ASSIGNMENT 2 FRONT SHEET

Qualification	**BTEC Level 5 HND Diploma in Computing**		
Unit number and title	Unit 19: Data Structures and Algorithms		
Submission date		**Date Received 1st submission**	
Re-submission Date		**Date Received 2nd submission**	
Student Name	LE VAN NHAT	**Student ID**	BHAF190185
Class	BH-AF-2005-2.3	**Assessor name**	NGO THI MAI LOAN

Student declaration

I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. I understand that making a false declaration is a form of malpractice.

Student's signature	LE VAN NHAT

Grading grid

P4	P5	P6	P7	M4	M5	D3	D4

❒ Summative Feedback:	❒ Resubmission Feedback:	
Grade:	Assessor Signature:	Date:
Internal Verifier's Comments:		
IV Signature:		

Table of Contents

A. INTRODUCTION **5**

B. CONTENTS **5**

I. IMPLEMENT A COMPLEX ADT AND ALGORITHM IN AN EXECUTABLE PROGRAMMING LANGUAGE TO SOLVE A WELL-DEFINED PROBLEM (P4) **5**

1. Flowchart *6*

2. Design the ADT and algorithm *9*

II. IMPLEMENT ERROR HANDLING AND REPORT TEST RESULTS (P5) **18**

1. Errror handling *18*

2. Test case *23*

III. DISCUSS HOW ASYMPTOTIC ANALYSIS CAN BE USED TO ASSESS THE EFFECTIVENESS OF AN ALGORITHM (P6) **24**

1. What is Algorithmic Analysis? *24*

2. What is Asymptotic Analysis? *30*

3. General Rules for Estimation *35*

IV. DETERMINE TWO WAYS IN WHICH THE EFFICIENCY OF AN ALGORITHM CAN BE MEASURED, ILLUSTRATING YOUR ANSWER WITH AN EXAMPLE (P7) **38**

1. Time complexity *39*

2. Spatial Complexity *42*

C. CONCLUDED **45**

D. REFERENCES **45**

A. INTRODUCTION

In this assignment, my report includes the following sections:

- First, I will implement a complex ADT and algorithm in a programming language (Java) to solve a well-defined problem.
- Second, I will implement error handling and report the test results.
- Third, I will explain how asymptotic analysis can be used to evaluate the effectiveness of an algorithm.
- Finally, I will give two possible ways of measuring the effectiveness of an algorithm with specific examples.

B. CONTENTS

I. Implement a complex ADT and algorithm in an executable programming language to solve a well-defined problem (P4)

The given ***scenario*** is:

"For the middleware that is currently developing, one part of the provisioning interface is how the message can be transferred and processed through layers. For transport, normally a buffer of queue messages is implemented and for processing, the systems require a stack of messages.

The team now has to develop these kinds of collections for the system. They should design ADT / algorithms for these 2 structures and implement a demo version with the message is a string of a maximum of 250 characters. The demo should demonstrate some important operations of these structures. Even it's a demo, errors should be handled carefully by exceptions and some tests should be executed to prove the correctness of algorithms/operations." Based on the given scenario, we can determine that we need some suitable ADTs to solve this problem.

They are stack and queue, which are defined in Assignment 1.

- Stack is used to storing the process messages.
- The queue is used to store the transported messages.

- *A stack* is a linear data structure that can only be accessed at one of its ends for storage and retrieve data. Such a stack is like a stack of trays in the cafeteria: the new trays are put on the top of the stack and take out top. The last tray put on the stack is the first tray removed from the stack. Hence, it is called the Last in First out (LIFO) or First in Last out (FILO) list. Thus, Stack is used to storing the process messages.
- *A queue* is simply a queue that grows by adding an element at the end and shrinking by talking elements from its front. Unlike a stack, a queue is a structure in which both ends are used: one to add new elements and one for removing them. So, the last element has to wait until all elements preceding it on the queue are deleted. Hence, it is called the First in First out (FIFO) or Lasts on Last out (LILO) list. Thus, Queue is used to store

the transported messages. The message will be added to the queue, if the queue is full then the message will not be added to the queue.

1. **Flowchart**

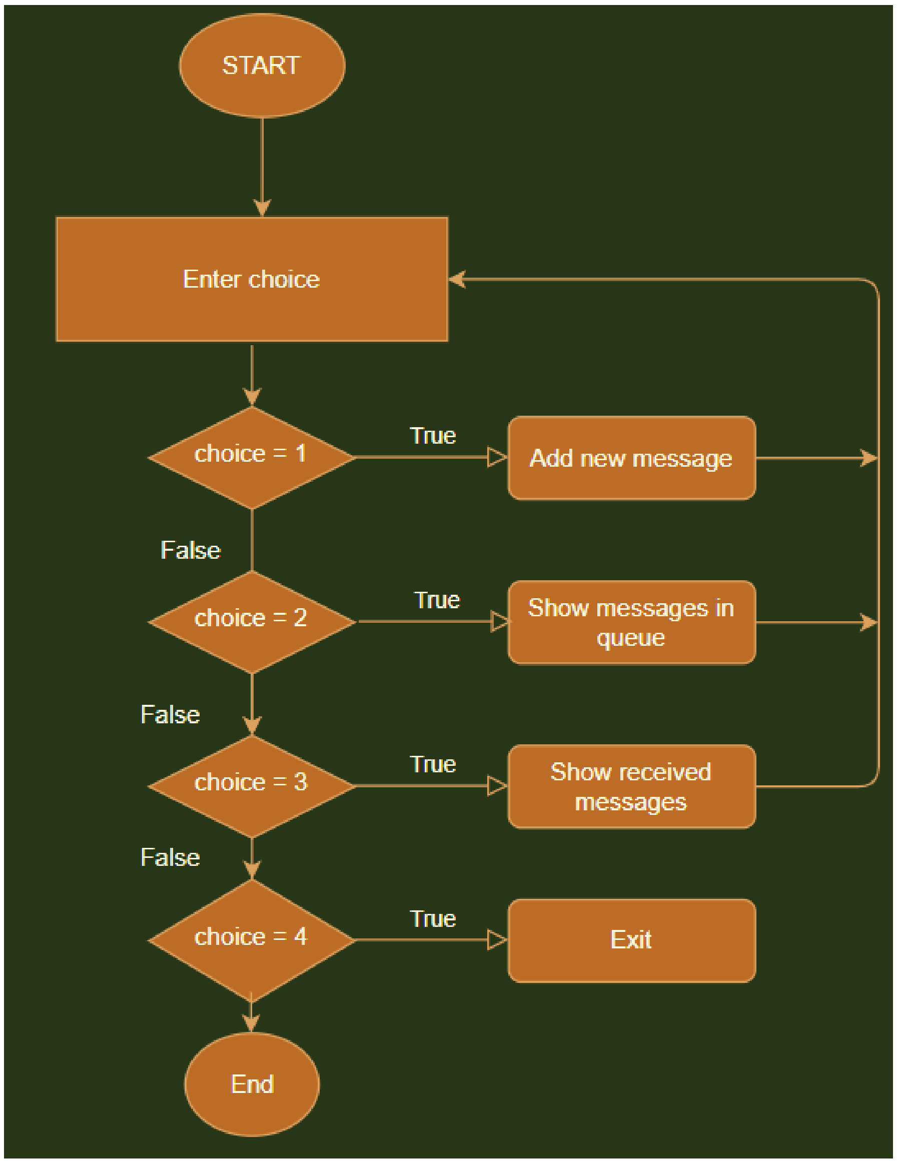


Figure 1: Main program flowchart

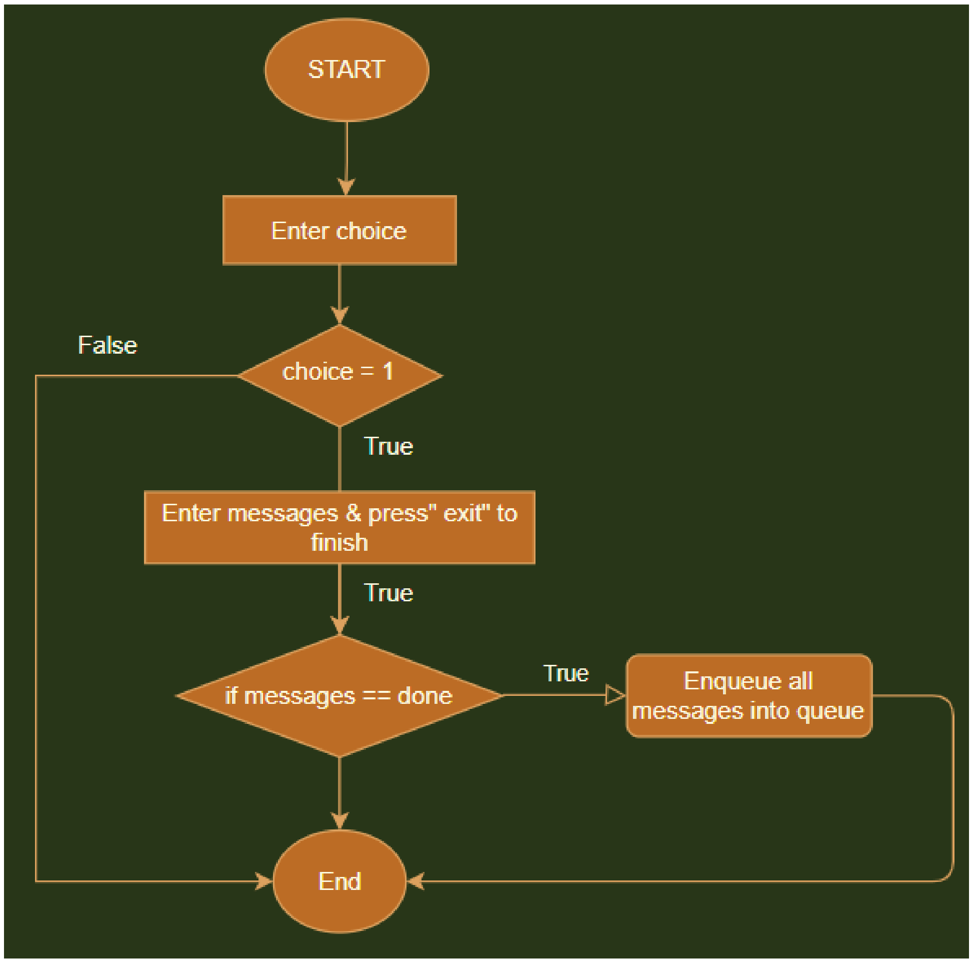

START
Enter choice
False
choice = 1
True
Enter messages & press" exit" to finish
True
if messages == done
True
Enqueue all messages into queue
End

Figure 2: Choice 1 flowchart

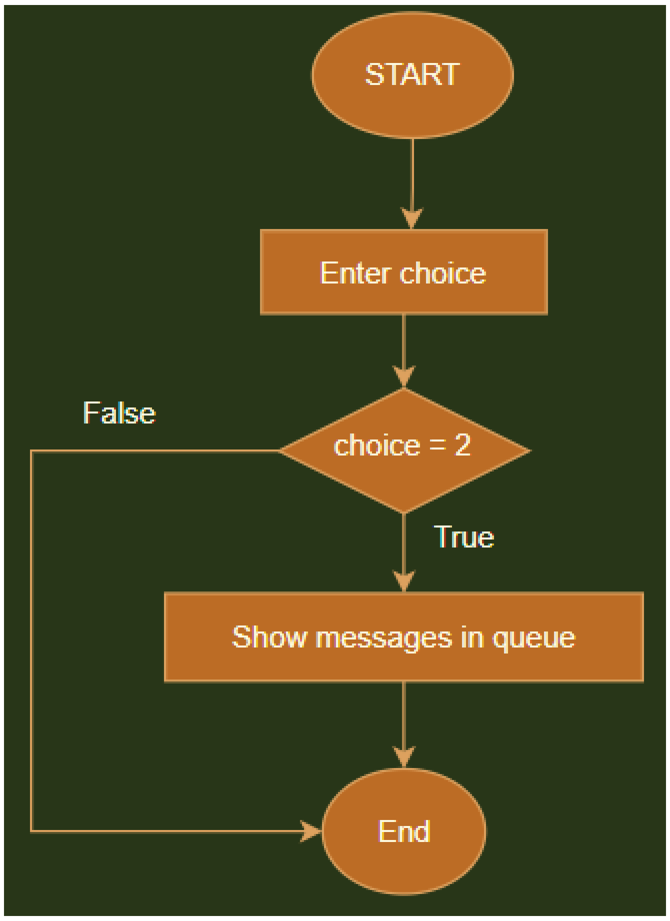


Figure 3: Choice 2 flowchart

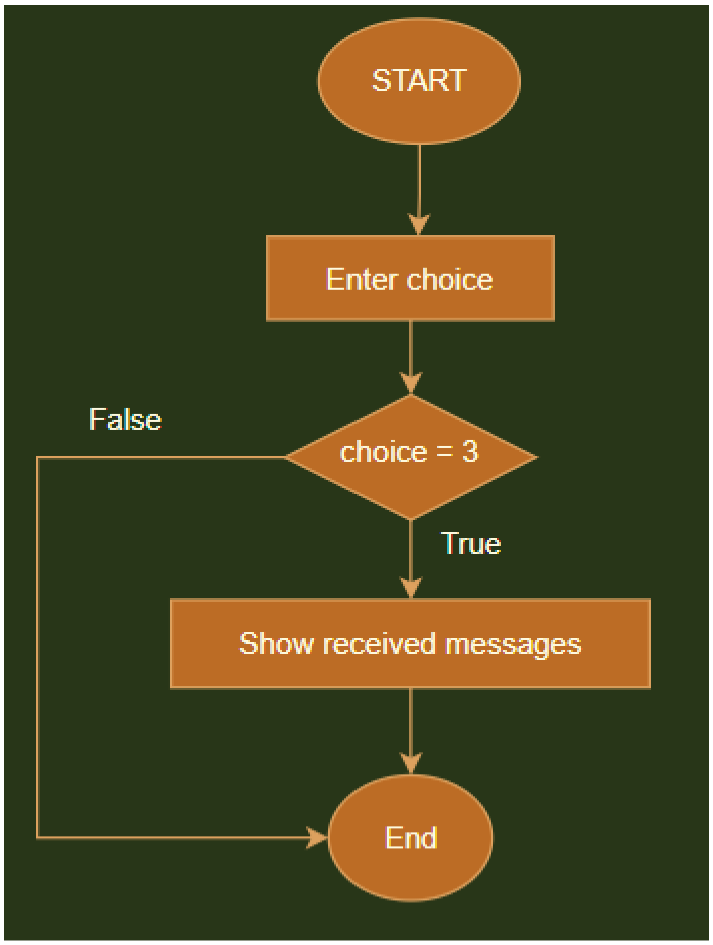
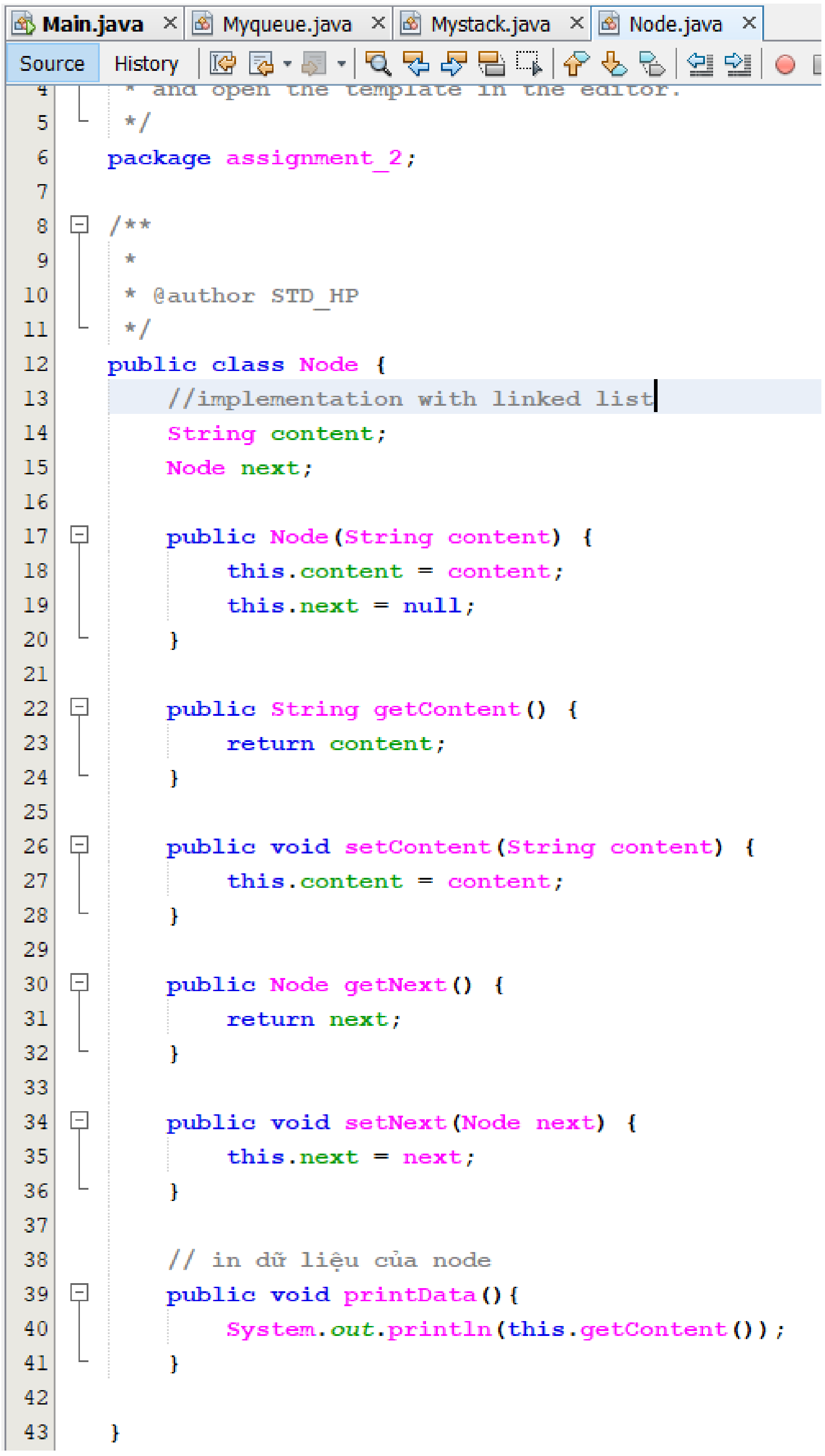


Figure 4: Choice 3 flowchart

2. Design the ADT and algorithm

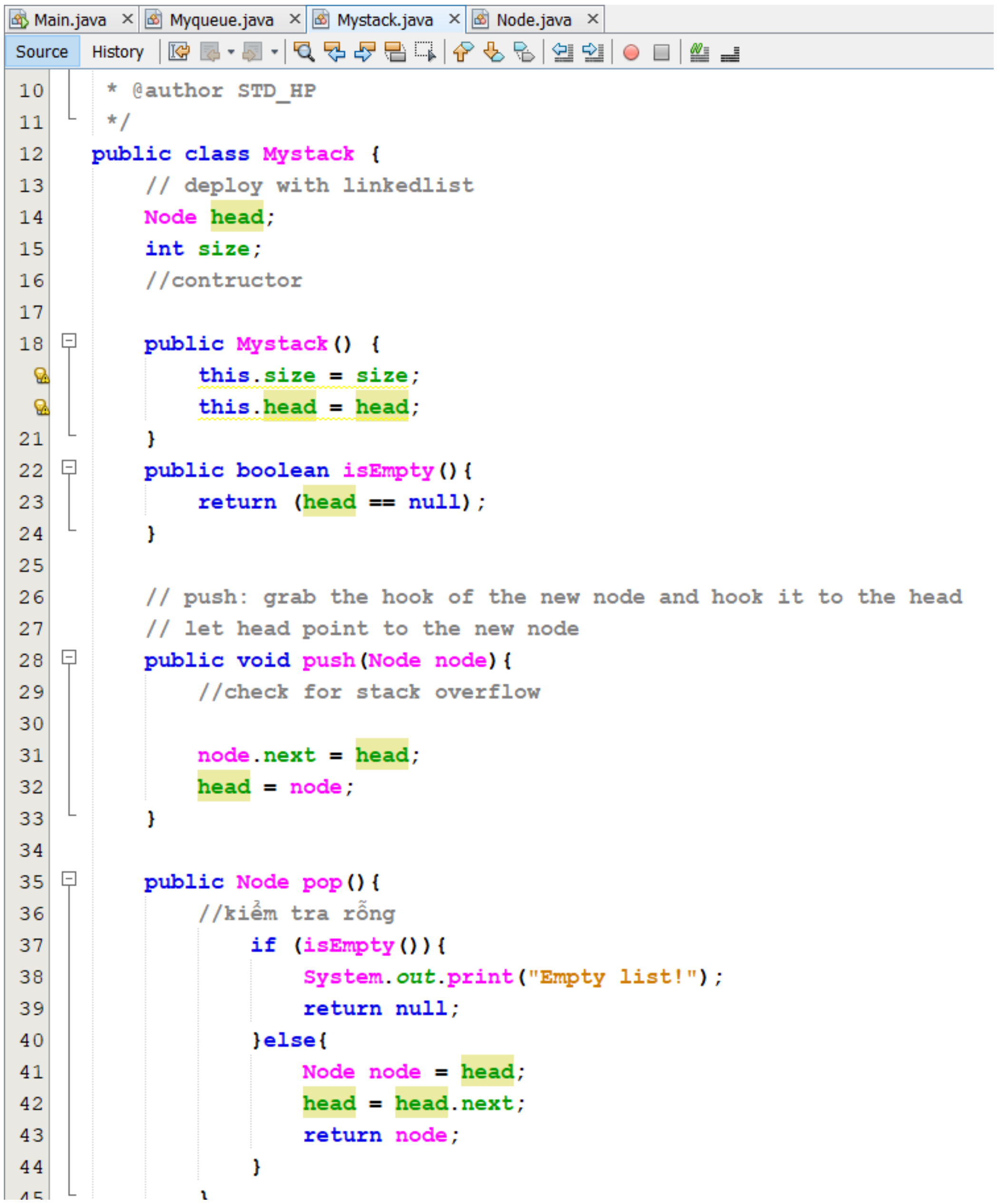
- To solve this problem, I have designed Stack and Queue ADT for handling the requirements as below:
    - First, I have designed a "Node" object to store the messages in the system



```java
 * and open the template in the editor.
 */
package assignment_2;

/**
 *
 * @author STD_HP
 */
public class Node {
    //implementation with linked list
    String content;
    Node next;

    public Node(String content) {
        this.content = content;
        this.next = null;
    }

    public String getContent() {
        return content;
    }

    public void setContent(String content) {
        this.content = content;
    }

    public Node getNext() {
        return next;
    }

    public void setNext(Node next) {
        this.next = next;
    }

    // in dữ liệu của node
    public void printData(){
        System.out.println(this.getContent());
    }

}
```

Figure 5: The "Main" class

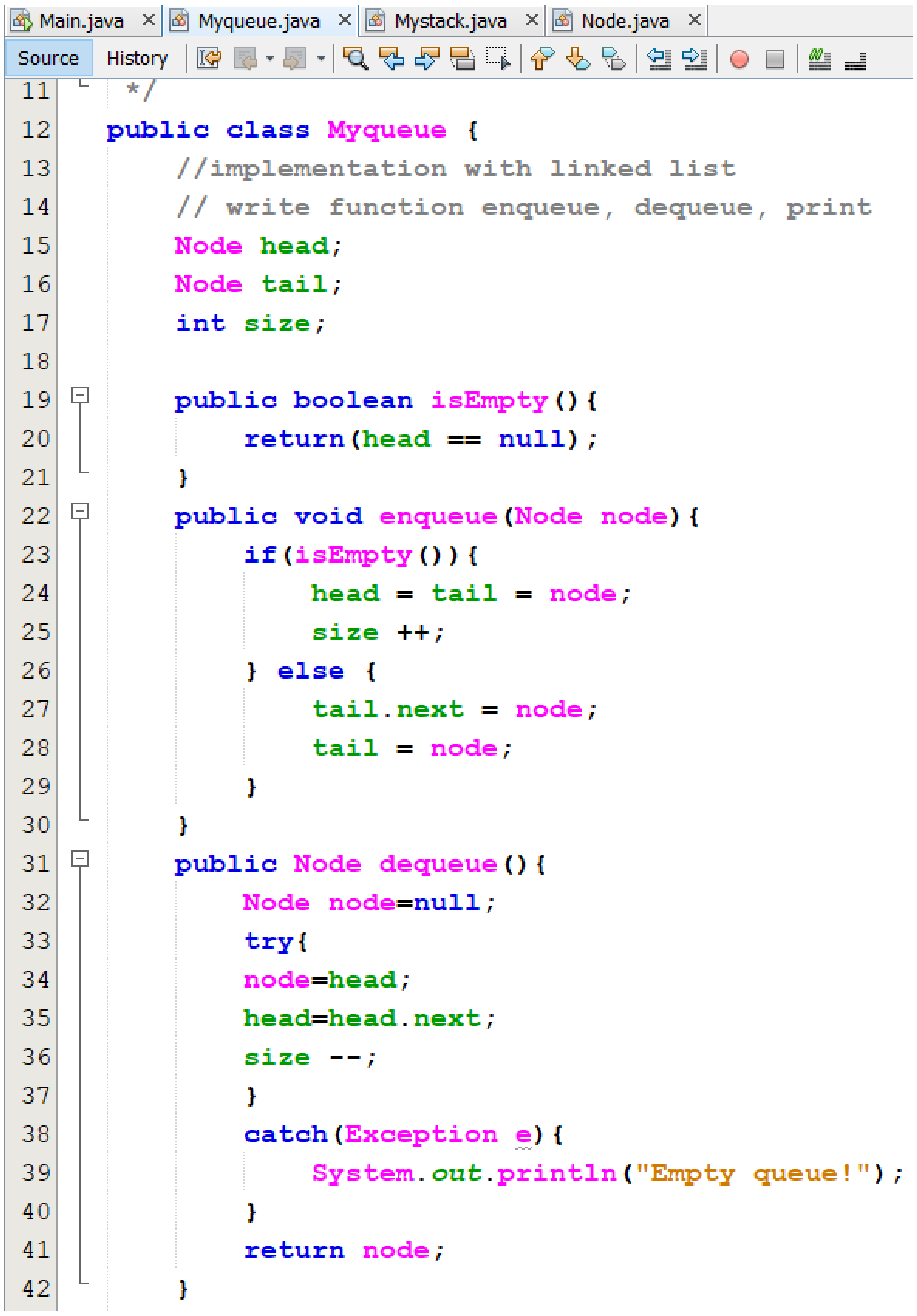
- Next, I will design the stack ADT for the system.



```
 * @author STD_HP
 */
public class Mystack {
    // deploy with linkedlist
    Node head;
    int size;
    //contructor

    public Mystack() {
        this.size = size;
        this.head = head;
    }
    public boolean isEmpty(){
        return (head == null);
    }

    // push: grab the hook of the new node and hook it to the head
    // let head point to the new node
    public void push(Node node){
        //check for stack overflow

        node.next = head;
        head = node;
    }

    public Node pop(){
        //kiểm tra rỗng
            if (isEmpty()){
                System.out.print("Empty list!");
                return null;
            }else{
                Node node = head;
                head = head.next;
                return node;
            }
```

Figure 6: The "Mystack" class

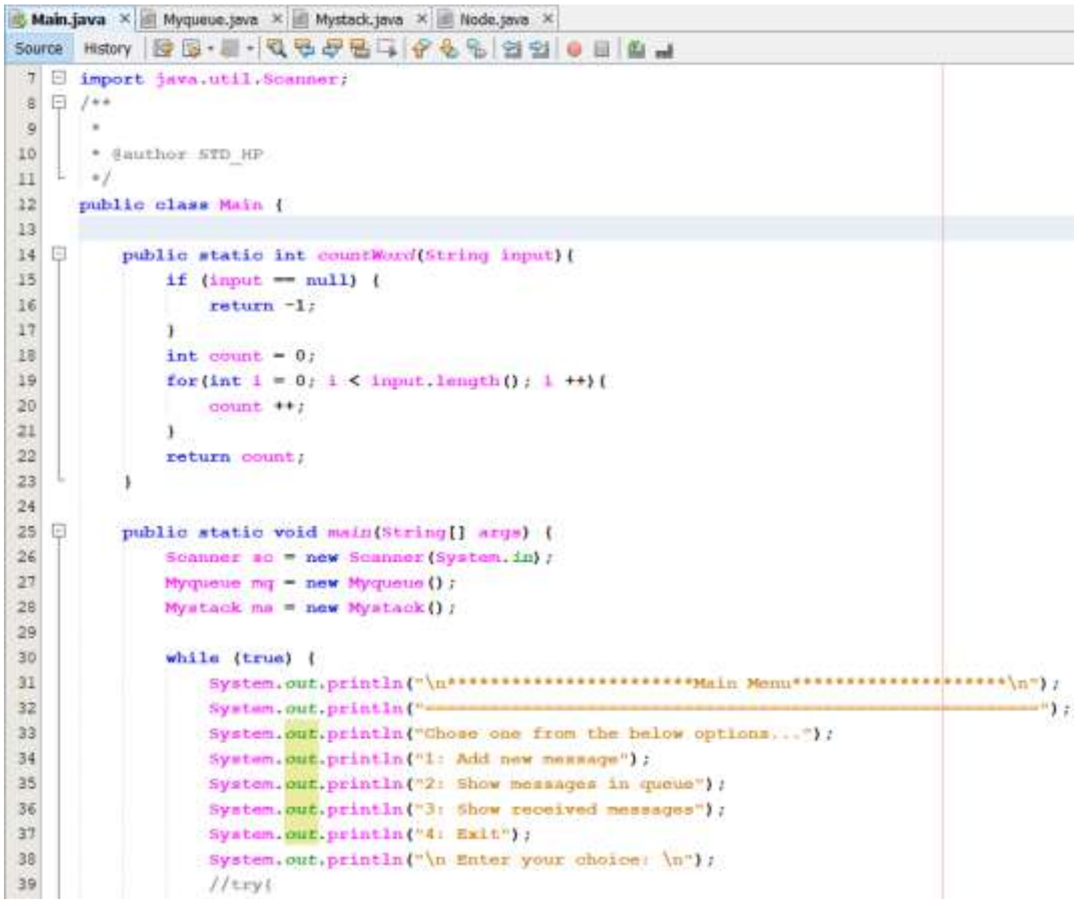
- Then, I will design the queue ADT for the system



```java
 */
public class Myqueue {
    //implementation with linked list
    // write function enqueue, dequeue, print
    Node head;
    Node tail;
    int size;

    public boolean isEmpty(){
        return(head == null);
    }
    public void enqueue(Node node){
        if(isEmpty()){
            head = tail = node;
            size ++;
        } else {
            tail.next = node;
            tail = node;
        }
    }
    public Node dequeue(){
        Node node=null;
        try{
        node=head;
        head=head.next;
        size --;
        }
        catch(Exception e){
            System.out.println("Empty queue!");
        }
        return node;
    }
```

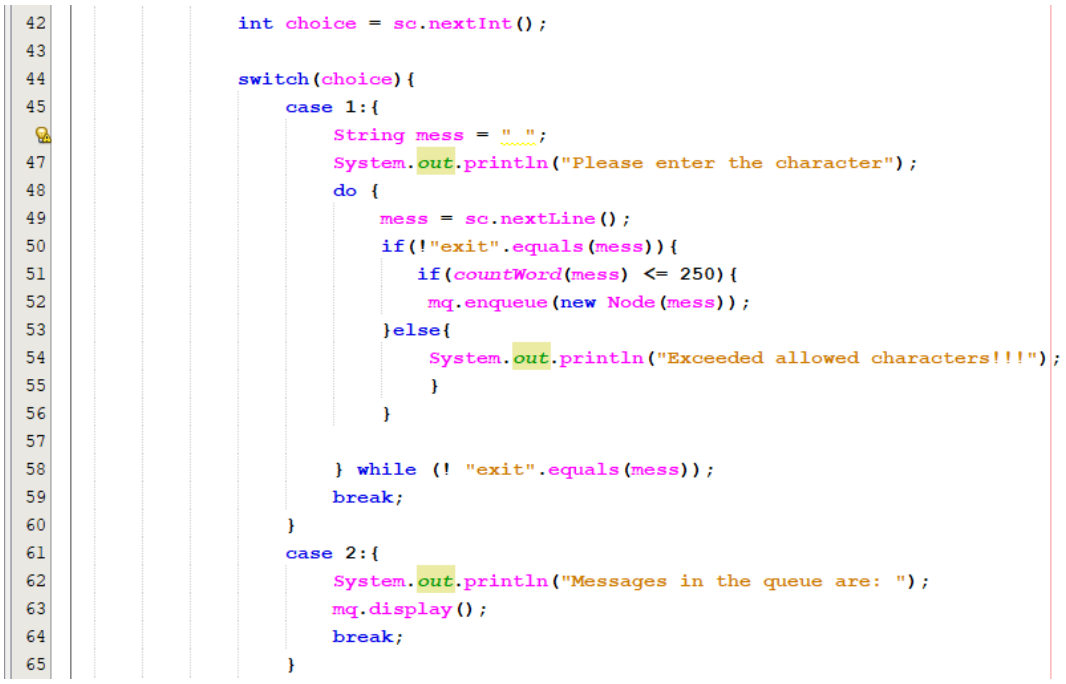
Figure 7: The "Myqueue" class

- Finally, I will create a "Main" class for the system with the following functions:



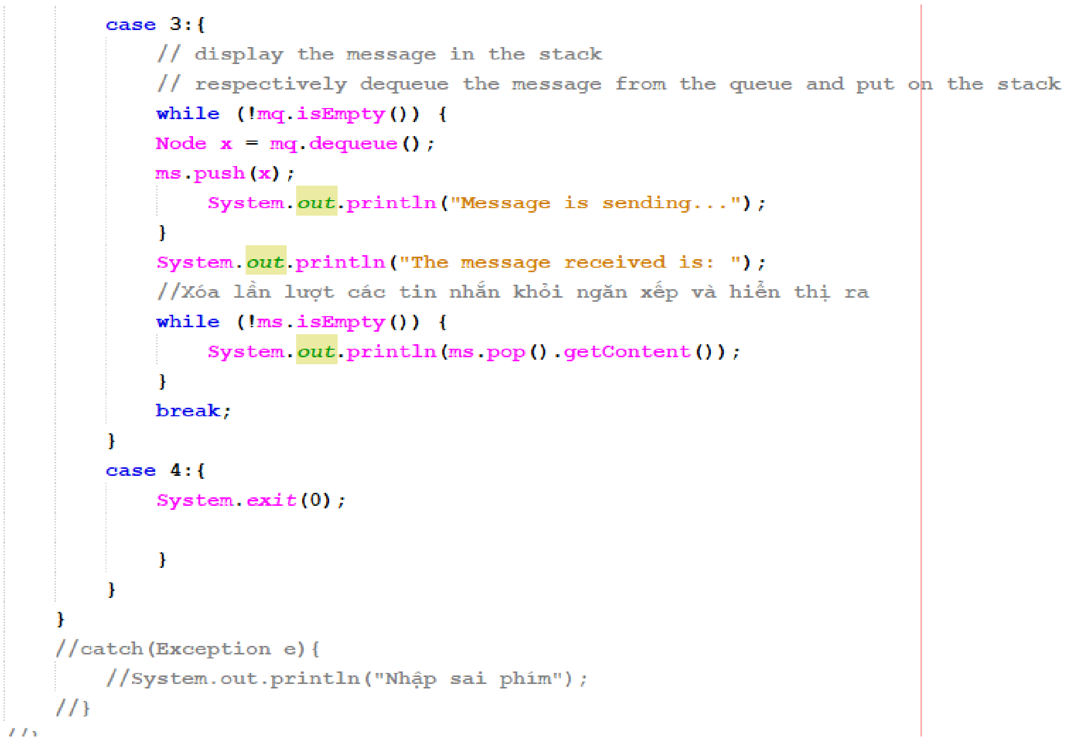
```java
import java.util.Scanner;
/**
 *
 * @author STD_HP
 */
public class Main {

    public static int countWord(String input){
        if (input == null) {
            return -1;
        }
        int count = 0;
        for(int i = 0; i < input.length(); i ++){
            count ++;
        }
        return count;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Myqueue mq = new Myqueue();
        Mystack ms = new Mystack();

        while (true) {
            System.out.println("\n************************Main Menu*********************\n");
            System.out.println("=====================================================================");
            System.out.println("Chose one from the below options...");
            System.out.println("1: Add new message");
            System.out.println("2: Show messages in queue");
            System.out.println("3: Show received messages");
            System.out.println("4: Exit");
            System.out.println("\n Enter your choice: \n");
            //try{
```

Figure 8: The functions in "Main" class



```
int choice = sc.nextInt();

switch(choice){
    case 1:{
        String mess = " ";
        System.out.println("Please enter the character");
        do {
            mess = sc.nextLine();
            if(!"exit".equals(mess)){
               if(countWord(mess) <= 250){
                mq.enqueue(new Node(mess));
            }else{
                System.out.println("Exceeded allowed characters!!!");
                }
            }

        } while (! "exit".equals(mess));
        break;
    }
    case 2:{
        System.out.println("Messages in the queue are: ");
        mq.display();
        break;
    }
```

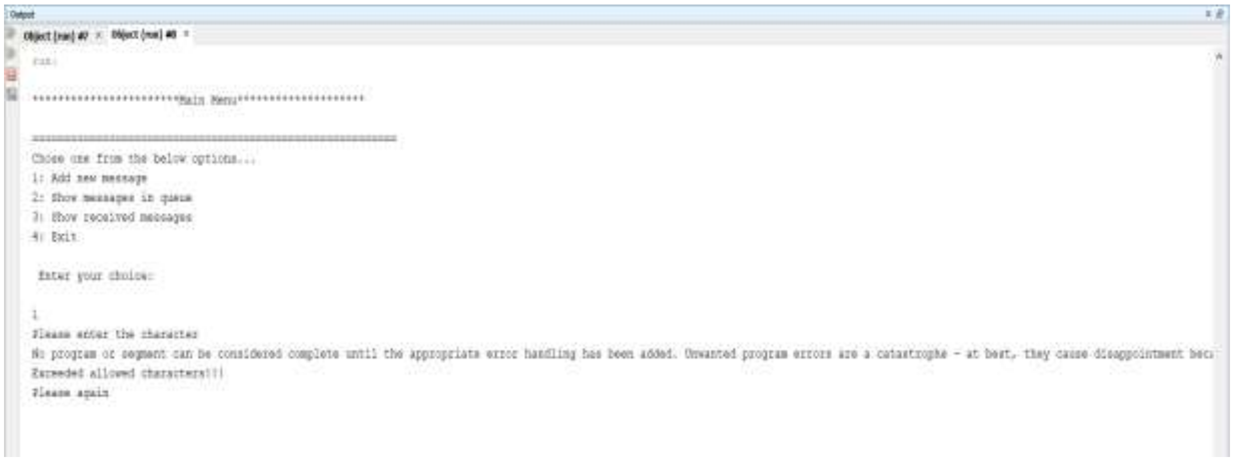
Figure 9: The case in "Main" class



```
        case 3:{
            // display the message in the stack
            // respectively dequeue the message from the queue and put on the stack
            while (!mq.isEmpty()) {
            Node x = mq.dequeue();
            ms.push(x);
                System.out.println("Message is sending...");
            }
            System.out.println("The message received is: ");
            //Xóa lần lượt các tin nhắn khỏi ngăn xếp và hiển thị ra
            while (!ms.isEmpty()) {
                System.out.println(ms.pop().getContent());
            }
            break;
        }
        case 4:{
            System.exit(0);

            }
        }
    }
    //catch(Exception e){
        //System.out.println("Nhập sai phím");
    //}
```

Figure 10: The case in "Main" class

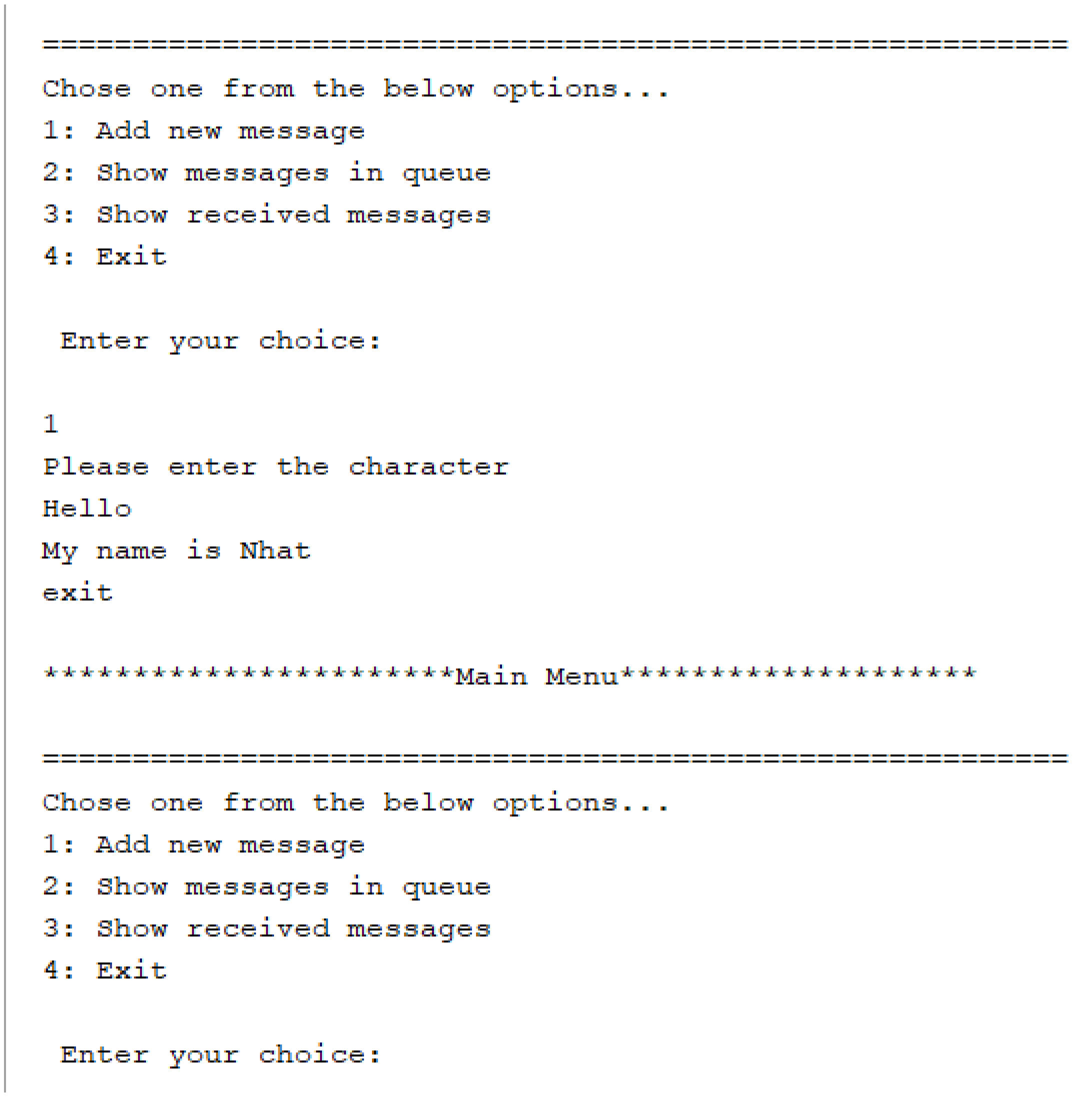
❖ **Results** of running program tests, with the number entered is the character exceeds the limit:



```
run:

**********************Main Menu*******************

=========================================================
Chose one from the below options...
1: Add new message
2: Show messages in queue
3: Show received messages
4: Exit

 Enter your choice:

1
Please enter the character
No program or segment can be considered complete until the appropriate error handling has been added. Unwanted program errors are a catastrophe - at best, they cause disappointment bec
Exceeded allowed characters!!!
Please again
```

Figure 11. Exceeded allowed characters

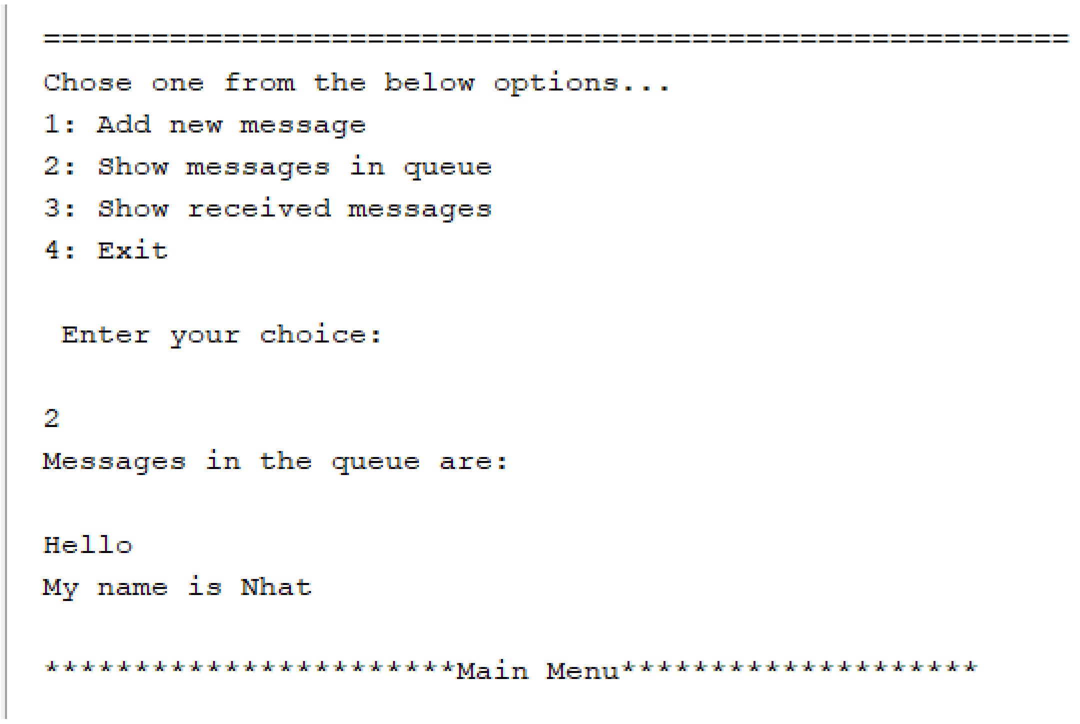
- After entering the allowed number of characters, you only need to enter the line of characters with the word "exit" and the program will return to the menu to select the function.
- Check the function of sending a message when the correct number of allowed characters has been entered.



```
==========================================================
Chose one from the below options...
1: Add new message
2: Show messages in queue
3: Show received messages
4: Exit

 Enter your choice:

1
Please enter the character
Hello
My name is Nhat
exit

***********************Main Menu********************

==========================================================
Chose one from the below options...
1: Add new message
2: Show messages in queue
3: Show received messages
4: Exit

 Enter your choice:
```

Figure 12. Function 1

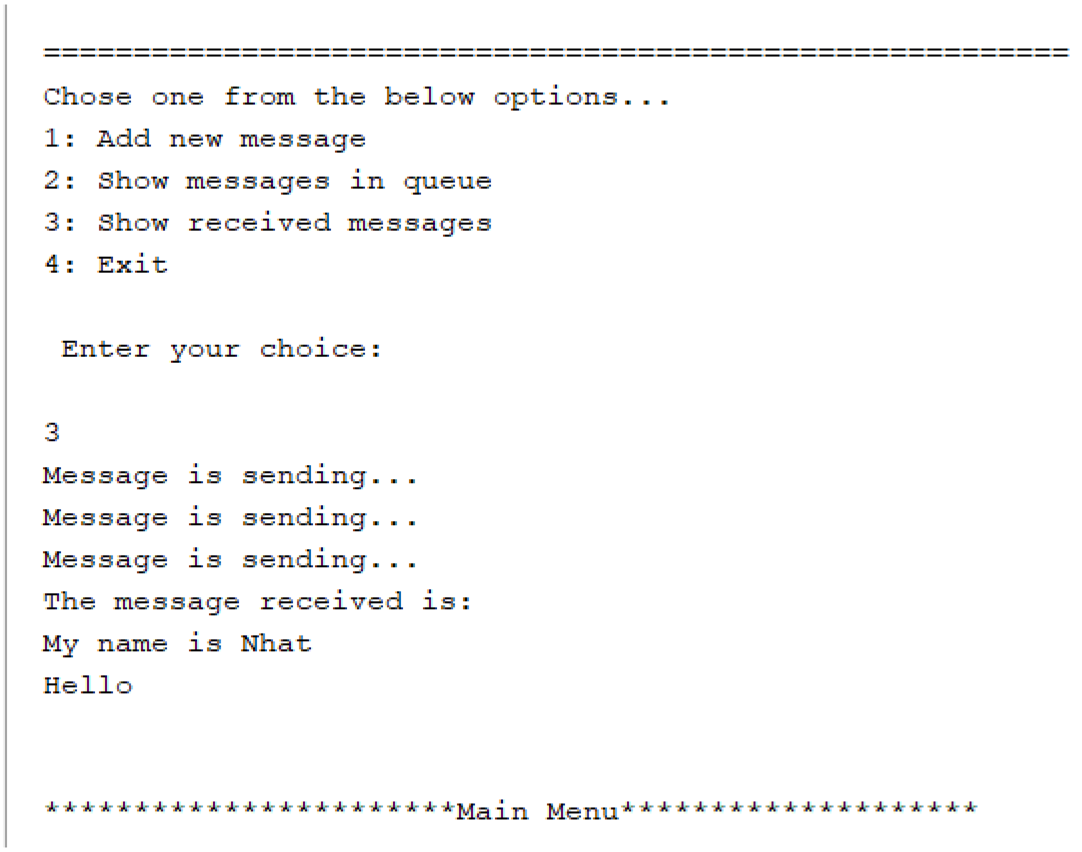
- After entering the message to send, we choose function 2 to check whether the message is already in the queue.



```
==========================================================
Chose one from the below options...
1: Add new message
2: Show messages in queue
3: Show received messages
4: Exit

 Enter your choice:

2
Messages in the queue are:

Hello
My name is Nhat

***********************Main Menu********************
```

Figure 13. Function 2

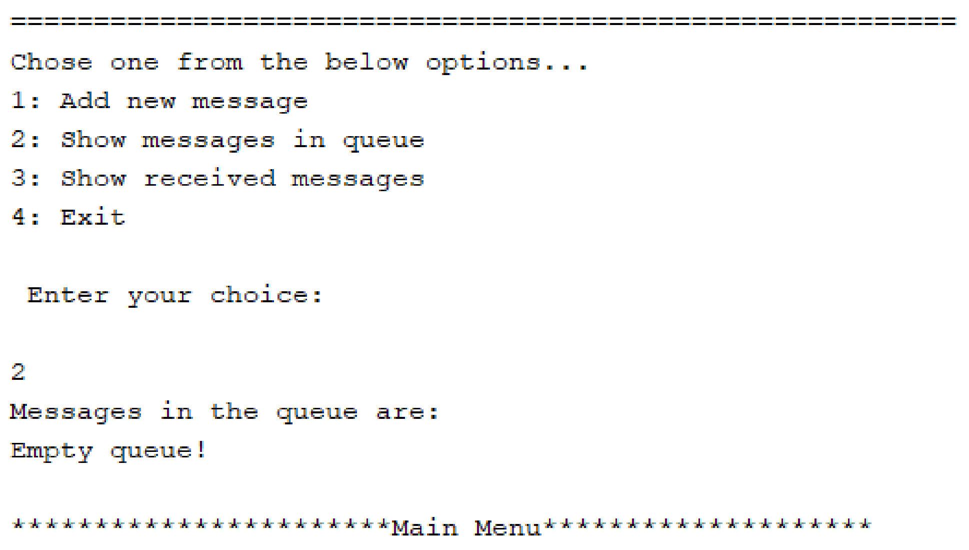
- If the message is already in the queue, next we choose function 3 to send the message.



```
==========================================================
Chose one from the below options...
1: Add new message
2: Show messages in queue
3: Show received messages
4: Exit

 Enter your choice:

3
Message is sending...
Message is sending...
Message is sending...
The message received is:
My name is Nhat
Hello


***********************Main Menu********************
```

Figure 14. Function 3

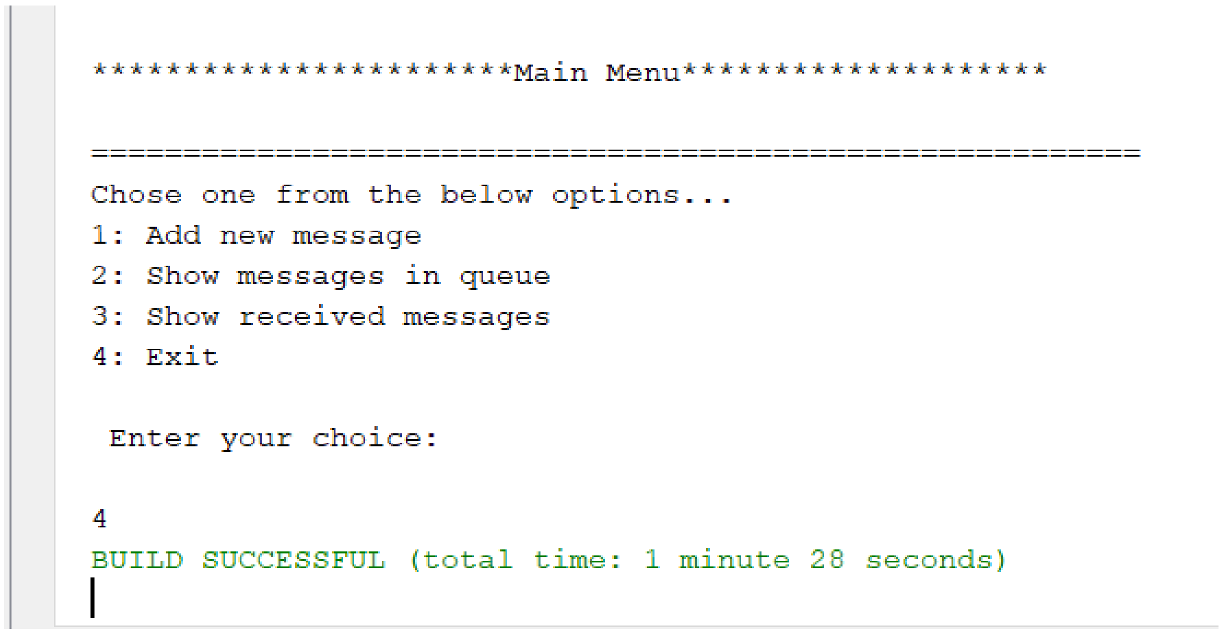
- After the message is sent, check the queue again and get an empty queue message.



```
============================================================
Chose one from the below options...
1: Add new message
2: Show messages in queue
3: Show received messages
4: Exit

 Enter your choice:

2
Messages in the queue are:
Empty queue!

***********************Main Menu********************
```

Figure 15. Empty queue!

- In the end you don't want to perform the action, you will choose function 4 to end the program



```
***********************Main Menu********************

============================================================
Chose one from the below options...
1: Add new message
2: Show messages in queue
3: Show received messages
4: Exit

 Enter your choice:

4
BUILD SUCCESSFUL (total time: 1 minute 28 seconds)
```

Figure 16. Function 4 "Exxit program"

II. Implement error handling and report test results (P5)

1. Error handling

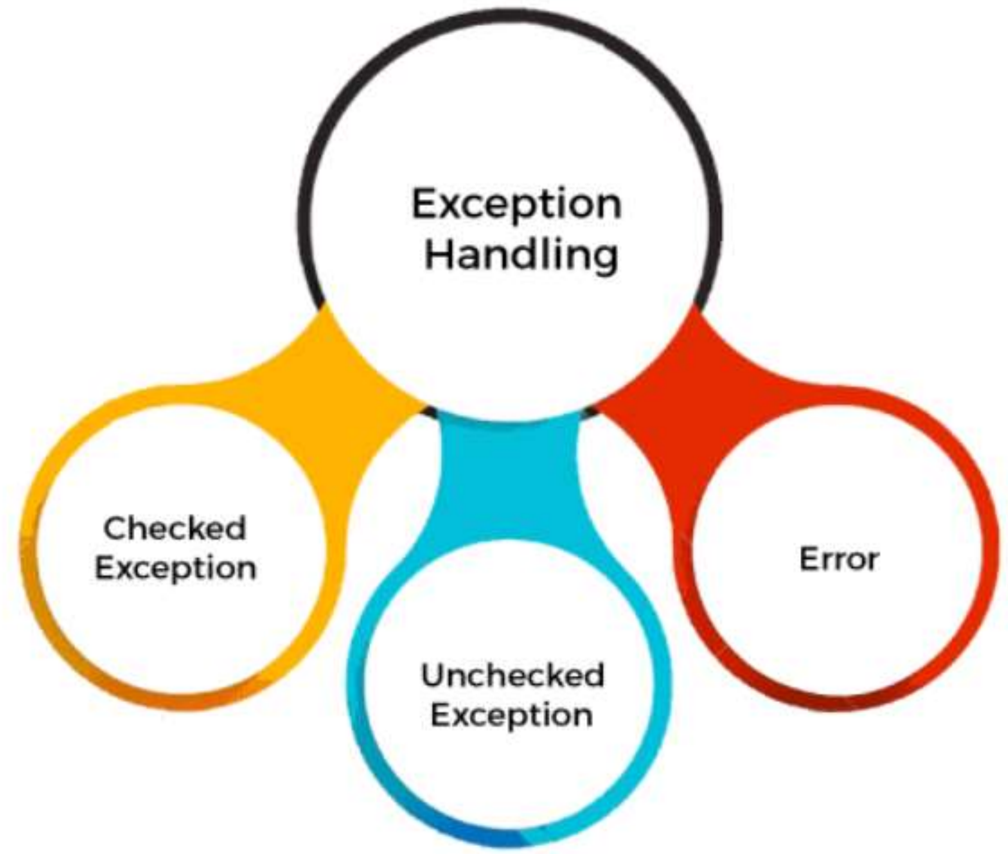


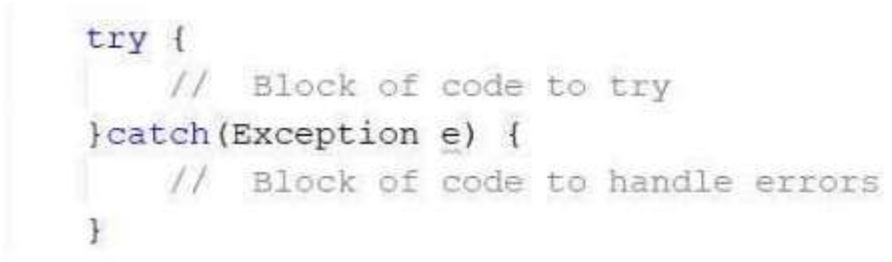
Figure 17: Type of Java Exceptions

- No program or segment can be considered complete until the appropriate error handling has been added. Unwanted program errors are a catastrophe - at best, they cause disappointment because program users have to repeat for minutes or hours of work, but in life-critical applications, even the most trivial program error, if not processed correctly, has the potential to kill someone. If an error is fatal, in the sense that a program cannot reasonably continue.
- This means that it must: informs its user (s) why it died and saves as many program states as possible.
- Error handling is the process of responding to and recovering from error conditions in your programs. When executing Java code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.
- When an error occurs, normally Java will stop and generate an error message. The technical term for this is: Java will throw an exception (throw an error). An Exception is an unusual event, often an error, that requires special processing. Exceptions can be caused by using a reference that is not yet initialized, dividing by zero, going beyond the length of an array, or even JVM not being able to assign objects on the heap.

- The try-catch block is used to execute the catch and throw pattern of exception handling. The try statement allows you to define a block of code to be tested for errors while it is being executed. The catch the statement allows you to define a block of code to be executed if an error occurs in the try block.
- The catch block in Java is used to handle the Exception by declaring the exception type in the parameter. The declared exception must be an exception of the parent class exception (i.e, Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.
- The catch block must be used after the try block only. You can use multiple catch blocks with a single try block.

(Exception Handling in Java | Java Exceptions - javatpoint, 2021)

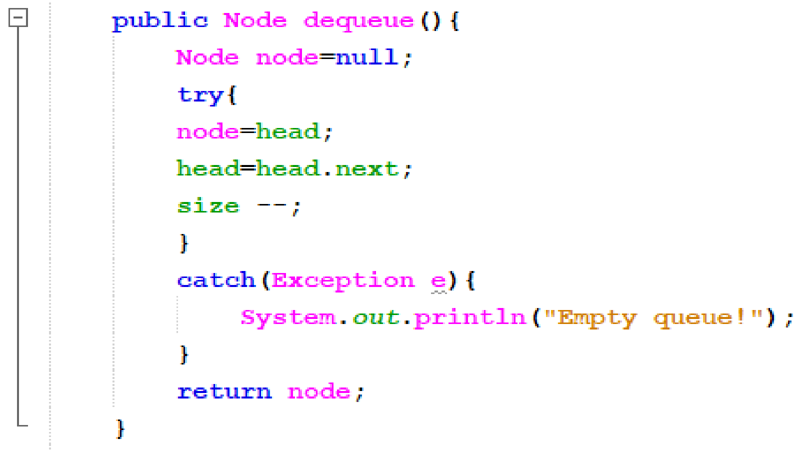
- The try and catch keywords come in pairs:



```
try {
    //  Block of code to try
}catch(Exception e) {
    //  Block of code to handle errors
}
```

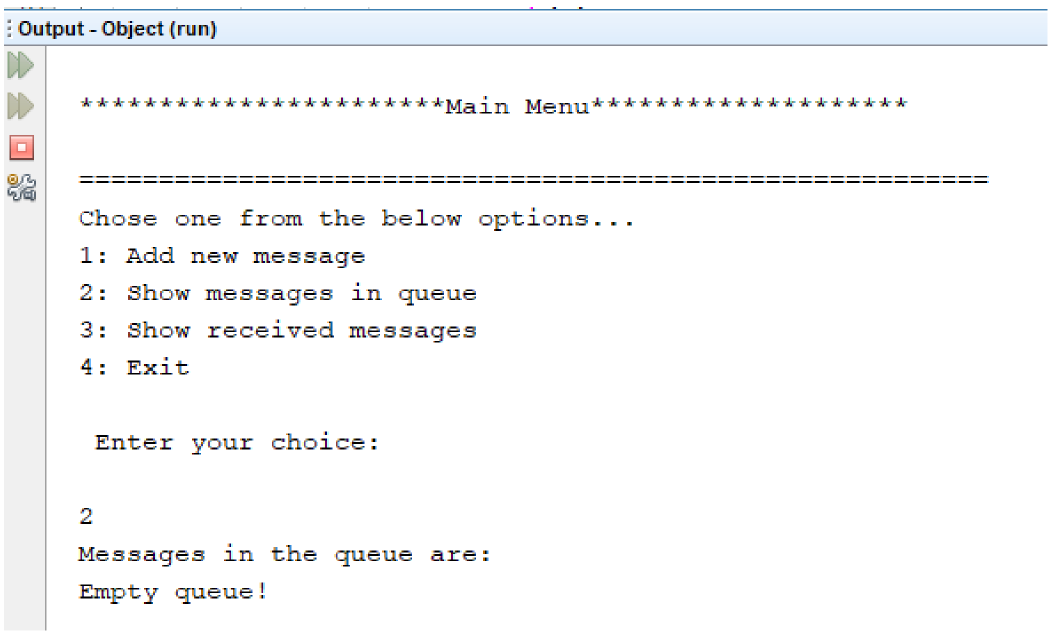
Figure 18. Syntax try-catch

❖ ***For example:*** After sending the message, the message in the queue will be dequeue, now if the user still chooses to display the message in the queue, it will immediately catch the error and give a warning that the queue is empty.



```
public Node dequeue(){
    Node node=null;
    try{
    node=head;
    head=head.next;
    size --;
    }
    catch(Exception e){
        System.out.println("Empty queue!");
    }
    return node;
}
```

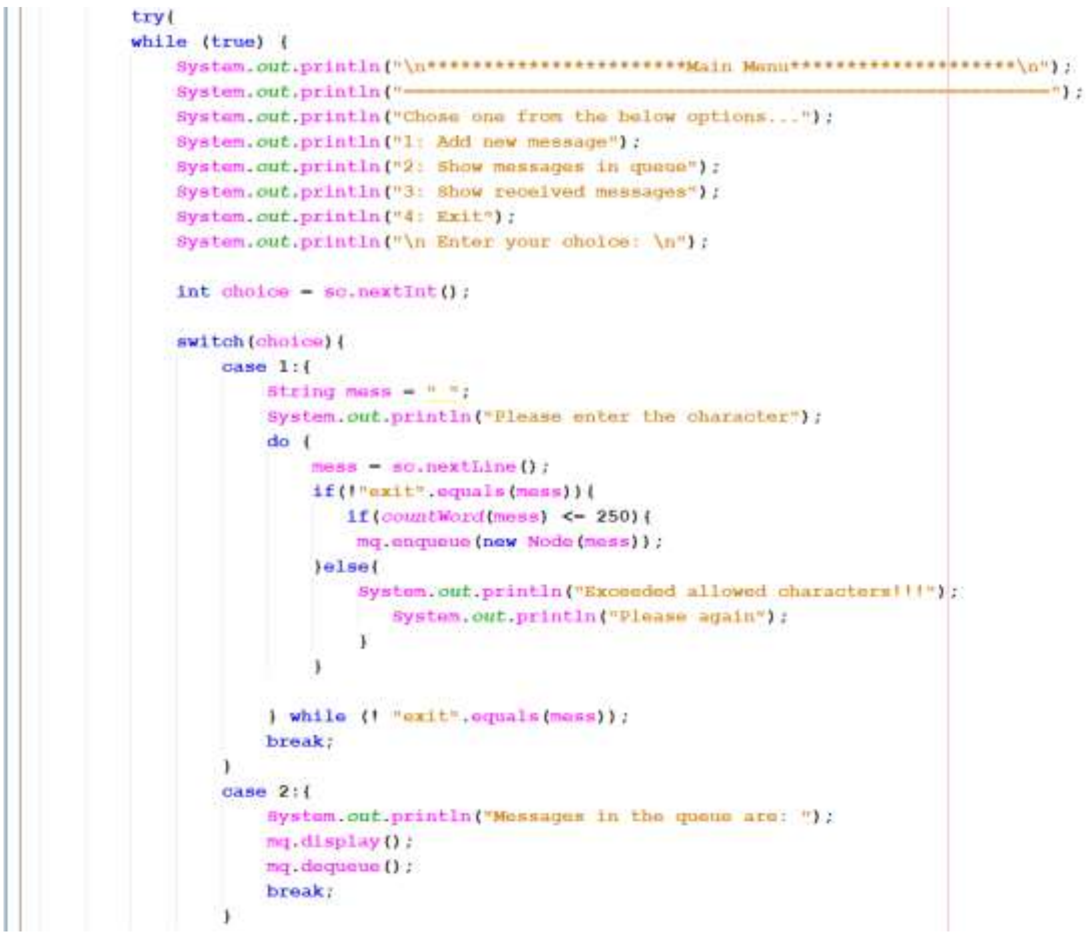
Figure 19. For example try-catch



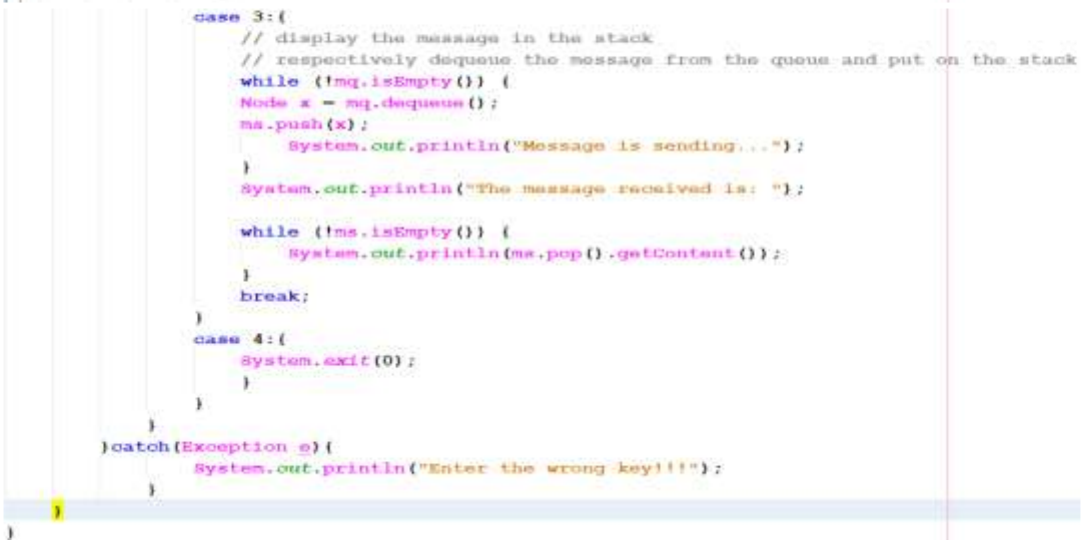
```
Output - Object (run)

**********************Main Menu********************

=======================================================
Chose one from the below options...
1: Add new message
2: Show messages in queue
3: Show received messages
4: Exit

 Enter your choice:

2
Messages in the queue are:
Empty queue!
```

Figure 20. Result try-catch

❖ ***For example:*** Or the user might choose the wrong feature, then try... catch could catch an error and issue a warning.

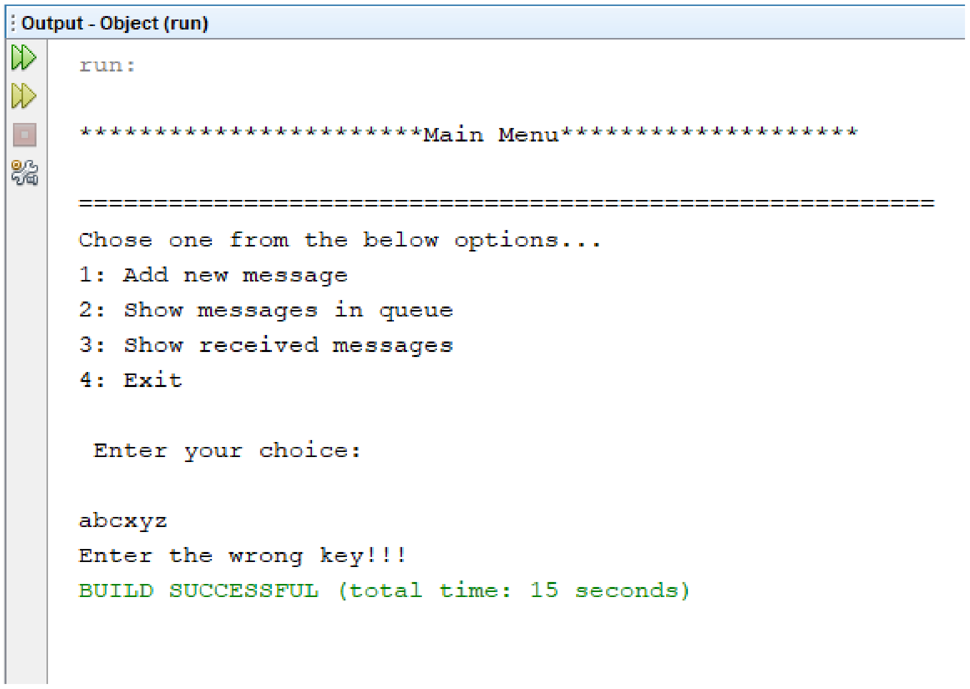


```
try{
while (true) {
    System.out.println("\n************************Main Menu*********************\n");
    System.out.println("--------------------------------------------------------------------------------");
    System.out.println("Chose one from the below options...");
    System.out.println("1: Add new message");
    System.out.println("2: Show messages in queue");
    System.out.println("3: Show received messages");
    System.out.println("4: Exit");
    System.out.println("\n Enter your choice: \n");

    int choice = sc.nextInt();

    switch(choice){
        case 1:{
            String mess = " ";
            System.out.println("Please enter the character");
            do {
                mess = sc.nextLine();
                if(!"exit".equals(mess)){
                   if(countWord(mess) <= 250){
                    mq.enqueue(new Node(mess));
                }else{
                    System.out.println("Exceeded allowed characters!!!");
                       System.out.println("Please again");
                    }
                }

            } while (! "exit".equals(mess));
            break;
        }
        case 2:{
            System.out.println("Messages in the queue are: ");
            mq.display();
            mq.dequeue();
            break;
        }
```



```
        case 3:{
            // display the message in the stack
            // respectively dequeue the message from the queue and put on the stack
            while (!mq.isEmpty()) {
            Node x = mq.dequeue();
            ms.push(x);
                System.out.println("Message is sending...");
            }
            System.out.println("The message received is: ");

            while (!ms.isEmpty()) {
                System.out.println(ms.pop().getContent());
            }
            break;
        }
        case 4:{
            System.exit(0);
            }
        }
    }
}catch(Exception e){
        System.out.println("Enter the wrong key!!!");
    }
}
}
```

Figure 21. For example try-catch


Output - Object (run)

```
run:

***********************Main Menu********************

==========================================================
Chose one from the below options...
1: Add new message
2: Show messages in queue
3: Show received messages
4: Exit

 Enter your choice:

abcxyz
Enter the wrong key!!!
BUILD SUCCESSFUL (total time: 15 seconds)
```

Figure 22. Result try-catch

2. Test case

The test case is describing an input, action, or event and expected response. The test case is to check whether each function of the software application is working properly or not.

Module test	IDE
Tester	LE VAN NHAT
Create Date	18/08/2021
Test environment	

Test ID	Description	Expected result	Date	Actual result	Result
1	Enter a message. The message is a string of a maximum of 250 characters. If it exceeds 250 characters, an error message is an output.	Enter your message. If it exceeds 250 characters, an error message will appear Exceeded allowed characters and ask you enter please again.	18/08/2021	The length limit of the string is 250. Please again.	Pass
	Enter a message. The message is a	Enter your message. If it		Enter the correct	

2	string of a maximum of 250 characters. If it exceeds 250 characters, an error message is an output.	exceeds 250 characters, an error message will appear Exceeded allowed characters and ask you enter please again.	18/08/2021	number of characters in the message.	Pass
3	The check function is sent the message.	Enter a message. This message has been added to the system	18/08/2021	The message has been added to the system.	Pass
4	Check the feature of receiving messages	The system will receive the sent message	18/08/2021	The system received the sent message	Pass
5	Check the feature of displaying all messages	Display all messages in the system.	18/08/2021	All messages have been displayed in the system.	Pass
6	Check if the queue after sending the message is empty or not	Empty queue message!	18/08/2021	Queue is empty	Pass

Table 1.0: Test case

III. Discuss how asymptotic analysis can be used to assess the effectiveness of an algorithm (P6)

1. What is Algorithmic Analysis?

- An algorithm is a set of clearly specified simple instructions to follow in order to solve a problem. Once an algorithm is given for a problem and decided (somehow) to be correct, an important step is to determine how much in the way of resources, such as time or space, the algorithm will require. An algorithm must be correct. It should correctly solve the problem. For example, for sorting, this means even if (1) the input is already sorted, or (2) it contains repeated elements. To analysis an algorithm we need some kind of syntax, and that forms the base for asymptotic analysis/notation. There are three types of analysis:

❖ Worst case

- Determine which input the algorithm takes a long time (slowest completion time).
- Input is the input on which the algorithm is slowest.

❖ Best case

- Determine which input the algorithm takes the least time (fastest completion time).
- The input is the input on which the algorithm runs fastest.

❖ Average case

- Provides predictions about the running time of the algorithm.
- Run the algorithm multiple times, using many different inputs that come from some distribution that generates these inputs, compute the total running time (by adding the individual times), and divide by the number of trials.
- Assumes that the input is random

Lower Bound < = Average Time < = Upper Bound

- For a given algorithm, we can represent the best, worst and average cases in the forms of expressions. For example, let f(n) be the function, which represents the given algorithm.

f(n) = n2 + 500, for worst case

f(n) = n + 10n + 50, for best case

- The same goes for the average case. The expression defines the inputs with which the algorithm takes the average running time (or memory).

❖ ***Why Analyze an Algorithm?***

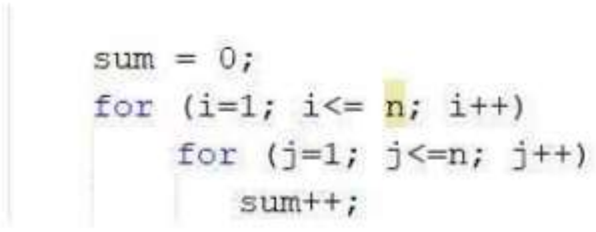
The most straightforward reason for analyzing an algorithm is to discover its characteristics in order to evaluate its suitability for various applications or compare it with other algorithms for the same application. Moreover, the analysis of an algorithm can help us understand it better, and can suggest informed improvements. Algorithms tend to become shorter, simpler, and more elegant during the analysis process.

❖ **The execution Time of Algorithms**

- The running time of an algorithm for a specific input depends on the number of operations executed. Therefore, each operation takes a certain amount of time. The greater the number of operations, the longer the running time of an algorithm.
- For example, The running time of a statement assigning the first value of an integer array to a variable is simply the time it takes to copy the value of the first array value. We can assume this assignment takes a constant amount of time regardless of the value. Let us call c1 the amount of time it takes to copy an integer. No matter how large an array on a typical computer is (given the reasonable terms of memory and array size), the time it takes to copy a value from the first position of the array is always c1. Hence, the equation for this algorithm is simply indicating that the size of the input n has no effect on run time. This is called a constant running time.

T(n) = c1;

Now, consider the following code:



```
sum = 0;
for (i=1; i<= n; i++)
    for (j=1; j<=n; j++)
        sum++;
```

- The basic operation in this example is the cumulative operation for the sum of the variables. We can assume that increment takes a constant time; call this time c2. (We can ignore the time required to initialize sum, and to increment the loop counters i and j. In practice, these costs can safely be bundled into time c2). The total number of increment operations is n2. Hence, we say that the runtime is T (n) = c2n2.
- The growth rate for an algorithm is the rate at which the cost of the algorithm grows as the size of its input grows.
    - ➢ Here is a list of commonly used growth rates:

Time Complexity	Name	Example
1	Constant	Inserts an element to the front of linked list
Log n	Logarithmic	Finding an element in a sorted array
nlogn	Linear Logarithmic	Sorting n item by "divide-and-conquer" – Mergesort
n	Linnear	Finding an element in an unsorted array
n^2	Quadratic	Shortest path between two nodes in a graph
n^3	Cubic	Matrix Multiplication
2^n	Exponential	The Roads of Hanoi problem

Table 1.1: Commonly used Rates of Growth

- ➢ Now, consider the following figure:

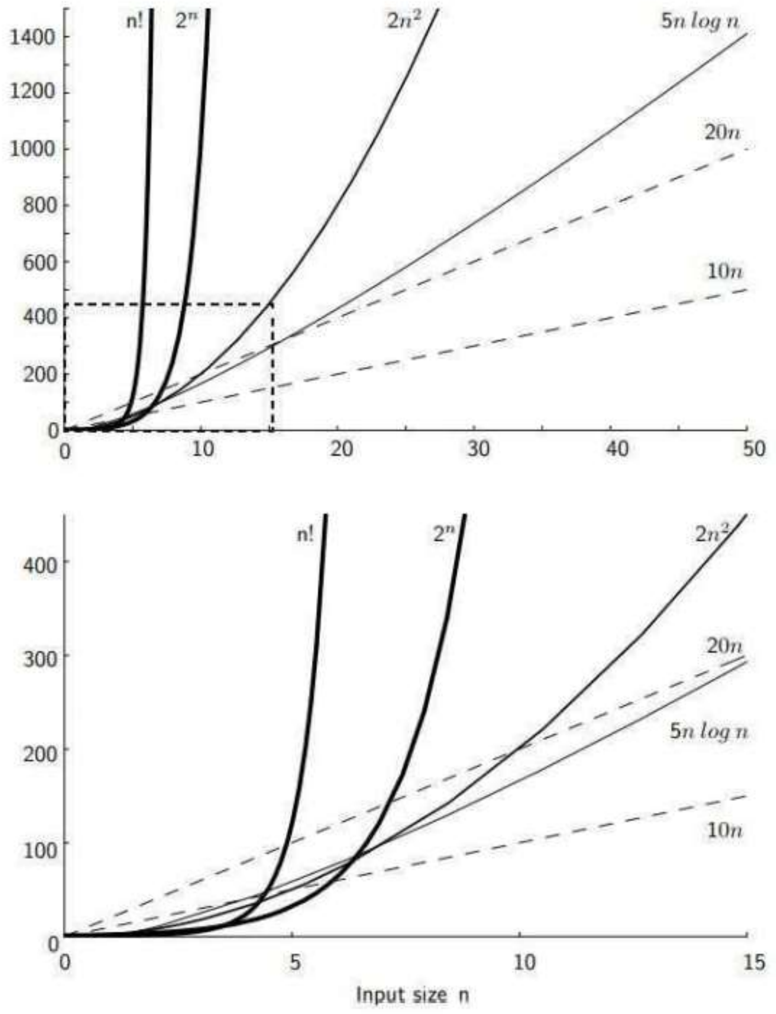
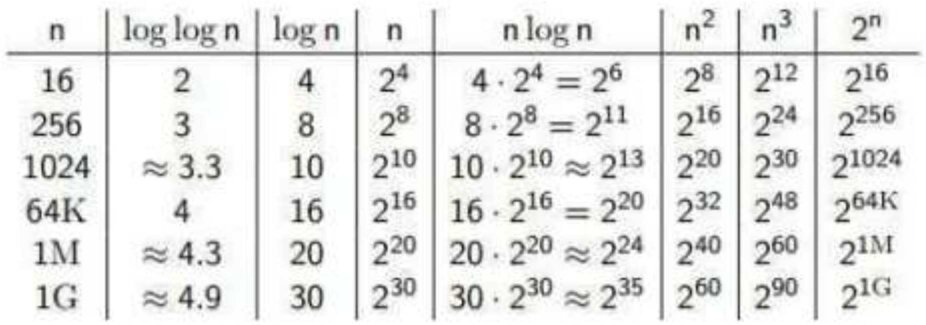


Figure 23. Two views of a graph illustrating the growth rates for six equations. The horizontal axis represents input size. The vertical axis can represent time, space, or any other measure of cost.

- The figure above shows a graph for six equations, each of which describes the running time for a particular program or algorithm. A variety of representative growth rates for the typical algorithms are shown. The two equations labeled 10n and 20n are graphed by straight lines. The growth rate can (with any positive constant c) is often referred to as the linear growth rate or run time. This means that as the value of n is growing, the running time of the algorithm also increases proportionally. Doubling the value of n containing a factor of n2 is said to have a quadratic growth rate. In the Figure above, the line labeled 2n2 represents the quadratic growth rate. The line labeled 2n represents the exponential growth rate. This name comes from the fact that n appears in the exponent. The line labeled n! is also increasing exponentially. As can be seen from the Figure above, the difference between an algorithm whose running time has cost T (n) = 10n and another with cost T (n) = 2n2 becomes large as n increases. For n> 5, the algorithm with running time T (n) = 2n2 was much slower. This is despite the fact that 10n has a constant coefficient greater than 2n2. Comparing the two curves marked 20n and 2n2 shows that changing the constant-coefficient for one of the equations only shifts the point where the two curves cross.

- We can get some further insight into relative growth rates for various algorithms from the Figure below:



n	log log n	log n	n	n log n	n^2	n^3	2^n
16	2	4	2^4	$4 \cdot 2^4 = 2^6$	2^8	2^{12}	2^{16}
256	3	8	2^8	$8 \cdot 2^8 = 2^{11}$	2^{16}	2^{24}	2^{256}
1024	≈ 3.3	10	2^{10}	$10 \cdot 2^{10} \approx 2^{13}$	2^{20}	2^{30}	2^{1024}
64K	4	16	2^{16}	$16 \cdot 2^{16} = 2^{20}$	2^{32}	2^{48}	2^{64K}
1M	≈ 4.3	20	2^{20}	$20 \cdot 2^{20} \approx 2^{24}$	2^{40}	2^{60}	2^{1M}
1G	≈ 4.9	30	2^{30}	$30 \cdot 2^{30} \approx 2^{35}$	2^{60}	2^{90}	2^{1G}

Figure 24. Cots for growth rates representative of most computer algorithm

- Each operation in an algorithm (or a program) has a cost.
  - ➢ Each operation takes a certain of time.

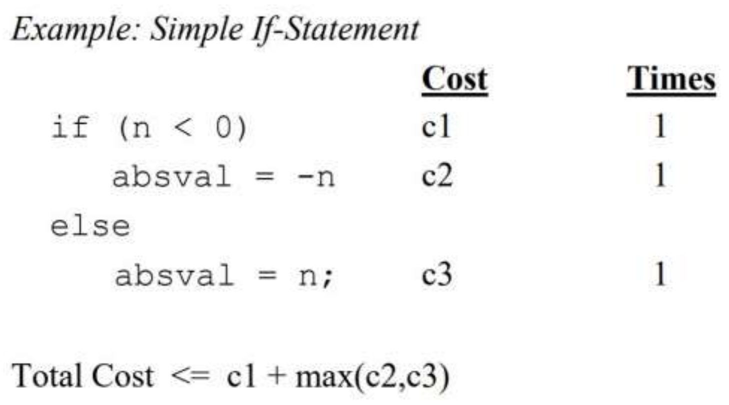
count = count + 1;

- Take a certain amount of time, but it is constant
- A sequence of operations:

  count = count + 1; Cost: c1

  sum = sum + count; Cost: c2

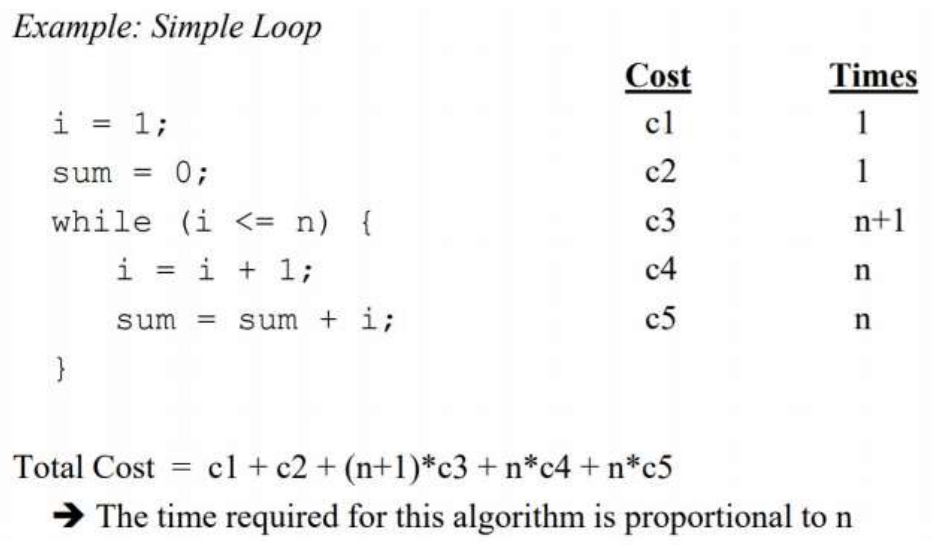
  - ➢ Total Cost = c1 + c2



Example: Simple If-Statement

	Cost	Times
`if (n < 0)`	c1	1
`    absval = -n`	c2	1
`else`		
`    absval = n;`	c3	1

Total Cost <= c1 + max(c2,c3)



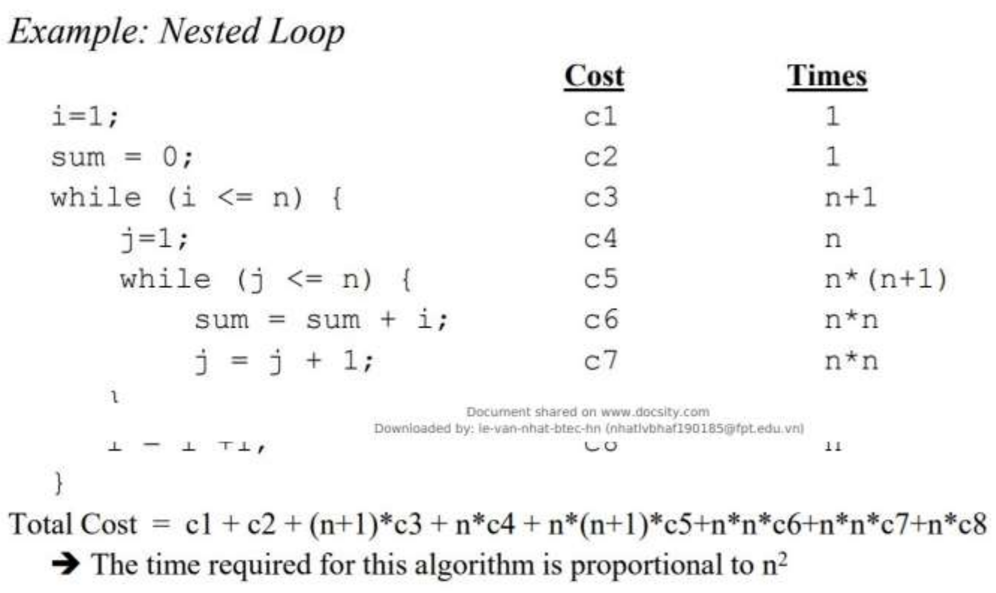
Example: Simple Loop

	Cost	Times
`i = 1;`	c1	1
`sum = 0;`	c2	1
`while (i <= n) {`	c3	n+1
`    i = i + 1;`	c4	n
`    sum = sum + i;`	c5	n
`}`		

Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*c5

➔ The time required for this algorithm is proportional to n

Figure 25. Example 1



Example: Nested Loop

	Cost	Times
`i=1;`	c1	1
`sum = 0;`	c2	1
`while (i <= n) {`	c3	n+1
`    j=1;`	c4	n
`    while (j <= n) {`	c5	n*(n+1)
`        sum = sum + i;`	c6	n*n
`        j = j + 1;`	c7	n*n
`    }`		
`    i = i +1;`	c8	n
`}`		

Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*(n+1)*c5+n*n*c6+n*n*c7+n*c8

➔ The time required for this algorithm is proportional to n^2

Figure 26. Example 2

2. What is Asymptotic Analysis?

- ***Asymptotic analysis*** of an algorithm refers to the mathematical / framework determination of its run-time performance. Using asymptotic analysis, we can most likely conclude the best-case, the average, and the worst-case of an algorithm.
- ***Asymptotic analysis*** is input bound i.e. if there is no input to the algorithm it is concluded to operate for a constant time. Aside from the "input", all other factors are considered constant. Asymptotic analysis refers to the runtime calculation of any activity in computational units. For example, the runtime of one activity is counted as f (n), and possibly for another activity, it is counted as g (n2). This means that the run time of the first operation will increase linearly with an increase of n and that the run time of the second operation will increase exponentially with an increase in n. Similarly, the runtime of both operations would be approximately the same if n were significantly smaller.
- There are mainly three asymptotic notations:
  - Big-O notation
  - Omega notation
  - Theta notation

❖ ***How Does Asymptotic Analysis Work?***

- This analysis needs a variable input to the algorithm, otherwise, the work is assumed to require a constant amount of time. All factors other than the input operation are considered constant.

❖ ***For example***: the running time of a given data mining query is considered f(n), and its corollary search operation is calculated as g(n2). So the first operation's running time increases linearly with the rise in n, while the running time of the second operation increases exponentially as n is enlarged. While run-time performance can be calculated with many different functions, the limiting behavior of the algorithm is expressed graphically using the simple notation:

- **O(n)**: Is the upper bound of an algorithm's running time and measures the worst-case scenario of how long an algorithm can possibly take to complete a given operation.
- **Ω(n)**: Is the lower bound of an algorithm's running time and measures the best-case scenario of how long an algorithm can possibly take to complete a given operation.
- **Θ(n)**: Is charting both the upper and lower running time boundaries, with the average case scenario express as the average between each border.

a. Big Oh Notation, O

- The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst-case time complexity or the longest amount of time an algorithm can possibly take to complete.

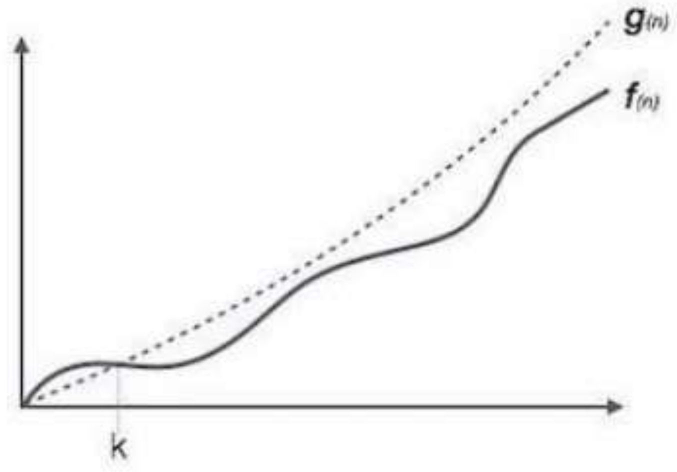


Figure 27. Big Oh Notation, O

- For example, for a function f(n)

  O(f(n)) = { g(n) : there exists c > 0 and n0 such that f(n) ≤ c.g(n) for all n > n0. }

b. *Omega Notation, Ω*

- The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

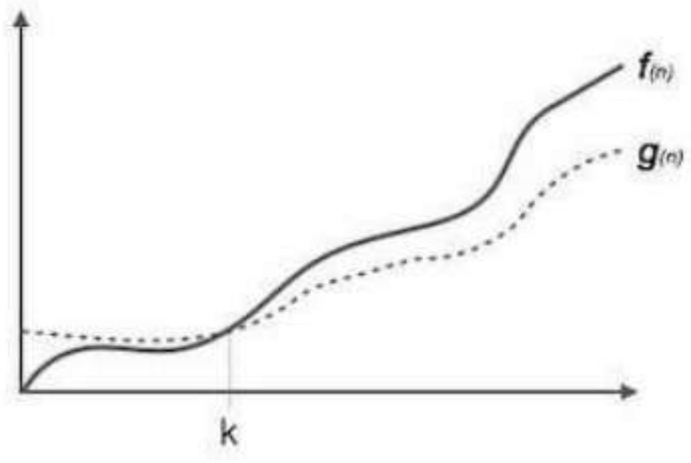


Figure 28. Omega Notation, Ω

- For example, for a function f(n)

  Ω(f(n)) ≥ { g(n) : there exists c > 0 and n0 such that g(n) ≤ c.f(n) for all n > n0. }

c. *Theta Notation, ϑ*

- The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows

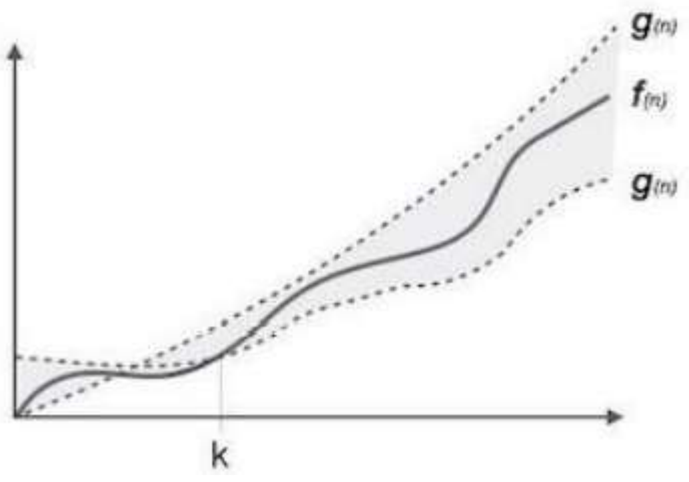
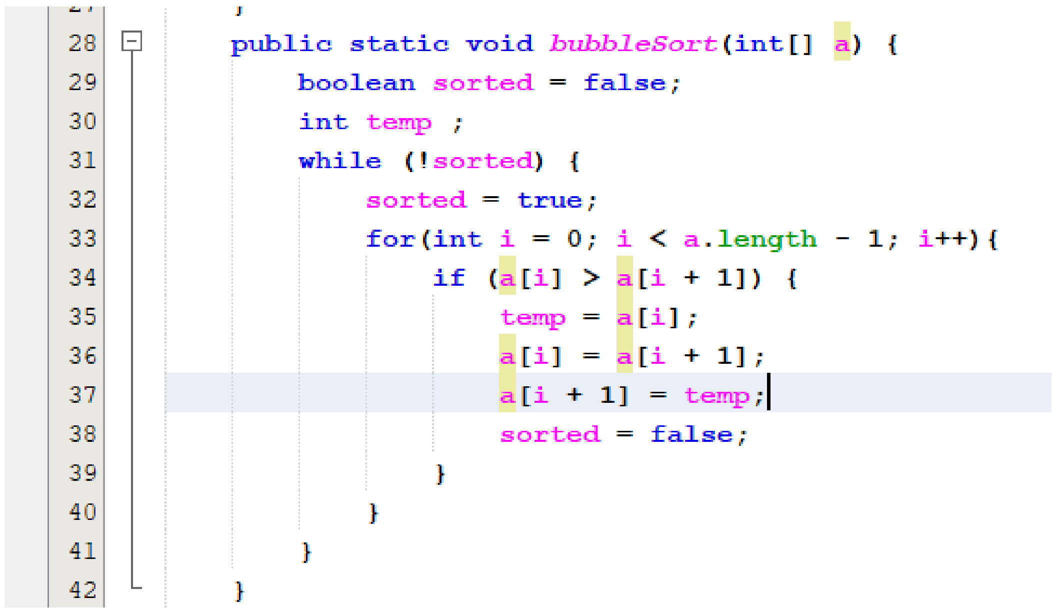


Figure 29. Theta Nodtation, ϑ

- For example, for a function f(n)

θ(f(n)) = { g(n) if and only if g(n) = O(f(n)) and g(n) = Ω(f(n)) for all n > n0. }

Example: Bubble Sort



```
    public static void bubbleSort(int[] a) {
        boolean sorted = false;
        int temp ;
        while (!sorted) {
            sorted = true;
            for(int i = 0; i < a.length - 1; i++){
                if (a[i] > a[i + 1]) {
                    temp = a[i];
                    a[i] = a[i + 1];
                    a[i + 1] = temp;
                    sorted = false;
                }
            }
        }
    }
```

Figure 30. Code Bubble Sort

➢ The performance of bubble sort O(n):
    - Best: n

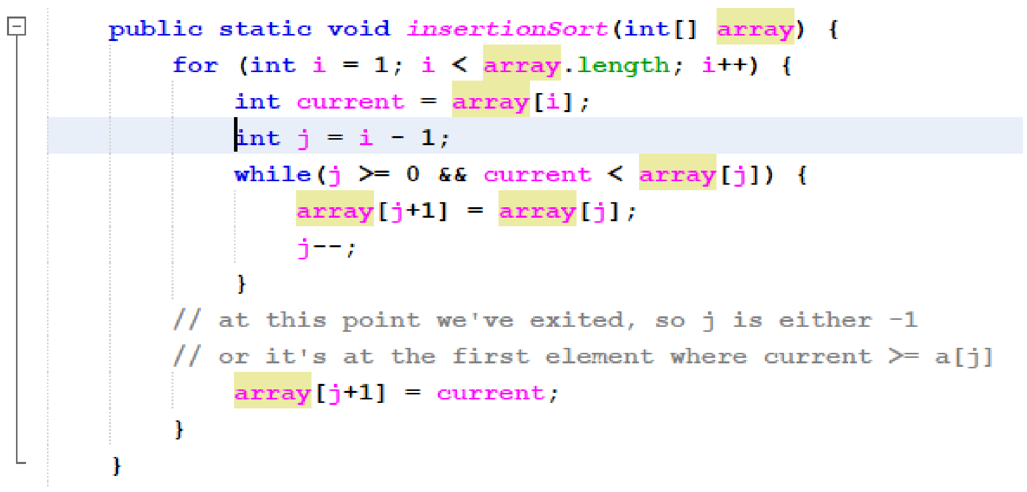
- Worst: n^2

➢ Advantages: The most concise code

➢ Cons: Low performance

Example: Insertion Sort

- Insertion sort is a simple sorting algorithm that allows for efficient, in-place sorting of the array, one element at a time. By in-place sorting, we mean that the original array is modified and no temporary structures are needed.
- Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.
- Insertion sort works similarly as we sort cards in our hands in a card game.
- To sort an array of size n in ascending order:



```
public static void insertionSort(int[] array) {
    for (int i = 1; i < array.length; i++) {
        int current = array[i];
        int j = i - 1;
        while(j >= 0 && current < array[j]) {
            array[j+1] = array[j];
            j--;
        }
        // at this point we've exited, so j is either -1
        // or it's at the first element where current >= a[j]
        array[j+1] = current;
    }
}
```

Figure 31. Code Insertion Sort

➢ Performance of insert sort O(n):

- Best: n
- Worst: n^2

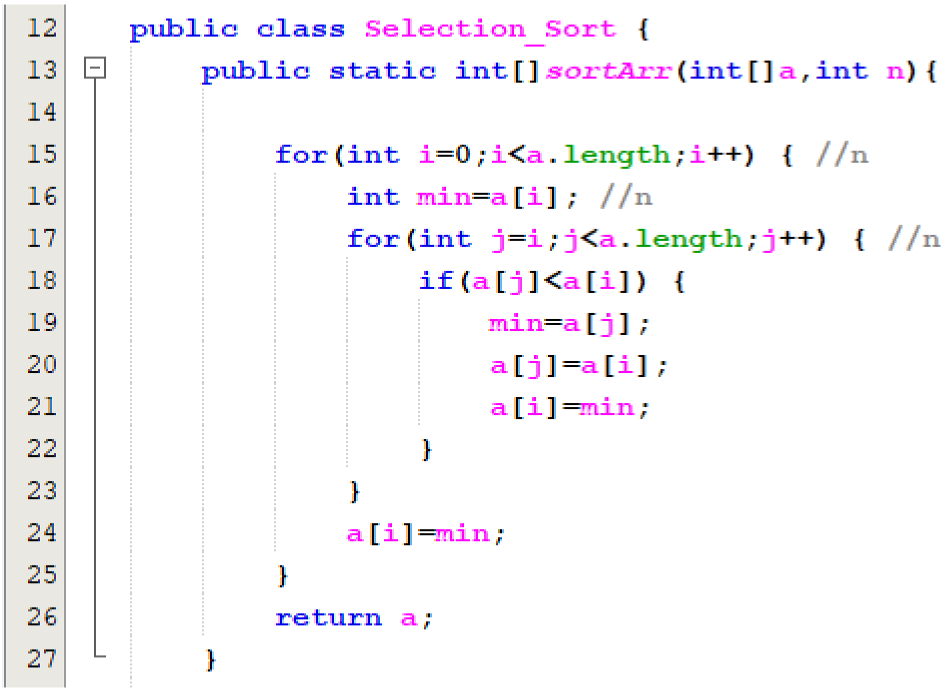
➢ Pros: Runs fast when arrays are small or partially sorted

➢ Cons: Low performance

Example: Selection sort

- Selection Sort is a simple algorithm. This sorting algorithm is an algorithm based on in-place comparison, where the list is divided into two parts, the sorted list on the left and the unsorted list on the left. right. Initially, the sorted part is empty, and the unsorted part is the entire original list.

- The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes the element of the sorted array. This process continues until all the elements in the unsorted array have been moved to the sorted array.
- This algorithm is not suitable for large data sets where the worst case and average case complexity is O(n2) where n is the number of elements.



```java
public class Selection_Sort {
    public static int[] sortArr(int[]a,int n){

        for(int i=0;i<a.length;i++) { //n
            int min=a[i]; //n
            for(int j=i;j<a.length;j++) { //n
                if(a[j]<a[i]) {
                    min=a[j];
                    a[j]=a[i];
                    a[i]=min;
                }
            }
            a[i]=min;
        }
        return a;
    }
```

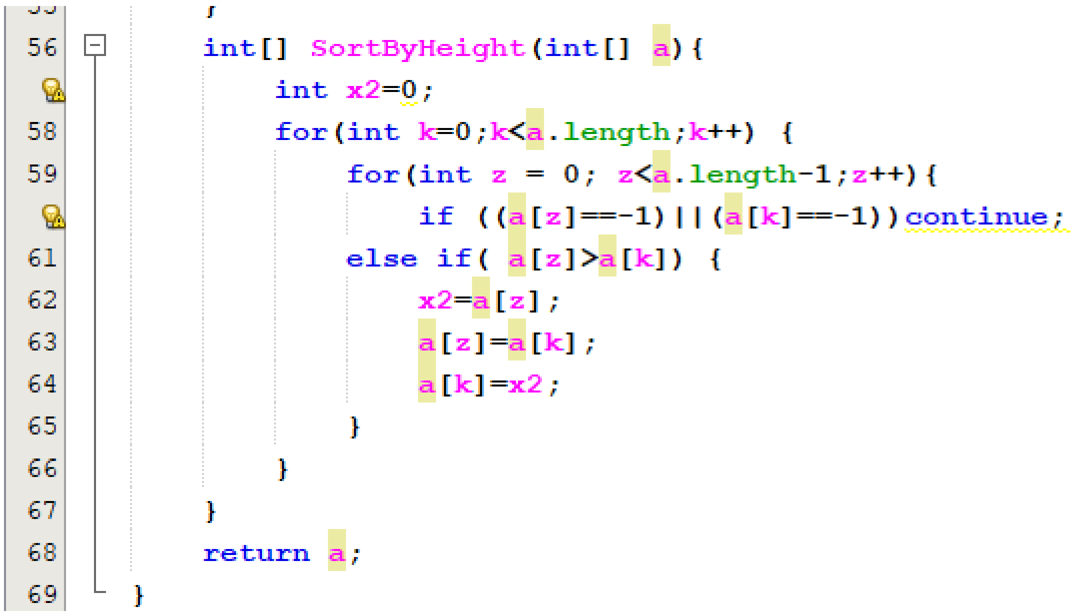
Figure 32. Code Selection Sort

- ➢ Performance of selection sort O(n):
  - Best: n^2
  - Worst: n^2
- ➢ Advantage: The algorithm runs faster when the array is partially sorted
- ➢ Cons: Performance is not high

Example: Quick sort

- Quick Sort is one of the different Sorting Techniques which is based on the concept of Divide and Conquer, just like merge sort. But in quicksort, all the heavy lifting(major work) is done while dividing the array into subarrays, while in the case of merge sort, all the real work happens during merging the subarrays. In the case of quicksort, the combined step does absolutely nothing.
- It is also called partition-exchange sort. This algorithm divides the list into three main parts:

- Elements less than the Pivot element.
- Pivot element(Central element).
- Elements greater than the pivot element.



```
    }
    int[] SortByHeight(int[] a){
        int x2=0;
        for(int k=0;k<a.length;k++) {
            for(int z = 0; z<a.length-1;z++){
                if ((a[z]==-1)||(a[k]==-1))continue;
                else if( a[z]>a[k]) {
                    x2=a[z];
                    a[z]=a[k];
                    a[k]=x2;
                }
            }
        }
        return a;
}
```

Figure 33. Code Quick Sort

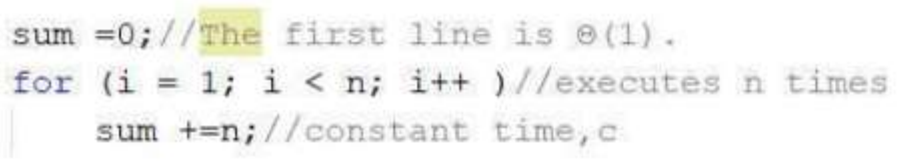
- ➢ Performance of quick sort O(n):
    - Best: n log n
    - Worst: n^2
- ➢ Advantages: Depending on how to choose the pivot, the speed of the algorithm is fast or slow
- ➢ Cons: Code is quite complicated
- ➢ The required space of quick sort is very little, just extra space O(n * log n). Quick sort is not a stable sorting technique, so it can change the appearance of two identical elements in the list when sorting.

3. General Rules for Estimation

- In this section, I will present the analysis with some simple code fragments. We begin with an analysis of a simple assignment to an integer variable.

```
a = b;
```

- Because the assignment statement takes constant time, it is Θ(1).
- There are some general rules to help us determine the running time of an algorithm.
    a. **Loops:** The running time of a loop is, at most, the running time of the statements inside the loop (including tests) multiplied by the number of iterations.
        - Consider a simple for loop:



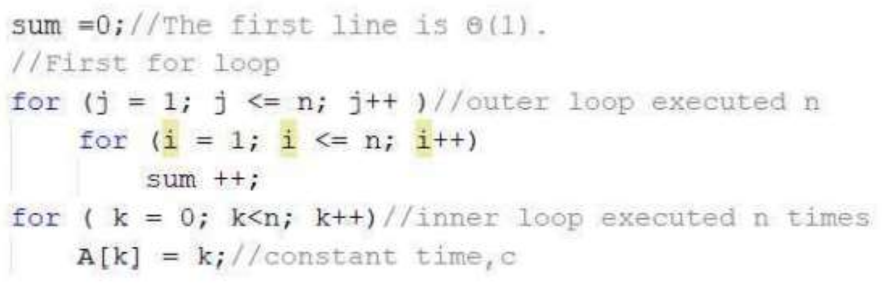
```
sum =0;//The first line is Θ(1).
for (i = 1; i < n; i++ )//executes n times
    sum +=n;//constant time,c
```

Figure 34. Loops

- The first line is Θ (1). The for loop is repeated n times. The third line takes constant time so, the total cost for executing the two lines making up the for loop is Θ(n). The cost of the entire code fragment is also Θ(n).
- Total time = a constant c × n = c n = O(n)

b. Nested loops: Analyze these insides and out. The total running time of a statement within a group of nested loops is the running time of the statement multiplied by the product of the sizes of all loops.

- We now analyze a code fragment with several for loops, some of which are nested.



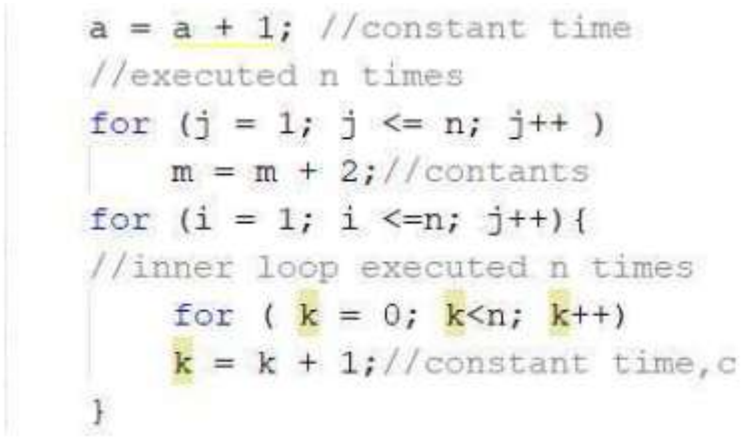
```
sum =0;//The first line is Θ(1).
//First for loop
for (j = 1; j <= n; j++ )//outer loop executed n
    for (i = 1; i <= n; i++)
        sum ++;
for ( k = 0; k<n; k++)//inner loop executed n times
    A[k] = k;//constant time,c
```

Figure 35. Nested Loops

- This code has three separate statements: the first assignment statement and the two for loops. Again, the assignment statement takes a constant time; call it c1. The second for loop takes c2n = Θ (n) time.
- We work from the inside of the loop outward. The expression sum++ requires constant time; call it c3. Because the inner for loop is executed I times, it has cost c3i. The outer for loop is executed n times, but each time the cost of the inner loop is different because it costs c3i with I changing each time. We should see that for the first execution of the outer loop, i is 1. For the second execution of the outer loop, i is 2. Each time through the outer loop, I become one greater, until the last time through the loop when i = n.
- Thus, the total cost of the loop is c3 times the sum of the integers 1 through n. we know that which is Θ(n2). Total time = Θ (c1 + c2 n + c3 n2) is simply Θ(n2).

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2},$$

c. **Consecutive statement:** Add the time complexities of each statement. As an example, the following program fragment, which has O(n) work followed by O(n2) work, is also O(n2):



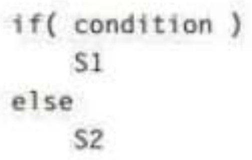
```
a = a + 1; //constant time
//executed n times
for (j = 1; j <= n; j++ )
    m = m + 2;//contants
for (i = 1; i <=n; j++){
//inner loop executed n times
    for ( k = 0; k<n; k++)
    k = k + 1;//constant time,c
}
```

Figure 36. Consecutive statement

Total time = c0 + c1 n + c2n2 = O(n2)

d. **If-then-else statements:**

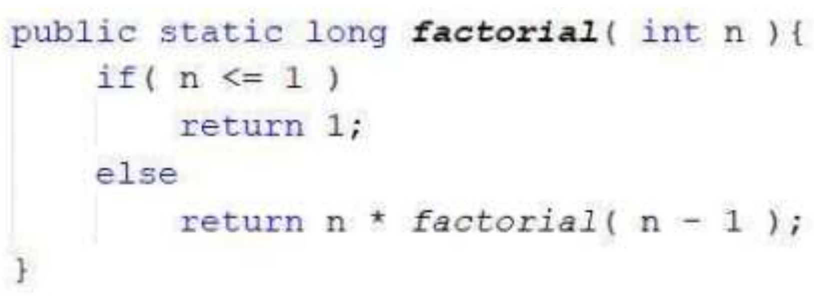
For the fragment



```
if( condition )
    S1
else
    S2
```

the running time of an if/else statement is never more than the running time of the test plus the larger of the running times of S1 and S2.

If there are method calls, these must be analyzed first. If there are recursive methods, there are several options. If the recursion is really just a thinly veiled for loop, the analysis is usually trivial. For example, the following method is really just a simple loop and is O (n).



```
public static long factorial( int n ){
    if( n <= 1 )
        return 1;
    else
        return n * factorial( n - 1 );
}
```

Figure 37. If-then-else statements

IV. Determine two ways in which the efficiency of an algorithm can be measured, illustrating your answer with an example (P7)

- Computer resources are limited that should be utilized efficiently. The efficiency of an algorithm is defined as the number of computational resources used by the algorithm. An algorithm must be analyzed to determine its resource usage. The effectiveness of an algorithm can be measured based on the use of different resources.
- The efficiency of an algorithm can be measured by determining the number of resources the algorithm consumes. Factors that influence the efficiency of an algorithm include the speed, timing, and size of the input. The main resources that an algorithm uses are as follows:
  - Space-Time tradeoff: The amount of memory used by an algorithm during its execution.
  - Time complexity: The CPU time it takes to execute the program.
- The time efficiency of an algorithm is measured by different factors. For example, write a program for a specified algorithm, execute it using any programming language, and measure the total time it takes to run. The execution time that you measure in this case will depend on a number of factors such as:
  - ➢ The speed of the machine
  - ➢ Compiler software and other systems
  - ➢ Operating system
  - ➢ The programming language is used
  - ➢ The volume of data required.
- The factor of time is usually more important than that of space, so efficiency considerations usually focus on the amount of time elapsed when processing data. However, the least efficient algorithm running on a Cray computer can execute much faster than the most efficient algorithm running on a PC, so the runtime always depends on the system. For instance, to compare 20 algorithms, all of them would have to be run on the same machine. Furthermore, the results of runtime tests depend on the language in which a given algorithm is written, even if the tests are performed on the

same machine. If programs are compiled, they execute much faster than when they are interpreted. A program written in C or Ada can be up to 20 times faster than a similar program encoded with BASIC.

1. Time complexity

- Time complexity is the number of operations an algorithm performs to fulfill its task against the input size (considering that each operation takes the same amount of time). The algorithm that performs the task with the smallest number of operations is considered the most efficient.
- Time-complexity can be expressed using the below three terms called Asymptotic Notations.
    - Big - Oh or Big O Notation (BIG O)
    - Big - Omega
    - Big – Theta.
- But most times, we will use the Big O notation because it will give us an upper limit of the execution time i.e., the execution time in the worst-case inputs. The Big O notation represents the runtime of an algorithm in terms of its growth rate relative to the input (this input is called "n"). In this way, if we say for example that an algorithm's runtime increases "in the order of the size of the input", we would state that it is "O (n)". If we say that the runtime of an algorithm increases "in order of the square of the size of the input", we will denote it as "O (n^2)".
- Typical complexities of an algorithm:

Big O Notation	Time Complexity Detail
O(1)	Constant Time Complexity O (1) occurs when the program doesn’t contain any loops, recursive functions or call to any other functions. The run time, in this case, won’t change no matter what the input value is
O(n)	Linear Time Complexity O(n) occurs when the run time of the code increases at an order of magnitude proportional to n. Here n is the size of the input.
O(log n)	Logarithmic Time Complexity O (log n) occurs when at each subsequent step in the algorithm, the time is decreased at a magnitude inversely proportional to N. This generally happens in Binary Search Algorithm.
O(n log n)	Linearities Time Complexity. One example of an algorithm that runs with this time complexity is Quick Sort, Heap Sort, Merge Sort
O(n^2)	Quadratic Time Complexity
O(2^n)	Exponential Time Complexity
O(n!)	Factorial Time Complexity

Table 1.2: Time Complexity Table

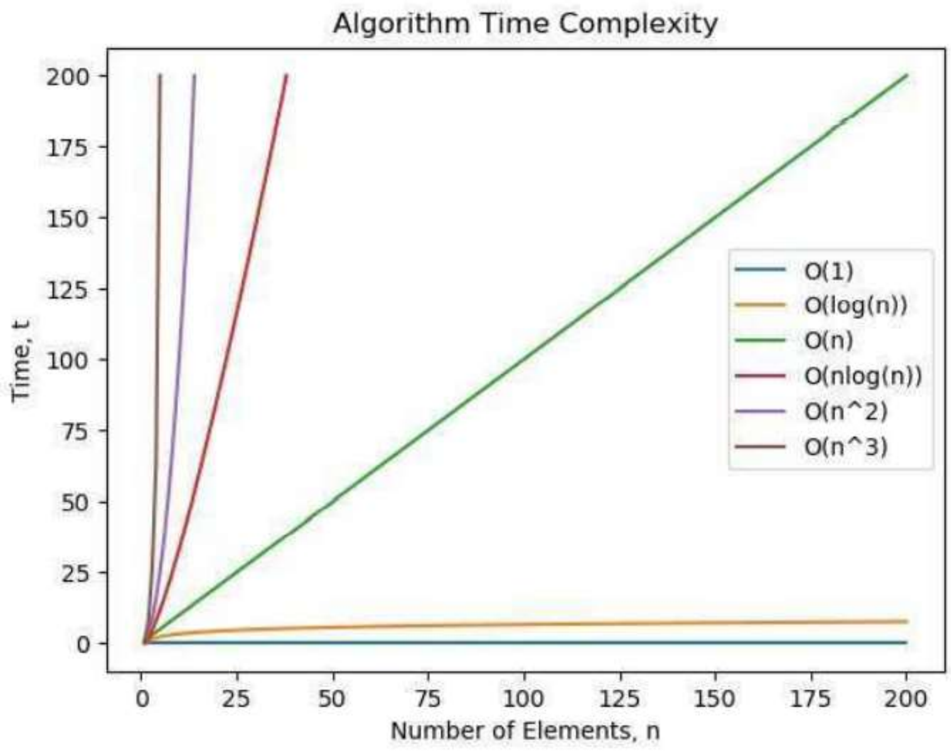
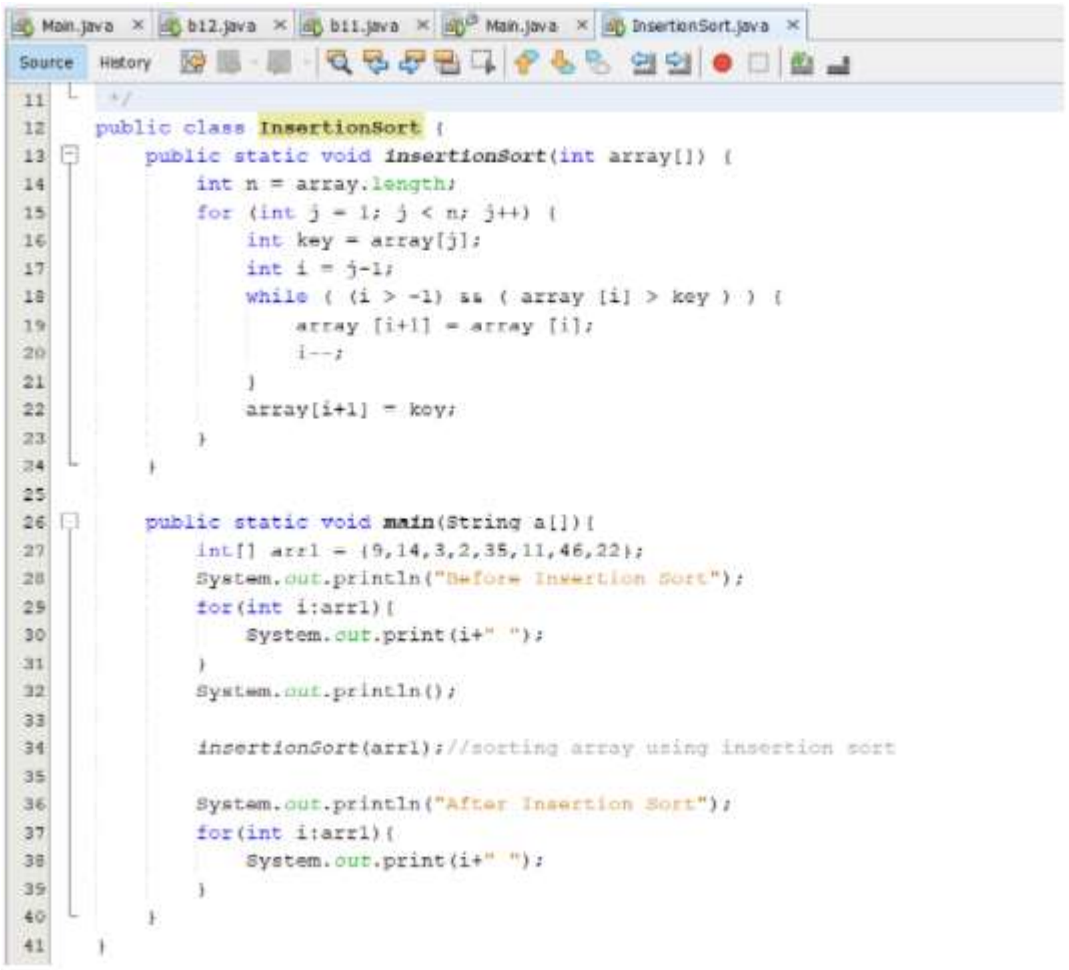


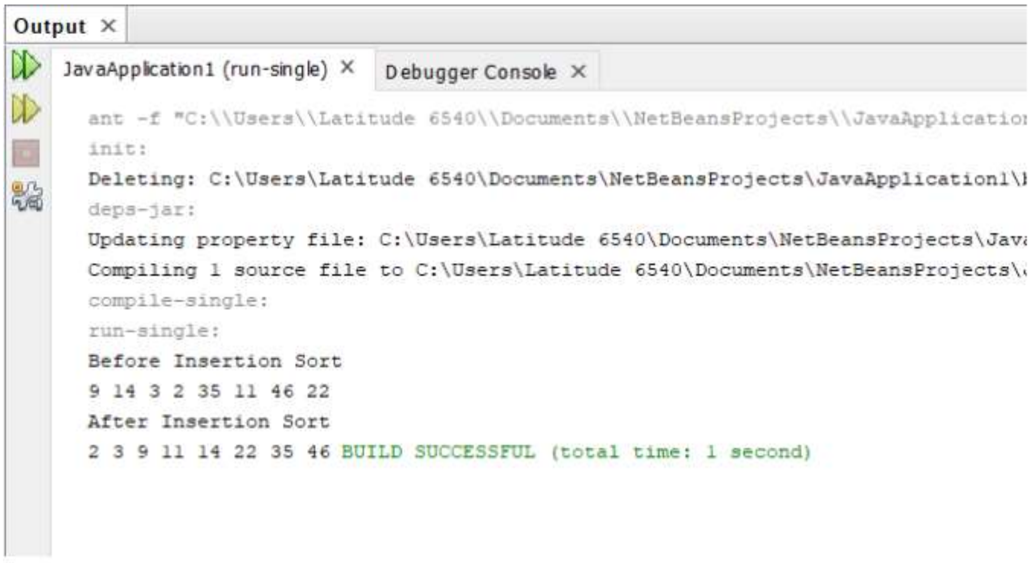
Figure 38. A graph showing the response of different time complexities as the number of element increase. Coefficients were choose so that the worst time complexities were the largest by the time n = 1000

- Here I have a code example of insertion sort for time complexity analysis:



```java
    */
public class InsertionSort {
    public static void insertionSort(int array[]) {
        int n = array.length;
        for (int j = 1; j < n; j++) {
            int key = array[j];
            int i = j-1;
            while ( (i > -1) && ( array [i] > key ) ) {
                array [i+1] = array [i];
                i--;
            }
            array[i+1] = key;
        }
    }

    public static void main(String a[]){
        int[] arr1 = {9,14,3,2,35,11,46,22};
        System.out.println("Before Insertion Sort");
        for(int i:arr1){
            System.out.print(i+" ");
        }
        System.out.println();

        insertionSort(arr1);//sorting array using insertion sort

        System.out.println("After Insertion Sort");
        for(int i:arr1){
            System.out.print(i+" ");
        }
    }
}
```

Figure 39. Code example



```
ant -f "C:\\Users\\Latitude 6540\\Documents\\NetBeansProjects\\JavaApplicatio
init:
Deleting: C:\Users\Latitude 6540\Documents\NetBeansProjects\JavaApplication1\
deps-jar:
Updating property file: C:\Users\Latitude 6540\Documents\NetBeansProjects\Java
Compiling 1 source file to C:\Users\Latitude 6540\Documents\NetBeansProjects\
compile-single:
run-single:
Before Insertion Sort
9 14 3 2 35 11 46 22
After Insertion Sort
2 3 9 11 14 22 35 46 BUILD SUCCESSFUL (total time: 1 second)
```

Figure 40. Output

- **Time complex:**

- Worst Case Complexity: O(n2)
    - Suppose, an array is in ascending order, and you want to sort it in descending order. In this case, worst case complexity occurs.
    - Each element has to be compared with each of the other elements so, for every nth element, (n-1) number of comparisons are made.
- Best Case Complexity: O(n)
    - When the array is already sorted, the outer loop runs for n number of times whereas the inner loop does not run at all. + So, there are only n number of comparisons. Thus, complexity is linear.
- Average Case Complexity: O(n2)
    - It occurs when the elements of an array are in jumbled order (neither ascending nor descending).
- Space Complexity
    - Space complexity is O(1) because an extra variable key is used.

2. Spatial Complexity

- Spatial complexity is the total amount of memory used by an algorithm/program including the space of the input values for execution. Therefore, to find space-complexity, it is sufficient to calculate the space occupied by the variables used in an algorithm/program.
- Similar to Time Complexity, Space Complexity is often expressed asymptotically in Big O Notation such as O (1), O (log n), O(n), O (n log n), O(n2), O(n3), O(2n), O(n!) where n is a characteristic of the input influencing space complexity and O is the worst-case scenario growth rate function.
- For any algorithm, memory is used for the following purpose:
    - To store variables (constant value, temporary value).
    - To store program instructions (or steps).
    - For the program to execute.
- Mathematically if we input inequation, the space complexity can be defined as,

Space Complexity = Auxiliary Space + Input space

- In most cases, Auxiliary Space is confused with Space Complexity. However, the Auxiliary Space is the additional space or temporary space used by the algorithm in its execution.
- When a program is under executed, it uses computing device memory for three main reasons:
    - *Instruction Space*: When code is compiled, we need to store it in memory somewhere, so that it can then be executed. The Instruction Space the place is where it is stored.

- *Environmental Stack*: It is used to store the addresses of partially called functions. It means, sometimes an algorithm (function) can be called inside another algorithm (function).
    - Example: If function A () calls function B () inside it, all variables of function A () will be stored on the system stack temporarily, while function B () is called and executed inside function A ().
- *Data Space*: A place to store data, variables, and constants of the program, and it is updated during execution. When we want to perform an analysis of an algorithm based on its Space complexity, we usually consider only Data Spaces and ignore the Instruction Spaces as well as the Environment Stack. That means we only calculate the memory required to store Variables, Constants, Structures,...

- Below is a list of some common Spatial complexity terms. It is arranged in order of their execution time when their input size increases.

S.NO	Big O Notation	Name
1	O (1)	Constant Space Complexity
2	O (log n)	Logarithmic Space Complexity
3	O (n)	Linear Space complexity
4	O (n log n)	Linearithmic Space Complexity
5	O (n^2)	Quadratic Space Complexity
6	O (n^3)	Cubic Space Complexity
7	O (n^y)	Polynomial Space Complexity
8	O (2^n)	Exponential Space Complexity
9	O (n!)	Factorial Space Complexity

Table 1.3: Space Complexity Table

- For better understanding take a look at the chart below, which is a combination of all charts for different Space Complexities. We can clearly see with the increase in the input data set how space complexity varies.

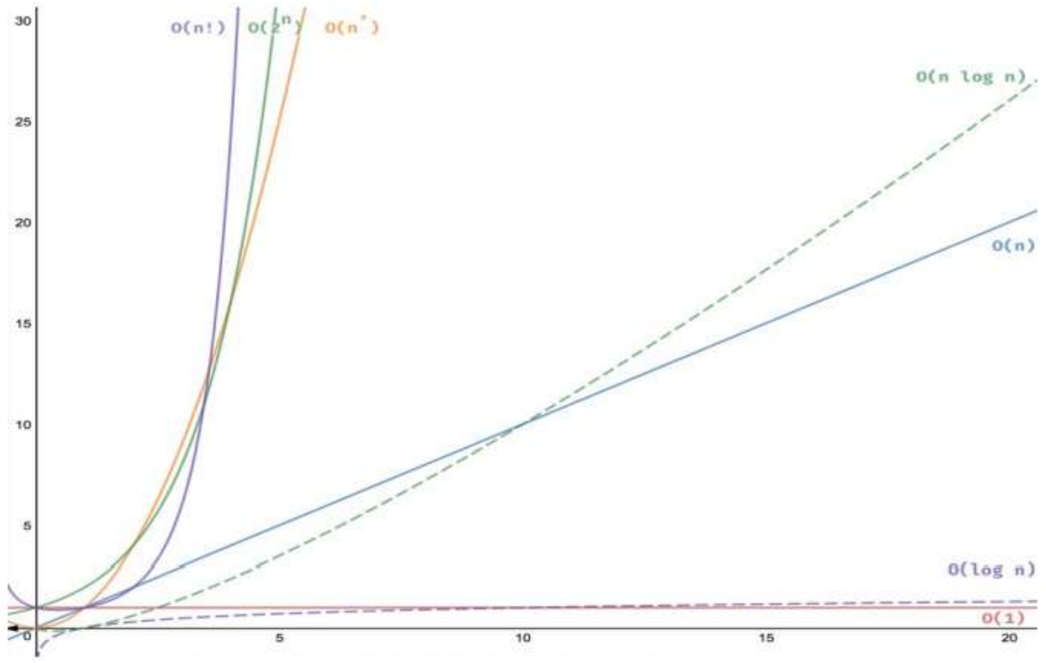


Figure 41. Space complexity

- To calculate the space complexity, we need to know the value of memory used by different types of data type variables, which often varies for different operating systems, but the method of calculating complexity the space is still the same. Example - In general (not always) C Programming Language compilers require the following:

Type	Size
bool, char, unsigned char, signed char, __int8	1 byte
__int16, short, unsigned short	2 bytes
float, __int32, int, unsigned int, long, unsigned long	4 bytes
double, __int64, long double, long	8 bytes

Figure 42. Type of data

- Now I will take an *example* and decode how space-time complexity is calculated:

```
int k = a + b + c;
```

- In the above expression, the variables a, b, c, and z are all integer types, so they will occupy 4 bytes each, so the total memory requirement will be (4 (4) + 4) = 20 bytes. This additional 4 bytes is for the return value. And because this spatial requirement is fixed for the example above, it is therefore called Constant Space Complexity.

- If any algorithm requires a fixed amount of space for all input values then that spatial complexity is said to be Constant Space Complexity or O (1) Space Complexity.

C. CONCLUDED

To summarize, in this exercise, I worked on complex algorithms and data structures by building a software chat system using stack and queue ADTs. Then I Perform the error handling and report the test results and use the asymptotic analysis technique used to evaluate the algorithm performance.

D. REFERENCES

www.javatpoint.com. 2021. *Exception Handling in Java | Java Exceptions - javatpoint*. [online] Available at: <https://www.javatpoint.com/exception-handling-in-java> [Accessed 11 August 2021].

Runestone.academy. 2020. 3.3. Big-O Notation — Problem Solving With Algorithms And Data Structures. [online] Available at: <https://runestone.academy/runestone/books/published/pythonds/AlgorithmAnalysis/BigONotation.html> [Accessed 1 March 2021].

Kumer, B., 2020. Space Complexity. [Online] Available at: <https://dev.to/bks1242/space-complexity-529b> [Accessed 3 3 2021].

ResearchGate. 2020. How Can I Measure The Performance Of An Algorithm?. [online] Available at: <https://www.researchgate.net/post/How_can_I_measure_the_performance_of_an_algorithm>[Accessed 3 March 2021]

HackerEarth. 2021. *Time and Space Complexity Tutorials & Notes | Basic Programming | HackerEarth*. [online] Available at: <https://www.hackerearth.com/practice/basic-programming/complexity-analysis/time-and-space-complexity/tutorial/> [Accessed 18 August 2021].