defines the custom control for comparisons.



Quick Comparison Table

Interface	Purpose	Key Method(s)	Used For
`Iterable<E>`	Enables for-each loops	`iterator()`	Collections (e.g., `List`, `Set`)
`Iterator<E>`	Manual iteration	`hasNext()`, `next()`	Looping over elements
`Comparable<E>`	Natural ordering	`compareTo(E)`	Sorting objects in a default way
`Comparator<E>`	Custom comparison	`compare(E, E)`	Sorting objects with external rules

(Credit to ChatGPT for this table)

Lecture 11 wrap-up

- Announcements: Compression part 1 HW released. More in lab tomorrow, but extension to **Thu 11:59pm** due to checkpoint/needing the JUnit lab next week. (Still, good to get started early: it's conceptually hard!)
- HW4: Calculator due 11:59pm tonight
- Lab tomorrow will be peer learning groups reviewing the practice problems + quiz (solutions are updated on the PDF)

Resources

- Comparable: https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html
- Comparator: https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html
- Exercise for the reader: what if we wanted to make the OddIterator in the first worksheet Q work for all ArrayLists, such that the for-each loop would only get odd elements? What edits would we need to make?