# Queue ADT

## 1 The Queue ADT

Queue represent another form of linear data structure. As in the case of the stack, the idea of a queue is to represent sequences of elements, all having the same type. As in the case of stacks, the order in which elements are indicated is part of the definition (thus sequences with the same elements in different order represent distinct values).

The difference between queues and stacks comes from the fact that, while stacks allow insertion and deletion of elements at one extreme of the sequence, queues allow insertion and deletion to occur only at opposite extremes of the sequence. Addition of new elements can be only performed at one fixed extreme of the sequence, while removal of elements can be only performed at the opposite extreme of the sequence. See Figure 1.

![Queue Structure](image)

This sort of structure is usually indicated with the term *FIFO* (First in First out), as elements can be removed only in the same order as they are inserted (the first element inserted will the first to be removed, the second element inserted will be the second to be removed, etc.).

Thus, this data type relies on four key operations:

- **enqueue** which inserts a new element in the queue; the new element becomes the new *tail* of the queue;
- **dequeue** which removes one element from the queue; the element removed is the current head of the queue; the new head will be the immediately preceding element in the sequence;
- **front** and **rear** which simply read the value which is located respectively in the head and in the tail of the queue.

Figure 2 shows with some examples the effect of the various operations on a queue.
Figure 2: Queue Example
2 Specification

Set of values: the set of possible values is represented by the set of all possible sequences of elements; i.e., each value is of the form \((a_1, \ldots, a_n)\), where \(a_i\) identifies the \(i^{th}\) element in the sequence.

In the queue we need to identify, at every point in time, the two elements that are at the two extremes of the stack. The first element in the queue is always indicated with the name head (see Figure 1). The last element in the queue is indicated with the name tail. Thus the head is the first element which has been inserted in the queue (the oldest element in the queue) while the tail is the last element inserted in the queue (the youngest element in the queue).

Operations:

- **Constructors**: the only constructor needed is the one to create a new object of type queue.
  
  \textbf{CreateQueue}  
  
  - preconditions: none  
  - postconditions: returns a new object of type queue containing, as value, the empty queue (i.e., the sequence containing zero elements);

- **Destructors**: the only destructor needed is used to trash a queue which is not needed any longer.
  
  \textbf{DeleteQueue}  
  
  - preconditions: operates on the current queue  
  - postconditions: none

- **Inspectors**: the following inspectors are included in the type specification:
  
  1. \textbf{queueSize} used to compute the size of a queue (i.e., the number of elements present in a given queue)  
     - preconditions: operates on the current queue  
     - postconditions: produces a number representing the count of the elements in the queue
  2. \textbf{emptyQueue} used to verify whether a given queue is empty or not  
     - preconditions: operates on the current queue  
     - postconditions: produces \textit{true} if the queue is empty, \textit{false} otherwise
  3. \textbf{equalQueues} used to verify whether two queues have the same value (i.e., they contain the same elements and in the same order).  
     - preconditions: receives one queue as input and operates on the current queue  
     - postconditions: returns \textit{true} if the values of the two queues contain the same elements and in the same order, \textit{false} otherwise.
  4. \textbf{front} returns the information which is currently stored as first element (the head) in the queue (i.e., the oldest element in the queue, which is also the element which is going to be removed in the next removal operation).  
     - preconditions: operates on the current queue  
     - postconditions: if the queue is not empty, then it returns the first element in the queue \(s\) (the head), otherwise it gives an error
  5. \textbf{rear} returns the information which is currently stored as last element (the tail) in the queue (i.e., the youngest element in the queue, which is also the element which has been inserted in the last enqueue operation).  
     - preconditions: operates on the current queue
postconditions: if the queue is not empty, then it returns the last element in the queue (the tail), otherwise it gives an error

• **Modifiers:** the following modifiers are included

1. **enqueue** used to insert a new element at the end of the queue
   - preconditions: receives as input an element \( x \) and operates on the current queue
   - postconditions: if \( x \) is of the correct type (i.e., it has the same type as all the other elements in the queue), then the queue is modified introducing the element \( x \) in the last position of the queue (i.e., the element \( x \) becomes the new tail of the queue); otherwise it returns an error

This is illustrated in Figure 3.

2. **dequeue** removes the first element from the queue (the head of the queue)
   - preconditions: operates on the current queue
   - postconditions: if the queue is not empty, then the queue is modified by removing the element in first position in the queue (the head of the queue); otherwise it returns an error

This is illustrated in Figure 4.

3. **copyQueue** creates a copy of the current queue
   - preconditions: it operates on the current queue
   - postconditions: copies the value of the queue in a newly created queue.

## 3 Public Part of the Implementation

The public part of the implementation can be obtained as a direct translation of the ADT specification. As we have done before, we start by providing an interface which specifies the prototypes of the methods that will be common to any implementation of the queue ADT.

```c
/**
 * Interface of the QUEUE ADT
 * queues are used to store sequences of objects of a given
 * type. The ADT allows to directly access the first and the
 * last element of the sequence. It is possible to modify the
 */
```
Figure 4: Dequeue Operation

* sequence by inserting elements at one side of the sequence
* and removing them from the opposite side. The interface is
* for queues of integers
* @author Enrico Pontelli
***/

public interface intQueueInterface {
    /** the constructors and destructors are omitted
        as they will be specific to the particular implementation
        of queues. ***/

    /*-------------------*/
    /* Inspectors */
    /*-------------------*/

    /**
        * Verify if queue is empty
        * @return true if queue is empty, false otherwise
        ***/
    public boolean emptyQueue();

    /**
        * Compute how long is the sequence of elements in the queue
        * @return the number of elements in the queue
        ***/
    public int computeQueueLength();
}
public int queueSize();

/***
 * Determines the current head of the queue
 * @return the integer in the head position of the queue
 ***/
public int front();

/***
 * Determines the current tail of the queue
 * @return the integer in the tail position of the queue
 ***/
public int rear();

FLICTHERE

public boolean enqueue (int element);

/***
 * remove an element from a queue and the element removed
 * is the current head of the queue
 * @return true if the operation is successful, false otherwise
 ***/
public boolean dequeue();

}

Some observations about the interface:

• the interface is for queues of integers

• some operations are missing from the interface; the constructors, destructors, the test for equality and the copy operation. The reason is that these operations will depend on the specific implementation of the queue. They will be included in each specific implementation.
4 Implementation Using Arrays

The following implementation is for queue of integers. We consider the use of arrays for implementing the queue data structure. The intuition is to store the elements of the queue in consecutive locations in an array. The head index denotes the first element inserted, while the tail index denotes the last element inserted. This is illustrated in Figure 5.

![Figure 5: Queue stored in an array](image)

The intuition is to use a circular array, as depicted in Figure 6.

![Figure 6: Circular Array](image)

```java
/**
 * this is the implementation of the intQueue ADT based on
 * the use of arrays. It is an implementation of the
 * interface intQueueInterface.
 *
 * @author Enrico Pontelli
 ***/

public class intQueueArray implements intQueueInterface {
    private final int MAXSIZE=10; // max number of elements in the queue
    private int space[]; // array to store the elements of queue
    private int head; // position of the head of the queue
    private int tail; // position of the tail of the queue
```
private int size; // how many elements are in the queue

/*-------------------------------------------------------*/
/* OPERATIONS */
/*-------------------------------------------------------*/

/*** CreateQueue: create a new queue
***/
public Queue()
{
    space = new int[MAXSIZE]; // create the array for the queue
    size = 0; // initially no elements in the queue
    head = -1; // there is no head in the queue
    tail = -1; // there is not tail in the queue
}

/*** no destructor is needed ***/

/*** Inspector used to test whether the queue is empty or not
***/
public boolean emptyQueue ()
{
    return (size == 0);
}

/*** Inspector used to determine how many elements are present
   * inside the queue at this time
   * @return the size of the queue
***/
public int queueSize ()
{
    return (size);
}

/*** Inspector used to determine the element in the front position
   * of the queue
   * @return the front of the queue
***/

public int front ()
{
    if (size == 0)
    {
        System.out.println('"The queue is empty"');
        System.exit(0);
        return -1;
    }
    else
    return (space[head]);
}

/**
 * Inspector used to determine the element in the tail position
 * of the queue
 *
 * @return the tail of the queue
 ***/
public int rear ()
{
    if (size == 0)
    {
        System.out.println('"The queue is empty"');
        System.exit(0);
        return -1;
    }
    else
    return (space[tail]);
}

/**
 * Inspector used to compare the queue with another queue
 * and see if they have the same value
 *
 * @param q the queue we want to compare against
 * @return true if the two queues have the same content
 ***/
public boolean equalQueue(intQueueArray q)
{
    intQueueArray temp;

    if (q.queueSize() != size)
        return false;
    else
    {
        int j = head;

        // copy the input queue

temp = q.copyQueue();

for (int i=0; i < size; i++)
    if (space[j] != temp.front())
        return false;
    else
    {
        temp.dequeue();
        j = (j+1)%MAXSIZE;
    }
return true;
}

/**
* Inspector used to help printing the content of the queue on
* the screen
*
* @return a string representing the content of the queue
***/
public String toString()
{
    String s;
    s="Content: ";
    for (int i=head; i != tail; i=(i+1)%MAXSIZE)
    {
        s = s + " " + space[i];
    }
    s = s + " " + space[tail];
    return s;
}

/**
* Modifier used to insert a new element in the tail
* position of the queue
*
* @param element the element to be inserted in the queue
* @return true if the operation is successful, false otherwise
***/
public boolean enqueue (int element)
{
    if (size == MAXSIZE)
    {
        System.out.println("Cannot enqueue; queue is full");
        return false;
    }
    else
    {

tail = (tail + 1) % MAXSIZE;
space[tail] = element;
if (size == 0)
    head = tail;
size++;
return true;
}
}

/**
 * Modifier used to remove an element from the queue; the element
 * to be removed is in the front of the queue
 * @return true if the operation is successful, false otherwise
 ***/
public boolean dequeue ()
{
    if (size == 0)
    {
        System.out.println("Cannot dequeue; queue is empty");
        return false;
    }
    else
    {
        head = (head + 1) % MAXSIZE;
        size--;
        return true;
    }
}

/**
 * Modifier used to create a copy of the current queue;
 * the new queue is returned as result
 * @return a new queue with the same content as the current one
 ***/
public intQueueArray copyQueue ()
{
    intQueueArray n = new intQueueArray();
    int j = head;
    for (int i=0; i <size ; i++)
    {
        n.enqueue(space[j]);
        j = (j+1) % MAXSIZE;
    }
    return n;
}
QUESTION: what happen if during an Enqueue we reach a queue of size greater than MAXSIZE ??

4.1 Making the Implementation more Flexible

There are various limitations in the current implementation of queues.

4.1.1 Extensible Arrays

The first problem is related to the fact that the queue has a limit on the number of elements it can store (due to the use of a fixed size array). We have already seen in the case of stacks how to overcome this problem. The same type of modification can be done here. The following is the modified version of the enqueue operation which is capable of extending the array. We assume that the declaration of MAXSIZE has been modified to

```java
private int MAXSIZE=10;
```

```java
/**
 * Modifier used to insert a new element in the tail
 * position of the queue
 *
 * @param element the element to be inserted in the queue
 * @return true if the operation is successful, false otherwise
 ***/

public boolean enqueue (int element)
{
    if (size == MAXSIZE)
    {
        int count = 0;
        int[] temp = new int[MAXSIZE*2];

        for (int i = head; count < size; count++)
        {
            temp[count] = space[i];
            i = (i+1)% MAXSIZE;
        }
        head = 0;
        tail = size-1;
        MAXSIZE = MAXSIZE*2;
        space = temp;
    }

tail = (tail + 1) % MAXSIZE;
space[tail] = element;
if (size == 0)
    head = tail;
size++;
return true;
}
```

Let us now look at the other two limitations of the current approach: error handling and type limitation.
4.1.2 Exceptions

In the current implementation, error handling is present in two points. First of all, in the front and rear operations we terminate the execution with an error message if the operations are applied to an empty queue. On the other hand, the dequeue operation returns a value false to denote an error (dequeue an empty queue). We can try to make the treatment of the error uniform by using exceptions.

When you execute Java programs, it is possible that some errors may occur (e.g., divide by zero, exceed the boundaries of an array). These types of errors are called exceptions. If the program generates an exception, and no code is provided to handle the exception, then the program will abnormally terminate with an error message. However, occasionally you want your program to be able to handle the exception without terminating. Java provides a mechanism to handle exceptions in a program.

Java provides a set of predefined classes to deal with exceptions. The main class is called Exception. Java provides a number of subclasses which represent particular types of exceptions, e.g., ArrayStoreException, IndexOutOfBoundsException, FileNotFoundException. The class Exception is present in the package java.lang (which has to be imported if you want to use it). The class exception comes with two constructors, one with no arguments and one with a single argument of type String (describing the error message):

```java
public Exception(); // default constructor
public Exception(String str); // constructor with error message
```

Various methods provided by classes are capable of generating exceptions describing errors. For example, in the class String, the method charAt (used to access the character at a certain position in the string) can generate the exception called StringIndexOutOfBoundsException, when the value of the index is not a valid index in the string. For example, if you execute the following simple program:

```java
import java.lang.*;
import java.util.*;

public class sillyExceptionTest
{
    public static void main(String args[])
    {
        String a="abcdefg";
        int index = 10;

        System.out.println("GOOD "+a.charAt(index));
    }
}
```

You will get during the execution the message:

```
Exception in thread 'main' java.lang.StringIndexOutOfBoundsException:
    String index out of range: 10
```

which indicates that the named exception has been issued. Since no code has been provided to handle it, the program is terminated.

Two classes of exception exist: Checked and Unchecked. The unchecked exceptions (like the one in the example above) might occur and, if the programmer does not specify anything, then the Java system will automatically handle them (by stopping the program and printing some message on the screen). The checked exceptions instead cannot be handled by the Java system, thus, the programmer is required to add code to the program to handle them. The compiler will generate a syntax error if the programmer does not include the required code.

If you do not want to do anything special when a checked expression is generated, you need to simply add the statement:

```java
throws ⟨Exception name⟩
```

next to the method which might cause the exception. For example, if the main method performs some operations that might cause the exception IOException, then we can write:

```java
throws IOException
```
public static void main(String args[]) throws IOException
{
   .... code ...
}

In this case, if an IOException occurs, the main will simply pass it on to the Java system (which will stop with an error message). Note that the declaration of throws is required only for exceptions that are of type checked.

As another example, let us assume that the method xyz performs an I/O operation which might cause an IOException, and the method xyz is directly called by the method main. If we do not want to do anything special with IOException apart from terminating the program, then we need to write:

// method xyz

public static void xyz(..) throws IOException
{
   .... code ...
}

public static void main(String args[]) throws IOException
{
   .... code ....
}

Note that both xyz and main need to include the statement throws. The idea is that if an exception occurs (and it will occur inside xyz), we first need to tell what to do inside xyz; the throws declaration states that we simply want to terminate xyz. But that’s not enough: once xyz is terminated, the control goes back to the main (which originally called xyz) and also here we need to state what to do with the exception. In this case, we are adding another throws declaration, which means that also main will be stopped.

Now, what if we want to do something else when the exception occurs? I.e., instead of terminating the method, we might want to execute some particular code. This can be done by using the try/catch statement of Java. The structure of the statement is:

try
{
   ... statements ...
}
catch ( EXCEPTIONType EXCEPTIONName )
{
   .... handler ....
}

The first set of statements are the ones that might cause the exception; the handler is another set of statements that will be executed if the exception occurs during the statements.

If different types of exceptions can be generated by the statements, then we can have multiple catch:

try
{
   ... statements ...
}
catch (ExceptionType1 ExceptionName1)
{

Let us consider the following simple example that shows how to handle divisions by zero:

```java
public static void main(String args[]) {
    int a, b, c;
    System.out.print(''Insert First number: ‘‘);
    a = SavitchIn.readInt();
    System.out.print(''Insert Second number: ‘‘);
    b = SavitchIn.readInt();

    try {
        c = a/b;
        System.out.println(''The Result is ‘’+c);
    } catch (ArithmeticException aex) {
        System.out.println(''You silly, you typed zero as denominator’’);
    } System.out.println(''END OF THE PROGRAM’’);
}
```

When an exception occurs, the first try block that has a catch clause for that exception is located and the code for the catch executed. The execution will then continue with the instructions that follow the try statement. For example, in the example above the final print message is executed no matter whether the exception has occurred or not.

To summarize the behavior of the try/catch:

- if no exception is generated inside the try block, all catch blocks associated with the try are ignored and program execution resumes with the statement that follows the last catch.
- if an exception is generated in a statement inside a try block, the remaining statements are ignored. The program searches the catch blocks associated to the try to find one that specifies the same type of exception as the one generated. If such catch block is found, then the statements in the catch block are executed and the execution continues with what follows.

Remember that the different types of exceptions are described by different classes. All the exception classes are subclasses of the class `Exception`. Thus, if we write something like:

```java
try {
    .... statements ....
} catch (Exception e) {
    ... handler ...
}
```
Then we have that *any* exception generated by the statements will be handled by the catch block, since any exception type is a subclass of Exception.

Let us view another example.

```java
public class ExceptionExample
{
    public static void main(String args[])
    {
        int dividend, divisor, quotient;

        try
        {
            System.out.print(''Enter dividend: '');
            dividend = SavitchIn.readInt();
            System.out.print(''Enter divisor: '');
            divisor = SavitchIn.readInt();
            quotient = dividend / divisor;
            System.out.println(dividend + '' / ''+divisor+''=''+quotient);
        }
        catch (ArithmeticException ae)
        {
            System.out.println(''EXCEPTION: arithmetic exception has occurred'');
        }
        catch (NumberFormatException nf)
        {
            System.out.println(''EXCEPTION: number format exception has occurred'');
        }
    }
}
```

If the given inputs are 45 and 2 then the program will successfully print 22. If the inputs are 18 and 0 then an arithmetic exception is reported. If the inputs are 75 and *fg* then a number format exception is reported.

It is also possible for the program to explicitly generate an exception, using the `throw` statement. It takes as argument an object which is an instance of an exception class. For example, if we want to generate an arithmetic exception if the value of the variable *x* is greater than 100, we can write the code:

```java
if (x > 100)
    throw(new ArithmeticException());
```

which creates an object of type ArithmeticException and generates the exception.

Finally, one can also create new types of exceptions by creating new subclasses of existing exceptions. For example, we can create a class

```java
public class mySillyException extends RuntimeException
{
    mySillyException()
    {
        super(''THIS IS MY SILLY EXCEPTION'');
    }
}
```

We can then define programs that generate such exception:

```java
public static void main(String args[])
```
The class `RuntimeException` contains unchecked exceptions (which can be automatically handled by Java if we do not provide a handler). In the example, if `x` is greater than 100 then the program will stop and report the exception (printing the message we provided).

Going back to our queues example, we can create a new type of exception:

```java
public class emptyQueueException extends RuntimeException {
   emptyQueueException() {
      super('Trying to access an empty queue');
   }
}
```

Then, we can replace the error mechanisms of the queue implementation with operations that generate this new exception. For example, the code for the `front` operation can be rewritten as:

```java
/***
 * Inspector used to determine the element in the front position
 * of the queue
 * @return the front of the queue
 ***/
public int front () {
   if (size == 0)
      throw new emptyQueueException();
   else
      return (space[head]);
}
```

5 Queue Applications

Typical kind of applications:

5.1 Buffer

Two system units communicating. E.g., an HTTP server which receives requests from undetermined clients.

```java
public class Simulation {
   intQueueArray Buffer;

   public static void Server()
   {
```
```java
int request;

for (int i = 0 ; i < 10 ; i++)
{
    request = Buffer.front();
    Buffer.dequeue();
    ServeRequest(request);
}

public static void client()
{
    int req;
    req = GenerateRequest();
    Buffer.enqueue(req);
}

public static void main(String args[])
{
    // create the buffer
    Buffer = new intQueueArray();

    // execute the client
    client();
    // execute the server
    server();
}
```