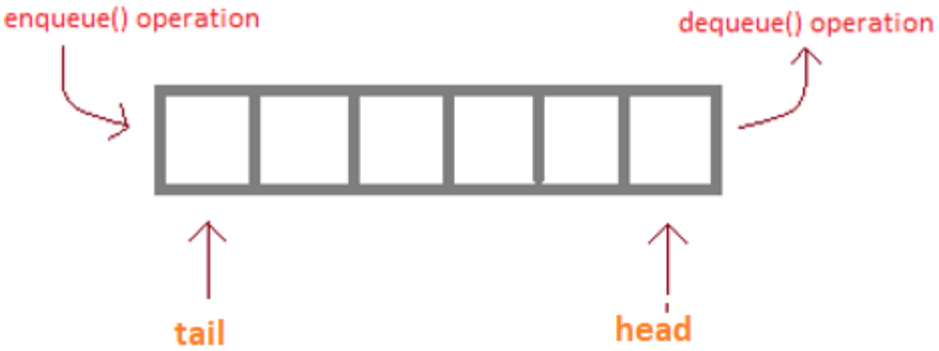
Stacks and Queues

2.5.1 Implementing a queue using an array

- A *queue* is a waiting line or a linear list in which items are added at one end and deleted from the other end

- Familiar examples are queues at a bank, a supermarket, etc. People are supposed to join the queue at the tail and exit from the head.



Applications of Queue Data Structure

Queue, as the name suggests is used whenever we need to manage any group of objects in an order in which the first one coming in, also gets out first (**First In First Out** or simply **FIFO**) while the others wait for their turn, like in the following scenarios:

- Serving requests on shared resources like in a printing center where computers are connected to a set of printers. What happens when a user issues a print order?
- On a webserver: it is busy working with previous requests and additional ones arrive. Where are the new **waiting** requests kept?
- In real life scenario, call center phone systems uses Queues to **hold** people calling them in an order, until a service representative is free.

Introduction

- These are the basic operations we want to perform on a queue:
  - Add an item to the queue (we say ***enqueue***)
  - Take an item off the queue (we say ***dequeue***)
  - Check whether the queue is empty
  - Inspect the item at the head of the queue
- Like with stacks, we can easily implement the queue data structure using arrays or linked lists. We will use a queue of integers for illustration purposes.

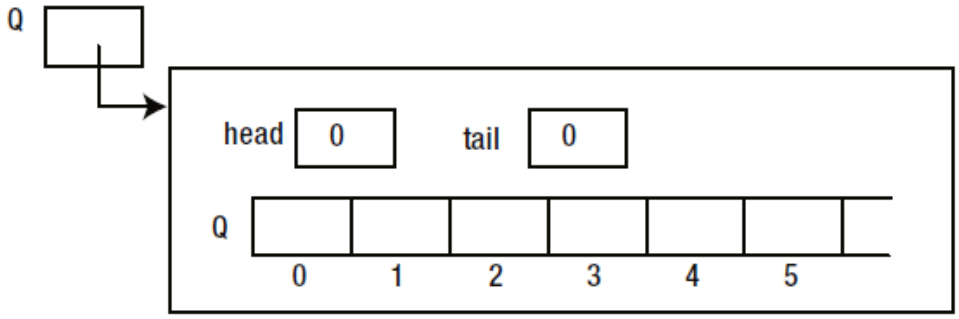
Implementing a Queue Using an Array

- We use:
  - An integer array (**queueArray**) for storing the numbers
  - An integer variable **head** that indicates the item at the head of the queue
  - An integer variable **tail** that indicates the item at the tail of the queue

As usual, we can create an empty queue, Q, with this:

```
Queue Q = new Queue();
```

When this statement is executed, the situation in memory can be represented as shown in Figure

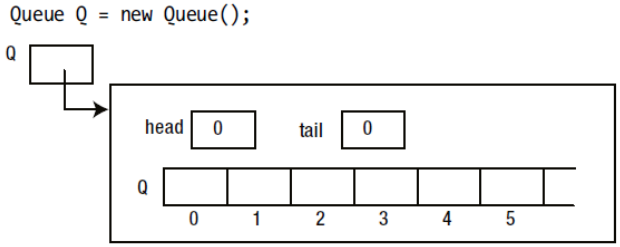


Data Fields and Constructors

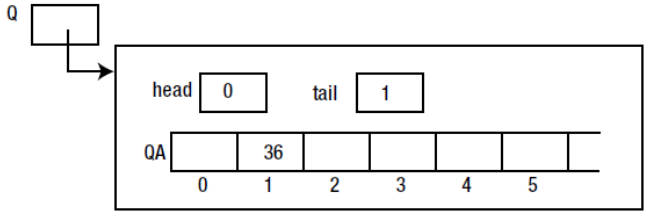
```
public class Queue {

    private int[] queueArray;
    private int head, tail;

    // A no-arg constructor to create an empty queue of a size 100
    public Queue() {
        queueArray = new int[100];
        head = tail = 0; // initially both head and tail are at cell 0
    }

    // A constructor to create an empty queue of a specified size
    public Queue(int size) {
        queueArray = new int[size];
        head = tail = 0; // initially both head and tail are at cell 0
    }
```

Basic operation enqueue (add)

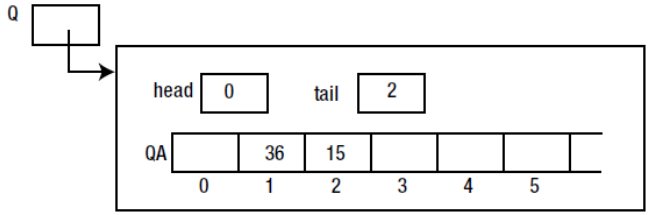
- Initially



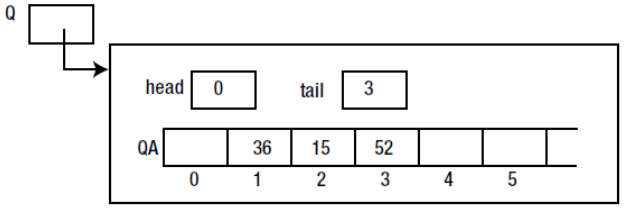
- For example, to add 36, say, to the queue, we increment tail to 1 and store 36 in QA[1]; head remains at 0.



- If we then add 15 to the queue, it will be stored in QA[2] and tail will be 2.

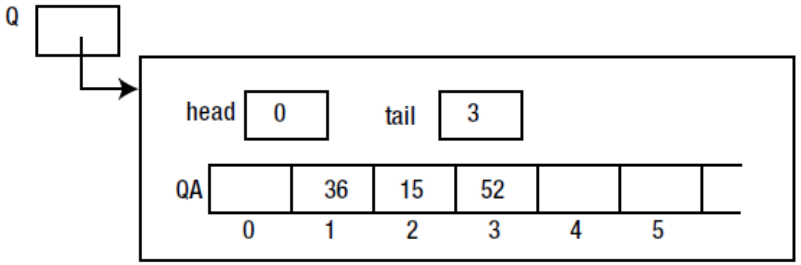


- If we now add 52 to the queue, it will be stored in QA[3] and tail will be 3.

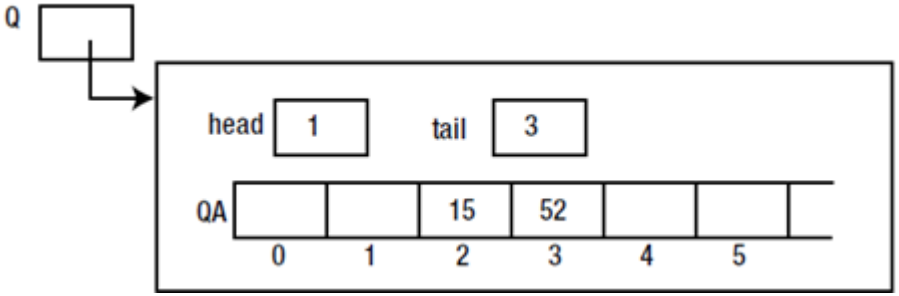


Basic operation dequeue (remove)

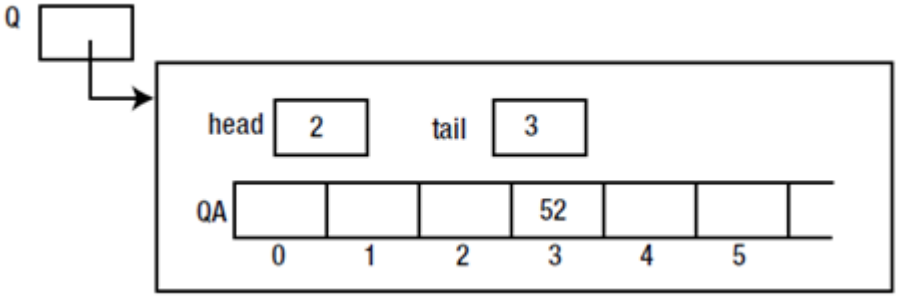
- To remove from the queue , we must *first* increment head and then return the value pointed to by head.



- if we remove 36, head will become 1, and it points “just in front of” 15

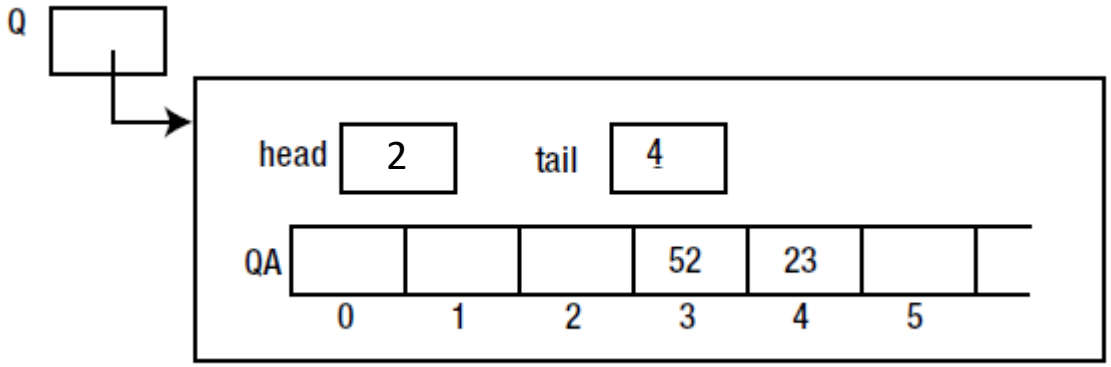


- if we remove 15, head will become 2, and it points “just in front of” 52



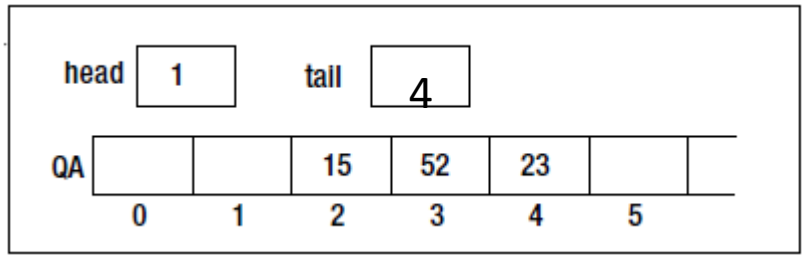
Basic operation enqueue()

- Suppose we now add 23 to the queue. It will be placed in location 4 with tail being 4 and head being 2.

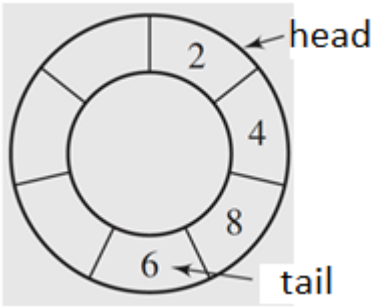


Basic operation enqueue()

- Suppose we now have this queue



- Consider what happens if we continuously add items to the queue without taking off any. The value of tail will keep increasing until it reaches queueArray.length-1.
- What do we do if another item needs to be added?
  - We *could* say that the queue is full and stop the program! However, there are two free locations, 0 and 1!
- It would be better to try to use one of these. This leads us to the idea of using the array in a *circular way.*



Array with circular indexing

- We have always associated `i++` or `i = i + 1` with `i < a.length` to make sure that `a[i]` does not cause an `IndexOutOfBoundsException`.
- The circular array is an array where there is no notion for first and last element. So:
  - We can't say that a[0] is the first element and a[a.length-1] is the last.
  - Instead:
    - The element before a[0] is a[a.length -1]
    - And the element after a[a.length-1] is a[0].

Array with circular indexing

- How can we implement a circular array behaviour?
  - We no longer use `i == a.length` to determine that we reached the end of the array
  - We use the remainder operation % with every index increment:

Example: int[] a = {12, 14, 16, 18};

int i = 3;

a[i] is 18

i = (i + 1) will give 4 a[4] is out of bounds

increment i in a way that from a[3] we go to a[0]

using a straight forward mathematical formula (no if and else)

i = (i + 1) % a.length;

this instruction will make the result 4 % 4 which is 0.

Code of isFull()

// A method that checks if the queue is full

```
public boolean isFull() {
   return (tail + 1) % queueArray.length == head;
            // only one empty cell
}
```

- We keep one empty cell so that head and tail do not meet except when the queue is empty.
- The queue is full if tail comes right before head which means if there is only empty cell .

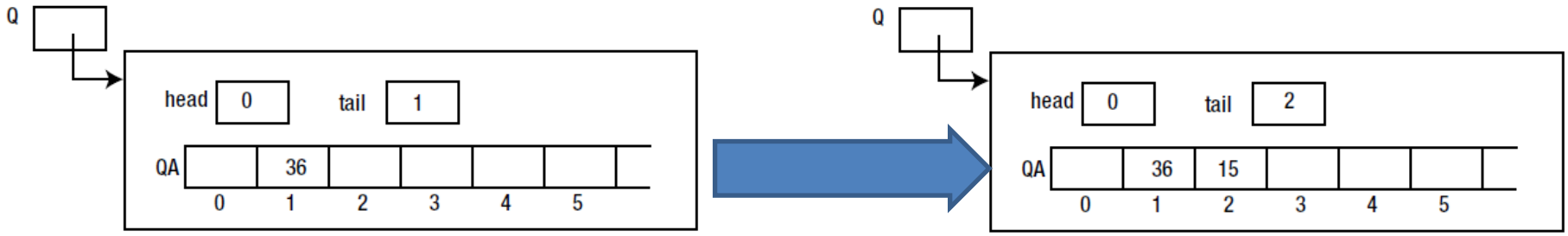
Code of enqueue()

```
// A method to add an element at the tail of the queue

public void enqueue(int item) {
  if (!isFull())  {
    tail = (tail + 1) % queueArray.length;
    queueArray[tail] = item;
  }
  else
    System.out.println("The queue is full.");
}// end enqueue
```

Compute index of next insertion (index of tail)

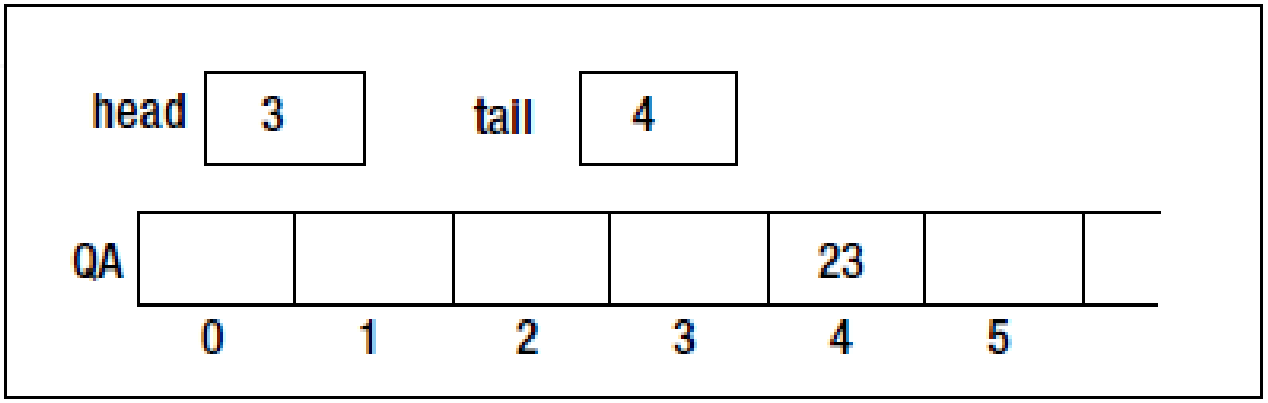
Insert the new element

Full queue: Unable to enqueue



Basic operation dequeue()

- Now, head has the value 3, tail has the value 4, and there is one item, 23, in the queue at location 4.
- If we delete this last item, head and tail would both have the value 4 and the queue would be empty.
- This suggests that we have an *empty* queue when head has the same value as tail.



Code of isEmpty()

```
// A method that checks if the queue is empty
public boolean isEmpty() {
    return tail == head;
}
```

- the queue will be empty whenever head and tail have the same value. This value will not be necessarily be 0. In fact, it may be any of the values from 0 to queueArray.length-1.

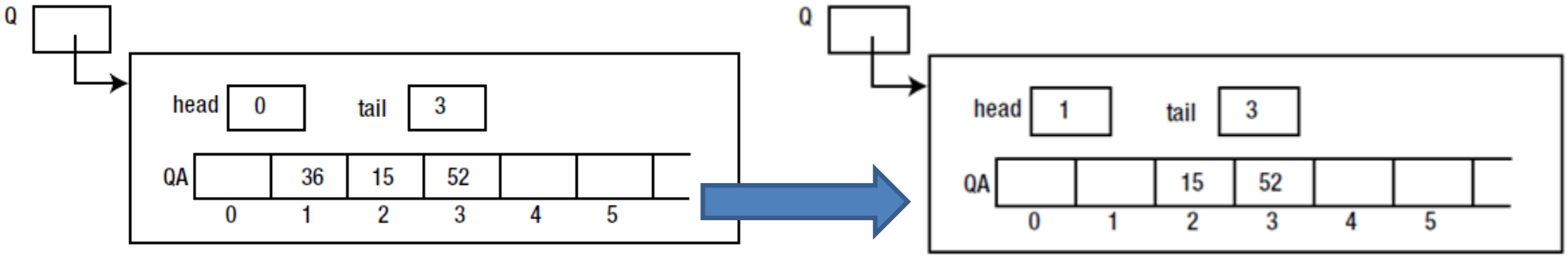
Code of dequeue()

```
// A method to remove and return the element at the head
of the queue
public int dequeue() {
  if (isEmpty())  {
    System.out.println("The queue is empty.");
    System.exit(1);
  }
  else
    head = (head + 1) % queueArray.length;
  return queueArray[head];
}// end dequeue
```

Empty queue: Unable to dequeue

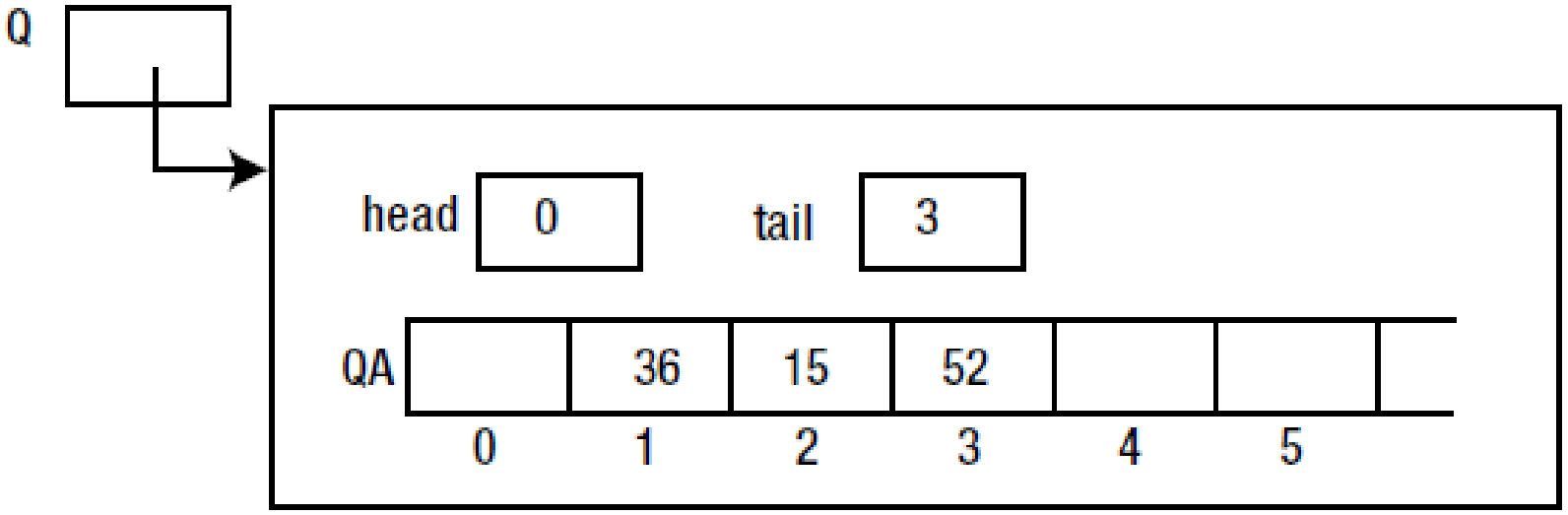
Update the head index

Return the element at old head index



Basic operation head()

- This method returns the head element without removing it.
- In the example below the method will return 36. Remember the head element is stored at index head + 1.



Code of head()

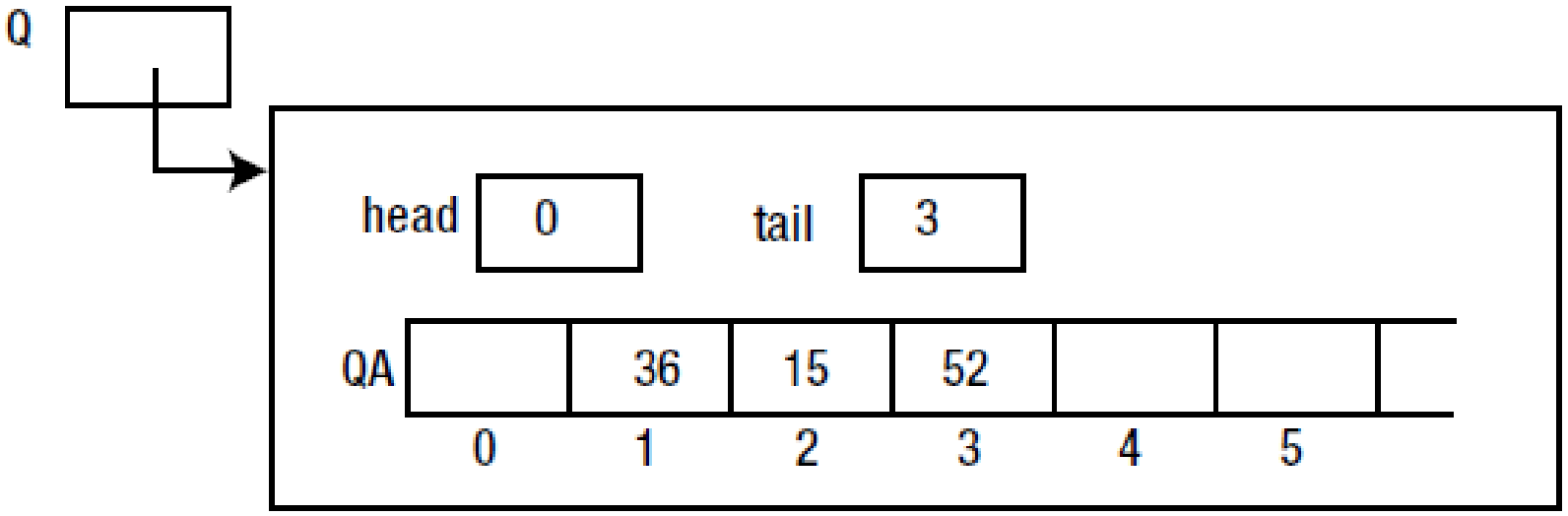
```
// A method that returns the element at the head of the queue
   public int head() {
       if (isEmpty()) {
           System.out.println("The queue is empty.");
           System.exit(1);
       }
       return queueArray[(head + 1) % queueArray.length];
   } // end head
```

Empty queue: Unable to return

Return the head element

Basic operation tail()

- This method returns the tail element without removing it.
- In the example below the method will return 52. Remember the tail element is stored at index tail.



Code of tail()

```
// A method to add an element at the tail of the queue


 public int tail() {
   if (isEmpty()) {
      System.out.println("The queue is empty.");
      System.exit(1);
   }
   else
     return queueArray[tail];
 }// end tail
```

Full queue: Unable to return

return the tail element;

Example Application

```
// An application to simulate a waiting line in a bank
public class Application {
  public static void main(String[] args) {

    for (int i = 1; i <= 5; i++)
    {
       q.enqueue(i);
       System.out.print("Customer " + q.tail());
       System.out.println(" is entering the line now.");
    }
    while(!q.isEmpty())
    {
       System.out.print("Customer " + q.head());
       System.out.println(" is leaving the line now.");
       q.dequeue();
    }

  }// end main
}
```

Code of Queue (Array Implementation)

Download the code of the Queue ADT posted to your CSCI378 Google Classroom along with this presentation. Run the application and test it.