Lecture 03

Outline

1. Stacks – Introduction
2. Applications of stacks
    a. String Reverse
    b. Page visited History in Web Browser
    c. Undo Sequence of Text Editor
    d. Recursive function calling
    e. Auxiliary data structure for Algorithms.
    f. Stack in memory for a process
3. Stack characteristics
    a. Insertion and deletion **restricted** from the **middle** and the front of the stack.
4. Stack operations
    a. Push
    b. Pop
    c. Peek
5. Uses of stacks
    a. Stack operations are built into the microprocessor.
    b. When a method is called, its return address and arguments are pushed onto a stack, and when it returns, they're popped off.
6. Stack Implementation
    a. Create the initial stack
    b. Create the push method
    c. Create the pop method
    d. Create the peek method
    e. Create the isEmpty methos
    f. Create the isFull method

supportbydv

Stacks - Introduction

- Stacks are data structures that allow access to **only one data item**. And it's the **last item inserted**.
- You cannot access the next-to-last item, without removing the last item.
- Assume you have some plates with printed text on them. Now you are going to make a stack of them.

- After making a stack from them you can only access the last plate (top plate). You cannot see the next to last plate text without removing the last plate.

- This is the exact logic we use to implement the stacks. We call it **LIFO** (**Last-In-First-Out**) principle.

📌 **Stacks – Introduction**

- A **stack** is a data structure that allows access to **only one item at a time**—the **last item inserted**.
- You **cannot access** the next-to-last item **without first removing the last item**.

🍽️ **The Plate Stack Analogy**

Imagine you have **several plates**, each with **text printed on them**. Now, you **stack them** on top of each other.

⬆ **You place plates one by one, forming a stack.**
⬇ **You can only remove the top plate first.**

✅ **Can you see the text on the last plate you placed?** – Yes!
❌ **Can you see the text on plates below without removing the top one?** – Nope!

This is exactly how **stacks work**. **You can only access the top item, and to reach the previous ones, you must remove items one by one.**

🔥 **The LIFO (Last-In-First-Out) Rule**

- The **last plate you placed** (Last-In) is the **first one you take out** (First-Out).
- This is called **LIFO (Last-In-First-Out).**

💡 **TL;DR:**

✅ **Stacks = A pile of plates** 🍽️
✅ **You can only remove the top plate first!**
✅ **LIFO = Last item placed is the first to be removed.**

Prepared by Dinindu Wishvajith

Applications of Stacks

String Reverse

- To reverse the string, we can use stacks.
- We'll first store string elements (characters) as a list.
- Assume the original string is hello.
- If we stack it in a stack, word by word, it will look like this.

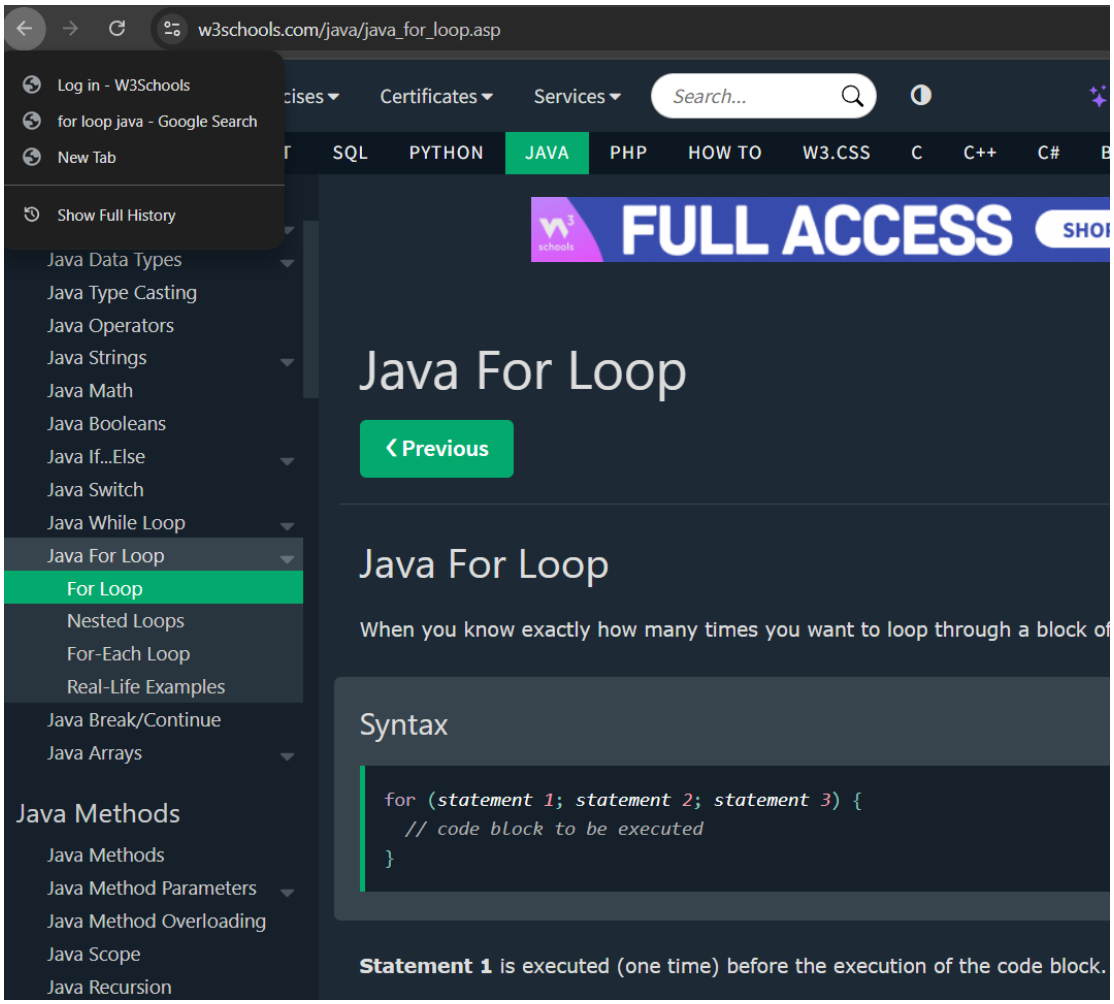
- Now we are going to remove the elements one by one starting from the top. We get,

1. O
2. L
3. L
4. E
5. H

- Using stacks, it's clear that we can reverse a string.

Page visited history in web browser

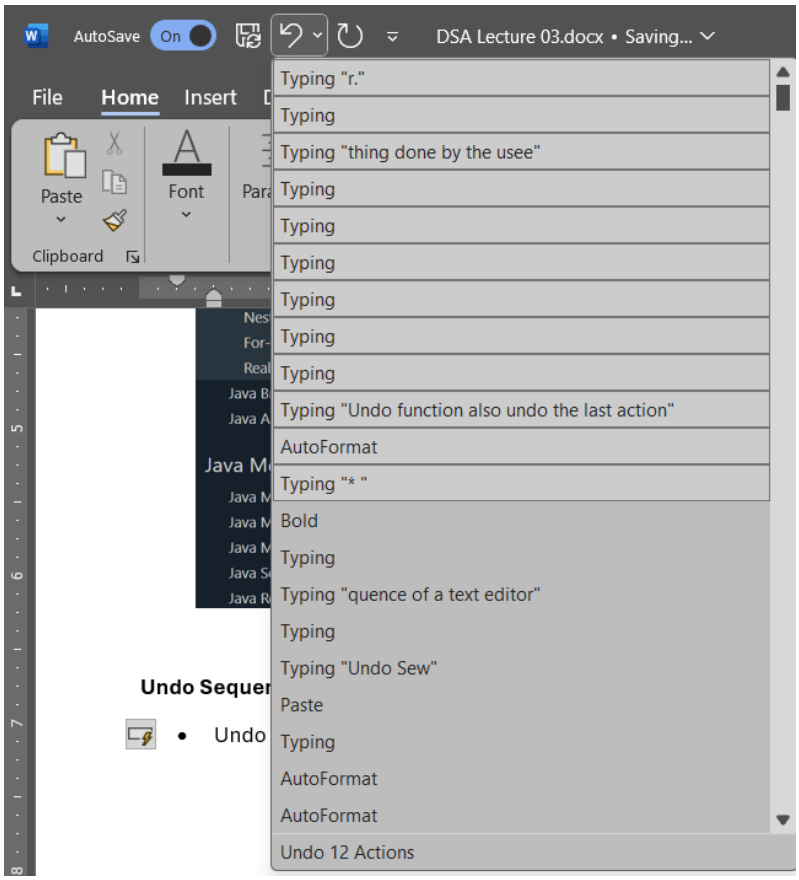
- When you press the back button in a web browser, you'll get redirected to the last visited web page.
- So, it's clear that web browsing history also uses stacks to function properly.



supportbydv

Undo Sequence of a text editor

- Undo function also undo the last thing done by the user.



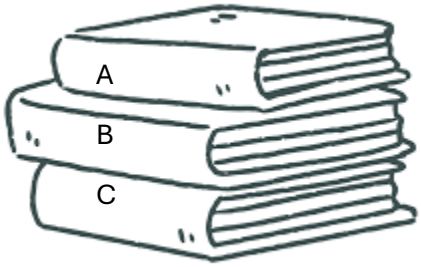
Prepared by Dinindu Wishvajith

Characteristics of Stacks

1. You cannot access the middle elements without removing the top elements.

   Tube of coins analogy
   Imagine you have a tube of pennies. You cannot grab the 10th coin directly even if you wanted to. You'll have to remove (pop) the top 9 coins to access the 10th coin. This is exactly how a stack works.

2. Elements Are Removed in Reverse Order of Insertion
   Book Pile Analogy
   Imagine you have three books named A, B and C.
   - You put book A on a table.
   - Then you put book B on top of book A.
   - Finally, you put book C on book B.



   - If you want to remove an element, it's going to be book C. Because you cannot remove other elements as book C lies top of them.

3. Elements Follow the LIFO (Last-In-First-Out) Principle.
   Pringle Can Analogy
   You open a fresh **can of Pringles** and try to grab the chip at the bottom.

   Can you pull it out directly? – No way!
   You have to take out the top chips first.
   - The **last chip placed (at the bottom) stays there until all the top ones are removed.**
   - **Last-In-First-Out (LIFO) means the last thing added is the first thing taken out.**

Characteristics of Queues – Summary – ChatGPT

✅ **Stack = Tube of Coins 🪙 → Remove top coins to reach the middle.**
✅ **Stack = Pile of Books 📚 → You remove them in reverse order.**
✅ **Stack = Pringles Can 🍟 → Last chip in is the first one out!**

supportbydv

Stack Operations

Push

Push means inserting an element into the stack. It's like putting a book on top of a pile of books.

Pop

Pop means removing an element from a list. It's like removing the top book from the pile of the book.

Peek

Peek is just looking at the top book without removing it from the pile. Which means accessing the topmost element from a stack without modifying it.

Visit W3Schools for interactive stack operations demonstration.

Prepared by Dinindu Wishvajith

Uses of Stacks

- **In microprocessors.**
  You don't have to know this in detail.

- **When a method is called, its return address and arguments are pushed onto a stack, and when it returns, they're popped off.**
  This is a bit complex. So, I'm not going to explain it. (No seriously! I get it but it's hard to explain.)

StackExample.java

```java
public class StackExample {
    public static void main(String[] args) {
        System.out.println("Main starts");
        functionA();
        System.out.println("Main ends");
    }

    public static void functionA() {
        System.out.println("Inside functionA");
        functionB();
        System.out.println("Exiting functionA");
    }

    public static void functionB() {
        System.out.println("Inside functionB");
    }
}
```

Output

```
Main starts
Inside functionA
Inside functionB
Exiting functionA
Main ends

=== Code Execution Successful ===
```

Step	What's Pushed to the Stack?	Stack Contents
main() starts	Push main() return address	main()
Calls functionA()	Push functionA() return address	functionA(), main()
Calls functionB()	Push functionB() return address	functionB(), functionA(), main()
functionB() ends	Pop functionB()	functionA(), main()
functionA() ends	Pop functionA()	main()
main() ends	Pop main()	(Empty)

- First it goes into the main method and prints "**Main Starts**". Now the stack looks like this.

Main()

- After that, it'll execute functionA. Which means functionA will get pushed into the top of the stack.

functionA
main()

- functionA first statement is print "**Inside functionA**". After that it'll go into functionB. Which means, it'll push the functionB on top of the stack.

functionB
functionA
main()

- Now, functionB will execute. It'll print "**Inside functionB**". After that stack's last item (functionB) will get removed (popped). Now the stack looks like this.

functionA
main()

- Now the rest of functionA will get executed (prints "**Exiting functionA**"), and functionA gets popped out by the list. Now the stack looks like this.

main()

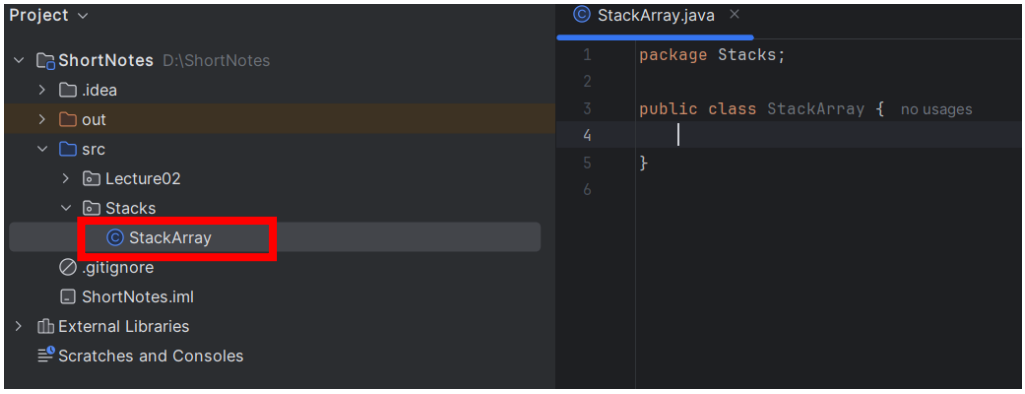
- After that the main method will get executed, which prints "**Main ends**".

⚠️ The actual concept is a bit different. But this would be enough to get the basic idea.

Implementing a stack

Create the Initial Stack

Now you know the necessary theory parts of stacks. It's time to implement a stack using java. First you have to create a java class. In my case I'll name it as **StackArray.java**.



Now, think! What do you need to create a stack?

- An array to store elements
- Top element (As it's the only element that gets modified)
- The capacity of the stack (As java arrays have fixed lengths).

These are the three values that you need to create a stack in Java. Now how do we initialize them?

- The array to store elements
  The size of array must be provided by the user/ program. So, we'll use a constructor that takes the capacity as an argument and create an array with that capacity.

- Top element
  We'll initialize this as -1. It'll help us to check whether the stack is empty or not. (Top == -1 means stack is empty).
  Every time we add a value we'll do a postfix increment, so the index starts with 0 without any problems.

- The capacity of the stacks
  As we discussed we'll get this from the user as an argument.

```java
package Stacks;

public class StackArray {
    private int[] stack;
    private int top;
    private int maxCap;

    public StackArray(int maxCap){
        stack = new int[maxCap];
        this.maxCap = maxCap;
        top = -1;
    }
}
```

Create the push method

What do we need to do when an item gets pushed into the stack?

- First, we need to validate whether the stack is full or not. If the stack is full, we cannot push an item into the stack.
- You see, index starts from 0. If the size of the array is 7, the last index of the array is 6.
- Now as we use **top** as the last element of the stack, we can simply use the logic,
    - `top == maxCap - 1`

- We'll first create the isFull method, so we can use it for validate push method.

```
public boolean isFull(){
    return top == maxCap - 1;
}
```

- What should happen when we push an item?
    - The item we push should go to the top of the array (last index).
        - The index of top should be incremented by one.
    - The push should not affect other stored values (Should not replace them).
        - The index of top must be incremented by one, before value got inserted into the stack.
- Let's code this down.

```
public void push(int item){
    if (isFull()){
        System.out.println("The stack is full!");
    }
    else {
        stack[++top] = item;
    }
}
```

Create the pop method

The pop method is used to retrieve an item and remove it from the stack.

However, we must do some validation before retrieving an item.

- The stack should not be empty. If it's empty, we cannot get any value from it.
- As we initialized top value as -1, we'll use that value to check whether the stack is empty or not. For that we'll implement isEmpty method.
    - top == -1

```
public boolean isEmpty(){
    return top == -1;
}
```

- With the help of isEmpty method, we'll validate the pop method and return the item.
- The code segment is as follows.

```
public int pop(){
    if (isEmpty()){
        System.out.println("The stack is empty!");
        return -1;
    }
    else{
        return stack[top--];
    }
}
```

- We are doing a postfix increment. So, this code will:
    - Return stack[top] value first
    - Then it'll do a decrement operation on **top** attribute.

Create the peek method

The Peek method is almost the same as the pop method. But it will not modify the stack.

- Therefore, we'll just copy paste and modify our pop method to implement the peek method.

```
public int peek(){
    if (isEmpty()){
        System.out.println("The stack is empty!");
        return -1;
    }
    else{
        return stack[top];
    }
}
```

- As you can see, the only difference is not doing a postfix decrement operation to the top value.

Extra Method – Print a stack

This is not a general stack method. But as it could help you to debug the problems, I'll just add it to this note.

```
public void printStack(){
    for (int i = top; i >= 0; i--){
        System.out.print(stack[i]);
        System.out.print(" ");
    }
    System.out.println();
}
```

To check the stack function, you can use this main method.

```
public static void main(String[] args) {
    StackArray newStack = new StackArray(5);
    newStack.push(10);
    newStack.push(20);
    newStack.push(30);

    System.out.print("The original stack: ");
    newStack.printStack();

    System.out.print("Popped element: ");
    System.out.println(newStack.pop());

    System.out.print("The updated stack: ");
    newStack.printStack();

    System.out.print("Peek into stack: ");
    System.out.println(newStack.peek());

    System.out.print("Stack after peeking: ");
    newStack.printStack();
}
```

```
The original stack: 30 20 10
Popped element: 30
The updated stack: 20 10
Peek into stack: 20
Stack after peeking: 20 10

Process finished with exit code 0
```

supportbydv

Full Code

```
package Stacks;

public class StackArray {
    private int[] stack;
    private int top;
    private int maxCap;

    public StackArray(int maxCap){
        stack = new int[maxCap];
        this.maxCap = maxCap;
        top = -1;
    }

    public boolean isFull(){
        return top == maxCap - 1;
    }

    public void push(int item){
        if (isFull()){
            System.out.println("The stack is full!");
        }
        else {
            stack[++top] = item;
        }
    }

    public boolean isEmpty(){
        return top == -1;
    }

    public int pop(){
        if (isEmpty()){
            System.out.println("The stack is empty!");
            return -1;
        }
        else{
            return stack[top--];
        }
    }

    public int peek(){
        if (isEmpty()){
            System.out.println("The stack is empty!");
            return -1;
        }
        else{
            return stack[top];
        }
    }

    public void printStack(){
        for (int i = top; i >= 0; i--){
            System.out.print(stack[i]);
            System.out.print(" ");
        }
```

Prepared by Dinindu Wishvajith

```
        System.out.println();
    }

    public static void main(String[] args) {
        StackArray newStack = new StackArray(5);
        newStack.push(10);
        newStack.push(20);
        newStack.push(30);

        System.out.print("The original stack: ");
        newStack.printStack();

        System.out.print("Popped element: ");
        System.out.println(newStack.pop());

        System.out.print("The updated stack: ");
        newStack.printStack();

        System.out.print("Peek into stack: ");
        System.out.println(newStack.peek());

        System.out.print("Stack after peeking: ");
        newStack.printStack();
    }
}
```