interface examples:
Comparable, Iterable & Iterator

Feb. 11, 2022
Recall: Java interface

interface is a reserved word in the Java language.

A Java interface is like a class, but the methods have no bodies.

e.g. List\<T\>

ArrayList\<T\> and LinkedList\<T\> implement List\<T\>. 
Java Comparable interface

Suppose you want to define an *ordering* on objects of some class.

Sorted lists and other data structures we’ll see later (binary search trees, priority queues) all *require* that an ordering exists.

You cannot use the “<“ operator to compare objects.
Comparable interface

interface Comparable<T> {
    int compareTo(T t);
}

It is part of the java.lang package (see API)

It has a generic type, like List<T>.
e.g. String implements Comparable<T>

https://docs.oracle.com/javase/7/docs/api/java/lang/String.html

The natural ordering on strings is called the lexicographic ordering (like in a dictionary).
**Comparable recommendation**

Suppose `class T implements Comparable<T>`

```java
T e1, e2;
```

Java API recommends `e1.compareTo(e2)` returns:

- negative number, if `e1 < e2`
- 0, if `e1.equals(e2)` is true
- positive number, if `e1 > e2`

Note “>” and “<” here do not refer to a Java operation, but rather to our model (in our heads) of the ordering.
Example: Circle

Q: How can we define a `compareTo(Circle)` and `equals(…)` method for ordering Circle objects?

A: Compare their radii.
public class Circle extends Shape implements Comparable<Circle> {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public int compareTo(Circle c) {
        if (this.radius > c.radius)
            return 1;
        else if (this.radius == c.radius)
            return 0;
        else
            return -1;
    }

    public boolean equals(Object obj) {
        return (obj instanceof Circle) &&
            this.radius == ((Circle) obj).radius;
    }
}

Example: Rectangle

Q: When are two Rectangle objects equal?
A: Their heights are equal and their widths are equal.

However, there is no unique and natural way to define a `compareTo()` method for ordering Rectangle objects. (e.g. compare areas? or perimeters? or heights? etc).

e.g. the left one has a larger width but smaller height
Example: Ork

Suppose we have created a new data type Ork. How should we compare elements of this type?

Based on their weapon? height? name?
Ork.compareTo() -- based on height only?

```java
public class Ork implements Comparable<Ork> {
    private Weapon w;
    private Integer height;
    private String name;

    public int compareTo(Ork o) {
        if (this.height > o.height) {
            return 1;
        } else if (this.height == o.height) {
            return 0;
        } else {
            return -1;
        }
    }
}
```

But we let’s say we want to consider two Orcs to be “equal” only if they have the same weapon, height, and name. Then, the above `compareTo()` method would violate the Java API recommendation that `e1.compareTo(e2)` is 0 if and only if `e1.equals(e2)` is true.
public class Ork implements Comparable<Ork> {
    private Weapon w; // implements Comparable
    private Integer height;
    private String name;

    public int compareTo(Ork o) {
        int result = this.w.compareTo( o.w );
        if (result==0) {
            result = this.height.compareTo( o.height );
        }
        if (result == 0) {
            result = this.name.compareTo( o.name );
        }
        return result;
    }
}

Note this definition uses overloaded methods for compareTo(), namely for classes Weapon, Integer, String.
How is Comparable used?

```java
interface Comparable<T> {
    int compareTo(T e);
}
```

This interface will be used later when we wish to sort and/or search a collection of (comparable) elements.

ASIDE: the Java Collections class is used for this.
COMP 250
Lecture 16

interface examples:
Comparable, Iterable & Iterator

Feb. 11, 2022
double dArray = {1.0,  7.5,  -2.67,  5.999};

for (double   d   :   dArray)   {
    System.out.println( d );
}

LinkedList<String>   list =  new LinkedList<String>();
....

for (String   s   :   list)   {
    System.out.println( s );
}
More generally....

We often want to visit (or “iterate through”) all the objects in some collection of objects.

• arraylist
• linked list
• hash map entries  (later in course)
• binary search tree  (later in course)
• vertices in a graph  (later in course)
• .....
Iterator and Iterable

• The enhanced for loop (*for-each*) makes use of two interfaces: *Iterator* and *Iterable*.

• We can implement these interfaces for our own classes, and iterate through a collection using the enhanced for loop, or in other ways.
Iterator interface

interface Iterator<T> {
    boolean hasNext();
    T next(); // returns current element
             // and advances to the next
    void remove(); // optional; ignore it
}

next() is a method, rather than a field.
Example: Singly Linked Lists

```java
class SLinkedList<E> implements Iterable<E>{{
    SNode<E> head;
    :
    class SNode<E> {
        SNode<E> next;
        E element;
        // etc
    };
    class SLL_Iterator<E> implements Iterator<E>{
        // implements the hasNext() and next() methods
    }
}
```
An `SLL_Iterator` object will reference a node in the singly linked list.

Q: How many objects do we have here?

A: $1 + 1 + 4 + 4 = 10$
The `SLL_Iterator` object will reference a node in the singly linked list.

The `SLL_Iterator` object will iterate through the nodes and return the element referenced by each node.
class SLL_Iterator<E> implements Iterator<E> {

    SNode<E> cur;

    SLL_Iterator( SLinkedList<E> list) {
        cur = list.getHead();
    }

    public boolean hasNext() {
        return (cur != null);
    }

    public E next() {
        E element = cur.getElement;
        cur = cur.getNext();
        return element;
    }
}

Don’t confuse use of “next()” and “hasNext()” above with how “next” is used in linked lists.
Q: Who constructs an SLL_Iterator object?

A: A SLinkedList object does this, similar to how it constructs a new SNode when it adds a new element to the list. See next slide.

(Recall from several slides ago that SLL_Iterator is an inner class of SLinkedList, just like SNode is an inner class.)
We say “a class is iterable” if it can make/construct an `Iterator` object that can iterate over its elements.

So, if a class implements `Iterable`, then this class has an `iterator()` method, which constructs an `Iterator` object.

ASIDE: I think `iterator()` should have been called `makeIterator()`, since it is easy to confuse the method `iterator()` with the interface `Iterator`.
Let’s add the `iterator()` method to the singly linked list class.

class SLinkedList<E> implements Iterable<E> {

    SNode<E> head;

    class SNode<E> {
        SNode<E> next;
        E element;
        // etc
    }

    class SLL_Iterator<E> implements Iterator<E> {
        // implements the hasNext() and next() methods
        // (see earlier slides)
    }

    SLL_Iterator<E> iterator() {
        return new SLL_Iterator(this);
    };
}
Example 1

Suppose a method in some class has this code:

```java
SLinkedList<Shape> list;
Shape s;

// make a list
```
SLinkedList<Shape> list;
Shape s;
// make a list

Iterator<Shape> iter = list.iterator();
SLinkedList<Shape> list;
Shape s;
// make a list

Iterator<Shape> iter = list.iterator();

s = iter.next()

Note that s references the first element!

The iterators iterate over LinkedList nodes, not Shapes. The next() method returns Shapes.
SLinkedList<Shape> list;
Shape s;
// make a list

Iterator<Shape> iter = list.iterator();

s = iter.next()
s = iter.next()

Note that s references the second element!

The iterators iterate over LinkedList nodes, not Shapes. The next() method returns Shapes.
SLinkedList<Shape> list;
Shape s;
// make a list

Iterator<Shape> iter = list.iterator();

s = iter.next()
s = iter.next()
s = iter.next()
Example 2

What if we want to have multiple “iterators”?

Analogy: Multiple TA’s grading a collection of exams.
SLinkedList<Shape> list;
Shape s;

Iterator<Shape> iter1 = list.iterator();
Iterator<Shape> iter2 = list.iterator();
SLinkedList<Shape> list;
Shape s;

Iterator<Shape> iter1 = list.iterator();
Iterator<Shape> iter2 = list.iterator();

s = iter1.next()

The iterators iterate over SNodes, not Shapes. The next() method returns a reference to a Shape.
SLinkedList<Shape> list;
Shape s;

Iterator<Shape> iter1 = list.iterator();
Iterator<Shape> iter2 = list.iterator();

s = iter1.next();

s = iter2.next();

s = iter2.next();

The iterators iterate over SNodes, not Shapes.
The next() method returns a reference to a Shape.
A Big Picture

interface Iterator<E>

next() : E
hasNext() : boolean

class SLLlIterator<E>

next() : SNode
hasNext() : boolean

class SLinkedList<E>

interface Iterable<E>

iterator()

interface Iterator<E>

next() : E
hasNext() : boolean

class SLLlIterator<E>

interface Collection<E>

extends java.util

interface List<E>

extends java.util

class LinkedList<E>

class ArrayList<E>

extends java.util
Coming up...

Lectures

Mon. Feb. 14  Stacks
Wed. Feb. 16  Queues
Fri. Feb. 18  Mathematical Induction

(The following week we start recursion.)

Assessments

Assignment 1
  - due today

Quiz 2  closes at 8 pm (finish by then)

Assignment 2 will be posted today