Calculator.cs

**using** **System**;

**using** **System.Collections.Generic**;

**using** **System.Linq**;

**using** **System.Text**;

**using** **System.Threading.Tasks**;

**using** **System.Collections.Generic**;

**using** **System.Diagnostics**;

**class** **Calculator**

{

*/\*\* Scanner to get input from the user from the command line. \*/*

*//private Scanner scin = new Scanner( System.in );*

**static** **void** Main(string[] args)

{

*// Instantiate a "Main" object so we don't have to make everything static*

Calculator app = **new** Calculator();

bool playAgain = **true**;

Console.WriteLine("\nPostfix Calculator. Recognizes these operators: + - \* /");

**while** (playAgain)

{

playAgain = app.doCalculation();

}

Console.WriteLine("Bye.");

}

*/// <summary>*

*/// Get input string from user and perform calculation, returning true when*

*/// finished. If the user wishes to quit this method returns false.*

*/// </summary>*

*/// <returns>true if a calculation succeeded, false if the user wishes to quit</returns>*

**private** bool doCalculation()

{

Console.WriteLine("Please enter q to quit\n");

string input = "2 2 +";

Console.WriteLine("> "); *// prompt user*

input = Console.ReadLine();

*// looks like nextLine() blocks for input when used on an InputStream (System.in). Docs don't say that!*

*// See if the user wishes to quit*

**if** ( input.StartsWith( "q" ) || input.StartsWith( "Q" ) )

{

**return** **false**;

}

*// Go ahead with calculation*

string output = "4";

output = EvaluatePostFixInput( input );

Console.WriteLine("\n\t>>> " + input + " = " + output);

**return** **true**;

}

*/// <summary>*

*/// Evaluate an arithmetic expression written in postfix form.*

*/// </summary>*

*/// <param name="input">Postfix mathematical expression as a string</param>*

*/// <returns>The answer as a string</returns>*

**public** string EvaluatePostFixInput(string input)

{

StackADT stack = **new** LinkedStack();

**if** (input == **null** || input.Equals("") )

{

**return** ("Null or the empty string are not valid postfix expressions.");

}

**else**

{

**for** (int inputIndex = 0; inputIndex < input.Length; inputIndex++)

{

int num;

**if** (Int32.TryParse(input[inputIndex].ToString(), **out** num))

{

stack.Push(num); *// if it's a number push it on the stack*

}

**else**

{

*// Must be an operator or some other character or word.*

string s = input[inputIndex].ToString();

*// it may be an operator so pop two values off the stack and perform the indicated operation*

**if** (stack.IsEmpty())

{

**return** ( "Improper input format. Stack became empty when expecting second operand." );

}

double b = Convert.ToDouble(stack.Pop());

**if** (stack.IsEmpty())

{

**return** ( "Improper input format. Stack became empty when expecting first operand." );

}

double a = Convert.ToDouble(stack.Pop());

*// Wrap up all operations in a single method, easy to add other binary operators this way if desired*

stack.Push(doOperation(a, b, s));

}

}*// End while*

}

**return** (stack.Pop().ToString());

}

*/// <summary>*

*/// Perform arithmetic. Put it here so as not to clutter up the previous method, which is already pretty ugly.*

*/// </summary>*

*/// <param name="a">First operand</param>*

*/// <param name="b">Second operand</param>*

*/// <param name="s">Operator</param>*

*/// <returns>The answer</returns>*

**public** double doOperation(double a, double b, string s)

{

double c = 0.0;

**if** (s.Equals( "+" ) ) *// Can't use a switch-case with Strings, so we do if-else*

{

c = ( a + b );

}

**else** if (s.Equals( "-" ) )

{

c = ( a - b );

}

**else** if (s.Equals( "\*" ) )

{

c = ( a \* b );

}

**else** if (s.Equals( "/" ) )

{

c = ( a / b );

**if** (c == Double.NegativeInfinity || c == Double.PositiveInfinity)

{

Console.WriteLine("Cannot divide by 0");

**return** (0);

}

}

**else**

{

Console.WriteLine("Improper operator: " + s + ", is not one of +, -, \*, or /" );

**return** (0);

}

**return** c;

}

}

StackADT.cs

**using** **System**;

**using** **System.Collections.Generic**;

**using** **System.Linq**;

**using** **System.Text**;

**using** **System.Threading.Tasks**;

**using** **System.Collections.Generic**;

**using** **System.Diagnostics**;

*/\*\* Java Interface defining a Stack. \*/*

**public** **interface** IStackADT

{

*/// <summary>*

*/// Adds a node onto the stack*

*/// </summary>*

*/// <param name="newItem">The node to add</param>*

*/// <returns>The node which was added</returns>*

object Push(object newItem);

*/// <summary>*

*/// Returns the node at the top of the stack and removes it*

*/// </summary>*

*/// <returns>The node at the top of the stack</returns>*

object Pop();

*/// <summary>*

*/// Returns the node at the top of the stack without removing it*

*/// </summary>*

*/// <returns>The node at the top of the stack</returns>*

object Peek();

*/// <summary>*

*/// Returns if the stack is empty*

*/// </summary>*

*/// <returns>If the stack is empty</returns>*

bool IsEmpty();

*/// <summary>*

*/// Remove all node data*

*/// </summary>*

**void** Clear();

}

LinkedListStack.cs

**using** **System**;

**using** **System.Collections.Generic**;

**using** **System.Linq**;

**using** **System.Text**;

**using** **System.Threading.Tasks**;

**using** **System.Collections.Generic**;

**using** **System.Diagnostics**;

*/\*\**

*\* A singly linked stack implementation.*

*\*/*

**public** **class** **LinkedStack** : IStackADT

{

**private** Node top;

*/// <summary>*

*/// Stack constructor*

*/// </summary>*

**public** LinkedStack()

{

top = **null**; *// Empty stack condition*

}

*/// <summary>*

*/// Adds a node onto the stack*

*/// </summary>*

*/// <param name="newItem">The item to add to the stack</param>*

*/// <returns>The new node which was added</returns>*

**public** object Push(object newItem)

{

**if** ( newItem == **null** )

{

**return** **null**;

}

Node newNode = **new** Node(newItem, top);

top = newNode;

**return** newItem;

}

*/// <summary>*

*/// Removes the top node and returns it.*

*/// </summary>*

*/// <returns>The top node</returns>*

**public** object Pop()

{

**if** ( IsEmpty() )

{

**return** **null**;

}

object topItem = top.data;

top = top.next;

**return** topItem;

}

*/// <summary>*

*/// Returns the top node without removing it*

*/// </summary>*

*/// <returns>Returns the top node</returns>*

**public** object Peek()

{

**if** ( IsEmpty() )

{

**return** **null**;

}

**return** top.data;

}

*/// <summary>*

*/// Returns if this stack is empty*

*/// </summary>*

*/// <returns>If the stack is empty</returns>*

**public** bool IsEmpty()

{

**return** top == **null**;

}

*/// <summary>*

*/// Clear away all nodes*

*/// </summary>*

**public** **void** Clear()

{

top = **null**;

}

}

Node.cs

**using** **System**;

**using** **System.Collections.Generic**;

**using** **System.Linq**;

**using** **System.Text**;

**using** **System.Threading.Tasks**;

**using** **System.Collections.Generic**;

**using** **System.Diagnostics**;

*/\*\**

*\* A simple singly linked node class. This implementation comes from*

*\* before Java had Generics.*

*\*/*

**public** **class** **Node**

{

**public** object data; *// The payload*

**public** Node next; *// Reference to the next Node in the chain*

*/// <summary>*

*/// Null node constructor*

*/// </summary>*

**public** Node()

{

data = **null**;

next = **null**;

}

*/// <summary>*

*/// Node constructor given obj data and the next node*

*/// </summary>*

*/// <param name="data">data which does in this node</param>*

*/// <param name="next">the next linked node</param>*

**public** Node(object data, Node next)

{

**this**.data = data;

**this**.next = next;

}

}