

Data Structures G5029

Lecture 2

Kingsley Sage
Room 5C16, Pevensey III
khs20@sussex.ac.uk

© University of Sussex 2006

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 $(3 + (2 * 5)) / 2$
- 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

Evaluating $(3+(2*5)) / 2$

2
*
5
+
3
/
2

The stack has 7 elements on it and 2 is at the top of the stack.

To evaluate, put the item at the head into a variable and pop each element in turn performing the specified calculation on the variable.

Finished when the stack is empty.

Simple implementation

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

```
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);
```

```
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:

```
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.

```
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.NoSuchElementException;
public class StackArray implements Stack {
    private Object[] stack;
    private int top;
    public StackArray() {
        stack = new Object[1];
        top = 0;
    }
    public boolean isEmpty() {
        return (top==0);
    }
    public void push(Object item) {
        if (top==stack.length) {
            // expand the stack
            Object[] newStack = new Object[2*stack.length];
            System.arraycopy(stack,0,newStack,0,stack.length);
            stack = newStack;
        }
        stack[top++] = item;
    }
    public Object pop() {
        if (top==0) {
            throw new NoSuchElementException();
        } else {
            return stack[--top];
        }
    }
}

```

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++) {
            a.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.