Lecture 2

- **Stacks**
  - The usual analogy is the “stack of plates”.
  - A way of buffering a stream of objects, in which the the Last In is the First Out (LIFO).
  - What might we use a stack for?

- **Functional requirements of a stack.**
  - Two basic methods, `push` and `pop`.
  - The first plate begins the pile, the next is placed on top of the first and so on.
  - A plate may be removed from the pile at any time, but only from the top.
  - Pushing a plate onto the pile increases the number on the pile.
  - Popping a plate from the pile decreases the number on the pile.
Stacks

- Say we want to build a calculator to evaluate \((3 + (2 \times 5)) / 2\)
- Work your way from the outside of the expression to the inside putting the operands (numbers and mathematical operators) onto the stack

Stacks – calculator example

<table>
<thead>
<tr>
<th>Evaluating ((3+(2\times5)) / 2)</th>
<th>The stack has 7 elements on it and 2 is at the top of the stack.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>To evaluate, put the item at the head into a variable and pop each element in turn performing the specified calculation on the variable.</td>
</tr>
<tr>
<td>*</td>
<td>Finished when the stack is empty.</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
Simple implementation

- Stack is represented by a **private** array of objects ...

```java
private Object[] stack;
private int top; // index for the top of the stack

// Constructor
public SimpleStack(int capacity) {
    stack = new Object[capacity];
    top = 0;
}

// To declare a stack ...
SimpleStack = new SimpleStack(20);
```

```java
public class SimpleStack {
    private Object[] stack;
    private int top;
    public SimpleStack(int capacity) {
        stack = new Object[capacity];
        top=0;
    }
    public boolean isEmpty() {
        return (top==0);
    }
    public boolean isFull() {
        return (top==stack.length);
    }
    public void push(Object item) throws Exception {
        if (isFull()) {
            throw New Exception("stack overflow");
        } else {
            stack[top++] = item;
        }
    }
    public Object pop() throws Exception {
        if (isEmpty()) {
            throw new Exception("stack underflow");
        } else {
            return stack[--top];
        }
    }
}
```
Interfaces

- Notice that clients of the SimpleStack class only have access to the stack through the public methods `isEmpty()`, `isFull()`, `push()` and `pop()`.
- The actual stack array and the pointer top are private and cannot be directly manipulated by the client.
- Arrays are not the only way to implement a stack – we can also use linked lists (later in the course).
- The Java construct of the interface:

```java
public interface Stack {
    public boolean isEmpty();
    public void push(Object items);
    public Object pop();
}
```

Implementation

- Now we shall see how we can use the stack interface to create a new stack class `StackArray`.
- `StackArray` will be able to deal with the situation when a stack needs to be re-sized because it is full. This will make use of a `System` Java method to copy one stack to another new one.

```java
System.arraycopy(stack, 0, newStack, 0, stack.length);
// This is equivalent to ...
for (int i=0; i<stack.length;i++) {
    newStack[i] = stack[i];
}
```
import java.util.
[215x668]NoSuchElementException
[314x668];
[139x659]public class
[197x659]StackArray
[247x659]implements Stack {
[139x650]private Object[] stack;
[139x641]private
[184x641]int
[202x641]top;
[139x632]public
[179x632]StackArray()
[224x632]{
[139x623]stack = new Object[1];
[139x614]top = 0;
[139x605]}
[139x596]public boolean isEmpty()
[247x596]{
[139x587]return (top==0);
[139x578]}
[139x569]public void push(Object item) {
[139x560]if (top==stack.length) {
[139x551]// expand the stack
[139x542]Object[] newStack = new Object[2*stack.length];
[139x533]System.arraycopy(stack,0,
[279x533]newStack,
[315x533],0,stack.length);
[139x524]stack = newStack;
[139x506]}
[139x506]stack[top++] = item;
[139x497]}
[139x488]public Object pop() {
[139x479]if (top==0) {
[139x470]throw new
[211x470]NoSuchElementException
[310x470]();
[139x461]} else {
[139x452]return stack[--top];
[139x443]}
[139x434]}
[139x425]}

Demonstration

- Now we can see StackArray in use.
- Can’t instantiate Stack directly, as it’s only an interface.
- Need to “object”-ify items to put them on the stack.

import DataStructures.*;

public class StackDemo {
    public static void main(String[] args) {
        Stack s = new StackArray();
        for (int i = 0; i < 8; i++) {
            s.push(new Integer(i));
        }
        while (!s.isEmpty()) {
            System.out.println(s.pop());
        }
    }
}
Note on Exceptions

- For simplicity, the present DataStructures package re-uses existing Java exceptions, rather than defining its own. Specifically, the exceptions used are:
  - ArithmeticException
  - IllegalArgumentException
  - IllegalStateException
  - NoSuchElementException
  - UnsupportedOperationException
  - These are all sub-class of the RuntimeException class.

- It is good programming practice to require the programmer to catch any exception which can arise for reasons beyond the programmer’s control, for example when interacting with an external device, such as a keyboard, mouse, server, whatever.

Next time ...

- OK, we’ve seen the LIFO stack, next time we shall look at the First In, First Out data structure – the queue.