**1**

// Refactor 1: Handled empty input string at the beginning.

public int Add(String numbers) {

if (numbers.isEmpty()) { return 0; }

}

// Refactor 2: Made the code concise by directly returning the single number.

public int Add(String numbers) {

if (numbers.isEmpty()) { return 0; }

int number = Integer.parseInt(numbers);

return number;

}

// Refactor 3: Optimized for two numbers, no need for splitting and iteration.

public int Add(String numbers) {

if (numbers.isEmpty()) { return 0; }

String[] nums = numbers.split(",");

int num1 = Integer.parseInt(nums[0]);

int num2 = Integer.parseInt(nums[1]);

return num1 + num2;

}

// Refactor 4: Code can handle an unknown number of arguments efficiently.

public int Add(String numbers) {

if (numbers.isEmpty()) { return 0; }

String[] nums = numbers.split(",");

int sumOfNumbers = 0;

for (String num : nums) { sumOfNumbers += Integer.parseInt(num); }

return sumOfNumbers;

}

// Refactor 5: Code can handle null inputs and newline charaters “\n”

public class AddMyAlphas {

public int Add (String numbers) {

if (numbers == null || numbers.isEmpty()) { return 0; }

String[] nums = numbers.split("[,\n]");

int sumOfNumbers = 0;

for (String num : nums) { sumOfNumbers += Integer.parseInt(num); }

return sumOfNumbers;

}

}

// Refactor 6: Adding a method for Integer Parsing and creating a bucket of negative integers

// and returning an error with elements in the bucket (if exists)

public class AddMyAlphas {

public int Add (String numbers) {

if (numbers== null || numbers.isEmpty()) { return 0; }

String[] nums = numbers.split("[,\n]");

int sumOfNumbers = 0;

List<Integer> negativeNumbers = new ArrayList<Integer>();

for (String num : nums) {

int currentNumber = Integer.parseInt(num);

if (currentNumber < 0) { negativeNumbers.add(currentNumber); }

sumOfNumbers += currentNumber;

}

if (!negativeNumbers.isEmpty()) {

throw new IllegalArgumentException("Negatives not allowed: " + negativeNumbers);

}

return sumOfNumbers;

}

}

// Refactor 7: Adding a check for avoiding numbers more than 1000

public class AddMyAlphas {

public int Add (String numbers) {

if (numbers== null || numbers.isEmpty()) { return 0; }

String[] nums = numbers.split("[,\n]");

int sumOfNumbers = 0;

List<Integer> negativeNumbers = new ArrayList<Integer>();

for (String num : nums) {

int currentNumber = Integer.parseInt(num);

if (currentNumber < 0) { negativeNumbers.add(currentNumber); }

else if (currentNumber > 1000) continue;

sumOfNumbers += currentNumber;

}

if (!negativeNumbers.isEmpty())

{ throw new IllegalArgumentException("Negatives not allowed: " + negativeNumbers); }

return sumOfNumbers;

}

}

// Refactor 8: Checking delimiters after “//” and before “\n”, and splitting the strings

// into numbers and parsing Integer values from it. Default delimiter is “,”.

public class AddMyAlphas {

public int Add(String numbers) {

if (numbers== null || numbers.isEmpty()) { return 0; }

String delimiter = ",";

String numbersPart = numbers;

if (numbers.startsWith("//")) { // Check for custom delimiter

int delimiterIndex = numbers.indexOf("\n"); // Find newline after delimiter

delimiter = numbers.substring(2, delimiterIndex); // Extract custom delimiter

numbersPart = numbers.substring(delimiterIndex + 1); // Extract numbers part

}

String[] nums = numbersPart.split("[,\n" + delimiter + "]");

// Split numbers using delimiter(s)

int sumOfNumbers = 0;

List<Integer> negatives = new ArrayList<>();

for (String num : nums) {

int currentNumber = Integer.parseInt(num);

if (currentNumber < 0) { negatives.add(currentNumber); }

else if (currentNumber <= 1000) { sumOfNumbers += currentNumber; }

}

if (!negatives.isEmpty())

{ throw new IllegalArgumentException("Negatives not allowed: " + negatives); }

return sumOfNumbers;

}

}

**QUESTION 1 CAN BE FOUND ON JUNIT TEST FOLDER**

**MAIN FINAL FILE**  
package assignment2;

import java.util.ArrayList;

import java.util.List;

public class AddMyAlphas {

public int Add(String numbers) {

if (numbers== null || numbers.isEmpty()) { return 0; }

String delimiter = ",";

String numbersPart = numbers;

if (numbers.startsWith("//")) { // Check for custom delimiter

int delimiterIndex = numbers.indexOf("\n"); // Find newline after delimiter

delimiter = numbers.substring(2, delimiterIndex); // Extract custom delimiter

numbersPart = numbers.substring(delimiterIndex + 1); // Extract numbers part

}

String[] nums = numbersPart.split("[,\n" + delimiter + "]"); // Split numbers using delimiter(s)

int sumOfNumbers = 0;

List<Integer> negatives = new ArrayList<>();

for (String num : nums) {

int currentNumber = Integer.parseInt(num);

if (currentNumber < 0) { negatives.add(currentNumber); }

else if (currentNumber <= 1000) { sumOfNumbers += currentNumber; }

}

if (!negatives.isEmpty()) { throw new IllegalArgumentException("Negatives not allowed: " + negatives); }

return sumOfNumbers;

}

}

**MAIN FINAL TEST FILE**

package assignment2;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

import static org.junit.jupiter.api.Assertions.assertThrows;

public class AddMyAlphasTest {

@Test

public void AddMyAlphaTest01\_Q1\_EmptyString() {

AddMyAlphas adder = new AddMyAlphas();

int result = adder.Add("");

assertEquals(0, result);

}

@Test

public void AddMyAlphaTest02\_Q1\_OneDigits() {

AddMyAlphas adder = new AddMyAlphas();

int result = adder.Add("1");

assertEquals(1, result);

}

@Test

public void AddMyAlphaTest03\_Q1\_TwoDigits() {

AddMyAlphas adder = new AddMyAlphas();

int result = adder.Add("1,4");

assertEquals(5, result);

}

@Test

public void AddMyAlphaTest04\_Q2\_MultipleDigits() {

AddMyAlphas adder = new AddMyAlphas();

int result = adder.Add("1,3,5,7,9,11,13,15,17,19");

assertEquals(100, result);

}

@Test

public void AddMyAlphaTest05\_Q3\_DoubleNewLineTest() {

AddMyAlphas adder = new AddMyAlphas();

int result = adder.Add("1\n2,3");

assertEquals(6, result);

}

@Test

public void AddMyAlphaTest06\_Q4\_MultipleNegativeNumbers() {

AddMyAlphas adder = new AddMyAlphas();

Exception exception = assertThrows(IllegalArgumentException.class, () -> {

adder.Add("2,-4,3,-5");

});

assertEquals("Negatives not allowed: [-4, -5]", exception.getMessage());

}

@Test

public void AddMyAlphaTest07\_Q5\_NumbersBiggerNumbers() {

AddMyAlphas adder = new AddMyAlphas();

int result = adder.Add("1,2\n2000\n3000");

assertEquals(3, result);

}

@Test

public void AddMyAlphaTest08\_Q6\_CustomDelimiterSemicolon() {

AddMyAlphas adder = new AddMyAlphas();

int result = adder.Add("//;\n1;2");

assertEquals(3, result);

}

@Test

public void AddMyAlphaTest09\_Q6\_CustomDelimiterPipe() {

AddMyAlphas adder = new AddMyAlphas();

int result = adder.Add("//\\|\n1|2|3");

assertEquals(6, result);

}

@Test

public void AddMyAlphaTest10\_Q6\_CustomDelimiterDollar() {

AddMyAlphas adder = new AddMyAlphas();

int result = adder.Add("//$\n1$2$3$4$5");

assertEquals(15, result);

}

}

**2C**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **PATH** | **1-2-4-5-7-8** | **1-3-1-2-5-7-8** | **1-3-1-3-8** | **1-2-4-6-7-8** |
| **INITIAL CONDITIONS** | User enters a parking space ID and is prompted to input a payment information | User enters a parking space ID and is prompted to input a payment information | User enters a parking space ID and is prompted to input a payment information | User enters a parking space ID and is prompted to input a payment information |
| **TEST STEPS** | 1. User enters a parking space ID.  2. User selects a right method of payment.  3. User decides to save their payment information. | 1. User enters a parking space ID.  2. User selects a wrong method of payment.  3. User decides to proceed and enters a correct payment method.  4. User decides to save their payment information. | 1. User enters a parking space ID.  2. User selects a wrong method of payment.  3. User decides to proceed and enters a wrong payment information again.  4. User decides not to proceed further. | 1. User enters a parking space ID.  2. User selects a right method of payment.  3. User decides to NOT save their payment information. |
| **EXPECTED RESULT** | Server sends a payment receipt to the user. | Server sends a payment receipt to the user. | No data is sent to the user and the server. | Server sends a payment receipt to the user. |

**3C**

**STATEMENT COVERAGE**

**1. Test Case 1: Empty List**

- Input: an empty list

- Expected Output: "No scores found in score list"

**2. Test Case 2: List with a single score**

- Input: [90]

- Expected Output:

- "1 total score found in score list"

- "The maximum score is 90.0"

- "The minimum score is 90.0"

**3. Test Case 3: List with multiple scores**

- Input: [75, 85, 95, 80, 70]

- Expected Output:

- "5 total scores found in score list"

- "The maximum score is 95.0"

- "The minimum score is 70.0"

**4. Test Case 4: List with negative scores**

- Input: [-10, -5, -15, -2]

- Expected Output:

- "4 total scores found in score list"

- "The maximum score is -2.0"

- "The minimum score is -15.0"

**BRANCH COVERAGE**

**1. Test Case 1: Empty List**

- Input: an empty list

- Expected Output: "No scores found in score list"

**2. Test Case 2: List with a single score**

- Input: [90]

- Expected Output:

- "1 total score found in score list"

- "The maximum score is 90.0"

- "The minimum score is 90.0"

**3. Test Case 3: List with multiple scores** (maxScore > minScore branch)

- Input: [75, 85, 95, 80, 70]

- Expected Output:

- "5 total scores found in score list"

- "The maximum score is 95.0"

- "The minimum score is 70.0"

**4. Test Case 4: List with multiple scores** (minScore >= maxScore branch)

- Input: [80, 85, 90, 85, 80]

- Expected Output:

- "5 total scores found in score list"

- "The maximum score is 90.0"

- "The minimum score is 80.0"