RPN Calculator Tests

|  |  |  |
| --- | --- | --- |
|  | RED | GREEN/REFACTOR |
| 1.1 | [TestFixture]  public class test\_RPNCalculator  {      [Test]      public void Push\_on\_empty\_stack()      {          var sut = new RPNCalculator();      }  } | public class RPNCalculator  {  } |
| 1.2 | [Test]  public void Push\_on\_empty\_stack()  {      var sut = new RPNCalculator();        var result = sut.Push(1);  } | public class RPNCalculator  {      public Tuple<IEnumerable<int>, int>  Push(int number)      {          throw new **NotImplementedException**();      }  } |
| 1.3 | [Test]  public void Push\_on\_empty\_stack()  {      var sut = new RPNCalculator();        var result = sut.Push(1);        Assert.AreEqual(new[]{1}, result.Item1);      Assert.AreEqual(1, result.Item2);  } | public Tuple<IEnumerable<int>, int> Push(int number)  {      return new Tuple<IEnumerable<int>, int>(new[]{number}, number);  } |
| 2.1 | [Test]  public void Push\_number\_on\_non\_empty\_stack()  {      var sut = new RPNCalculator(initialStack);  } | public class RPNCalculator  {      private readonly Stack<int> \_stack;        public RPNCalculator() : this(new Stack<int>()) {}      public RPNCalculator(Stack<int> stack)      {          \_stack = stack;      } |
| 2.2 | [Test]  public void Push\_number\_on\_non\_empty\_stack()  {      var initialStack = new Stack<int>();      initialStack.Push(1);      var sut = new RPNCalculator(initialStack);      var result = sut.Push(2);        Assert.AreEqual(new[] { 1, 2 }, result.Item1);      Assert.AreEqual(2, result.Item2);  } | public Tuple<IEnumerable<int>, int>  Push(int number)  {      \_stack.Push(number);      return new Tuple<IEnumerable<int>, int>(\_stack.Reverse(), number);  } |
| 3.1 | [Test]  public void Add\_number\_to\_single\_value\_stack()  {      var initialStack = new Stack<int>();      initialStack.Push(2);      var sut = new RPNCalculator(initialStack);        Tuple<IEnumerable<int>, int> result = null;      sut.Result += \_ => result = \_;      sut.Calculate(new Tuple<string, int>("+", 3));  } | public class RPNCalculator  {  …        public void Calculate(Tuple<string, int> tuple)      {          throw new **NotImplementedException**();      } |
| 3.2 | [Test]  public void Add\_number\_to\_single\_value\_stack()  {      var initialStack = new Stack<int>();      initialStack.Push(2);      var sut = new RPNCalculator(initialStack);        Tuple<IEnumerable<int>, int> result = null;      sut.Result += \_ => result = \_;      sut.Calculate(new Tuple<string, int>("+", 3));        Assert.AreEqual(new int[]{}, result.Item1);      Assert.AreEqual(5, result.Item2);  } | public void Calculate(Tuple<string, int> calcRequest)  {      var leftOperand = \_stack.Pop();      var rightOperand = calcRequest.Item2;      var result = leftOperand + rightOperand;      Result(new Tuple<IEnumerable<int>, int>(\_stack.Reverse(), result));  } |
| 3 | [Test]  public void Add\_number\_to\_stack\_top()  {      var initialStack = new Stack<int>();      initialStack.Push(2);      initialStack.Push(3);      var sut = new RPNCalculator(initialStack);        Tuple<IEnumerable<int>, int> result = null;      sut.Result += \_ => result = \_;      sut.Calculate(new Tuple<string, int>("+", 4));        Assert.AreEqual(new[]{2}, result.Item1);      Assert.AreEqual(7, result.Item2);  } | it just works ☺ |
| 4.1 | [Test]  public void Drop\_number\_from\_empty\_stack()  {      var initialStack = new Stack<int>();      var sut = new RPNCalculator(initialStack);        Tuple<IEnumerable<int>, int> result = null;      sut.Result += \_ => result = \_;      sut.Drop();  } | public class RPNCalculator  {  …      public void Drop()      {          throw new **NotImplementedException**();      } |
| 4.2 | [Test]  public void Drop\_number\_from\_empty\_stack()  {      var initialStack = new Stack<int>();      var sut = new RPNCalculator(initialStack);        Tuple<IEnumerable<int>, int> result = null;      sut.Result += \_ => result = \_;      sut.Drop();        Assert.AreEqual(new int[] {}, result.Item1);      Assert.AreEqual(0, result.Item2);  } | public void Drop()  {      Result(new Tuple<IEnumerable<int>, int>(\_stack.Reverse(), 0));  } |
| 5.1 | [Test]  public void Drop\_number\_from\_non\_empty\_stack()  {      var initialStack = new Stack<int>();      initialStack.Push(1);      var sut = new RPNCalculator(initialStack);        Tuple<IEnumerable<int>, int> result = null;      sut.Result += \_ => result = \_;      sut.Drop();        Assert.AreEqual(new int[] { }, result.Item1);      Assert.AreEqual(1, result.Item2);  } | public void Drop()  {      var number = \_stack.Pop();      Result(new Tuple<IEnumerable<int>, int>(\_stack.Reverse(), number));  } |
| 5.2 | Test 4 now fails ☹ | public void Drop()  {      var number = \_stack.Any() ? \_stack.Pop() : 0;      Result(new Tuple<IEnumerable<int>, int>(\_stack.Reverse(), number));  } |
| 6.1 | [Test]  public void Calculate\_factorial\_for\_current\_number()  {      var sut = new RPNCalculator();        Tuple<IEnumerable<int>, int> result = null;      sut.Result += \_ => result = \_;      sut.Calculate(new Tuple<string, int>("!", 3));        Assert.AreEqual(new int[] {}, result.Item1);      Assert.AreEqual(6, result.Item2);  } | public class RPNCalculator  {      private readonly Stack<int> \_stack;      private readonly Dictionary<string, Func<Stack<int>, int>> \_operations;        public RPNCalculator() : this(new Stack<int>()) {}      public RPNCalculator(Stack<int> stack)      {          \_stack = stack;            \_operations = new Dictionary<string, Func<Stack<int>, int>>          {              {                  "+",                  operands =>                  {                      var right = operands.Pop();                      var left = operands.Pop();                      return left + right;                  }},              {                  "!",                  operands =>                  {                      var operand = operands.Pop();                      return Factorial(operand);                  }              }          };      }        …        public void Calculate(Tuple<string, int> calcRequest)      {          \_stack.Push(calcRequest.Item2);          var operation = \_operations[calcRequest.Item1];          var result = operation(\_stack);          Result(new Tuple<IEnumerable<int>, int>(\_stack.Reverse(), result));      }  …      int Factorial(int n)      {          if (n <= 1) return 1;          return Factorial(n - 1)\*n;      } |
| 7.1 | [TestCase(12, "-", 5, 7)]  [TestCase(12, "\*", 2, 24)]  [TestCase(12, "/", 3, 4)]  public void More\_basic\_operations(int leftOperand,                                      string op,                                      int number,                                      int calcResult)  {      var initialStack = new Stack<int>();      initialStack.Push(leftOperand);      var sut = new RPNCalculator(initialStack);        Tuple<IEnumerable<int>, int> result = null;      sut.Result += \_ => result = \_;      sut.Calculate(new Tuple<string, int>(op, number));        Assert.AreEqual(new int[] { }, result.Item1);      Assert.AreEqual(calcResult, result.Item2);  } | public RPNCalculator(Stack<int> stack)  {      \_stack = stack;        \_operations = new Dictionary<string, Func<Stack<int>, int>>          {              …              {                  "-",                  operands =>                      {                          var right = operands.Pop();                          var left = operands.Pop();                          return left - right;                      }              }, |
| 7.2 | Still 2 test cases red ☹ | \_operations = new Dictionary<string, Func<Stack<int>, int>>      {          …          {              "\*",              operands =>                  {                      var right = operands.Pop();                      var left = operands.Pop();                      return left \* right;                  }          }, |
| 7.3 | Still 1 test case red ☹ | \_operations = new Dictionary<string, Func<Stack<int>, int>>      {          …            {              "/",              operands =>                  {                      var right = operands.Pop();                      var left = operands.Pop();                      return left / right;                  }          }, |