COMP 250

Lecture 32

interfaces
(Comparable, Iterable & Iterator)

Nov. 22/23, 2017
Java Comparable interface

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

Sorted lists, binary search trees, priority queues all require that an ordering exists. (Elements are “comparable”).
Comparable interface

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

Comparable interface

T implements Comparable<T>

T t1, t2;
Comparable interface

T implements Comparable<T>

T t1, t2;

Java API recommends that t1.compareTo(t2) returns:

0, if t1.equals(t2) returns true
positive number, if t1 > t2
negative number, if t1 < t2
Some classes assume comparable generic types. Their implementations call the `compareTo()` method.

e.g. `PriorityQueue<E>`

   (uses a heap with comparable E)

`TreeSet<E>`

   (uses a balanced binary search tree with comparable E)

`TreeMap<K, V>`

   (uses a balanced binary search tree with comparable K)
e.g. String implements Comparable<T>

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

```java
public int compareTo(String anotherString)

Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings. The character sequence represented by this String object is compared lexicographically to the character sequence represented by the argument string. The result is a negative integer if this string object lexicographically precedes the argument string. The result is a positive integer if this string object lexicographically follows the argument string. The result is zero if the strings are equal; compareTo returns 0 exactly when the equals(Object) method would return true.

This is the definition of lexicographic ordering. If two strings are different, then either they have different characters at some index that is a valid index for both strings, or their lengths are different, or both. If they have different characters at one or more index positions, let k be the smallest such index; then the string whose character at position k has the smaller value, as determined by using the < operator, lexicographically precedes the other string. In this case, compareTo returns the difference of the two character values at position k in the two string -- that is, the value:

    this.charAt(k) - anotherString.charAt(k)

If there is no index position at which they differ, then the shorter string lexicographically precedes the longer string. In this case, compareTo returns the difference of the lengths of the strings -- that is, the value:

    this.length() - anotherString.length()
```
e.g. Character, Integer, Float, Double, BigInteger, etc all implement `Comparable<T>`.

You cannot compare objects of these classes using the “<” operator. Instead use `compareTo()`.
Example: Circle

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

A: Compare their radii or their areas.

Java recommends overriding `equals(Object)`, rather than overloading.
public class Circle implements Comparable<Circle> {
    private radius;

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

    public boolean equals(Circle c) {
        return radius == c.getRadius();
    }

    public int compareTo(Circle c) {
        return radius - c.getRadius();
    }
}
Example: Rectangle

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

These are not equal:

Q: How can we define a `compareTo()` and `equals(...)` method for ordering Rectangle objects?
class Rectangle implements Comparable<Rectangle>

Rectangle t1, t2;

Java API *recommends* that \( t1\.compareTo( t2 ) \) returns:

\[
\begin{align*}
0, & \quad \text{if } t1\.equals( t2 ) \text{ returns true} \\
\text{positive number}, & \quad \text{if } t1 > t2 \\
\text{negative number}, & \quad \text{if } t1 < t2
\end{align*}
\]
class Rectangle implements Comparable<Rectangle>{

    // constructor
    ....

    // getArea method

    boolean equals( Rectangle other ) {
        return (this.height == other.height) && (this.width == other.width);
    }

    int compareTo( Rectangle r ){
        return this.getArea() - other.getArea();
    }
}

This is not consistent with Java API recommendation on the previous slide. Why not?
class Rectangle implements Comparable<Rectangle>{

    ...  // constructor

    ....  // getArea method

    boolean equals( Rectangle other ) {
        return this.getArea() == other.getArea();
    }

    int compareTo( Rectangle r ){
        return this.getArea() - other.getArea();
    }
}

This is consistent with Java API recommendation.
But it is maybe not such a natural way to order rectangles.
COMP 250

Lecture 32

interfaces
(Comparable, Iterable & Iterator)

Nov. 22/23, 2017
Java Iterator interface

Motivation 1: we often want to visit all the objects in some collection.

e.g. linked list, binary search tree, hash map entries, vertices in a graph
Java Iterator interface

Motivation 2: We sometimes want to have multiple “iterators”.

Analogy: Multiple TA’s grading a collection of exams.
Java Iterator interface

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

next() is a method, not a field like in the linked list class.
Recall lecture 5 and Exercises 3

class SLinkedList<E> {

    SNode<E> head;

    :

    private class SNode<E> {

        SNode<E> next;
        E element;
        ....
    }

    private class SLL_Iterator<E> implements Iterator<E> {
        ....
    }

}
As we will see, the iterator object will reference a node in the list.
private class SLL_Iterator<E> implements Iterator<E> {

    private SNode<E> cur;

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

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

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

Q: Who constructs the Iterator object for a collection class such as LinkedList, ArrayList, HashMap, ... ?

A:
Java Iterator interface

Q: Who constructs the Iterator object for a collection class such as LinkedList, ArrayList, HashMap, ... ?

A: The class itself does it.

How?

A collection class is “iterable” if the class is able to make an iterator object that iterates over the elements.
Java  Iterable  interface

interface  Iterable<T>  {
   Iterator<T>  iterator();
}

It could have been called makeIterator().

If a class implements Iterable, then the class has an iterator() method, which constructs an Iterator object.
class SLinkedList<E> implements Iterable<E> {

    SNode<E> head;

    private class SNode<E> {
        SNode<E> next;
        E element;
        ....
    }

    private class SLL_Iterator<E> implements Iterator<E> {
        ....
    }

    SLL_Iterator<E> iterator() {
        return new SLL_Iterator<>(this);
    }
}
private class SLL_Iterator<E> implements Iterator<E> {
    private 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;
    }
}
LinkedList<Shape> list;
Shape s;
LinkedList<Shape> list;
Shape s;

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

s = iter1.next()

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

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

s = iter1.next()
s = iter2.next()
s = iter1.next()
s = iter2.next()
s = iter2.next()

The iterators iterate over LinkedList nodes, not Shapes. The next() method returns Shapes.
SLLIterator might be an inner class of SLinkedList. The iterator() method calls the constructor of the SLLIterator class.
Assignment 4: MyHashTable

You will implement a hashtable (or hashmap).

You will use the SLinkedList class from Exercises 4 to implement a HashLinkedList class which you will use for the buckets.

You will implement a HashIterator class for your hash table.
ASIDE: Java enhanced for loop

It can be used for any class that implements Iterable.

Example:

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

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