Queues

- Linear list.
- One end is called front.
- Other end is called rear.
- Additions are done at the rear only.
- Removals are made from the front only.
Bus Stop Queue

The Interface Queue

```java
public interface Queue
{
    public boolean isEmpty();
    public Object getFrontElement();
    public Object getRearElement();
    public void put(Object theObject);
    public Object remove();
}
```

Revisit Of Stack Applications

- Applications in which the stack cannot be replaced with a queue.
  - Parentheses matching.
  - Towers of Hanoi.
  - Switchbox routing.
  - Method invocation and return.
  - Try-catch-throw implementation.
- Application in which the stack may be replaced with a queue.
  - Rat in a maze.
    - Results in finding shortest path to exit.

Wire Routing
Label all reachable squares 1 unit from start.

Label all reachable unlabeled squares 2 units from start.

Label all reachable unlabeled squares 3 units from start.

Label all reachable unlabeled squares 4 units from start.
Label all reachable unlabeled squares 5 units from start.

End pin reached. Traceback.

Label all reachable unlabeled squares 6 units from start.

End pin reached. Traceback.
Derive From ArrayLinearList

- when front is left end of list and rear is right end
  - Queue.isEmpty() => super.isEmpty()
    - $O(1)$ time
  - getFrontElement() => get(0)
    - $O(1)$ time
  - getRearElement() => get(size() - 1)
    - $O(1)$ time
  - put(theObject) => add(size(), theObject)
    - $O(size)$ time
  - remove() => remove(0)
    - $O(size)$ time

Derive From ExtendedChain

- when front is left end of list and rear is right end
  - Queue.isEmpty() => super.isEmpty()
    - $O(1)$ time
  - getFrontElement() => get(0)
    - $O(1)$ time
  - getRearElement() => get(size() - 1)
    - $O(1)$ time
  - put(theObject) => add(0, theObject)
    - $O(size)$ time
  - remove() => remove(size() - 1)
    - $O(1)$ time

- to perform each operation in $O(1)$ time (excluding array doubling), we need a customized array representation.
Derive From ExtendedChain

- getRearElement() => getLast() … new method
  - O(1) time
- put(theObject) => append(theObject)
  - O(1) time
- remove() => remove(0)
  - O(1) time

Custom Linked Code

- Develop a linked class for Queue from scratch to get better performance than obtainable by deriving from ExtendedChain.
Custom Array Queue

• Use a 1D array queue.
  queue[]

• Circular view of array.

Custom Array Queue

• Possible configuration with 3 elements.

Custom Array Queue

• Another possible configuration with 3 elements.

Custom Array Queue

• Use integer variables front and rear.
  – front is one position counterclockwise from first element
  – rear gives position of last element

Custom Array Queue
Add An Element

- Move rear one clockwise.

Remove An Element

- Move front one clockwise.

Add An Element

- Move rear one clockwise.
- Then put into queue[rear].

Remove An Element

- Move front one clockwise.
- Then extract from queue[front].
Moving rear Clockwise

- \texttt{rear++;
  if (rear == queue.length) rear = 0;}

- \texttt{rear = (rear + 1) \% queue.length;}

Empty That Queue
• When a series of removes causes the queue to become empty, $\text{front} = \text{rear}$.
• When a queue is constructed, it is empty.
• So initialize $\text{front} = \text{rear} = 0$. 

• When a series of removes causes the queue to become empty, $\text{front} = \text{rear}$.
• When a queue is constructed, it is empty.
• So initialize $\text{front} = \text{rear} = 0$. 

• When a series of removes causes the queue to become empty, $\text{front} = \text{rear}$.
• When a queue is constructed, it is empty.
• So initialize $\text{front} = \text{rear} = 0$. 

• When a series of removes causes the queue to become empty, $\text{front} = \text{rear}$.
• When a queue is constructed, it is empty.
• So initialize $\text{front} = \text{rear} = 0$. 

• When a series of removes causes the queue to become empty, $\text{front} = \text{rear}$.
• When a queue is constructed, it is empty.
• So initialize $\text{front} = \text{rear} = 0$. 

• When a series of removes causes the queue to become empty, $\text{front} = \text{rear}$.
• When a queue is constructed, it is empty.
• So initialize $\text{front} = \text{rear} = 0$. 

• When a series of removes causes the queue to become empty, $\text{front} = \text{rear}$.
• When a queue is constructed, it is empty.
• So initialize $\text{front} = \text{rear} = 0$. 

• When a series of removes causes the queue to become empty, $\text{front} = \text{rear}$.
• When a queue is constructed, it is empty.
• So initialize $\text{front} = \text{rear} = 0$. 

When a series of adds causes the queue to become full, \( \text{front} = \text{rear} \).
- So we cannot distinguish between a full queue and an empty queue!

**Ouch!!!!!**

- Remedies.
  - Don’t let the queue get full.
    - When the addition of an element will cause the queue to be full, increase array size.
    - This is what the text does.
  - Define a boolean variable \( \text{lastOperationIsPut} \).
    - Following each put set this variable to true.
    - Following each remove set to false.
    - Queue is empty iff \( (\text{front} == \text{rear}) \) \&\& \( !\text{lastOperationIsPut} \)
    - Queue is full iff \( (\text{front} == \text{rear}) \) \&\& \( \text{lastOperationIsPut} \)

**Ouch!!!!!**

- Remedies (continued).
  - Define an integer variable \( \text{size} \).
    - Following each put do \( \text{size}++ \).
    - Following each remove do \( \text{size}-- \).
    - Queue is empty iff \( (\text{size} == 0) \)
    - Queue is full iff \( (\text{size} == \text{queue.length}) \)
  - Performance is slightly better when first strategy is used.