Java Arrays

1. Declaring an Array

```
int[] numbers;
```

2. Initializing an Array

a. Specify the size

```
int[] numbers = new int[5]; // Creates an array of size 5 with default values (0 for int)
```

b. Initialize with values

```
int[] numbers = {10, 20, 30, 40, 50}; // below is a screenshot from my code
```

```
public class Main {
    public static void main(String[] args) {
//        Scanner scanner = new Scanner(System.in);
        //--------------- Arrays----------
        int[] array = {6,2,3,4,5,8,9,8,8};
        for(int num : array)
        {
        System.out.println(num);
        }
        for (int i =0;  i < Array.getLength(array); i++)
        {
            System.out.println("Element Number " + (i + 1) + " = " + array
        }
```

Implementing Dynamic Arrays

A **dynamic array** is a resizable array that grows as needed when new elements are added. Unlike a fixed-size array, which has a set capacity, a dynamic array can expand or shrink during runtime.

How It Works

1. **Start with a Fixed Array**
   a. Create an array with an initial capacity.
2. **Resize When Full**
   a. If the array is full and a new element is added, create a **new larger array** (usually double the size).
   b. Copy existing elements to the new array.
   c. Add the new element.
3. **Shrink When Necessary**
   a. If too many elements are removed, the array can shrink to save memory.

Implementation in Java

```java
        // -------------- Dynamic Arrays ---------------
        String userInput;
        ArrayList<Integer> dynamicArray = new ArrayList<>();
        while (true) {
            System.out.println("Please Enter the array element value:");
            int num = scanner.nextInt();
            dynamicArray.add(num);

            while (true) {
                System.out.println("Would you like to add another element? (y/n)");
                userInput = scanner.next().trim().toLowerCase();

                if (userInput.equals("y") || userInput.equals("n")) {
                    break;
                }
                System.out.println("Invalid input. Please enter 'y' for Yes or 'n' for No.");
            }

            if (userInput.equals("n")) {
                break;
            }
        }

        System.out.println("Elements in the dynamic array:");
        for (int number : dynamicArray) {
            System.out.println(number);
        }
        scanner.close();
```

.

Implementing a Linked List

A **linked list** is a data structure that stores elements in **nodes**, where each node contains a value and a reference (or link) to the next node. Unlike arrays, linked lists do not use continuous memory and can grow dynamically.

How It Works

1. **Nodes and Pointers**
    a. Each node has two parts:
        i. **Data** (stores the value)
        ii. **Next** (points to the next node)
2. **Insertion**
    a. To add a new node, adjust the links so the new node connects to the list.
3. **Deletion**
    a. To remove a node, update the previous node's pointer to skip the deleted node.

Implementation in Java

```java
public class Node {   7 usages
    int data;   4 usages
    Node next;   13 usages
    public Node(int data)   1 usage
    {
        this.data=data;
        this.next=null;
    }
}
```

نظرا لأن التعامل مع برنامج الورد طلع صعب بقيت الكود في الصفحة التانية

```java
public class LinkedList {  2 usages
    private Node head;  9 usages
    public void insert(int data) {  1 usage
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            return;
        }
        Node temp = head;
        while (temp.next != null) {
            temp = temp.next;
        }
        temp.next = newNode;
    }
    public void delete(int data){  1 usage
        if (head==null) return;
        if(head.data == data){
            head = head.next;
            return;
        }
        Node temp = head;
        while(temp.next != null && temp.next.data != data){
            temp = temp.next;
        }
        if(temp.next != null){
            temp.next= temp.next.next;
        }
    }
    public void display() {  1 usage
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data + " -> ");
            temp = temp.next;
        }
        System.out.println("null");
    }
}
```

.

Implementing a Stack

A **stack** is a data structure that follows the **Last In, First Out (LIFO)** principle, meaning elements are added and removed from the same end (**top**).

How It Works

1. **Push (Insertion)** – Add an element to the top.
2. **Pop (Removal)** – Remove the top element.
3. **Peek (Top Element)** – View the top element without removing it.

Implementation

```java
class Stack {
    private Node top;

    public void push(int data) {
        Node newNode = new Node(data);
        newNode.next = top;
        top = newNode;
    }

    public void pop() {
        if (top == null) {
            System.out.println("Stack is empty");
            return;
        }
        top = top.next;
    }

    public void display() {
        Node temp = top;
        while (temp != null) {
            System.out.print(temp.data + " -> ");
            temp = temp.next;
        }
        System.out.println("null");
    }
}
```

Implementing a Queue

A **queue** is a data structure that follows the **First In, First Out (FIFO)** principle, meaning elements are added from one end (**rear**) and removed from the other (**front**).

How It Works

1. **Enqueue (Insertion)** – Add an element to the rear.
2. **Dequeue (Removal)** – Remove an element from the front.
3. **Peek (Front Element)** – View the front element without removing it.

Implementation

```java
class Queue {
    private Node front, rear;

    public void enqueue(int data) {
        Node newNode = new Node(data);
        if (rear == null) {
            front = rear = newNode;
            return;
        }
        rear.next = newNode;
        rear = newNode;
    }

    public void dequeue() {
        if (front == null) {
            System.out.println("Queue is empty");
            return;
        }
        front = front.next;
        if (front == null) rear = null;
    }

    public void display() {
        Node temp = front;
        while (temp != null) {
            System.out.print(temp.data + " <- ");
            temp = temp.next;
        }
        System.out.println("null");
    }
}
```

Implementing a Map

A **map** is a data structure that stores key-value pairs, allowing efficient retrieval of values using unique keys. It is useful for scenarios where fast lookups, insertions, and deletions are needed.

How It Works

- **Put (Insertion):** Add a key-value pair.
- **Get (Retrieval):** Retrieve a value using a key.
- **Remove (Deletion):** Delete a key-value pair.
- **Contains (Check):** Verify if a key exists.

Implementation in Java

Using **HashMap**:

```
//------------------------ Maps ---------------------
HashMap<String, Integer> map = new HashMap<>();

// Inserting key-value
map.put("Noah", 25);
map.put("Mohamed", 30);
map.put("Ahmed", 22);

// Retrieving a value
System.out.println("Noah's age: " + map.get("Noah")); // Output: 25

// Removing a key-value pair
map.remove( key: "Ahmed");

// Checking if a key exists
System.out.println("Contains Mohamed? " + map.containsKey("Mohamed")); // Outp

// Iterating through the map
for (String key : map.keySet()) {
    System.out.println(key + " -> " + map.get(key));
}
```