JSoup, a tool you need to parse HTML

Part I: Introduction, Set Up and Partitioning

Introduction to JSoup

`JSoup` is a very useful Java library for working with real-world HTML/XML. During the real world process, we usually meet with HTML or XML files. In Python, we mostly use `beautifulSoup` to handle these messy and dummy HTML files into neat and clear data structures. In Java, we use `JSoup`, converting HTML or XML into neat and clear data structures by its internal `parser` function. Its speed is much faster than Python and one of your best choice for Java projects.

Our Github link is https://github.com/duke326/SWE261. Our collaborators are Lai Wang, Xinyi Hu and Yu Sun. We have already added TAs and Professor Jones into our project.

Useful Functions of JSoup

It uses the best DOM method of HTML5 and CSS selector to provide a very convenient API for obtaining URLs and extracting and processing data.

jsoup implements the WHATWG HTML5 specification, and parses HTML to the same DOM as modern browsers.

- scrape and parse HTML from a URL, file, or string
- find and extract data, using DOM traversal or CSS selectors
- manipulate the HTML elements, attributes, and text
- clean user-submitted content against a safe-list, to prevent XSS attacks
- output tidy HTML

jsoup is designed to handle all kinds of HTML/XML found; from raw and verified to invalid tag soup; jsoup will create a wise parse tree.

Detailed analysis into JSoup

To dive deep into some data of `JSoup`, we have counted the lines of code for `JSoup` and it appears to be 32887.

In ubuntu system, I used this command to calculate the number of code except the comments.

```
find . -type f -name '*.java' | xargs cat | wc -l
```

Also, we have counted number of java files, and the result is 132.

In ubuntu system, I used this command to calculate the number of java files.

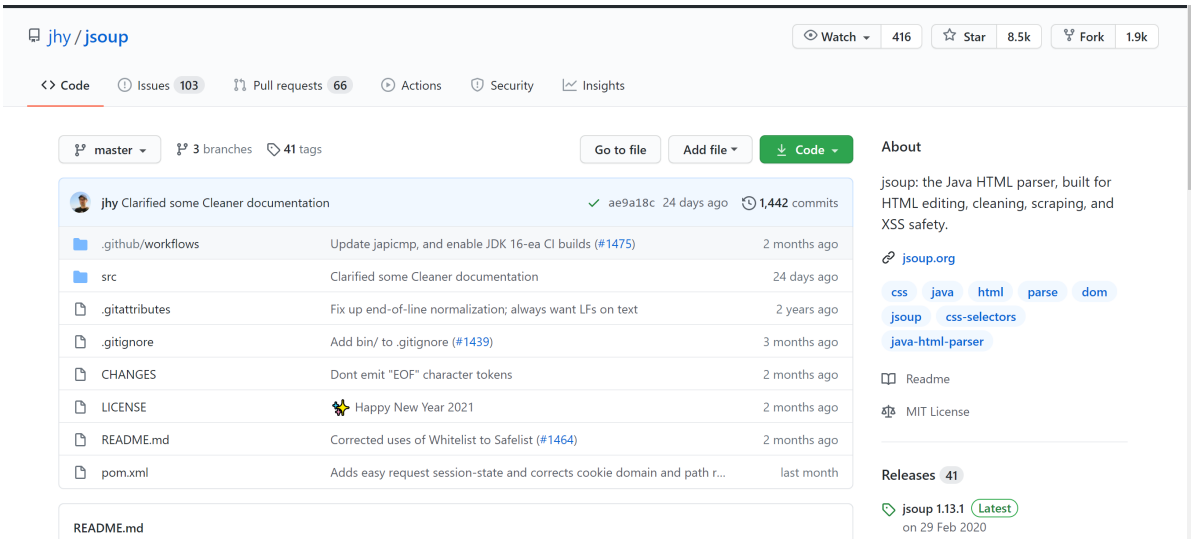
```
find . -name '*.java' | wc -l
```

The language of `JSoup` is 100% of Java.

Set Up environment for JSoup

Fork & Add

Firstly, we need to fork this project into our own repository. Using GitHub, the original github link is https://github.com/jhy/jsoup/. To fork, we press the button in the right upper corner of this page, and put it into our own account. One of our team member, Yu Sun forked this repository, and add other team members into collaborators.



To update the code in this repository, normally we discuss together, so that we push the changes directly into the master, or fork the repository into our own branch, and pull request the changes into the master.

Our link is here: https://github.com/duke326/SWE261. We have all of our team members, Lai Wang, Sun Yu and Xinyi Hu in the repo. Also we have included Prof. Jones and TA Maruf.

Build

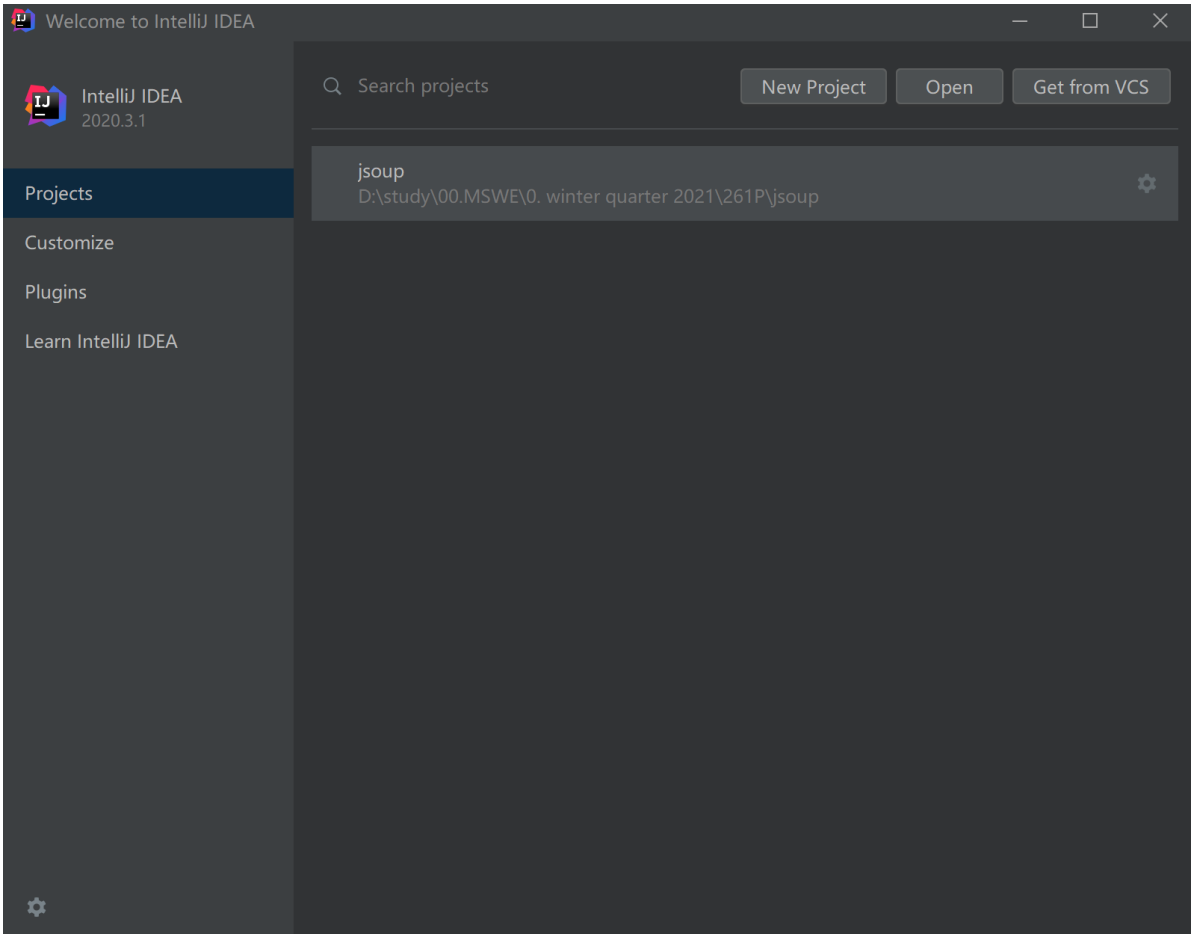
After the fork, next step, we want to know how to build our project. All of our team members use IntelliJ IDEA provided by JetBrain so that we use IntelliJ IDEA as our main development IDE.

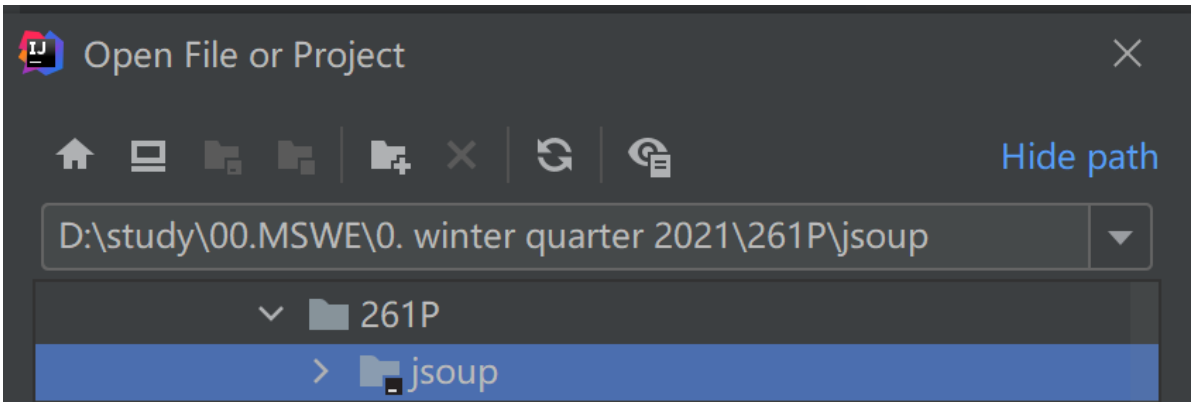
Using IntelliJ IDEA, this project uses Maven as the build infrastructure for Java projects.

Maven uses rules and patterns to provide an integrated build system. Each Maven project uses a set of shared plugins retrieved from the Maven repository, and when the project is built, Maven performs a set of predefined tasks throughout its life cycle.

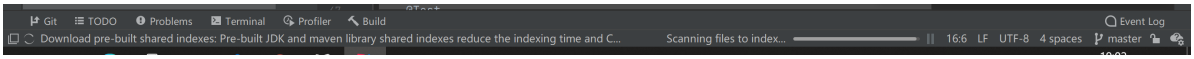
Process of using IntellJ IDEA to open Maven project

(Sample Environment: Windows 10)





Then your idea will automatically download the dependency.



How to run test cases?

When the progress bar is full, now you can run your program! Including any test case in the test folder!

Setup for using JUnit 5

To use Maven you have to use updated version for your build plug-ins and add several dependencies.

To use JUnit 5 with Maven, we need to change `pom.xml`. In the `pom.xml`, you nd to add its plugin as `maven-surefire-plugin`, and its version is `2.22.2`. To see how to write that, please refer to our code in the line 221-241.

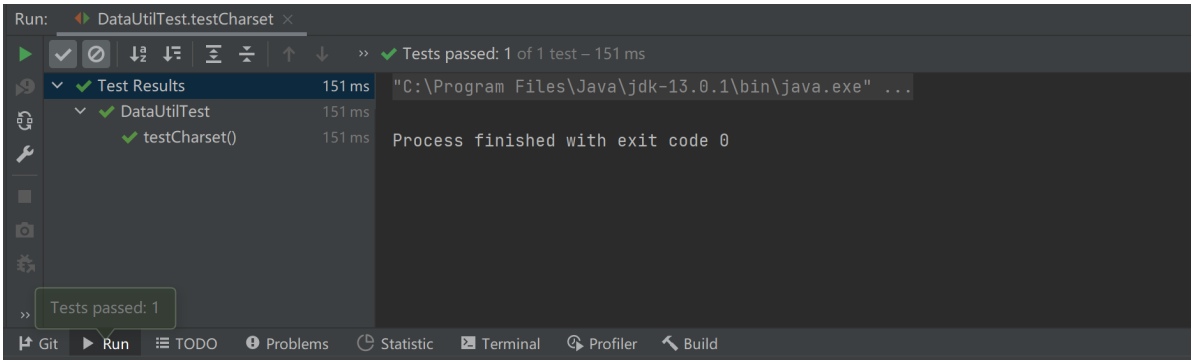
Sample to run

For example, in `test/java/ org.json/helper/DataUtilTest.java`

```
public class DataUtilTest {
    @Test
    public void testCharset() {
        assertEquals( expected: "utf-8", DataUtil.getCharsetFromContentType("text/html;charset=utf-8 "));
        assertEquals( expected: "UTF-8", DataUtil.getCharsetFromContentType("text/html; charset=UTF-8"));
        assertEquals( expected: "ISO-8859-1", DataUtil.getCharsetFromContentType("text/html; charset=ISO-8859-1"));
        assertNull(DataUtil.getCharsetFromContentType("text/html"));
        assertNull(DataUtil.getCharsetFromContentType(null));
        assertNull(DataUtil.getCharsetFromContentType("text/html;charset=Unknown"));
    }

    @Test
```

Press the little green run button, the output would be like this:



Existing Test cases

In the first place, the existing test case we found is **black box testing**, which means that we ignore the internal functions, just check whether the output provided by function are the same as our desired output. In IEEE, black box testing means that one kind of testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions.

- In one word, it is a testing method that has no view of code.

Usually, using black box testing, it try to select inputs that are especially valuable so that it can test the functions equally. The reason for that is simple and clear.

- Using non-uniform method, also called **systematic testing** can deal with the sparse problem in the input space. A classic example is Java class "root" mentioned in the class.
- Using this kind of systematic testing can find bugs and remove them from hay more effectively.

As evolved in our class, the existing test case also included **continuous integration**. As we will talk about in report part IV, continuous integration means the practice of merging the working copies of all developers into the shared mainline several times a day. This kind of software testing technique is proposed by Grady Booch. It has four important components.

- Firstly, continuous integration is originated from extreme programming development process. Due to high amount of development, there would be lots of changes of code and different progress align with different colleagues. It is super important under such extreme programming environment
- Secondly, continuous integration needs to be performed, even for minor changes. Therefore, every developer are in the same pace for the project.
- Thirdly, every developer in the group and in the project needs to commit their changes every day. Therefore, every developer are in the same pace for the project.

- Fourthly, every version, especially the latest version, needs to build and pass all the tests. Otherwise, this program cannot work, and need to check its methods and functions.

In our project, it used **JUnit**. *JUnit 5* is the next generation of JUnit. The goal is to create an up-to-date foundation for developer-side testing on the JVM. This includes focusing on Java 8 and above, as well as enabling many different styles of testing.

How to run test cases? JUnit!

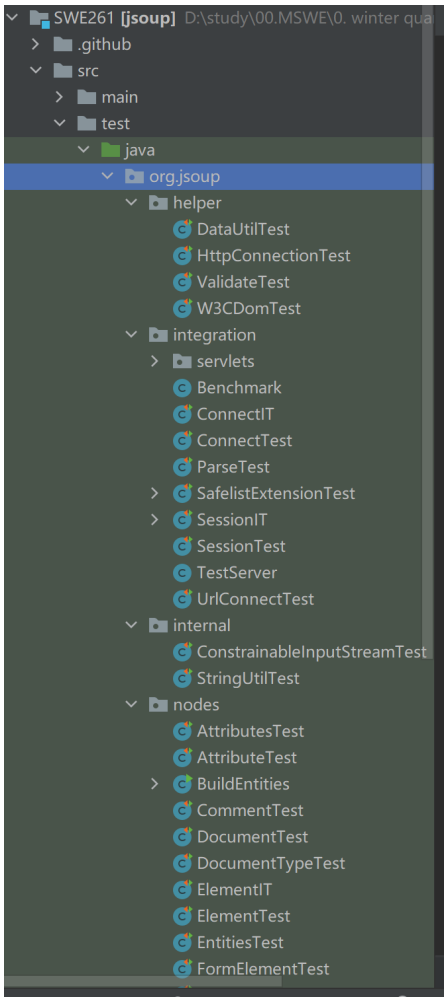
To run the test cases, we need to add JUnit into our project! As we talked about in the build section, we have already put JUnit in our pom.xml file, so that you can easily run JUnit in the IntellJ IDEA without worrying about installing it.

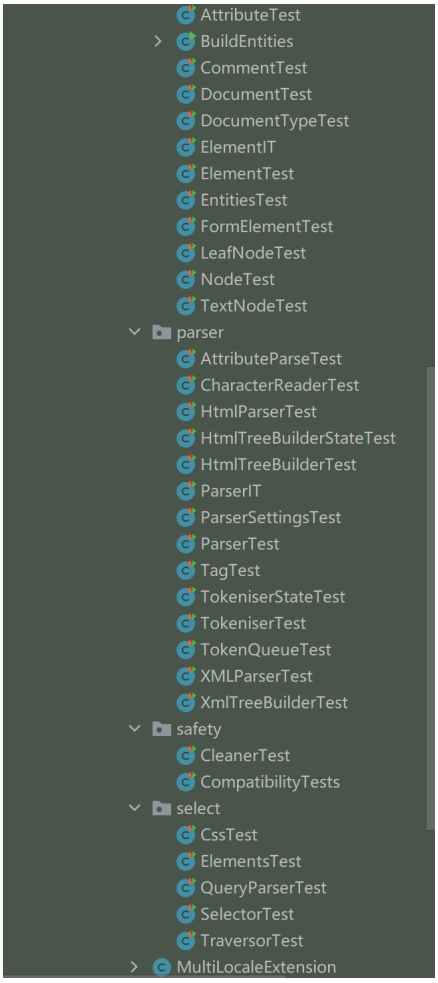
JUnit *test* is a method contained in a class that is only used for testing. This is called *test category*. To define a specific method as a test method, we usually annotate the method with `@Test`.

This method executes the code under test. Use the *assert* method provided by JUnit or other assert frameworks to compare the expected result with the actual result. These calls are usually called *assert* or *assert statement*.

The assert statement can usually define the message that will be displayed if the test fails. You should provide meaningful messages here to help users identify and resolve issues. This is especially true if the code under test or someone who did not write the test code sees the problem.

Existing test cases





Sample test case

In `org/jsoup/parser/parsesSimpleDocument.java`

```
    @Test
    public void parsesSimpleDocument() {
        String html = "<html><head><title>First!</title></head><body><p>First
post! <img src=\"foo.png\" /></p></body></html>";
        Document doc = Jsoup.parse(html);
        // need a better way to verify these:
        Element p = doc.body().child(0);
        assertEquals("p", p.tagName());
        Element img = p.child(0);
        assertEquals("foo.png", img.attr("src"));
        assertEquals("img", img.tagName());
    }
```

This method mainly test the parser. This function parse HTML(String) to the **Document** object. **Document** object contains attributes that can be used to analysis the **Elements** in the HTML(String). This `parsesSimpleDocument.java` test method mainly test whether the result from parse method is actually match the real data using `assertEquals`. In other word, to test the validation of function parse.

In `org/jsoup/nodes/testTitles.java`

```
    @Test
    public void testTitles() {
        Document noTitle = Jsoup.parse("<p>Hello</p>");
```

```
        Document withTitle = Jsoup.parse("<title>First</title>
<title>Ignore</title><p>Hello</p>");

        assertEquals("", noTitle.title());
        noTitle.title("Hello");
        assertEquals("Hello", noTitle.title());
        assertEquals("Hello", noTitle.select("title").first().text());

        assertEquals("First", withTitle.title());
        withTitle.title("Hello");
        assertEquals("Hello", withTitle.title());
        assertEquals("Hello", withTitle.select("title").first().text());

        Document normaliseTitle = Jsoup.parse("<title>   Hello\nthere   \n   now
  \n");
        assertEquals("Hello there now", normaliseTitle.title());
    }
```

This method mainly test the validation of **Document** object and its attribute. There are several assertion test in this test unit, including `noTitle` and `withTitle`. They tested the **title** attributes and validate the accuracy of the method parse.

In `org/jsoup/parser/TagTest.java`

```
  @Test
    public void isCaseSensitive() {
        Tag p1 = Tag.valueOf("P");
        Tag p2 = Tag.valueOf("p");
        assertNotEquals(p1, p2);
    }

  @MultiLocaleTest
    public void canBeInsensitive(Locale locale) {
        Locale.setDefault(locale);

        Tag script1 = Tag.valueOf("script", ParseSettings.htmlDefault);
        Tag script2 = Tag.valueOf("SCRIPT", ParseSettings.htmlDefault);
        assertSame(script1, script2);
    }
@Test
    public void equality() {
        Tag p1 = Tag.valueOf("p");
        Tag p2 = Tag.valueOf("p");
        assertEquals(p1, p2);
        assertSame(p1, p2);
    }
```

This Test class contains many unit test to validate the **Tag** and test the functionality of tag attributes in the **Document** object. The above code mainly test the case sensitive, equal and the Insensitive of different **Tag**.

Partitioning

Introduction to systematic functional testing and partition testing

Systematic functional testing is usually used in functional testing, also called black box testing method. It is a non-uniform method that try to select inputs that are especially valuable. Usually, systematic functional testing choose representatives of classes that are apt to fail often or not at all.

Partition testing is similar to the concept of partition in data structure - divide and conquer. In debug and testing field, partition testing means that the input space is divided into classes that merge into the entire space. These classes might overlap.

It is also similar to Heine-Borel theorem in mathematics in my perspective, which is to cover all of the cases using pieces of subcases. That is the second statement of this theorem: S is compact, that is, every open cover of S has a finite subcover. These subcover might overlap too. Back in my bachelors' study, my professor Pang, Xuechen gave an example - In the fall, leaves fall and they cover the ground from your dorm to campus. Leaves are finite and they do cover all the road. That's Heine-Borel theorem.

Importance of these testing methods

As mentioned in existing test cases in set up environment part, we described the importance and advantages of systematic function testing. Usually, it try to select inputs that are especially valuable so that it can test the functions equally. The reason for that is simple and clear.

- Using non-uniform method, it can deal with the sparse problem in the input space. A classic example is Java class "root" mentioned in the class.
- Using this kind of systematic testing can find bugs and remove them from hay more effectively.

For partition testing, the input space is divided into classes that merge into the entire space. It also has advantages and great importance.

- It can sample each class. And in one quasi-partition, if there is bug in one function, at least one class will reveal the fault and it can clearly lead to where the bug is. The reason why there is at least one class can will reveal the fault is that classes might overlap, and more than one class tests with the same function.
- Each fault is in dense space in some class of inputs.

The step of our testing is divided into four steps.

1. Decompose the specification into equivalence partitions
2. Select representatives
3. Form test specifications
4. Produce and execute actual tests

JSoup Partitioning Test case

In `JSoup` project, we have XML parser and HTML parser, but in test cases, `JSoup` only has HTML parser testing method. Therefore, we decided to choose XML parser and test the functions in XML parser.

Our partitions and boundaries

In this project, we select XML parse as our partition feature.

In test case, we try to figure out whether `Jsoup` can process the correct xml string as well as incorrect xml string using `assertEquals` function.

For boundaries, we talked about Leap Year in class, and its boundaries are range of years and certain set of years, like 2000. For our case, we don't have ranges of xml. Then we figure out that we can use the **length** of each xml file as ranges. Also, we try boundaries case like xml with large tag name case.

Write new test cases in JUnit

Test whether `Jsoup` could get elements from the tag we assign:

```
    @Test
    public void parsesDocumentSize() {
        Document doc = Jsoup.parse(xml);
        Elements name = doc.getElementsByTag( "name" );
        assertEquals(2, name.size());
    }
```

Test whether `Jsoup` could get element from the elements array:

```
    @Test
    public void parsesSimpleDocumentElement() {
        Document doc = Jsoup.parse(xml);
        Elements name = doc.getElementsByTag( "name" );
        Element element = name.get( 0 );
        String text = element.text();
        assertEquals("tom", text);
    }
```

Test whether `Jsoup` could return correct answer when we input a wrong tag:

```
    @Test
    public void parsesNullExistDocumentElement() {
        Document doc = Jsoup.parse(xml);
        Elements id = doc.getElementsByTag( "id" );
        assertEquals(0, id.size());
    }
```

Test whether `Jsoup` could get xml's attribution:

```
@Test
public void parsesSimpleDocumentElementAttr() {
    Document doc = Jsoup.parse(xml);
    Elements name=doc.getElementsByTag("student");
    Element element = name.get( 0 );
    //System.out.println(element.text());
    String studentNumber=element.attr("number");
    //System.out.println(studentNumber);
    assertEquals("0001", studentNumber);
}
```

Test whether a very large tag name can be parse by `Jsoup` :

```
@Test
public void handleSuperLargeTagNames() {
    // unlikely, but valid. so who knows.

    StringBuilder sb = new StringBuilder(maxBufferLen);
    do {
        sb.append("LargeTagName");
    } while (sb.length() < maxBufferLen);
    String tag = sb.toString();
    String xml = "<" + tag + ">One</" + tag + ">";
    Document doc =
Parser.xmlParser().settings(ParseSettings.preserveCase).parseInput(xml, "");
    Elements els = doc.select(tag);
    assertEquals(1, els.size());
    Element el = els.first();
    assertNotNull(el);
    assertEquals("One", el.text());
    assertEquals(tag, el.tagName());
}
```

Team Members

Sun Yu(http://github.com/duke326/)

Lai Wang(https://github.com/laiwang2020/)

Xinyi Hu(https://github.com/samaritanhu)

Reference

https://www.vogella.com/tutorials/JUnit/article.html