String Calculator

Difficulty: Beginner

Estimated Duration: 1 hour

Introduction

This is the third stop on the Test-Driven Development learning track. By now, you should be reasonably familiar with the concepts of "red, green, refactor" and "baby steps". If not, check out the earlier katas on this track: FizzBuzz and Leap Year.

This kata gives further practice at designing software feature-by-feature - an important part of Agile development processes - using TDD. Each time you add a new feature, you'll need to adapt your algorithm by refactoring to allow it to support the new behaviour.

The rules of this kata get steadily more complex. It is recommended that you approach this by adding one rule at a time. Try not to look ahead: imagine that the rules had arrived in separate briefings spread over a six-month project!

Instructions

Step 1: Simple Calculator

Create a simple String calculator with a single method:

```
class StringCalculator {
    int Add(string numbers);
}
```

The method can take 1 or 2 comma-separated numbers, and will return their sum.

The method returns 0 when passed the empty string.

Example:

```
Add("") // 0
Add("4") // 4
Add("1,2") // 3
```

Start with the simplest test case of an empty string and move to 1 and two numbers.

Step 2: Arbitrary number size

Allow the Add method to handle an unknown amount of numbers.

Example:

```
Add("1,2,3,4,5,6,7,8,9") // 45
```

Step 3: Newline separator

Allow the Add method to recognise newlines as well as commas as separators. The two separator types can be used interchangeably.

NB: Focus on the happy path - since this is not production code, it's fine if the code crashes if it's given invalid input (e.g. `"1,\n2"`).

Example:

```
Add("1\n2,3") // 6
```

Step 4: Custom separators

Optionally support custom separators. To change separator, the beginning of the string will contain a separate line that looks like this: “`//<separator>\n<numbers>`”

Example:

```
Add("//;\n1;2") // 3
```

Note that all existing scenarios should still be supported.

Step 5: Disallow negatives

Calling Add with a negative number will throw an exception `negatives not allowed`, and the negative that was passed.

If there are multiple negatives, show all of them in the exception message.

Example:

```
Add("1,-2,-3") // error: negatives not allowed: -2 -3
```

Step 6: Ignore numbers bigger than 1000

Numbers bigger than 1000 should be ignored.

Example:

```
Add("1001, 2") // 2
```

Step 7: Arbitrary-length separators

Separators can be of any length if surrounded by square brackets.

Example:

```
Add("//[***]\n1***2***3") // 6
```

Step 8: Multiple single-length separators

Allow multiple single-character separators like this: “`//[delim1][delim2]\n`”

Example:

```
Add("//[*][%]\n1*2%3") // 6
```

Step 9: Multiple longer-length separators

Handle multiple separators with any character length.

Example:

```
Add("//[foo][bar]\n1foo2bar3") // 6
```

Credit: Roy Osherove

Tips

If you add a new test and then realise that you're going to need a big code change to make it pass, it's OK to back out the test (either by deleting it or marking it as ignored), and refactor your code until it's ready for the new test and functionality.

In fact, this is quite a powerful way of working: just be careful not to inadvertently add any new features while refactoring.

Useful Links

Solutions

- Partial solutions in many functional languages by Functional Katas Meetup

Algorithm Design | Classicist TDD | Data Structures

Psst!

Katalyst is in beta and we'd love your feedback.

Email us at katalyst@codurance.com and let us know what you think.