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.");

}

*/\*\**

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

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

*\**

*\*@return true if a calculation succeeded, false if the user wishes to quit*

*\*/*

**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**;

}

*/\*\**

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

*\**

*\*@param input Postfix mathematical expression as a String*

*\*@return Answer as a String*

*\*@exception IllegalArgumentException Something went wrong*

*\*/*

**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());

}

*/\*\**

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

*\**

*\*@param a First operand*

*\*@param b Second operand*

*\*@param s operator*

*\*@return The answer*

*\*@exception IllegalArgumentException Something's fishy here*

*\*/*

**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** StackADT

{

*/\*\**

*\* Push an item onto the top of the stack. Pushing an object that*

*\* doesnâ€™t exist should result in an error and should not succeed.*

*\* Pushing an object that is not an item should result in an error.*

*\* This operation returns a reference (pointer or link, but not a copy)*

*\* to the item pushed so that an anonymous object can be pushed and then used.*

*\* @param newItem The object to push onto the top of the stack. Should not be null*

*\* @return A reference to the object that was pushed, or null if newItem == null*

*\*/*

Object Push(Object newItem);

*/\*\**

*\* Remove and return the top item on the stack. This operation should*

*\* result in an error if the stack is empty. Returns a reference to the*

*\* item removed.*

*\* @return A reference that was popped (and removed) from the stack or null if*

*\* the stack is empty*

*\*/*

Object Pop();

*/\*\**

*\* Return the top item but do not remove it. Generally should result in*

*\* an error if the stack is empty. An acceptable alternative is to return*

*\* something which the user can use to check to see if the stack was in fact empty.*

*\* @return A reference to the item currently on the top of the stack or null if*

*\* the stack is empty*

*\*/*

Object Peek();

*/\*\**

*\* Query the stack to see if it is empty or not. Cannot produce an error.*

*\* @return True if the stack is empty, false otherwise*

*\*/*

bool IsEmpty();

*/\*\**

*\* Reset the stack by emptying it. The exact technique used to clear*

*\* the stack is up to the implementor. The user should pay attention to what*

*\* this behavior is.*

*\*/*

**void** Clear();

}

LinkedList.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** : StackADT

{

**private** Node top;

**public** LinkedStack()

{

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

}

**public** Object Push(Object newItem)

{

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

{

**return** **null**;

}

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

top = newNode;

**return** newItem;

}

**public** Object Pop()

{

**if** ( IsEmpty() )

{

**return** **null**;

}

Object topItem = top.data;

top = top.next;

**return** topItem;

}

**public** Object Peek()

{

**if** ( IsEmpty() )

{

**return** **null**;

}

**return** top.data;

}

**public** bool IsEmpty()

{

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

}

**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*

**public** Node()

{

data = **null**;

next = **null**;

}

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

{

**this**.data = data;

**this**.next = next;

}

}