CSE 373
Data Structures and Algorithms

Lecture 1: Introduction; ADTs; Stacks; Eclipse
Course objectives

- Learn basic data structures and algorithms
  - data structures – how data is organized
  - algorithms – unambiguous sequence of steps to compute something
  - algorithm analysis – determining how long an algorithm will take to solve a problem

- Become a better software developer
  - "Data Structures + Algorithms = Programs"
    -- Niklaus Wirth, author of Pascal language
Abstract Data Types

- **abstract data type (ADT):** A specification of a collection of data and the operations that can be performed on it.
  - Describes *what* a collection does, not *how* it does it
  - Described in Java with interfaces (e.g., `List`, `Map`, `Set`)
  - Separate from implementation

- ADTs can be implemented in multiple ways by classes:
  - `ArrayList` and `LinkedList` implement `List`
  - `HashSet` and `TreeSet` implement `Set`
  - `LinkedList`, `ArrayDeque`, etc. implement `Queue`
  - Java messed up on Stack—there's no Stack interface, just a class.
List ADT

- An ordered collection the form $A_0, A_1, ..., A_{N-1}$, where $N$ is the size of the list

- Operations described in Java's List interface (subset):

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(elt, index)</td>
<td>inserts the element at the specified position in the list</td>
</tr>
<tr>
<td>remove(index)</td>
<td>removes the element at the specified position</td>
</tr>
<tr>
<td>get(index)</td>
<td>returns the element at the specified position</td>
</tr>
<tr>
<td>set(index, elt)</td>
<td>replaces the element at the specified position with the specified element</td>
</tr>
<tr>
<td>contains(elt)</td>
<td>returns true if the list contains the element</td>
</tr>
<tr>
<td>size()</td>
<td>returns the number of elements in the list</td>
</tr>
</tbody>
</table>

- ArrayList and LinkedList are implementations
Stack ADT

- **stack**: a list with the restriction that insertions/deletions can only be performed at the top/end of the list
  - Last-In, First-Out ("LIFO")
  - The elements are stored in order of insertion, but we do not think of them as having indexes.
  - The client can only add/remove/examine the last element added (the "top").

- **basic stack operations**:
  - **push**: Add an element to the top.
  - **pop**: Remove the top element.
  - **peek**: Examine the top element.
Applications of Stacks

- **Programming languages:**
  - method calls are placed onto a stack (*call*=push, *return*=pop)

- **Matching up related pairs of things:**
  - find out whether a string is a palindrome
  - examine a file to see if its braces `{ }` and other operators match

- **Sophisticated algorithms:**
  - searching through a maze with "backtracking"
  - many programs use an "undo stack" of previous operations
Class Stack

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack&lt;E&gt;()</td>
<td>constructs a new stack with elements of type E</td>
</tr>
<tr>
<td>push(value)</td>
<td>places given value on top of stack</td>
</tr>
<tr>
<td>pop()</td>
<td>removes top value from stack and returns it; throws EmptyStackException if stack is empty</td>
</tr>
<tr>
<td>peek()</td>
<td>returns top value from stack without removing it; throws EmptyStackException if stack is empty</td>
</tr>
<tr>
<td>size()</td>
<td>returns number of elements in stack</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>returns true if stack has no elements</td>
</tr>
</tbody>
</table>

Stack<Integer> s = new Stack<Integer>();
s.push(42);
s.push(42);
s.push(-3);
s.push(17); // bottom [42, -3, 17] top

System.out.println(s.pop()); // 17
Stack limitations/idioms

- Remember: You can’t loop over a stack like you do a list.

```java
Stack<Integer> s = new Stack<Integer>();
...
for (int i = 0; i < s.size(); i++) {
    do something with s.get(i);
}
```

- Instead, you pull contents out of the stack to view them.
  - Idiom: Remove each element until the stack is empty.

```java
while (!s.isEmpty()) {
    do something with s.pop();
}
```
Exercise

- **Write a method** `symbolsBalanced` **that accepts a String as a parameter and returns whether or not the parentheses and the curly brackets in that String are balanced as they would have to be in a valid Java program.**

- **Use a Stack to solve this problem.**
Eclipse concepts

- **workspace**: a collection of projects
  - stored as a directory

- **project**: a Java program
  - must have your files in a project in order to be able to compile, debug and run them
  - by default stored in a directory in your workspace

- **perspective**: a view of your current project using a set of pre-laid-out windows and menus
  - Java perspective
  - debugging perspective