

Introduction

What is the tool that you are testing?

We are testing the tool called jsoup. Our Github link is <https://github.com/duke326/SWE261>.

What is its purpose?

jsoup is a Java library for working with real-world HTML/XML.

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.

See jsoup.org for downloads and the full [API documentation](#).

Any other aspects that are relevant

Lines of code: 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
```

Number of java files: 132

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

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

Language: 100% of Java

Set Up

Fork & Add

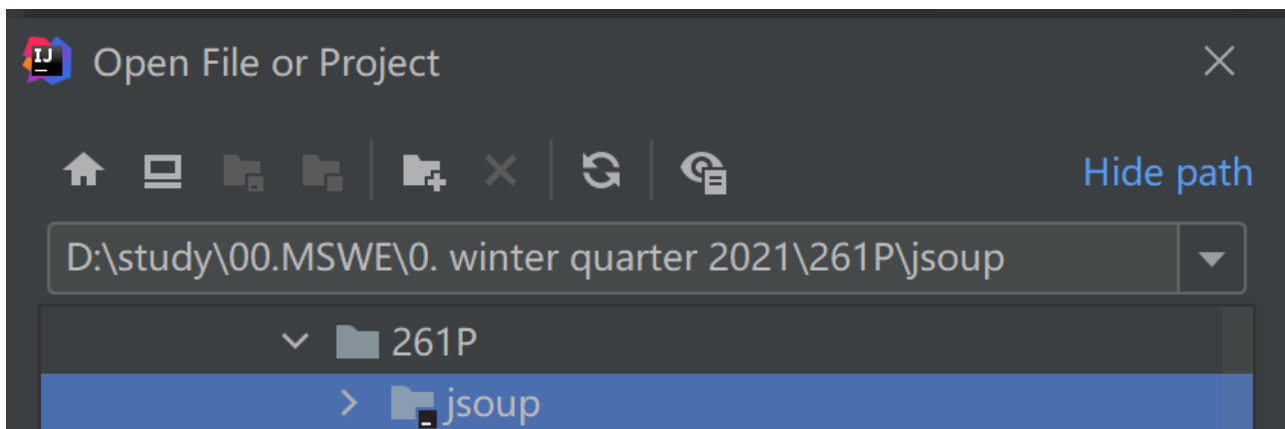
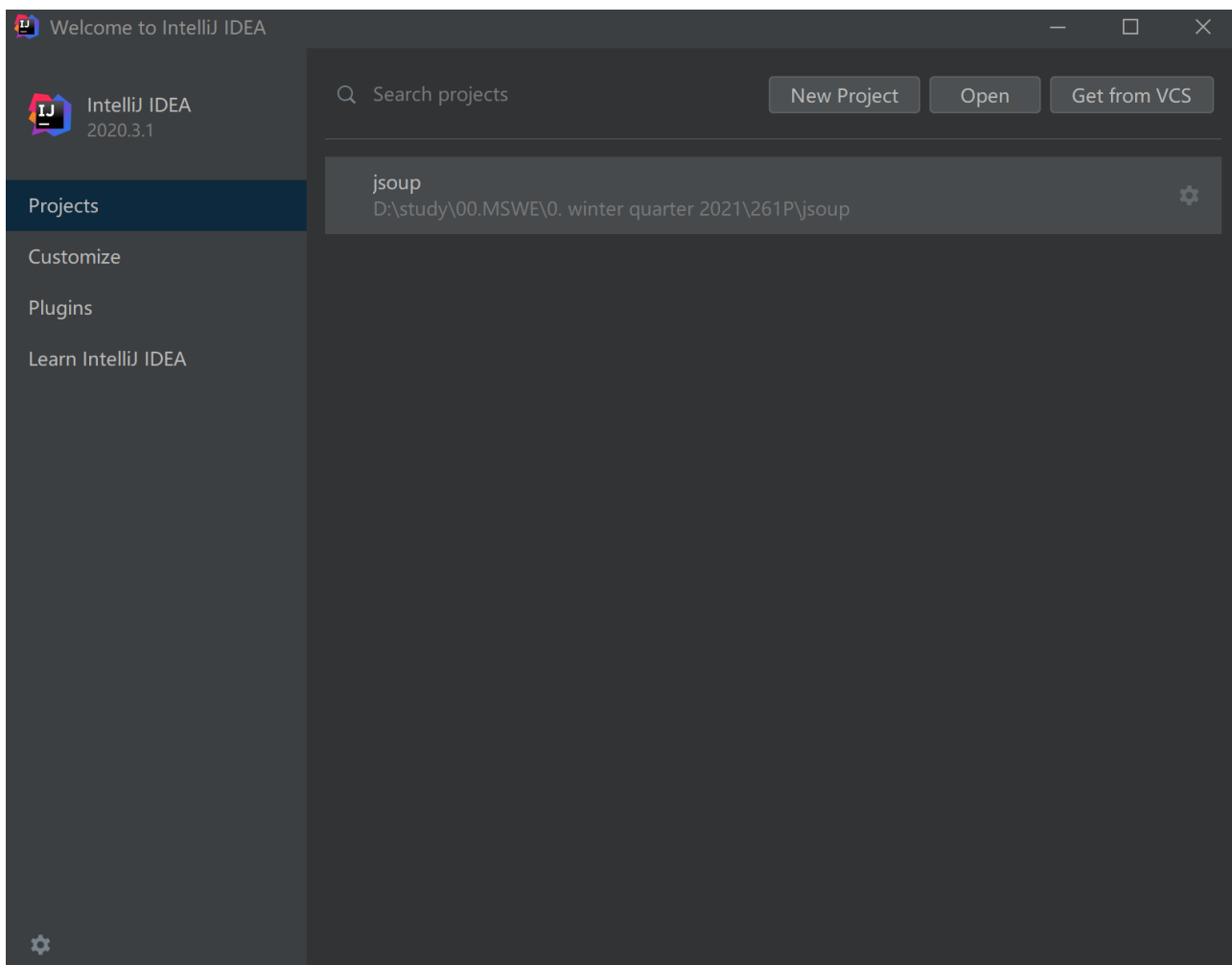
Our link is here: <https://github.com/duke326/SWE261>. We have all of our team members, Lai Wang, Sun Yu and Xinyi Hu in the repo. Also we have included Prof. Jones and TA Maruf.

Build

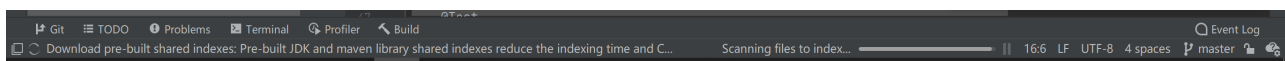
This project uses Maven as the build infrastructure for Java projects. Maven uses rules and patterns to provide an integrated build system. Every Maven project uses a set of shared plugins retrieved from the Maven repository, and when the project is built, Maven performs a set of predefined tasks throughout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

Using IntelliJ 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, you can run any test case in the test folder.

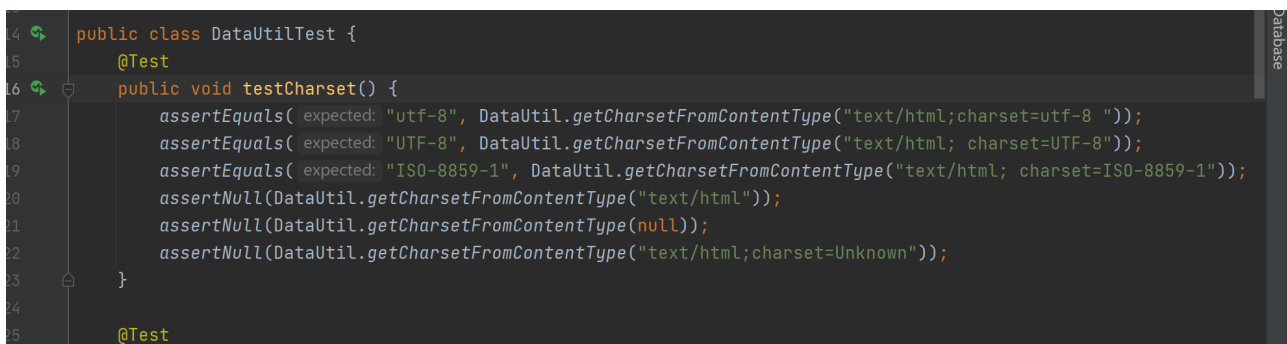
Setup for using JUnit 5

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 need 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.

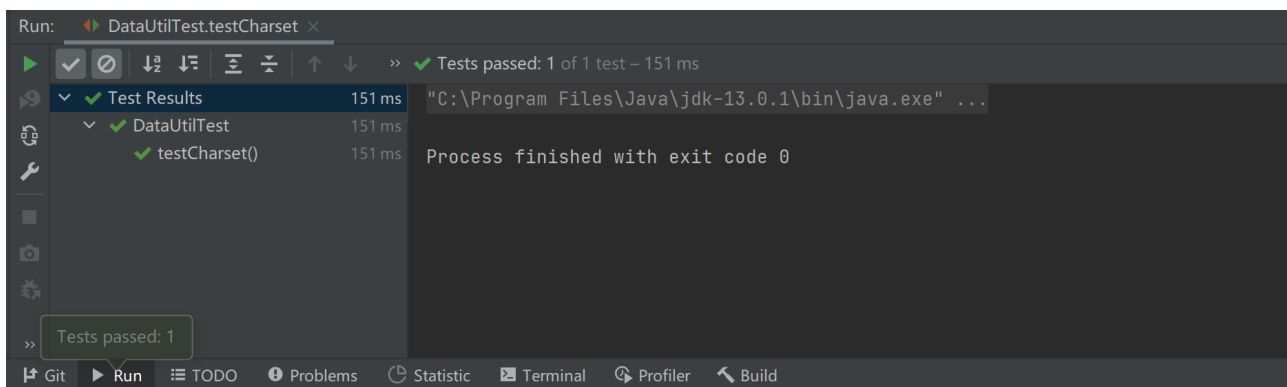
Sample to run

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



```
14 public class DataUtilTest {
15     @Test
16     public void testCharSet() {
17         assertEquals( expected: "utf-8", DataUtil.getCharsetFromContentType("text/html;charset=utf-8 "));
18         assertEquals( expected: "UTF-8", DataUtil.getCharsetFromContentType("text/html; charset=UTF-8"));
19         assertEquals( expected: "ISO-8859-1", DataUtil.getCharsetFromContentType("text/html; charset=ISO-8859-1"));
20         assertNull(DataUtil.getCharsetFromContentType("text/html"));
21         assertNull(DataUtil.getCharsetFromContentType(null));
22         assertNull(DataUtil.getCharsetFromContentType("text/html;charset=Unknown"));
23     }
24
25     @Test
```

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



```
Run: DataUtilTest.testCharSet
>> Tests passed: 1 of 1 test - 151 ms
Test Results
  DataUtilTest
    testCharSet()
    Tests passed: 1
"C:\Program Files\Java\jdk-13.0.1\bin\java.exe" ...
Process finished with exit code 0
```

Test cases

Firstly, the existing test case is **black box testing**. 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.

Black box testing is also known as functional testing. Black box testing derive sets of inputs that will fully exercise all of the functional requirements of a system.

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.

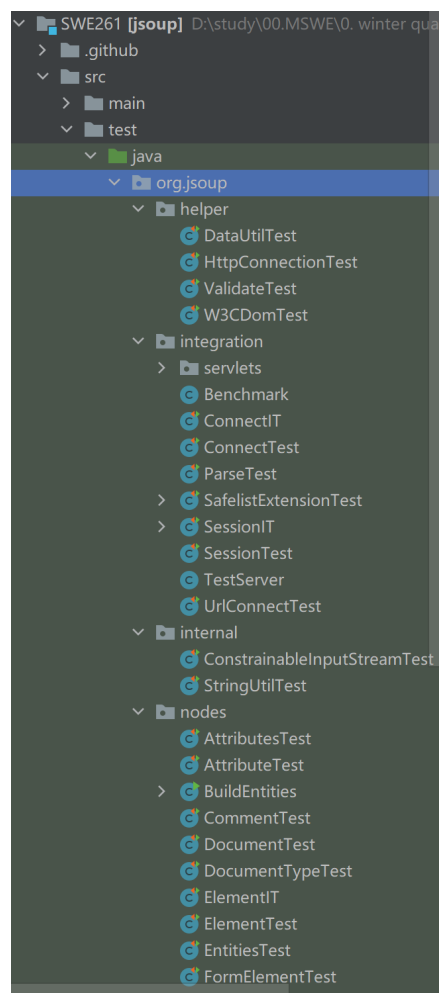
Introduction into JUnit

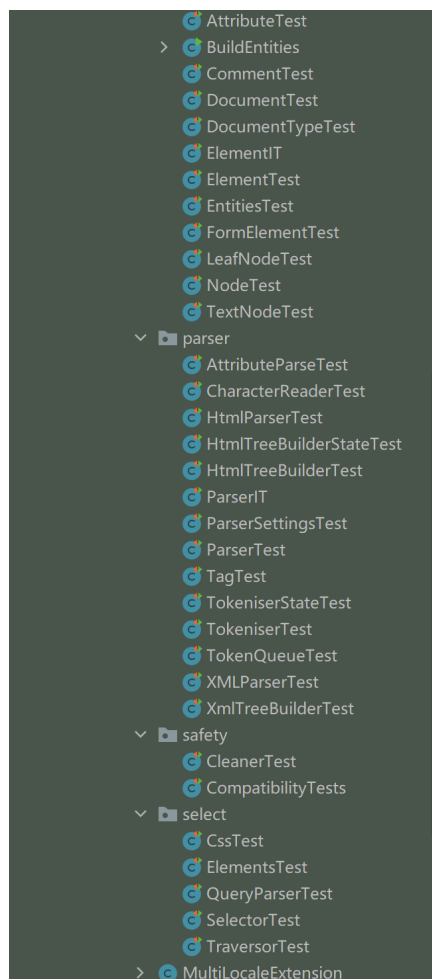
JUnit *test* is a method contained in a class that is only used for testing. This is called *test category*. To define a specific method as a test method, please annotate the method with `@Test`.

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

In class, we talked about six general engineering principles. One of them is **partition**, which means that divide and conquer.

The need for systematic functional testing and partition testing.

In our class, “Functional testing” usually implies systematic testing. We use functional testing instead of random or uniform testing is **to find needles and remove them from hay**. In detail, failures are rare in the space of possible inputs, but are densely populated in a certain small space. If we systematically test some cases from each part, we will include the dense parts.

Describe these concepts.

Functional testing: Deriving test cases from program specifications.

Systematic testing: Try to choose particularly valuable input. Usually, by selecting class representatives who tend to fail often or not at all.

Partition testing: The input space is divided into classes that merge into the entire space- "equivalent partitions".

A feature that allows for partitioning.

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

Specify your partitions (and boundaries when appropriate) in English — describe them.

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>