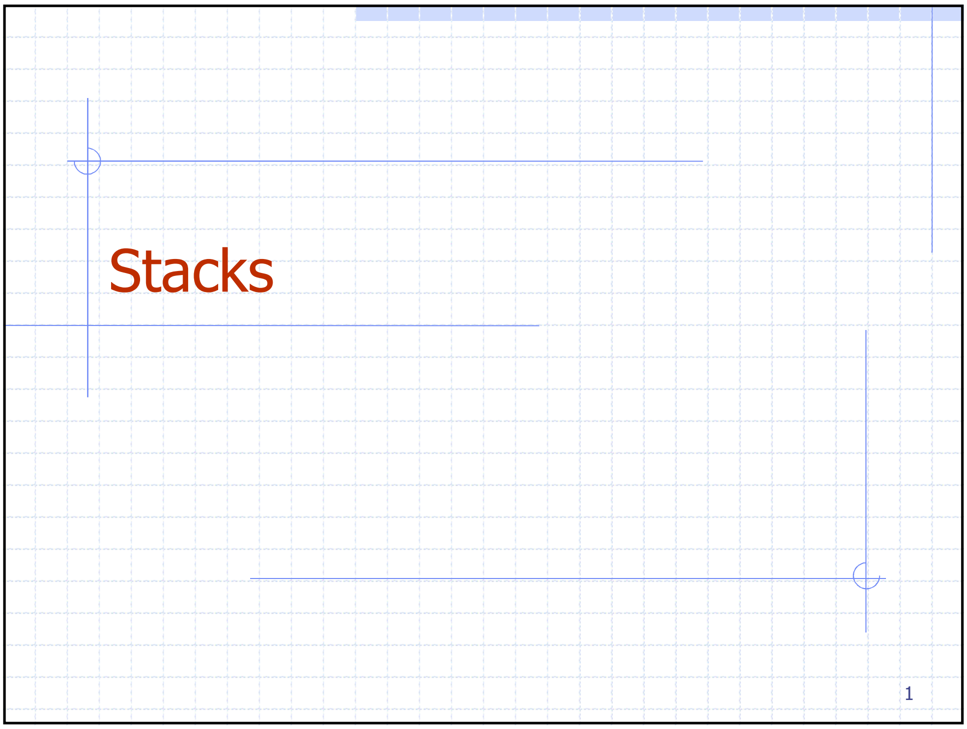
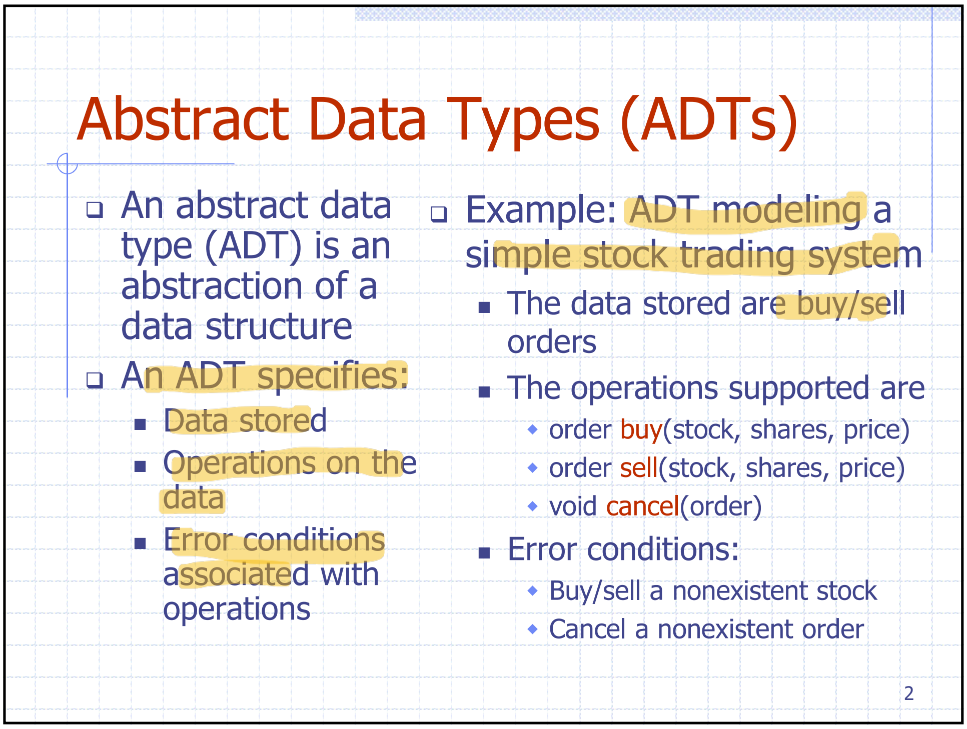


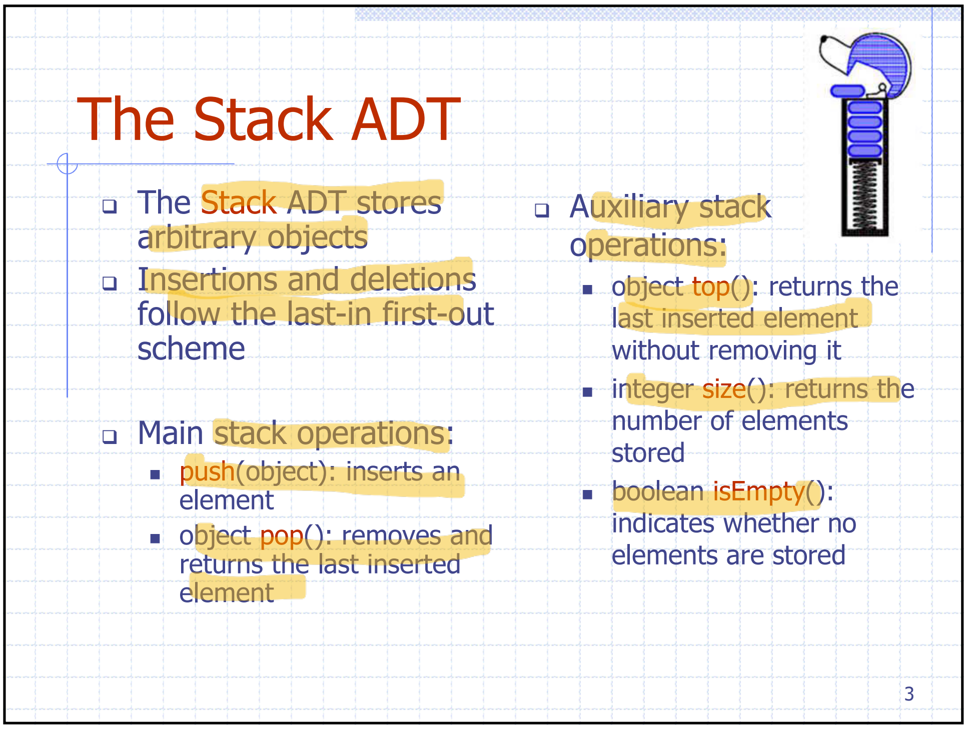
Stacks



Abstract Data Types (ADTs)

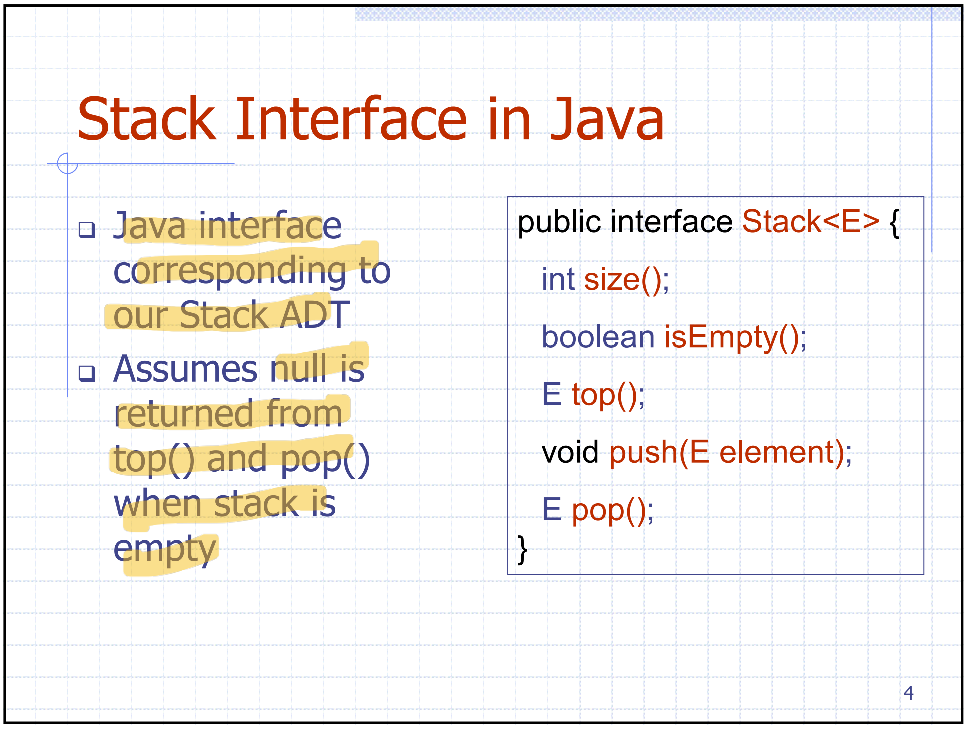
- An abstract data type (ADT) is an abstraction of a data structure
- An ADT specifies:
  - Data stored
  - Operations on the data
  - Error conditions associated with operations

- Example: ADT modeling a simple stock trading system
  - The data stored are buy/sell orders
  - The operations supported are
    - order buy(stock, shares, price)
    - order sell(stock, shares, price)
    - void cancel(order)
  - Error conditions:
    - Buy/sell a nonexistent stock
    - Cancel a nonexistent order



The Stack ADT

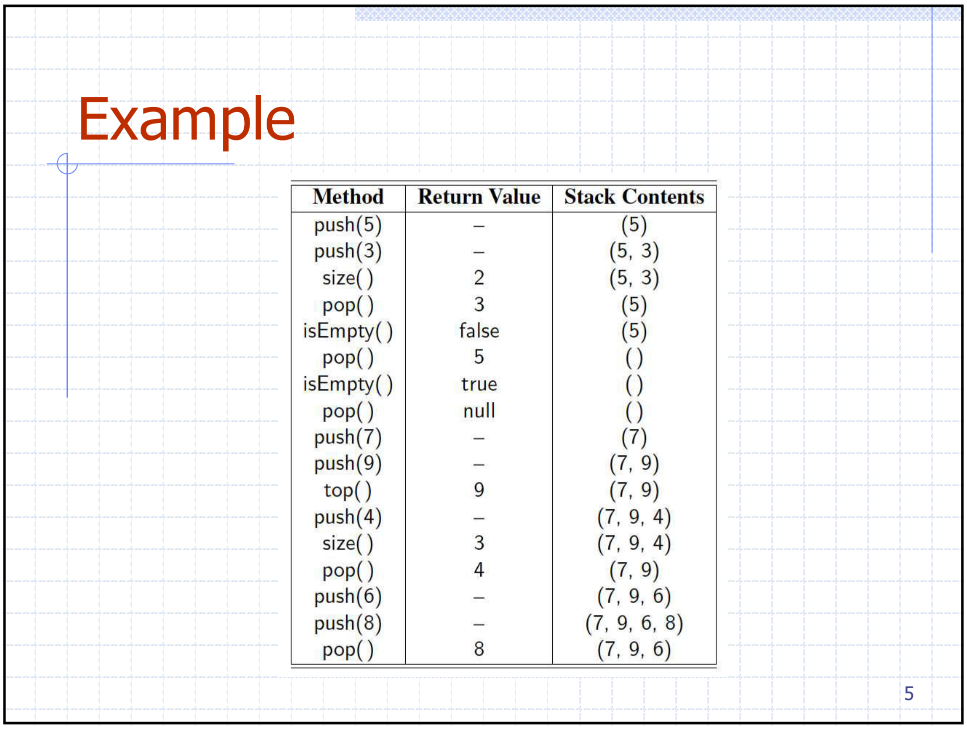
- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the last-in first-out scheme
- Main stack operations:
  - push(object): inserts an element
  - object pop(): removes and returns the last inserted element
- Auxiliary stack operations:
  - object top(): returns the last inserted element without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored



Stack Interface in Java

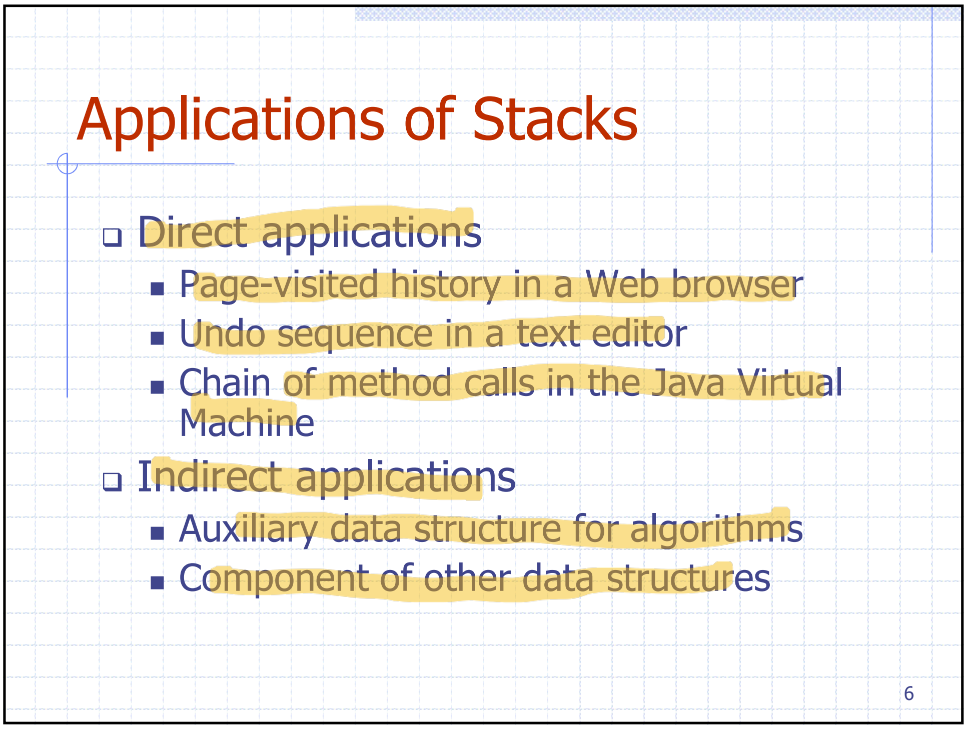
- Java interface corresponding to our Stack ADT
- Assumes null is returned from top() and pop() when stack is empty

```
public interface Stack<E> {
   int size();
   boolean isEmpty();
   E top();
   void push(E element);
   E pop();
}
```



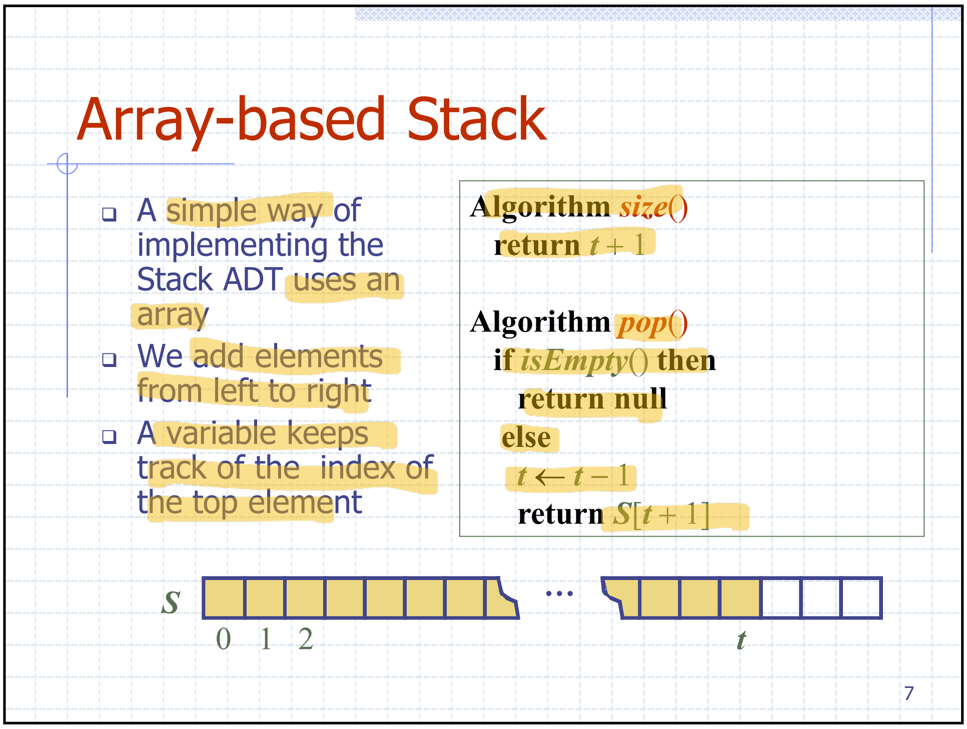
Example

Method	Return Value	Stack Contents
push(5)	–	(5)
push(3)	–	(5, 3)
size( )	2	(5, 3)
pop( )	3	(5)
isEmpty( )	false	(5)
pop( )	5	( )
isEmpty( )	true	( )
pop( )	null	( )
push(7)	–	(7)
push(9)	–	(7, 9)
top( )	9	(7, 9)
push(4)	–	(7, 9, 4)
size( )	3	(7, 9, 4)
pop( )	4	(7, 9)
push(6)	–	(7, 9, 6)
push(8)	–	(7, 9, 6, 8)
pop( )	8	(7, 9, 6)



Applications of Stacks

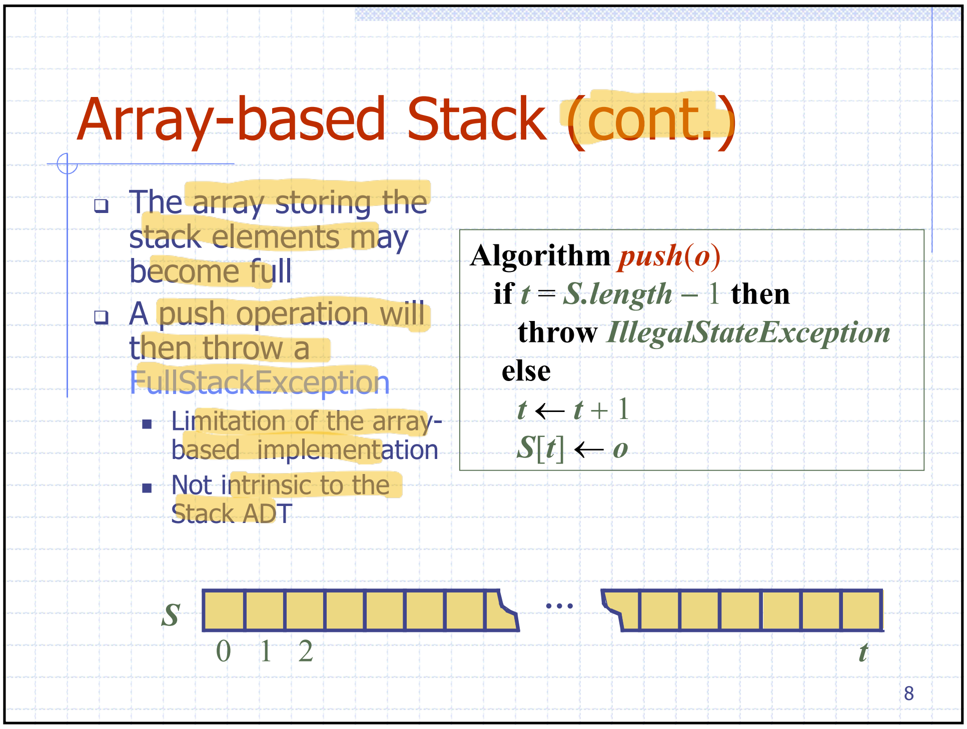
- ❑ Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
- ❑ Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures



Array-based Stack

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element

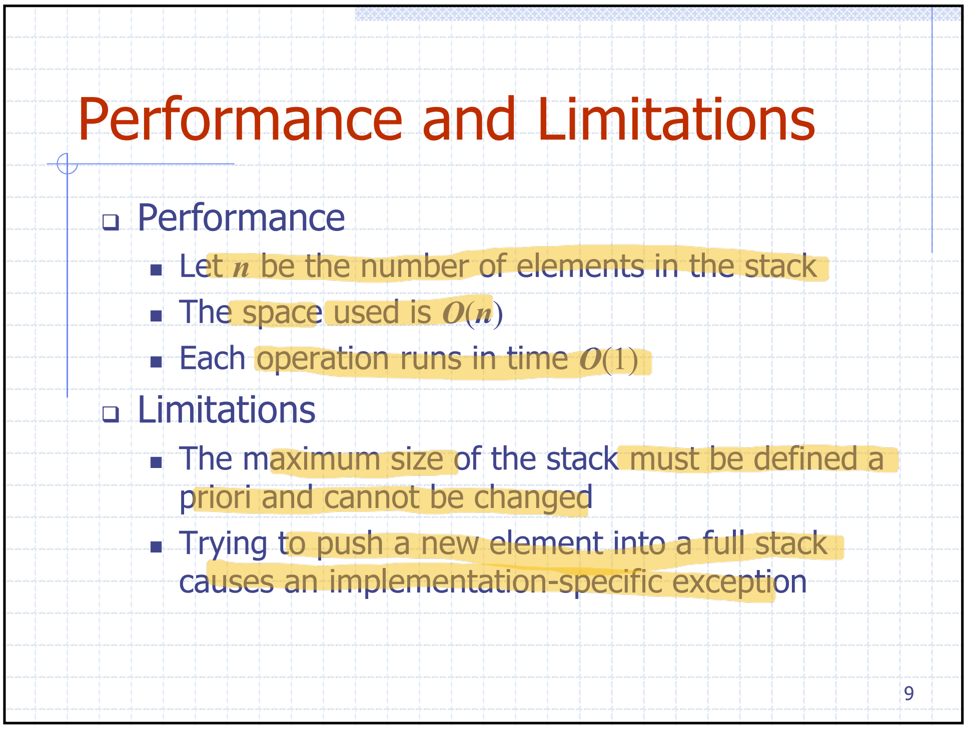
```
Algorithm size()
    return $t + 1$

Algorithm pop()
    if isEmpty() then
        return null
    else
        $t \leftarrow t - 1$
        return $S[t + 1]$
```



Array-based Stack (cont.)

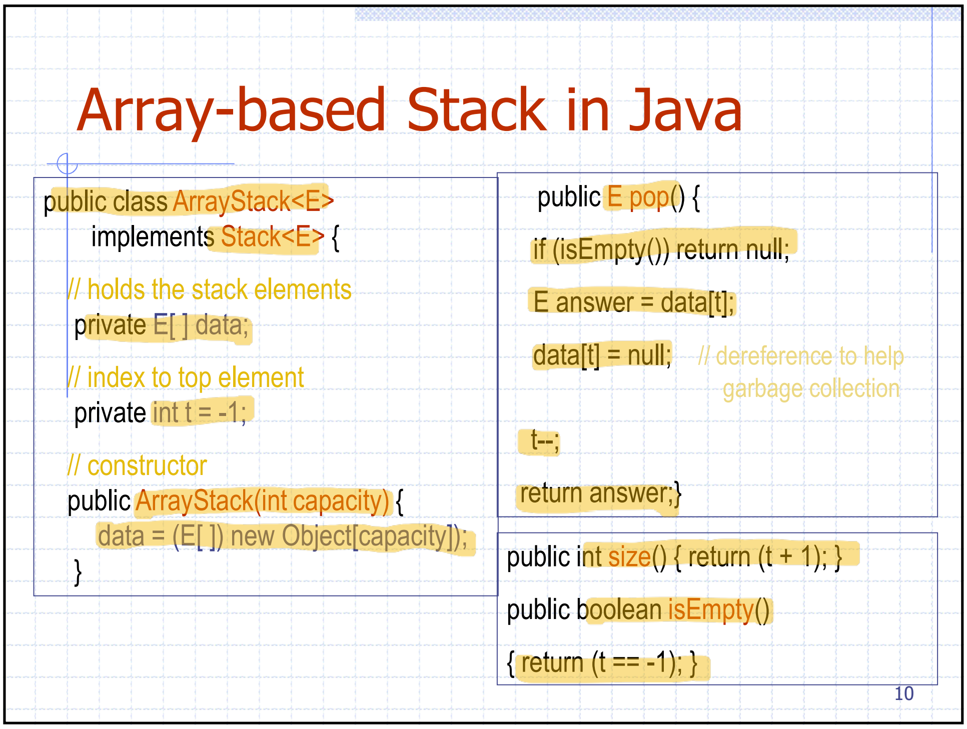
- The array storing the stack elements may become full
- A push operation will then throw a FullStackException
    - Limitation of the array-based implementation
    - Not intrinsic to the Stack ADT

```
Algorithm push(o)
    if $t = S.length - 1$ then
        throw IllegalStateException
    else
        $t \leftarrow t + 1$
        $S[t] \leftarrow o$
```



Performance and Limitations

- Performance
  - Let n be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$
- Limitations
  - The maximum size of the stack must be defined a priori and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific exception

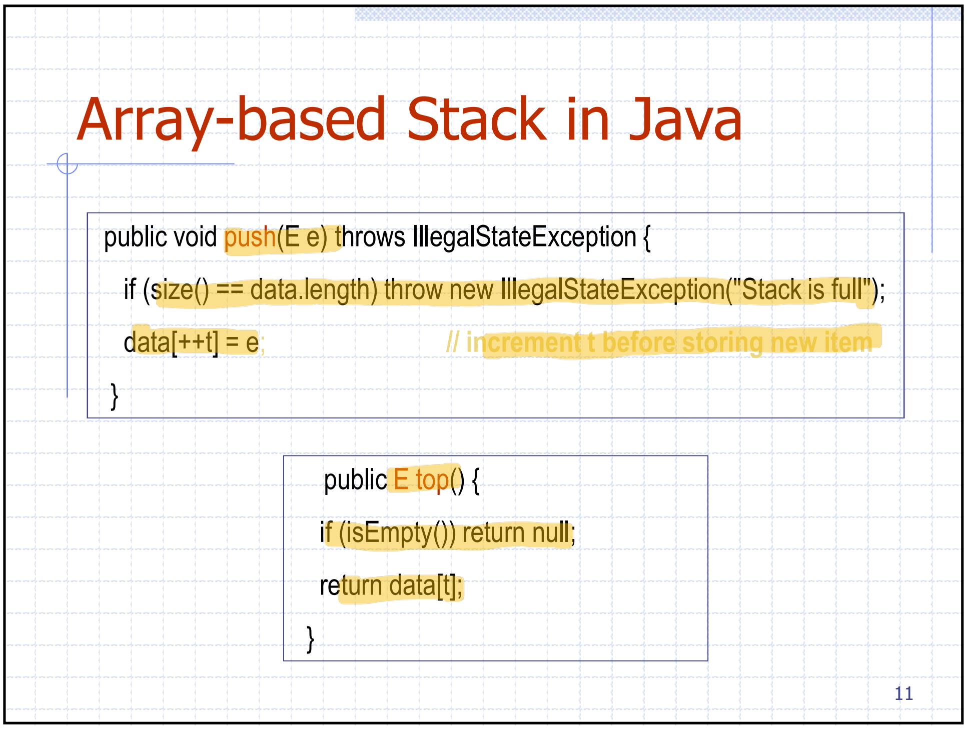


Array-based Stack in Java

```
public class ArrayStack<E>
        implements Stack<E> {

    // holds the stack elements
    private E[ ] data;

    // index to top element
    private int t = -1;

    // constructor
    public ArrayStack(int capacity) {
        data = (E[ ]) new Object[capacity]);
    }
```

```
public E pop() {
    if (isEmpty()) return null;
    E answer = data[t];
    data[t] = null;    // dereference to help
                          garbage collection
    t--;
    return answer;}
```

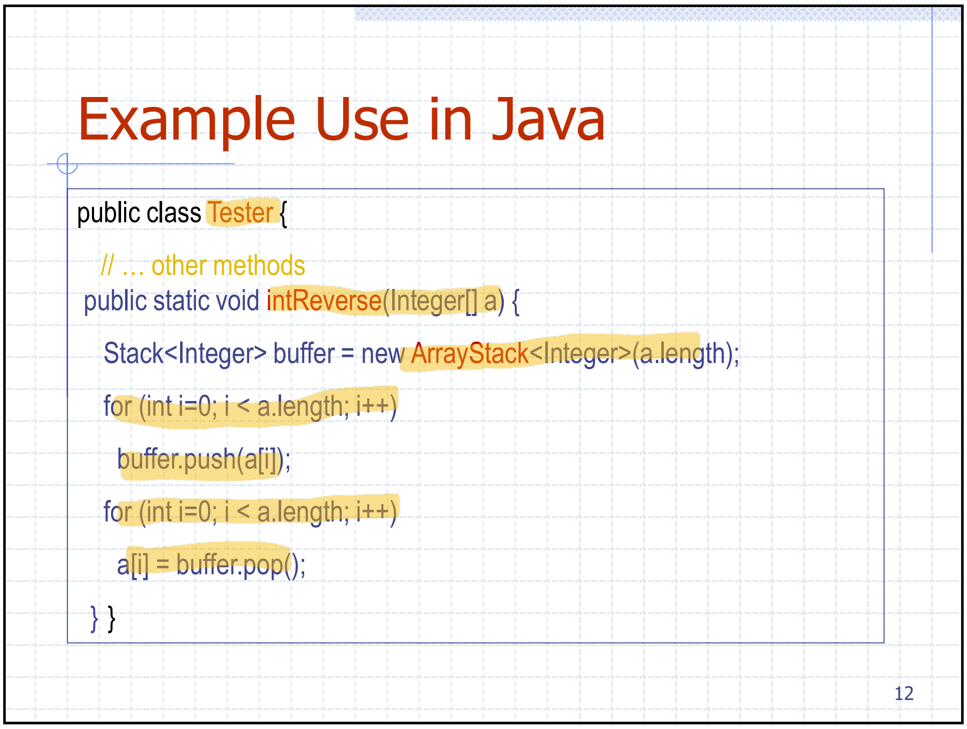
```
public int size() { return (t + 1); }
public boolean isEmpty()
{ return (t == -1); }
```



Array-based Stack in Java

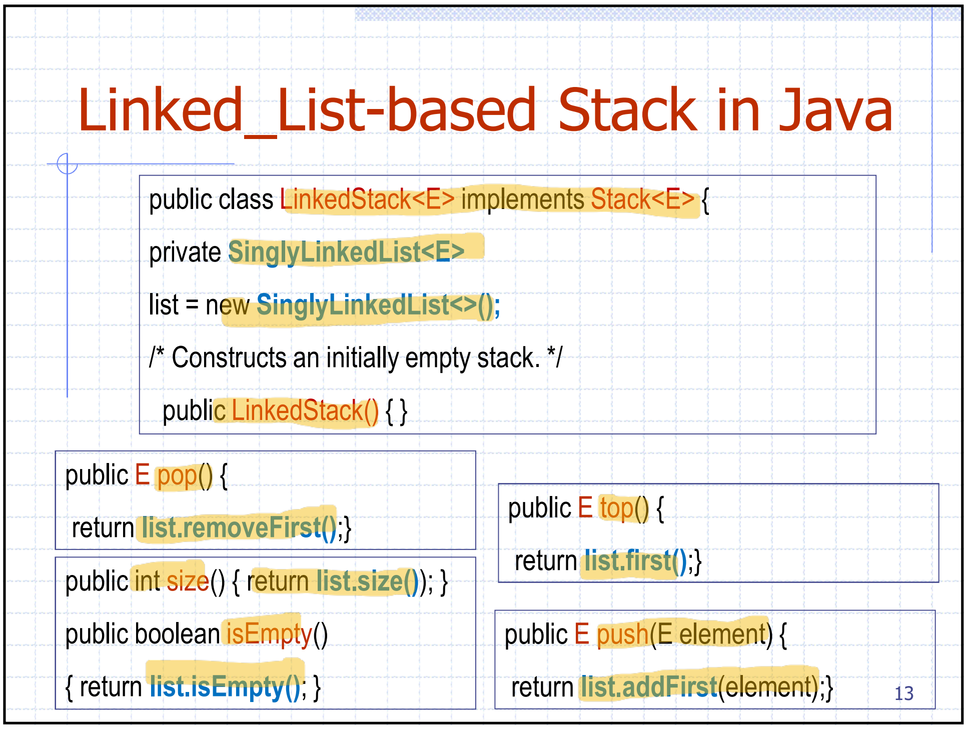
```
public void push(E e) throws IllegalStateException {
  if (size() == data.length) throw new IllegalStateException("Stack is full");
  data[++t] = e;                          // increment t before storing new item
 }
```

```
 public E top() {
 if (isEmpty()) return null;
 return data[t];
}
```



Example Use in Java

```
public class Tester {
  // … other methods
 public static void intReverse(Integer[] a) {
   Stack<Integer> buffer = new ArrayStack<Integer>(a.length);
   for (int i=0; i < a.length; i++)
     buffer.push(a[i]);
   for (int i=0; i < a.length; i++)
     a[i] = buffer.pop();
 } }
```



Linked_List-based Stack in Java

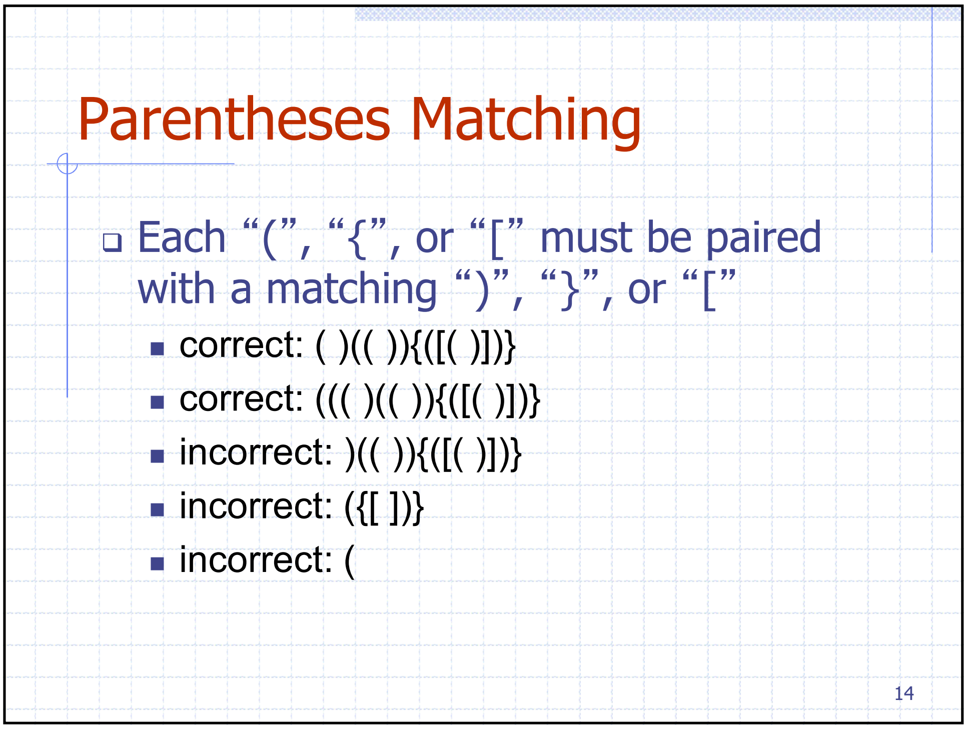
```
public class LinkedStack<E> implements Stack<E> {
private SinglyLinkedList<E>
list = new SinglyLinkedList<>();
/* Constructs an initially empty stack. */
  public LinkedStack() { }
```

```
public E pop() {
 return list.removeFirst();}
```

```
public E top() {
 return list.first();}
```

```
public int size() { return list.size()); }
public boolean isEmpty()
{ return list.isEmpty(); }
```

```
public E push(E element) {
 return list.addFirst(element);}
```



Parentheses Matching

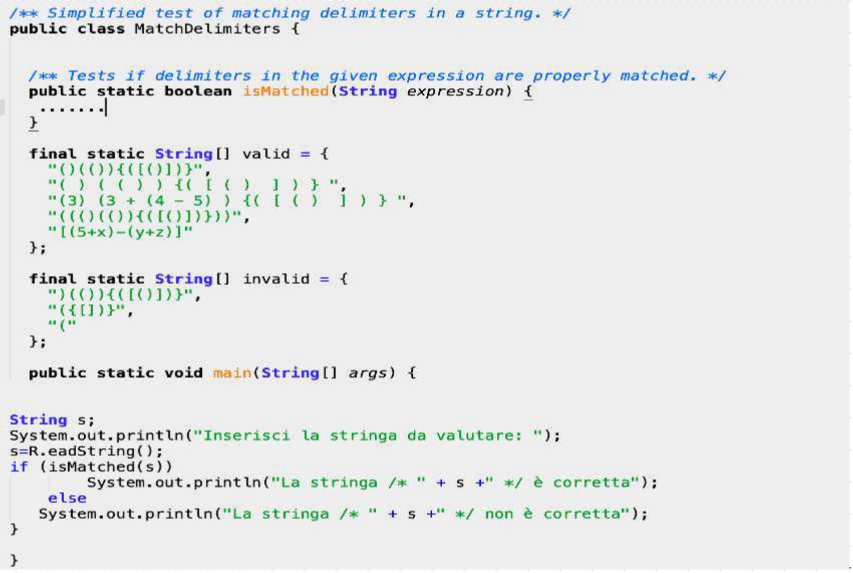
- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”
  - correct: ( )(( )){([( )])}
  - correct: ((( )(( )){([( )])}
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

Parenthesis Matching (Java)

```
public static boolean isMatched(String expression) {
  final String opening = "({["; // opening delimiters
  final String closing = ")}]"; // respective closing delimiters
  Stack<Character> buffer = new LinkedStack<>( );
  for (char c : expression.toCharArray( )) {
    if (opening.indexOf(c) != −1) // this is a left delimiter
      buffer.push(c);
    else if (closing.indexOf(c) != −1) { // this is a right delimiter
      if (buffer.isEmpty( )) // nothing to match with
        return false;
      if (closing.indexOf(c) != opening.indexOf(buffer.pop( )))
        return false; // mismatched delimiter
    }
  }
  return buffer.isEmpty( ); // were all opening delimiters matched?
}
```

Valore che estraggo è diverso rispetto a quello chiuso.

Parenthesis Matching (Java)



```
/** Simplified test of matching delimiters in a string. */
public class MatchDelimiters {


  /** Tests if delimiters in the given expression are properly matched. */
  public static boolean isMatched(String expression) {
   .......
  }

  final static String[] valid = {
    "()(()){([()])}",
    "( ) ( ( ) ) {( [ ( )  ] ) } ",
    "(3) (3 + (4 - 5) ) {( [ ( )  ] ) } ",
    "((()(()){([()])}))",
    "[(5+x)-(y+z)]"
  };

  final static String[] invalid = {
    ")(()){([()])}",
    "({[])}",
    "("
  };

  public static void main(String[] args) {


String s;
System.out.println("Inserisci la stringa da valutare: ");
s=R.eadString();
if (isMatched(s))
        System.out.println("La stringa /* " + s +" */ è corretta");
    else
   System.out.println("La stringa /* " + s +" */ non è corretta");
}

}
```