STACKS IN DATA AND PROCEDURES

The Stack ADT is for representing a *stack* of information, where you can add or remove elements at the top of the stack, but you cannot access data below the top. It includes four operations.

- `push(x)` to add `x` to the top of the stack.
- `pop()` to remove the top element from the stack, returning it.
- `peek()` to query what the top element of the stack is.
- `isEmpty()` to query whether the stack is currently empty.

The words *push* and *pop* are technical terms that computer scientists use to refer to adding and removing elements from a stack.

A stack is sometimes called a LIFO structure; *LIFO* abbreviates the phrase last in, first out, which reflects that with each removal (pop), the value that is *first* removed is the value within the stack that was pushed *last*.

Stacks are used in many common programs in ways visible to users, though the programs rarely explicitly use the word *stack*. Web browsers, for example, use a stack to track pages that have been visited: When you select a link, the page is loaded and pushed onto a stack; when you click the Back button, the top page is popped off the stack and the page now on the stack's top is displayed. (Typically, this is not a pure stack, because the browser usually allows the user to peek at pages beyond the stack's top.) Another stack comes in the multiple-undo feature available in many programs: Each user action is pushed onto a stack, and selecting Undo pops the top action off the stack and reverses it.

7.1.1. Implementing Stack

The Stack ADT is a restriction of the List ADT: The elements are kept in a particular order, but you can `add`, `remove`, and `get` only at one end of the list. For this reason, it's natural to consider using the existing List data structures, the ArrayList and the LinkedList.

The ArrayList is a good choice: If we maintain the top of the stack at the list's end, then all of ArrayList's relevant operations take $O(1)$ time. This is the inspiration behind the ArrayStack implementation below.

```
public class ArrayStack<E> {
    private ArrayList<E> data;

    public ArrayStack()       { data = new ArrayList<E>();          }
    public void push(E value) { data.add(value);                    }
```

```
    public E    pop()         { return data.remove(data.size() - 1); }
    public E    peek()        { return data.get(data.size() - 1);    }
    public boolean isEmpty()  { return data.size() == 0;             }
}
```

In fact, the `java.util` package includes a class named `Stack` that uses basically this approach. Note that in contrast to the other ADTs we've studied, the `java.util` package does not include a corresponding interface — instead, it includes a class with that name.

7.1.2. Case study: Program stack

Stacks show up in many situations in the design of computing systems. One of the most important is in remembering the variables of the methods currently in execution.

Each method has a collection of variables that it uses while it is working, including both the parameter values and other local variables declared within the method. This collection of variables is called that method's activation record. (This phenomenon is especially notable with recursive methods: Changing a local variable at lower levels of the recursion has no effect on the variable of the same name in upper levels. This occurs because each recursive call gets its own activation record.) Computers store activation records for methods currently in execution on a stack called the program stack.

Suppose the computer is currently in the process of executing a method *A* (and so *A*'s activation record is on the top of the stack), and it reaches a point where *A*'s code makes a call to a method *B*. The computer goes through a three-step process.

1. It stores its location within *A* inside *A*'s activation record.
2. It pushes an activation record for *B* onto the top of the stack, initializing the variables corresponding to *B*'s parameters based on the values designated by *A*.
3. It begins executing the code for *B*.

As the computer executes *B*'s code, it will have no cause to access variables of other methods: It will only access *B*'s activation record, which is on the stack's top. Eventually, the computer will complete *B* (likely once it reaches a `return` statement within *B*). Then, the computer completes two steps.

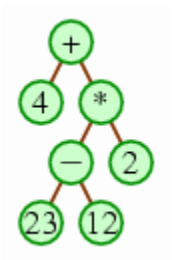
1. It pops *B*'s activation record from the top of the stack, so that *A*'s activation record is at the top once again.
2. It continues executing *A*, from the location it had previously stored within *A*'s activation record.

All of this occurs as a part of the implementation underlying the execution of Java programs, so it's not something that we can demonstrate in Java code. But understanding how the computer actually processes the execution of methods is a key component to understanding how Java programs work.

7.1.3. Case study: Postfix expression evaluation

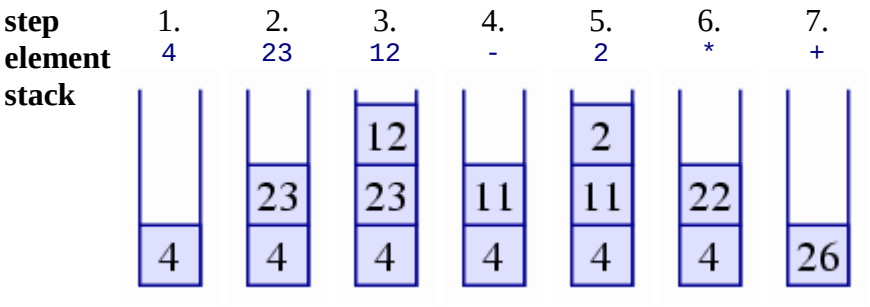
Expression evaluation is a case where one might want to use a stack in Java programs. We'll look at the simplest case of evaluating expressions given in postfix order. Recall that with postfix order, the operator is given after the arguments. For example, the expression normally written 2 + 3 would be written 2 3 + instead. The following is a more sophisticated example, with the corresponding expression tree.

4 23 12 − 2 * +



A simple approach to evaluating the expression is to use a stack, initially empty, and then go through the expression's elements in order. When we see a number, we'll push that number onto the stack. When we see an operator, we'll pop the top two numbers off the stack and push the result of applying the operator to those two numbers. If the expression is valid, then once we reach the expression's end, the stack will have just one number in it, which will be the overall result of the expression.

In our example, the process would proceed as follows.



Since we end up with 26 in the stack, we would conclude that 26 is the correct answer.

This algorithm is easy enough to implement in Java. Figure 7.1 contains a method whose parameter is an array of strings, representing the words within the postfix expression. It returns the expression's integer value.

Figure 7.1: Evaluating a postfix expression using a stack.

```
public static int evaluatePostfix(String[] expr) {
    Stack<Integer> nums = new Stack<Integer>();
    for(int i = 0; i < expr.length; i++) {
        String elt = expr[i];
        if(elt.equals("+")) {
            int b = nums.pop().intValue();
            int a = nums.pop().intValue();
            nums.push(new Integer(a + b));
        } else if(elt.equals("-")) {
            int b = nums.pop().intValue();
            int a = nums.pop().intValue();
            nums.push(new Integer(a - b));
        } else if(elt.equals("*")) {
            int b = nums.pop().intValue();
            int a = nums.pop().intValue();
            nums.push(new Integer(a * b));
        } else if(elt.equals("/")) {
            int b = nums.pop().intValue();
            int a = nums.pop().intValue();
            nums.push(new Integer(a / b));
        } else { // it must be a number
            int x = Integer.parseInt(elt);
            nums.push(new Integer(x));
        }
    }
    return nums.pop().intValue();
}
```

Stacks are also useful for evaluating prefix-order expressions and infix-order expressions, but the algorithms are more complex. Calculators that allow parentheses and recognize an order of operations use stacks to evaluate their solution. (In fact, many calculators can be configured so that users can enter expressions in postfix order, because many people who frequently use calculators find postfix order easier to manage. Of course, the calculators use a stack internally.)

Source : http://www.toves.org/books/data/ch07-sq/index.html