ECS401: Procedural Programming

Paul Curzon

Unit 7: ADTs and Methods

Abstract Data Types

Big programs, big problems

Writing small simple programs is easy! (with practice)

The real problem is BIG programs.
millions of lines long

If you are trying to write a big program:

- how can you get it to work in the first place?
- how can you write it so that anybody can understand it?
- how can you write it so that somebody else can change it?

(NB Software lasts **much** longer than hardware)

Solutions

Solution1

- break the program into methods
  - each method is a small “program” so easy

Solution 2

- define data structures as abstract data types

BOTH are about hiding the implementation and working instead with clean interfaces

Abstract Data Type

An Abstract data type is a model for a data type where the **actual details** of how the data type is really implemented are **hidden**.

It is a way of structuring a program.

- The data type is defined via how the programmer uses it via
- **operations** that can be applied to it, and
- **values** that are visible.

Queues to illustrate the idea

Any data structure can be implemented as an abstract data type (ADT).

We will use **queues** as one example to illustrate what we mean by an ADT.

See the notebook and the booklets in the reading section for a variety of other examples.

Our code needs a Queue

To create an ADT we define a set of primitive operations.
What defines something as a queue?

We can **create** a new (empty) queue

Things can **join** the queue
(at the ‘back’)

Things **leave** the queue
but only in the order they joined it

We can **look** at what is at the front of the queue

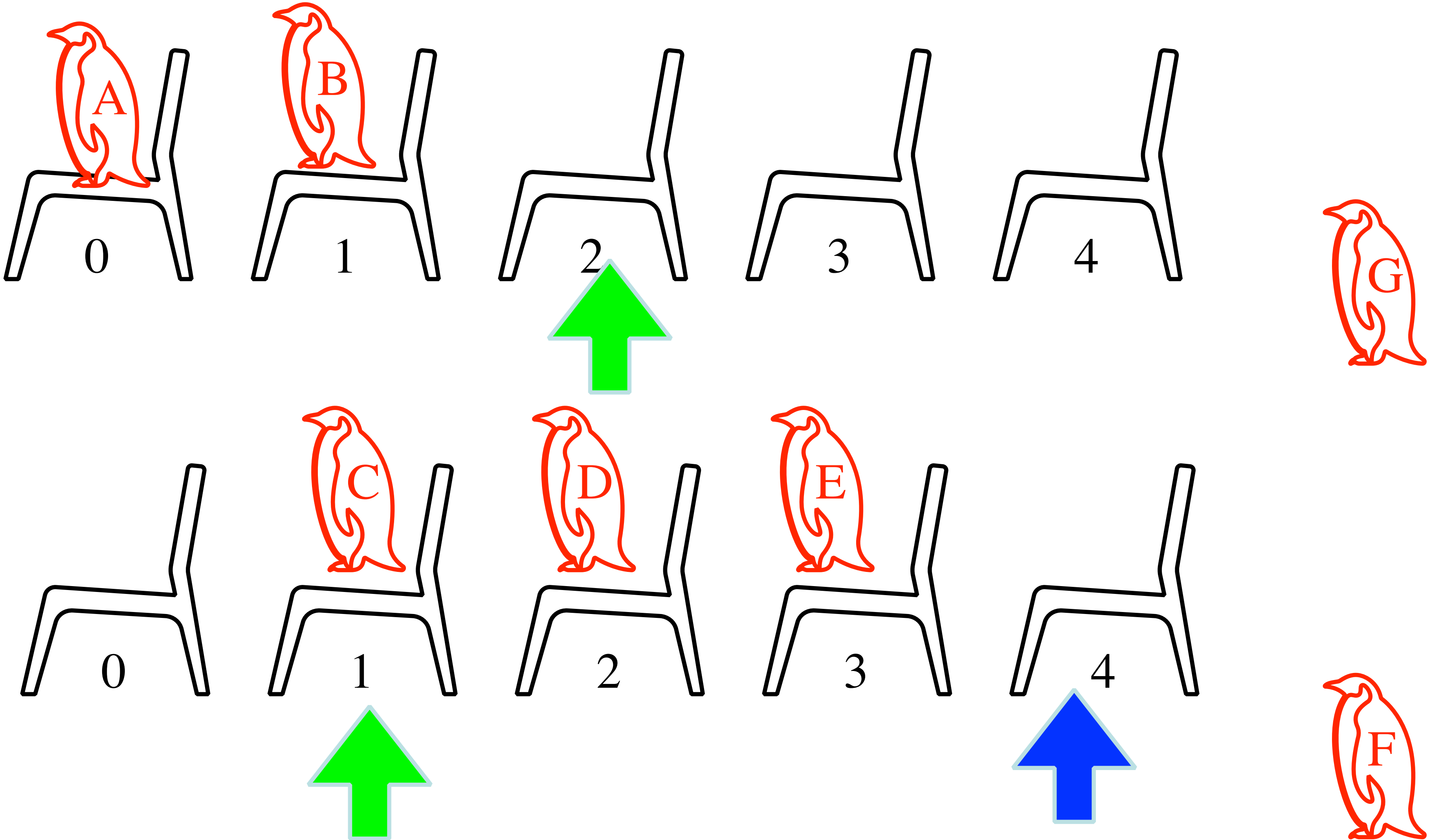
We can check if a queue is **empty?**

We don’t want to worry how it is implemented

- How? Write methods for the operations

```
Queue q = createQueue(5);
q = joinQueue(q,"Alistair Brownlee");
q = joinQueue(q,"Mo Farah");
q = joinQueue(q,"Laura Trott");
q = joinQueue(q,"Nicola Adams");
q = joinQueue(q,"Amir Khan");
System.out.println(firstInQueue(q));
q = leaveQueue(q);
System.out.println(firstInQueue(q));
q = leaveQueue(q);
System.out.println(firstInQueue(q));
q = joinQueue(q,”Tanni Grey-Thompson“);
```

2 ways to organise a Queue



It shouldn't matter which we implement…
the operations should behave the same

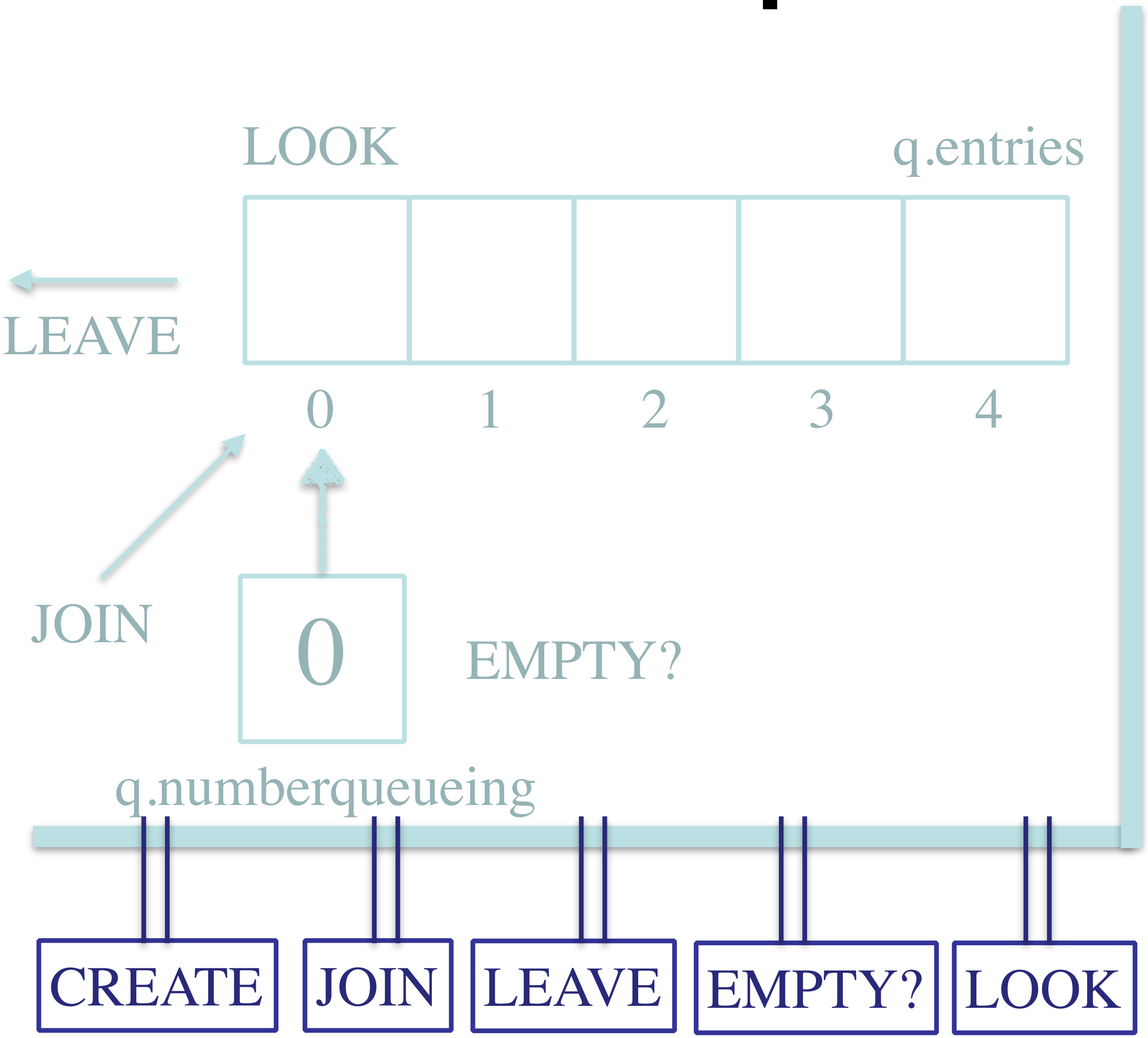
Implementation 1
First create a new class

```
class Queue
{
   String [] entries;
   int numberqueueing;
}
```

Here are some example calls

```
Queue q = createQueue(5);
q = joinQueue(q,"Mo");
q = joinQueue(q,"Laura");
q = joinQueue(q,"Nicola");
q = joinQueue(q,"Amir");
System.out.println(firstInQueue(q));
q = leaveQueue(q);
System.out.println(firstInQueue(q));
q = leaveQueue(q);
System.out.println(firstInQueue(q));
q = joinQueue(q,”Tanni“);
```

A Queue Implementation



LOOK

q.entries

LEAVE

0 1 2 3 4

JOIN

0

EMPTY?

q.numberqueueing

CREATE

JOIN

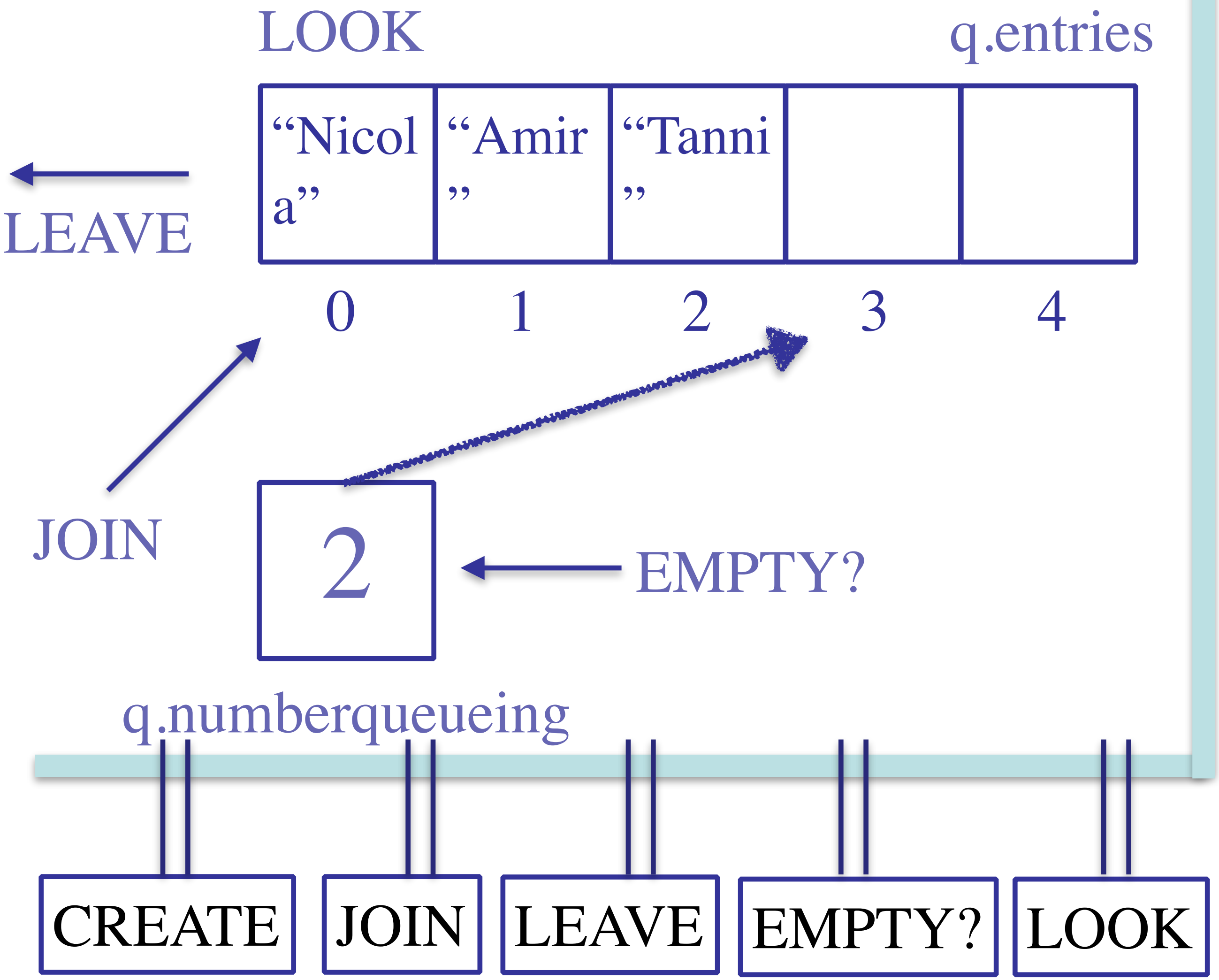
LEAVE

EMPTY?

LOOK

```
CREATE 5
JOIN Mo
JOIN Laura
JOIN Nicola
JOIN Amir
LOOK
LEAVE
LOOK
LEAVE
JOIN Tanni
```

A Queue Implementation



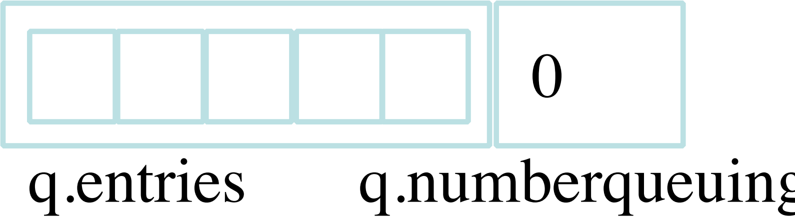
```
CREATE 5
JOIN Mo
JOIN Laura
JOIN Nicola
JOIN Amir
LOOK
LEAVE
LOOK
LEAVE
Join Tanni
```

Implementation 1
Create a new empty a queue

```
public static Queue createQueue(int size)
{
       Queue q = new Queue ();
       String [] a = new String[size];

       q.entries = a;
       q.numberqueueing = 0;

       return q;
}
```



Exercise: complete the code
Is a queue empty

```
public static boolean isQueueEmpty(Queue q)
{
  if (q.numberqueueing == 0)
       return  true;
  else
       return false;
}
```

Implementation 1
Is a queue empty (another way)

```
public static boolean isQueueEmpty(Queue q)
{
  boolean queueisempty = (q.numberqueueing == 0);

  return queueisempty;
}
```

Implementation 1
Is a queue empty (another way)

```
public static boolean isQueueEmpty(Queue q)
{
  return (q.numberqueueing == 0);
}
```

Implementation 1
Join a queue

```
public static Queue joinQueue
                    (Queue q, String newentry)
{
   if (q.numberqueueing < q.entries.length)
   {
      q.entries[q.numberqueueing] = newentry;
      q.numberqueueing = q.numberqueueing + 1;
   }

   return q;
}
```

Implementation 1
Look at the front of the queue

```
public static String firstInQueue(Queue q)
{
   if (isQueueEmpty(q))
   {
      return "Queue Empty";
   }
   else
   {
      String firstentry = q.entries[0];
      return firstentry;
   }
}
```

Implementation 1
Look at the front of the queue (another way)

```
public static String firstInQueue(Queue q)
{
   if (isQueueEmpty(q))
   {
      return "Queue Empty";
   }
   else
   {
     return q.entries[0];
   }
}
```

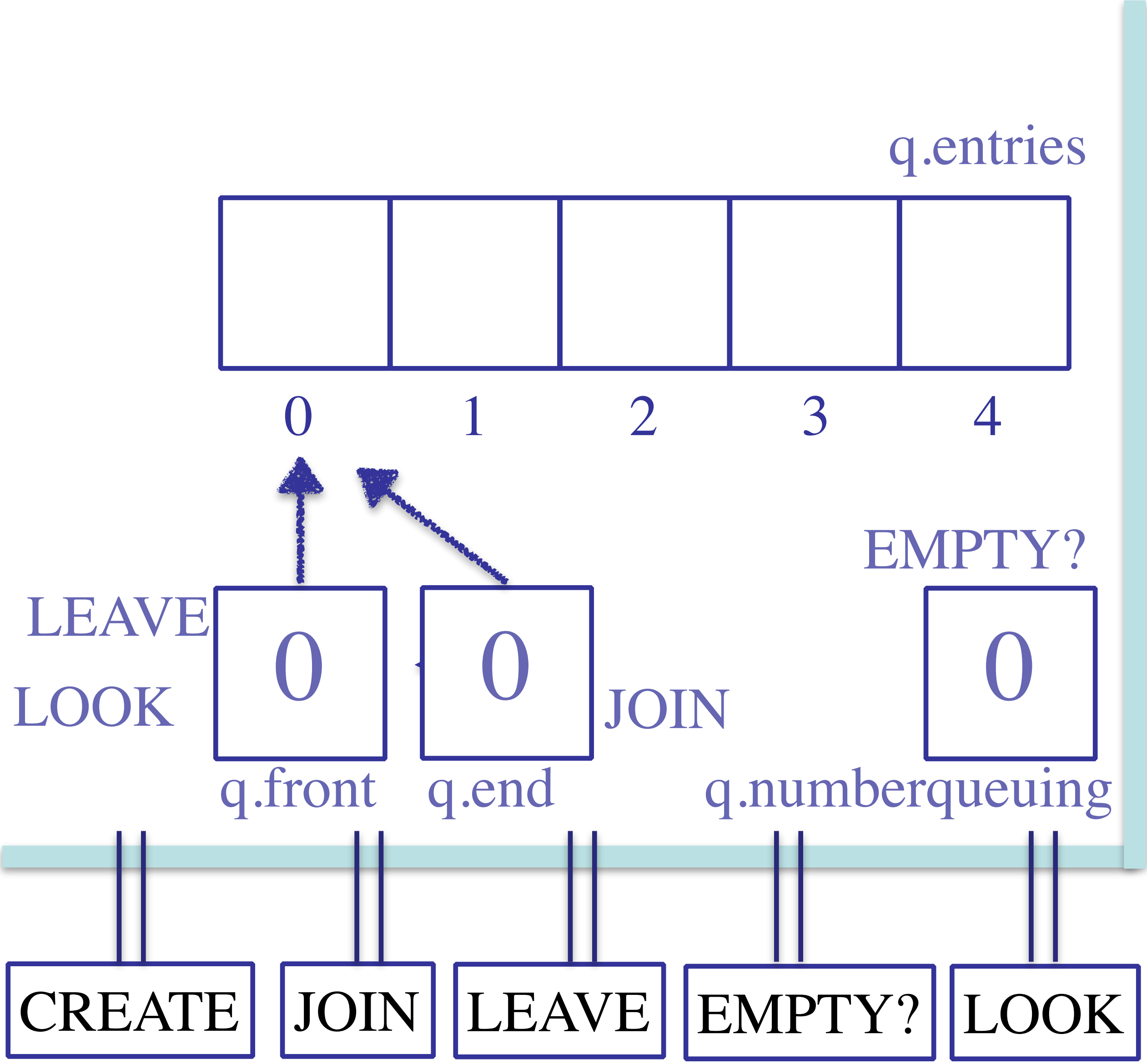
Implementation 1
leave a queue (from front)

```
public static Queue leaveQueue(Queue q)
{
  if (!(isQueueEmpty(q))) // queue not empty
  {
    for (int i = 0; i < q.numberqueueing; i++)
    {
      q.entries[i] = q.entries[i + 1];
            //Shuffle all entries down
    }
    q.numberqueueing = q.numberqueueing - 1;
  }
  return q;
}
```

A different implementation of the Queue ADT

We can implement it in a completely different way …without changing the code that uses the Queue methods

A different Queue Implementation



```
CREATE 5
JOIN Mo
JOIN Laura
JOIN Nicola
JOIN Amir
LOOK
LEAVE
LOOK
LEAVE
Join Tanni
```

Implementation 2
First create a new type

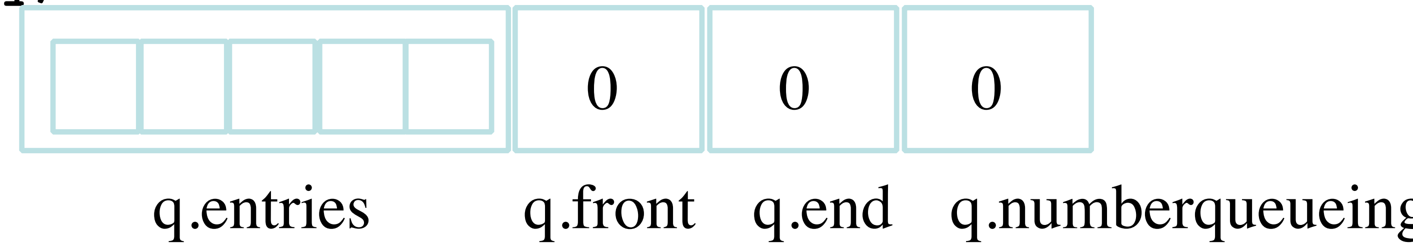
```
class Queue
{
   String [] entries;
   int front;
   int end;
   int numberqueuing;
}
```

Implementation 2
Create a new empty a queue

```
public static Queue createQueue(int size)
{
  Queue q = new Queue ();
  String [] a = new String[size];

  q.entries = a;
  q.front = 0;
  q.end = 0;
  q.numberqueuing = 0

  return q;
}
```



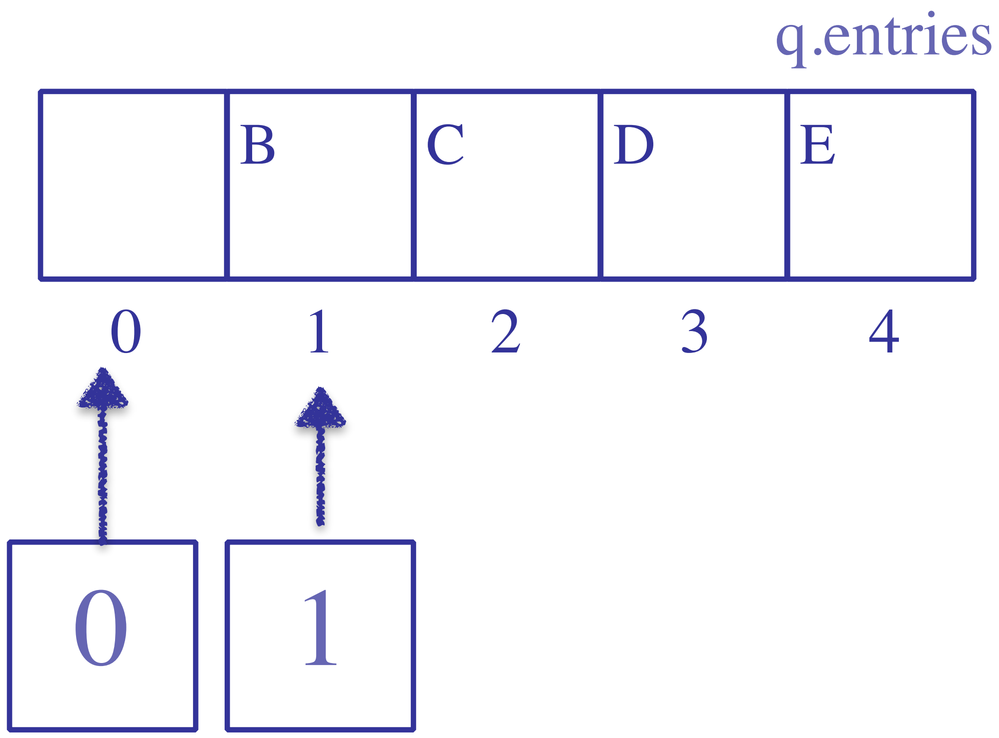
Implementation 2
Join a queue

```
public static Queue joinQueue
                    (Queue q, String newentry)
{
   if (q.numberqueueing < q.entries.length)
   {
      q.entries[q.end] = newentry;
      q.end = (q.end + 1) % q.entries.length;
      q.numberqueuing = q.numberqueuing + 1;
   }

   return q;
}
```

Modulus (%)

Just means the remainder after dividing



q.end = (q.end + 1) % 5

0 %5 =	0
1 %5 =	1
2 % 5=	2
3 % 5=	3
4 %5=	4
5 % 5=	0
6 % 5=	1
7 % 5=	2
8 %5=	3
9 %5=	4
10%5=	0

It is a way to make numbers wrap back round to 0 like clock arithmetic as you keep adding one

Implementation 2
leave a queue (from front)

```
public static Queue leaveQueue(Queue q)
{
  if (!(emptyQueue(0))) // queue not empty
  {
    q.front = (q.front + 1) % q.entries.length;
    q.numberqueueing = q.numberqueueing - 1;
  }
  return q;
}
```

Implementation 2
Look at the front of the queue

```
public static String firstInQueue(Queue q)
{
  if (q.numberqueueing == 0)
  {
    return "Queue Empty";
  }
  else
  {
    String firstentry = q.entries[q.front];
    return firstentry;
  }
}
```