Testing I/O Methods & JavaDocs

`System.out` and `System.in` are Streams

- Streams are objects designed to send data to and from a source.
- `System.out` is configured by default to be your console's output
- `System.in` is configured by default to be your console's input
- `FileInputStream` objects can be created to open a file for reading
- `FileOutputStream` objects can be created to open a file for writing

Setting `System.out` and `System.in`

Let's venture over to `System` JavaDocs to get an idea.

Here we go!

Setting `System.in`

`System.setIn(InputStream in)` reassigns the "standard" input stream to be `in`.

Replacing `System.in` with a file

```
InputStream file = new FileInputStream("my_file.txt");
System.setIn(file);
```

Replacing `System.in` with some particular String

```
String data = "Some String that has the data you want";
InputStream stringData = new ByteArrayInputStream(data.getBytes());
System.setIn(stringData);
```

Why is this useful?

Imagine, for example, that I want to test some code that takes in a sequence of user inputs. Each input indicates where a drawn card should go on a board, and once we've received enough inputs, we'll calculate a score.

```
public int playGame() {
    Scanner scnr = new Scanner(System.in);
    while (emptySpaces()) {
        String choice = scnr.next(); // reads from System.in
        handleChoice(choice);
    }
    return score();
}
```

Why is this useful?

A basic test case for this would look like the following: (pretend for now that nothing is random!)

```
@Test
public void testScoringPlacingInOrder() {
    BlackjackSolitaire b = new BlackjackSolitaire();
    int actual = playGame(); // the program will wait for you to respond over and over
    int expected = 14;
    assertEquals(expected, actual);
}
```

This would take forever, even though it's a valid test!

Why is this useful?

Instead, try this:

```
@Test
public void testScoringPlacingInOrder() {
    String data = "1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16";
    InputStream stringData = new ByteArrayInputStream(data.getBytes());
    System.setIn(stringData);

    BlackjackSolitaire b = new BlackjackSolitaire();
    int actual = playGame(); // automatically reads from our String now
    int expected = 14;
    assertEquals(expected, actual);
}
```

When we ask for user input, it's read from the Stream we made! ✅

What if we wanted to go back to the original `System.in`?

Naively, `System.setIn(System.in)`...

What's wrong with this?

What if we wanted to go back to the original `System.in`?

```
String data = "1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16";
InputStream stringData = new ByteArrayInputStream(data.getBytes());
System.setIn(stringData);
// then, later:
System.setIn(System.in)
```

is, essentially, just:

```
System.in = somethingElse;
System.in = System.in; // this would never do anything ever!
```

What if we wanted to go back to the original `System.in`?

Best practice: save a reference to the original `System.in`, 'cuz we don't really have a good way of getting it back.

```
InputStream original = System.in;

String data = "1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16";
InputStream stringData = new ByteArrayInputStream(data.getBytes());
System.setIn(stringData);
// then, later:
System.setIn(original)
```

Exercise:

1. Write a class with a method `public static double avgOfInputs()`
    1. The method should read an int from the console indicating how many numbers it will read in.
    2. The method should then read that many numbers, and then compute and return the average.
2. Test your method manually by calling it and providing the input by typing.
3. Swap computers with a partner (or send your code, or something) and write three JUnit test cases manipulating `System.in` to avoid retyping inputs and making testing automatic.

Another Improvement to JUnit Testing

In each test case, you will be resetting `System.in`, so:

1. You should be saving the original `System.in` in case you need it again
2. If you are giving input `Strings` or files through your `Streams`, you should be prepared to close them off after each test to avoid weird bugs.

```
class AveragerTest {
    static InputStream originalIn;

    static final double DELTA = 1e-6;

    @BeforeAll
    static void setUpBeforeClass() throws Exception {
        originalIn = System.in;
    }

    @AfterEach
    void tearDown() throws Exception {
        System.in.close();
        System.setIn(originalIn);
    }

    // Test cases go here, like usual
}
```

`BeforeAll` and `AfterEach`

- A method annotated with `@BeforeAll`...
  - gets run before any test case gets run
  - must be `static void`
- A method annotated with `@AfterEach`...
  - gets run after each test case finishes
  - must be `void`

Capturing Printed Output Directly

We'll replace `System.out` with a `PrintStream`.

- The `PrintStream` has all the methods that `System.out` already had for printing (`println, print, printf`)
- All print calls will be stored in this Stream of data.
- We can inspect the stream's contents directly with `toString()`.

Replacing `System.out`

```
OutputStream outputStream = new ByteArrayOutputStream();
PrintStream printStream = new PrintStream(outputStream);
System.setOut(printStream);

System.out.println("Hello!!!!");
```

Nothing appears on the console, all text is held in the `PrintStream`.

Inspecting the Replaced `System.out`

```
OutputStream outputStream = new ByteArrayOutputStream();
PrintStream printStream = new PrintStream(outputStream);
System.setOut(printStream);

System.out.println("Hello!!!!");

String printedContents = outputStream.toString();
```

Exercise:

1. Write a class with a method
   `public static void sortedChars(String s)`
   1. The method should take a String as input.
   2. The method should create a set of characters in that String.
   3. The method should repeatedly call `Collections.min()` to print and remove the smallest remaining character in the set until there are no remaining characters.
2. Write three JUnit test cases manipulating `System.out` to compare the printed outputs against an expected value.
   e.g. "harry" --> a h r y