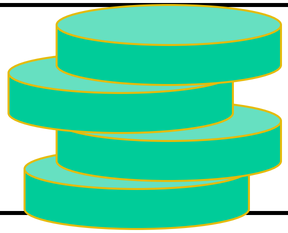
Stacks
Keeping an ArrayList sorted (part 2)

Command Lines

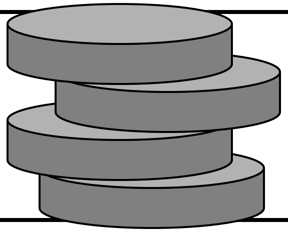
- Lots of assignments call for the ability to take something as a “command line parameter”
  - e.g., a file name
- VSC is not command line friendly
- So code at right will use command line when it is provided and will use hard coded defaults when not.

```
public class CommandLine
{
    public static void main(String[] args)
    {
        String[] defaultArgs = {“fileName"};
        System.out.println("Hello");
        if (args.length==0) {
            args = defaultArgs;
        }
        for (int i=0; i<args.length; i++)
        {
            System.out.println(i + " " + args[i]);
        }
        System.out.println("Goodbye");
    }

}
```



Stacks



- Insertion and deletions are First In Last Out
  - FILO
  - or LIFO
- Physical stacks are everywhere!
- Function names (in the following slides) follow java.util.Stack rather than Goodrich.

`Stack` Interface

- How do you inform user of stack that it is empty `peek` and `pop?`
  - throw exception?
  - return null?
  - Something else?
- **REQUIREMENT**
  - every method O(1)

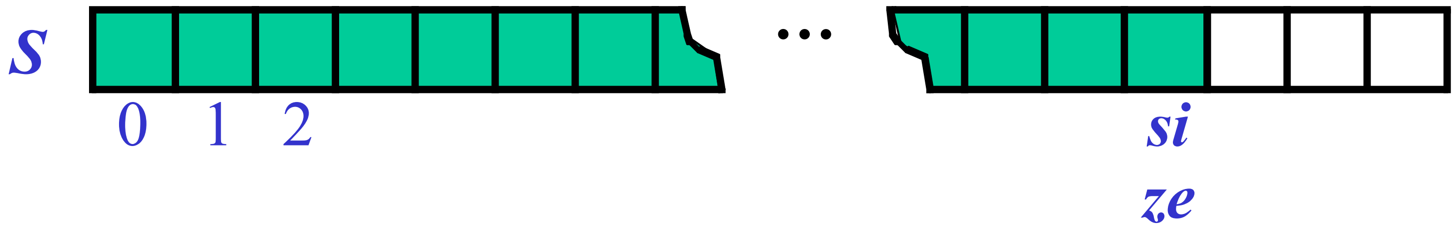
```
public interface StackInft<E> {
    public boolean empty();
    public E push(E e);
    public E peek();
    public E pop();
    public int size();
}
```

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];
    }
}
```

S_{orted}A_{rray}L_{ist}

- extends ArrayList
- Which of these need to be overridden?
  - add(E) — discussed
  - add(index, E)?
    - depends on implementation
  - remove(int) ??
  - remove(Obj) ??
    - lab for today!

```
boolean add(E e)
void add(int index, E element)
void clear()
Object clone()
boolean contains(Object o)
E get(int index)
int indexOf(Object o)
boolean isEmpty()
int lastIndexOf(Object o)
E remove(int index)
boolean remove(Object o)
E set(int index, E element)
int size()
```

Code for SortedArrayList

```
public void add(String stringToAdd) {
    int loc = findPlace(stringToAdd);
    insertAtLoc(stringToAdd, loc);
}

private int findPlace(String toAdd) {
    int place=0;
    while (place<size()) {
        if (toAdd.compareTo((String)get(place))<0) {
            break;
        } else {
        place++;
}
    }
    return place;
}
```

More SortedArrayList

```java
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 function from arraylist
    super.add((String)get(size()-1));
    for (int ll=size()-2; ll>=atLoc; ll--) {
        set(ll+1, get(ll));
    }
    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)

SortedArrayList further analysis

- Biggest issue is limitation to String!
- Otherwise some nice features
  - sorting would make the finding common words task from HW3 a lot easier.

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

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 wit elements (or keys) whose natural ordering is inconsistent with equals. In particular, such a sorted set (or sorte 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 s 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 {
int compareTo(T o);
}
```

- return 0 if they are equal
- return <0 if caller is less than compared
- return >0 if caller > compared
- new Integer(3).compareTo(4) ==> -1

Comparable Example

```
private class InsenceString implements Comparable {
        private String theString;
        public InsenceString(String strng) {theString = strng; }
        public String getString() { return theString; }
        public String toString() { return theString; }

        public int compareTo(Object o) {
            if (!(o instanceof InsenceString)) return -1;
            String s1 = theString.toLowerCase();
            String s2 = ((InsenceString) o).getString().toLowerCase();
            int l = s1.length();
            if (s2.length() < l) l = s2.length();
            for (int i = 0; i < l; i++) {
                int d = s1.charAt(i) - s2.charAt(i);
                if (d != 0) return d;
            }
            if (s1.length() == s2.length()) return 0;
            if (s1.length() > s2.length()) return 1;
            return -1;
        }    }
```

Comparable Example (pt 2)

```
    public static void main(String[] args) {
         String[] a = { "A", "B", "B", "BBB"};
         String[] b = { "B", "b", "BB", "BB" };
         new CompEx().test(a, b);
     }

     public void test(String[] ss1, String[] ss2) {
         for (int i = 0; i < ss1.length; i++) {
             InsenceString is1 = new InsenceString(ss1[i]);
             InsenceString is2 = new InsenceString(ss2[i]);
             System.out.println(is1 + " " + is2 + " " +
is1.compareTo(is2));
         }
     }
```

Re-implement SAL to use comparable

- Horrific syntax, but mostly just replace String with generic class C
- We will use the Comparable interface frequently for the rest of the semester

```
public class SalComp<C extends Comparable<C>>
      extends ArrayList<C> {

    public boolean add(C toAdd) { … }

    private int findPlace(C toAdd) { … }

    private void insertAtLoc(C toAdd,
                    int atLoc) { … }
```

In class

- Define a class MyIntPair that has two integer instance variables (i1 and i2)
- the class implements Comparable such that
  - total ordering is based on the difference between i1 and i2
    - if (i1-i2) in class class is greater than (i1-i2) in compared class then return +1
    - etc