Keeping items sorted
A better way

Stacks

Midterm 1

- I will post solutions over break
- Grades later today

No Lab Today

Hw5 is available to be worked on over break
I assume must of you will not but it will be due on the Wednesday after break.

SortedArrayList

```
public class Sal<C> {
```

- Problem
  - how to guarantee that the Generic class C has an ordering …
    - Homework 4 convert to string and compare those strings
    - That method is less than optimal …. Why?
- It would be better to require that items have an ordering
  - or at least that items know ordering with respect to each other.

- In Java — require the Comparable interface

Comparable Interface

- Part of Java language
- Idea, give a way for classes to define a total ordering of instances
- Java classes that implement:
  - String
  - All descendants of Number
  - Lots of others

The Comparable Interface

- `public interface` **`Comparable<T>`**
  This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred as the class's *natural ordering*, and the class's `compareTo` method is referred to as its *natural comparison method*.Lists (and arrays) of objects that implement this interface can be sorted automatically by `Collections.sort` (and `Arrays.sort`). Objects that implement this interface can be used as keys in a sorted map or as elements in a sorted set, without the need to specify a comparator.
  The natural ordering for a class `C` is said to be *consistent with equals* if and only if `e1.compareTo(e2) == 0` the same boolean value as `e1.equals(e2)` for every `e1` and `e2` of class `C`. Note that `null` is not an instance o any class, and `e.compareTo(null)` should throw a `NullPointerException` even though `e.equals(null)` returns `false`.
  It is strongly recommended (though not required) that natural orderings be consistent with equals. This is so because sorted sets (and sorted maps) without explicit comparators behave "strangely" when they are used with elements (or keys) whose natural ordering is inconsistent with equals. In particular, such a sorted set (or sorted map) violates the general contract for set (or map), which is defined in terms of the `equals` method.
  For example, if one adds two keys `a` and `b` such that `(!a.equals(b) && a.compareTo(b) == 0)` to a sor set that does not use an explicit comparator, the second `add` operation returns false (and the size of the sorted does not increase) because `a` and `b` are equivalent from the sorted set's perspective.
  Virtually all Java core classes that implement `Comparable` have natural orderings that are consistent with equ One exception is `java.math.BigDecimal`, whose natural ordering equates `BigDecimal` objects with equal values and different precisions (such as 4.0 and 4.00).
  For the mathematically inclined, the *relation* that defines the natural ordering on a given class C is:

  ```
  {(x, y) such that x.compareTo(y) <= 0}.
  ```

Comparable interface (shortened)

`int compareTo(T o)`
Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.
The implementor must ensure `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))` for all `x` and `y`. (This implies that `x.compareTo(y)` must throw an exception iff `y.compareTo(x)` throws an exception.)

The implementor must also ensure that the relation is transitive: `(x.compareTo(y)>0 && y.compareTo(z)>0)` implies `x.compareTo(z)>0`.

Finally, the implementor must ensure that `x.compareTo(y)==0` implies that `sgn(x.compareTo(z)) == sgn(y.compareTo(z))`, for all `z`.

It is strongly recommended, but *not* strictly required that `(x.compareTo(y)==0) == (x.equals(y))`. Generally speaking, any class that implements the `Comparable` interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

In the foregoing description, the notation `sgn(`*expression*`)` designates the mathematical *signum* function, which is defined to return one of `-1`, `0`, or `1` according to whether the value of *expression* is negative, zero or positive.

Parameters:
`o - the object to be compared.`
Returns:
`a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.`

Comparable Interface (even shorter)

```
public interface Comparable<T> {
   int compareTo(T o);
}
```

- return 0 if they are equal
- return <0 if caller is less than compared
- return >0 if caller greater than compared
  - Integer v3 = Integer.valueOf(3).
  - v3.compareTo(4) ==> -1
  - "D".compareTo("A") ==> 3

Comparable Example

```
public class MyInteger implements Comparable<MyInteger> {
    protected final int value;
    public MyInteger(int value) {
        this.value = value;
    }
    public int getValue() {
        return value;
    }
    public int compareTo(MyInteger other) {
        return value - other.getValue();
    }
    public boolean equals(Object ob) {
        if (ob instanceof MyInteger)
            return value == ((MyInteger) ob).value;
        if (ob instanceof Integer)
            return value == ((Integer) ob).intValue();
        return false;
    }
}
```

It look like this should not be allowed since value is protected. But you are within g the MyIntegerClass so can look at private and protected values of other instances. (Or just use getValue() method)

Comparable Practice

```
public class CompExRect implements Comparable<CompExRect> {
    protected int width, height;

    public CompExRect(int w, int h) {
       width = w;
       height = h;
    }

    public int area() { return width * height; }
```

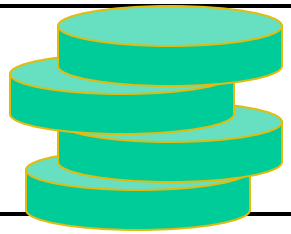
Finish this class so that it implements the Comparable interface, using area to judge relation. Also write toString and equals
Suppose wanted to change to using perimeter?

Testing CompExRect

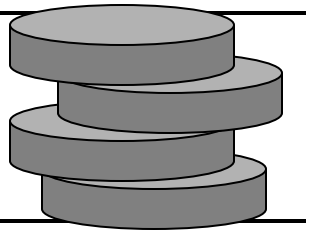
```
public static void main(String[] args) {
        CompExRect[] ce = { new CompExRect(3,4),  new CompExRect(5, 2), new
CompExRect(11, 1), new CompExRect(1, 7) };

        for (int i=0; i<ce.length-1; i++) {
            for (int j=i+1; j<ce.length; j++) {
                System.out.println(ce[i] + " " + ce[j] + " " +
ce[i].compareTo(ce[j]));
            }
        }

    }
```



Stacks



- Insertion and deletions are First In Last Out
  - FILO
  - or LIFO
- Physical stacks are everywhere!

- **REQUIREMENT**
  - every method O(1)
- What functions does a stack need?

`Stack` Interface

- How do you inform user of stack that it is empty `peek` and `pop`?
  - throw exception?
    - runtime or checked?
  - return null?
  - Something else?

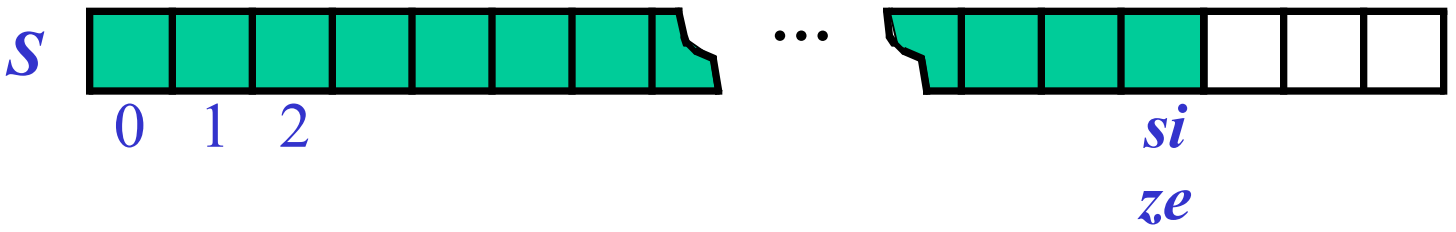
```
public interface StackInft<E> {
    public boolean empty();
    public E push(E e);
    public E peek();
    public E pop();
    public int size();
    public void clear();
}
```

Example

Method	Return Value	Stack Comtents
push(5)	5	{5}
push(3)	3	{5, 3}
size()	2	{5, 3}
pop()	3	{5}
empty()	FALSE	{5}
pop()	5	{}
empty()	TRUE	{}
pop()	null	{}
push(7)	7	{7}
push(9)	9	{7,9}
peek()	9	{7,9}

Array-based Stack

- Implement the stack with an array
- Add elements onto the end of the array
- Use an int `size` to keep track of the top



Performance and Limitations of Array Stack

- Performance
  - let n be the number of objects in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$
- Limitations
  - Max size is limited and can not be changed
  - Pushing onto a full stack will fail
    - need to handle that

Why not ArrayList?

- Every operation in Array stack is O(1)
- NOT true for ArrayList
  - Consider grow
    - unlike Hashtables, no wink and smile at ignored issues

- So if want O(1) guarantee for Stack cannot use ArrayList.
- For now, bound to array which means
  - fixed size
  - wasted space

Push

- Array has set size and may become full
- A `push` will fail if the array becomes full
  - Limitation of the array-based implementation
  - Alternatives?
    - Make the array grow (use ArrayList)?
      - why not?
  - What do to on fail?
    - return null
    - throw exception

Implementing an Array-based stack

```
public class ArrayStack<K> implements StackIntf<K> {
    private static final int DEFAULT_CAPACITY = 40;
    private int size;
    private K[] underlyingArray;

    public ArrayStack() {
        this(DEFAULT_CAPACITY);
    }

    public ArrayStack(int capacity) {
        size=0;
        underlyingArray = (K[]) new Object[capacity];
    }
}
```

Implementing SAL

- Suppose you want to have an underlying ArrayList
  - Why not just an array?

- Question
  - a new class that has an ArrayList as a private element (like we just did with String)?
  - a class that extends ArrayList?
    - How do you choose?

SAL — Extending ArrayList

- Methods to add to ArrayList
  - the SortedListInterface
  - Others?
- Methods to remove from ArrayList
  - How do you remove???

```
boolean add(E e)
void add(int index, E element)
boolean addAll(Collection<? extends E> c)
boolean addAll(int index, Collection<? exte
void clear()
E get(int index)
int indexOf(Object o)
boolean isEmpty()
E remove(int index)
boolean remove(Object o)
boolean removeAll(Collection<?> c)
E set(int index, E element)
int size()
<T> T[] toArray(T[] a)
```

SAL — new class

- Advantages
- Disadvantages

- Must implement interface — else?

SortedArrayList

- extends ArrayList!!!
  - should be able to hold almost anything
    - Generic!!
- overrides
  - add(E) — certainly
  - add(index, E)?
    - Implementation may do nothing!
      - or just so a sorted add, ignoring index
  - set(index, element)
    - like add(index, element)
  - remove(int) ??

Code for SortedArrayList

```
public class SALextending<B extends Comparable<B>> extends ArrayList<B>
implements SortedListInterface<B> {
    public boolean add(B obToAdd) {
        int loc = findPlace(obToAdd);
        insertAtLoc(obToAdd, loc);
        return true;
    }
   private int findPlace(B toAdd) {
        int place=0;
        while (place<size()) {
            if (toAdd.compareTo(get(place))<0) {
                break;
            } else {
            place++;
}
        }
        return place;
```

More SortedArrayList

```
private void insertAtLoc(String toAdd, int atLoc) {
    if (size()==0) {
        // use the original add function from ArrayList
        super.add(toAdd);
        return;
    }
    // Use the original Add and set function from arraylist
    super.add((String)get(size()-1));
    for (int ll=size()-2; ll>=atLoc; ll--) {
        super.set(ll+1, get(ll));
    }
    super.set(atLoc, toAdd);
}
```

To keep in sorted order

- Figure out where something should be put
  - O(n)
- put it there
  - O(n)
- Overall Complexity for 1 add
  - O(n) + O(n) = O(n)
- Complexity for N add
  - O(n) * O(n) = O(n^2)