Project 3: Sparkzilla
Due: 5:00 PM, Apr 12, 2019

Contents

1 Introduction ................................................................. 1

2 The Assignment .......................................................... 2

3 URLs .............................................................................. 4

4 Parsing ........................................................................... 5
   4.1 HTML ...................................................................... 5
   4.2 Parser Subpackage ................................................... 7

5 HTTP .............................................................................. 8
   5.1 The Request ............................................................ 8
      5.1.1 The First Line (Path to Retrieve) ...................... 9
      5.1.2 HTTP Headers ............................................... 9
      5.1.3 Form Data ..................................................... 10
   5.2 The Response .......................................................... 11
      5.2.1 The Status Line .............................................. 11
      5.2.2 Response Headers ....................................... 11
   5.3 A Complete Example ........................................... 12

6 Sockets ........................................................................ 13

7 Demos ........................................................................... 13

8 Handin .......................................................................... 14
   8.1 Design Check .......................................................... 14
   8.2 Final Handin ............................................................ 14
   8.3 Testing Requirements ........................................... 15
   8.4 Grading ................................................................. 15
1 Introduction

For this project and the next one, you’re going to be writing a set of programs that perform a pretty good approximation of what happens on the World Wide Web. You’ll also write a few programs that use the Web infrastructure you create to do some interesting things.

The Web is built on a client-server architecture. The client programs (i.e., applications) run on your computer, and connect to another program, called a server, that is usually running on another (possibly far-away) computer. Over the connection, your client submits a request to the server, and the server responds with data.

Over the connection, the client submits a request to the server, and the server responds with data.

In the case of the Web, the client is often called a browser. Popular browsers include Firefox, Safari, Internet Explorer, and Google Chrome. At your request, a browser submits a request for a page. The request contains a few pieces of information (details below). The server then sends back a document written in a simple language, and your browser displays it on your screen. For this project, your job is to write a browser that connects to a server that we provide for you.

The details of how two computers talk to each other are, unfortunately, rather complicated. Luckily, Java (like many other languages) provides a simple abstraction for all of that, called sockets. Even though you have to complete this project using Scala, you’ll be using Java sockets.

The pages that your browser will display will be written in a very small subset of HTML (the language in which web pages are written). The subset supports just enough features to make things interesting, but not enough to make processing it boring.

The protocol that your client and server will communicate over will be a subset of the standard web protocol, HTTP. Like HTML, the complete version of HTTP is quite complex, but the subset we are requiring you to implement is not, and it’s sufficient for our purposes. It allows the client to submit requests for pages, and to submit data that a user fills into a form on a page.

Your task for this project is to write a text-based browser. The browser will be very simple, and not very practical to use, but it will be enough to show off the capabilities of your system. You will then do a follow-up assignment (GUIzilla) in which you reuse much of your code to build a GUI-based browser. Since the two projects are connected, you will work with the same partner for both projects.

Glossary

- **Hypertext** is structured text with hyperlinks within the text.
- **Hyperlinks** are references within a document to data that reside elsewhere.
- **HTML**: HyperText Markup Language — the language in which web pages are written.
- **HTTP**: HyperText Transfer Protocol — the standard protocol for web communication.
- **GUI**: Graphical User Interface — an interface that uses graphical icons instead of text.
- **URL**: Uniform Resource Locator — an address for resources on the Internet consisting of a protocol (e.g., http and a resource name).

---

1. The computer itself is also called a server.

HTTPS: Hypertext Transfer Protocol Secure — a protocol for secure web communication.

XHTML: Extensible Hypertext Markup Language — an extension of HTML.

2 The Assignment

A browser’s job is to display a page of HTML and await user input. In response to said input, the browser might send a request to a server, get a new page from the server, and display that page. Since the browser for this project is text-based, showing a page consists of writing a string to a BufferedWriter (which should print to the console).

Roadmap This handout has a separate section on each of these parts: sending requests is in Section 5.1, the format of server responses is in Section 5.2, and information about the page structure that you need to render is in Section 4.1. The remaining sections contain details about the contents of requests and responses, and some project mechanics. Further down in this section, you’ll find a sample user interaction; Section 5.3 shows an example of requests and responses. Taken together, these examples illustrate what you’ll be aiming to implement for this project (we also provide a demo version (Section 7) that you can experiment with to understand the project).

A Sample Interaction When your browser starts, it should display some predefined HTML page hard-coded into your browser, like an empty page or one that welcomes the user (you design your start page). Once a page has been displayed, the user can select from among many different actions: quit, enter the URL of a desired page, go back to the previous page viewed by the browser, click on a link, fill in a text field, or click on a button to submit the data in a collection of such fields, etc..

In a traditional, GUI-based browser, the user can simply point and click with the mouse. You can replicate this behavior by implementing the following: after printing out the page, the browser can request that the user enter an integer between 1 and some \( n \) to select an action. Use ‘1’ for back, ‘2’ for enter URL, and ‘3’ for quit; the remaining integers are for selecting actions that are part of the page being displayed: clicking on a link, filling in a text field in a form, or submitting the data in a form. On any other input, all of which are invalid, print an error message, and re-render the previous page.

Here’s a sample interaction between a browser and a user.

Parsing page...
Rendering page...
---
Welcome to Sparkzilla!
---
Action (1) Back, (2) New URL, (3) Quit: 2
Enter URL: http://idaho/Test/page1
Connecting to server 'idaho'...
Requesting page '/Test/page1'...
Server returned status OK.
Parsing page...
Rendering page...
---
This is a silly page for testing. <This>[4] is a link.

Moony was here!

Here's a form:

Favorite color: _________[5]
Favorite food: _________[6]
[Submit][7]
---
Action (1) Back, (2) New URL, (3) Quit, (4-7) Select Page Element: 5
Enter value for field "color": cyan
Action (1) Back, (2) New URL, (3) Quit, (4-7) Select Page Element: 6
Enter value for field "food": eggplant
Action (1) Back, (2) New URL, (3) Quit, (4-7) Select Page Element: 7
Submitting form to URL `http://idaho/id:6adf96775832d5/formssubmit'...
Connecting to server `idaho'...
Requesting page `/id:6adf96775832d5/formssubmit'...
Server returned status OK.
Parsing page...
Rendering page...
---
Thanks for submitting that form. You said that your favorite color is cyan and that you like eggplant.

<Try the second test page>[4]
---
Action (1) Back, (2) New URL, (3) Quit, (4-4) Select Page Element: 4
Following link to URL `http://idaho/id:6adf96775832d5/page2'...
Connecting to server `idaho'...
Requesting page `/id:6adf96775832d5/page2'...
Server returned status OK.
Parsing page...
Rendering page...
---
Welcome to the second test page.
This page is a dead end; it has no active elements.
---
Action (1) Back, (2) New URL, (3) Quit: 3
Goodbye!

To follow a link, and view the ensuing page, the browser should:
1. Connect to the specified host and submit a request for the page.

2. Download the page from the server.

3. Transform the page into a list of HTMLElements. (Note: We provide this class and its subclasses for you; more information following later.)

4. Render the page by iterating through the list of HTMLElements and rendering each element individually.

Rendering is a fancy term for “print to the console.” You should identify special elements, like links and forms, in your pages somehow, but otherwise, you can render a page however you like.

Active elements means “stuff that can be clicked on” (except you won’t be doing any clicking in your text-based browser). These are: links, text fields in forms, and submit buttons.

As your program renders a page, it should keep track of all the active elements. For each, it should create an object that is capable of carrying out the associated action; e.g., asking the user to input the text for a specific text field and storing the input.

Hint: You’ll probably find it helpful to design your program so that your active elements all implement some trait. That way, the main browser can simply invoke a method on the chosen one.

There are two main ways to handle when a user instructs the browser to go back. You can either store a list of visited URLs, find the last one, and re-fetch the page, or you can cache the pages themselves. The former can be problematic if the last URL involved form inputs. In particular, if a user fills in text fields in a form, then navigates away from the page, and then uses the back button to go return to the form, their inputs should still be there. As a result, we’d like you to use the caching method. Caching a page means maintaining a local copy of the actual content of the page, so that you can just render it again without having to fetch it from the server again.

If the server returns an error for a particular request, you should handle it by printing the error message returned by the server and then prompting the user with appropriate actions (e.g., quit). If the server returns a page successfully, but the page can’t be displayed (e.g., due to a parsing error), you should again handle it similarly, but not identically—for example, print a different error message.

3 URLs

Before your browser can submit a request to the server for a web page, it must process the so-called URL of that page, which uniquely identifies that page’s address. A URL consists of a few parts. The first part, a few characters followed by a colon, specifies the protocol to be used. HTTP is the only protocol your browser need support. HTTP is identified by http: (other protocols used on the real Web include FTP, HTTPS, and a long time ago, Gopher).

After the protocol comes two slashes followed by a hostname, which identifies a computer on the network. The client will use the hostname to figure out which host to connect its socket to. The rest of the URL (after the hostname) is called a path. There’s no specified format for a path, but here’s how ours will look: a slash followed by either a class name or an object id, then another slash, then a method id. The last slash and method id is not necessary. If the first part is an object id, its first
three characters will be \texttt{id:}, followed by the actual object id. If it’s a class name, it’s just the full class name (which obviously shouldn’t start with the characters \texttt{id:}).

You’ll learn about \textbf{object ids} in the next project. For this project, you don’t have to worry about the contents of the path; you only need to be able to split a URL into its hostname and its path, and then the server will return the page corresponding to the path.

Here are some examples of URLs:

- \url{http://host/class/id:18}
- \url{http://myserver/id:111/dothis}
- \url{http://yourserver/myclass}

4 Parsing

The language that our pages are written in is a simplified version of HTML. While we leave out many parts of HTML (including tables and images), we preserve all of the features that are essential for basic dynamic Web applications, like forms.

Something a bit unique about this project is we are providing both \texttt{src} and \texttt{sol} code– be sure to download both. We are providing certain files as \texttt{sol} code because we allow you to– dare we say, \textsl{encourage} you to– modify these files.

To make things easier for you, we’ve provided a method \texttt{getHTMLElementList} in \textit{Browser.scala} that takes as input \texttt{BufferedReader}, and returns \texttt{List[HTMLElement]}.

\textbf{Note:} Please move all files in \texttt{/course/cs0180/sol/sparkzilla/sol} and \texttt{/course/cs0180/src/sparkzilla/src} into your Sparkzilla \texttt{sol} and \texttt{src} directories respectively.

Below is an in-depth description of how the HTML is formatted for different elements (e.g. a paragraph or a link). You should read through this to understand how our parser works, which will help you later to convert each \texttt{HTMLElement} back to the rendered output.

4.1 HTML

HTML is based on the XML markup language, which you already have some familiarity with from Search. There are six HTML tags which you will have to handle in this project.

- \texttt{html}: Every HTML page consists of one \texttt{html} element, which consists of one \texttt{body} element (and nothing else).
- \texttt{body}: The page body. This can contain \texttt{form} and \texttt{p} elements.
- \texttt{p}: Stands for “paragraph”. This can contain text and \texttt{a} elements.
- \texttt{a}: Denotes a link. (The “a” is for “anchor”.) This can contain text.
- \texttt{form}: A form. This can contain \texttt{p} and \texttt{input} elements.
- \texttt{input}: An input field or a submit button. This doesn’t contain anything.
Outermost Tags

Every HTML page has the form:

```html
<html>
  <body>
    ...
  </body>
</html>
```

The body tag contains the body of the document, meaning its text, any links, etc.

Paragraphs and Links

A paragraph has both opening and closing tags. Inside, it can have links, as well as any text you want.

Links have both opening and closing tags. The opening tag must have a value for the `href` attribute. The value of `href` is the URL that selecting that link will send you to. The text between the opening and closing tags is the text that will describe the link.

Here's an example of a paragraph with a link in it:

```html
<p>
  This paragraph has a link in it.
  <a href="http://idaho/Test/homePage">This text describes the link.</a>
</p>
```

As a special case, if the URL for a link starts with a single slash, it's a relative link, and the client should use the same host that the current page was retrieved from for that link. For example, the relative URL `/Test/homeAwayFromHome` would translate to the full URL `http://idaho/Test/homeAwayFromHome` if the page came from the server named `idaho`.

Here's an example of a paragraph with a relative link in it:

```html
<p>
  This paragraph has a relative link in it.
  <a href="/Test/homeAwayFromHome">This text describes the link.</a>
</p>
```

Note the `Paragraph` class has as its member variable a `List[HTMLElement]`. This is comprised of `Link` and `PageText` elements, both of which have fields corresponding to the data they store as described above.
Forms and Inputs

A form is a set of fields where the user can enter data to be submitted to the server. In a form, you can display some text, some text input boxes, and a button to send the inputs provided to the server.

A form has both opening and closing tags. The opening tag must have the method attribute with the value post. It must also have an action attribute. The value of action is the URL where the browser should send the data when a form is submitted. The value of method describes how the form data will be submitted. Forms can support many kinds of methods, but you will only be implementing post.

There are two types of inputs. The first is designated by the attribute type="text". This corresponds to a field for text entry. This type of input must have a value for the name attribute. The name is not printed, but is used to identify the field when communicating with the server.

The second type is designated by the attribute type="submit". This corresponds to a button which submits the form. This type of input must also have a value attribute with the value submit. Inputs are self-closing.

Here’s an example of a form with inputs:

```html
<form method="post" action="http://idaho/Test/formSubmit">
  <p>Favorite food:</p>
  <input type="text" name="food" />
  <input type="submit" value="submit" />
</form>
```

This example would correspond to a Form containing in its HTMLElement list the following elements: Paragraph, TextInput, and SubmitInput.

A page can contain multiple forms, each with different inputs and target URLs.

4.2 Parser Subpackage

Generally, in software, we try to isolate particular pieces of functionality as much as possible. This is a concept referred to as encapsulation. By focusing on small portions of the problem at hand, you are less at risk of code that is buggy, hard to maintain, or confusing. A good example of a place you can isolate functionality is parsing. The problem of parsing an HTML page can be entirely separated from the problem of implementing a web browser. Therefore, we have placed all of our parsing code in a parser subpackage in the src directory.

One thing to note about Java/Scala’s package structure is that it mimics the code’s file structure. You should already have a parser subfolder in your usual source folder (i.e., src/sparkzilla/src/parser).

When you need to import a class from the parser elsewhere, use:

```java
import sparkzilla.src.parser.Class
```
If you are using Eclipse, there is nothing else to be done. However, if you would like to run through the terminal, you will need to use a modified command from the one you might expect. The usual compilation command is:

```bash
scalac -d bin */<assignment>/*/*.scala
```

This command compiles all files that have the path `<some-folder>/<assignment>/<some-folder>/<some-name>.scala`. However, we now need to compile files with the path `src/sparkzilla/src/parser/<some-name>.scala`. To handle this, we can add another argument to our command:

```bash
scalac -d bin */sparkzilla/*/*.scala */sparkzilla/*/*/*.scala
```

Running the actual program is the same as before.

**Note:** If there is a parser subpackage in your sol folder, you may delete this. Your parser package should reside in src.

5 HTTP

The protocol that your client and server should speak and understand is a subset of HTTP, the real World Wide Web's protocol. HTTP normally communicates over TCP port 80, but we will use TCP port 8080 to avoid conflicts with real web servers.

5.1 The Request

A request sent from the client to the server consists of four parts, separated by "\n", in order:

1. A one-line request indicating the path to retrieve (Section 5.1.1).
2. A series of additional lines, each one representing a single HTTP header (Section 5.1.2).
3. A blank line.
4. Form data, if a form was submitted (Section 5.1.3)

5.1.1 The First Line (Path to Retrieve)

The first line of a HTTP request consists of a method, a space, the path, a space, and the HTTP version.

The method is either GET or POST, because there are two different kinds of requests that can be made on the web. In response to clicking a link or going to a new URL, the browser is merely requesting information, or getting it. Not surprisingly, GET is used for this kind of request. In response to filling out a form and clicking the submit button, the browser is sending information, or posting it. Not surprisingly, POST is used in this case.
The path is just the path from the URL that the user requested, and the HTTP version, for our purposes, is always HTTP/1.0. The request line should be terminated by a carriage return and a newline character (\r\n).

Here are some examples of HTTP request lines:

- GET / HTTP/1.0\r\n
- GET /Survey/cookieForm HTTP/1.0\r\n
- POST /Survey/formSubmit HTTP/1.0\r\n
### 5.1.2 HTTP Headers

An HTTP header is a single line consisting of a header name (or key), followed by a colon and a space, followed by the header value (name: value). You must append a carriage return and a newline character to the end of each header value.

HTTP headers are used to convey auxiliary information to the server related to the request. There are a lot of possible headers, but there are only a couple that you need to send:

- Connection: close\r\n
- User-Agent: Sparkzilla/1.0\r\n
The Connection header tells the server to close the connection at the end of the response, and the second line tells the server what type of browser is being used.

If it is a POST request (i.e. because a form is being submitted), there are two additional headers that must be sent:

- Content-Type: application/x-www-form-urlencoded\r\n
- Content-Length: 38\r\n
Here, “38” is the length of the form data (explained below).

When you are done sending headers, you must send an additional blank line ("\r\n")

### 5.1.3 Form Data

When a form is being submitted, or posted, the information entered into the forms needs to be sent to the server. If no form is being submitted, this part of the request is not applicable.

To generate the form data, each form input is represented as its name followed by an equal sign, followed by the text the user entered. The form data for each input gets joined together and delimited by ampersands (‘&’) to generate the data for the entire form.

For example, the form data would be
food=Sushi&icecream=Vanilla&drink=Coke

if a user entered “Sushi”, “Vanilla”, and “Coke” into the following form:

<form method="post" action="http://idaho/Test/formSubmit">
  <p>Favorite food:</p>
  <input type="text" name="food" />
  <p>Favorite ice cream flavor:</p>
  <input type="text" name="icecream" />
  <p>Favorite drink:</p>
  <input type="text" name="drink" />
  <input type="submit" value="submit" />
</form>

The length of this form data is 38 (count the characters, if you’d like); that’s why the Content-Length header above is 38.

Note: If a form has multiple inputs, they can be given in any order within the request.

Note: The proper formatting for an empty input to a field is to just leave the response blank but still send the input field. So, if the user had not entered “Vanilla”, the form data would have been food=Sushi&icecream=&drink=Coke. Your browser should allow users to submit forms without filling out one or more of the input fields.

Encoding

There is a small problem with the system as currently designed. What if the user had entered an ampersand in one of the form inputs? For example, if someone’s favorite ice cream flavor was “Cookies & Cream” the form data would be:

food=Sushi&icecream=Cookies & Cream&drink=Coke

A server that is processing this form data will think that the ampersand in “Cookies & Cream” is separating two different form inputs. It will assign the value “Cookies ” to the “icecream” input, and think that “Cream” is an error because there’s no equal sign. Therefore, any ampersand or other special character appearing in a form input must be encoded into an unambiguous form. Use the Java method URLEncoder.encode\(^2\) to encode both form names and form values.

Example:

URLEncoder.encode("Cookies & Cream", "UTF-8")
=> "Cookies+%26+Cream"

The first argument is the string to encode, and the second argument should always be "UTF-8"\(^3\). After encoding all form names and values, the form data becomes unambiguous:

food=Sushi&icecream=Cookies+%26+Cream&drink=Coke

---

\(^2\)Yes, there is a corresponding procedure URLDecoder.decode.

\(^3\)You should not use the deprecated single-argument version of encode.
Note that the Content-Length is calculated after all encoding occurs, so the Content-Length of this form data is 48.

5.2 The Response

When the client is done sending the request, it closes its side of the connection. The server then processes the request, generates a page, and sends back its response. The response follows the same form whether the request was GET or POST. Like a request, a HTTP response consists of 4 parts, separated by "\n", in order:

1. One line containing a status code (Section 5.2.1).
2. A series of lines, each one representing a single HTTP header (Section 5.2.2).
3. A blank line.
4. The HTML document (Section 5.2.2).

5.2.1 The Status Line

The first line of the response consists of the server’s protocol (either ‘HTTP/1.0’ or ‘HTTP/1.1’), a space, a status number, a space, and a status message. If everything went well, the status number will be 200. If something went wrong, this number will be something else depending on the error. For example, if the page wasn’t found on the server, the status code will be 404. However, regardless of the status code, the server will still send a page, so you can technically ignore the status line.

Example:

HTTP/1.0 200 OK

5.2.2 Response Headers

Response headers follow the same format as request headers (name: value). A HTTP server may send any number of headers, but for our simple browser, you can ignore them all. After the last header the server sends a blank line.

The HTML Document

Following the blank line, the server will send the HTML page.

5.3 A Complete Example

Here’s an example of a client requesting a simple page:

The client sends this request string (Section 5.1):
GET /Test/survey HTTP/1.0\r\nConnection: close\r\nUser-Agent: Sparkzilla/1.0\r\n\r
and then closes its connection. The server responds with this string (Section 5.2)

HTTP/1.0 200 OK
Server: Sparkserver/1.0
Connection: close
Content-Type: text/html

<html>
<body>
<p>
This is a test page.
<a href="/Test/page2">This</a> is a link.
</p>
<p>Here is a form:</p>
<form method="post" action="/id:6adf96775832d5/formsubmit">
<p>Favorite food:</p>
<input type="text" name="food" />
<p>Favorite ice cream flavor:</p>
<input type="text" name="icecream" />
<p>Favorite drink:</p>
<input type="text" name="drink" />
<input type="submit" value="submit" />
</form>
</body>
</html>

Once the user fills out the form and submits it, the following request is sent (Section 5.1 again):

POST /id:6adf96775832d5/formsubmit HTTP/1.0\r\nConnection: close\r\nUser-Agent: Sparkzilla/1.0\r\nContent-Type: application/x-www-form-urlencoded\r\nContent-Length: 48\r\n\r
food=Sushi&icecream=Cookies+%26+Cream&drink=Coke

And the server responds as usual.

6 Sockets

Although you will be implementing your browser in Scala, you'll be using a Java abstraction, namely sockets, to do so. Sockets provide a simple interface to the TCP protocol, which is a protocol
designed to provide reliable stream-oriented communication between two computers. TCP has a notion of a port, which is a number used to distinguish between connections. The server will set up a socket on port 8080 where it can listen for connections. Connections can support both input streams and output streams. Your client should initiate a connection with the server, (i.e., open a socket to port 8080); when the server receives such a request, it can accept the connection. Now, the client can transmit data on its output stream, close the output stream, and then wait for the server to transmit data back to it on its input stream. Once the server receives the transmitted data, it can process it and send back an appropriate reply. Upon receipt of that reply, your client should close its input stream. And then finally, your client should close the socket.

The Java classes you will most likely need are:

- `java.net.Socket`
- `java.io.InputStream`
- `java.io.OutputStream`
- `java.io.BufferedReader`
- `java.io.InputStreamReader`
- `java.io.OutputStreamWriter`

Lab 10 is designed to familiarize you with sockets. If you are eager to learn more about them sooner, please refer to the Java documentation pages.

7 Demos

The demos (and the server) will be accessible after the HTAs return from spring break

You can run the TAs’ version of Sparkzilla by typing `sparkzilla_demo` into a shell. You should use the demo to answer as many functionality questions as you can.

While logged in to a department machine, you can access a demo server at the following URLs (all servers serve the same content):

- `http://rna/Index`
- `http://thufir/Index`

Your browser should follow the specifications spelled out in this document, with your user’s interactions roughly mimicking the demo.

---

4Coming soon to a lab near you!
8 Handin

8.1 Design Check

Design checks will be held on April 4-5. We will send out an email detailing how to sign up for design checks; it is in your best interest to sign up as soon as possible (before all the prime time slots have been filled).

Reminder: You are required to pair program at least the design check portion of all CS 18 projects. We recommend finding a partner as soon as possible, as you will not be able to sign up for a design check without one.

For the design check, you must do the following:

1. Bring a handwritten class diagram of the structure of each HTMLElement. Be able to explain which HTML elements can be contained in others and why these classes are set up the way they are (i.e., what each field in each class means).

2. Have a general strategy for how you will print a page using the provided HTMLElement case classes.

3. After a user requests an action (e.g., visit a new URL, go back, or submit a form), your browser will have to perform the requested action, print the result, and wait for the next input. Describe generally (in words or in pseudocode) how you will implement this basic browser functionality.

As always, you’re welcome to come to the design check with questions, but we encourage you to also use the design check time to talk with your grader about aspects of the project beyond those which are required for the design check.

8.2 Final Handin

The final handin is due by April 15. For the final handin, your scalaproject should contain the packages sparkzilla.src (in the src directory) and sparkzilla.sol (in the sol directory). Your code should be part of the sparkzilla.sol package. That package should also contain a README.txt file.

Your ‘README’ file should include:

- instructions for use, describing how a user would interact with your program
- a brief overview of your design, including how all of the pieces of your program fit together
- a description of any features you failed to implement, as well as any extra features you chose to implement
- a description of any known bugs in your program
- a description of how you tested your program
- a list of the people with whom you collaborated
To hand in your files, navigate to the `~/course/cs018/workspace/scalaproject` directory, and run the command `cs018_handin sparkleza`. This will automatically hand in the contents of your entire `scalaproject` directory. Once you have handed in your project, you should receive an email, more or less immediately, confirming that fact. If you don’t receive this email, try handing in again, or ask the TAs what went wrong.

**Note:** Only one of you or your partner must hand in the project.

### 8.3 Testing Requirements

In terms of grading, we will look mainly at your system tests. However, you will end up writing some methods that produce essential components of your browser. Writing unit tests for these parts is simply good practice, and will likely help you in debugging your work. So while we won’t be grading your unit tests for thoroughness, we will be looking at whether you identified reasonable parts of your implementation that could be unit tested. For these parts, write at least one unit test on a not-entirely-trivial input, so we can see what testing decisions you might make in a professional-grade setting.

**What About Error Handling?** Mostly, we want you to handle errors such that your browser won’t crash. If a page returned by the server fails to parse, recover and let the user enter another input after printing some reasonable error message (we don’t care about the exact contents, but there should be different messages for different problems).

### 8.4 Grading

As with all CS 17/18 projects, a good design will make coding this project significantly easier; so you should spend a fair amount of time working on your program’s design before you begin writing any code.

The design check counts for 15% of your grade, including:

- A diagram and explanation of the **HTMLElement** class hierarchy
- A plan for rendering a webpage
- A description of the flow of control of your browser: i.e., how it will respond to the various user actions

Functionality counts for 70% of your grade, including:

- Networking: 13 points
- Rendering web pages: 13 points
- User interaction: 13 points
- Form handling: 16 points
- Back button: 15 points
As always, partial functionality merits partial credit.

The final 15 points will be reserved for comments, testing, and style. You should include documentation comments and test cases for all non-trivial methods. You should also perform system testing, to test interactions among methods. Additionally, comment any code which would otherwise be unclear.

Note: Comprehensive testing of a browser can be particularly tricky. In order to test your renderer, you can write the output of your browser to a file.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: https://cs.brown.edu/courses/cs018/feedback