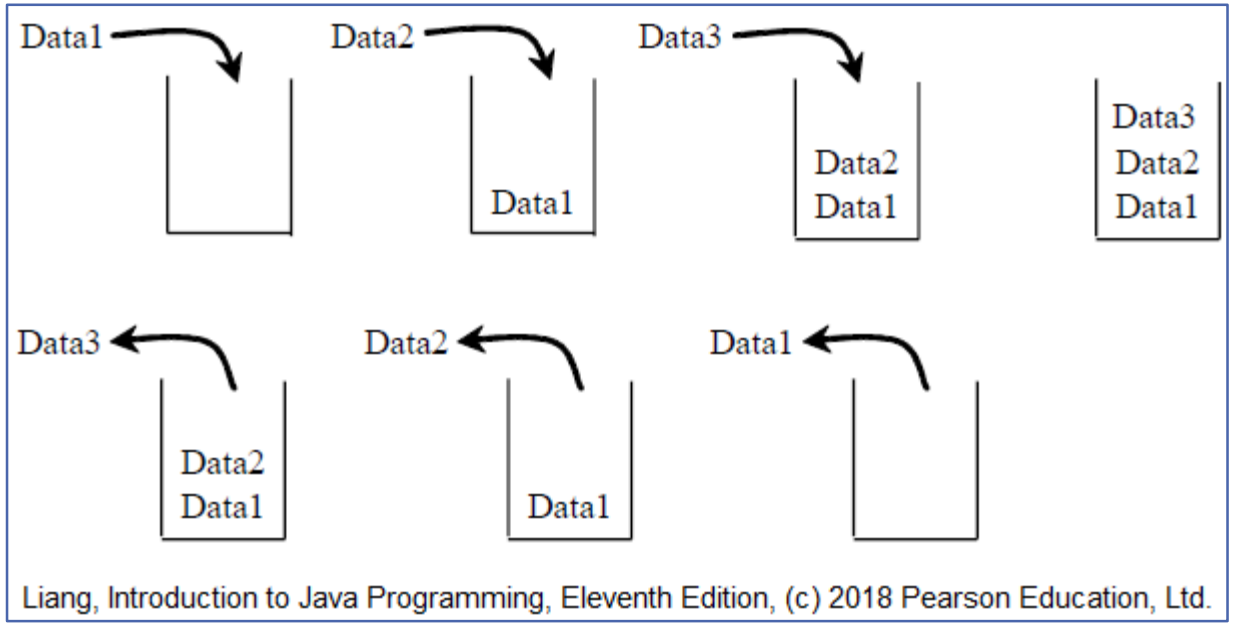
COMP 201 Data Structures and Algorithms
Lab7 – Array-Based Stack and Queue Implementations

A Generic Resizable Array Based Stack Implementation

An overview of the stack data structure and explanations for the given codes MyStack.java and MyStackDemo.java

Stack Data Structure

- A stack is a data structure based on the last-in, first-out (LIFO) principle.
- Elements are accessed, added and removed only from the top of the stack as illustrated on the right.



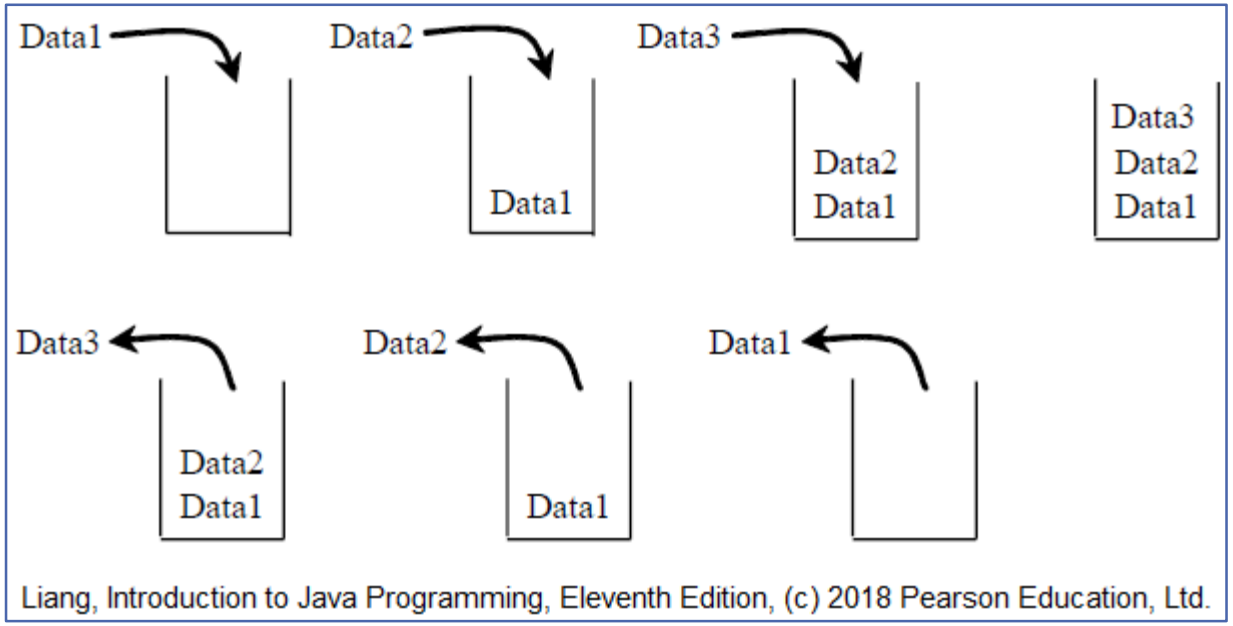
Liang, Introduction to Java Programming, Eleventh Edition, (c) 2018 Pearson Education, Ltd.

Real-Life Examples:

- A stack of books in a box
- A stack of boxes in a warehouse
- A stack of neatly folded t-shirts
- A stack of plates in a cupboard
- ...

Stack Data Structure

- A stack is a data structure based on the last-in, first-out (LIFO) principle.
- Elements are accessed, added and removed only from the top of the stack as illustrated on the right.
- Here you can find an animation that shows how the stack data structure works.
- Below is the UML diagram of the MyStack<E> class given with these lab description slides (MyStack.java).
  - It is a generic class that implements an array-based stack.

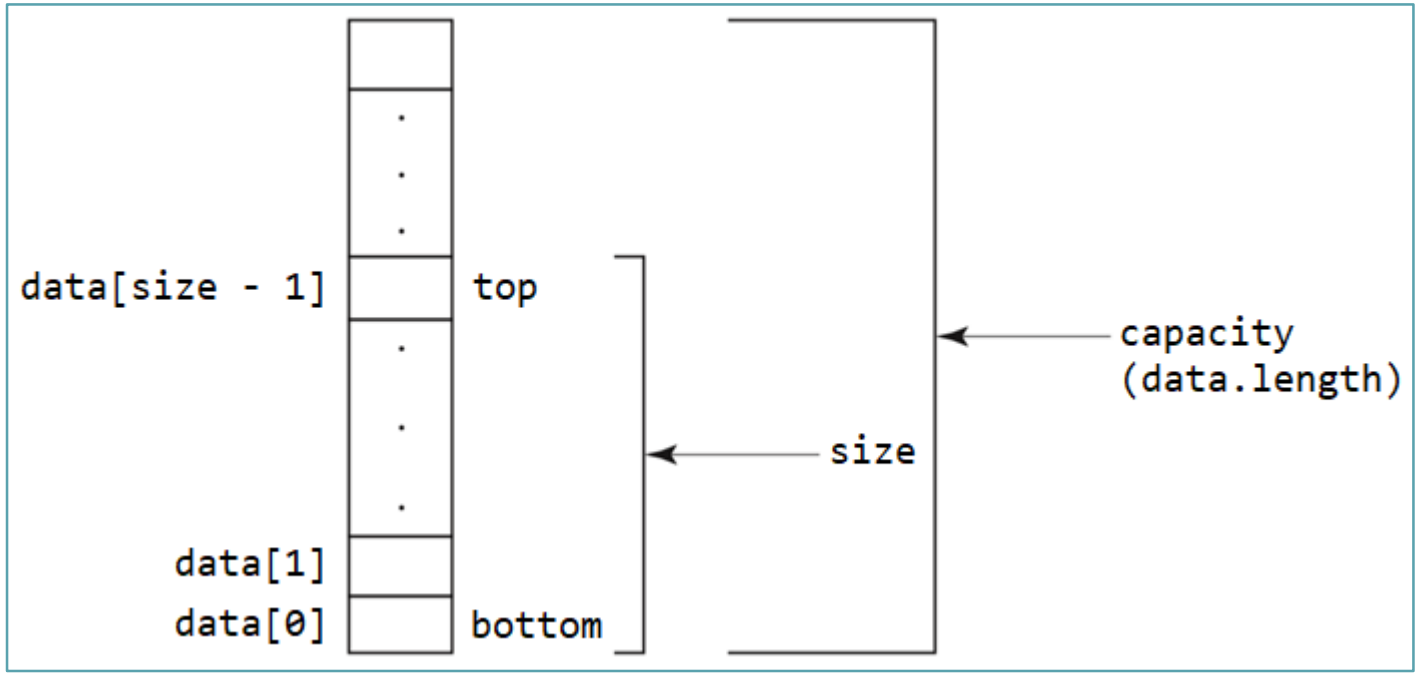


Liang, Introduction to Java Programming, Eleventh Edition, (c) 2018 Pearson Education, Ltd.

MyStack<E>	
-data: E[]	A generic array for storing the elements in the stack.
-size: int	The number of the stored elements (initially 0).
-DEFAULT_CAPACITY: int	The default capacity for the array (a constant equal to 5).
+MyStack()	Creates an empty stack with the default initial capacity.
+MyStack(capacity: int)	Creates an empty stack with the given initial capacity.
+isEmpty(): boolean	Returns true if the stack is empty and false otherwise.
+peek(): E	Returns the top element in the stack without removing it (returns null if the stack is empty).
+pop(): E	Removes and returns the top element in the stack (returns null if the stack is empty).
+push(e: E): void	Adds the given element e to the top of the stack.
-resize(capacity: int): void	Resizes the data array to a given capacity. It is invoked before adding or after removing an element (if necessary).
+toString(): String	Returns the stack contents as a string.

MyStack: A Generic Resizable Array Based Stack Implementation

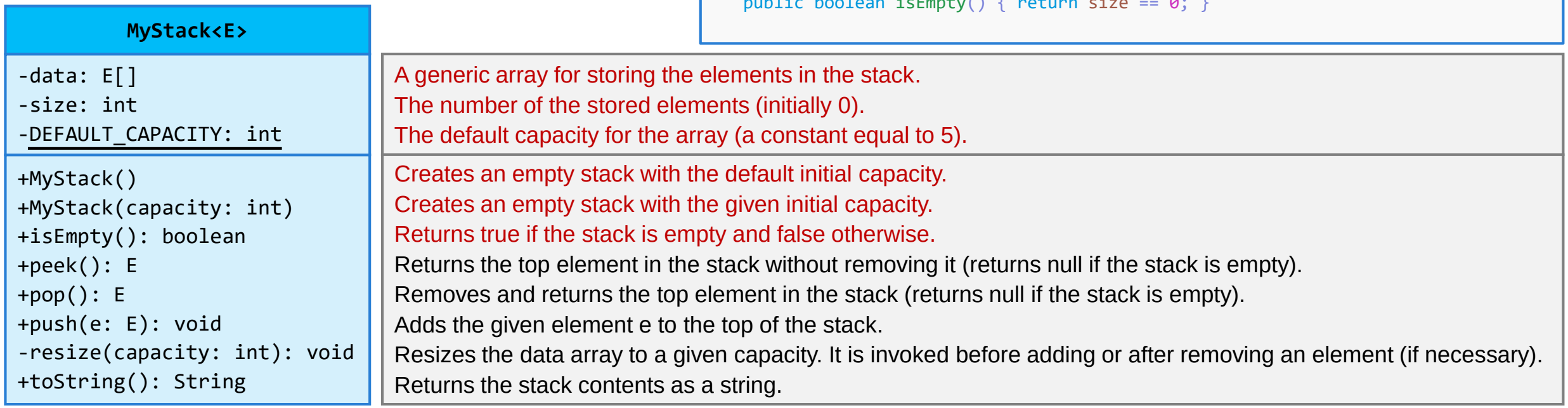
- The elements in the stack are stored in an array named as data as shown below.
- The variable size is used to keep track of the number of the stored elements in the stack and size - 1 is the index of the element at the top of the stack. For an empty stack, size is 0.
  - This way there is no need to shift the stored elements for push or pop operations.



- The data array is **resized** as follows to provide enough space for adding new elements and to ensure an efficient usage of the memory (also considering the cost of the resize operation on the running time).
  - **Before the Push Operation:** If the data array is full before pushing an element to the stack, its capacity is doubled.
  - **After the Pop Operation:** If the remaining elements after popping an element from the stack use only less than a quarter of the available space in the data array, the capacity of the data array is halved.

MyStack<E> Class

```java
// A generic class that implements an array-based stack. E is a generic type.
public class MyStack<E> {
   // data fields
   private E[] data; // a generic array for storing the elements in the stack
   private int size = 0; // the number of the stored elements (initially 0)
   private static final int DEFAULT_CAPACITY = 5; // the default array capacity

   // constructor that creates an empty stack with the default initial capacity
   public MyStack() {
      // invoking the constructor MyStack(int capacity) with DEFAULT_CAPACITY
      this(DEFAULT_CAPACITY);
   }

   // constructor that creates an empty stack with the specified initial capacity
   public MyStack(int capacity) {
      // the array is created with the type Object instead of the generic type E
      // then casted to E[] as generic array creation is not possible
      data = (E[]) new Object[capacity];
   }

   // returns true if the stack is empty and false otherwise
   public boolean isEmpty() { return size == 0; }
```



MyStack<E> Class

```java
// returns the top element in the stack without removing it
public E peek() {
   // return null if the stack is empty
   if (isEmpty()) return null;
   // return the last element in the array (the top element in the stack)
   return data[size - 1];
}

// removes and returns the top element in the stack
public E pop() {
   // return null if the stack is empty
   if (isEmpty()) return null;
   // store the top element in a variable to return after it is removed
   E top = data[size - 1];
   // decrease size by 1 and remove the top element
   data[--size] = null; // dereference to help garbage collection
   // when only less than a quarter of the data array is used
   if (size < data.length / 4 && !isEmpty()) // and the array is not empty
      // shrink the capacity of the data array to its half
      resize(data.length / 2); // by using the resize method
   // return the removed top element that is stored in the variable top
   return top;
}
```

MyStack<E>
-data: E[] -size: int -DEFAULT_CAPACITY: int
+MyStack() +MyStack(capacity: int) +isEmpty(): boolean +peek(): E +pop(): E +push(e: E): void -resize(capacity: int): void +toString(): String

A generic array for storing the elements in the stack.
The number of the stored elements (initially 0).
The default capacity for the array (a constant equal to 5).

Creates an empty stack with the default initial capacity.
Creates an empty stack with the given initial capacity.
Returns true if the stack is empty and false otherwise.
Returns the top element in the stack without removing it (returns null if the stack is empty).
Removes and returns the top element in the stack (returns null if the stack is empty).
Adds the given element e to the top of the stack.
Resizes the data array to a given capacity. It is invoked before adding or after removing an element (if necessary).
Returns the stack contents as a string.

MyStack<E> Class

```java
// adds the specified element e to the top of the stack
public void push(E e) {
   // when the data array is full
   if (size == data.length)
      // double the capacity of the data array by using the resize method
      resize(2 * data.length);
   // add the element e after the last stored element in the array
   // and increase size by 1
   data[size++] = e;
}

// private (inner) method that resizes the data array to a given capacity
private void resize(int capacity) {
   // print a message that the stack capacity is updated
   System.out.println("Stack capacity: " + data.length + " -> " + capacity);
   // create a new array with the given capacity
   E[] newArray = (E[]) new Object[capacity];
   // copy the stored elements to the new array
   System.arraycopy(data, 0, newArray, 0, size);
   // set the data array as the new array
   data = newArray;
}
```

MyStack<E>	
-data: E[] -size: int -DEFAULT_CAPACITY: int	A generic array for storing the elements in the stack. The number of the stored elements (initially 0). The default capacity for the array (a constant equal to 5).
+MyStack() +MyStack(capacity: int) +isEmpty(): boolean +peek(): E +pop(): E +push(e: E): void -resize(capacity: int): void +toString(): String	Creates an empty stack with the default initial capacity. Creates an empty stack with the given initial capacity. Returns true if the stack is empty and false otherwise. Returns the top element in the stack without removing it (returns null if the stack is empty). Removes and returns the top element in the stack (returns null if the stack is empty). Adds the given element e to the top of the stack. Resizes the data array to a given capacity. It is invoked before adding or after removing an element (if necessary). Returns the stack contents as a string.

MyStack<E> Class

```
    // returns the stack contents as a string
    @Override // overriding the toString method in the Object class
    public String toString() {
        // return "[]" if the stack is empty
        if (isEmpty())
            return "[]";
        // str is initialized with "[" to show the beginning of the elements
        String str = "[";
        // traverse the array backward to add each element from top to bottom
        for (int i = size - 1; i >= 0; i--) {
            // append the string representation of the current element to str
            str = str + data[i];
            // append ", " to str if this is not the last element
            if (i != 0)
                str = str + ", ";
            // otherwise append "]" to str to show the end of the elements
            else
                str = str + "]";
        }
        // return the resulting string
        return str;
    }
} // End of the MyStack<E> class
```

MyStack<E>
-data: E[] -size: int -DEFAULT_CAPACITY: int
+MyStack() +MyStack(capacity: int) +isEmpty(): boolean +peek(): E +pop(): E +push(e: E): void -resize(capacity: int): void +toString(): String

A generic array for storing the elements in the stack.
The number of the stored elements (initially 0).
The default capacity for the array (a constant equal to 5).

Creates an empty stack with the default initial capacity.
Creates an empty stack with the given initial capacity.
Returns true if the stack is empty and false otherwise.
Returns the top element in the stack without removing it (returns null if the stack is empty).
Removes and returns the top element in the stack (returns null if the stack is empty).
Adds the given element e to the top of the stack.
Resizes the data array to a given capacity. It is invoked before adding or after removing an element (if necessary).
Returns the stack contents as a string.

Demo Code for the MyStack<E> Class

- Below is the given code (MyStackDemo.java) that demonstrates the capabilities of the MyStack<E> class.
- The console output of the code is given on the right.

```java
// A program that demonstrates the capabilities of the MyStack<E> class.
public class MyStackDemo {
   public static void main(String[] args) {
      // create a stack that can store 3 integer elements
      // by giving 3 as the capacity input argument of the constructor
      MyStack<Integer> stack = new MyStack<>(3);

      // push each number from 0 until 10 to the stack and print the stack
      for (int num = 0; num < 10; num++) {
         stack.push(num);
         System.out.println(num + " is pushed. Stack: top -> " + stack);
      }
      System.out.println(); // adding a new line

      // pop each number in the stack and print the stack
      while (!stack.isEmpty()) { // loop until the stack becomes empty
         System.out.println(stack.peek() + " is the top element in the stack.");
         System.out.println(stack.pop() + " is popped. Stack: top -> " + stack);
      }
   }
}
```

Console Output of the Program:

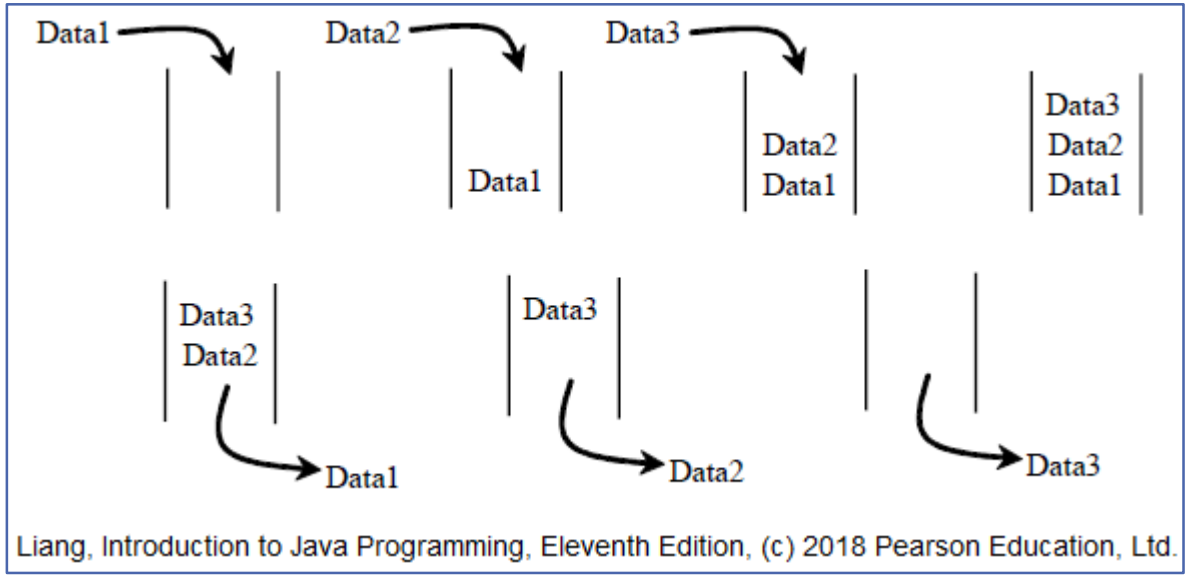
```
0 is pushed. Stack: top -> [0]
1 is pushed. Stack: top -> [1, 0]
2 is pushed. Stack: top -> [2, 1, 0]
Stack capacity: 3 -> 6
3 is pushed. Stack: top -> [3, 2, 1, 0]
4 is pushed. Stack: top -> [4, 3, 2, 1, 0]
5 is pushed. Stack: top -> [5, 4, 3, 2, 1, 0]
Stack capacity: 6 -> 12
6 is pushed. Stack: top -> [6, 5, 4, 3, 2, 1, 0]
7 is pushed. Stack: top -> [7, 6, 5, 4, 3, 2, 1, 0]
8 is pushed. Stack: top -> [8, 7, 6, 5, 4, 3, 2, 1, 0]
9 is pushed. Stack: top -> [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

9 is the top element in the stack.
9 is popped. Stack: top -> [8, 7, 6, 5, 4, 3, 2, 1, 0]
8 is the top element in the stack.
8 is popped. Stack: top -> [7, 6, 5, 4, 3, 2, 1, 0]
7 is the top element in the stack.
7 is popped. Stack: top -> [6, 5, 4, 3, 2, 1, 0]
6 is the top element in the stack.
6 is popped. Stack: top -> [5, 4, 3, 2, 1, 0]
5 is the top element in the stack.
5 is popped. Stack: top -> [4, 3, 2, 1, 0]
4 is the top element in the stack.
4 is popped. Stack: top -> [3, 2, 1, 0]
3 is the top element in the stack.
3 is popped. Stack: top -> [2, 1, 0]
2 is the top element in the stack.
Stack capacity: 12 -> 6
2 is popped. Stack: top -> [1, 0]
1 is the top element in the stack.
1 is popped. Stack: top -> [0]
0 is the top element in the stack.
0 is popped. Stack: top -> []
```

An Array-Based Queue Implementation

An overview of the queue data structure and the strategies for implementing an array-based queue.

Queue Data Structure

- A queue is a data structure based on the first-in, first-out (FIFO) principle.
- Elements are inserted only from the end and accessed or removed only from the beginning of the queue as shown on the right.
- Here you can find an animation that shows how the queue data structure works.



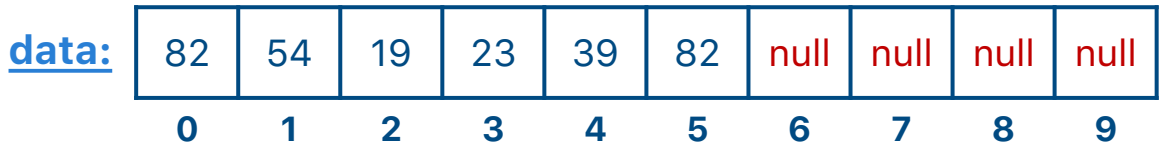
Liang, Introduction to Java Programming, Eleventh Edition, (c) 2018 Pearson Education, Ltd.

- **Some Real-Life Examples:**

Array-Based Queue Implementation

Implementation Strategy 1:

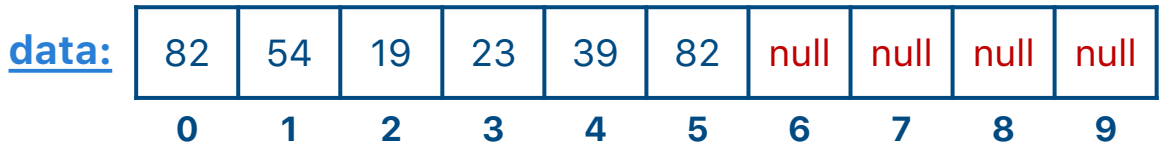
- The internal array can be maintained in a similar way as we did for the stack.
  - Each added element to the queue can be placed in the array one after another starting from the index 0.



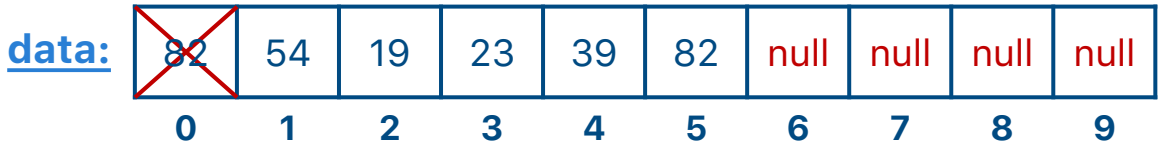
Array-Based Queue Implementation

Implementation Strategy 1: → Inefficient

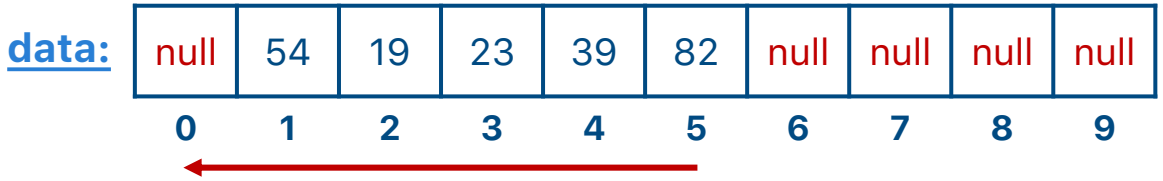
- The internal array can be maintained in a similar way as we did for the stack.
  - Each added element to the queue can be placed in the array one after another starting from the index 0.



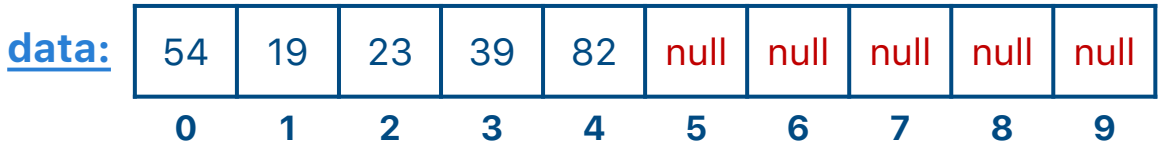
- This is an inefficient way for maintaining the internal array as the dequeue operation requires shifting a potentially large number of elements.



dequeue: removing and returning the first element 82



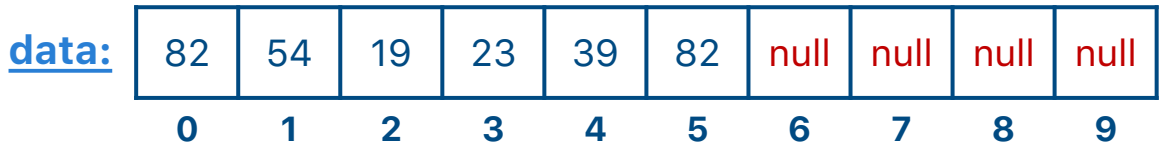
then shifting all the stored elements left by one



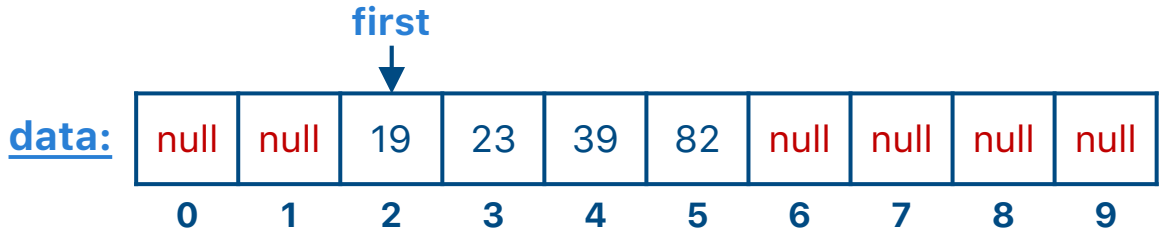
Array-Based Queue Implementation

Implementation Strategy 2:

- The internal array can be maintained in a similar way as we did for the stack.
  - Each added element to the queue can be placed in the array one after another starting from the index 0.



- Each dequeued element can be replaced with a null reference and a variable (first) can be used to keep track of the index of the first (head) element in the queue.

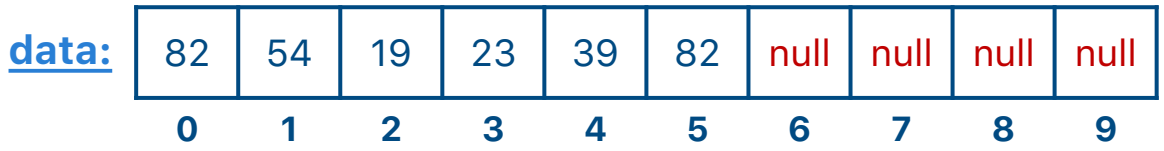


- This way we allow the front of the queue to drift away from index 0.

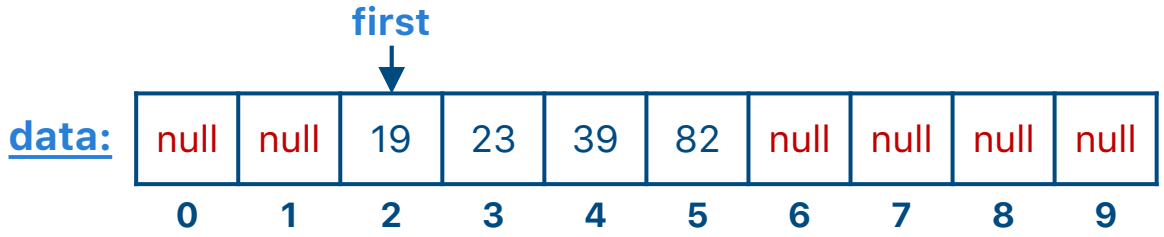
Array-Based Queue Implementation

Implementation Strategy 2: → Inefficient

- The internal array can be maintained in a similar way as we did for the stack.
  - Each added element to the queue can be placed in the array one after another starting from the index 0.



- Each dequeued element can be replaced with a null reference and a variable (first) can be used to keep track of the index of the first (head) element in the queue.



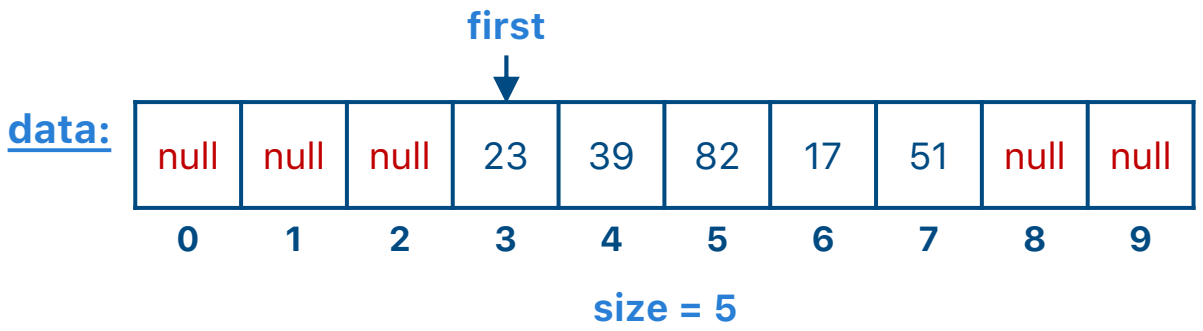
- This way we allow the front of the queue to drift away from index 0.
- This leads to an inefficient usage of the memory as the slots for the dequeued elements are not used again.
  - Assume that the data array has the capacity to store N elements. Using this implementation strategy, it is not possible to store N elements after some elements are dequeued.

Array-Based Queue Implementation

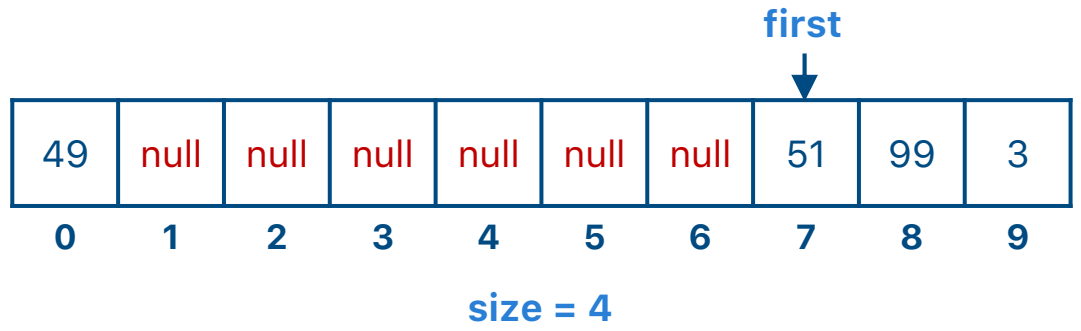
Implementation Strategy 3:

- Each added element to the queue can be placed in the array one after another starting from the index 0.
- Each dequeued element can be replaced with a null reference.
- Two variables can be used for indexing the elements in the internal array circularly.
  - **first:** for keeping track of the index of the first (head) element in the queue
  - **size:** for keeping track of the number of the stored elements
- When the data array is used as usual, the element at the index `i` can be accessed as `data[i]`.
- When the data array is indexed **circularly**, the index `i` becomes `(first + i) % data.length` and the element at the index `i` can be accessed as `data[(first + i) % data.length]`.
- As the array is maintained circularly the contents of the array can be in one of the two different configurations shown below.

Normal Configuration



Wrapped-Around Configuration



The Task for This Lab Work

Array-Based Queue Implementation

- **Task:** Implement an array-based queue that uses the internal array circularly.
- Below is the UML diagram for the MyQueue<E> class that you will implement as the task of this lab.
- The given code for testing the MyQueue<E> class (TestMyStackAndMyQueue.java) is shown on next slide.

MyQueue<E>	
-data: E[] -size: int -first: int	A generic array for storing the elements in the queue. The number of the stored elements (initially 0). The index of the first (head) element in the queue (initially 0).
+MyQueue(capacity: int) +isEmpty(): boolean +peek(): E +dequeue(): E +enqueue(e: E): boolean +toString(): String	Creates an empty queue with the given initial capacity. Returns true if the queue is empty and false otherwise. Returns the element at the start (head) of the queue without removing it (returns null if the queue is empty). Removes and returns the element at the start (head) of the queue (returns null if the queue is empty). Adds the given element e to the end (tail) of the queue and returns true if the queue is not full, returns false otherwise. Returns both the queue contents and the internal array contents as a string (See the output on the next slide).

Test Code for MyQueue<E> Class

```java
// A program for testing MyStack<E> and MyQueue<E> classes
public class TestMyStackAndMyQueue {
   public static void main(String[] args) {
      // create a queue that can store 5 integer elements
      // by giving 5 as the capacity input argument of the constructor
      MyQueue<Integer> queue = new MyQueue<>(5);
      // try to enqueue numbers from 1 to 7 and print the queue contents each
      // time the number is enqueued successfully
      for (int num = 1; num <= 7; num++)
         if (queue.enqueue(num))
            System.out.println(num + " is enqueued. Queue: head -> " + queue);
         else
            System.out.println(num + " cannot be enqueued as the queue is full.");
      System.out.println(); // adding a new line

      // create a stack to store integer elements
      MyStack<Integer> stack = new MyStack<>();
      // transfer the first 3 items in the queue to the stack
      for (int i = 0; i < 3; i++) {
         int item = queue.peek(); // get the first item (head) of the queue
         // push the item to the stack and print the stack
         stack.push(item);
         System.out.println(item + " is pushed. Stack: top -> " + stack);
         // dequeue the first item (head) of the queue and print the queue
         queue.dequeue();
         System.out.println(item + " is dequeued. Queue: head -> " + queue);
      }
      System.out.println(); // adding a new line

      // transfer each item in the stack back to the queue
      while (!stack.isEmpty()) { // loop until the stack becomes empty
         // pop the top item and print the stack
         int item = stack.peek(); // get the top item of the stack
         System.out.println(stack.pop() + " is popped. Stack: top -> " + stack);
         // enqueue the item and print the queue
         queue.enqueue(item);
         System.out.println(item + " is enqueued. Queue: head -> " + queue);
      }
   }
}
```

- The given code for testing the MyQueue<E> class (TestMyStackAndMyQueue.java) is shown on the left.
- It also uses the given MyStack<E> class (MyStack.java).
- The console output of the code is given below.

Console Output of the Program:

```
1 is enqueued. Queue: head -> [1] and Inner Array: [1, null, null, null, null]
2 is enqueued. Queue: head -> [1, 2] and Inner Array: [1, 2, null, null, null]
3 is enqueued. Queue: head -> [1, 2, 3] and Inner Array: [1, 2, 3, null, null]
4 is enqueued. Queue: head -> [1, 2, 3, 4] and Inner Array: [1, 2, 3, 4, null]
5 is enqueued. Queue: head -> [1, 2, 3, 4, 5] and Inner Array: [1, 2, 3, 4, 5]
6 cannot be enqueued as the queue is full.
7 cannot be enqueued as the queue is full.

1 is pushed. Stack: top -> [1]
1 is dequeued. Queue: head -> [2, 3, 4, 5] and Inner Array: [null, 2, 3, 4, 5]
2 is pushed. Stack: top -> [2, 1]
2 is dequeued. Queue: head -> [3, 4, 5] and Inner Array: [null, null, 3, 4, 5]
3 is pushed. Stack: top -> [3, 2, 1]
3 is dequeued. Queue: head -> [4, 5] and Inner Array: [null, null, null, 4, 5]

3 is popped. Stack: top -> [2, 1]
3 is enqueued. Queue: head -> [4, 5, 3] and Inner Array: [3, null, null, 4, 5]
2 is popped. Stack: top -> [1]
2 is enqueued. Queue: head -> [4, 5, 3, 2] and Inner Array: [3, 2, null, 4, 5]
1 is popped. Stack: top -> []
1 is enqueued. Queue: head -> [4, 5, 3, 2, 1] and Inner Array: [3, 2, 1, 4, 5]
```

Hints

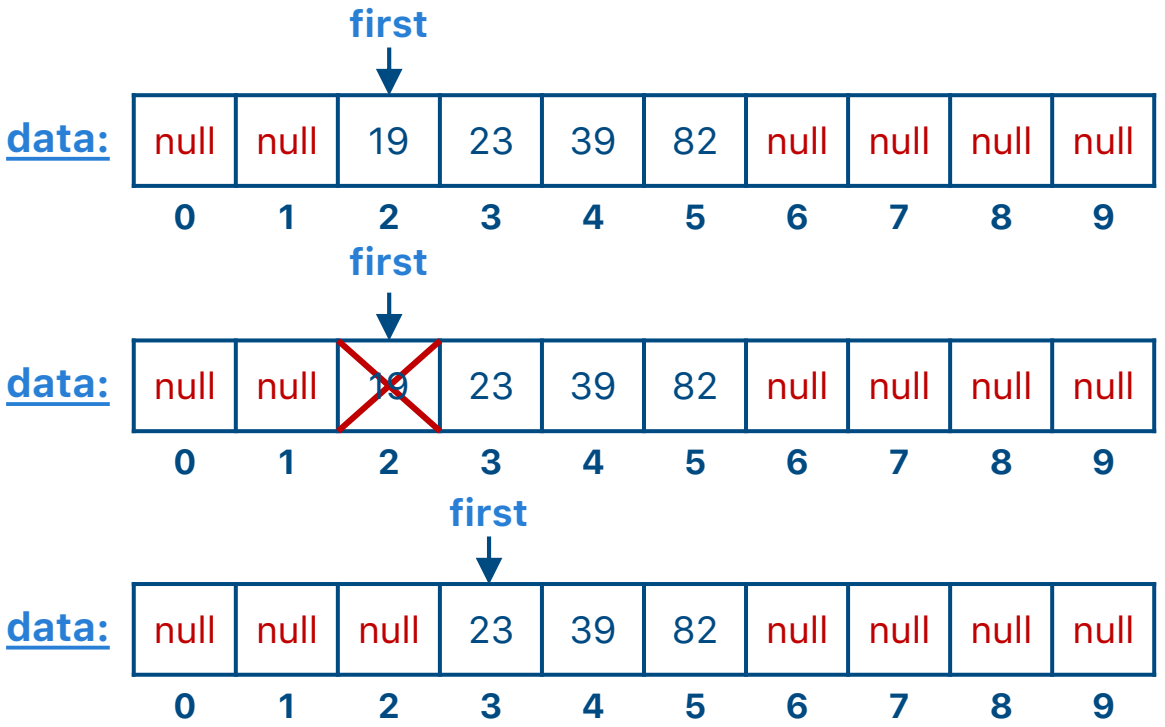
MyQueue<E>	
-data: E[]	A generic array for storing the elements in the queue.
-size: int	The number of the stored elements (initially 0).
-first: int	The index of the first (head) element in the queue (initially 0).
+MyQueue(capacity: int)	Creates an empty queue with the given initial capacity.
+isEmpty(): boolean	Returns true if the queue is empty and false otherwise.
+peek(): E	Returns the element at the start (head) of the queue without removing it (returns null if the queue is empty).
+dequeue(): E	Removes and returns the element at the start (head) of the queue (returns null if the queue is empty).
+enqueue(e: E): boolean	Adds the given element e to the end (tail) of the queue and returns true if the queue is not full, returns false otherwise.
+toString(): String	Returns both the queue contents and the internal array contents as a string (See the output on the next slide).

- The element at the start (head) of the queue is stored in the internal array (data) at the index first.

Hints

MyQueue<E>	
-data: E[] -size: int -first: int	A generic array for storing the elements in the queue. The number of the stored elements (initially 0). The index of the first (head) element in the queue (initially 0).
+MyQueue(capacity: int) +isEmpty(): boolean +peek(): E +dequeue(): E +enqueue(e: E): boolean +toString(): String	Creates an empty queue with the given initial capacity. Returns true if the queue is empty and false otherwise. Returns the element at the start (head) of the queue without removing it (returns null if the queue is empty). Removes and returns the element at the start (head) of the queue (returns null if the queue is empty). Adds the given element e to the end (tail) of the queue and returns true if the queue is not full, returns false otherwise. Returns both the queue contents and the internal array contents as a string (See the output on the next slide).

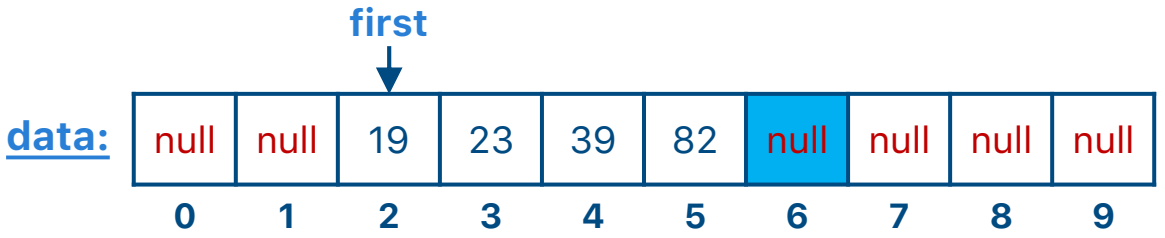
- You need to update the data field first as well as the data field size in the dequeue method.
- The index first must be increased by 1 circularly.
- **Note:**
  - When the data array is used as usual, the element at the index `i` can be accessed as `data[i]`.
  - When the data array is indexed circularly, the index `i` becomes `(first + i) % data.length` and the element at the index `i` can be accessed as `data[(first + i) % data.length]`.



Hints

MyQueue<E>	
-data: E[]	A generic array for storing the elements in the queue.
-size: int	The number of the stored elements (initially 0).
-first: int	The index of the first (head) element in the queue (initially 0).
+MyQueue(capacity: int)	Creates an empty queue with the given initial capacity.
+isEmpty(): boolean	Returns true if the queue is empty and false otherwise.
+peek(): E	Returns the element at the start (head) of the queue without removing it (returns null if the queue is empty).
+dequeue(): E	Removes and returns the element at the start (head) of the queue (returns null if the queue is empty).
+enqueue(e: E): boolean	Adds the given element e to the end (tail) of the queue and returns true if the queue is not full, returns false otherwise.
+toString(): String	Returns both the queue contents and the internal array contents as a string (See the output on the next slide).

- You need to compute the index in the data array for adding the new element.
- This index must be computed circularly.
- **Note:**
  - When the data array is used as usual, the element at the index `i` can be accessed as `data[i]`.
  - When the data array is indexed **circularly**, the index `i` becomes `(first + i) % data.length` and the element at the index `i` can be accessed as `data[(first + i) % data.length]`.



Hints

MyQueue<E>	
-data: E[]	A generic array for storing the elements in the queue.
-size: int	The number of the stored elements (initially 0).
-first: int	The index of the first (head) element in the queue (initially 0).
+MyQueue(capacity: int)	Creates an empty queue with the given initial capacity.
+isEmpty(): boolean	Returns true if the queue is empty and false otherwise.
+peek(): E	Returns the element at the start (head) of the queue without removing it (returns null if the queue is empty).
+dequeue(): E	Removes and returns the element at the start (head) of the queue (returns null if the queue is empty).
+enqueue(e: E): boolean	Adds the given element e to the end (tail) of the queue and returns true if the queue is not full, returns false otherwise.
+toString(): String	Returns both the queue contents and the internal array contents as a string (See the output on the next slide).

- You need to traverse all the stored elements (size) circularly starting from the element at the index first and append each element to the string that is returned.
- For appending the internal array (data) contents, you can use the toString method of the java.util.Arrays class.
- **Note:**
  - When the data array is used as usual, the element at the index `i` can be accessed as `data[i]`.
  - When the data array is indexed **circularly**, the index `i` becomes `(first + i) % data.length` and the element at the index `i` can be accessed as `data[(first + i) % data.length]`.

Submission of Lab Work

- Submit your **Java code file(s)** (no report required) to Blackboard.
- You can work as a team of 2 students, if you like. Each team member should upload his/her code to Blackboard individually.

Grading
Your lab work will be graded on a scale of 0: Incorrect/NA, 1: Partially Correct and 2: Correct.