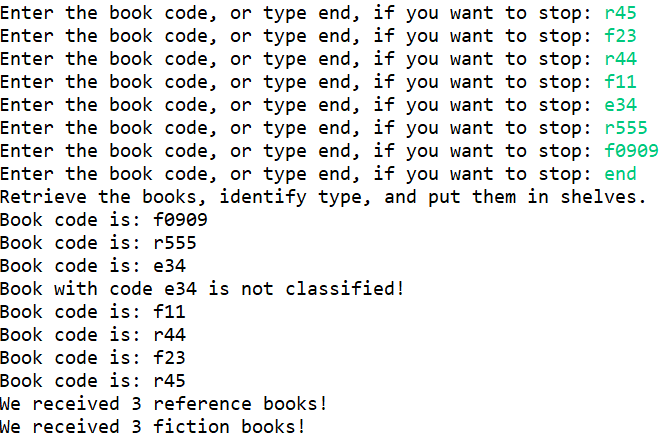
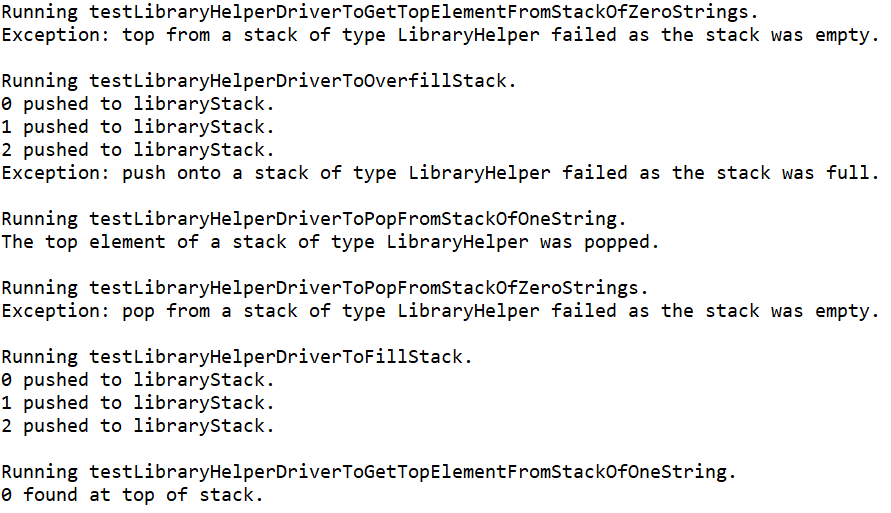
**2**

Main Output



Test Output



Code

**package** com.tsl.library\_helper;

**import** java.util.Scanner;

/\*\*

\* **@author** EMILIA BUTU

\* version 1.0

\* since 2020-05

\*

\* Student name: Tom Lever

\* Completion date: 05/27/21

\*

\* LibraryHelperDriver.java: demonstrates the use of LibraryHelper class

\*

\* Creates and pushes elements of type String in a stack,

\* Retrieves the elements, and counts the ones that have code starting with R or F

\*

\* Student tasks: complete tasks specified in the file

\*/

**public** **class** LibraryHelperDriver {

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

/\*\*

\* main is the entry point of the program. The program requests that a user enter books codes, followed by "end". The

\* program prints the book codes, the number of book codes corresponding to reference books, and the number of book

\* codes corresponding to fiction books. The program throws exceptions if the program is terminated while input is

\* being requested from the user, a push of a book code onto the stack is requested when the stack is full, a look

\* at the top book code on the stack is requested when the stack is empty, and when a pop off the stack is requested

\* when the stack is empty.

\* **@param** args

\*/

//\*\*\* Task #1: define a libraryStack variable of type StackInterface with element type String

StackInterface<String> libraryStack;

//\*\*\* Task #2: instantiate the libraryStack of LibraryHelper type with 10 elements cap

libraryStack = **new** LibraryHelper<String>(10);

// prepare the Scanner object to enter data from the user

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

String bookCode="";

//\*\*\* Task #3: create a loop in which you read book codes until user enters "end"

**while**(!bookCode.equalsIgnoreCase("end")) {

System.***out***.print("Enter the book code, or type end, if you want to stop: ");

bookCode=input.next();

**if**(!bookCode.equalsIgnoreCase("end")) {

libraryStack.push(bookCode);

}

}

input.close();

System.***out***.println("Retrieve the books, identify type, and put them in shelves.");

//\*\*\* Task #4: define and initialize variables to count reference and fiction books

**int** refNo=0;

**int** fictionNo=0;

//\*\*\* Task #5: pop the elements off the stack

// count the elements that start with 'R' or 'r' and elements that start with 'F' or 'f'

// if the code starts with other letter, display a message announcing that the code is not classified

**while**(!libraryStack.isEmpty())

{

bookCode=libraryStack.top();

libraryStack.pop();

System.***out***.println("Book code is: " + bookCode);

**if**(bookCode.charAt(0)=='R' ||bookCode.charAt(0)=='r') {

refNo++;

}

**else** {

**if**(bookCode.charAt(0)=='F' ||bookCode.charAt(0)=='f') {

fictionNo++;

}

**else** {

System.***out***.println("Book with code " + bookCode + " is not classified!");

}

}

}

//\*\*\* Task #6: display the number of reference book, and the number of fiction books

System.***out***.println("We received " + refNo + " reference books!");

System.***out***.println("We received " + fictionNo + " fiction books!");

}

}

**package** com.tsl.library\_helper;

/\*\*

\* StackInterface.java

\* Interface for a class that implements a stack of <T>.

\* A stack is a last-in, first-out structure.

\*/

**public** **interface** StackInterface<T> {

**void** push(T element) **throws** StackOverflowException;

// Throws StackOverflowException if this stack is full,

// otherwise places element at the top of this stack.

**void** pop() **throws** StackUnderflowException;

// Throws StackUnderflowException if this stack is empty,

// otherwise removes top element from this stack.

T top() **throws** StackUnderflowException;

// Throws StackUnderflowException if this stack is empty,

// otherwise returns top element of this stack.

**boolean** isEmpty();

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

**boolean** isFull();

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

}

**package** com.tsl.library\_helper;

/\*\*

\* StackOverflowException represents the structure for an exception that occurs when a push onto a stack is requested,

\* but the stack is full.

\* **@author** Tom Lever

\*

\*/

**public** **class** StackOverflowException **extends** RuntimeException {

**public** StackOverflowException() {

/\*\*

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

\* Exception's zero-parameter constructor.

\*/

**super**();

}

**public** StackOverflowException(String message) {

/\*\*

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

\* error message to Exception's one-parameter constructor with a message parameter.

\* **@param** message

\*/

**super**(message);

}

}

**package** com.tsl.library\_helper;

/\*\*

\* StackUnderflowException represents the structure for an exception that occurs when a pop off a stack is requested,

\* or a look at the top element in the stack is requested, but the stack is empty.

\* **@author** Tom Lever

\* **@since** 05/27/21

\*

\*/

**public** **class** StackUnderflowException **extends** RuntimeException {

**public** StackUnderflowException() {

/\*\*

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

\* Exception's zero-parameter constructor.

\*/

**super**();

}

**public** StackUnderflowException(String message) {

/\*\*

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

\* error message to Exception's one-parameter constructor with a message parameter.

\* **@param** message

\*/

**super**(message);

}

}

**package** com.tsl.library\_helper;

/\*\*

\* **@author** EMILIA BUTU

\* version 1.0

\* since 2020-05

\*

\* Student name: Tom Lever

\* Completion date: 05/27/21

\*

\* LibraryHelper.txt: save it as LibraryHelper.java

\* Implements StackInterface using an array to hold the stack elements.

\*

\* Two constructors are provided: one that creates an array of a

\* default size and one that allows the calling program to

\* specify the size.

\*

\* Student tasks: complete tasks specified in the file

\*/

**public** **class** LibraryHelper<T> **implements** StackInterface<T> {

**protected** **final** **int** DEFCAP = 100; // default capacity

**protected** T[] elements; // holds stack elements

**protected** **int** topIndex = -1; // index of top element in stack

**public** LibraryHelper() {

/\*\*

\* LibraryHelper() is a zero-parameter constructor for LibraryHelper that sets this stack's array of elements to a

\* new array of objects of type T, with a number of elements equal to this stack's default capacity.

\*/

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

}

**public** LibraryHelper(**int** maxSize) {

/\*\*

\* LibraryHelper(int maxSize) is a one-parameter constructor for LibraryHelper that sets this stack's array of

\* elements to a new array of objects of type T, with a number of elements equal to argument maxSize.

\* **@param** maxSize

\*/

//\*\*\* Task #1: implement this constructor

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

}

**public** **void** push(T element) {

// Throws StackOverflowException if this stack is full,

// otherwise places element at the top of this stack.

//\*\*\* Task #2: implement this method throwing the right exception if necessary

**if** (isFull()) {

**throw** **new** StackOverflowException(

"Exception: push onto a stack of type LibraryHelper failed as the stack was full.");

}

**this**.topIndex++;

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

}

**public** **void** pop() {

// Throws StackUnderflowException if this stack is empty,

// otherwise removes top element from this stack.

//\*\*\* Task #3: implement this method throwing the right exception if necessary

**if** (isEmpty()) {

**throw** **new** StackUnderflowException(

"Exception: pop from a stack of type LibraryHelper failed as the stack was empty.");

}

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

**this**.topIndex--;

}

**public** T top() {

// Throws StackUnderflowException if this stack is empty,

// otherwise returns top element of this stack.

//\*\*\* Task #4: implement this method throwing the right exception if necessary

**if** (isEmpty()) {

**throw** **new** StackUnderflowException(

"Exception: top from a stack of type LibraryHelper failed as the stack was empty.");

}

**return** **this**.elements[**this**.topIndex];

}

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

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

//\*\*\* Task #5: implement this method

**return** (**this**.topIndex == -1);

}

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

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

//\*\*\* Task #6: implement this method

**return** (**this**.topIndex == (**this**.elements.length - 1));

}

}

**package** com.tsl.library\_helper;

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

/\*\*

\* LibraryHelperTest encapsulates JUnit tests of core methods of class LibraryHelper.

\* **@author** Tom Lever

\* **@version** 1.0

\* **@since** 05/28/21

\*

\*/

**public** **class** LibraryHelperTest {

@Test

**public** **void** testLibraryHelperToFillStack() {

/\*\*

\* testLibraryHelperToFillStack tests LibraryHelper by correctly filling a stack of type LibraryHelper.

\*/

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

StackInterface<String> libraryStack = **new** LibraryHelper<String>(3);

**try** {

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

libraryStack.push(String.*valueOf*(i));

System.***out***.println(i + " pushed to libraryStack.");

}

}

**catch** (StackOverflowException theStackOverflowException) {

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

}

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

}

@Test

**public** **void** testLibraryHelperToOverfillStack() {

/\*\*

\* testLibraryHelperToOverfillStack tests LibraryHelper by attempting to overfill a stack of type LibraryHelper.

\*/

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

StackInterface<String> libraryStack = **new** LibraryHelper<String>(3);

**try** {

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

libraryStack.push(String.*valueOf*(i));

System.***out***.println(i + " pushed to libraryStack.");

}

}

**catch** (StackOverflowException theStackOverflowException) {

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

}

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

}

@Test

**public** **void** testLibraryHelperToGetTopElementFromStackOfOneString() {

/\*\*

\* testLibraryHelperToGetTopElementFromStackOfOneString tests LibraryHelper by getting the top element from a

\* stack of type LibraryHelper with one string.

\*/

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

StackInterface<String> libraryStack = **new** LibraryHelper<String>(3);

**try** {

libraryStack.push(String.*valueOf*(0));

System.***out***.println(libraryStack.top() + " found at top of stack.");

}

**catch** (StackUnderflowException theStackUnderflowException) {

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

}

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

}

@Test

**public** **void** testLibraryHelperToGetTopElementFromStackOfZeroStrings() {

/\*\*

\* testLibraryHelperToGetTopElementFromStackOfZeroStrings tests LibraryHelper by attempting to get the top element

\* from a stack of type LibraryHelper with zero strings.

\*/

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

StackInterface<String> libraryStack = **new** LibraryHelper<String>(3);

**try** {

System.***out***.println(libraryStack.top() + " found at top of stack.");

}

**catch** (StackUnderflowException theStackUnderflowException) {

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

}

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

}

@Test

**public** **void** testLibraryHelperToPopFromStackOfOneString() {

/\*\*

\* testLibraryHelperToPopFromStackOfOneString tests LibraryHelper by popping the top element from a stack of type

\* LibraryHelper with one string.

\*/

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

StackInterface<String> libraryStack = **new** LibraryHelper<String>(3);

**try** {

libraryStack.push(String.*valueOf*(0));

libraryStack.pop();

System.***out***.println("The top element of a stack of type LibraryHelper was popped.");

}

**catch** (StackUnderflowException theStackUnderflowException) {

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

}

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

}

@Test

**public** **void** testLibraryHelperToPopFromStackOfZeroStrings() {

/\*\*

\* testLibraryHelperToPopFromStackOfZeroStrings tests LibraryHelper by attempting to pop the top element from a

\* stack of type LibraryHelper with zero strings.

\*/

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

StackInterface<String> libraryStack = **new** LibraryHelper<String>(3);

**try** {

libraryStack.pop();

System.***out***.println("The top element of a stack of type LibraryHelper was popped.");

}

**catch** (StackUnderflowException theStackUnderflowException) {

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

}

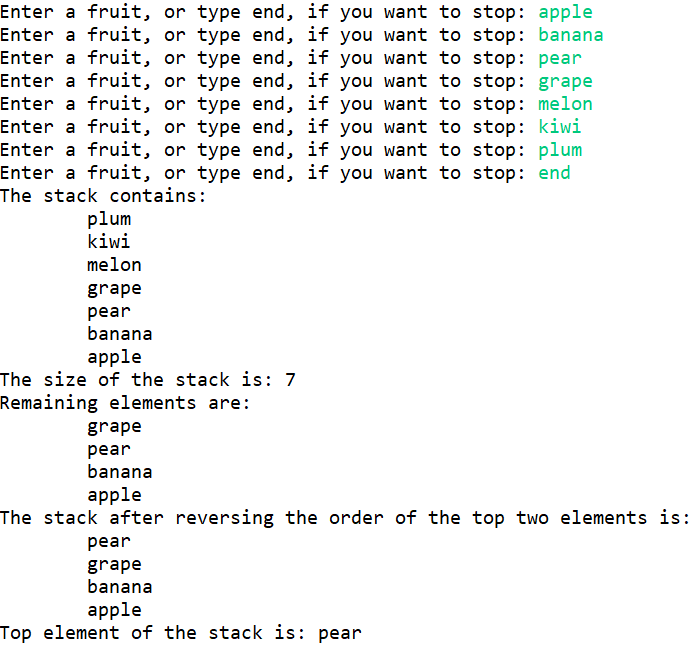
System.***out***.println();

}

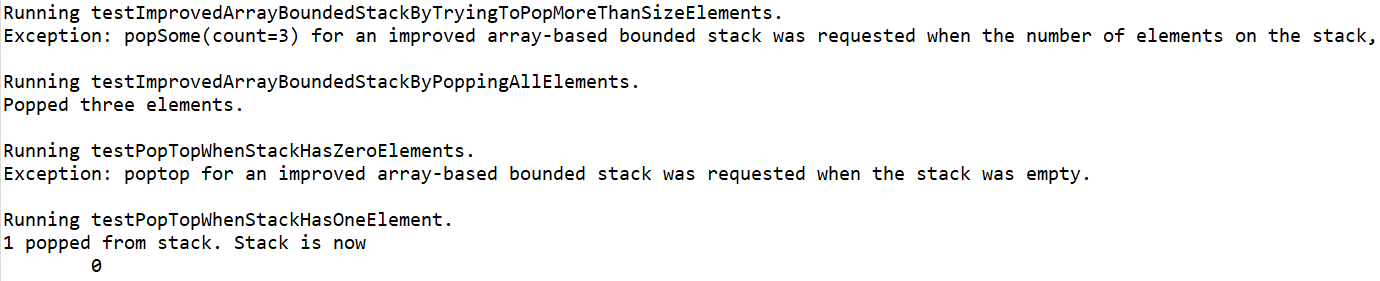
}

**3**

Main Output



Test Output



Code

**package** com.tsl.array\_based\_bounded\_stack;

**import** java.util.Scanner;

/\*\*

\* **@author** EMILIA BUTU

\* version 1.0

\* since 2020-05

\*

\* Student name: Tom Lever

\* Completion date: 05/28/21

\*

\* ImprovedArrayBoundedStackDriver.java: demonstrates the use of the new methods

\* from ImprovedArrayBoundedStack class

\*

\* Student tasks: complete tasks specified in the file

\*/

**public** **class** ImprovedArrayBoundedStackDriver {

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

/\*\*

\* main is the entry point of the program. The program requests that a user enter fruit names, followed by "end".

\* The program prints the fruit names on the stack, the size of the stack, the remaining names on the stack when

\* three names are popped from the stack, the names on the stack when the top two names on the stack are swapped,

\* and the top name that was on the stack before that name was popped from the stack. The program throws exceptions

\* if the program is terminated while input is being requested from the user, a push of a fruit name onto the stack

\* is requested when the stack is full, a pop of n names from the stack is requested when the number of names on the

\* stack is less than n, and when a classic pop from the stack is requested when the stack is empty.

\*/

ImprovedArrayBoundedStack<String> myStack;

myStack = **new** ImprovedArrayBoundedStack<String>(10);

// prepare the Scanner object to enter data from the user

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

String word="";

// read words in a loop,

**while**(!word.equalsIgnoreCase("end"))

{

System.***out***.print("Enter a fruit, or type end, if you want to stop: ");

word=input.next();

**if**(!word.equalsIgnoreCase("end"))

myStack.push(word);

}

input.close();

//\*\*\* Task #1: test method toString()

System.***out***.println("The stack contains:\n" + myStack);

//\*\*\* Task #2: test method toString()

**int** stackSize=myStack.size();

System.***out***.println("The size of the stack is: " + stackSize);

//\*\*\* Task #3: test method popSome(int count)

//\* pop three elements from the top of the stack

//\* display the resulting stack

myStack.popSome(3);

System.***out***.println("Remaining elements are: ");

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

//\*\*\* Task #4: test method swapStart()

//\* and display the resulting stack

myStack.swapStart();

System.***out***.println("The stack after reversing the order of the top two elements is: ");

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

//\*\*\* Task #5: test method poptop()

String topFruit=myStack.poptop();

System.***out***.println("Top element of the stack is: "+topFruit);

}

}

**package** com.tsl.array\_based\_bounded\_stack;

/\*\*

\* **@author** EMILIA BUTU

\* version 1.0

\* since 2020-03

\*

\* Student name: Tom Lever

\* Completion date: 05/28/21

\*

\* ImprovedArrayBoundedStack.txt: download and save as ImprovedArrayBoundedStack.java

\* Tasks are marked throughout the file

\*

\* Student tasks: complete tasks specified in the file

\*

\*/

**public** **class** ImprovedArrayBoundedStack<T> **implements** StackInterface<T>

{

**protected** **final** **int** DEFCAP = 100; // default capacity

**protected** T[] elements; // holds stack elements

**protected** **int** topIndex = -1; // index of top element in stack

**public** ImprovedArrayBoundedStack() {

/\*\*

\* ImprovedArrayBoundedStack is a zero-parameter constructor for ImprovedArrayBoundedStack that sets this stack's

\* array of elements to a new array of objects of type T, with a number of elements equal to this stack's default

\* capacity.

\*/

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

}

**public** ImprovedArrayBoundedStack(**int** maxSize) {

/\*\*

\* ImprovedArrayBoundedStack is a one-parameter constructor for ImprovedArrayBoundedStack that sets this stack's

\* array of elements to a new array of objects of type T, with a number of elements equal to argument maxSize.

\*/

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

}

**public** **void** push(T element)

// Throws StackOverflowException if this stack is full,

// otherwise places element at the top of this stack.

{

**if** (isFull())

**throw** **new** StackOverflowException("Push attempted on a full stack.");

**else**

{

topIndex++;

elements[topIndex] = element;

}

}

**public** **void** pop()

// Throws StackUnderflowException if this stack is empty,

// otherwise removes top element from this stack.

{

**if** (isEmpty())

**throw** **new** StackUnderflowException("Pop attempted on an empty stack.");

**else**

{

elements[topIndex] = **null**;

topIndex--;

}

}

**public** T top()

// Throws StackUnderflowException if this stack is empty,

// otherwise returns top element of this stack.

{

T topOfStack = **null**;

**if** (isEmpty())

**throw** **new** StackUnderflowException("Top attempted on an empty stack.");

**else**

topOfStack = elements[topIndex];

**return** topOfStack;

}

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

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

{

**return** (topIndex == -1);

}

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

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

{

**return** (topIndex == (elements.length - 1));

}

//\*\*\* Task #1: define and implement method toString(): String

//\* creates and returns a string that correctly represents the current stack.

@Override

**public** String toString() {

/\*\*

\* toString arranges strings representing the elements in this stack in an indented column.

\*/

String theStringRepresentingTheStack = "";

**for** (**int** i = topIndex; i > 0; i--) {

theStringRepresentingTheStack += "\t" + **this**.elements[i].toString() + "\n";

}

**if** (topIndex > -1) {

theStringRepresentingTheStack += "\t" + **this**.elements[0].toString();

}

**return** theStringRepresentingTheStack;

}

//\*\*\* Task #2: define and implement method size(): int

//\* returns a count of how many items are currently on the stack.

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

// returns a count of how many elements are on the stack

**return** **this**.topIndex + 1;

}

//\*\*\* Task #3: define and implement method popSome(int count): void

//\* removes the top count elements from the stack

**public** **void** popSome(**int** count) {

// if possible, removes top count elements from stack;

// otherwise throws StackUnderflowException

**if** (size() < count) {

**throw** **new** StackUnderflowException(

"Exception: popSome(count=" + count + ") for an improved array-based bounded stack was requested " +

"when the number of elements on the stack, size = " + size() + ", was less than count = " + count + "."

);

}

**while** (count > 0) {

pop();

count--;

}

}

//\*\*\* Task #4: define and implement method swapStart(): boolean

//\* if there are less than two elements on the stack returns false;

//\* otherwise it reverses the order of the top two elements on the stack and returns true

**public** **boolean** swapStart() {

// if possible, reverses order of top 2 elements and returns true;

// otherwise returns false

**if** (size() < 2) {

**return** **false**;

}

T theElementInStorage = **this**.elements[topIndex];

**this**.elements[topIndex] = **this**.elements[topIndex - 1];

**this**.elements[topIndex - 1] = theElementInStorage;

**return** **true**;

}

//\*\*\* Task #5: define and implement method poptop( ): T

//\* the “classic” pop operation, if the stack is empty it throws StackUnderflowException;

//\* otherwise it both removes and returns the top element of the stack.

**public** T poptop() {

// Throws StackUnderflowException if this stack is empty,

// otherwise removes and returns top element from this stack.

**if** (isEmpty()) {

**throw** **new** StackUnderflowException(

"Exception: poptop for an improved array-based bounded stack was requested when the stack was empty."

);

}

T theTopElement = top();

pop();

**return** theTopElement;

}

}

**package** com.tsl.array\_based\_bounded\_stack;

/\*\*

\* StackInterface.java

\* Interface for a class that implements a stack of <T>.

\* A stack is a last-in, first-out structure.

\*/

**public** **interface** StackInterface<T> {

**void** push(T element) **throws** StackOverflowException;

// Throws StackOverflowException if this stack is full,

// otherwise places element at the top of this stack.

**void** pop() **throws** StackUnderflowException;

// Throws StackUnderflowException if this stack is empty,

// otherwise removes top element from this stack.

T top() **throws** StackUnderflowException;

// Throws StackUnderflowException if this stack is empty,

// otherwise returns top element of this stack.

**boolean** isEmpty();

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

**boolean** isFull();

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

}

**package** com.tsl.array\_based\_bounded\_stack;

/\*\*

\* StackOverflowException represents the structure for an exception that occurs when a push onto a stack is requested,

\* but the stack is full.

\* **@author** Tom Lever

\*

\*/

**public** **class** StackOverflowException **extends** RuntimeException {

**public** StackOverflowException() {

/\*\*

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

\* Exception's zero-parameter constructor.

\*/

**super**();

}

**public** StackOverflowException(String message) {

/\*\*

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

\* error message to Exception's one-parameter constructor with a message parameter.

\* **@param** message

\*/

**super**(message);

}

}

**package** com.tsl.array\_based\_bounded\_stack;

/\*\*

\* StackUnderflowException represents the structure for an exception that occurs when a pop off a stack is requested,

\* or a look at the top element in the stack is requested, but the stack is empty.

\* **@author** Tom Lever

\* **@since** 05/27/21

\*

\*/

**public** **class** StackUnderflowException **extends** RuntimeException {

**public** StackUnderflowException() {

/\*\*

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

\* Exception's zero-parameter constructor.

\*/

**super**();

}

**public** StackUnderflowException(String message) {

/\*\*

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

\* error message to Exception's one-parameter constructor with a message parameter.

\* **@param** message

\*/

**super**(message);

}

}

**package** com.tsl.array\_based\_bounded\_stack;

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

/\*\*

\* ImprovedArrayBoundedStackTest encapsulates JUnit tests of core methods of class ImprovedArrayBoundedStack.

\* **@author** Tom Lever

\* **@version** 1.0

\* **@since** 05/28/21

\*

\*/

**public** **class** ImprovedArrayBoundedStackTest {

@Test

**public** **void** testImprovedArrayBoundedStackByPoppingAllElements() {

/\*\*

\* testImprovedArrayBoundedStackByPoppingAllElements tests ImprovedArrayBoundedStack by correctly popping all

\* elements from a stack of type ImprovedArrayBoundedStack.

\*/

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

ImprovedArrayBoundedStack<String> myStack = **new** ImprovedArrayBoundedStack<String>(10);

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

myStack.push(String.*valueOf*(i));

}

**try** {

myStack.popSome(3);

System.***out***.println("Popped three elements.");

}

**catch** (StackUnderflowException theStackUnderflowException) {

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

}

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

}

@Test

**public** **void** testImprovedArrayBoundedStackByTryingToPopMoreThanSizeElements() {

/\*\*

\* testImprovedArrayBoundedStackByTryingToPopMoreThanSizeElements tests ImprovedArrayBoundedStack by trying to pop

\* more than size elements from a stack of type ImprovedArrayBoundedStack, where size is the number of elements on

\* the stack.

\*/

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

ImprovedArrayBoundedStack<String> myStack = **new** ImprovedArrayBoundedStack<String>(10);

**for** (**int** i = 0; i < 2; i++) {

myStack.push(String.*valueOf*(i));

}

**try** {

myStack.popSome(3);

System.***out***.println("Popped three elements.");

}

**catch** (StackUnderflowException theStackUnderflowException) {

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

}

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

}

@Test

**public** **void** testPopTopWhenStackHasTwoElements() {

/\*\*

\* testPopTopWhenStackHasTwoElements tests ImprovedArrayBoundedStack by correctly displaying an element that was

\* on top of a stack before it was popped off the stack, and the stack after the pop, when the stack had two

\* elements.

\*/

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

ImprovedArrayBoundedStack<String> myStack = **new** ImprovedArrayBoundedStack<String>(10);

myStack.push(String.*valueOf*(0));

myStack.push(String.*valueOf*(1));

**try** {

System.***out***.println(myStack.poptop() + " popped from stack. Stack is now\n" + myStack.toString());

}

**catch** (StackUnderflowException theStackUnderflowException) {

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

}

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

}

@Test

**public** **void** testPopTopWhenStackHasZeroElements() {

/\*\*

\* testPopTopWhenStackHasZeroElements tests ImprovedArrayBoundedStack by attempting to display an element that was

\* on top of a stack before it was popped off the stack, and the stack after the pop, when the stack was empty.

\*/

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

ImprovedArrayBoundedStack<String> myStack = **new** ImprovedArrayBoundedStack<String>(10);

**try** {

System.***out***.println(myStack.poptop() + " popped from stack. Stack is now\n" + myStack.toString());

}

**catch** (StackUnderflowException theStackUnderflowException) {

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

}

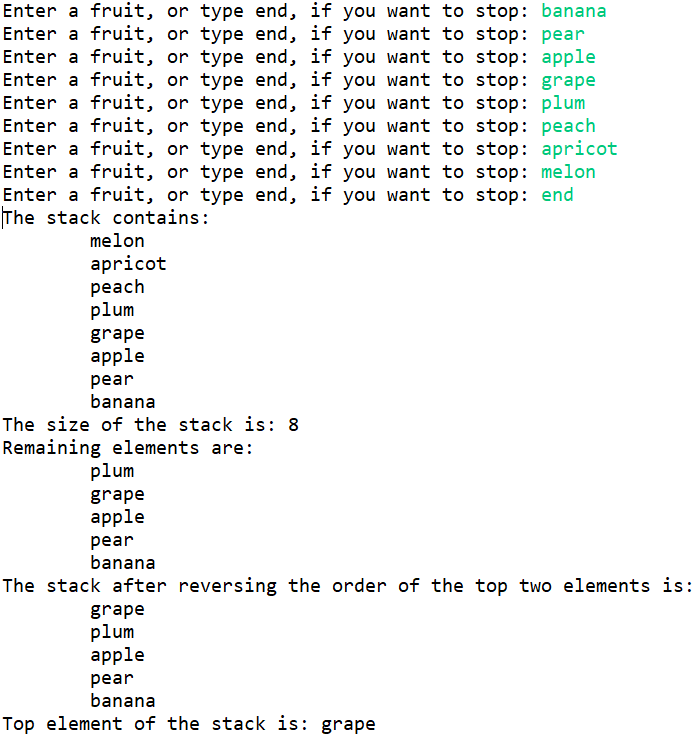
System.***out***.println();

}

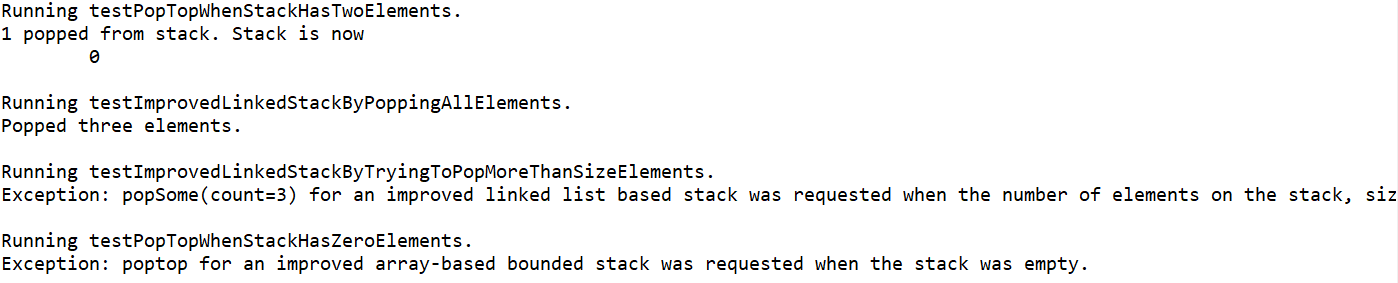
}

**4**

Main Output



Test Output



**package** com.tsl.linked\_list\_based\_bounded\_stack;

**import** java.util.Scanner;

/\*\*

\* **@author** EMILIA BUTU

\* version 1.0

\* since 2020-05

\*

\* Student name: Tom Lever

\* Completion date: 05/28/21

\*

\* ImprovedLinkedStackDriver.java: demonstrates the use of the new methods

\* from ImprovedLinkedStack class

\*

\* Student tasks: complete tasks specified in the file

\*/

**public** **class** ImprovedLinkedStackDriver {

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

/\*\*

\* main is the entry point of the program. The program requests that a user enter fruit names, followed by "end".

\* The program prints the fruit names on the stack, the size of the stack, the remaining names on the stack when

\* three names are popped from the stack, the names on the stack when the top two names on the stack are swapped,

\* and the top name that was on the stack before that name was popped from the stack. The program throws exceptions

\* if the program is terminated while input is being requested from the user, a pop of n names from the stack is

\* requested when the number of names on the stack is less than n, and when a classic pop from the stack is requested

\* when the stack is empty.

\*/

ImprovedLinkedStack<String> myStack;

myStack = **new** ImprovedLinkedStack<String>();

// prepare the Scanner object to enter data from the user

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

String word="";

// read words in a loop,

**while**(!word.equalsIgnoreCase("end"))

{

System.***out***.print("Enter a fruit, or type end, if you want to stop: ");

word=input.next();

**if**(!word.equalsIgnoreCase("end"))

myStack.push(word);

}

//\*\*\* Task #1: test method toString()

System.***out***.println("The stack contains:\n" + myStack);

//\*\*\* Task #2: test method toString()

**int** stackSize=myStack.size();

System.***out***.println("The size of the stack is: " + stackSize);

//\*\*\* Task #3: test method popSome(int count)

//\* pop three elements from the top of the stack

//\* display the resulting stack

myStack.popSome(3);

System.***out***.println("Remaining elements are: ");

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

//\*\*\* Task #4: test method swapStart()

//\* and display the resulting stack

myStack.swapStart();

System.***out***.println("The stack after reversing the order of the top two elements is: ");

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

//\*\*\* Task #5: test method poptop()

String topFruit=myStack.poptop();

System.***out***.println("Top element of the stack is: "+topFruit);

}

}

**package** com.tsl.linked\_list\_based\_bounded\_stack;

/\*\*

\* **@author** EMILIA BUTU

\* version 1.0

\* since 2020-03

\*

\* Student name: Tom Lever

\* Completion date: 05/28/21

\*

\* ImprovedLinkedStack.txt: download and save as ImprovedLinkedStack.java

\* Tasks are marked throughout the file

\*

\* Student tasks: complete tasks specified in the file

\*

\*/

**public** **class** ImprovedLinkedStack<T> **implements** StackInterface<T>

{

**protected** LLNode<T> top; // reference to the top of this stack

**public** ImprovedLinkedStack() {

/\*\*

\* ImprovedLinkedStack() is a zero-parameter constructor for ImprovedLinkedStack that sets the top linked-list node

\* of this linked list based stack to null.

\*/

top = **null**;

}

**public** **void** push(T element) {

// Places element at the top of this stack.

LLNode<T> newNode = **new** LLNode<T>(element);

newNode.setsItsReferenceTo(top);

top = newNode;

}

**public** **void** pop() {

// Throws StackUnderflowException if this stack is empty,

// otherwise removes top element from this stack.

**if** (isEmpty())

**throw** **new** StackUnderflowException("Pop attempted on an empty stack.");

**else**

top = top.getTheReferenceToTheNextLinkedListNode();

}

**public** T top() {

// Throws StackUnderflowException if this stack is empty,

// otherwise returns the information of the top element of this stack.

**if** (isEmpty())

**throw** **new** StackUnderflowException("Top attempted on an empty stack.");

**else**

**return** top.getTheInformation();

}

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

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

**return** (top == **null**);

}

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

// Returns false - a linked stack is never full

**return** **false**;

}

//\*\*\* Task #1: define method toString(): String

//\* creates and returns a string that correctly represents the current stack.

@Override

**public** String toString() {

/\*\*

\* toString arranges strings representing the information of the linked-list nodes in this stack in an indented

\* column.

\*/

String theStringRepresentingTheStack = "";

LLNode<T> theCurrentNode = top;

theStringRepresentingTheStack += "\t" + top.getTheInformation();

theCurrentNode = theCurrentNode.getTheReferenceToTheNextLinkedListNode();

**while** (theCurrentNode != **null**) {

theStringRepresentingTheStack += "\n\t" + theCurrentNode.getTheInformation();

theCurrentNode = theCurrentNode.getTheReferenceToTheNextLinkedListNode();

}

**return** theStringRepresentingTheStack;

}

//\*\*\* Task #2: define method size(): int

//\* returns a count of how many items are currently on the stack.

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

// returns a count of how many elements are on the stack

**int** theNumberOfElementsOnTheStack = 0;

LLNode<T> theCurrentNode = top;

**while** (theCurrentNode != **null**) {

theNumberOfElementsOnTheStack++;

theCurrentNode = theCurrentNode.getTheReferenceToTheNextLinkedListNode();

}

**return** theNumberOfElementsOnTheStack;

}

//\*\*\* Task #3: define method popSome(int count): void

//\* removes the top count elements from the stack

**public** **void** popSome(**int** count) {

// if possible, removes top count elements from stack;

// otherwise throws StackUnderflowException

**if** (size() < count) {

**throw** **new** StackUnderflowException(

"Exception: popSome(count=" + count + ") for an improved linked list based stack was requested when " +

"the number of elements on the stack, size = " + size() + ", was less than count = " + count + "."

);

}

**while** (count > 0) {

pop();

count--;

}

}

//\*\*\* Task #4: define method swapStart(): boolean

//\* if there are less than two elements on the stack returns false;

//\* otherwise it reverses the order of the top two elements on the stack and returns true

**public** **boolean** swapStart() {

// if possible, reverses order of top 2 elements and returns true;

// otherwise returns false

**if** (size() < 2) {

**return** **false**;

}

LLNode<T> theNodeInStorage = top.getTheReferenceToTheNextLinkedListNode();

top.setsItsReferenceTo(top.getTheReferenceToTheNextLinkedListNode().getTheReferenceToTheNextLinkedListNode());

theNodeInStorage.setsItsReferenceTo(top);

top = theNodeInStorage;

**return** **true**;

}

//\*\*\* Task #5: define method poptop( ): T

//\* the “classic” pop operation, if the stack is empty it throws StackUnderflowException;

//\* otherwise it both removes and returns the top element of the stack.

**public** T poptop() {

// Throws StackUnderflowException if this stack is empty,

// otherwise removes and returns top element from this stack.

**if** (isEmpty()) {

**throw** **new** StackUnderflowException(

"Exception: poptop for an improved array-based bounded stack was requested when the stack was empty."

);

}

T theInformationOfTheTopLinkedListNode = top();

pop();

**return** theInformationOfTheTopLinkedListNode;

}

}

**package** com.tsl.linked\_list\_based\_bounded\_stack;

/\*\*

\* StackInterface.java

\* Interface for a class that implements a stack of <T>.

\* A stack is a last-in, first-out structure.

\*/

**public** **interface** StackInterface<T> {

**void** push(T element) **throws** StackOverflowException;

// Throws StackOverflowException if this stack is full,

// otherwise places element at the top of this stack.

**void** pop() **throws** StackUnderflowException;

// Throws StackUnderflowException if this stack is empty,

// otherwise removes top element from this stack.

T top() **throws** StackUnderflowException;

// Throws StackUnderflowException if this stack is empty,

// otherwise returns top element of this stack.

**boolean** isEmpty();

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

**boolean** isFull();

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

}

**package** com.tsl.linked\_list\_based\_bounded\_stack;

/\*\*

\* StackOverflowException represents the structure for an exception that occurs when a push onto a stack is requested,

\* but the stack is full.

\* **@author** Tom Lever

\* **@version** 1.0

\* **@since** 05/28/21

\*

\*/

**public** **class** StackOverflowException **extends** RuntimeException {

**public** StackOverflowException() {

/\*\*

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

\* Exception's zero-parameter constructor.

\*/

**super**();

}

**public** StackOverflowException(String message) {

/\*\*

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

\* error message to Exception's one-parameter constructor with a message parameter.

\* **@param** message

\*/

**super**(message);

}

}

**package** com.tsl.linked\_list\_based\_bounded\_stack;

/\*\*

\* StackUnderflowException represents the structure for an exception that occurs when a pop off a stack is requested,

\* or a look at the top element in the stack is requested, but the stack is empty.

\* **@author** Tom Lever

\* **@since** 05/27/21

\*

\*/

**public** **class** StackUnderflowException **extends** RuntimeException {

**public** StackUnderflowException() {

/\*\*

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

\* Exception's zero-parameter constructor.

\*/

**super**();

}

**public** StackUnderflowException(String message) {

/\*\*

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

\* error message to Exception's one-parameter constructor with a message parameter.

\* **@param** message

\*/

**super**(message);

}

}

**package** com.tsl.linked\_list\_based\_bounded\_stack;

/\*\*

\* LLNode provides a structure for objects that represent linked-list nodes.

\* **@author** Tom Lever

\* **@version** 1.0

\* **@since** 05/28/21

\* **@param** <T>

\*/

**public** **class** LLNode<T> {

**private** T information;

/\*\*

\* information is a component of this linked-list node.

\*/

**private** LLNode<T> referenceToTheNextLinkedListNode;

/\*\*

\* referenceToTheNextLinkedListNode is a component of this linked-list node.

\*/

**public** LLNode(T theInformationToUse) {

/\*\*

\* LLNode(T theInformationToUse) is a one-parameter constructor for LLNode that sets this linked-list node's

\* information to theInformationToUse and referenceToTheNextLinkedListNode to null.

\* **@param** theInformationToUse

\*/

**this**.information = theInformationToUse;

**this**.referenceToTheNextLinkedListNode = **null**;

}

**public** **void** setsItsInformationTo(T theInformationToUse) {

/\*\*

\* setsItsInformationTo sets this linked-list node's information to theInformationToUse.

\* **@param** theInformationToUse

\*/

**this**.information = theInformationToUse;

}

**public** T getTheInformation() {

/\*\*

\* getTheInformation provides this linked-list node's information.

\* **@return**

\*/

**return** **this**.information;

}

**public** **void** setsItsReferenceTo(LLNode<T> theReference) {

/\*\*

\* setsItsReferenceTo sets this linked-list node's reference to TheNext linked-list node to theReference.

\* **@param** theReference

\*/

**this**.referenceToTheNextLinkedListNode = theReference;

}

**public** LLNode<T> getTheReferenceToTheNextLinkedListNode() {

/\*\*

\* references provides the linked-list node that this linked-list node references.

\* **@return**

\*/

**return** **this**.referenceToTheNextLinkedListNode;

}

}

**package** com.tsl.linked\_list\_based\_bounded\_stack;

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

/\*\*

\* ImprovedArrayBoundedStackTest encapsulates JUnit tests of core methods of class ImprovedArrayBoundedStack.

\* **@author** Tom Lever

\* **@version** 1.0

\* **@since** 05/28/21

\*

\*/

**public** **class** ImprovedLinkedStackTest {

@Test

**public** **void** testImprovedLinkedStackByPoppingAllElements() {

/\*\*

\* testImprovedLinkedStackByPoppingAllElements tests ImprovedLinkedStack by correctly popping all elements from a

\* stack of type ImprovedLinkedStack.

\*/

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

ImprovedLinkedStack<String> myStack = **new** ImprovedLinkedStack<String>();

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

myStack.push(String.*valueOf*(i));

}

**try** {

myStack.popSome(3);

System.***out***.println("Popped three elements.");

}

**catch** (StackUnderflowException theStackUnderflowException) {

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

}

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

}

@Test

**public** **void** testImprovedLinkedStackByTryingToPopMoreThanSizeElements() {

/\*\*

\* testImprovedLinkedStackByTryingToPopMoreThanSizeElements tests ImprovedLinkedStack by trying to pop more than

\* size elements from a stack of type ImprovedLinkedStack, where size is the number of elements on the stack.

\*/

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

ImprovedLinkedStack<String> myStack = **new** ImprovedLinkedStack<String>();

**for** (**int** i = 0; i < 2; i++) {

myStack.push(String.*valueOf*(i));

}

**try** {

myStack.popSome(3);

System.***out***.println("Popped three elements.");

}

**catch** (StackUnderflowException theStackUnderflowException) {

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

}

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

}

@Test

**public** **void** testPopTopWhenStackHasTwoElements() {

/\*\*

\* testPopTopWhenStackHasTwoElements tests ImprovedLinkedStack by correctly displaying an element that was

\* on top of a stack before it was popped off the stack, and the stack after the pop, when the stack had two

\* elements.

\*/

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

ImprovedLinkedStack<String> myStack = **new** ImprovedLinkedStack<String>();

myStack.push(String.*valueOf*(0));

myStack.push(String.*valueOf*(1));

**try** {

System.***out***.println(myStack.poptop() + " popped from stack. Stack is now\n" + myStack.toString());

}

**catch** (StackUnderflowException theStackUnderflowException) {

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

}

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

}

@Test

**public** **void** testPopTopWhenStackHasZeroElements() {

/\*\*

\* testPopTopWhenStackHasZeroElements tests ImprovedLinkedStack by attempting to display an element that was

\* on top of a stack before it was popped off the stack, and the stack after the pop, when the stack was empty.

\*/

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

ImprovedLinkedStack<String> myStack = **new** ImprovedLinkedStack<String>();

**try** {

System.***out***.println(myStack.poptop() + " popped from stack. Stack is now\n" + myStack.toString());

}

**catch** (StackUnderflowException theStackUnderflowException) {

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

}

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

}

}