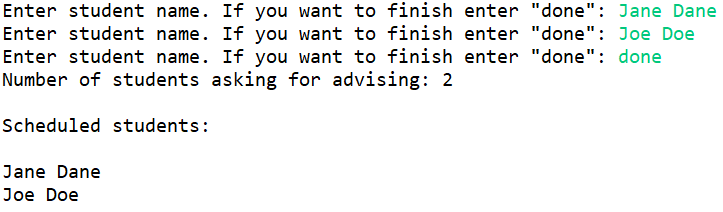
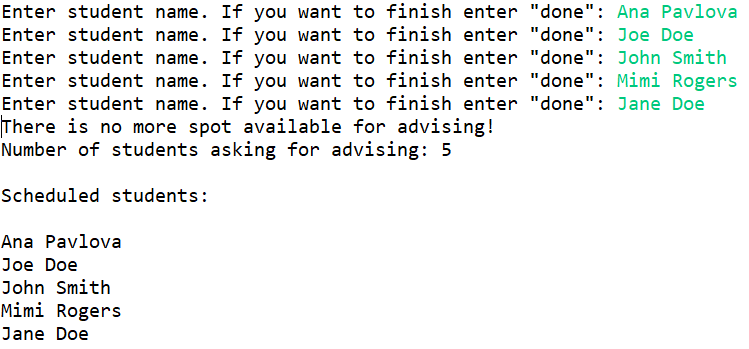
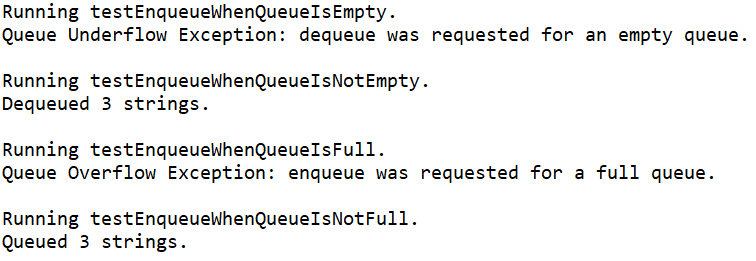
**1**

Main Output





Test Output



Code

**package** Com.TSL.AdvisingScheduleUtilities;

**import** java.util.Scanner;

/\*\*

\* **@author** EMILIA BUTU

\* version 1.0

\* since 2020-03

\*

\* Student name: Tom Lever

\* Completion date: 06/09/21

\*

\* Implements the application for AdvisingSchedule\_Array class, using queue.

\* Outputs strings in same order of entry.

\*/

**public** **class** AdvisingSchedule\_ArrayDriver

{

/\*\*

\* main is the entry point of this program, which repeatedly prompts a user to enter a student name until "done" is

\* entered. The program proceeds to enqueue each name in a queue of names. If a name has been entered but the queue

\* is full, the program overwrites the name with "done" and outputs a warning. The program prints information about

\* the number of names in the queue and displays the names in the queue in the order in which they were entered into

\* the queue.

\*

\* **@param** args

\*/

**public** **static** **void** main(String[] args)

{

Scanner scan = **new** Scanner(System.***in***);

//\*\*\* Task #1: defime a queue with elements of type String using QueueInterface

QueueInterface<String> nameQueue;

//\*\*\* Task #2: instantiate the queue using the constructor that provides the size, using a smaller number for test

nameQueue = **new** AdvisingSchedule\_Array<String>(5);

String name="";

//\*\*\* Task #3: enter the names of students that ask for advising.

//\* allow the user to terminate the entry

//\* terminate the loop when the queue is full, and display appropriate message to signal it

**while**(!name.equalsIgnoreCase("done"))

{

System.***out***.print("Enter student name. If you want to finish enter \"done\": ");

name = scan.nextLine();

**if**(!name.equalsIgnoreCase("done"))

nameQueue.enqueue(name);

// test if the queue is full

**if**(nameQueue.isFull())

{

name="done";

System.***out***.println("There is no more spot available for advising!");

}

}

//\*\*\* Task #4: identify the number of students in the advising queue and display it

**int** noStudents = nameQueue.size();

System.***out***.println("Number of students asking for advising: " + noStudents);

//\*\*\* Task #5: display the list of students presents in the advising queue

System.***out***.println("\nScheduled students:\n");

**while** (!nameQueue.isEmpty())

{

name = nameQueue.dequeue();

System.***out***.println(name);

}

}

}

**package** Com.TSL.AdvisingScheduleUtilities;

/\*\*

\* **@author** EMILIA BUTU

\* version 1.0

\* since 2020-03

\*

\* Student name: Tom Lever

\* Completion date: 06/09/21

\*

\* AdvisingSchedule\_Array.java

\*

\* Implements QueueInterface with an array to hold the queue elements.

\*

\* Two constructors are provided: one that creates a queue of a default

\* capacity and one that allows the calling program to specify the capacity.

\*/

**public** **class** AdvisingSchedule\_Array<T> **implements** QueueInterface<T>

{

**protected** **final** **int** DEFCAP = 10; // default capacity

**protected** T[] elements; // array that holds queue elements

**protected** **int** numElements = 0; // number of elements in this queue

**protected** **int** front = 0; // index of front of queue

**protected** **int** rear; // index of rear of queue

/\*\*

\* AdvisingSchedule\_Array() is a zero-parameter constructor for AdvisingSchedule\_Array, which defines this advising

\* schedule's array of elements as a new array of <default capacity> elements of type T, and sets this advising

\* schedule's index for the rear-most element in the array as the default capacity of this advising schedule minus 1.

\*/

**public** AdvisingSchedule\_Array()

{

**this**.elements = (T[]) **new** Object[DEFCAP];

**this**.rear = DEFCAP - 1;

}

/\*\*

\* AdvisingSchedule\_Array(int maxSize) is a one-parameter constructor for AdvisingSchedule\_Array, which defines this

\* advising schedule's array of elements as a new array of <argument> elements of type T, and sets this advising

\* schedule's index for the rear-most element in the array as the argument minus 1.

\* **@param** maxSize

\*/

**public** AdvisingSchedule\_Array(**int** maxSize)

{

**this**.elements = (T[]) **new** Object[maxSize];

**this**.rear = maxSize - 1;

}

**public** **void** enqueue(T element)

// Throws QueueOverflowException if this queue is full;

// otherwise, adds element to the rear of this queue.

{

**if** (isFull()) {

**throw** **new** QueueOverflowException("Queue Overflow Exception: enqueue was requested for a full queue.");

}

**this**.rear = (**this**.rear + 1) % **this**.elements.length;

**this**.elements[**this**.rear] = element;

**this**.numElements++;

}

**public** T dequeue()

// Throws QueueUnderflowException if this queue is empty;

// otherwise, removes front element from this queue and returns it.

{

**if** (isEmpty()) {

**throw** **new** QueueUnderflowException("Queue Underflow Exception: dequeue was requested for an empty queue.");

}

T theStorage = **this**.elements[**this**.front];

**this**.elements[**this**.front] = **null**;

**this**.front = (**this**.front + 1) % **this**.elements.length;

**this**.numElements--;

**return** theStorage;

}

**public** **boolean** isEmpty()

// Returns true if this queue is empty; otherwise, returns false.

{

**return** (**this**.numElements == 0);

}

**public** **boolean** isFull()

// Returns true if this queue is full; otherwise, returns false.

{

**return** (**this**.numElements == **this**.elements.length);

}

**public** **int** size()

// Returns the number of elements in this queue.

{

**return** **this**.numElements;

}

}

**package** Com.TSL.AdvisingScheduleUtilities;

//----------------------------------------------------------------------------

//QueueInterface.java

//

//Interface for a class that implements a queue of T.

//A queue is a "first in, first out" structure.

//

// Author: Emilia Butu

// Version: 1.0

// Since: 06/09/21

//----------------------------------------------------------------------------

**public** **interface** QueueInterface<T>

{

**void** enqueue(T element) **throws** QueueOverflowException;

// Throws QueueOverflowException if this queue is full;

// otherwise, adds element to the rear of this queue.

T dequeue() **throws** QueueUnderflowException;

// Throws QueueUnderflowException if this queue is empty;

// otherwise, removes front element from this queue and returns it.

**boolean** isFull();

// Returns true if this queue is full; otherwise, returns false.

**boolean** isEmpty();

// Returns true if this queue is empty; otherwise, returns false.

**int** size();

// Returns the number of elements in this queue.

}

**package** Com.TSL.AdvisingScheduleUtilities;

/\*\*

\* QueueOverflowException represents an exception that is thrown when an enqueue operation is requested for a full queue.

\*

\* **@author** Tom Lever

\* **@version** 1.0

\* **@since** 06/09/21

\*/

**public** **class** QueueOverflowException **extends** RuntimeException

{

/\*\*

\* QueueOverflowException() is a conventional zero-parameter constructor for QueueOverflowException, which calls

\* Exception's zero-parameter constructor.

\*/

**public** QueueOverflowException()

{

**super**();

}

/\*\*

\* QueueOverflowException(String message) is a one-parameter constructor for QueueOverflowException, which passes

\* argument message to Exception's one-parameter constructor.

\* **@param** message

\*/

**public** QueueOverflowException(String message)

{

**super**(message);

}

}

**package** Com.TSL.AdvisingScheduleUtilities;

/\*\*

\* QueueUnderflowException represents an exception that is thrown when an dequeue operation is requested for an empty

\* queue.

\*

\* **@author** Tom Lever

\* **@version** 1.0

\* **@since** 06/09/21

\*/

**public** **class** QueueUnderflowException **extends** RuntimeException

{

/\*\*

\* QueueUnderflowException() is a conventional zero-parameter constructor for QueueUnderflowException, which calls

\* Exception's zero-parameter constructor.

\*/

**public** QueueUnderflowException()

{

**super**();

}

/\*\*

\* QueueUnderflowException(String message) is a one-parameter constructor for QueueUnderflowException, which passes

\* argument message to Exception's one-parameter constructor.

\* **@param** message

\*/

**public** QueueUnderflowException(String message)

{

**super**(message);

}

}

**package** Com.TSL.AdvisingScheduleUtilities;

**import** org.junit.jupiter.api.Test;

/\*\*

\* AdvisingScheduleTest encapsulates JUnit tests of the enqueue and dequeue methods of AdvisingSchedule\_Array.

\* **@author** Tom Lever

\* **@version** 1.0

\* **@since** 06/09/21

\*/

**public** **class** AdvisingScheduleTest {

/\*\*

\* testEnqueueWhenQueueIsNotFull tests enqueue by adding three names to an advising schedule with a capacity of

\* more than 3 names.

\*/

@Test

**public** **void** testEnqueueWhenQueueIsNotFull() {

System.***out***.println("Running testEnqueueWhenQueueIsNotFull.");

AdvisingSchedule\_Array<String> theAdvisingSchedule = **new** AdvisingSchedule\_Array<String>();

**try** {

**for** (**int** i = 0; i < 3; i++) {

theAdvisingSchedule.enqueue("aTestString");

}

System.***out***.println("Queued 3 strings.");

}

**catch** (QueueOverflowException theQueueOverflowException) {

System.***out***.println(theQueueOverflowException.getMessage());

}

System.***out***.println();

}

/\*\*

\* testEnqueueWhenQueueIsFull tests enqueue by attempting to add 1000 names to an advising schedule with a capacity

\* significantly less than 1000.

\*/

@Test

**public** **void** testEnqueueWhenQueueIsFull() {

System.***out***.println("Running testEnqueueWhenQueueIsFull.");

AdvisingSchedule\_Array<String> theAdvisingSchedule = **new** AdvisingSchedule\_Array<String>();

**try** {

**for** (**int** i = 0; i < 1000; i++) {

theAdvisingSchedule.enqueue("aTestString");

}

System.***out***.println("Queued 1000 strings.");

}

**catch** (QueueOverflowException theQueueOverflowException) {

System.***out***.println(theQueueOverflowException.getMessage());

}

System.***out***.println();

}

/\*\*

\* testEnqueueWhenQueueIsNotFull tests dequeue by removing three names from an advising schedule with three names

\* in it.

\*/

@Test

**public** **void** testDequeueWhenQueueIsNotEmpty() {

System.***out***.println("Running testEnqueueWhenQueueIsNotEmpty.");

AdvisingSchedule\_Array<String> theAdvisingSchedule = **new** AdvisingSchedule\_Array<String>();

**for** (**int** i = 0; i < 3; i++) {

theAdvisingSchedule.enqueue("aTestString");

}

**try** {

**for** (**int** i = 0; i < 3; i++) {

theAdvisingSchedule.dequeue();

}

System.***out***.println("Dequeued 3 strings.");

}

**catch** (QueueUnderflowException theQueueUnderflowException) {

System.***out***.println(theQueueUnderflowException.getMessage());

}

System.***out***.println();

}

/\*\*

\* testEnqueueWhenQueueIsNotFull tests dequeue by attempting to remove 1000 names from an advising schedule with

\* significantly fewer elements in it.

\*/

@Test

**public** **void** testDequeueWhenQueueIsEmpty() {

System.***out***.println("Running testEnqueueWhenQueueIsEmpty.");

AdvisingSchedule\_Array<String> theAdvisingSchedule = **new** AdvisingSchedule\_Array<String>();

**for** (**int** i = 0; i < 3; i++) {

theAdvisingSchedule.enqueue("aTestString");

}

**try** {

**for** (**int** i = 0; i < 1000; i++) {

theAdvisingSchedule.dequeue();

}

System.***out***.println("Dequeued 1000 strings.");

}

**catch** (QueueUnderflowException theQueueUnderflowException) {

System.***out***.println(theQueueUnderflowException.getMessage());

}

System.***out***.println();

}

}