SortingMachine
SortingMachine

• The SortingMachine component family allows you to add elements of type \( T \) to a collection of such elements, and then to remove them one at a time in sorted order according to a client-supplied ordering.
  - Queue and Stack support removal in FIFO and LIFO order, respectively.
The `SortingMachine` component family allows you to add elements of type `T` to a collection of such elements, and then to remove them one at a time in sorted order according to a client-supplied ordering.

- Queue and Stack support removal in FIFO and LIFO order, respectively.

FIFO and LIFO are time-based orderings; a `SortingMachine` uses a value-based ordering.
Why Not Use The `sort` Method?

```java
while (/*input values remain*/) {
    /* let x = next input value */
    q.enqueue(x);
}
q.sort(order);
while (q.length() > 0) {
    T x = q.dequeue();
    /* output x */
}
```
Why Not Use The \texttt{sort} Method?

```java
while (/*input values remain*/) {
    /* let x = next input value */
    q.enqueue(x);
}
q.sort(order);
while (q.length() > 0) {
    T x = q.dequeue();
    /* output x */
}
```

The Java libraries have similar methods to sort built-in arrays and other "collections".
Example

• Suppose you want to find the students with the 50 highest GPAs among all Ohio State students
  – Modify the previous code to show how you might do this ...
Performance in the Example

• Code using a sort method spends time to sort all 50,000 GPAs just to find the top 50
• Code based on SortingMachine can be much more efficient, because if you don’t remove all of the elements you don’t necessarily pay for sorting all of them
Interfaces and Classes

- Standard
- Iterable
- SortingMachine-Kernel
- SortingMachine
  - implements SortingMachine1L
  - implements SortingMachine5
Interfaces and Classes

- **Standard**
  - extends
  - SortingMachineKernel

- **Iterable**
  - extends
  - Standard

**SortingMachineKernel**

- has contracts for six methods:
  - add
  - changeToExtractionMode
  - removeFirst
  - isInInsertionMode
  - order
  - size

- extends
  - SortingMachine

- implements
  - SortingMachine5
Interfaces and Classes

There is really an abstract class as usual in the chain here, but it is not shown because these slides describe the client view, and a client needs only interface `SortingMachine` and a class like `SortingMachine1L`.

...
Mathematical Model

SORTING_MACHINE_MODEL is (  
  insertion_mode: boolean,  
  ordering: binary relation on T,  
  contents: finite multiset of T 
)

exemplar m

constraint
  IS_TOTAL_PREORDER(m.ordering)

type SortingMachineKernel is modeled by
  SORTING_MACHINE_MODEL
The mathematical model is an ordered triple (a.k.a. three-tuple): a boolean, a binary relation on $T$, and a finite multiset of $T$. 
Mathematical Model

\[
\text{SORTING\_MACHINE\_MODEL } \text{is} \text{ (}
\text{insertion\_mode: boolean,}
\text{ordering: bin\_relation \_on \ T,}
\text{contents: finite \ multiset \ of \ T}
\text{)}
\]

\text{exemplar } m
\text{constraint}
\text{IS\_TOTAL\_PR}

\text{type SortingMachineKernel}
\text{SORTING\_MACHINE\_MODEL}

Recall: a \textit{binary relation} on \( T \) may be viewed as a set of ordered pairs of \( T \), or as a \textit{boolean}-valued function \( R \) of two parameters of type \( T \) that is \textit{true} iff that pair is in the set.
Mathematical Model

SORTING_MACHINE_MODEL is (  
insertion_mode: boolean,  
ordering: binary relation on T,  
contents: finite multiset of T  
)

exemplar m  
constraint IS_TOTAL_PREORDER(m.ordering)  
type SortingMachine #  
SORTING_MACHINE_MODEL

A finite multiset is essentially a finite set with multiple copies of elements allowed, so there are effectively (non-negative) “counts” of all values of the element type T; details as necessary.
Mathematical Model

SORTING_MACHINE_MODEL is (  
  insertion_mode: boolean,  
  ordering: binary relation on T,  
  contents: finite multiset of T  
)

exemplar m

constraint

  IS_TOTAL_PREORDER(m.ordering)

type SortingMachineKernel is modeled by

SORTING_MACHINE_MODEL
Review: Comparators

• The Java interface Comparator\<T\> is:

```java
public interface Comparator\<T\> {
    /**
     * Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.
     */
    int compare(T o1, T o2);
}
```
Review: Comparators

• The notion of “less than” and “greater than” can be anything
• The notion of “equal to” is actually supposed to be “equivalent to”, in the sense that the first argument is neither “less than” nor “greater than” the other
• There are important technicalities …
private static class IntegerLT
    implements Comparator<Integer> {
@Override
    public int compare(Integer o1, Integer o2) {
        if (o1 < o2) {
            return -1;
        } else if (o1 > o2) {
            return 1;
        } else {
            return 0;
        }
    }
}
private static class IntegerLT implements Comparator<Integer> {
    @Override
    public int compare(Integer o1, Integer o2) {
        if (o1 < o2) {
            return -1;
        } else if (o1 > o2) {
            return 1;
        } else {
            return 0;
        }
    }
}

A class that implements Comparator is usually a nested class (i.e., declared for local use inside another class), and if so should be declared private static.
private static class IntegerLT implements Comparator<Integer> {
    @Override
    public int compare(Integer o1, Integer o2) {
        return o1 - o2;
    }
}

Note that the results are not specified to be $-1$ for “less than” and $+1$ for “greater than”, but merely negative and positive values, respectively! Does this code work?
private static class IntegerLT
    implements Comparator<Integer> {
    @Override
    public int compare(Integer o1, Integer o2) {
        return o1.compareTo(o2);
    }
}

Since a generic parameter must be a reference type, and each wrapper type T (here, Integer) implements the interface Comparable<T>, each has a compareTo method that can be called like this; it simplifies the code for compare from the previous Comparator<T> implementation (using <), if \( \leq \) is the mathematical ordering we want.
SortingMachine Constructor

• The constructor has one parameter `order` of type `Comparator<T>`
  – Sorting is based on the ordering provided by the `compare` method from `order`

• Requires:
  `IS_TOTAL_PREORDER(order)`

• Ensures:
  `this = (true, order, {})`
## Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Comparator&lt;Integer&gt; ci = new IntegerLT ();</code></td>
<td></td>
</tr>
<tr>
<td><code>SortingMachine&lt;Integer&gt; si = new SortingMachine1L&lt;&gt;(ci);</code></td>
<td></td>
</tr>
</tbody>
</table>

8 February 2019 OSU CSE
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparator&lt;Integer&gt; ci = new IntegerLT();</td>
<td>si = (true, ci, {})</td>
</tr>
<tr>
<td>SortingMachine&lt;Integer&gt; si = new SortingMachine1L&lt;&gt;(ci);</td>
<td></td>
</tr>
</tbody>
</table>
**add**

```java
void add(T x)
• Adds $x$ to the contents of `this`.
• Aliases: reference $x$
• Updates: `this.contents`
• Requires: `this.insertion_mode`
• Ensures:

```java
this.contents = #this.contents union {x}
```
add

`void` `add(T x)`

- Adds `x` to the contents.
-Aliases: reference `x`.
-Updates: `this.contents`.
-Requires: `this.insertion_mode`.
-Ensures:

  `this.contents = #this.contents union {x}`

For multisets (like `this.contents`), the `union` operator means the “counts” of all values are added.
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>( si = (true, ci, {13, 8}) )</td>
<td>( x = 13 )</td>
</tr>
<tr>
<td>( si.add(x); )</td>
<td></td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th><strong>Code</strong></th>
<th><strong>State</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>si = (true, ci, {13, 8})</code>&lt;br&gt;<code>x = 13</code>&lt;br&gt;<code>si.add(x);</code></td>
<td><code>si = (true, ci, {13, 13, 8})</code>&lt;br&gt;<code>x = 13</code></td>
</tr>
</tbody>
</table>
Example

Note the alias created here, which you cannot see in the tracing table.

\[
si = (true, ci, \{13, 8\})
\]

\[
x = 13
\]

\[
\text{si.add}(x);
\]

\[
si = (true, ci, \{13, 13, 8\})
\]

\[
x = 13
\]
void changeToExtractionMode()

• Change the mode of this from insertion to extraction.

• Updates: this.insertion_mode

• Requires: this.insertion_mode

• Ensures: not this.insertion_mode
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>si = (true, ci, {13, 13, 8})</code></td>
<td></td>
</tr>
<tr>
<td><code>si.changeToExtractionMode();</code></td>
<td></td>
</tr>
</tbody>
</table>
# Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$si = (\text{true}, \text{ci}, {13, 13, 8})$</td>
</tr>
<tr>
<td><code>si.changeToExtractionMode();</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$si = (\text{false}, \text{ci}, {13, 13, 8})$</td>
</tr>
</tbody>
</table>
removeFirst

T removeFirst()

• Removes and returns some “first” (“smallest”) element from the contents of this.

• Updates: this.contents

• Requires:

  not this.insertion_mode and
  this.contents /= {}

• Ensures:

  removeFirst is in #this.contents and
  for all x: T where (x is in this.contents)
    (this.ordering(removeFirst, x)) and
  this.contents = #this.contents \ {removeFirst}
removeFirst

T removeFirst()

• Removes and returns some "first" ("smallest") element from the contents of this.
• Updates: this.contents
• Requires: not this.insertion_mode and this.contents /= {}  
• Ensures:

removeFirst is in #this.contents and
for all x: T where (x is in this.contents)
    (this.ordering(removeFirst, x)) and
this.contents = #this.contents \ {removeFirst}

For multisets (like this.contents), is in means the “count” of the given value is positive, i.e., that value is an element of the multiset.
removeFirst

T removeFirst()

• Removes and returns some "first" ("smallest") element from the contents of this.

• Updates: this.

• Requires: not this.insertion_mode and this.contents /= {}

• Ensures: removeFirst is in #this.contents and for all x: T where (x is in this.contents) (this.ordering(removeFirst, x)) and this.contents = #this.contents \ {removeFirst}

For multisets (like this.contents), \ means the “counts” of all values are subtracted (but all remain non-negative).
## Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{si} = (false, ci, {13, 13, 8}) )</td>
<td></td>
</tr>
<tr>
<td>( x = -425 )</td>
<td></td>
</tr>
<tr>
<td>( x = \text{si}.\text{removeFirst}(); )</td>
<td></td>
</tr>
</tbody>
</table>
## Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
</table>
|                           | \( \text{si} = (\text{false, ci, } \{13, 13, 8\}) \)  
                           | \( x = -425 \)                             |
| \( x = \text{si}.removeFirst(); \) |                                            |
|                           | \( \text{si} = (\text{false, ci, } \{13, 13\}) \)  
                           | \( x = 8 \)                                |
The element removed is a “first” element based on the `IntegerLT` ordering, i.e., a smallest integer value.

\[
\text{Example}
\]

State

- \(si = (false, ci, \{13, 13, 8\})\)
- \(x = \text{-}425\)

\[x = si.removeFirst()\]

- \(si = (false, ci, \{13, 13\})\)
- \(x = 8\)
### Example: Remove Another

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>si = (false, ci, {13, 13})</code></td>
</tr>
<tr>
<td></td>
<td><code>x = 8</code></td>
</tr>
<tr>
<td><code>x = si.removeFirst();</code></td>
<td></td>
</tr>
</tbody>
</table>
### Example: Remove Another

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$si = (false, ci, {13, 13})$</td>
</tr>
<tr>
<td></td>
<td>$x = 8$</td>
</tr>
<tr>
<td>$x = si.removeFirst();$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$si = (false, ci, {13})$</td>
</tr>
<tr>
<td></td>
<td>$x = 13$</td>
</tr>
</tbody>
</table>
The element removed is a “first” element based on the IntegerLT ordering, i.e., a smallest integer value; another such element remains!

<table>
<thead>
<tr>
<th>State</th>
<th>x = si.removeFirst()</th>
</tr>
</thead>
<tbody>
<tr>
<td>(false, ci,</td>
<td>x = 8</td>
</tr>
<tr>
<td>{13, 13})</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>(false, ci,</td>
<td>x = 13</td>
</tr>
<tr>
<td>{13})</td>
<td></td>
</tr>
</tbody>
</table>

Example: Remove Another
isInInsertionMode

```java
boolean isInInsertionMode() {
    // Reports whether this is in insertion mode.
    // Ensures:
    boolean isInInsertionMode = this.insertion_mode;
}
```
Comparator<T> order()

• Reports Comparator<T> being used for sorting by this.

• Ensures:

  \[ order = \text{this}.ordering \]
size

`int size()`

• Reports the number of elements in `this.contents`.

• Ensures:

\[
size = |this.contents|
\]
size

int size()

• Reports the number of elements in this.contents.

• Ensures:

  size = |this.contents|

For multisets (like this.contents), \(|\cdot|\) means the sum of the multiset’s “counts” of all values.
iterator

Iterator<T> iterator()

• Returns an iterator over a multiset of elements of type T.

• Ensures:

\[ \text{multiset\_entries} \left( \sim this.\text{seen} \times \sim this.\text{unseen} \right) = this.\text{contents} \]
iterator

Iterator<T> iterator()

• Returns an iterator over a set of elements of type T.

• Ensures:

  \[
  \text{multiset\_entries} (\sim\text{this.seen} \ast \sim\text{this.unseen}) = \text{this.contents}
  \]

  \text{multiset\_entries} is like \text{entries} for a \text{string}, except that each string entry’s “count” in the \text{multiset} is the same as its (occurrence) \text{count} in the string.
Temporal Flexibility

• The contract for `SortingMachineKernel` does not tell the client when “sorting” occurs, so an implementation of `SortingMachineKernel` may pay the cost of comparing elements to each other:
  – During `add`
  – During `changeToExtractionMode`
  – During `removeFirst`
Instead of `sort` ...

```java
while (/*input values remain*/) {
    /* let x = next input value */
    q.enqueue(x);
}
q.sort(order);
while (q.length() > 0) {
    T x = q.dequeue();
    /* output x */
}
```
Instead of `sort` ...

```java
while (/*input values remain*/) {
    /* let x = next input value */
    sorter.add(x);
}

sorter.changeToExtractionMode();

while (sorter.size() > 0) {
    T x = sorter.removeFirst();
    /* output x */
}
```
Instead of sort ...

```java
while (/*input values remain*/) {
    /* let x = next input value */
    sorter.add(x);
}
sorter.changeToExtractionMode();
for (int i = 0; i < 50; i++) {
    T x = sorter.removeFirst();
    /* output x */
}
```

This is how you might remove only the “first 50” (“smallest 50”) values that were added to sorter.
Why Is This Better Than \texttt{sort}?

• If all elements are \textit{already sorted} by the end of \texttt{changeToExtractionMode}, then there is no difference in performance compared to using \texttt{sort}.

• If all elements are \textit{not already sorted} by the end of \texttt{changeToExtractionMode}, then there can be a performance advantage if you don’t need to remove all the elements!
  – See a future project ...
Sorting Algorithms

- Implementer of `SortingMachineKernel` may embed any *sorting algorithm* in it, e.g.:
  - *insertion sort* (add does all the work)
  - *quicksort* (changeToExtractionMode does all the work)
  - *selection sort* (removeFirst does all the work)
• Implementer of SortingMachineKernel may embed any sorting algorithm in it, e.g.:
  – insertion sort (add does all the work)
  – quicksort (changeToExtractionMode does all the work)
  – selection sort (removeFirst does all the work)

Actually, any sorting algorithm could be used when sorting is done entirely in changeToExtractionMode.
Resources

- OSU CSE Components API: SortingMachine
  - [http://cse.osu.edu/software/common/doc/](http://cse.osu.edu/software/common/doc/)