CS 240 Project 1

Alex Vondrak

Fall 2011

Overview

Many programming languages offer a *REPL*, which is a program that performs these steps:

Read: Prompt the user to input code in whatever language the REPL is for.

Eval: Evaluate the code.

Print: Outputs the results of evaluating the input.

Loop: Repeat the process, continually asking the user for code to evaluate.

This gives programmers a quick, convenient way of testing what some particular code will do without having to perform the standard edit-compile-run cycle that you normally do in Java.

While we won't be implementing an entire programming language, we can still build a REPL for relatively simple tasks. Here, we'll write a REPL-style user interface for an RPN calculator by writing a class, `RPN`, which implements the following **`interface`**.

```
interface REPL {
   public String[] read(Scanner in);
   public void eval(String expr);
   public void eval(String[] exprs);
   public void print();
   public void loop();
}
```

The `RPN` class will, of course, need to use a stack to perform its calculations. Thus, you must also write an `ArrayStack` class that implements the following **`interface`**.

```
interface Stack {
   public void push(int value);
   public int top() throws StackUnderflowException;
   public int pop() throws StackUnderflowException;
   public int size();
   public boolean isEmpty();
   public String toString();
}
```

Read

The language of the RPN calculator will very simply consist of whitespace-separated tokens. Your `read` method will need to prompt the user and read a single line of input, then return an array of the tokens that were on that line. For instance, the user may immediately hit the Enter key, giving us an empty array `{}`. If the input line is `"1"`, `read` will return `{"1"}`. If the input is `"a␣b"`, `read` will return `{"a", "b"}`. If the input is `"123␣456␣+"`, `read` will return `{"123", "456", "+"}`. And so on.

Hint: If `s` is a `String`, you can call `s.split("\\s+")` to get an array of substrings of `s` that are separated by one or more whitespace characters.

Eval

Given the input from the user, your `eval(String)` method must recognize the following inputs / commands:

- If the argument represents an integer, push the **`int`** value to the stack of operands.
- If the argument is any one of the standard arithmetic operators (`"+"`, `"*"`, `"-"`, `"/"`), pop the top two elements from the operand stack, perform the correct operation, and push the result back to the stack. If there are fewer than two elements, do not pop anything, but display an error message.
- If the argument is `"!"`, then pop every element that is currently on the stack. (You might pronounce this command “clear”.)
- If the argument is `"."`, then pop a single element off of the stack and print it to the output. (You might pronounce this command “pop”.)
- Otherwise, you have been given unrecognized input. Display an error message to the user and proceed.

The `eval(String[])` method should just call the `eval(String)` method on each element of the array in turn.

Print

After reading and evaluating some input, you must show the user the current state of the data stack. If the stack is empty, you shouldn't print anything. Otherwise, print out the elements of the stack on separate lines, with the topmost element printed at the bottom of the output (so that it's the closest thing the user will see when they're prompted again).

Loop

Continually perform the read, eval, and print actions. Do not stop when the user gives any particular input or when there's a stack error.

Examples

Here is a sample interaction with the RPN REPL. Lines beginning with `rpn>` are what the user inputs at the prompt (the prompt being the string `"rpn>␣"`).

```
$ java RPN
rpn> 1

--- Data stack:
1
rpn> 2

--- Data stack:
1
2
rpn> +

--- Data stack:
3
rpn> 3 * 4 -

--- Data stack:
5
rpn> 5 /

--- Data stack:
1
rpn> .
1
rpn> 1 2 3

--- Data stack:
1
2
3
rpn> .
3

--- Data stack:
1
2
rpn> .
2

--- Data stack:
1
```

```
rpn> .
1
rpn> 1 2 3

--- Data stack:
1
2
3
rpn> !
rpn> 1 2 3 !
rpn>
rpn>
rpn>
rpn>
rpn> Note that empty input isn't an error, but these unrecognized words are
Unknown operator: Note

Unknown operator: that

Unknown operator: empty

Unknown operator: input

Unknown operator: isn't

Unknown operator: an

Unknown operator: error,

Unknown operator: but

Unknown operator: these

Unknown operator: unrecognized

Unknown operator: words

Unknown operator: are

rpn> 5

--- Data stack:
5
rpn> 10
```

```
--- Data stack:
5
10
rpn> 15

--- Data stack:
5
10
15
rpn> +

--- Data stack:
5
25
rpn> +

--- Data stack:
30
rpn> +
Data stack underflow.

--- Data stack:
30
rpn> unknown_operator
Unknown operator: unknown_operator


--- Data stack:
30
rpn>
```