Queue
Queue

- The *Queue* component family allows you to manipulate strings of entries of any (arbitrary) type in *FIFO* (first-in-first-out) order
  - "First" here refers to the *temporal* order in which entries are put into the string and taken out of it, not about the left-to-right or right-to-left order in the string when it is written down
Interfaces and Classes

Standard

QueueKernel

Queue

Queue1L

Queue2

Standard has contracts for three methods:
- clear
- newInstance
- transferFrom

implements

extends

implements
QueueKernel has contracts for three methods:
- enqueue
- dequeue
- length
Queue has contracts for six other methods:
  - append
  - flip
  - front
  - replaceFront
  - sort
  - rotate
Mathematical Model

• The value of a Queue variable is modeled as a string of entries of type $T$

• Formally:

\[
\text{type Queue is modeled by string of } T
\]
Generics

• Note that Queue is a **generic type** (also called a **parameterized type**)

• The actual type of the entries is selected only later by the client when the type Queue is used to declare or instantiate a variable, e.g.:

```java
Queue<Integer> qi =
    new Queue1L<Integer>();
```
Generics

• Note that Queue is a generic type (also called a parameterized type).
  The actual type of the entries is selected only later by the client when the type Queue is used to declare or instantiate a variable, e.g.:
  
  ```java
  Queue<Integer> qi = new Queue1L<Integer>();
  ```

The formal type parameter was called T; here, the actual type or argument type is Integer.
Generics

• Note that Queue is a generic type (also called a parameterized type)

• The actual type of the entries is selected only later by the client when the type Queue is used to declare or instantiate a variable, e.g.:

```java
Queue<Integer> qi = new Queue1L<Integer>();
```

As of Java 7, generic arguments in a constructor call are inferred from the declared type...
Generics

• Note that Queue is a generic type (also called a parameterized type)

• The actual type of the entries is selected only later by the client when the type Queue is used to declare or instantiate a variable, e.g.:

  Queue<Integer> qi =
    new Queue<int> ();

... so this diamond operator means the same thing as the constructor with explicit generic arguments.
Wrapper Types

• Note the use of `Integer` here, not `int`
• Java demands that generic arguments must be `reference types`
• Each of the primitive types has a corresponding `wrapper type` that is a reference type (in part to satisfy this requirement)
## Wrapper Types

<table>
<thead>
<tr>
<th>primitive type</th>
<th>wrapper type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>char</td>
<td>Character</td>
</tr>
<tr>
<td>int</td>
<td>Integer</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
</tr>
</tbody>
</table>
Wrapper Types

• Each wrapper type is an immutable type
• There is a constructor from the corresponding primitive type
• Java includes features called auto-boxing and auto-unboxing so wrapper types can be used with primitive-type syntax almost as if they were primitive types
  – Details later (for now, look it up if it seems to matter to your code)
Constructors

• There is one *constructor* for each implementation class for `Queue`

• As always:
  – The name of the constructor is the name of the implementation class
  – The constructor has its own contract (which is in the kernel interface `QueueKernel`)
No-argument Constructor

• Ensures:

\texttt{this} = < >
Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Queue&lt;Integer&gt; qi = new Queue1L&lt;&gt;();</code></td>
<td></td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Queue&lt;Integer&gt; qi = new Queue1L&lt;&gt;();</code></td>
<td></td>
</tr>
<tr>
<td><code>qi = &lt; &gt;</code></td>
<td></td>
</tr>
</tbody>
</table>
Methods for Queue

• All the methods for Queue are instance methods, i.e., you call them as follows:
  q.methodName(arguments)
where q is an initialized non-null variable of type Queue<T> for some T
enqueue

void enqueue(T x)

• Adds \( x \) at the back (right end) of \( \text{this} \).
• Aliases: reference \( \text{x} \)
• Updates: \( \text{this} \)
• Ensures:

\[
\text{this} = \#\text{this} \star \langle x \rangle
\]
enqueue

void enqueue(T x)
• Adds \( x \) at the back (right end) of this.
• Aliases: reference \( x \)
• Updates: this
• Ensures:
  \[
  \text{this} = \#\text{this}
  \]

The list of references that might be aliases upon return from the method is advertised here, because aliasing is important and otherwise is not specified.
## Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>qi = &lt; 49, 3 &gt;</td>
<td>k = 70</td>
</tr>
<tr>
<td>qi.enqueue(k);</td>
<td></td>
</tr>
</tbody>
</table>

# Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>qi = \langle 49, 3 \rangle &lt;br&gt; k = 70</td>
<td>qi = \langle 49, 3, 70 \rangle &lt;br&gt; k = 70</td>
</tr>
<tr>
<td>qi.enqueue(k);</td>
<td></td>
</tr>
</tbody>
</table>
Meaning of “Aliases: ...”
Meaning of "Aliases: ..."

After...
Meaning of “Aliases: ...”

• The tracing table notation with $\rightarrow$ gives us no easy way to describe this situation
  – The picture is, however, a handy way to see what’s going on, so draw pictures!

• Since Integer is immutable, there is no consequence to this case of aliasing
  – But consider:

\[
\text{Queue<NaturalNumber> } qn = \ldots
\]
dequeue

T dequeue()

• Removes and returns the entry at the front (left end) of this.

• Updates: this

• Requires:
  
  this /= < >

• Ensures:
  
  #this = <dequeue> * this
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>qi = 49, 3, 70</td>
<td>k = -584</td>
</tr>
<tr>
<td>k = qi.dequeue();</td>
<td></td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>qi = &lt; 49, 3, 70 &gt;</code></td>
<td><code>qi = &lt; 49, 3, 70 &gt;</code></td>
</tr>
<tr>
<td><code>k = -584</code></td>
<td></td>
</tr>
<tr>
<td><code>k = qi.dequeue();</code></td>
<td></td>
</tr>
</tbody>
</table>

```cpp
k = qi.dequeue();
```

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>qi = &lt; 3, 70 &gt;</code></td>
<td><code>qi = &lt; 3, 70 &gt;</code></td>
</tr>
<tr>
<td><code>k = 49</code></td>
<td></td>
</tr>
</tbody>
</table>

```cpp
k = 49
```
\textbf{length}

\texttt{int length()}

- Reports the length of \texttt{this}.
- Ensures:

\[ length = |this| \]
<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>qi = &lt; 49, 3, 70 &gt;</td>
<td>n = -45843</td>
</tr>
<tr>
<td>n = qi.length();</td>
<td></td>
</tr>
</tbody>
</table>
## Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>qi = (&lt; 49, 3, 70 &gt;)</td>
<td>n = -45843</td>
</tr>
<tr>
<td>n = qi.length();</td>
<td></td>
</tr>
<tr>
<td>qi = (&lt; 49, 3, 70 &gt;)</td>
<td>n = 3</td>
</tr>
</tbody>
</table>
front

T front()

• Returns the entry at the the front (left end) of this.

• Aliases: reference returned by front

• Requires:
  
  this /= < >

• Ensures:
  
  <front> is prefix of this
## Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>qi = &lt; 49, 3, 70 &gt;</td>
<td>k = -58</td>
</tr>
<tr>
<td>k = qi.front();</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>qi = &lt; 49, 3, 70 &gt;&lt;br&gt;k = -58&lt;br&gt;k = qi.front();</td>
<td>qi = &lt; 49, 3, 70 &gt;&lt;br&gt;k = 49</td>
</tr>
</tbody>
</table>
Meaning of “Aliases: ...”

Before...

-58

49

3

70

k

qi
Meaning of “Aliases: …”
replaceFront

T replaceFront(T x)
• Replaces the front of \texttt{this} with \texttt{x}, and returns the old front.
• Aliases: reference \texttt{x}
• Updates: \texttt{this}
• Requires:
  \texttt{this} /= < >
• Ensures:
  \texttt{<replaceFront>} is prefix of \texttt{#this} and \texttt{this} = \texttt{<x>} * \texttt{#this[1, |#this|]}
## Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>qi = &lt; 49, 70 &gt; k = -58 j = 16</td>
<td></td>
</tr>
<tr>
<td>k = qi.replaceFront(j);</td>
<td></td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k = qi.replaceFront(j);$</td>
<td>$qi = &lt; 49, 70 &gt;$</td>
</tr>
<tr>
<td></td>
<td>$k = -58$</td>
</tr>
<tr>
<td></td>
<td>$j = 16$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$qi = &lt; 16, 70 &gt;$</td>
</tr>
<tr>
<td></td>
<td>$k = 49$</td>
</tr>
<tr>
<td></td>
<td>$j = 16$</td>
</tr>
</tbody>
</table>
Meaning of “Aliases: …”

Before...
Meaning of “Aliases: ...”
Another Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>qi = &lt; 49, 70 &gt; j = 16</td>
<td></td>
</tr>
<tr>
<td>j = qi.replaceFront(j);</td>
<td></td>
</tr>
</tbody>
</table>
## Another Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>qi = (&lt; 49, 70 &gt;) j = 16</td>
<td>qi = (&lt; 49, 70 &gt;) j = 16</td>
</tr>
<tr>
<td>(j = qi.replaceFront(j);)</td>
<td>qi = (&lt; 16, 70 &gt;) j = 49</td>
</tr>
</tbody>
</table>
Another Example

This use of the method avoids creating an alias: it swaps $j$ with the entry previously at the front.

\[
j = \text{qi}.\text{replaceFront}(j);
\]

<table>
<thead>
<tr>
<th>State</th>
<th>qi = $\langle 49, 70 \rangle$</th>
<th>j = 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>qi = $\langle 16, 70 \rangle$</td>
<td>j = 49</td>
<td></td>
</tr>
</tbody>
</table>
append

**void** append(Queue<T> q)

- Concatenates ("appends") \( q \) to the end of \( \textit{this} \).
- Updates: \( \textit{this} \)
- Clears: \( q \)
- Ensures:

\[
\textit{this} = \#\textit{this} \times \#q
\]
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
</table>
| q1 = [4, 3, 2]  
q2 = [1, 0]  
q1.append(q2); | |

```python
q1 = [4, 3, 2]  
q2 = [1, 0]  
q1.append(q2);
```
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$q_1 = &lt; 4, 3, 2 &gt;$</td>
</tr>
<tr>
<td></td>
<td>$q_2 = &lt; 1, 0 &gt;$</td>
</tr>
<tr>
<td>$q_1.append(q2);$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$q_1 = &lt; 4, 3, 2, 1, 0 &gt;$</td>
</tr>
<tr>
<td></td>
<td>$q_2 = &lt; &gt;$</td>
</tr>
</tbody>
</table>
flip

void flip()

• Reverses ("flips") this.
• Updates: this
• Ensures:

  \[ this = \text{rev}(\#this) \]
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>qi = \langle 18, 6, 74 \rangle</td>
<td>qi = \langle 18, 6, 74 \rangle</td>
</tr>
<tr>
<td>qi.flip();</td>
<td></td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>qi = &lt; 18, 6, 74 &gt;</td>
<td>qi = &lt; 18, 6, 74 &gt;</td>
</tr>
<tr>
<td>qi.flip();</td>
<td></td>
</tr>
<tr>
<td></td>
<td>qi = &lt; 74, 6, 18 &gt;</td>
</tr>
</tbody>
</table>
sort

**void** sort(Comparator<T> order)

- Sorts **this** according to the ordering provided by the `compare` method from `order`.
- Updates: **this**
- Requires:
  
  ```plaintext
  [the relation computed by order.compare is a total preorder]
  ```
- Ensures:
  
  ```plaintext
  this = [#this ordered by the relation computed by order.compare]
  ```
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);
}
```
Comparators

• The notion of “less than” and “greater than” is quite flexible
• The notion of “equal to” is not flexible
  – It is based on mathematical equality, not on a flexible notion of being “equivalent to”
• There are important technicalities...
sort

A **total preorder** means that any two values of type `T` are comparable, and that there are no “cycles”, e.g., nothing like `a < b < c < a`.

- **Requires:**
  
  ![The relation computed by `order.compare` is a total preorder]

- **Ensures:**

  `this = [#this ordered by the relation computed by `order.compare`]`
Creating a Comparator

```java
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;
        }
    }
}
```
Creating a Comparator

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.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 to simplify the code for compare in a Comparator<T> implementation.
## Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>qi = &lt; 8, 6, 92, 1 &gt;</td>
<td></td>
</tr>
<tr>
<td>Comparator&lt;Integer&gt; ci = new IntegerLT (); qi.sort(ci);</td>
<td></td>
</tr>
</tbody>
</table>
## Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>qi = &lt; 8, 6, 92, 1 &gt;</code></td>
<td><code>Comparator&lt;Integer&gt; ci = new IntegerLT (); qi.sort(ci)</code></td>
</tr>
<tr>
<td><code>qi = &lt; 1, 6, 8, 92 &gt;</code></td>
<td></td>
</tr>
</tbody>
</table>
rotate

**void** rotate**(int** distance)

- Rotates **this**.
- Updates: **this**
- Ensures:

  ```
  if #this = <> then
    this = #this
  else
    this =
      #this[distance mod |#this|, |#this|) * #this[0, distance mod |#this|)
  ```
Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>qi = [8, 6, 92, 3]</td>
<td>qi = [8, 6, 92, 3]</td>
</tr>
<tr>
<td>qi.rotate(1);</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>qi = &lt; 8, 6, 92, 3 &gt;</td>
<td>qi = &lt; 8, 6, 92, 3 &gt;</td>
</tr>
<tr>
<td><strong>qi.rotate(1);</strong></td>
<td>qi = &lt; 6, 92, 3, 8 &gt;</td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>qi = &lt; 8, 6, 92, 3 &gt;</td>
<td></td>
</tr>
<tr>
<td>qi.rotate(3);</td>
<td></td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>qi = &lt; 8, 6, 92, 3 &gt;</td>
<td>qi = &lt; 3, 8, 6, 92 &gt;</td>
</tr>
<tr>
<td>qi.rotate(3);</td>
<td></td>
</tr>
</tbody>
</table>

### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>qi = &lt; 8, 6, 92, 3 &gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>qi.rotate(-1);</code></td>
<td></td>
</tr>
</tbody>
</table>
## Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>qi = [8, 6, 92, 3]</td>
<td>qi = [8, 6, 92, 3]</td>
</tr>
<tr>
<td>qi.rotate(-1);</td>
<td>qi = [3, 8, 6, 92]</td>
</tr>
</tbody>
</table>
Resources

• OSU CSE Components API: Queue

• Java Libraries API: Comparator, Comparable
  – [http://docs.oracle.com/javase/8/docs/api/](http://docs.oracle.com/javase/8/docs/api/)