The big picture... so far

DATA STRUCTURES

Array

Unordered Array

Ordered Array

Stacks

Queues

ALGORITHMS

insert, delete, linear search, shuffle, sorting

insert, delete, shuffle, binary search

push, pop, peek, etc.

insert, remove, etc.
Stacks and Queues - Philosophy

- They are abstract data structures defined only by their interfaces (examples: insert, remove, etc.)
- The underlying mechanism (for example: an array) is not visible to the user
- Interfaces are designed to enforce restricted access – only one item should be removed or added at a time.... designed to be O(1)
- These data structures are not appropriate to operate on database, they are used more often as programmer's tools
- Once created and used to carry out a particular task, they are often discarded (short life-time)
Stacks- overview

Last In First Out (LIFO)

Input

Output

C  
B  
A

Stacks allow to get access to the last item inserted. If removed, you get access to the next-to-last item inserted, and so on.

- Analogies:
  - Mail: open the first letter on top of the stack and work your way to the bottom
  - Re-prioritize tasks of your work day as they come along
  - Daily life examples: elevators, bus, etc.
- Simple but very useful in programming
Stacks- overview

- Adding an item on top of the stack → "Push"
- Removing an item from the top → "Pop"

Applications of stack (few examples):
- Reverse a string
- Parsing arithmetic expressions
- Undo operations in many editor tools
- Stack-based memory allocation (local variables in methods)
- Compilers (delimiters)
- Hardware stack-based architecture in CPU (memory registers)
Stacks- *undo* operation

- In an Excel file (for example), input your data in a row:
  - 100, 200, 300, 400
  - Find something wrong? Use *undo* to return to previous state

![Operations using a Stack](image)

- 400
- 300
- 200
- 100

![Operations using a Stack](image)
Stacks- reverse operations

- What do we get after Pop, Pop, Pop? …… CBA

Operations using a Stack

1. Push A
2. Push B
3. Push C
4. 1st Pop → C
5. 2nd Pop → B
6. 3rd Pop → A
Stacks- implementation

- One can use an array of object to implement a stack

```
maxSize-1
...
...
4
3
2
C
1
B
0
A
```

- **maxSize** is the capacity of the stack. Stacks are typically small.
- **top** is the index of the item on the top.
- A, B, C are primitive data (int, long, double, etc.) or objects.

List of basic operations using a Stack?

- push(item);
- pop();
- peek()
- isEmpty()
- isFull()
- size()

**To do:** Test Java applet Stack.html
Stacks- generic array implementation 1/3

- "Object" stands for any data types or objects

```java
class Stack {
    // variables
    private int maxSize; // Stack capacity initialized by user
    private int top; // top of stack
    private Object[] array; // array that holds the items

    // constructors
    public Stack(int maxSize) {
        this.maxSize = maxSize; // set array size
        array = new Object[maxSize]; // create array
        top = -1; // no item yet
    }
}
```
class Stack
{
    // variables

    // constructors

    // methods

    public boolean isEmpty()
    {
        return (top == -1); // true if stack is empty
    }

    public boolean isFull()
    {
        return (top == maxSize - 1); // true if stack is full
    }

    public int size()
    {
        return (top + 1); // return current number of items
    }
}
public Object peek() {
    if (!isEmpty()){
        return array[top]; // return top item
    } else {
        throw new IllegalStateException("Stack is empty");
    }
}

public Object pop() {
    if (!isEmpty()){
        top--; // decrement top
        return(array[top+1]); // return top item
    } else {
        throw new IllegalStateException("Stack is empty");
    }
}

public void push(Object item) {
    if (!isFull()){
        top++;
        array[top]= item; // insert item
    }
}
We consider a Stack composed of *int*, can you guess the result?

```java
class StackApp1 {
    public static void main(String[] args) {
        Stack mystack = new Stack(5);
        for (int i = 1; i <= 10; i++) mystack.push(i);
        mystack.pop();
        mystack.pop();
        System.out.print(mystack.size());
        System.out.print(mystack.peek());
        System.out.print(mystack.pop());
        System.out.print(mystack.peek());
    }
}
```

Result is 3332
Reversing a word using a Stack composed of `char`,

```java
class StackApp2 {
    public static void main(String[] args) {
        EasyIn easy = new EasyIn();
        String word = easy.readString();
        Stack mystack = new Stack(word.length());
        char ch;
        for(int i = 0; i < word.length(); i++) {
            ch = word.charAt(i);
            mystack.push(ch);
        }
        while(!mystack.isEmpty())
            System.out.print(mystack.pop());
    }
}
```
Stacks - Application III

- **Delimiter Matching (Bracket Balance)**
- **Examples:**
  - ‘[’ matches ‘]’, and ‘(’ matches ‘)’
  - Bracket matched: (a+b), [a/b], a/[(b-c)*d]
  - Bracket unmatched: a+b), m*(n]+k
- **How does the stack look like after each char is read from the String?**
  - Using a[b(c[d)e)f]
  - Character read: 'a', '[', 'b', '(', 'c', '[', 'd', ']', 'e', ')', 'f', ']'

Opening delimiters
push on the Stack

<table>
<thead>
<tr>
<th>push</th>
<th>push</th>
<th>push</th>
<th>pop check</th>
<th>pop check</th>
<th>pop check</th>
</tr>
</thead>
<tbody>
<tr>
<td>[</td>
<td>[</td>
<td>[</td>
<td>(</td>
<td>(</td>
<td>(</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
class StackApp3n {
    public static void main(String[] args) {
        EasyIn easy = new EasyIn();
        String word = easy.readString();
        Stack mystack = new Stack(word.length());
        char ch, cp;
        boolean test = true;
        int i = 0;
        while (test && i < word.length()) {
            ch = word.charAt(i);
            i++;
            if (ch == '(' || ch == '[') mystack.push(ch);
            else if (ch == ')' || ch == ']') {
                if (!mystack.isEmpty()) {
                    cp = (Character)mystack.pop();
                    if (((cp == '[' && ch != ']') || (cp == '(' && ch != ')'))
                        test = false;
                } else test = false;
            }
        }
        if (!(test) || (!mystack.isEmpty())) System.out.println("matching problem!");
    }
}