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

Stack Interface in Java

- Java interface corresponding to our Stack ADT
- Requires the definition of class EmptyStackException
- Different from the built-in Java class java.util.Stack

```java
public interface Stack {
    public int size();
    public boolean isEmpty();
    public Object top() throws EmptyStackException;
    public void push(Object o);
    public Object pop() throws EmptyStackException;
}
```
Exceptions

- Attempting the execution of an operation of ADT may sometimes cause an error condition, called an exception.
- Exceptions are said to be “thrown” by an operation that cannot be executed.
- In the Stack ADT, operations pop and top cannot be performed if the stack is empty.
- Attempting the execution of pop or top on an empty stack throws an EmptyStackException.

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

Method Stack in the JVM

- The Java Virtual Machine (JVM) keeps track of the chain of active methods with a stack.
- When a method is called, the JVM pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed.
- When a method ends, its frame is popped from the stack and control is passed to the method on top of the stack.
- Allows for recursion.

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
  throw EmptyStackException
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 FullStackException
else
    t ← t + 1
    S[t] ← o
```

---

Array-based Stack in Java

```java
public class ArrayStack implements Stack {
    // holds the stack elements
    private Object S[];
    // index to top element
    private int top = -1;
    // constructor
    public ArrayStack(int capacity) {
        S = new Object[capacity];
    }
    public Object pop() throws EmptyStackException {
        if isEmpty()
            throw new EmptyStackException("Empty stack: cannot pop");
        Object temp = S[top];
        S[top] = null;
        top = top – 1;
        return temp;
    }
}
```

---

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.

---

Parentheses Matching

- Each "(", "{", or "[" must be paired with a matching ")", "}", or "["
  - correct: ( )(( )){{(( ))}}
  - correct: ((( () ))){{( ))}}
  - incorrect: )(( )){( ))}}
  - incorrect: {{{ }}
  - incorrect: (}
Parentheses Matching Algorithm

**Algorithm ParenMatch(X,n):**

**Input:** An array X of n tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number

**Output:** true if and only if all the grouping symbols in X match

Let S be an empty stack

for i = 0 to n do

if X[i] is an opening grouping symbol then

S.push(X[i])

else if X[i] is a closing grouping symbol then

if S.isEmpty() then

return false (nothing to match with)

if S.pop() does not match the type of X[i] then

return false (wrong type)

if S.isEmpty() then

return true (every symbol matched)

else

return false (some symbols were never matched)

HTML Tag Matching

For fully-correct HTML, each <name> should pair with a matching </name>

The Little Boat

The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage. </p>
<ol>
<li>Will the salesman die?</li>
<li>What color is the boat?</li>
<li>And what about Naomi?</li>
</ol>
We show how to use a stack as an auxiliary data structure in an algorithm.

Given an array \( X \), the span \( S[i] \) of \( X[i] \) is the maximum number of consecutive elements \( X[j] \) immediately preceding \( X[i] \) and such that \( X[j] \leq X[i] \).

Spans have applications to financial analysis, e.g., stock at 52-week high.

**Quadratic Algorithm**

**Algorithm spans1(\( X, n \))**

- **Input**: array \( X \) of \( n \) integers
- **Output**: array \( S \) of spans of \( X \)
- **\( S \) ← new array of \( n \) integers**
- **\( S[i] ← 0 \) to \( n - 1 \) do**
  - **\( s ← 1 \) to \( n - i \)**
  - **while \( s \leq i \) and \( X[i-s] \leq X[i] \)**
    - **\( s ← s + 1 \) to \( n - i \)**
    - **\( S[i] ← s \) to \( n \)**
- **return \( S \)**

- **Algorithm spans1 runs in \( O(n^2) \) time**

**Linear Algorithm**

**Algorithm spans2(\( X, n \))**

- **\( S ← new array of \( n \) integers**
- **\( A ← new empty stack**
- **for \( i ← 0 \) to \( n - 1 \)**
  - **while \( ¬A.isEmpty() \) and \( X[A.top()] \leq X[i] \)**
    - **\( A.pop() \)**
  - **if \( A.isEmpty() \)**
    - **\( S[i] ← i \) to \( n \)**
  - **else**
    - **\( S[i] ← i - A.top() \)**
- **return \( S \)**

- **Algorithm spans2 runs in \( O(n) \) time**