Programming in .NET

Microsoft Development Center Serbia programming course

Real examples of using C# features

The following problem divides a time frame into the collection of time intervals where the longest frame is as long as the *interval* parameter. This example uses `Tuple` and `IEnumerable<T>`, where 2 `DateTimeOffset` items represents left and right boundary of every time interval.

```
 public static IEnumerable<Tuple<DateTimeOffset,DateTimeOffset>> SplitTimeFrame(DateTimeOffset
startTime, DateTimeOffset endTime, TimeSpan interval)
{
    if(endTime <= startTime)
    {
        throw new ArgumentException("Start time must be smaller than the End time.");
    }

    DateTimeOffset newStart = startTime;
    while(newStart + interval < endTime)
    {
        yield return Tuple.Create(newStart, newStart + interval);
        newStart += interval;
    }
    yield return Tuple.Create(newStart, endTime);
}
```

Reading text files

The following example shows how a text file can be read line by line. That output then can be further pipelined into by using additional transformation or filtering. In our example we filtered out blank lines and lines with only whitespaces.

```
        static IEnumerable<string> ReadLines(string filename)
        {
            using (System.IO.TextReader reader = System.IO.File.OpenText(filename))
            {
                string line;
                while ((line = reader.ReadLine()) != null)
                {
                    yield return line;
                }
            }
        }

        static IEnumerable<string> IgnoreBlankOrWhitespace(IEnumerable<string> lines)
        {
            foreach (string line in lines)
            {
```

```
            if (!string.IsNullOrWhiteSpace(line))
                yield return line;
        }
    }

    static void UseReadLines()
    {
        foreach (string line in IgnoreBlankOrWhitespace(ReadLines("foo.txt")))
            Console.WriteLine(line);
    }
```

Arithmetic Expression Calculator

The following example shows how an arithmetic expression calculator can be created. Arithmetic expression can be constructed out of the simple single digits and operands “+”,”-“,”*”,”/”. Although the following example would work just fine, it’s not fully created in a spirit of C# language. Second example immediately below this one shows a better way to construct an arithmetic expression calculator additional feature of supporting floating point numbers. The second example uses many of the features and techniques presented in this course so far.

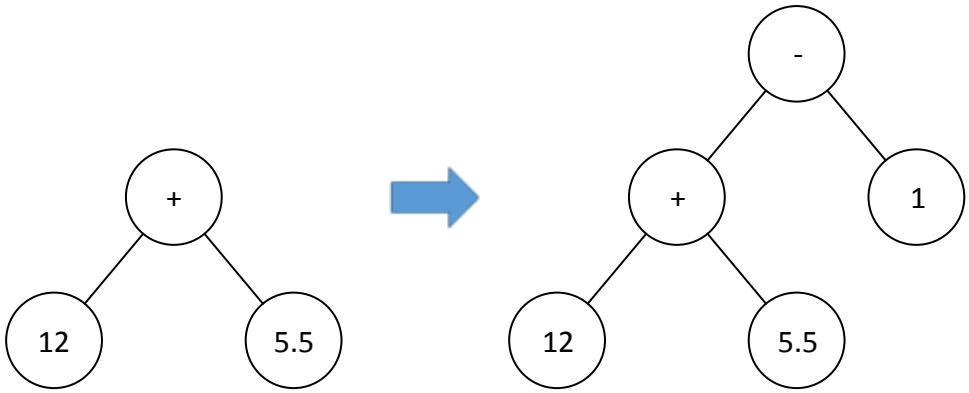
The main idea for both examples is to construct an expression tree out of the string, and calculate final result recursively going from the leaf nodes all way up to the root.

The following examples show how expression tree is calculated:

Expression: “12+5.5-1”

Tokens: “12”, “+”, “5.5”, “-“, “1”

Expression tree constructed:



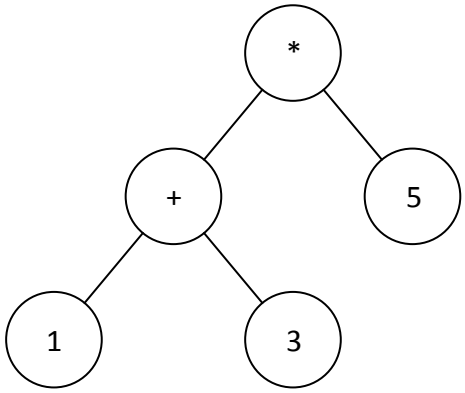
Left image shows tree after first operation with two operands is done (12+5.5), and right image shows how tree looks once entire expression is processed (12+5.5-1).

Expression tree is constructed in a way that every new operator (in our case “-” sign) adds new root node and pushes entire tree to the left. Second operand of the last operation (in our case value “1”) is added to the right of the root.

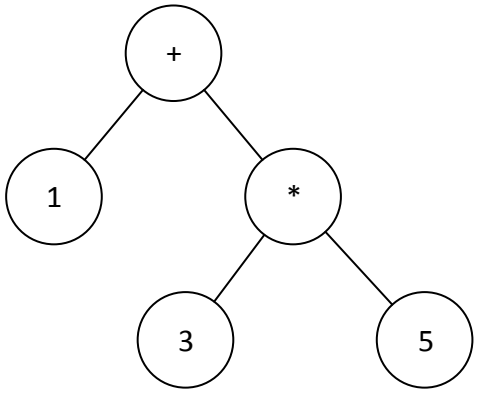
The tricky part is to process operands with higher priority correctly. That is achieved by pushing them down the tree to be executed before the parent is.

So, for the following expression: 1+3*5

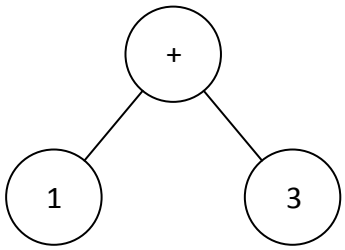
Expression tree should NOT look like



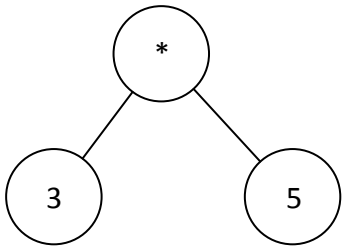
But rather



And this is done by substituting node 3 (right child) when the tree looks like this (after 3rd element is processed):



With



And that's how we get the final expression.

Simpler example presented immediately below uses limitation that every number can have only one digit to simplify the code.

```
// Make a expression calculator that can accept arbitrary expression composed
// out of single digit numbers and basic 4 aritmetic operations

class SimpleCalculator
{
    public class TreeNode<T>
    {
        public T Value;
        public TreeNode<T> Left;
        public TreeNode<T> Right;
    }

    public class OpNode : TreeNode<char> { }

    public static int CalculateResult(TreeNode<char> node)
    {
        switch (node.Value)
        {
            case '+':
                return CalculateResult(node.Left) + CalculateResult(node.Right);
            case '-':
                return CalculateResult(node.Left) - CalculateResult(node.Right);
            case '*':
                return CalculateResult(node.Left) * CalculateResult(node.Right);
            case '/':
                return CalculateResult(node.Left) / CalculateResult(node.Right);
            default:
                return int.Parse(node.Value.ToString());
        }
    }

    static void  Main()
    {

        while (1 == 1)
        {
            // read from a standard input
            string expression = Console.ReadLine();

            // exit if enter is pressed
            if (expression == "")
                return;

            // if only single number is passed
            if (expression.Length == 1)
            {
                Console.WriteLine(expression);
                continue;
            }

            // first operand x we can treat as 0 + x
            OpNode root = new OpNode {
```

```
                    Value = '+',
                    Right = new OpNode { Value = expression[0] },
                    Left = new OpNode { Value = '0' }
            };

            for (int i = 1; i < expression.Length - 1; i += 2)
            {
                // since both operations and operands are 1 char each
                // we can easily extract them
                char operation = expression[i];
                char nextOperand = expression[i + 1];

                // if operation is + or -
                // just create new top level node
                if (operation == '+' || operation == '-')
                {
                    OpNode newRoot = new OpNode
                    {
                        Value = operation,
                        Left = root,
                        Right = new OpNode { Value = nextOperand }
                    };
                    root = newRoot;
                }
                else
                {
                    // if operation is * or /
                    // last operand needs to be associated to the new node
                    // and new node will come in place of the old operand
                    OpNode newRoot = new OpNode
                    {
                        Value = operation,
                        Left = root.Right,
                        Right = new OpNode { Value = nextOperand }
                    };
                    root.Right = newRoot;
                }

            }

            Console.WriteLine(CalculateResult(root));
        }
    }
}
```

The second example demonstrates better way of writing calculator app that will be more robust and easier to extend and maintain. Compared to the previous one, it uses interfaces to represent expression trees, `ICalculable` interface and derived types `Operand` and `Operation`. In addition to that `Calculator` class makes clear pipeline that transforms the input data:

Input expression [string] → `Tokenize()` → String tokens [IEnumerable<string>]

String tokens → `BuildExpressionNodes()` → Operators and operands [IEnumerable<ICalculable>]

Operators and operands → `BuildExpressionTree()` → A tree that represents entire expression

The tree is constructed in this way since in the end we calculate entire expression (`Calculate` property for `ICalculable` interface) recursively bottom up, which means that result of a child expression is used by the parent to calculate parent expression.

Finally, once entire expression is constructed calling `Calculate` property on a root node will return result of entire expression. Thus finally calling simple line like

```
double result = BuildExpressionTree(BuildExpressionNodes(Tokenize(expression))).Calculate;
```

Will give us a result of a string arithmetic expression.

```
using System;
using System.Collections.Generic;

namespace MDCSEduExamples.ModernCalculatorExample
{

    /// <summary>
    /// Interface used to represent a node in an arithmetic tree
    /// </summary>
    /// <typeparam name="T">Type of a result.</typeparam>
    public interface ICalculable<out T>
    {
        T Calculate { get; }
    }

    /// <summary>
    /// Represents an integer operand
    /// </summary>
    public class Operand : ICalculable<double>
    {
        public double Value { get; set; }

        public double Calculate
        {
            get { return Value; }
        }
    }

    public enum OperationType
    {
        Addition,
        Subtraction,
        Multiplication,
        Division
    }

    /// <summary>
    /// Represents a simple arithmetic operation
    /// </summary>
    public class Operation : ICalculable<double>
    {
        public OperationType Type { get; set; }
        public ICalculable<double> Left { get; set; }
        public ICalculable<double> Right { get; set; }

        public double Calculate
        {
            get
            {
                System.Diagnostics.Debug.Assert(Left != null && Right != null);

                switch (Type)
                {
                    case OperationType.Addition:
                        return Left.Calculate + Right.Calculate;
                    case OperationType.Subtraction:
                        return Left.Calculate - Right.Calculate;
                    case OperationType.Division:
                        return Left.Calculate / Right.Calculate;
                    case OperationType.Multiplication:
                        return Left.Calculate * Right.Calculate;
```

```
                    default:
                        throw new InvalidOperationException(Type.ToString() + " is not recognized
 operation.");
                    // Note: delegates can be used instead
                    // for operation added to operatorTokens dictionary
                    // and this switch could be removed completely
                }
            }
        }
    }

    class Calculator
    {
        static readonly IDictionary<string, OperationType> operatorTokens = null;

        static Calculator()
        {
            // We are setting the mapping:
            // the character that represents an operation -> enum value for that operation
            operatorTokens = new Dictionary<string, OperationType>(4);
            operatorTokens.Add("*", OperationType.Multiplication);
            operatorTokens.Add("/", OperationType.Division);
            operatorTokens.Add("+", OperationType.Addition);
            operatorTokens.Add("-", OperationType.Subtraction);
        }

        /// <summary>
        /// Splits raw expression into tokens.
        /// A token can be an integer number or an arithmetic operations
        /// </summary>
        /// <param name="expression"></param>
        /// <returns></returns>
        /// <remarks>
        /// We will simply split every when we get an operator,
        /// since they are single character long and sitting between the numbers
        /// </remarks>
        public static IEnumerable<string> Tokenize(string expression)
        {
            int lastpos = 0, pos = 0;
            while (pos < expression.Length)
            {
                // if current character is an operator
                // then the token before it should be an operand
                if (operatorTokens.ContainsKey(expression[pos].ToString()))
                {
                    // yield operand
                    yield return expression.Substring(lastpos, pos - lastpos);

                    // yield operator
                    yield return expression[pos].ToString();
                    lastpos = pos + 1;
                }
                pos++;
            }
            // yield last operand
            yield return expression.Substring(lastpos, pos - lastpos);
        }

        /// <summary>
        /// Builds a collection of structured nodes out of list of tokens provided.
        /// </summary>
        /// <param name="tokens">Input set of tokens.</param>
        /// <returns>A collection of ICalculable<double> nodes of type Operand and Operation.</re
turns>
        /// <remarks> Returned collection is fully structured,
        /// original expression is not present anymore.
        /// </remarks>
        public static IEnumerable<ICalculable<double>> BuildExpressionNodes(IEnumerable<string> t
okens)
        {
            foreach (var token in tokens)
```

```
            {
                // if succeed to parse as double, it's a number
                double number;
                if (double.TryParse(token, out number))
                {
                    yield return new Operand() { Value = number };
                    continue;
                }

                // otherwise it must be an operator
                // without a dictionary we would have to have a big ugly switch here
                OperationType type;
                if (operatorTokens.TryGetValue(token, out type))
                {
                    yield return new Operation() { Type = type };
                    continue;
                }

                throw new ArgumentException(token + " is not a valid number nor operation.");
            }
        }

        /// <summary>
        /// Builds a full expression tree out of the given math expression
        /// built out of floating point numbers and basic 4 arithmetic operations.
        /// </summary>
        /// <param name="expression">Math Expression </param>
        /// <returns>A root node of the ICalculable<int>. </returns>
        public static ICalculable<double>
                      BuildExpressionTree(IEnumerable<ICalculable<double>> nodes)
        {
            ICalculable<double> root = null;
            ICalculable<double> lastOp = null;

            foreach (ICalculable<double> e in nodes)
            {
                // if e is an operand
                if (e is Operand)
                {
                    // first operand is added to an empty tree (special case)
                    // all other operands need to have last operation defined
                    if (root == null)
                    {
                        root = e;
                    }
                    else
                    {
                        System.Diagnostics.Debug.Assert(lastOp is Operation);
                        (lastOp as Operation).Right = e;
                    }
                }
                else
                {
                    // else e must be an operator
                    System.Diagnostics.Debug.Assert(e is Operation);

                    var operation = e as Operation;

                    // if operation is + or - or current expression is simple number
                    // so we will add it to be new root
                    if (operation.Type == OperationType.Addition || operation.Type == OperationTy
pe.Subtraction || root is Operand)
                    {
                        operation.Left = root;
                        root = operation;
                    }
                    else
                    {
                        // this operation must be * or /
                        // so last operand must must be associated to this operation
                        // and this operation will be placed instead of that last operand
```

```
                        System.Diagnostics.Debug.Assert(operation.Type == OperationType.Multiplic
ation || operation.Type == OperationType.Division);

                        operation.Left = lastOp;
                        (root as Operation).Right = operation;  // previous lastOp
                    }
                    // operation.Right will be assigned in the next iteration (lastOp) to an oper
and
                }
                lastOp = e;
            }

            return root;
        }

        // Let's test our arithmetic calculator
        static void Main()
        {

            Console.WriteLine("try typing 10+2*3-1");
            Console.WriteLine("press enter to exit");
            while (1 == 1)
            {
                string expression = Console.ReadLine();

                if (expression == "")
                    return;

                Console.WriteLine(
                      BuildExpressionTree(
                            BuildExpressionNodes(
                                  Tokenize(expression))).Calculate);
            }
        }
    }
}
```

Tokens

The following example uses an old manual way to implement enumerator (without using iterator block with `yield return`).

```
  // Declare the Tokens class:
    public class Tokens : IEnumerable<string>
    {
        private string[] elements;

        Tokens(string source, char[] delimiters)
        {
            // Parse the string into tokens:
            elements = source.Split(delimiters);
        }

        // IEnumerable<T> Interface Implementation:
        //   Declaration of the GetEnumerator() method
        //   required by IEnumerable to just reuse IEnumerable<T> implementation
        IEnumerator IEnumerable.GetEnumerator()
        {
            return GetEnumerator();
        }

        //   Declaration of the GetEnumerator() method
        //   required by IEnumerable<T>
        public IEnumerator<string> GetEnumerator()
        {
```

```
            return new TokenEnumerator(this);
        }

        // Inner class implements IEnumerator<T> interface:
        private class TokenEnumerator : IEnumerator<string>
        {
            private int position = -1;
            private Tokens t;

            public TokenEnumerator(Tokens t)
            {
                this.t = t;
            }

            // Declare the MoveNext method required by IEnumerator<T>:
            public bool MoveNext()
            {
                if (position < t.elements.Length - 1)
                {
                    position++;
                    return true;
                }
                else
                {
                    return false;
                }
            }

            // Declare the Reset method required by IEnumerator<T>:
            public void Reset()
            {
                position = -1;
            }

            // Declare the Current property required by IEnumerator<T>:
            public string Current
            {
                get
                {
                    return t.elements[position];
                }
            }

            public void Dispose()
            {

            }

            // Declare the Current property required by IEnumerator:
            object IEnumerator.Current
            {
                get { return Current; }
            }
        }

        // Test Tokens, TokenEnumerator
        static void Main3()
        {
            // Testing Tokens by breaking the string into tokens:
            Tokens f = new Tokens("This is a well-done program.",
               new char[] { ' ', '-' });
            foreach (string item in f)
            {
                Console.WriteLine(item);
            }
        }

    }
```

In this example, the following code uses `Tokens` class to break "This is a well-done program." into tokens (using ' ' and '-' as separators) and enumerating those tokens with the `foreach` statement:

```
Tokens f = new Tokens("This is a well-done program.",
   new char[] {' ','-'});
foreach (string item in f)
{
   Console.WriteLine(item);
}
```

Notice that, internally, Tokens uses an array, which implements `IEnumerator` and `IEnumerable` itself. The code sample could have leveraged the array's enumeration methods as its own, but that would have defeated the purpose of this example.

Internal class `TokenEnumerator` is used to take care of an Enumerator state, `private int position` variable in our example. There could be more than one enumerator for the same type at the same time. For example the following code would not work correctly if Token implemented `IEnumerator` directly, since internal loop will use the same position as the external one, so the external loop will never go through all iterations.

```
      foreach (string item in new Tokens(...))
      {
            foreach (string item in new Tokens(...))
            {
                  // Some code
            }
      }
```

Therefore starting from the code above, we can create Tokens type which implements `IEnumerable<T>` and provides strong type iteration over tokens.