

CLASS 24 NOTES

We will now begin with some elementary data structures:

Let's start with a definition,

Definition:

A **data structure** is an ordered collection of data together with a set of operations that manipulate the data

Examples:

A String is a data structure. The data is a sequence of characters. The operations are the String methods such as `substring(...)`, `charAt(...)`, `length()` etc.

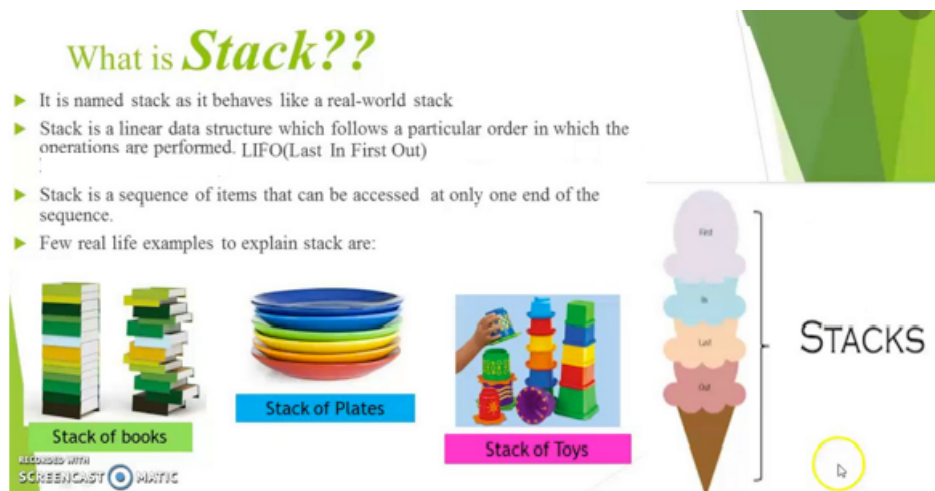
An ArrayList is a data structure. The data is an ordered collection of objects. The operations are `add(..)`, `size()`, `remove(..)` etc -- all the ArrayList methods

We are going to look at a new data structure called a **stack**. In fact, you have seen a stack before on one of the homework problems, it is a LIFO list. Last in, first out.

Definition:

A **stack** is an ordered collection of data into which data can be inserted and from which data can be removed—at one end called the top.

Here is something I ripped from the web:



When you add an item to a stack you place the item on the top. Look at the pictures above. If you want to add another plate or scoop of ice cream you add it to the top. When you remove a plate, you take the top plate. Or when you eat the ice cream (remove), you eat the top scoop – you can't eat the bottom scoop first. Look at the toys (above) the mystery hand places a new toy on top.

Example: Here is a stack of integers

Insert 3, insert 5, insert 7 Places 3, then 5 and then 7 on the stack 7 is at the top	insert 3 insert 5 insert 7	<div>7 5 3</div>	top
remove → removes the top element (7) Leaving 3 and 5, with 5 now at the top	remove	<div>5 3</div>	top

There are just five operations that manipulate a stack. They are

- push(x) → places x on the top of the stack (insert)
- pop() → removes and returns the top item (remove)
- empty() → returns true if there is nothing on the stack; otherwise returns false
- peek() → returns the top item of the stack but **does not remove it**
- size() → returns the number of items in the stack

Example: Imagine you have a stack of Strings, S.

<p>push("Dopey") push ("Doc") push("Happy")</p> <div data-bbox="516 275 696 495"> <div>"Happy"</div> <div>"Doc"</div> <div>"Dopey"</div> </div> <p>top</p> <p>S</p>	<p>Three push operations. "Happy" is the top item.</p>
<p>String name = pop()</p> <div data-bbox="480 602 617 768"> <div>"Doc"</div> <div>"Dopey"</div> </div> <p>top</p> <p>S</p> <div data-bbox="760 606 862 653"> <div>"Happy"</div> </div> <p>name</p>	<p>The pop() operation removes and returns the top item. "Happy" is removed and stored in the String variable, <i>name</i>. Now "Doc" is the top item.</p>
<p>push ("Sleepy")</p> <div data-bbox="506 825 690 1050"> <div>"Sleepy"</div> <div>"Doc"</div> <div>"Dopey"</div> </div> <p>top</p> <p>S</p>	<p>push("Sleepy") places "Sleepy" on the stack—the top item</p>
<p>String x = peek()</p> <div data-bbox="548 1140 716 1341"> <div>"Sleepy"</div> <div>"Doc"</div> <div>"Dopey"</div> </div> <p>top</p> <p>S</p> <div data-bbox="886 1161 1044 1207"> <div>"Sleepy"</div> </div> <p>x</p>	<p>peek() lets you view the top item but does not remove it. peek() just returns the top item ("Sleepy") but the stack is unchanged</p>
<p>print (size())</p> <div data-bbox="420 1411 594 1621"> <div>"Sleepy"</div> <div>"Doc"</div> <div>"Dopey"</div> </div> <p>top</p> <p>S</p>	<p>Prints 3 because there are three items in the stack</p>

IMPORTANT: when using a stack in a program you have access only to the top item. This is a LIFO list → Last in, First Out

Example: A stack application

Suppose an expression or String or even a Java program allows three types of parentheses/braces/brackets : () , { } , or []. Here is an algorithm, using a stack, that determines whether or not parens/braces/brackets are balanced.

For example: Are the parens/braces/brackets of following string are balanced

([(a+b)] [{ x- y}])

But in this next expression they are not:

[xyx { abc }]

Here is an algorithm that determines whether or not the parentheses or braces of an expression are balanced.

- Scan the expression left to right, character by character
- When a **left** paren, brace, or bracket is encountered, push it on the stack
- When a right paren, brace, or bracket is encountered
 - if the stack is empty → report error (unbalanced) and stop
 - else pop the stack, if the popped symbol is a match -- { } () or [] -- continue otherwise report error (unmatched item) and stop
- The expression is balanced if the stack is empty and all input is consumed

In short, the algorithm just says

Read the expression one character at a time, left to right.

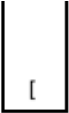








When you read a left paren/brace/bracket, push it.

When you read a right paren/brace/ bracket pop the stack,|pop if not empty




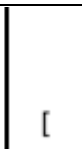


If it matches keep going, if it does not match (or the stack was empty) stop and report an unbalanced expression.

If you have read all the data and the stack is empty the expression is balanced.

Example : Determine whether or not the expression [(abc)+{xyz}] is balanced.
The red character is the current character, i.e. the character being read

[(abc)+{xyz}] → read and push [
[(abc)+ {xyz}] → read and push (
[(abc)+ {xyz}] Read abc and do nothing with the stack	
[(abc)+ {xyz}] read) -- it is a right paren Pop the stack and get (. That matches) so continue	
[abc)+ {xyz}] Read + and do nothing with the stack	
[abc)+ {xyz}] read and push {	
[abc)+ {xyz}] Read xyz and do nothing with the stack	
[abc)+ {xyz}] read } Pop the stack and get {. That matches) so continue	
[abc)+ {xyz}] read] Pop the stack and get [. That matches). Stack is empty, all data is read so the expression is balanced	

Here is another example : Is [(abc)+xyz] balanced ?

[(abc)+xyz] → read and push [
[(abc)+ xyz}] → read and push (
[(abc)+ xyz}] Read abc and do nothing with the stack	
[(abc)+ xyz}] read) -- it is a right paren Pop the stack and get (That matches) so continue	
[abc)+ xyz}] Read + xyz and do nothing with the stack	
[abc)+ xyz}] read }. It is a right brace. Pop the stack and get [. This is not a match for { Error → unbalanced expression. Done	

Implementation of a Stack

Unlike data structures String or ArrayList, which Java provides, we will have to write our own Stack class. You did a simple version of this when you did the LIFO list for homework.

So we must decide

- How will we store the data of a stack
- How do we implement the methods push(..), pop(), peek(), size() and empty()

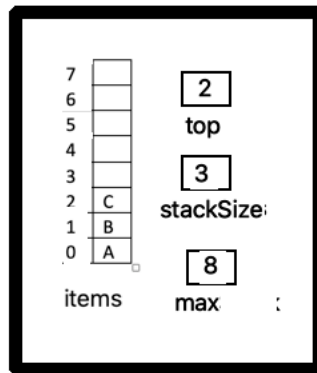
There are several ways we can do this. We will start with the simplest.

The Array Implementation

We will

- Store the data in an array
- Keep track of the number of data -- for the stack size
- Keep track of the array index of the top item
- Keep track of the maximum size – remember arrays are of a fixed size

Here is how you might visualize a Stack object using an array



- There is an array called **items** that is used to hold the data of the stack. Here items is indexed 0..7. (There are three data currently on the stack .)
- There is a variable **top** (int) that stores the index of the top item. In this case the top item is “C” and top is 2. The top item (“C”) is at items[2].
- There is a variable **stackSize** (int) that stores the number of items stored in the stack. In this case it is 3. The stack contains “A”, “B”, and “C” -- three data
- There is a variable **max** (int) that holds the maximum number of items that the stack can hold. In our picture the array can hold just 8 items, so max is 8. Remember an array has a fixed size, and cannot expand.

Now I will design a class, Stack<E>

So what is <E>? <E> is a generic...and we saw this when we used ArrayLists. Remember?

For Example

```
ArrayList<String> a = new ArrayList<String>(); // an ArrayList of Strings
ArrayList<Integer> b = new ArrayList<Integer>(); // an ArrayList of Integer objects
ArrayList<Person> c = new ArrayList<Person>() // an ArrayList of Person objects
```

Generics allows an ArrayList to hold any type of **object** (but not primitives)

We will use this same idea with our Stack class. We will design a Stack class that can hold any type of object. So we will be able to make lots of different kinds of stacks:

```
Stack<String> x = new Stack<String>(); // a stack of Strings
Stack<Integer> y = new Stack<Integer>(); // a stack of Integer objects
Stack<Person> z = new Stack<Person>(); // a stack of Person objects
```

Here is the code. Make sure you understand how it works

```
class Stack<E>
//E is a placeholder for the type of object we store on the stack e.g Person, String
{
    private E[] items;           // an array to hold data
    private int top;             // the index of the top item
    private int stackSize;       // the number of items stored on the stack
    int max;                     // the maximum size of the stack

    public Stack() // default constructor. I'll explain the red statements after the code
    {
        items = (E[]) new Object [10]; // make the array, capacity is 10.
        top = -1;                       // denotes an empty stack...no items ;
        stackSize = 0;
        max = 10;
    }

    public Stack(int max) //one argument constructor, creates a stack of capacity max
    {
        items = (E[]) new Object [max]; // make the array items of size max
        top = -1;
        stackSize = 0;
        this.max = max;
    }

    public void push(E x) // places x on the stack
    {
        if (stackSize == max) // stack is full, no more room
        {
            System.out.println("push:Stack Overflow");
            System.exit(0);
        }
        top++; // change top to the next available position
        items[top] = x; // place x into items[top]
        stackSize++; // increase the size of the stack
    }
}
```



```

    }

    public E pop() // removes the top item and returns it
    {
        if (stackSize == 0) // if the stack is empty , cannot pop anything
        {
            System.out.println("pop::Stack Underflow");
            System.exit(0);
        }
        E topItem = items[top]; // get the top item and store it in a temporary variable
        top--;                  // change top to the previous position
        stackSize--;            // reduce the size
        return topItem;         //return the item that was on top
    }

    public E peek()          // just returns the top item, does not remove it
    {
        if (stackSize == 0) // if the stack is empty, you cannot peek.
        {
            System.out.println("peek:Stack Underflow");
            System.exit(0);
        }
        return items[top]; // just return the top item...nothing on the stack changes
    }

    public boolean empty() // true if the stack is empty
    {
        return stackSize == 0; // this expression is true or false
    }
    public int size() // returns the number of items on the stack
    {
        return stackSize;
    }
}

```

In the constructors there is a line in red:

items = (E[]) new Object [10]; // capacity is 10.

This line creates the array items .

You might think that you should create the array as

items = new E[10]; // capacity is 10.

But Java insists that you

1. Create an array of **Object** : new Object[10]
2. And the downcast it to an array of E : new **(E[])** Object[10]

Now here is a class that uses a stack (to do nothing)

You MUST put **objects** on a stack, for example, Integer not int

```

public class UseStack
{
    public static void main(String[] args)
    {
        Stack<Integer> s = new Stack<Integer>(8); // max is 8
        s.push(4); // 4 will be autoboxed to an Integer
        s.push(5);
        s.push(6);
        int y = s.pop(); // Integer 6 popped and unboxed
    }
}

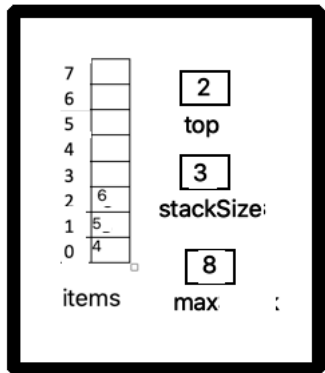
```

```

        System.out.println(y+ " was just popped");
        System.out.println(" Size is " + s.size());
        System.out.println("Top item is now "+s.peek());
        System.out.println("Size is "+s.size());
    }
}

```

After the first three push statements the stack looks like this



The output is

6 was just popped

// 6 was popped and stored in y

Size is 2

// the stack size is now 2 and 5 is on top

Top item is now 5

// peek just returns the top item. Stack unchanged

Size is 2

// size is still 2, peek does not alter the stack

ArrayList Implementation of a Stack Data is stored in an ArrayList

The main problem with the array implementation of a Stack is that **the size is fixed**.

However if we store the data in an ArrayList, rather than an array, the ArrayList (and hence, the stack) will expand as needed.

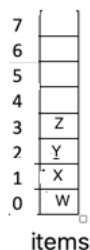
We will use the ArrayList methods:

- add(x) → adds x to the **end** of the list
- remove(i) → removes and returns the i^{th} item.
- get(i) → returns the i^{th} item
- size() → returns the number of items in the ArrayList

NOTE:

If stack data is stored in an ArrayList. **Then the index of the top item is $\text{size}() - 1$**

For example. Here is a stack stored in an ArrayList, items



The size of the stack is 4 and the top position is 3. That is $\text{top} = \text{items.size()} - 1 = 4 - 1 = 3$

```
class Stack<E>
{
    private ArrayList<E> items;    // data is stored in an ArrayList

    public Stack() // default constructor; creates an empty stack
    {
        items = new ArrayList<E>(); // initial capacity is 10, the default
    }

    public Stack(int n)
    //one argument constructor, creates a stack with initial capacity n, but can grow
        items = new ArrayList<E>(n);
    }

    public void push(E x) // building the stack from position 0 upward
    {
        items.add(x); //uses the ArrayList method (add(x)) that adds to the end—that's it
    }

    public boolean empty()
    {
        return items.isEmpty();    //uses the ArrayList method isEmpty()
    }

    public E pop()
    {
        if (empty())    // determine whether or not there is an item to remove
        {
            System.out.println("pop:Stack underflow");
            System.exit(0);
        }
        // the top of the stack is at position items.size() -1
        int top = items.size() - 1;    // top is one less than the size
        return items.remove(top); //uses the ArrayList method remove(int n)
    }

    public int size()
    {
        return items.size();    //uses the ArrayList method size()
    }

    public E peek()
    {
        if (empty()) // determine whether or not there is an item on the stack
        {
            System.out.println("peek:stack underflow");
            System.exit(0);
        }
        // top item is stored at position items.size() -1
        int top = items.size() - 1;
        return items.get(top); //uses the ArrayList method get(int i)
    }
}
```

 You should understand the data structure stack as a LIFO list and

- The five basic operations: push(x), pop(), peek(), size(), empty()
- The two ways we can implement a stack:
 - a. using an array to store the data
 - b. using an ArrayList to store the data

For the next two checkpoint problems you can download the stack class from the class website. (other stuff). You can use either the array or ArrayList implementation

Example:

Given an expression such as ((a+b)*[[4+4]]+ {abc} with three types of brackets. Are the brackets properly formatted, i.e. balanced

((a+b)*[[4+4]]+ {abc} -- yes

{{a+b}*[[4+4]]+ {abc} -- no

<pre> import java.util.*; public class Brackets { public static boolean match (char ch1, char ch2) // checks whether two brackest match { if (ch1 == '{' ch2 == '}') return true; if (ch1 == '[' ch2 == ']') return true; if (ch1 == '(' ch2 == ')') return true; return false; } public static boolean isBalanced(String expression) { Stack<Character> stack = new Stack<Character>(100); // must be a stack of objects for (int i = 0; i < expression.length(); i++) { char ch = expression.charAt(i); if (ch == '{' ch == '[' ch == '(') stack.push(ch); // auto boxing else if (ch == '}' ch == ']' ch == ')') { if(stack.empty()) return false; else { char ch1 = stack.pop(); // unboxing if (!match (ch1,ch)) return false; } } } return true; } } </pre>	<p>This method accepts two characters and if they are balanced braces , returns true Otherwise returns false</p> <p>match ('(', ')') returns true match ('}', '[') returns false and match('{', ')'} returns false and match('a','b') returns false</p> <p>returns true if expression is balanced</p> <p>Notice Character not char. Stack needs objects</p> <p>For each character</p> <p>If the character is (or { or [push</p> <p>If char is) or } or]</p> <p>if the stack is empty – not balanced (false)</p> <p>else Pop the stack, if the current character and the popped character do not match return false</p> <p>Otherwise continue</p> <p>Finally return true</p>
---	---

```
public static void main(String[] args)
{
    Scanner input = new Scanner(System.in);
    System.out.print("Enter an experssion: ");
    String s = input.nextLine();

    while(!s.equals("xxx"))
    {
        if (isBalanced(s)) // returns true
            System.out.println("Balanced");
        else
            System.out.println(" Not Balanced");
        System.out.print("Enter an experssion: ");
        s = input.nextLine();
    }
}
```

You can use a stack to convert a decimal (base 10) number to binary (base 2).
For example, 7 in binary is 111, 20 in binary is 10100, and 103 is 1100111

The method is very simple. We will use a stack of Integer.
Suppose you want to convert an int num, such as 20, to its binary form. Here is how you start

```
while num is not equal to 0
{
    push num % 2 onto a stack // this will be 0 or 1
    let num = num/ 2
}
```

OK so suppose num = 20 (In binary form, 20 is 10100). Here is how the loop works.

num % 2 is $20\%2 = 0 \rightarrow$ push 0 num = num/2 or num = $20/2 = 10$ num is now 10				
			0	
num % 2 is $10\%2 = 0 \rightarrow$ push 0 num = num/2 or num = $10/2 = 5$ num is now 5				
			0	
			0	
num % 2 is $5\%2 = 1 \rightarrow$ push 1 num = num/2 or num = $5/2 = 2$ num is now 2				
			1	
			0	
			0	
num % 2 is $2\%2 = 0 \rightarrow$ push 0 num = num/2 or num = $2/2 = 1$ num is now 1				
			0	
			1	
			0	
num % 2 is $1\%2 = 1 \rightarrow$ push 1 num = num/2 or num = $1/2 = 0$ num is finally 0 --STOP				
			1	
			0	
			1	
			0	

Now, continually pop the stack , APPENDING DIGITS LEFT TO RIGHT, you get 10100, which is the binary value of 20

Here is the algorithm:

```
public static String toBinary(int num)
{
    Declare a stack of Integer
    while num is not 0
    {
        Push num%2 onto the stack
        Let num = num/2
    }

    String binary= "";
    Pop the stack until it is empty appending the digits to binary
    Return binary
}
```

Here is the code color-coded with the algorithm

```
public class Binary
{
    public static String convertToBinary(int num)
    {
        Stack <Integer> s = new Stack<Integer>(100);
        while (num != 0)
        {
            int digit = num%2;
            s.push(digit);
            num = num/2;
        }
        String binary = "";
        while (!s.empty())
            binary = binary+ s.pop();
        return binary;
    }
}
```

You can use this static method in another program by using the class name (Binary):

```
System.out.println(Binary.convertToBinary(20));
Or
String bin = Binary.convertToBinary(100)
```