Assignment 5

SWE 437 Section 002

Professor: Jeff Offutt

Partners: Fatemeh Nouri & Danielle Schwartz

Date: 10/10/2019

Code Location: https://github.com/dschwar4/SWE437-002

see: /src/assignment5, /singleLines.txt, /multipleLines.txt

I. Collaboration Summary

Danielle and Fatemeh began this assignment by first creating user stories. We collaborated and referenced our customer (assignment 5) and captured its requirements using separate user stories. Next, Danielle suggested that we proceed with our simplest and most basic user story. Next, Fatemeh referenced the Acceptance Tests in Agile Methods cycle from the lecture notes and suggested creating acceptance test for the story. Then, they both tried to get a clear idea of what the expected behavior of the feature captured using the story should be and wrote the needed test case. Then they ran the acceptance test and noticed that the file did not even compile which was not surprising considering the fact that they didn't have a class for the object they were testing: StringPrint. Next they decided to use the process of TDD to implement the code needed for the acceptance test. Next, Fatemeh restated the general cycle of TDD from the notes and lectures which is: Test-code-refactor, and Danielle suggested we should start with the test case that checks for the most basic part of the program: being able to return a character. So they wrote the test case together, noticed the file still does not compile since they don't have any class or method just yet. So they created a class named StringPrint and just wrote 1 method in it that returned a hard-coded value. Then they ran the test again together and noticed their test passes. They both started to think if they could do any refactoring at this point and came to the conclusion that there was nothing to refactor at the moment because class only has 1 method that returns 1 character. So they moved on to write another test case. They wanted to know what would happen if their program was given a string instead of a character, so they wrote a test case for it. Their file did not compile because the compiler complained about how the method was given a string when it was expecting a character. So they went back to their StringPrint class and changed the parameter type of the method to string instead of char. Of course the first test case that tested for a character then complained and they decided to refactor the first test case by sending it as a string instead of a character. Then they proceeded with the rest of their user stories in a similar fashion: they wrote a test case to test the overall expected behavior captured on the user story, then used TDD to write smaller test cases, implement, and refactor, ensured the acceptance TDD test passes, and then moved on to the next user story on the list.

Partners: Fatemeh Nouri & Danielle Schwartz
SWE 437 – Section 002
10/10/2019

II. User Stories

User Story 1

1. The user provides a text file.
2. The user specifies how they would like to retrieve a line (with or without replacement).
3. The user requests one line.
4. A random line is returned.
5. Steps 3-4 repeat until the user specifies to exit.

Partners: Fatemeh Nouri & Danielle Schwartz
SWE 437 – Section 002
10/10/2019

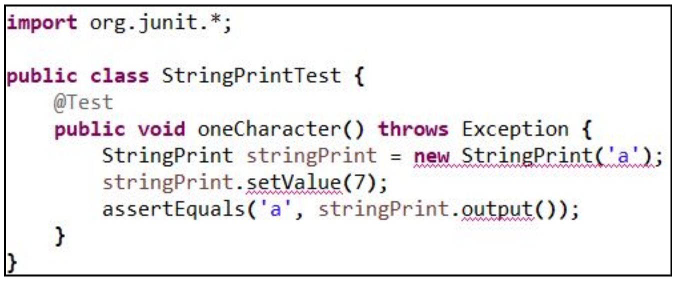
III. User Story 1: A-TDD Cycles

1. The user provides a text file.
2. The user specifies how they would like to retrieve a line (with or without replacement).
3. The user requests one line.
4. A random line is returned.
5. Steps 3-4 repeat until the user specifies to exit.

Round 1

Test:

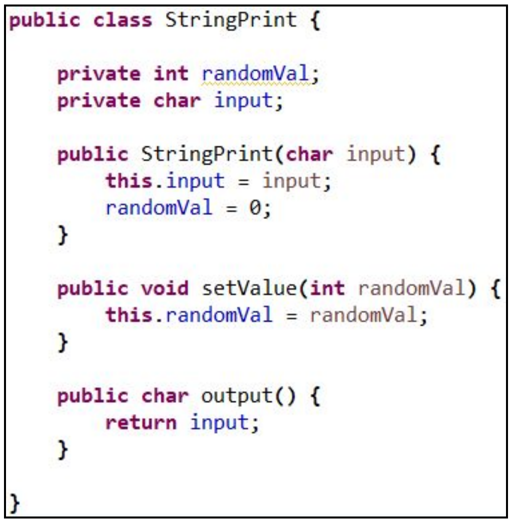
1. The user provides a single character.
2. The user asks for a random value and the character is returned.



```java
import org.junit.*;

public class StringPrintTest {
    @Test
    public void oneCharacter() throws Exception {
        StringPrint stringPrint = new StringPrint('a');
        stringPrint.setValue(7);
        assertEquals('a', stringPrint.output());
    }
}
```

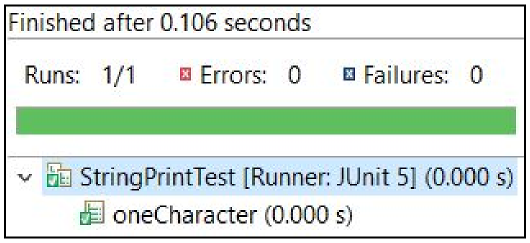
Code:

Created a `StringPrint` class with a constructor that accepts a character, which is stored in a class-level variable, `input`. Created a public `setValue()` method within `StringPrint` that sets the value of a class-level variable, `randomVal`, to the provided number. Created a public `output()` method that prints the character stored in `input`.



```java
public class StringPrint {

    private int randomVal;
    private char input;

    public StringPrint(char input) {
        this.input = input;
        randomVal = 0;
    }

    public void setValue(int randomVal) {
        this.randomVal = randomVal;
    }

    public char output() {
        return input;
    }
}
```

Test Results:



```
Finished after 0.106 seconds

Runs:  1/1     Errors:  0     Failures:  0

StringPrintTest [Runner: JUnit 5] (0.000 s)
    oneCharacter (0.000 s)
```

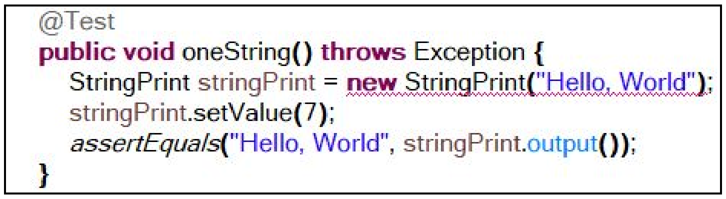
Refactor: No refactoring required yet.

Partners: Fatemeh Nouri & Danielle Schwartz
SWE 437 – Section 002
10/10/2019

Round 2

Test:

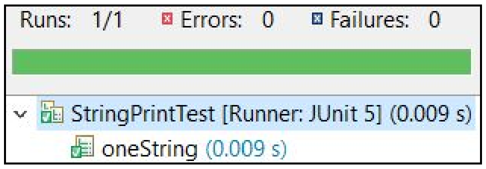
1. The user provides a string of characters.
2. The user asks for a random value and the string is returned.



```
@Test
public void oneString() throws Exception {
    StringPrint stringPrint = new StringPrint("Hello, World");
    stringPrint.setValue(7);
    assertEquals("Hello, World", stringPrint.output());
}
```

Code:
Changed `input` variable from `char` type to `String`; same for the return type of `output()`. These changes result in a syntax error in our first test since we are no longer passing a character. Removed the original test; it is no longer needed.

Test Results:



```
Runs:  1/1    Errors:  0    Failures:  0

StringPrintTest [Runner: JUnit 5] (0.009 s)
    oneString (0.009 s)
```

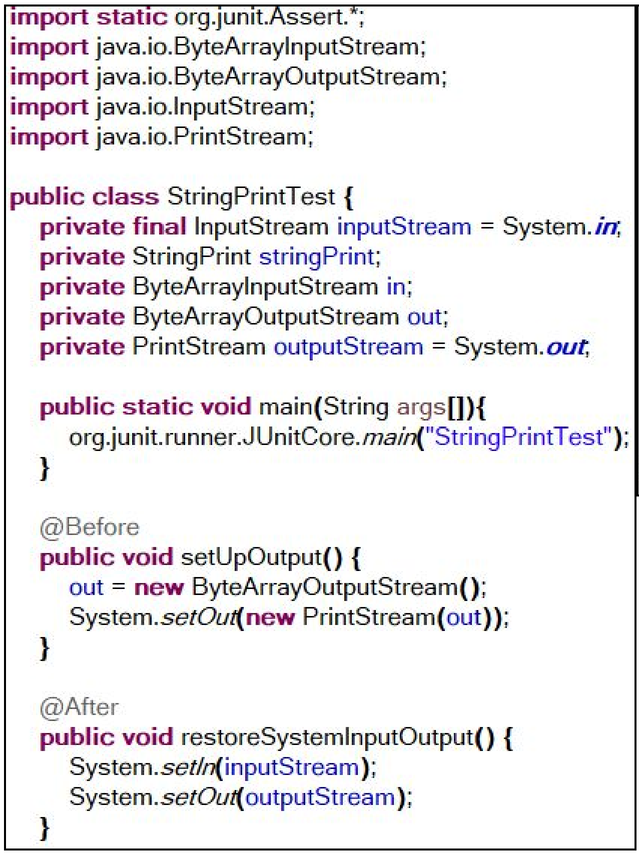
Refactor:
We had to remove our initial `oneCharacter()` test in order to expand our functionality to Strings. No refactoring was done to `StringPrint` at this time (unless we consider our "design replacements" above as refactoring).

Partners: Fatemeh Nouri & Danielle Schwartz
SWE 437 – Section 002
10/10/2019

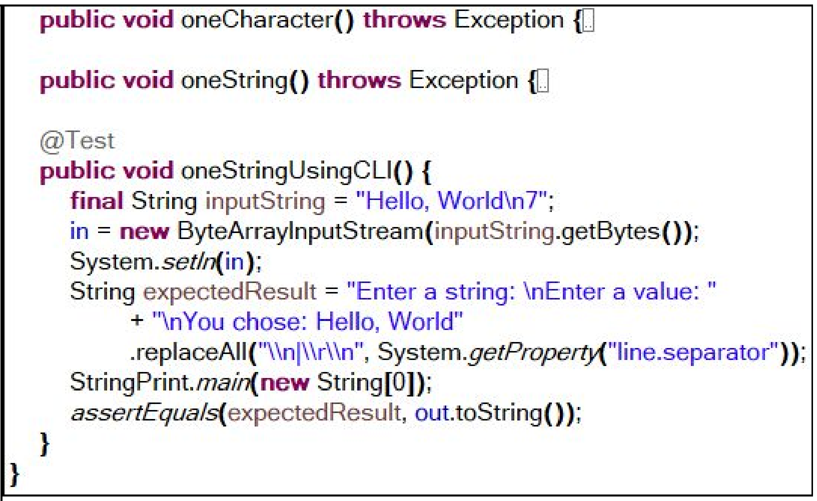
Round 3

Test:

1. The user provides a string via command prompt.
2. The user provides a random value via command prompt.
3. The string is returned via standard output.



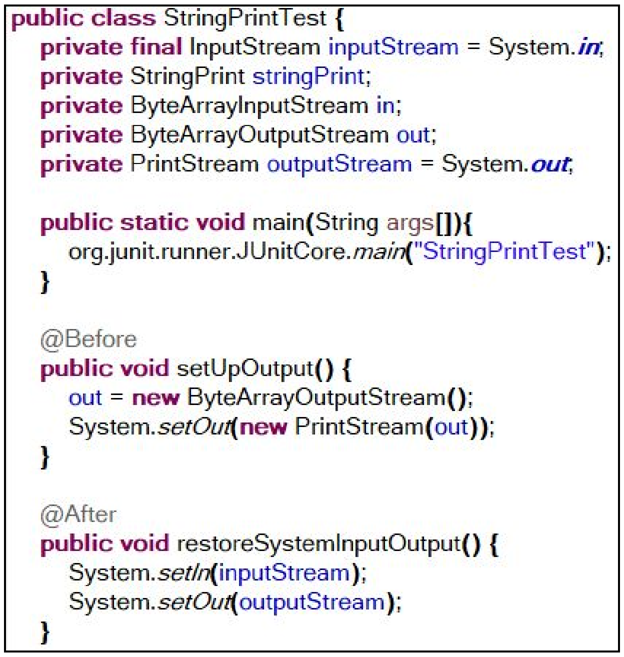
```
import static org.junit.Assert.*;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.InputStream;
import java.io.PrintStream;

public class StringPrintTest {
    private final InputStream inputStream = System.in;
    private StringPrint stringPrint;
    private ByteArrayInputStream in;
    private ByteArrayOutputStream out;
    private PrintStream outputStream = System.out;

    public static void main(String args[]){
        org.junit.runner.JUnitCore.main("StringPrintTest");
    }

    @Before
    public void setUpOutput() {
        out = new ByteArrayOutputStream();
        System.setOut(new PrintStream(out));
    }

    @After
    public void restoreSystemInputOutput() {
        System.setIn(inputStream);
        System.setOut(outputStream);
    }
```



```
    public void oneCharacter() throws Exception {▯

    public void oneString() throws Exception {▯

    @Test
    public void oneStringUsingCLI() {
        final String inputString = "Hello, World\n7";
        in = new ByteArrayInputStream(inputString.getBytes());
        System.setIn(in);
        String expectedResult = "Enter a string: \nEnter a value: "
                + "\nYou chose: Hello, World"
                .replaceAll("\\n|\\r\\n", System.getProperty("line.separator"));
        StringPrint.main(new String[0]);
        assertEquals(expectedResult, out.toString());
    }
}
```

Code:

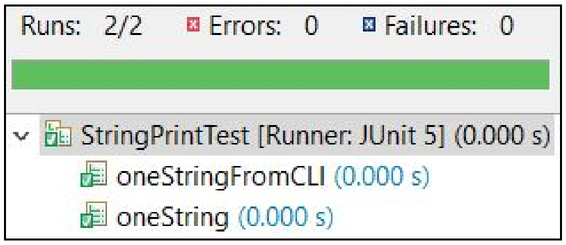
This test required creating a `main` method in `StringPrint`, which prompts the user for a String, and then for a random value; this is done by creating a `Scanner` for `System.in`. Main calls a new method, `printOutput()`, which prints the "chosen" string to standard out. Currently, the user input for random value is stored, but ignored (and untested).



```
public class StringPrintTest {
    private final InputStream inputStream = System.in;
    private StringPrint stringPrint;
    private ByteArrayInputStream in;
    private ByteArrayOutputStream out;
    private PrintStream outputStream = System.out;

    public static void main(String args[]){
        org.junit.runner.JUnitCore.main("StringPrintTest");
    }

    @Before
    public void setUpOutput() {
        out = new ByteArrayOutputStream();
        System.setOut(new PrintStream(out));
    }

    @After
    public void restoreSystemInputOutput() {
        System.setIn(inputStream);
        System.setOut(outputStream);
    }
```

Partners: Fatemeh Nouri & Danielle Schwartz
SWE 437 – Section 002
10/10/2019

Test Results:



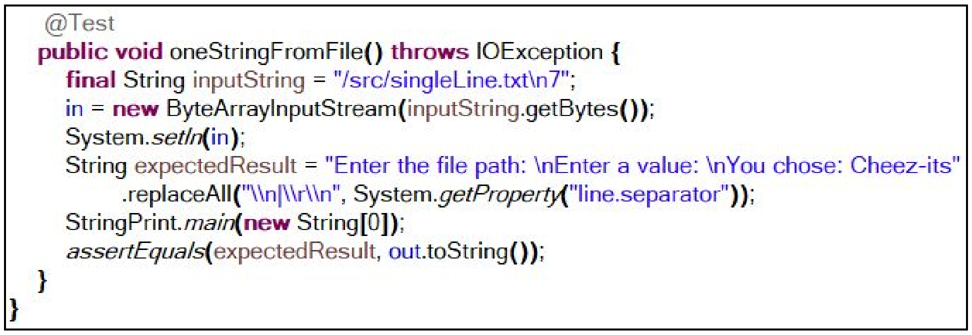
```
Runs: 2/2    Errors: 0    Failures: 0

StringPrintTest [Runner: JUnit 5] (0.000 s)
    oneStringFromCLI (0.000 s)
    oneString (0.000 s)
```

Refactor:
We renamed the original `output()` method to `getString()`, which more accurately reflects its function: acting as a "getter" method to obtain the value of `input`. This required updating the corresponding method calls in our test class. Modified `main()` to call `setValue()` to update the random value, rather than directly accessing it.

Round 4

Test:

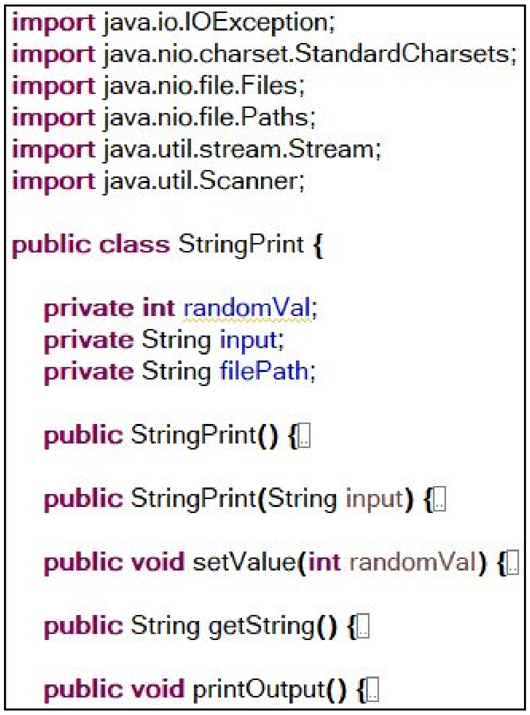
1. The user provides a string via text file.
2. The user provides a random value via command prompt.
3. The string is returned.



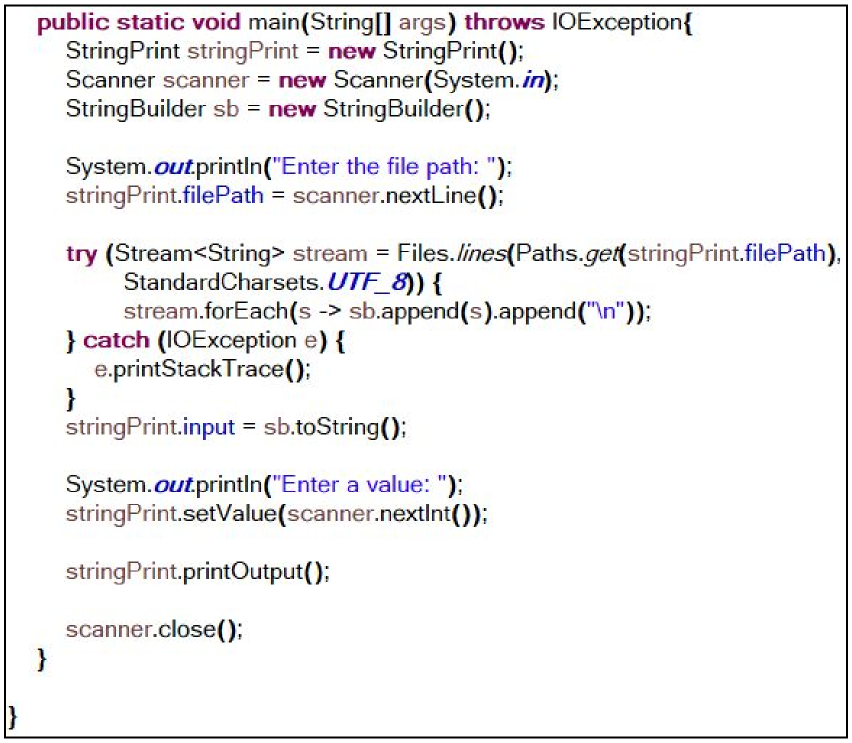
```
 @Test
public void oneStringFromFile() throws IOException {
    final String inputString = "/src/singleLine.txt\n7";
    in = new ByteArrayInputStream(inputString.getBytes());
    System.setIn(in);
    String expectedResult = "Enter the file path: \nEnter a value: \nYou chose: Cheez-its"
            .replaceAll("\\n|\\r\\n", System.getProperty("line.separator"));
    StringPrint.main(new String[0]);
    assertEquals(expectedResult, out.toString());
  }
}
```

Code:
The prompt was changed from "Enter a string: " to "Enter the file path: ". A new class-level variable, `filePath`, was added and initialized to null in the constructor. A spike was required to learn/recall an efficient way of reading from a file, line by line. In doing so, we learned a way that we found was easier to implement as seen here, although we realize this may not be the *simplest* solution to make the test pass. Several imports were added to accommodate implementing the file stream. The string read from the file was stored in `input`.

Partners: Fatemeh Nouri & Danielle Schwartz
SWE 437 – Section 002
10/10/2019

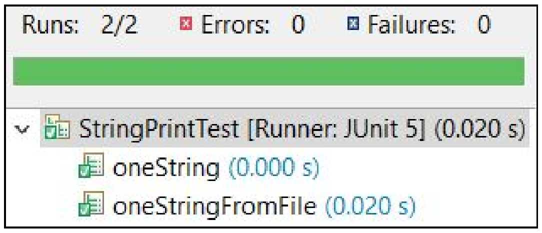


```
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.stream.Stream;
import java.util.Scanner;

public class StringPrint {

    private int randomVal;
    private String input;
    private String filePath;

    public StringPrint() {

    public StringPrint(String input) {

    public void setValue(int randomVal) {

    public String getString() {

    public void printOutput() {
```



```
  public static void main(String[] args) throws IOException{
      StringPrint stringPrint = new StringPrint();
      Scanner scanner = new Scanner(System.in);
      StringBuilder sb = new StringBuilder();

      System.out.println("Enter the file path: ");
      stringPrint.filePath = scanner.nextLine();

      try (Stream<String> stream = Files.lines(Paths.get(stringPrint.filePath),
              StandardCharsets.UTF_8)) {
              stream.forEach(s -> sb.append(s).append("\n"));
      } catch (IOException e) {
          e.printStackTrace();
      }
      stringPrint.input = sb.toString();

      System.out.println("Enter a value: ");
      stringPrint.setValue(scanner.nextInt());

      stringPrint.printOutput();

      scanner.close();
  }

}
```

Test Results:



```
Runs: 2/2    Errors: 0    Failures: 0

StringPrintTest [Runner: JUnit 5] (0.020 s)
    oneString (0.000 s)
    oneStringFromFile (0.020 s)
```

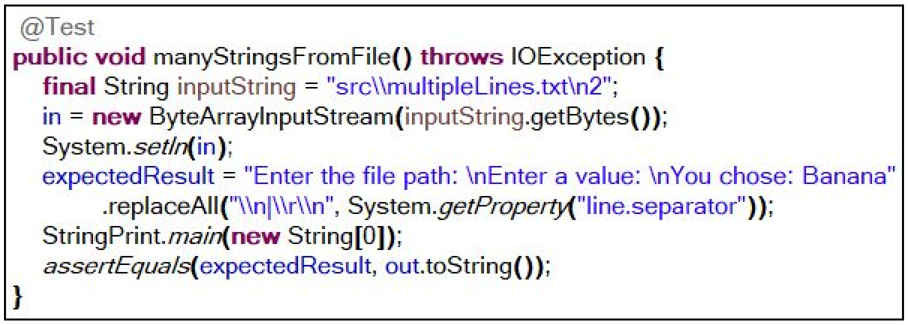
Refactor:
Since our prompt has changed to ask for a filepath, our previous test, `oneStringFromCLI()`, now fails. We chose to remove it (comment it out), as it was just a stepping-stone to get our desired functionality of reading from a file. We moved the declaration of `expectedResult` to the class level, set it to null in `@Before`, and set it in both tests. It is then passed as the argument to our asserts. `StringPrint` does not need any further refactoring at this time.

Partners: Fatemeh Nouri & Danielle Schwartz
SWE 437 – Section 002
10/10/2019

Round 5

Test:

1. The user provides multiple strings via text file.
2. The user provides a random value via command prompt.
3. The string associated with that value is returned.

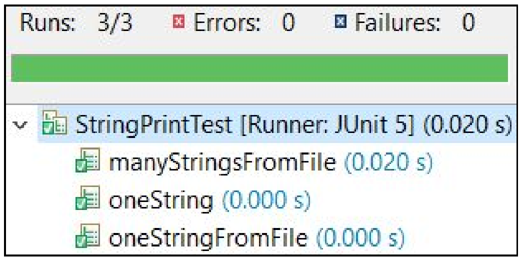


```java
@Test
public void manyStringsFromFile() throws IOException {
    final String inputString = "src\\multipleLines.txt\n2";
    in = new ByteArrayInputStream(inputString.getBytes());
    System.setIn(in);
    expectedResult = "Enter the file path: \nEnter a value: \nYou chose: Banana"
            .replaceAll("\\n|\\r\\n", System.getProperty("line.separator"));
    StringPrint.main(new String[0]);
    assertEquals(expectedResult, out.toString());
}
```

Code:

The prompt for random value was relocated to above the file stream. The line involving `StringBuilder sb` was replaced with a call to `skip().findFirst().get()`. This approach was used after another spike.

Test Results:



```
Runs:  3/3    Errors:  0    Failures:  0

StringPrintTest [Runner: JUnit 5] (0.020 s)
    manyStringsFromFile (0.020 s)
    oneString (0.000 s)
    oneStringFromFile (0.000 s)
```

Refactor:

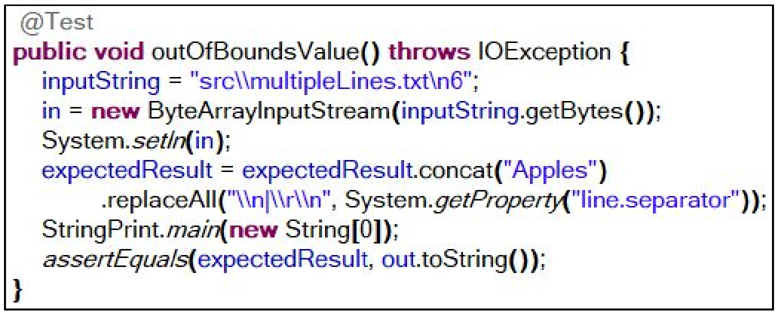
The arbitrary random value provided in our `oneStringFromFile()` test now results in an error, since there is no 7th line of our single-line file. This value was replaced with 0 so our test can pass (see results), and we have made a mental note to add input-validation to our list of test cases. We moved the declaration of `inputString` to the class level, set it to null in `@Before`, and set it in both tests. The variable `expectedResult` is now set to the common String "Enter the file path: \nEnter a value: \nYou chose: " in @Before. Our expected returned Strings can now be concatenated with this base String in each test. All references to `sb` have been removed from `StringPrint`. A set method, `setString()`, was created to replace direct access to `input`. Similarly, a get method, `getValue()`, was created to replace direct access to `randomVal`. A getter/setter were created for `filePath`, as well, to minimize maintenance debt. Tests were re-run to ensure nothing has been broken.

Partners: Fatemeh Nouri & Danielle Schwartz
SWE 437 – Section 002
10/10/2019

Round 6

Test:

1. The user provides multiple strings via text file.
2. The user provides a large number for random value via command prompt.
3. The string associated with that value is returned.

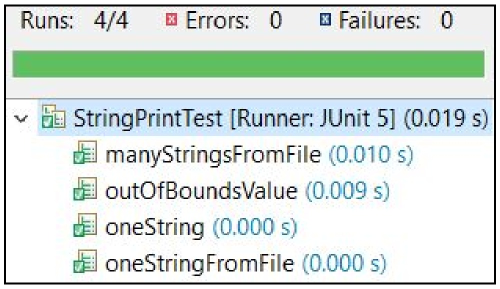


```
@Test
public void outOfBoundsValue() throws IOException {
    inputString = "src\\multipleLines.txt\n6";
    in = new ByteArrayInputStream(inputString.getBytes());
    System.setIn(in);
    expectedResult = expectedResult.concat("Apples")
            .replaceAll("\\n|\\r\\n", System.getProperty("line.separator"));
    StringPrint.main(new String[0]);
    assertEquals(expectedResult, out.toString());
}
```

Code:

Rather than choosing to interpret a random value that is larger than the number of lines in our file, we are instead "wrapping" the number around, using the modulus operator. A new variable, `long lineCount`, was created. Again using `Files.lines`, we call `count()`, which retrieves the number of lines in the specified file, and store this value in `lineCount`. We then calculate the random value mod the line count to obtain a random value within our range.

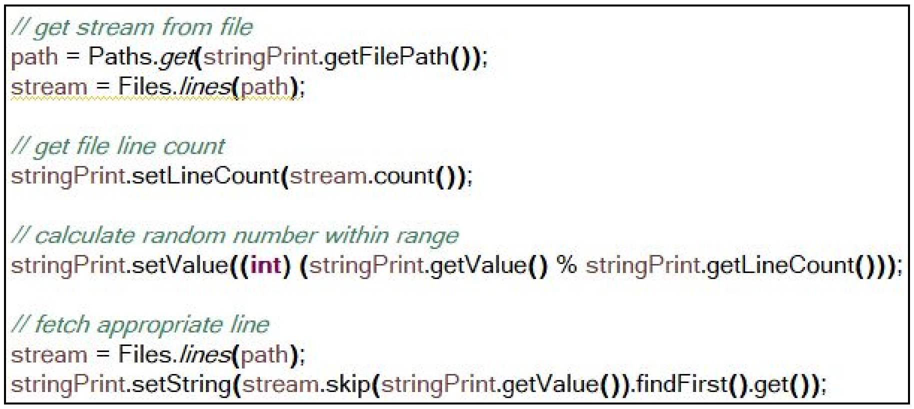
Test Results:



```
Runs:  4/4    Errors:  0    Failures:  0

StringPrintTest [Runner: JUnit 5] (0.019 s)
    manyStringsFromFile (0.010 s)
    outOfBoundsValue (0.009 s)
    oneString (0.000 s)
    oneStringFromFile (0.000 s)
```

Refactor:

Get and set methods were created for lineCount. Our usage of NIO was getting messy, so we created a local `Path` variable within `main()`, `path`, to store the path. Likewise, a local `Stream` variable was created.



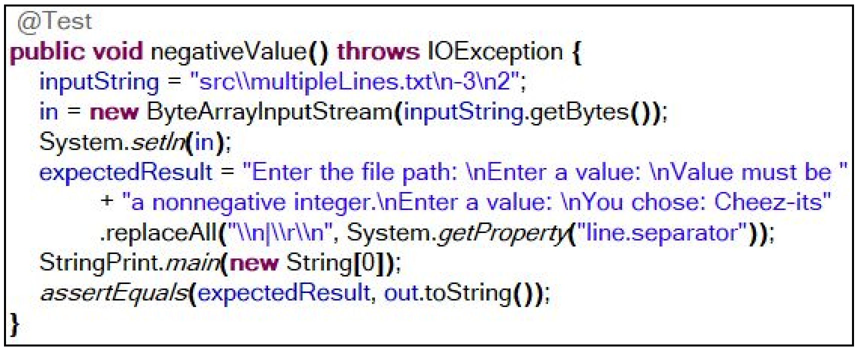
```
// get stream from file
path = Paths.get(stringPrint.getFilePath());
stream = Files.lines(path);

// get file line count
stringPrint.setLineCount(stream.count());

// calculate random number within range
stringPrint.setValue((int) (stringPrint.getValue() % stringPrint.getLineCount()));

// fetch appropriate line
stream = Files.lines(path);
stringPrint.setString(stream.skip(stringPrint.getValue()).findFirst().get());
```

Partners: Fatemeh Nouri & Danielle Schwartz
SWE 437 – Section 002
10/10/2019

Round 7

Test:

1. The user provides multiple strings via text file.
2. The user provides invalid input (negative int) for random value via command prompt.
3. The user is re-prompted to enter valid input.
4. The user provides a random value via command prompt.
5. The string associated with that value is returned.

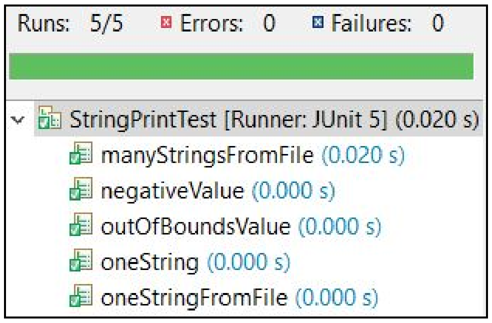


```
@Test
public void negativeValue() throws IOException {
    inputString = "src\\multipleLines.txt\n-3\n2";
    in = new ByteArrayInputStream(inputString.getBytes());
    System.setIn(in);
    expectedResult = "Enter the file path: \nEnter a value: \nValue must be "
            + "a nonnegative integer.\nEnter a value: \nYou chose: Cheez-its"
            .replaceAll("\\n|\\r\\n", System.getProperty("line.separator"));
    StringPrint.main(new String[0]);
    assertEquals(expectedResult, out.toString());
}
```

Code:

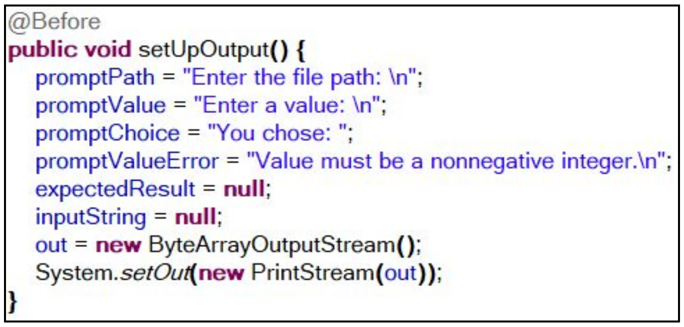
Wrapped prompt for random value in a `do...while loop`. Check to see if the number is greater than or equal to zero. If not, prints an error message and prompts again.

Test Results:



```
Runs: 5/5    Errors: 0    Failures: 0

StringPrintTest [Runner: JUnit 5] (0.020 s)
    manyStringsFromFile (0.020 s)
    negativeValue (0.000 s)
    outOfBoundsValue (0.000 s)
    oneString (0.000 s)
    oneStringFromFile (0.000 s)
```

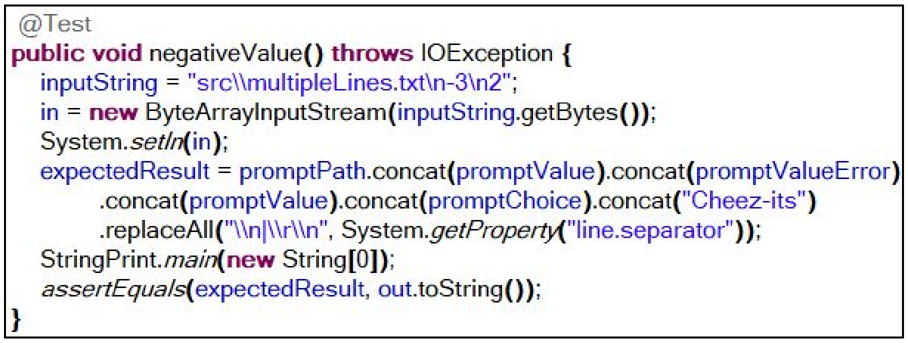
Refactor:



```
@Before
public void setUpOutput() {
    promptPath = "Enter the file path: \n";
    promptValue = "Enter a value: \n";
    promptChoice = "You chose: ";
    promptValueError = "Value must be a nonnegative integer.\n";
    expectedResult = null;
    inputString = null;
    out = new ByteArrayOutputStream();
    System.setOut(new PrintStream(out));
}
```

Refactor:

Our beginning `expectedResult` String in our test oracle now has multiple possibilities, so we broke it into parts, define them in `@Before`, and build the string as needed in each Test (see left above right).



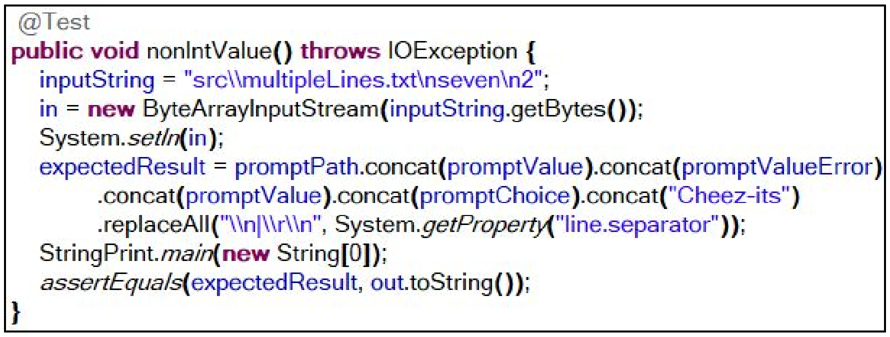
```
@Test
public void negativeValue() throws IOException {
    inputString = "src\\multipleLines.txt\n-3\n2";
    in = new ByteArrayInputStream(inputString.getBytes());
    System.setIn(in);
    expectedResult = promptPath.concat(promptValue).concat(promptValueError)
            .concat(promptValue).concat(promptChoice).concat("Cheez-its")
            .replaceAll("\\n|\\r\\n", System.getProperty("line.separator"));
    StringPrint.main(new String[0]);
    assertEquals(expectedResult, out.toString());
}
```

Partners: Fatemeh Nouri & Danielle Schwartz
SWE 437 – Section 002
10/10/2019

Round 8

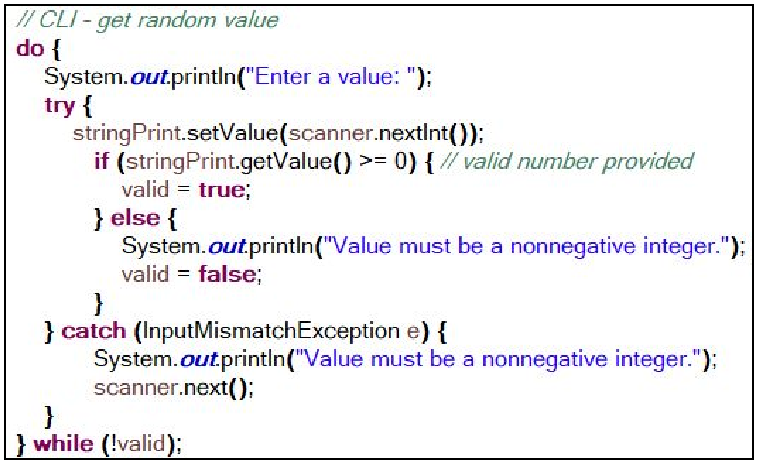
Test:

1. The user provides multiple strings via text file.
2. The user provides invalid input (non-int) for random value via command prompt.
3. The user is re-prompted to enter valid input.
4. The user provides a random value via command prompt.
5. The string associated with that value is returned.



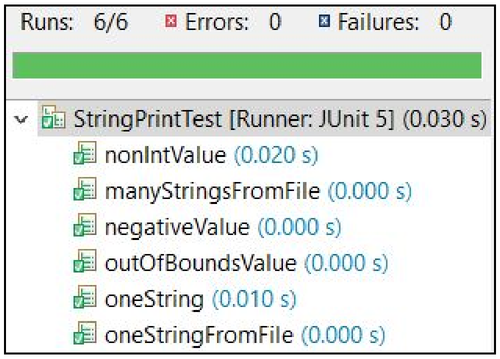
```
 @Test
public void nonIntValue() throws IOException {
    inputString = "src\\multipleLines.txt\nseven\n2";
    in = new ByteArrayInputStream(inputString.getBytes());
    System.setIn(in);
    expectedResult = promptPath.concat(promptValue).concat(promptValueError)
            .concat(promptValue).concat(promptChoice).concat("Cheez-its")
            .replaceAll("\\n|\\r\\n", System.getProperty("line.separator"));
    StringPrint.main(new String[0]);
    assertEquals(expectedResult, out.toString());
}
```

Code:
We surrounded scanner.nextInt() (and the following error-checking) in a try...catch to account for non-integer input. If a non-integer was provided, prints an error message and prompts again.



```
// CLI - get random value
do {
    System.out.println("Enter a value: ");
    try {
        stringPrint.setValue(scanner.nextInt());
        if (stringPrint.getValue() >= 0) { // valid number provided
            valid = true;
        } else {
            System.out.println("Value must be a nonnegative integer.");
            valid = false;
        }
    } catch (InputMismatchException e) {
        System.out.println("Value must be a nonnegative integer.");
        scanner.next();
    }
} while (!valid);
```

Test Results:



```
Runs:  6/6    Errors:  0    Failures:  0

StringPrintTest [Runner: JUnit 5] (0.030 s)
    nonIntValue (0.020 s)
    manyStringsFromFile (0.000 s)
    negativeValue (0.000 s)
    outOfBoundsValue (0.000 s)
    oneString (0.010 s)
    oneStringFromFile (0.000 s)
```

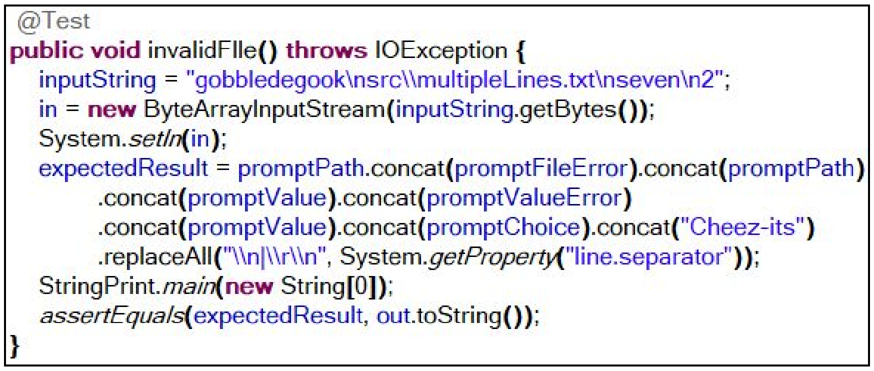
Refactor:
Changed our default randomValue from 0 to -1 so that we eliminate the potential for false positives from our test oracle.

Partners: Fatemeh Nouri & Danielle Schwartz
SWE 437 – Section 002
10/10/2019

Round 9

Test:

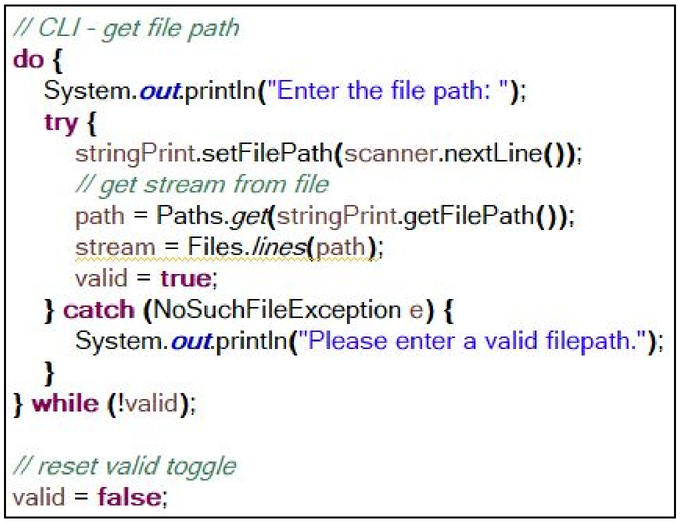
1. The user provides an invalid text file.
2. The user is re-prompted to enter a valid path.
3. The user provides invalid input for random value via command prompt.
4. The user is re-prompted to enter valid input.
5. The user provides a random value via command prompt.
6. The string associated with that value is returned.



```
@Test
public void invalidFIle() throws IOException {
   inputString = "gobbledegook\nsrc\\multipleLines.txt\nseven\n2";
   in = new ByteArrayInputStream(inputString.getBytes());
   System.setIn(in);
   expectedResult = promptPath.concat(promptFileError).concat(promptPath)
         .concat(promptValue).concat(promptValueError)
         .concat(promptValue).concat(promptChoice).concat("Cheez-its")
         .replaceAll("\\n|\\r\\n", System.getProperty("line.separator"));
   StringPrint.main(new String[0]);
   assertEquals(expectedResult, out.toString());
}
```

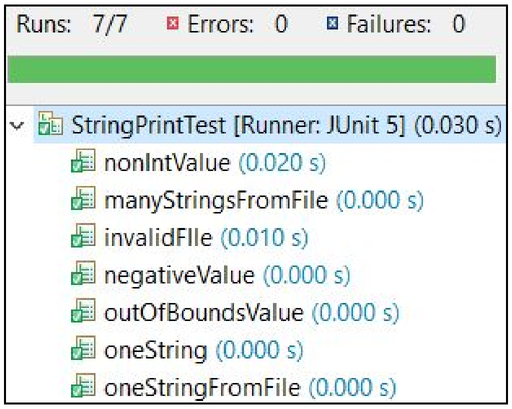
Code:

Mimicking rounds 7 and 8, the prompt for the file path was surrounded in a `do...while` loop and the `Paths.get()` call was placed in a `try...catch`. We reset our `valid` toggle before continuing on to the random value checks.



```
// CLI - get file path
do {
   System.out.println("Enter the file path: ");
   try {
      stringPrint.setFilePath(scanner.nextLine());
      // get stream from file
      path = Paths.get(stringPrint.getFilePath());
      stream = Files.lines(path);
      valid = true;
   } catch (NoSuchFileException e) {
      System.out.println("Please enter a valid filepath.");
   }
} while (!valid);

// reset valid toggle
valid = false;
```

Test Results:



```
Runs: 7/7    Errors: 0    Failures: 0

StringPrintTest [Runner: JUnit 5] (0.030 s)
   nonIntValue (0.020 s)
   manyStringsFromFile (0.000 s)
   invalidFIle (0.010 s)
   negativeValue (0.000 s)
   outOfBoundsValue (0.000 s)
   oneString (0.000 s)
   oneStringFromFile (0.000 s)
```

Refactor:

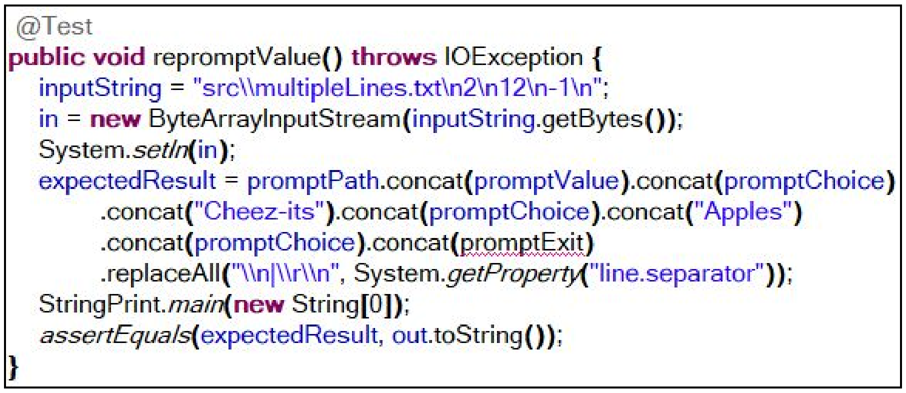
No additional changes were made at this time.

Partners: Fatemeh Nouri & Danielle Schwartz
SWE 437 – Section 002
10/10/2019

Round 10

Test:

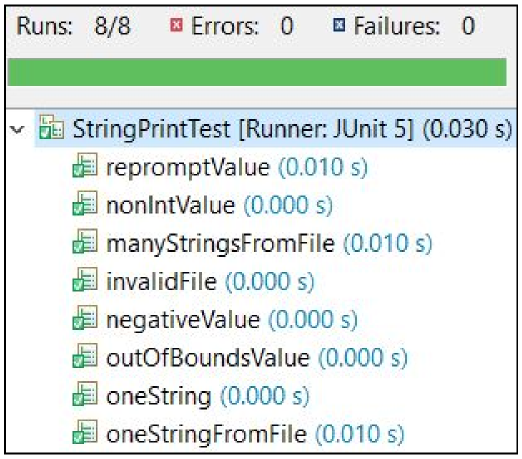
1. The user provides multiple strings via a text file.
2. The user provides a random value via command prompt.
3. The string associated with that value is returned.
4. Steps 2-3 repeat until the user indicates they want to exit.



```java
@Test
public void repromptValue() throws IOException {
    inputString = "src\\multipleLines.txt\n2\n12\n-1\n";
    in = new ByteArrayInputStream(inputString.getBytes());
    System.setIn(in);
    expectedResult = promptPath.concat(promptValue).concat(promptChoice)
            .concat("Cheez-its").concat(promptChoice).concat("Apples")
            .concat(promptChoice).concat(promptExit)
            .replaceAll("\\n|\\r\\n", System.getProperty("line.separator"));
    StringPrint.main(new String[0]);
    assertEquals(expectedResult, out.toString());
}
```

Code:
Changed default random value to `-2` so that `-1` could be the user's signal to exit. Surrounded the value prompt/print with a `do...while` loop; repeats until user enters `-1`.

Test Results:



```
Runs:  8/8     Errors:  0     Failures:  0

StringPrintTest [Runner: JUnit 5] (0.030 s)
    repromptValue (0.010 s)
    nonIntValue (0.000 s)
    manyStringsFromFile (0.010 s)
    invalidFile (0.000 s)
    negativeValue (0.000 s)
    outOfBoundsValue (0.000 s)
    oneString (0.000 s)
    oneStringFromFile (0.010 s)
```

Refactor:
We were required to adjust all previous tests to accept a final round of user input, so that the user may signal to exit the program.

Partners: Fatemeh Nouri & Danielle Schwartz
SWE 437 – Section 002
10/10/2019

Round 11

Test:

1. The user provides multiple strings via a text file.
2. The user specifies whether they would like to make a selection with or without replacement.
3. The user provides a random value via command prompt.
4. The string associated with that value is returned.
5. Steps 2-3 repeat until the user indicates they want to exit.