A **queue** is like a line of people. The first person to join a line is the first person served and is thus the first to leave the line. Queues are appropriate for many real-world situations. For instance, we often wait in a queue to buy a movie ticket, or to use an automatic ticket machine. The person at the front of the queue is served, while new people join the queue at its back. A queue is then a form of list. The difference from the previous forms of list is in its access procedures.

Whereas a stack can be seen as a list with only one end, because all the operations are performed at the top of the stack, the queue can be seen, in contrast, as a list with two ends, the **front** and the **back**.

Queues have the main property of allowing new items to be added only at the back of the list, and the item first inserted in the list (i.e. the item at the front of the list) is the first to be removed. This behaviour is commonly referred to as “first-in-first-out” or simply **FIFO**. So the first item in a queue is the “front” item, whereas the last item in a queue is the “back” item.

The access procedures for a queue include operations such as examining whether the queue is empty, inspecting the item at the front of the queue but not others, placing an item at the back of the queue, but at no other position, and removing an item from the front of the queue, but from no other position. The next slides will describe these procedures in more detail.

A typical application of queues in computer science is:

- resource allocation by operating systems on multi-user machines. For instance, if you share a printer with other users, a request to print enters a queue to wait its turn.
Access Procedures

Constructor operations to construct a queue:

i. `createQueue()`  
   // post: creates an empty queue

ii. `enqueue(newItem)`  // [equivalent to `add(newItem)`]  
   // post: adds newItem at the end of the queue.

Predicate operations to test Queues

i. `isEmpty()`  
   // post: determines whether a queue is empty

Selector operations to select items of a queue

i. `front()`  
   // post: returns the item at the front of the queue. It does not change the queue.

ii. `dequeue()`  
   // post: retrieves and removes the item at the front of the queue.

This slide provides the list of the main access procedures and operations for the ADT queue. The operation “enqueue”, also called add, is a short form of “entering the queue”: it adds a new item to the end of a queue. The operation for retrieving and removing an item from a queue is called a “dequeue” operation, which means deleting from the queue. The operation “front” retrieves the item at the front of a queue leaving the queue unchanged.

Note that so far we have defined the access procedure operation delete for an ADT as just a procedure that deletes an item from the ADT. In this case the procedure dequeue also returns the element that is deleted from the queue. We could have still defined this operation as simply deleting the first element from a queue, and use it in combination with `front()` in order to get the next element in the queue. We have adopted here this definition, because this is the one most commonly used in text books.

So far we have only given the definition of these operations in terms of their partial post-conditions. Note that these post-conditions do not include the exception cases. We’ll show later which of these access procedures require exceptions.
Using a Queue: a simple example

A method `ReadString` that reads a string of characters from a single line of input into a queue:

The pseudocode:

```plaintext
ReadString( )
// post: reads a string of characters from a
// single line of input into a queue;
queue.createQueue( );
while (not end of line) {
    Read a new character ch;
    queue.enqueue(ch);
} //end while
```

This slide presents a simple application of the ADT queue. The example uses operations of the ADT queue independently from their implementations.

When we enter characters at a keyboard, the system must retain them in the order in which you typed them. For this purpose an ADT queue is more appropriate than an ADT stack. A pseudocode solution to this problem is given in this slide.

Once the characters are in a queue, the system can process them as necessary.

Note that this example is just in pseudocode. In a real Java implementation, to create a queue we would need to use the constructor of the class that implements the queue. Let’s see in the next few slides how we can declare such a class in both a dynamic and static implementation of queues.
Axioms for ADT Queues

The access procedures must satisfy the following axioms, where Item is an item and aQueue is a given queue:

1. \((\text{aQueue.createQueue( )).isEmpty} = \text{true}\)
2. \((\text{aQueue.enqueue(Item)).isEmpty()} = \text{false}\)
3. \((\text{aQueue.createQueue( )}).\text{front( )} = \text{error}\)
4. \((\text{aQueue.enqueue(Item)).\text{front( )} = \text{Item}\)
5. \((\text{aQueue.createQueue( )}).\text{dequeue( )} = \text{error}\)
6. \(\text{pre: aQueue.isEmpty( )}=\text{true}:\)
7. \((\text{aQueue.enqueue(Item)).\text{dequeue( )}=(\text{aQueue.dequeue()}).\text{enqueue(Item)}\)

A queue is like a list with \(\text{front( ), dequeue( ) and enqueue(item)}\) equivalent to get 1st item, get and remove 1st item, and add item at the end of a list respectively.

In this slide I have listed the main axioms that the access procedures for an ADT queue have to satisfy. In this case, we have two axioms for each operation.

Note that the access procedures for a queue are somewhat equivalent to the access procedures for a list. In particular, the \(\text{front( )}\) procedure for a queue can be seen as equivalent to the \(\text{get(1)}\) procedure for a list, which takes the first element of a given list. In the same way, the procedure \(\text{dequeue( )}\) for a queue is similar to the procedure \(\text{remove(1)}\), which removes the first element in a list, (but in addition it returns the value of the removed item), and the procedure \(\text{enqueue(item)}\) is equivalent to the procedure \(\text{add(list.size( ) +1, item)}\), where list is the specific data structure used to implement the queue. This last operation adds the given new item to the end of a list.

In the next slides we’ll look at the two different types of implementations of a queue, one based on arrays and the other based on linked lists. The definition of the data structure for a queue in the case of a dynamic implementation will further illustrate the fact that queues are essentially lists with specific use of their access procedures.
The interface QueueInterface given in this slide provides the main definition of each access procedure for the ADT queue. As in the case of the ADT list, the constructor createQueue() is not included here as it is normally given as constructor of the particular Java class that implements the Queue.

The two main types of implementations (array-based, reference-based) for a queue are classes that implement this interface. This interface provides, therefore, a common specification for the two implementations that we are going to illustrate in the next few slides: an array-based and a reference-based implementation.

An example of the QueueException for a queue can be the following:

```java
public class QueueException extends java.lang.RuntimeException {
    public QueueException(String s)
        super(s);
    } // end constructor
} // end QueueException
```

Note that this exception can be avoided by including in the program application method a check to see whether the queue is empty before calling any of the access procedures front(), dequeue(), and enqueue(item). In the case of enqueue(item), if the implementation is a static implementation using arrays, an exception caused by the underlying array data structure being full needs also to be considered. In a reference-based implementation, such exception is not necessary since the memory is dynamically allocated and not fixed a priori.
Like stacks, queues can have an array-based or a reference-based implementation. For the queues the reference-based implementation is a bit more straightforward than the array-based one, so we start with this.

A reference-based implementation of a queue can use the same data structure as that used by a linked list, but with two external references, one to refer to the front of the queue, i.e. the first item in the queue, and one to refer to the back of the queue, i.e. the last item in the queue. We call these two reference variables “first” and “last” respectively.

When we first create an empty queue (no item is inserted), the two instance variables first and last still exist and their reference value is null. This is shown in the main constructor of the class `QueueReferenceBased` given here. This corresponds diagrammatically to the following situation:

The implementation of the operation `isEmpty()` is therefore pretty simple, and it consists of checking that the reference variable last is equal to null, as shown below. Note that it is actually sufficient to check that either of the two variables last or first is equal to null. If the queue were including at least one element both first and last reference variables would be different from null, as shown in the next slide.

```java
public boolean isEmpty() {
    return last == null;
}
```
Dynamic Implementation of a Queue (continued)

Inserting a new Node in a queue

1. last.setNext(newNode);
2. last = newNode;

For non-empty queues

Deleting a Node in a queue

1. first = first.getNext( );

For non-empty queues.

Insertion at the back of a non-empty queue is straightforward. A diagrammatical representation is given in this slide, where the dashed lines indicate reference values before the operation. Addition of an item at the back of a non-empty queue requires two reference changes: the “next” reference in the back node needs to be changed so as to reference to newNode instead of null (this is denoted with ① in the diagram), and the external reference “last” needs to be changed so as to reference also newNode (this is denoted with ② in the diagram).

When the queue is empty both instance variables first and last have value equal to null. The addition of one new item requires then the variable “first” and also “last” to reference the new variable Node since the single item in the queue is both the first and the last item.

The case of deleting an item from the front of a non-empty queue requires just one reference change: the “first” variable has now to reference the second item in the list.

Note that if the queue includes only one single element, then the command “first = first.getNext( )” will automatically set the reference variable equal to null, but we will still need to change the value of the reference variable “last” to be equal to null.
Dynamic Implementation of a Queue (continued)

public class QueueReferenceBased implements QueueInterface{
    private Node first;
    private Node last;
    public void enqueue(Object item) {
        Node newNode = new Node(item);
        if (isEmpty( )) {                                      // insertion into empty queue
            first = newNode;}                             // new node is referenced by first
        else {last.setNext(newNode);}               // insertion into non-empty queue
        last = newNode;
    }
    public Object dequeue( ) throws QueueException{
        if (!isEmpty( )) {
            Node firstNode = first;
            if (first = = last) {last = null;}         // queue is not empty. Retrieve the front item;
            return firstNode.getItem( );
        } else {throws new QueueException(“QueueException on dequeue: empty”);}  // first = first.getNext( );
        else {throws new QueueException(“QueueException on dequeue: empty”);}  // last = last.getNext( );
        else{
        }
    }
}
This slide gives the basic implementation of the operations “add” and “dequeue” for a queue. Note a slightly different implementation would be required if instead of using just a linked list as data structure, we had chosen to use a circular linked list.

What is missing is an example implementation of the access procedure “front ( )”. The code is given below:

public Object front( ) {
    if (!isEmpty( )) {
        return first.getItem( );
    } else {
        throws new QueueException(“QueueException on front: queue empty”);}
}
Note that a different data structure could have been chosen to implement the queue dynamically. An example could have been to use a circular linked list and just a single reference variable referring to the last node in the circular list. This would have allowed us to get the reference to the first item in a queue by just getting the link value included in the last item of the queue. This example implementation is given in the tutorial sheet 3.
For applications where a fixed-sized queue does not present a problem, we could also use an array data structure to represent a queue. A naïve array-based implementation of a queue is to consider an array, whose size gives the maximum size of the queue, and two variables “first” and “last”, which contain, respectively, the index value of the first item and the last item in the queue. In the slide we have named this array “items”. At the beginning “first=0” and “last = -1” to indicate that the queue is empty. To add an item to the queue, we would in this case increment the value of “last” and place the new item in items[last]. To dequeue an item we would just need to increment the variable “first”. To check if the queue is empty we would just need to check that last<first. On the other hand the queue is full when last=max_queue-1.

One of the problem with this implementation is that after a sequence of enqueue and dequeues, the items in the queue will drift toward the end of the array, and last could become equal to max_queue-1 without the array being full. We could think of shifting the elements in the array to the left after each dequeue operation or whenever last becomes equal to max_queue-1. But this is not an efficient solution.

A more elegant solution is to view the array as circular. In this case first and last advance clockwise to dequeue the first item and add a new item respectively. At the beginning, the empty queue is given with first=0 and last=max_queue-1. To obtain the effect of circular array, the values of first and last are calculated using modulo arithmetic (that is the Java % operator) as illustrated in the slide. The main question is now: what conditions define the queue to be full and what conditions define the queue to be empty? We can think of the queue to be empty whenever first passes the value of last. However, this condition would also be possible when the queue becomes full. Because the array is circular consecutive add operations will just increase the value of last until it will eventually catch up with first (i.e. when the queue is full). We therefore need to have a way to distinguish between these two different situations. This can be implemented using a “counter” variable, called here “count”, which is incremented at each enqueue operation and decremented at each dequeue operation. count=0 will indicate an empty queue, whereas count ≠ 0 will indicate a non-empty queue.
A circular array implementation of a Queue

```java
public class QueueArrayBased implements QueueInterface {
    private final int max_queue = 50;
    private Object items[];  // an array of list items
    private int first, last, count; // number of items in the list

    public QueueArrayBased() {
        items = new Object[max_queue];
        first = 0; count = 0; last = max_queue-1;
    } // end default constructor;

    public void enqueue(Object newItem) throws QueueException {
        if (!isFull())
            {last = (last+1) % (max_queue);
                items[last]=newItem;
                count++;
            }
        else {throw new QueueException("Queue is full");}
    }
```

Example implementations for the main access procedures of a queue in the case of circular array implementation of the queue, are given here and in the next slide.
The ADT Queues

...A circular array implementation of a Queue

public Object dequeue( ) throws QueueException {
    if (!isEmpty( ))
        { Object queuefront = item[first];
            first=(first+1)%max_queue;
            count--;
            return queuefront;
        }
    else {throw new QueueException("Queue is empty");}
}

public isFull( )
{
    return count == max_queue;
}

What would be the implementation of the access procedures isEmpty( ) and front( )? This are left as exercise because they are really very simple.
The ADT Priority Queue

Definition

- The **ADT Priority Queue** is a linear sequence of an arbitrary number of items, **which are ordered**, together with access procedures.
- The access procedures permit **addition** of items into the queue in priority order, not at the back, and **deletion** of items with highest priority first. If several items have same priority, the one that was placed first in the priority queue is removed first.

We might think of implementing it as a queue, but the class Node has to include also information about priority. The enqueue and dequeue operations will also have to take into account the priorities.

Using a queue ensures that customers are served in the exact order in which they arrive. However, we often want to assign priorities to customers and serve the higher priority customers before those with lower priority. For instance, a computer operating system that keeps a queue of programs waiting to use a printer, may give priority to short programs and lower priority to longer programs. A “priority queue” is an ADT that stores items along with a priority for each item. Items are removed in order of priority.

We might think of implementing this specific ADT using similar technique to that used for the standard ADT queues. The class Node will have to be changed so as to include information about the priority of the individual item in the queue. In the same way the implementation of the operations “enqueue” and “dequeue” will have to take into account the priority value of the item to be inserted or deleted from the priority queue.

In the array-based implementation, we might think of having the items inserted in such a way that they are sorted in order of priority value. The add operation in this case will have to first use a binary search to find the correct position for insertion in the array, and then shift the elements in the array to make room for the new item.

In the case of a reference-based implementation, items could be thought of as organised in descending order of priority value, so that the item with the highest priority value is at the beginning of the linked list.
The ADT Double-Ended Queue

**Definition**
- The ADT Double-ended Queue is a queue.
- The access procedures permit **addition** of items and **deletion** of items at both ends of the sequence.

A possible dynamic implementation is to have a double-linked list with two additional reference variables to refer to the first and last items in the queue.

Another particular form of queue is a Double-Ended queue, also called DEQueue. This is similar to a queue with the particular feature that any item can be added or deleted from the queue at either end. A diagrammatical representation is given in this slide.

A possible dynamic implementation is to use a double-linked list with the addition of two extra reference variables, called first and last, which refer to the first item and the last item in the queue.

The list of access procedure properties for this special type of queues is given in the next slide.
Access Procedures for DEQueue

i. createDeQueue( )
   // post: creates an empty doubled-ended queue

ii. addLeft(newItem)
    // post: adds newItem at the front of the queue.

iii. addRight(newItem)
     // post: adds newItem at the end of the queue.

iv. isEmpty( )
    // post: determines whether a queue is empty

i. front( )
   // post: returns the item at the front of the queue. It does not change the queue.

ii. back( )
    // post: returns the item at the back of the queue. It does not change the queue.

iii. dequeueLeft( )
     // post: changes the queue by removing the item at the front of the queue.

iv. dequeueRight( )
     // post: changes the queue by removing the item at the back of the queue.

Constructor operations

Predicate operations
An ADT Queue is:

- A queue is a First-in/First-out (FIFO) data structure.
- The insertion and deletion operations require efficient access to both ends of the queue. Reference-based implementations use a linear linked list that has both a first item reference and a last item reference; or it uses a circular linked list with just a last item reference.
- Array-based implementations are prone to “rightward drift”, which makes queues look full when they are not. More efficient solutions use a circular array.
- In a circular array we must distinguish between the queue-full and queue-empty conditions. This can be done using appropriate counters.
- Priority queues and double ended queues are specific types of queues. Their implementation varies from the implementation of an ordinary queue.