

Name-Abhi Mehta

Batch A: IT

REG NO: 221080001

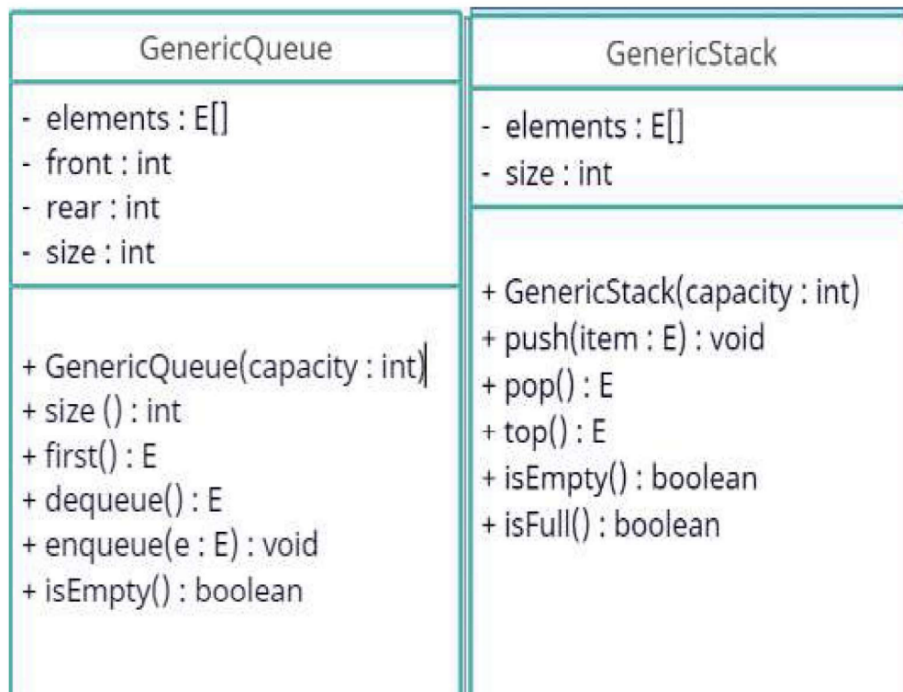
FDS LAB ASSIGNMENT 4

Laboratory-4

AIM: 1. Array based implementation of Java Stack and Queue Java API using generic classes

2. Implementation of applications discussed in the class using Stack and Queue classes implemented above

CLASS DIAGRAM:



GenericQueue

Class:

1) GenericQueue(int capacity):

- Constructor :
- Creates a new instance of the GenericQueue class with a specified capacity.
- Parameters :
 - ``capacity` (int)`: The maximum number of elements the queue can hold.

This constructor initializes the internal array elements to store the elements, sets ``front`` to 0 (indicating the start of the queue), ``rear`` to 1 (indicating an empty queue), and ``size`` to 0 (indicating that there are no elements in the queue yet).

2) public void enqueue(E e):

- Method to Enqueue an Element.
- Adds the specified element ``e`` to the rear of the queue.
- Parameters:
 - `e E`: The element to be enqueued.
- Throws an `'IllegalStateException'` if the queue is full.

3) public E dequeue():

- Method to Dequeue an Element
 - Removes and returns the element at the front of the queue.
 - Returns ``null`` if the queue is empty.
 - Returns the element that was removed from the queue.

- After dequeuing, the `front` index is incremented to point to the next element in the queue.

4) public E first():

- Method to Retrieve the First Element.
- Returns the element at the front of the queue without removing it.
- Returns `null` if the queue is empty.

5) public int size():

Method to Get the Current Size.

Returns the number of elements currently in the queue.

6) public boolean isEmpty():

- Method to Check if the given Queue is Empty.
- Returns `true` if the queue is empty (contains no elements), otherwise `false`.

GenericQueueApplication Class:

1) public static void main(String[] args):

- Main Method for the Application.
- The entry point of the application.
- Initializes a `Scanner` object to read user input from the console.
- Presents a menu to the user with various options for interacting with the GenericQueue object.
- The menu options include:
 - o Enqueue an element.
 - o Dequeue an element.
 - o Retrieve the first element.

- Check if the queue is empty.
- Get the size of the queue.
- Exit the program.
- Accepts user input to perform the selected operation.
- Continues to execute until the user chooses to exit (selecting option 0).
- Performs the selected operation based on user input and displays appropriate messages or results.

CODE:

IN Generic Queue.java file:

```
/**
 * A generic queue data structure that allows enqueueing
 * and dequeuing elements.
 * The queue has a specified capacity, and it supports
 * checking if it's empty,
 * finding the first element, and retrieving its size.
 */
 *
 * @param <E> the type of elements stored in the queue.
 * @author MehtaAbhi
 */
class GenericQueue<E> {
/**
 * An array to store the elements in the queue.
 */
private final E[] elements;
/**
 * Index representing the front of the queue.
 */
private int front = 0;
/**
 * Index representing the rear of the queue.
 */
private int rear = -1;
/**
 * The current size of the queue.
 */
private int size = 0;
/**
 * Creates a new GenericQueue with the specified
 * capacity.
 */
}
```

```

*
@param capacity the maximum number of elements the
queue can hold.
*/
public GenericQueue(int capacity) {
    elements = (E[]) new Object[capacity];
}
/**
 * Enqueues an element into the queue.
 *
 * @param e the element to be enqueued.
 *
 * @throws IllegalStateException if the queue is full.
 */
public void enqueue(E e) throws IllegalStateException {
    if (size == elements.length) throw new
    IllegalStateException("Queue is full");
    rear = (rear + 1) % elements.length;
    elements [rear] = e;
    size++;
}
/**
 *
 * Dequeues and removes the element at the front of the
queue.
 *
 *
 *
 * @return the element that was dequeued, or null if the
queue is empty.
 */
public E dequeue() {
    if (isEmpty()) return null;
    E answer = elements [front];
    elements [front] = null;
    front =(front + 1) % elements.length;
    size--;
    return answer;
}
/**
 * Retrieves the first element in the queue without
removing it.
 *
 *
 *
 * @return the first element in the queue, or null if the
queue is empty.
 */
public E first() {

```

```

if (isEmpty()) return null;
return elements [front];
}
/**
 * Gets the current size of the queue.
 *
 * @return the number of elements currently in the queue.
 */
public int size() {
return size;
}
/**
 * Checks if the queue is empty.
 *
 * @return true if the queue is empty, false otherwise.
 */
public boolean isEmpty() {
return size == 0;
}
}

```

IN Generic Queue.java

Application:

```

import java.util.*;
/**
 * An application for the GenericQueue class, allowing
 the user to interact with a queue.
 */
public class GenericQueueApplication {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Define the limit for the queue: ");
int capacity= sc.nextInt();
GenericQueue<Integer> queue = new
GenericQueue<>(capacity);
int select;
do {
System.out.print(
"Menu:\n" +
"1. Enqueue Element\n" +
"2. Dequeue Element\n" +
"3. First Element\n" +
"4. Check if Empty\n" +
"5. Size of Queue\n" +
"O. Exit Program\n" +
"Enter Operation: ");
select =

```

```
sc.nextInt();
switch (select) {
case 1:
System.out.print("Please input the element you wish to
enqueue: ");
int temp = sc.nextInt();
queue.enqueue(temp);
break;
case 2:
System.out.println("The dequeued element is: " +
queue.dequeue());
break;
case 3:
System.out.println("The element at the front is: " +
queue.first());
break;
case 4:
if (queue.isEmpty()) {
System.out.println("The Queue is Empty");
} else {
System.out.println("The Queue is not Empty");
}

break;
case 5:
System.out.println("Size of Queue is: " + queue.size());
break;
case 0:
System.out.println("Exited program successfully");
break;

default:
throw new IllegalArgumentException("Invalid Option");
}
} while (select != 0);
sc.close();
}
}
```

OUTPUT:

```
Define the limit for the queue: 10000
Menu:
1. Enqueue Element
2. Dequeue Element
3. First Element
4. Check if Empty
5. Size of Queue
0. Exit Program
Enter Sr. No. of Operation: 1
Please input the element you wish to enqueue: 4
Menu:
1. Enqueue Element
2. Dequeue Element
3. First Element
4. Check if Empty
5. Size of Queue
0. Exit Program
Enter Sr. No. of Operation: 3
The element at the front is: 4
Menu:
1. Enqueue Element
2. Dequeue Element
3. First Element
4. Check if Empty
5. Size of Queue
0. Exit Program
Enter Sr. No. of Operation: 5
Size of Queue is: 1
Menu:
1. Enqueue Element
2. Dequeue Element
3. First Element
4. Check if Empty
5. Size of Queue
0. Exit Program
Enter Sr. No. of Operation: 2
The dequeued element is: 4
Menu:
1. Enqueue Element
2. Dequeue Element
3. First Element
4. Check if Empty
5. Size of Queue
0. Exit Program
Enter Sr. No. of Operation: 4
The Queue is Empty
Menu:
1. Enqueue Element
2. Dequeue Element
3. First Element
4. Check if Empty
5. Size of Queue
0. Exit Program
Enter Sr. No. of Operation: 0
Exited program successfully
```


STACK:

GenericStack Class:

1) GenericStack<E>:

- This is a generic class that can hold elements of any data type specified by the type parameter `<E>`.
- `private final E[] elements`: An array to store the elements in the stack. It is of generic type `E`.
- `private int top`: An index pointing to the top element of the stack.

2) GenericStack(int capacity):

- Constructor: Creates a new GenericStack instance with the specified capacity.
- Parameters:
 - `capacity (int)`: The maximum number of elements the stack can hold.
- This constructor initializes the internal array elements to store the elements and sets `top` to 1, indicating an empty stack.

3) public void push(E item):

- Method to Push an Element onto the Stack: Adds the specified element `item` to the top of the stack.
- Parameters:
 - `item (E)`: The element to be pushed onto the stack.
- Throws an `IndexOutOfBoundsException` if the stack is already full.

4) public E pop():

- Method to Pop the Top Element from the Stack: Removes and returns the element at the top of the stack.

Returns the element that was removed from the stack.

- After popping, the `top` index is decremented to point to the new top element.

5) public E top():

- Method to Peek at the Top Element: Retrieves the top element of the stack without removing it.
- Returns the top element of the stack.

6) public boolean isEmpty():

- Method to Check if the Stack is Empty: Returns true if the stack is empty (contains no elements), otherwise false.

7) public boolean isFull():

Method to Check if the Stack is Full: Returns true if the stack is full (has reached its capacity), otherwise false.

GenericStackApplication Class:

1) public static void main(String[] args):

- Main Method for the Application: The entry point of the application.
- Initializes a `Scanner` object to read user input from the console.
- Presents a menu to the user with various options for interacting with the GenericStack object.
- The menu options include:
 - Push an element onto the stack.
 - Pop an element from the stack.
 - Peek at the top element of the stack.

- Check if the stack is empty.
- Check if the stack is full.
- Exit the program.
- Accepts user input to perform the selected operation.
- Continues to execute until the user chooses to exit (selecting option 0).
- Performs the selected operation based on user input and displays appropriate messages or results.

CODE:

In Generic Stack.java

```
/**
 * A generic stack data
 * structure that allows pushing
 * and popping elements.
 *
 * The stack has a specified
 * capacity, and it supports
 * checking if it's empty or full.
 *
 *
 *
 * @param <E> the type of
 * elements stored in the stack.
 */
class GenericStack <E>
{
/**
 * An array to store the
 * elements in the stack.
```

```
*/
```

```
private final E[] elements;
```

```
/**
```

```
 * Index pointing to the top  
element of the stack.
```

```
*/
```

```
private int top;
```

```
/**
```

```
 * Creates a new  
GenericStack with the  
specified capacity.
```

```
 *
```

```
 *
```

```
 *
```

```
@param capacity the  
maximum number of  
elements the stack can hold.
```

```
*/
```

```
public GenericStack (int  
capacity)
```

```
{
```

```
elements = (E[])new  
Object[capacity];
```

```
top = -1;
```

```
}
```

```
/**
```

```
* Pushes an element onto  
the stack.
```

```
*
```

```
* @param item the element  
to be pushed onto the stack.
```

```
* @throws  
IndexOutOfBoundsException  
if the stack is full.
```

```
*/
```

```
public void push (E item)
```

```
{
```

```
top++;
```

```
if (top== elements.length) {
```

```
throw new  
IndexOutOfBoundsException  
("Stack is Full");
```

```
}
```

```
elements[top] = item;
```

```
}
```

```
/**
```

```
*
```

```
*
```

```
Pops the top element from  
the stack.
```

*

@return the element that
was removed from the top of
the stack.

*/

public E pop () {

top--;

return elements [top + 1];

}

/**

*

*

Tops at the top element of
the stack without removing it.

*

@return the top element of
the stack.

{

*/

public E Top () {

return elements[top];

}

/**

* Checks if the stack is
empty.

*

* @return true if the stack is
empty, false otherwise.

*/

public boolean isEmpty()

{

return top == -1;

}

/**

* Checks if the stack is full.

*

*

*

@return true if the stack is
full, false otherwise.

*/

public boolean isFull ()

{

return top ==

elements.length - 1;

}

```
}
```

IN GenericStack Application:

```
import java.util.*;

/**
 * A test application for
 * the GenericStack class,
 * allowing the user to
 * interact with a stack.
 *
 *
 *
 *
 * @author MehtaAbhi
 */

public class
GenericStackApplicatio
n{

    public static void main
    (String[] args)

    {

        Scanner sc = new
        Scanner (System.in);

        System.out.print
        ("Define the limit for the
        stack: ");

        int capacity =
        sc.nextInt();

        GenericStack <Integer>
        stack = new
```



```
GenericStack <>
(capacity);

int select;

do

{

System.out.print
("Menu:\n" +

"1. Push Element\n" +

"2. Pop Element\n" +

"3. Top Element\n" +

"4. Check if Empty\n" +

"5. Check if Full\n" +

"O. Exit Program\n" +

"Enter Sr. No. of
Operation: ");

select = sc.nextInt();

switch (select)

{

case 1:

System.out.print
("Please input the
element you wish to
push: ");

int temp = sc.nextInt();

stack.push (temp);

break;

case
2: System.out.println
("The element popped
is: " + stack.pop());

break;

case
```

```
3: System.out.println  
("The element at the top  
is: " + stack.Top ());
```

```
break;
```

```
case 4: if  
(stack.isEmpty())
```

```
{
```

```
System.out.println ("The  
Stack is Empty");
```

```
}
```

```
else
```

```
{
```

```
System.out.println ("The  
Stack is not Empty");
```

```
}
```

```
break;
```

```
case 5:
```

```
if (stack.isFull ())
```

```
{
```

```
System.out.println ("The  
Stack is Full");
```

```
}
```

```
else
```

```
{
```

```
System.out.println ("The  
Stack is not Full");
```

```
}
```

```
break;
```

```
case 0:
```

```
System.out.println  
("Exited program  
successfully");
```

```

break;

default:

throw new
IllegalArgumentException
n ("Invalid Option");

}

}

while (select != 0);

sc.close();

}

}

```

OUTPUT:

```

Define the limit for the stack: 10000
Menu:
1. Push Element
2. Pop Element
3. Top Element
4. Check if Empty
5. Check if Full
0. Exit Program
Enter Sr. No. of Operation: 1
Please input the element you wish to push: 4
Menu:
1. Push Element
2. Pop Element
3. Top Element
4. Check if Empty
5. Check if Full
0. Exit Program
Enter Sr. No. of Operation: 3
The element at the top is: 4
Menu:
1. Push Element
2. Pop Element
3. Top Element
4. Check if Empty
5. Check if Full
0. Exit Program
Enter Sr. No. of Operation: 2
The element popped is: 4
Menu:
1. Push Element
2. Pop Element
3. Top Element
4. Check if Empty
5. Check if Full
0. Exit Program
Enter Sr. No. of Operation: 4
The Stack is Empty
Menu:
1. Push Element
2. Pop Element
3. Top Element
4. Check if Empty
5. Check if Full
0. Exit Program
Enter Sr. No. of Operation: 5
The Stack is not Full
Menu:
1. Push Element
2. Pop Element
3. Top Element
4. Check if Empty
5. Check if Full
0. Exit Program
Enter Sr. No. of Operation: 0
Exited program successfully

```

IMPLEMENTATION OF APPLICATION OF STACK:

```
/**
 * Reverse WordWithStack is a Java
 program that reverses a word entered
 by the
 user
 * using a stack data structure. It reads a
 word from the user, pushes its
 characters
 * onto a stack, and then pops them off
 to build the reversed word.
 *
 * @author MehtaAbhi
 * @version 1.0
 */
import java.util.Scanner; // Importing
the Scanner class for input reading
import java.util.Stack; /** It is a class
that implements the stack data
structure.
It extends the Vector class and
provides a Last-In-First-Out (LIFO)
data structure,
where the most recently added element
is the first one to be removed.
*/

public class Reverse {
    /**
     * The main method of the program,
     responsible for reading a word,
     reversing it, and printing the result.
     *
     * @param args The command-line
     arguments (not used in this program).
     */
    public static void main(String[]
args) {
        // Create a Scanner object for user
input
        Scanner scanner = new
Scanner(System.in);

        // Prompt the user to enter a word
        System.out.print("Enter a word:
");

        // Read the user's input as a string
```

```

String word = scanner.nextLine();

// Call the reverseWord function
to reverse the input word
String reversedWord =
reverseWord(word);

// Display the reversed word on
the console
System.out.println("Reversed
word: " + reversedWord);

// Close the Scanner to release
resources
scanner.close();
}

public static String
reverseWord(String word) {
    // Create a Stack to hold
characters
    Stack<Character> stack = new
Stack<>();

    // Convert the input word into an
array of characters
    char[] characters =
word.toCharArray();

    // Push each character onto the
stack
    for (char c : characters) {
        stack.push(c);
    }

    // Create a StringBuilder to build
the reversed word
    StringBuilder reversedWord =
new StringBuilder();

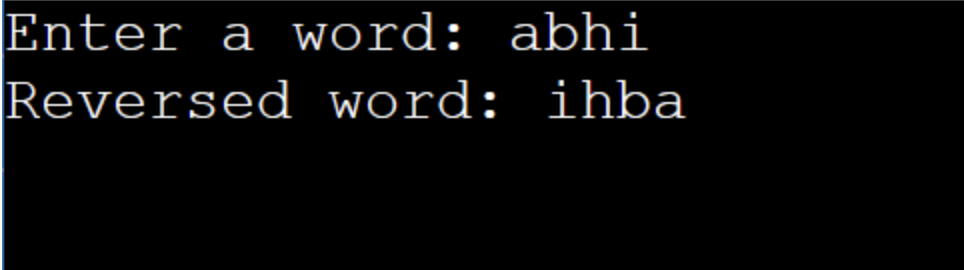
    // Pop characters from the stack
and append them to the reversedWord
    while (!stack.isEmpty()) {
        reversedWord.append(stack.pop());
    }

    // Return the reversed word as a

```

```
string  
    return reversedWord.toString();  
}  
}
```

OUTPUT:

A terminal window with a black background and light blue text. The first line shows the prompt 'Enter a word:' followed by the input 'abhi'. The second line shows the output 'Reversed word: ihba'.

```
Enter a word: abhi  
Reversed word: ihba
```

LEARNINGS:

1) Data Structures:

- I acquired a deeper comprehension of foundational data structures like stacks and queues, which have widespread applications in computer science and software development.

2) Generics:

- I discovered the art of designing and utilizing generic classes in Java. Generics empower us to work with diverse data types while maintaining type safety.

3) Array Implementation:

- I become well-acquainted with the art of implementing data structures using arrays. Proficiency in managing array elements to replicate the functionality of stacks and queues becomes a fundamental skill.

4) Error Handling:

- I encountered scenarios where mastering error handling is indispensable, such as managing full queues or empty stacks.
- I gained proficiency in throwing and handling exceptions, enhancing the resilience of my programs.

5) User Interaction:

- I cultivated expertise in constructing text-based user interfaces for interacting with data structures.
- This valuable experience can be applied to crafting user-friendly command line applications.

6) Menu Driven Programs:

I became adept at crafting menu driven programs that empower users to select various operations from a menu and execute them according to their preferences.

7) Input Validation:

I implemented robust input validation techniques to ensure that user supplied input adheres to anticipated ranges or formats, thereby elevating the reliability of my programs.

8) Algorithmic Thinking:

- I Implemented operations like string reversal (added to the menu) necessitates algorithmic thinking and problem-solving skills.
- This showcases how data structures can be harnessed to tackle real-world problems.

9) Java

API:

- I harnessed Java's Scanner class to read user input, garnering Hands-on experience with the Java API for common programming tasks.

ERRORS:

- Errors were introduced due to typos and omissions in the program's menu
- Syntax errors in the generic class implementation.

CONCLUSION:

In this laboratory experiment, we delved into the fundamental linear data structures, namely stacks and queues, and unveiled their crucial functions in a range of industrial applications as indispensable data storage systems. To enhance flexibility, we harnessed the capabilities of Generic classes, allowing us to work effortlessly with various data types. This versatility proved to be extremely valuable. As a component of the experiment, we developed a text-based application to thoroughly evaluate our implementations. This practical activity not only reinforced our understanding of the principles but also familiarized us with using the Java API.