A RPN Calculator
POSE2

Prof. DI P. Frey
HTBLA Leonding

Introduction

Overview

Handheld calculators are operated in two main ways, in either infix notation or in postfix notation. In infix notation, the mathematical operations are entered more or less as written, e.g. $2+4=?$ is entered as 2, +, 4 and `Exec/=`, which performs the calculation and displays the result. In postfix notation, the calculation is entered as 2, 4 and +, which performs the calculation and displays the result. This notation is quite easy to implement, and it also has certain benefints for the operator once used to this style.

Hewlett Packard produced (still produces?) several widely used RPN-calculators, such as the HP48 depicted above. Today, many mobile apps emulate RPN calculators, there are several HP48 implementations.

Links:
```
https://en.wikipedia.org/wiki/Reverse_Polish_notation
https://en.wikipedia.org/wiki/Infix_notation
https://de.wikipedia.org/wiki/HP_48
https://play.google.com/store/apps/details?id=org.ab.x48
```

Task overview

In this series of laboratory sessions, a desktop-UI application shall be developed that implements a basic RPN calculator. The components shall be developed using OO concepts and tested intensively. The following basic building blocks shall be developed:

- RPN math module: Responsible for performing RPN math, including the stack.
- RPN list processing module: Responsible for manipulating and calculating with lists of numbers.
- Basic RPN UI: A Desktop UI that exposes a basic RPN calculator with a display of 5-8 lines of stack.
- Advanced RPN UI: Extends the functionality to draw polynomic functions.

Specification

RPN Math Module

The RPN math module is the core computational component of the application. It is responsible for maintaining a stack of numbers (minimum depth: 8), and for implementing the following operations:

- Push: Add a number to the top of the stack.
- Pop: Remove and return the number from the top of the stack.
- Add (+): Pop the two topmost values, add them, and push the result.
- Subtract (-): Pop the two topmost values, subtract the second-from-top value from the top value, and push the result.
- Multiply (*): Pop the two topmost values, multiply them, and push the result.
- Divide (/): Pop the two topmost values, divide the second-from-top value by the top value, and push the result.
- Swap: Swap the top two elements on the stack.
- Clear: Empty the entire stack.

The module should be implemented as a reusable and testable class, completely decoupled from any UI logic. Unit tests should verify correct operation and handling of edge cases (e.g., division by zero, insufficient stack depth).

A domain specific exception hierarchy is expected to report operator or other errors, e.g. when adding but there is only one number on the stack.

Interface of the math module

The following interface shows the expected operations of the Rpn Calculator.

```
using System.Collections.Generic;
using RpnCalc.Exceptions;

namespace RpnCalc.Core
{
    /// <summary>
    /// Defines the interface for a basic Reverse Polish Notation
       calculator.
    /// </summary>
    public interface IRpnCalculator
    {
        /// <summary>
        /// Provides read-only access to the current stack.
        /// </summary>
        IReadOnlyCollection<double> Stack { get; }

        /// <summary>
        /// Pushes a number onto the stack.
        /// </summary>
        void Push(double value);

        /// <summary>
        /// Pops and returns the top value of the stack.
        /// Throws RpnStackUnderflowException if the stack is
           empty.
        /// </summary>
        /// <exception cref="RpnStackUnderflowException"/>
        double Pop();

        /// <summary>
        /// Performs addition with the top two elements on the
           stack.
        /// Throws RpnStackUnderflowException if fewer than two
           elements exist.
        /// </summary>
        /// <exception cref="RpnStackUnderflowException"/>
        void Add();

        /// <summary>
        /// Performs subtraction with the top two elements on the
           stack (second - top).
        /// Throws RpnStackUnderflowException if fewer than two
           elements exist.
        /// </summary>
        /// <exception cref="RpnStackUnderflowException"/>
        void Subtract();

        /// <summary>
        /// Performs multiplication with the top two elements on
           the stack.
        /// Throws RpnStackUnderflowException if fewer than two
           elements exist.
        /// </summary>
        /// <exception cref="RpnStackUnderflowException"/>
        void Multiply();

        /// <summary>
        /// Performs division with the top two elements on the
           stack (second / top).
        /// Throws RpnStackUnderflowException if fewer than two
           elements exist.
        /// Throws RpnDivisionByZeroException if division by zero
           occurs.
        /// </summary>
        /// <exception cref="RpnStackUnderflowException"/>
        /// <exception cref="RpnDivisionByZeroException"/>
        void Divide();

        /// <summary>
        /// Swaps the top two elements on the stack.
        /// Throws RpnStackUnderflowException if fewer than two
           elements exist.
        /// </summary>
        /// <exception cref="RpnStackUnderflowException"/>
        void Swap();

        /// <summary>
        /// Clears all elements from the stack.
        /// </summary>
        void Clear();

        /// <summary>
        /// Returns a snapshot of the current stack as an array
           (top of stack is last).
        /// </summary>
        double[] GetStackSnapshot();
    }
}
```

Avalonia RPN User Interface

The user interface shall be built using Avalonia in code-only style (i.e., no XAML). It must provide the following components:

- A vertical display showing the current stack with 5–8 lines (top of stack at the bottom).
- A numeric keypad (digits 0–9) and a decimal point button.
- Operation buttons for the supported math functions: +, −, *, /, Swap, Clear, and Enter.
- A text input field for entering numbers before pushing them onto the stack.
- Basic keyboard input support (Enter key to push, etc.).

The UI should update immediately to reflect the current state of the stack after each operation. The display must be implemented with Avalonia layout containers (e.g., StackPanel, Grid) and styled in code where necessary.

Drawing Polynomic Functions

As an extension task, implement support for plotting simple polynomial functions of the form: $f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 \ldots$

Requirements:

- Polynomial coefficients should be provided as a list on the stack, e.g., `[1 0 -2]` for $x^2 - 2$.
- The plotting module should interpret this list and draw the function over a predefined range (e.g., $x \in [-10, 10]$);
- A simple 2D plotting area must be created using Avalonia drawing primitives.

Display (Top of Stack at Bottom)
`Line 5:`
`Line 4:`
`Line 3:`
`Line 2:`
`Line 1:`

7	8	9	/
4	5	6	*
1	2	3	-
0	.	Enter	+
Swap	Clear		

Figure 1: Sketch of the RPN Calculator User Interface

- Axes and grid lines are optional, but clearly labeled ticks or axis lines are encouraged.

The module must be implemented cleanly, and integrated with the existing UI in a separate view or panel.

RPN List Processing Module

This module adds the capability to handle and process lists of numbers. List literals may be entered via a special syntax or UI support (e.g., using brackets: `[1 2 3]`).

Supported operations on lists:

- Length: Push the number of elements in the list.
- Sum: Push the sum of all elements.
- Average: Push the average value.
- Map (operation): Apply a math operation (e.g., +2) to each element.
- To Polynomial: Treat list as coefficients for polynomial plotting.

Lists must be stored and handled as stack elements, just like numbers. The system should clearly distinguish between scalars and lists in the display and internal logic.

Implementation

Github Repository

The implementation has to be pushed to a github repository. Each student will receive a link to a github repository (e.g. by github classroom). Details will be published outside of this text.

The milestones of the development shall be marked with simple branches which tag the current status.

Sequence of implementation

The calculator shall be implemented in a top-down way, starting at the User interface, and extending its functionality.

The following table defines the milestones and expected content:

Milestone	Branch	Content
UI Sketch	uisketch	First draft of UI. No functionality, only layout.
UI Prototype	uiproto	Clickable prototype of the UI. At least $+$, $-$, $Clear$ shall perform some visible, even faked, action, without using the Rpn-module.
RPN Base functionality	rpnsimple	Implement, test and attach the RPN basic operations. No plotting or lists.
RPN Function plot	rpnplot	Implement, test and extend the UI to draw a graph of a polynomial function.
RPN List functions	rpnlist	Implement, test and extend the UI to use lists of numbers.

Technical Requirements

The application shall be developed using the following technologies:

- .NET 8.0
- Avalonia (latest stable version)

- No XAML; the entire UI must be implemented programmatically
- Desktop only (must run on both Linux and Windows)
- No external libraries (no NuGet packages except official Avalonia and .NET SDK)
- Use only standard .NET types and collections (e.g., List, Stack, Dictionary, etc.)

The project structure must follow object-oriented principles, with a clear separation between logic and UI. Students are encouraged to write unit tests for all computational modules. The code must be readable, modular, and documented where necessary.

Solution structure

The solution shall have the following structure:

- `rpnCalc.sln`
  - `RpnCalc.Core`: Class library. Datatypes and Interfaces.
  - `RpnCalc.Logic`: Class library. RPN Calculator implementation.
  - `RpnCalc.App`: Avalonia App. Main application, UI.
  - `RpnCalc.Test`: XUnit Unit tests for the components.