Glasgow Calculator Framework Requirements Document

Tim Storer
February 24, 2010

This document enumerates the requirements that must be satisfied by calculators produced using the *Glasgow Calculator Framework*.

1 Problem

Since the 1970s, when Hewlett-Packard Company introduced the first hand-held calculators based upon Reverse Polish Notation (RPN), such calculators have been popular amongst scientists and engineers. With the substantial portability of Java applications across laptops, desktops, servers, PDAs, and now many types of mobile telephones, our company is in a position to develop a family of RPN calculators for use across these different types of platforms.

There are many different types of calculators providing features appropriate to different groups of users e.g. scientific, business, and actuarial. A potential benefit to using a framework for calculator development is that the look and feel of the calculators in the family will be identical, all of them being built upon the same basic calculator.

Sometimes users need a graphical user interface to the calculator functionality, especially when performing ad hoc calculation activities; at other times, a line-oriented console user interface is appropriate, especially when performing repetitive or bulk calculation activities. Calculators built using the framework should support both types of user interface (i.e. graphical and console).

2 Background

2.1 Reverse Polish Notation

Reverse Polish Notation (RPN) is sometimes referred to as *postfix* notation. In RPN the operators follow their operands; for instance, to add three and four one would write "`3 4 +`" rather than "`3 + 4`".

If there are multiple operations, the operator is given immediately after its second operand; so the expression written "`3 - 4 + 5`" in conventional infix notation would be written "`3 4 - 5 +`" in RPN: first subtract `4` from `3`, then add `5` to that. An advantage of RPN is that it obviates the need for parentheses that are required by infix. While"`3 - 4 * 5`" can also be written "`3 - (4 *5)`" , that means something quite different from "`(3 - 4) *5`" , and only the parentheses disambiguate the two meanings. In postfix, the former would be written "`3 4 5 * -`" , which unambiguously means "`3 (4 5 *) -`".

Dijkstra's *shunting yard algorithm* can be used to convert from infix to RPN notation.

2.2 Calculator Interface

RPN calculators have a number of features that are different from their infix counterparts:

- RPN calculators perform calculations as input proceeds, with the result of any operation being pushed back onto the stack.

- RPN does not make use of parentheses to guard operator precedence
- RPN calculators have no *equals* key, since computation occurs as soon as an operation is read.
- RPN calculators require an explicit *enter* key to indicate that an operand has been specified by input. This is necessary because operands are not separated by operators as in infix notation. Without an explicit separator the expression `12 34+` could be interpreted as `1 234+` or `1234+` which would of course cause an error, since only one operand is specified for the operation.

2.3 Stack Implementation

Interpreters of Reverse Polish notation are often stack-based; that is, operands are pushed onto a stack, and when an operation is performed, its operands are popped from a stack and its result pushed back on. Stacks, and therefore RPN, have the advantage of being easy to implement and very fast.

Consider the following infix expression:

```
5 + ((1 + 2) * 4) - 3
```

and the equivalent RPN expression:

```
5 1 2 + 4 * + 3 -
```

The expression is evaluated left-to-right, with the inputs interpreted as shown in the following table (the Stack is the list of values the algorithm is "keeping track of" after the operation given in the middle column has taken place):

Input	Operation	Stack	Comment
5	Push operand	5	
1	Push operand	5, 1	
2	Push operand	5, 1, 2	
+	Add	5, 3	Pop two values (1, 2) and push result (3)
4	Push operand	5, 3, 4	
*	Multiply	5, 12	Pop two values (3, 4) and push result (12)
+	Add	17	Pop two values (5, 12) and push result (17)
3	Push operand	17, 3	
-	Subtract	14	Pop two values (17, 3) and push result (14)

When a computation is finished, its result remains as the top (and only) value in the stack; in this case, 14.

2.4 Use Patterns

There are three patterns of usage for a Glasgow Calculator Framework RPN calculators (Double type operand entry, unary operators and binary operators).

- An *enter* command causes a token representing a double value to be pushed onto the calculator stack:

```
Double result = Double.parseValue(token);
stack.push(value);
```

  A GCF *view* may support a number of constants (for example π). An internal representation of that constant as a double type operand to be pushed onto the stack.

- A unary operator command causes that operation to be applied to one value popped from the top of the stack. The result is then pushed back onto the stack:

```
Double x= stack.pop();
stack.push(unaryOp(x));
```

- A binary operator command causes that operation to be applied to two values popped from the top of the stack. The result is then pushed back onto the stack:

```
Double x = stack.pop();
Double y = stack.pop();
stack.push(binaryOp(x,y));
```

2.5 Accumulators

To support more complex operations, ten accumulators are provided by the basic calculator. All accumulator operations expect to find the numeric index for the accumulator (in the range `[0..9]` on the top of the stack. Four standard operations are supported on an accumulator in the Basic Calculator:

- Unary operators
  - Aclear the top item on the stack, used as an index to the appropriate accumulator, is replaced by the value of the accumulator after it has been cleared e.g. 0.0
  - Arecall the top item on the stack, used as an index to the appropriate accumulator, is replaced by the current value of the accumulator
- Binary operators
  - Astore the top two items on the stack are replaced by the result of applying the store operation on the accumulator indexed by the top of the stack, equivalent to the following pseudocode:

```
y = stack.pop();
x = stack.peek();
stack.push(accumulator[y].store(x));
```

  - Aplus the top two items on the stack are replaced by the result of applying the plus operator on the accumulator indexed by the top of the stack, equivalent to the following pseudocode:

```
y = stack.pop();
x = stack.pop()
stack.push(accumulator[y].plus(x));
```

Note that for both of these operators, the result of applying the `store()` and `plus()` operations on the accumulator is simply the value of the argument, `x`' in the cases above.

Calculators created by extending the Basic Calculator may define particular uses of these accumulators for complex functions, and may avail themselves of additional information maintained by the accumulator (e.g. each accumulator keeps track of the mean, variance, and standard deviation of the values which have been added to it).

3 Environment and System Models

Calculators built using the Glasgow Calculator Framework expect the full functionality of version 6 of the Java Standard Edition platform (J6SE). They should work on all platforms that claim J6SE compliance.

4 Functional Requirements

All GCF calculators must:

1. Support the functionality of the `BasicCalculator` (see Appendix).
2. Support both line-oriented and graphical user interfaces.
3. Enable the input of numbers, operators, and named constants
4. Provide informative diagnostic messages when errors are encountered
5. The Line-oriented user interface (Lui) must:
   (a) Provide a super-set of the Lui provided by the BasicCalculator
   (b) Enable free form input of numbers, operators, and constants
   (c) Support case-insensitive operator and constant names
6. The Graphical user interface (Gui) must:
   (a) Provide a super-set of the Gui provided by the BasicCalculator
   (b) Provide visual cues to distinguish between numbers, operators, and constants

5 Quality, Platform and Process Requirements

All GCF calculators must:

1. (PL) Operate on any J6SE-compliant platform.
2. (Q) Be thoroughly tested. This means the implementation of at least one test case per supported operation, and multiple test cases as appropriate per input equivalence set.
3. (PR) Use JUnit 4.x for unit testing.

A Summary of Basic Calculator Operations and Commands

Note S refers to the stack, S_y refers to the value on the top of the stack and S_x refers to the next value on the stack. All binary and unary operations remove their operands from the stack and push their result back on.

Operation	LUI Symbol	Category	Type	Effect
Enter	N/A	Stack operation		Pushes a value onto S.
Clear	N/A	Stack operation		clears S
Pop	N/A	Stack operation		removes and returns S_Y
Top	N/A	Stack operation		returns S_Y
A-Clear	Aclear	Accumulator	unary	Clears accumulator [S_y]
Recall	Arecall	Accumulator	unary	Pushes accumulator [S_Y] onto S.
Store	AStore	Accumulator	binary	Stores S_x in accumulator[S_y]
A-Plus	Aplus	Accumulator	binary	Adds S_x to accumulator[S_y]
Sign	Sign	Arithmetic	unary	-S_y
Invert	Invert	Arithmetic	unary	$1/S_y$
Square	Square	Arithmetic	unary	S_y^2
Square root	Sqrt	Arithmetic	unary	$S_y^{1/2}$
Plus	+	Arithmetic	binary	$S_y + S_x$
Minus	-	Arithmetic	binary	$S_y - S_x$
Times	*	Arithmetic	binary	$S_y * S_x$
Divide	/	Arithmetic	binary	S_y / S_x
Fix	Fix	Presentation	unary	Sets the number of decimal places for display to S_Y