

SWE 619 Assignment 5

// TODO

Goal: Iteration Abstraction, and a bit of Type Abstraction.

The point of this exercise is to show some of the power of interfaces in Java. In particular, the Iterable interface allows for the writing of extremely safe and compact for loops.

Build a version of the String class called MyString that implements the Java Iterable interface. Like String, MyString should be an immutable class. The iterator() method that you supply in MyString should produce the characters in the string (as String objects), one at a time, from left to right. For example, iterating over the integer "cat" should produce a "c", and then an "a", and then a "t". The MyString class doesn't actually have to have much additional functionality; you may find a simple public String get() method sufficient.

Provide a quality set of JUnit tests to test your implementation. One of your tests should iterate over a MyString and produce the palindrome (ie reverse) of that string. For example:

```
MyString m = new MyString("bat");
String res = "";
for (String i : m) {
    res = i + res;
}
// res has the value "tab";
```

Note: If you wish, you can implement the iterator "from scratch". Or you can use an existing iterator. The latter option is much shorter, and hence is probably better. But you have to be careful: The fact that MyString is immutable tells you how you have to implement the remove() method.

Turn in a **story**. This means that it is possible to grade your assignment simply by looking at a document. Hence, the document needs to include code fragments, individual tests, screenshots, etc. -- as needed -- to show that you have satisfied each item in the assignment.

We need to create a new class **MyString** and write JUnit tests to test the corresponding class. Hence, I first created a **new gradle application** as follows:

- Open command prompt
- `cd` to the directory where you want to initialise your project
- Type `gradle init`

```
Starting a Gradle Daemon (subsequent builds will be faster)
Select type of project to generate:
 1: basic
 2: application
 3: library
 4: Gradle plugin
Enter selection (default: basic) [1..4] 2

Select implementation language:
 1: C++
 2: Groovy
 3: Java
 4: Kotlin
 5: Swift
Enter selection (default: Java) [1..5] 3

Select build script DSL:
 1: Groovy
 2: Kotlin
Enter selection (default: Groovy) [1..2] 1

Select test framework:
 1: JUnit 4
 2: TestNG
 3: Spock
 4: JUnit Jupiter
Enter selection (default: JUnit 4) [1..4] 4

Project name (default: SWE 619 Assignment 5):
Source package (default: SWE.Assignment):

> Task :init
Get more help with your project: https://docs.gradle.org/5.6.2/userguide/tutorial_java_projects.html

BUILD SUCCESSFUL in 1m 17s
2 actionable tasks: 2 executed
C:\Users\haram\OneDrive\Desktop\SWE 619 Assignment 5>
```

Now, we have our project ready to roll!

LET'S GET STARTED!!!

1. Creating the class MyString:

The MyString class has to be immutable, which means:

- It is a `final` class
- Its fields are private.
- All its immutable fields are final as well.

```
package MyString;

public final class MyString implements Iterable<String> {
    private final String input;
    private final ArrayList<String> aList;
}
```

2. Constructor for MyString

The constructor takes a single `String` parameter as input and initialises the `input` variable to that parameter. Then, we convert the given `String` input to an `ArrayList` of `String` characters.

The characters are returned as `Strings` and not as `chars` in order to maintain the contract.

```
public MyString(String input) {  
    this.input = input;  
    String str[] = input.split("");  
    this.aList = new ArrayList<String>(Arrays.asList(str));  
}
```

3. Implementing the `iterator()` method

Since our class `MyString` implements the `Iterable<>` interface, it must override the `iterator()` method. In this case, we implement a custom `Iterator` class called `MyStringIterator`. Hence, all the `iterator()` method does is return an instance of `MyStringIterator`.

```
@Override  
public Iterator<String> iterator() {  
    return new MyStringIterator();  
}
```

4. Implementing `MyStringIterator` class

In order to implement a custom iterator, the custom class must implement the `Iterator<>` interface. This means we need to override `hasNext()` and `next()` methods in our class. Our class also consists of a private `int` field named `position` which acts like a pointer in our iterator.

```
private class MyStringIterator implements Iterator<String> {  
    private int position = 0;  
  
    @Override  
    public boolean hasNext() {
```

```
        //To be implemented
    }

    @Override
    public String next() {
        //To be implemented
    }
}
```

5. Implementing `next()` and `hasNext()` methods:

```
@Override
public boolean hasNext() {
    return position < input.length();
}

@Override
public String next() {
    position++;
    return aList.get(position-1);
}
```

Now, we're ready with our `MyString` class along with our custom iterator.
It's time to do some TESTING!

TESTING:

1. Test if `next()` returns the next character

The first test we write is to check whether the implementations of `next()` and `hasNext()` in our iterator works as expected.

```
@Test public void testNextReturnsNextCharacter() {  
    MyString mynewString = new MyString("cat");  
    Iterator<String> itr = mynewString.iterator();  
    assertEquals(true, itr.hasNext());  
    assertEquals("Returned the next character in the MyString  
object", itr.next(), "c");  
}
```

2. Test Palindrome

The second test we write performs a check on a use case of our `MyString` class - palindrome.

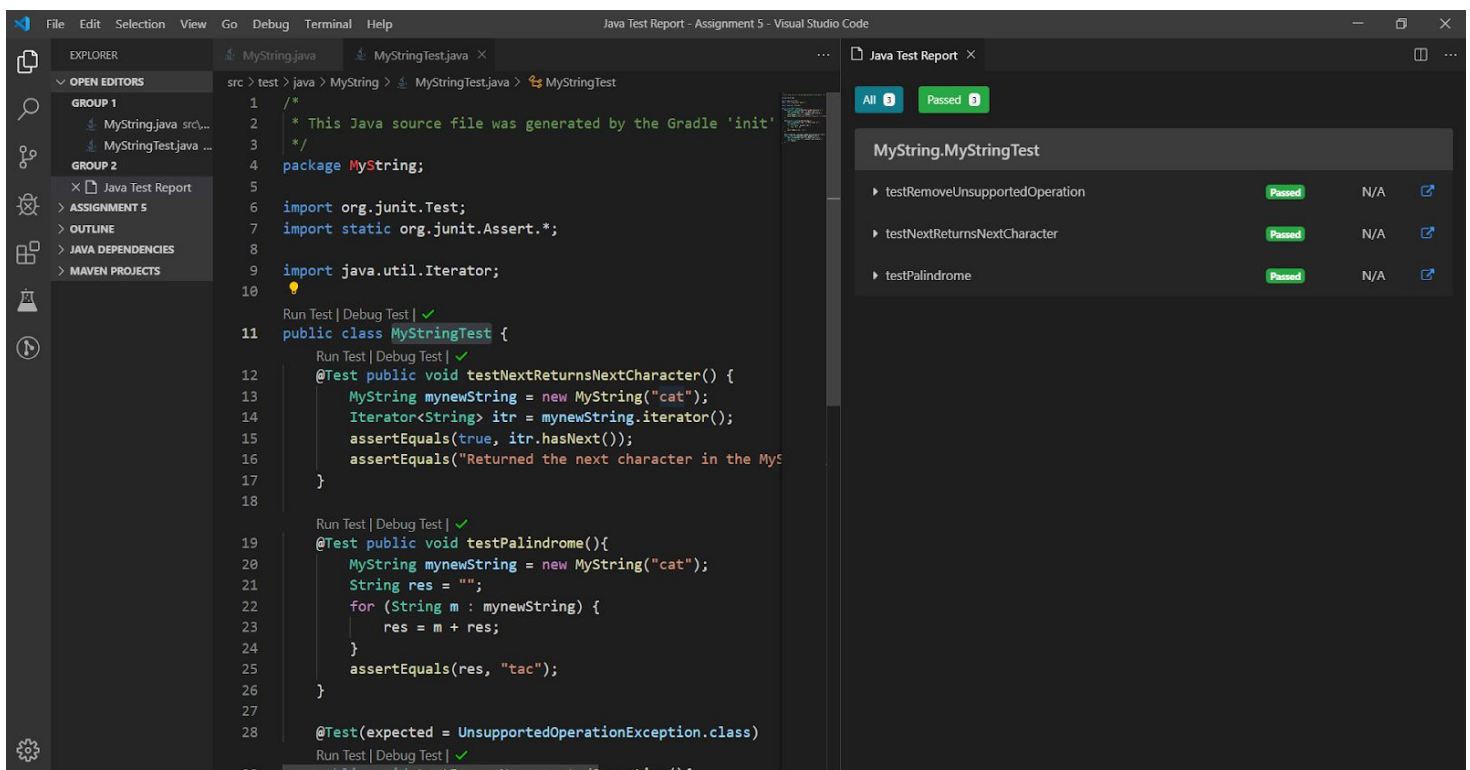
```
@Test public void testPalindrome(){  
    MyString mynewString = new MyString("cat");  
    String res = "";  
    for (String m : mynewString) {  
        res = m + res;  
    }  
    assertEquals(res, "tac");  
}
```

3. Test if `remove()` throws `UnsupportedOperationException`

The third test makes sure that our implementation of `MyStringIterator` and `MyString` treats `remove()` as an unsupported operation (by throwing `UnsupportedOperationException`). This is because since `MyString` is immutable, `remove()` method cannot be implemented.

```
@Test(expected = UnsupportedOperationException.class)
public void testRemoveUnsupportedOperation(){
    MyString mynewString = new MyString("cat");
    Iterator<String> itr = mynewString.iterator();
    itr.remove();
}
```

The test results of the above three tests are as follows:



The screenshot displays the Visual Studio Code interface with the Java Test Report for `MyStringTest`. The report shows three tests, all of which passed:

Test Name	Status	Duration	Link
testRemoveUnsupportedOperation	Passed	N/A	View Details
testNextReturnsNextCharacter	Passed	N/A	View Details
testPalindrome	Passed	N/A	View Details

Conclusion:

Thus, we implemented an immutable `MyString` class implementing the `Iterable<>` interface and `MyStringIterator` class implementing the `Iterator<>` interface as well as performed JUnit testing on the class to make sure we abided by the contract.