

Stacks

(CISS-111 William Jojo)

Overview

Stacks are a wondrous structure. This is mostly because they are nothing more than a linked list. Well, if they are nothing more than a linked list, why do we call them stacks? The underlying structure is a linked list, but how we enforce its use is what makes it a stack.

For example, let us say we need to call a series of methods, and those methods will call more methods and those methods call even more methods, and so on. When a method calls a method which calls a method we need a way to return to the previous method and then to the previous method before that. Of course, we think of the `return` statement and all is well. The ability to call a method and later use a `return` statement only frees us of the knowledge and responsibility of stack management for just so long.

Imagine ourselves on the cafeteria line or our favorite buffet. As we enter the line at the end we form a *queue* (see Queues discussion). And the rules of a queue are we enter at the end and exit at the front. (There are no such rules for governing a generic linked list.) We come up a pile of plates, or rather a *stack* of plates. Each person in the line (queue) takes a plate from the top. Now imagine that in the kitchen, someone is washing the plates. This person starts with no plates, and the rapid cleaning dishwasher appliance has many cleaned plates. The person at this station then begins to remove plates from the dishwasher and places them one on top of another.

It is these actions that define the purpose and use of a stack which we will define now.

Stack Operations

Recall that a stack is nothing more than a linked list. The actions we allow on this linked list is what makes it special. Simply put, to place an object on the stack is to *push* onto the stack. To remove an object from the stack is to *pop* the stack. All push and pop actions may only occur at the top of the stack. We may inspect the value at the top of the stack without actually

popping it off. Any actions within the stack are prohibited.

Using our linked list code as a foundation, we will now define a stack in Java. *Example 1* has some basic stack code.

```
import java.util.*;

public class IntStack {

    private Node top;

    private class Node {

        private int data;
        private Node next;

        private Node(int item) {
            data = item;
            next = null;
        }

    }

    public IntStack () {
        top = null;
    }

    public void push(int item) {
        Node n = new Node(item);

        n.next = top;
        top = n;
    }

    public int pop() throws EmptyStackException {
        int v;

        if ( top == null )
            throw new EmptyStackException();

        v = top.data;
        top = top.next;

        return v;
    }
}
```

Example 1: The class IntStack which uses a Node and depicts the basic operations of push and pop.

Notice how we build the stack. Everything is pushed onto the top or head of the chain. Subsequently we only pop from the top. This is all that is needed to perform the necessary operations on a stack.

The `pop()` method must be sensitive to the possibility that the stack may be empty. This is called a *stack underflow*, but Java has an exception already set aside for this purpose. The `pop()` method will throw an `EmptyStackException` in the event we attempt to pop an empty stack. This would allow an application using this class to catch the exception and have an opportunity to make a decision about what to do next.

When pushing a value on to the stack, there is the possibility of running out of memory and not being able to add another value. Ordinarily we would call this a *stack overflow*, but since the true issue is that we have exhausted memory, we have chosen to not deal with this at all. Rather, it is left to the programmer to decide what to do if the application exceeds available memory.

We can introduce a couple of other tools for use with stacks. For example, we may want to look at the top of the stack, but not actually pop the value. This is useful in determining if some action should be taken based on the top of stack value. A tool like this is more advantageous than popping the value, inspecting it to determine the action taken and then pushing it *back* onto the stack if we determine we actually did not want it. A suitable piece of code is as follows:

```
public int peek() throws EmptyStackException {
    if ( top == null )
        throw new EmptyStackException();

    return top.data;
}
```

The `peek()` method allows us to inspect the value on top being mindful of the possibility that the stack is empty and throwing an exception when appropriate.

One more useful tool to stack operations is a debug method to dump the stack to the screen for viewing. Sometimes this, in addition to a debugger, is helpful in ascertaining why a program is performing a certain way.

Below is a method that dumps the contents of our integer stack to the screen:

```
public void dumpStack() {
    Node p = top;

    while ( p != null ) {
        System.out.println(p.data);
        p = p.next;
    }
}
```

The first item displayed is the top and all subsequent values are shown below it.

One more useful tool is the boolean method `isEmpty()` shown here:

```
public boolean isEmpty() {
    return top == null;
}
```

The `isEmpty()` method is useful when trying to set up a `while` loop to deal with all of the items on the stack. Remember that there is no method to tell how many items are on the stack.

Stacks are used in many everyday pieces of software including compilers, operating systems and applications. Anytime we need the ability to keep historical information a stack is likely a good candidate.