Contents

1 Introduction
   1.1 Support Code .................................................. 1
   1.2 Server .......................................................... 1
   1.3 A Simple Dynamic Web Server ................................. 2
   1.4 Keeping State ............................................... 3
   1.5 Come back soon! ........................................... 7

2 The Server Specification ........................................ 8

3 Testing the Server .................................................. 10

4 Technical Notes ....................................................... 11
   4.1 Cloning ....................................................... 11
   4.2 Reflection .................................................... 12

5 Search Integration .................................................. 14

6 GUIzilla
   6.1 Requirements .................................................. 15
   6.2 FXML .......................................................... 15
   6.3 Suggested Design ........................................... 16
   6.4 A Few of Things to Consider ................................ 16
   6.5 Evaluation .................................................... 16
   6.6 Packages ..................................................... 16

7 Internet Policy ....................................................... 17

8 Demos ................................................................. 17

9 Testing
   9.1 GUI .......................................................... 17
   9.2 Server ......................................................... 18
1 Introduction

By now, you have written a sleek textual web client that lets you browse all the spiffy HTML pages you would ever want to. You may, however, have some lingering questions, such as “What the heck was happening on the other end?”, “What were those weird ID numbers?”, and “How did those pages get my form input?”. In this project, you’re going to write the server, and all will be made clear.

You also might be asking yourself, “Who wants to use a text-based web browser?” The answer, nowadays, is “no one” and so for the second part of this two-part project, you’ll be writing a graphical user interface (GUI) for your browser.

1.1 Support Code

We’ve provided you with support code for your src and sol directories. In sol, you’ll find three subpackages: client, pages, server. In client you’ll find two scala files:

1. RunApplication.scala need not be modified, this will simply give you a place to run your program from.

2. GUIBrowser.scala this class contains important information you’ll need for rendering your page elements on the GUI. There are a few TODOs, including extending your Browser class copied over from sparkzilla, handling actions from the back and submit button and implementing or making a call to render.

You might find it useful to transfer files from sparkzilla.sol into your guizilla project! Instead of referring to your sparkzilla files, copy them into your guizilla project and make the modifications as necessary.

1.2 Server

What does a web server do? Back in 1991, when the very first web pages were simple HTML documents that sat on a disk somewhere, the server didn’t have a very interesting job: a client might have said “Hey, I’d like a file”, and the server would have then sent the contents of that file to the client. As time went on, pages got more interesting. They started to include images, background MIDIs, in-lined Java applets, and so on. Yet, the job of the server was still just to send the picture/song/program to the client, who would then show/play/run it. Pages like these are referred to as static content because the content is simply files on the disk that the server packages up and sends over the network to the client.

1The earliest web browsers, however, were text-based!
The designers of the web were smart, though, and they realized that static content could only go so far. You’ve probably at some point in your life visited a web site that asked you to enter your name in a field (say, “Ash Ketchum”) and then on the next page said something really original like “Hey Ash Ketchum. Welcome to our web site!”. How do they do that?

Obviously, they don’t have a different page sitting on disk for each possible name you could type in. The solution is hopefully a familiar idea to you: since we can’t a priori list all of the pages that we might need to send, we’ll write a program that makes the ones we want to send when we want to send them. Instead of thinking of a page as a static file on the server, you can think of the page as a program that you run to generate the web page that you see. Pages that are generated by programs are called dynamic content.

When a server hosts dynamic content, its job becomes more complicated: it takes the client’s request, runs the program named in the request, and then sends the program’s output back to the client. Since programs without arguments are not very interesting, the server is also responsible for passing the arguments (like, Ash Ketchum) from the client to the program.

The server you will write will be responsible for dynamic content. You’ll be able to handle static content as well, of course. It’s easy to write a program that “generates” a static page; it just sends the client a constant string regardless of input. We’ll take a look at our implementation of a dynamic server now.

1.3 A Simple Dynamic Web Server

The client is responsible for sending the server the name of a program to invoke, and the arguments to give that program; the server then runs the program and sends its output back to the client.

Recall from Sparkzilla that HTTP requests look like this:

```
POST /Forum/dopost HTTP/1.1
Connection: close
User-Agent: Sparkzilla/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 32

name=Sparkdino&text=Hello+World!
```

You already know that the last line represents the form data entered by the user. These are arguments to the program. But what about the path in the first line? We were intentionally vague about what exactly the path referred to, but now all is revealed: the path is the name of the program to be invoked.

As advertised, our dynamic pages are going to be program-generated. A set of related pages will be implemented in one class; the part of the path between the first slash and the second is the name of this class (Forum). Each individual page will then be implemented by a method; the part of the path after the second slash is the name of this method (dopost). So, the path above (/Forum/dopost) tells the server to run the dopost method of an instance of the Forum class.

What arguments should the dopost method take? It at least needs to receive all of the form inputs that the client sent.
And what should it return? It needs to return the text of an HTML page. The server will then send the page back to the client.

Thus, the simple web server waits for a client’s request, invokes the method named in the path with the arguments specified by the form data, and then sends the page it generates back to the client.

Other things we didn’t mention before about requests:

- “GET” and “POST” must be capitalized
- Header key-value pairs (e.g., “Connection: close”) are case-insensitive
- Headers can come in any order
- The order of the key-value pairs in the form data is also arbitrary

1.4 Keeping State

In the standard web model, a program generating a page is stateless: the client asks the server to run it and generate a page, and once it has sent back the generated page, the program that generated the page terminates. That is, each page request by the client produces a separate invocation of the page-generating method. Being stateless is good for scalability: the server doesn’t need to keep track of any connections other than the ones it is currently processing. However, it makes certain kinds of dynamic web pages cumbersome to write.

Consider a web program to add two numbers: on the first page, you enter a number; the result of this is a second page where you enter another number; the third page displays the result of these two numbers.

This page is tricky in a stateless model because by the time the server gets the second number, it has already forgotten about the first one. The client sends the first number to add, and the server might use this to generate a page that says “The first number you want to add is 18. Enter the second number:”. However, the only input field in this second page is the one corresponding to the second number. Without state, the program trying to calculate the sum has no way of knowing that the first number was 18.

One possible solution to this problem is for the method that generates the page requesting the second number to write the first number to a file, and then have the method that is sent the second number read it back in. Another would be that the program generating the second page somehow includes the first number in the page so that it gets sent back as an input a second time, unchanged. These solutions are not very desirable: they both require whoever is creating each individual page to go through contortions to keep track of all past inputs. Wouldn’t it be nice if we could somehow make this easier?

We can, and we will. The key observation is this: if we have one program/class on the server that generates a set of related pages, then it can use its state to keep track of past inputs. Essentially, you want to be able to pause the program between generating different pages, and resume it again later.

Specifically, in our model, we have a set of related pages implemented by one class and an individual page implemented by one of that class’ methods. In this way, we can store inputs that are supposed to be shared between different individual pages (methods) in the class representing those pages.

---

2This actually happens on the web through hidden input fields.
For example, an `AddTwo` class can store the first number in a variable, so then when it gets the second number, it can easily recall its value. Now, rather than writing to disk or sending the data back and forth across the network, someone writing a page can keep track of the data in instance variables. Here’s what it might look like (with comments removed for brevity):

```scala
class AddTwo extends Page {

  private var num1 = 0

  override def defaultHandler(inputs: Map[String, String]): String =
    addTwoNumbers(inputs)

  def addTwoNumbers(inputs: Map[String, String]): String =
    "<html><body><p>Add Two Numbers</p>"
    +
    "<form method="post" action="/AddTwo/secondNumber">"
    +
    "<p> Please enter the first number you would like to add: </p>"
    +
    "<input type="text" name="num1" />" +
    "<input type="submit" value="submit" />" +
    "</form></body></html>"

  def secondNumber(inputs: Map[String, String]): String =
    inputs.get("num1")
      match {
      case Some(num) =>
        try {
          num1 = num.toInt
          "<html><body><p>The first number entered was " + num1 + "</p>"
          +
          "<form method="post" action="/AddTwo/displayResult">"
          +
          "<p> Please enter the second number you would like to add: </p>"
          +
          "<input type="text" name="num2" />" +
          "<input type="submit" value="submit" />" +
          "</form></body></html>"
        }
        catch {
          case _: NumberFormatException =>
            "<html><body><p>I'm sorry you did not input a valid number.</p>"
            +
            "<a href="/Index">Return to the Index</a></p></body></html>"
        }
      case None =>
        "<html><body><p>I'm sorry, there was an error retrieving your input.</p>"
        +
        "<a href="/Index">Return to the Index</a></p></body></html>"
    }

  def displayResult(inputs: Map[String, String]): String =
    inputs.get("num2")
      match {
      case Some(num) =>
        try {
          val num2 = num.toInt
          "<html><body><p>" + num1 + " + " + num2 + " = " + (num1 + num2) + "</p>"
          +
          "</body></html>"
        }
        catch {
          case _: NumberFormatException =>
            "<html><body><p>I'm sorry you did not input a valid number.</p>"
            +
            "<a href="/Index">Return to the Index</a></p></body></html>"
        }
    }
}
```
As you can see, the page generated by `addTwoNumbers` has a form that submits its data to the `secondNumber` method of the `AddTwo` class. This method records the first number, and then generates a page with a form that submits its data to the `displayResult` method. This method records the second number, and then uses the two saved values to compute the sum. After generating a page, each of these individual methods “pauses” the computation, which is resumed when the client submits another request.

This all sounds very reasonable; however, there is a (small) problem with the code for generating these pages. The form in the page generated by `secondNumber` says it invokes `/AddTwo/displayResult`; yet, if `displayResult` is supposed to know what the first number was, then it is very important that this request invokes `displayResult` on the instance of `AddTwo` that stored the user’s first number. Thus, we need a way for the client to request a specific instance of `AddTwo`.

To deal with this issue, we’ll create **session IDs** that we’ll use to keep track of the different instances of each class. The client will then be able to say “I’d like to call `displayResult` on the same instance of `AddTwo` that I called `secondNumber` on”, and thus remember everything they had entered up to that point (in this case, the value they entered for the first number). To implement this idea, we’ll add a prefix ‘id’ to each path, and then the server should be able to call the requested method on the instance associated with that session ID.

Why should it be the server’s responsibility to differentiate between instances? Well, it was the server that generated the instances in the first place. That is, when the client requests a page simply by class name (a page that does not require any previous state, like `/AddTwo/addTwoNumbers`), the server should create a new instance of the page-generating class.

One approach to differentiating between instances is to associate with each instance a unique ID, that is, to maintain a **session map**. The server is then responsible for passing this ID to the page-generating method. To implement this, we’ll add another argument to all page-generating methods: a `String` that represents the unique ID of an instance. By supplying this session ID as the target of a form, an instance of the page-generating class can build a form that, when submitted, invokes a method on that same page-generating instance.

The revised `AddTwo` would look like this:

```scala
class AddTwo extends Page {

  private var num1 = 0

  override def defaultHandler(inputs: Map[String, String], sessionId: String): String =
    addTwoNumbers(inputs, sessionId)
}
```

---

3Theoretically, you get into trouble if the client is hacking around with the HTML pages you’re serving, because then they could try requesting random ID/page combinations from the server to see if they can resume someone else’s computation. Making IDs longer and more random will decrease their likelihood of success.
def addTwoNumbers(inputs: Map[String, String], sessionId: String): String =
"<html><body><p>Add Two Numbers</p>" +
"<form method="post" action="/id:" + sessionId + "/secondNumber">"
+ "\n<p> Please enter the first number you would like to add: </p>" +
"<input type="text" name="num1" />" +
"<input type="submit" value="submit" />" +
"</form></body></html>"

def secondNumber(inputs: Map[String, String], sessionId: String): String =
inputs.get("num1") match {
  case Some(num) =>
    try {
      num1 = num.toInt
      "<html><body><p>The first number entered was " + num1 + "</p>" +
      "<form method="post" action="/id:" + sessionId + "/displayResult">"
      + "\n<br>Please enter the second number you would like to add: </br>" +
      "<input type="text" name="num2" />" +
      "<input type="submit" value="submit" />" +
      "</form></body></html>"
    } catch { 
      case _: NumberFormatException =>
        "<html><body><p>I'm sorry you did not input a valid number." +
        "<a href="/Index">Return to the Index</a></p></body></html>"
  }
  case None =>
    "<html><body><p>I'm sorry, there was an error retrieving your input."
    +
    "<a href="/Index">Return to the Index</a></p></body></html>"
}

def displayResult(inputs: Map[String, String], sessionId: String): String =
inputs.get("num2") match {
  case Some(num) =>
    try {
      val num2 = num.toInt
      "<html><body><p>" + num1 + " + " + num2 + " = " + (num1 + num2) + "</p>" +
      "</body></html>"
    } catch {
      case _: NumberFormatException =>
        "<html><body><p>I'm sorry you did not input a valid number." +
        "<a href="/Index">Return to the Index</a></p></body></html>"
    }
  case None =>
    "<html><body><p>I'm sorry, there was an error retrieving your input."
    +
    "<a href="/Index">Return to the Index</a></p></body></html>"
Now we have a server design that easily supports state-sharing among different related pages\(^4\).

### 1.5 Come back soon!

Consider the following scenario:

You modify `AddTwo` so that the “enter the second number” page has a link to go back and reenter the first one. You then try doing the following:

1. Browse to the first page, and enter 22 as the first number.
2. On the resulting page (“You entered 22. Enter the second number: ”), you follow the link to reenter the first number, and then change it to 33.
3. After loading the resulting page (“You entered 33. Enter the second number: ”), you use your browser’s back button to go back twice, trying to get back to the “You entered 22. Enter the second number: ” page.

What do you see, instead? That it says, “You entered 33”\(^5\)?

What went wrong? In our earlier discussion, we decided that a session ID mapped to a specific instance of a class. Because all the requests in the above step refer to the same instance, the server would have assigned the same ID to each, which means that as we progressed through the steps, we mutated the instance being referenced by that ID. While the first number was “22”, we changed it to “33”, so when we go back to the “You just entered X...” page that used to say “22”, it now says “33”. This is wrong. (If you don’t believe so from this example, then consider the parallel situation where you are booking airplane reservations, and you look at one flight, look at another, then go back to the first and hit “reserve!” only to get tickets to the second. You can see the same problem with bookmarks, as well as the back button.)

How can you solve this problem? Well, you have to make sure the same path always takes you not only to the same instance, but to the same instance *in the same state!* You can implement this by cloning (See Section 4.1) the state of the page-generating object before proceeding, and then manipulating the server’s session map and the IDs handed to page-generating methods appropriately. Essentially, you have different IDs for the same instance in different states.

### 2 The Server Specification

The preceding section\(^6\) was meant to provide intuition about why you’re implementing the server that you are. We described some of the problems that people have encountered since the beginning of the internet, as well as ways to go about solving them. Hopefully you will now have a better appreciation for what we expect you to implement. In short:

\(^4\)One limitation of our design is that pages that want to share state must be implemented as methods of the same class.

\(^5\)Assuming you’re not doing any browser-side caching. If you are, then the page will say “22”, but when you choose a second number (say, 10), the server will come back with “43”, not “32”, as you might have expected.

\(^6\)or diatribe, novella, whatever you want to call it
You will be implementing stateful dynamically-generated web pages, where a group of pages that share
state is implemented as a single class and an individual page is implemented as a single method.

The core of the server is a listener that will listen for incoming connections to a socket on port 8080.
When it gets an incoming connection, it should first read the whole HTTP request from the client,
and then process it. Ultimately, the request will boil down to a path and possibly name-value pairs
comprising form data. The listener will then pass this request on to a dispatcher.

The dispatcher then needs to access the object capable of generating the page requested by the
client. If the client requested a page from an already-existing object, the server should look up
the object ID in its session map, then clone that object (see Section 4.1 on cloning). If the client
included a class name instead of an ID, the dispatcher should create a new instance of that class
(see section 4.2). In either case, the server will generate a new ID for the new instance of the class,
and add this association to its internal session map.

Note: You are not responsible for removing objects from the server's session map.

The dispatcher should then extract the method name from the request and invoke the method
(again using reflection) on the instance, passing it the form data sent by the client. If the client
does not specify a method name in the path (i.e., it has no second slash, or has nothing after the
second slash), then the server should invoke a default method of that page. (Indeed, all pages should
have a default method!) In either case, the result of this method call will be a page that should be
sent back to the client over the open connection (prefixed with the HTTP status line and response
headers, as usual). Then the connection should be closed, and the server can go back to listening
for incoming connections.

At all steps, your server is responsible for gracefully handling any errors and reporting useful error
messages back to the client. Your server should run an infinite-loop, and it should not crash.

Finally, you are required to write a set of pages that show off the functionality of your server. Be
creative! The web pages that you write will be worth a significant chunk of your grade. We will be
looking for form handling and dynamic content, both of which you will need to keep track of in
your program's state.

Page Requirements

To demonstrate the functionality of your server, as well as your understanding of stateful dynamically-
generated web pages, you will be required to write your own web pages; that is, to create your
own content. Start out by writing at least one simple page just to ensure you understand how
page-generation works in our server model. Once you've got that under your belt, you should create
a page that makes use of stateful content in an interesting way. If you like, you may augment the
simple page you wrote to include such content. Be creative! We look forward to seeing what kinds
of pages you can create.

---

7The methods for pages can, of course, call helper methods.
8See the “Technical Notes” section below for details on processing an HTTP request.
Error Handling

If the request is successful (e.g., if the server sends back the requested page), the first line of the response (the status line) should say 'HTTP/1.0 200 OK'.

If not, there are three kinds of errors you must handle gracefully:

- If the page is not found, the status line should be 'HTTP/1.0 404 Not Found'.
- If a bad request was received, the status line should be 'HTTP/1.0 400 Bad Request'.
- If an exception was thrown during the processing of a page, the status line should be 'HTTP/1.0 500 Internal Server Error'.

In the case of an error, you should still send back a full response, including the headers and a simple HTML error page telling the user what went wrong.

Page Interface

Your session map will be a map from session IDs to page objects (Map[Int, Page]). Each Page will implement various methods for generating various pages besides the required methods, however, we can’t directly call those methods without knowing what type of Page it is. We will solve this problem using reflection (See Section 4.2). What is important right now is that each method that implements a page must match this signature:

```scala
def myMethod(inputs: Map[String, String], sessionId: String): String
```

This method will correspond to the path /MyClass/myMethod on the server (where MyClass is the name of the class that defines myMethod, i.e., if you implement a page in some method and class of different names, the path that corresponds to changes accordingly). The inputs parameter is a map from strings representing keys (form field names) to strings representing values (form field responses).

Additionally, every class representing a collection of pages should extend the Page class we gave you.

See Section 4.1 for an explanation of the strange Cloneable thing.

The defaultHandler method of the Page class generates the default page (which is invoked if no method name is specified).

3 Testing the Server

After implementing an HTTP browser, you should have a rough idea of how to implement an HTTP server. However, there are a few additional things you will need to know to effectively process an HTTP request. Following these instructions will allow you to use your server with not just Sparkzilla, but ordinary HTTP browsers like Firefox, Chrome, Safari, etc....
The server should be prepared to accept both ‘HTTP/1.0’ and ‘HTTP/1.1’ in the first line of the request. While Sparkzilla will always send ‘HTTP/1.0’, modern browsers will use ‘HTTP/1.1’.

Web browsers like to send a lot of request headers. However, there is only one that you need to care about, and only when handling a POST request: Content-Length. Save this value so you know how much form data to read (see below). Remember that headers are terminated by a blank line.

If the request is a POST request, use the value of the Content-Length header to know how much form data to read. You must use this value. Form data does not have to end with a newline, and there may be a newline in middle of the form data, so using Content-Length is the only way to know when to stop reading. Also, modern web browsers may not close their side of the connection after sending a request, so you can’t simply read until there’s no more to read.

When you are done reading the request, close the server’s end of the socket.

Recall from Sparkzilla that the names and values from the form data have been encoded. Be sure to decode them with the URLDecoder.decode procedure.

```java
URLDecoder.decode("I+ride+the+bus!", "UTF-8")
=> "I ride the bus!"
```

```java
URLDecoder.decode("Plusle+%26+Minun", "UTF-8")
=> "Plusle & Minun"
```

The only response header you absolutely need is ‘Content-Type: text/html’. Modern browsers, which understand many different kinds of documents (including images, plain text files, PDFS, etc.) need to know what kind of document is coming back. Also remember to include a blank line after the last header.

You should test your server using both Sparkzilla and Guizilla. However, since the server uses the standard web technologies HTTP and HTML, you should be able to request pages from your server using Chrome or another standard web browser. Not only will this be a good test of your server, you can also use it to show off the power of your creation. Since your server will be listening on port 8080, if you want to test it using a modern web browser, you need to explicitly specify the port number in the URL by adding :8080 to the end of the hostname like this:

```
http://idaho:8080/
```

4 Technical Notes

This section provides additional notes on various technical topics this project encompasses.
4.1 Cloning

In order to support going back properly, you will have to make a copy of each page object before requesting a new page. Cloning in Scala is usually done the Java way, so we will actually just use Java’s implementation of cloning.

Java provides an easy way to clone—that is, make a copy of an object: the clone method, which is inherited from java.lang.Object. There are a few subtle points, though, that you need to be aware of when using clone.

The least confusing way to support cloning is to write your own clone method (overriding the inherited one) that creates a new instance of the class and duplicates all of its fields. If you had to write one of those for every class you wanted to clone, it might get really tedious, so Java provides an alternative. The clone method of Object does all that work for you: it creates a new instance of whichever class you invoke it on and automatically copies all of the fields into the new one. Any class that inherits from Object will have this clone method, but if you try to use it, you might be surprised: instead of making a clone, you’ll get a CloneNotSupportedException. What gives?!

The designers of Java wanted programmers to be able to control whether their classes were clonable, and chose to make everything not clonable by default. So if you want your class to be clonable, you have to indicate as much by declaring your class as implementing the (mis-spelled) Cloneable interface. This is an empty interface (there are no methods declared in it)! Its only purpose is to indicate to anything that wants to call the clone method that “yes, it’s OK to clone me.”

The reason Java’s designers made everything not clonable by default has to do with the properties of references. Recall that in this piece of code, listA and listB refer to the same LinkedList object, and changes made to it through listA can be seen through listB. That is, the code below will print true!

```java
LinkedList<String> listA = new LinkedList<String>();
listA.append("References are weird!");
LinkedList<String> listB = listA;
listB.removeFirst();
System.out.println(listA.isEmpty());
```

Cloning an object with Object.clone is like creating a new instance of that object’s class, then assigning all of its fields to the same values as the original object’s, even if they are references. So if an object has a reference to a LinkedList, and you clone it, both the old and the new objects will refer to the same LinkedList. While sometimes you might want that behavior, almost all of the time it’s going to be the wrong thing to do, which is why they made you declare something extra if you want it.

To get around this problem, you’re going to need to write your own clone method, which is what we were trying to avoid in the first place. Don’t worry, though: you can still take advantage of Object.clone to do some of the work for you. The idea is to use Object.clone to create the new instance, and then to manually clone any mutable objects that your object refers to. Actually, it’s a good idea to use super.clone, which may or may not be Object, to clone its own fields properly. Your clone method will then look something like this:

```java
class Thingy extends Cloneable {
    private var myInt = 0
}
```
The default behavior of Object.clone is a shallow copy. References are copied, but not the things that they refer to. Your goal is to perform a deep copy, that is, to make copies all the way down the tree of references until you hit primitives or immutable objects. As long as every object implements clone properly, you can do this by cloning every object you refer to.

One final note: cloning a MutableList, or any of Scala’s container classes, will create a fully independent object, but any objects contained in that list will not be cloned themselves. This is fine if those objects are immutable objects, but sometimes you need to make sure that the old and new objects don’t share any references. Unfortunately, you have to write code to do this yourself; Java provides no shortcut for this case. Consider this as yet another incentive to make as many classes as you can immutable!

In summary, to make a class clonable, you should:

- declare that it extends Cloneable
- write a public clone method that calls super.clone
- manually clone all fields that refer to mutable objects

If you’re sure that your class has only immutable fields, then you can skip writing your own clone method.

### 4.2 Reflection

We kept saying that the client would be sending the server the name of a class and method to invoke, and that your server should call that method. It’s actually quite nice that Java allows you to do this, if you think about it: you’re saying “I have the name of a class and a method; could you make an instance of that class for me and then call this method on it?” This functionality is not something you could implement using the Java or Scala syntax and primitives you know.

The way to do accomplish this feat is through reflection. Reflection allows a running Java program to gain knowledge about classes, methods, and fields, even if the program didn’t know about those classes when it was compiled. Without reflection, you would have to hard-code the page objects and page-generating methods explicitly into your server, which would make for a whole lot of tedious dispatching code.

---

And would be a potential security hole. Can you think of why?
There is a built-in Java class, called `java.lang.Class`, which is used to represent another class. You can’t create new instances of `Class` directly (with a constructor), but you can through several other methods. One method allows you to get a `Class` object for some class given only its name, as a `String`. To do this, you use the static function `forName` of the `Class` class, which takes one argument, a `String`, and returns a `Class` object if it finds a class by that name. If not, it’ll throw a `ClassNotFoundException` (which you’ll be expected to handle gracefully). If you do get an object, though, you can call `newInstance` to instantiate the class using its default constructor (using any other constructor involves more work).

Here is an example of how to use reflection. Assume we have elsewhere defined a class called `Concatenator` with a method named `concatenate`, which takes as input two `String`s and concatenates them together. Here we present the code for a method that uses reflection to create an instance of the `Concatenator` class and invoke the `concatenate` method. Note that, for conciseness, exception handling has been left out of this example. However, you should be sure to handle gracefully all of the myriad exceptions that could arise when using reflection. In particular, reflection requires casting, as seen below. Although we generally try to avoid casting as much as possible, this is another instance in which it is necessary. Additionally, your code need not conform exactly to this example. Feel free to write or structure your code in the way which you think is best. This example is by no means the inarguably “best” way to use reflection. Now, for the code:

```scala
def reflectionExampleMethod: String = {

  // obtain the class object
  val concatenatorClass = Class.forName("examplePackage.Concatenator")

  // use the newly obtained class object to instantiate an Concatenator object
  val concatenator = concatenatorClass.newInstance.asInstanceOf[Concatenator]

  // get the method that we will be calling by calling getMethod on a Class object
  val concatenateMethod = concatenator.getClass.getMethod("concatenate", classOf[String], classOf[String])

  // call invoke to use the method which was just obtained and cast to a String, then return
  concatenateMethod.invoke(concatenator, "String1", "String2")
    .asInstanceOf[String]
}
```

Once you have an instance of a class, you probably want to call some method on it. This requires two steps: you first have to locate the method, with the `getMethod` method of `Class`, then you have to invoke it, with the `invoke` method. To get the method, you have to provide a method name, and followed by classes representing all the arguments’ types. The easiest way to get a class representing an argument type (say, `String`) is to call the method `classOf[String]`. The `getMethod` then returns an object of type `java.lang.reflect.Method`, but note that it will only find a method if it is declared to be `public`. You can use that object to invoke the method, by calling the `invoke` method and passing it the object it should be invoked on and its actual parameters. The return value of `invoke` will be the return value of the method. Wow! That was a mouthful. Feel free to refer the Java API for more information about how the `Class` class works.
5 Search Integration

In addition to the above requirement for a set of self-created web pages, you will be required to write a page that augments the functionality of the Search project you completed earlier in the semester. We have set up a Query server that is running the TA solution for Search. All you have to do is write a page to connect with it! Using what you know about client-side sockets, you will need to make a page that is able to connect to port 8081 on the host eckert (Inside the department only).

Just as you have been using HTTP for your server, the Query server you connect with here has a protocol that must be followed as well (although it’s significantly less complicated). After retrieving your user’s query, you should set up a socket inside your Search page to the given host and port. The protocol goes as follows: you will write

REQUEST\t<query>\n
where <query> is the user’s query. Due to the nature of the protocol, the user’s query cannot contain a tab. The Query server will respond with

RESPONSE\t1 <res1>\t2 <res2>\t...\t10 <res10>\n
where <res1> through <res10> are the top 10 results. Note that there may be fewer than 10 results. It will be up to you to figure out how to interpret the response and display it to the user in a way that maximizes the page’s usability.

But what fun is it to just see the titles of the top ten pages? With dynamic web pages, we can do so much better! Another part of your web page will include the functionality to actually view the content of one of those top ten pages! Again using client-side sockets, connect to our Page server running on port 8082 on the host eckert (Inside the department only).

Instead of sending a query, you will send to this Page server the title of a page in the wiki, and receive the text content of the page itself. You may find it helpful to store the results returned by the Query server to ensure the user selects a valid page. Make sure you store the exact title of each page, as the Page server will be very sensitive to the String you write! After the user selects a title, write the title to the Page server, followed by a newline. You should then read as many lines as the server sends back to you, as this will be the text of the article itself. We will release a demo of this functionality after all design checks.

To be clear, this part of the project falls under the Server component of the assignment. In addition to the other web pages you are expected to write, you will be writing a web page, to be served by your Server, that allows the user to search our wiki. You are not expected to add any form of a search feature to your browser (described below).

6 GUIzilla

At this point, you’re well on your way to creating a spiffy text-based browser and a state-of-the-art web server. But let’s face it: when you want to check the CS 18 website, or the Wikipedia page for
your favorite superheroes, you’re not going to use a text-based browser. So, to round out this series of programs, you’ll be writing a GUI-based browser using the JavaFX libraries.

You should start from your Sparkzilla code, and when you have completed all of the GUI work detailed below, you’ll be able to plug the resulting renderer into the rest of your client code. That is, your client will ultimately be able to follow links and submit forms—and look pretty cool while doing it!

6.1 Requirements

You are required to write a program that takes an HTML page, parses it, displays it in a clean graphical manner, and allows for user interaction with the interactive aspects of a browser and the interactive elements of a web page.

Here’s a suggested incremental plan for going about this part of the assignment:

1. Create a GUI browser that works on HTML pages containing no active elements. That is, it renders HTML pages containing only paragraphs.

2. Modify the GUI browser to work on all supported features of HTML pages except forms. That is, it should render paragraphs and links and follow links when clicked.

3. Add the necessary functionality to your GUI browser to handle pages that contain forms. Submitting a form should submit the text in the form’s input fields.

4. Configure the functionality for the user to:
   - manually enter a URL to visit
   - go back
   - quit

6.2 FXML

We are providing the FXML template for you! Be sure to copy this over from /course/cs180/sol/guizilla/sol/client.

Although we are providing the general structure of the GUI, it is still your primary responsibility to render the actual contents of each web page.

6.3 Suggested Design

- Your GUI browser should call a render method on your List[HTMLElement] to display that page. Each element/node in your representation should have a render method that renders itself, and calls render on its children.

---

10 Although not as user-friendly, text-based browsers are extremely useful for purposes such as web crawling and writing scripts to extract information from a webpage.

11 This, of course, you already wrote for Sparkzilla.
• As you render, you will have to set up event handling for the elements a user interacts with (e.g., adding an ActionListener or MouseListener to links). Just as in the text-based browser, rendering active elements will require keeping track of things that you didn’t have to worry about for inactive elements. These event handlers will need information about the browser and, in some cases, about other things in the page.

6.4 A Few of Things to Consider

• The URL bar used for entering URLs should also display the URL of the current page.

• Just as with Sparkzilla, your Back button should make use of a cached version of the pages in history. This includes any modifications made to the page, namely, text inputs that were filled out should still be filled out when you return to the page.

• The URL bar of our Guizilla demo will navigate to the URL when you hit the Enter key. You are not required to implement this functionality.

• Form input fields do not need to handle multi-line inputs.

6.5 Evaluation

For this part of the assignment, you will be evaluated based on two things: (1) your browser’s ability to take a parsed page and successfully render it, and (2) successful user interaction.

Note that you’ll need a working implementation of Sparkzilla (at least the parsing and networking parts) for this assignment. If you didn’t complete Sparkzilla, please email the TAs for assistance so we don’t end up penalizing you twice for the same errors.

The parsing and networking parts of your Sparkzilla solution shouldn’t need to be updated very much, however, you may need to update some auxiliary aspects of those parts of Sparkzilla (namely, the code that ‘rendered’ a web page to a String now needs to render using JavaFX). To do this, it is okay if you need to copy the relevant files from Sparkzilla into your Guizilla package so you can update them. However, if you don’t intend to change anything in some particular class, you should simply import that class. For example, we will not give out another copy of HTMLTokenizer nor Token, so you should import those from your sparkzilla.src package.

6.6 Packages

As we’ve described, there are various parts of this project. Each part will require various classes/files. You may attempt to contain all those files in one guizilla.sol package, however, things will get messy very quickly. To solve this, just like how we used a sparkzilla.src.parser subpackage in Sparkzilla, you may create subpackages for the various parts of this assignment. Some packages that may be useful to create: guizilla.sol.client, guizilla.sol.server, guizilla.sol.server.pages. These packages would then contain the appropriate files. For example, all the classes you write that extend Page would be in guizilla.sol.server.pages.

Using this structure is not a requirement, but it will save you and your grader a lot of headache when you are trying to keep track of which files do which thing. Again, be aware of how you are
specifying the files to compile when you compile from the command line. As always, if you are having trouble with this, or any part of the assignment, feel free to come to TA hours.

**Note:** You are free to modify the sol code we provide to you in any way—this also includes changing the package declarations if you so desire!

7 Internet Policy

As you know, the CS 18 collaboration policy is very particular about what is and what is not an acceptable use of the web when working on course assignments. That said, for this assignment only, we are making an exception. You may search the web for information regarding *JavaFX only*, for this assignment only. For any other topic, and any other assignment, the usual policy applies.

If you are ever in doubt about whether a certain query is acceptable, you can always ask the TAs (or even ask them your question!).

8 Demos

After design checks are completed, you will be able to run the TAs’ version of GUIzilla by typing ‘guizilla-demo’ into a shell on a department machine. Note that the demo may not work over SSH. You can also access our demo servers from Sparkzilla (which serve the Search page at /Search in addition to the previous set of pages) at the following URLs:

- [http://thufir/Index](http://thufir/Index) (Inside the department only)
- [http://stilgar/Index](http://stilgar/Index) (Inside the department only)

9 Testing

You should show us that you have extensively tested your code for both the server and the browser. You will **only** receive credit for testing which is clearly documented in your README file.

9.1 GUI

You should test the GUI by clicking around in the browser and using its various components making sure it properly renders things such as text, and also that it does not throw exceptions. To show us that you thoroughly tested your GUI, you should write up in the README the various tests that you performed including a description of what you expected and an image/video of what actually happened.

9.2 Server

You should test to make sure that your server can correctly receive and respond to requests of any type (GET or POST) with both good and bad inputs. You can test the server however you like.
Feel free to use your GUI/Sparkzilla or the provided demo GUI/Sparkzilla. Similarly to the GUI, we expect you to provide a full explanation in the README of what you are testing, how you tested it, and what the actual and expected results are.

10 Handin

10.1 Design Check

Design checks will be held on April 16-18. Your Sparkzilla grader will remain your grader for GUIzilla and will contact you about scheduling your design check.

Reminder: You are required to pair program at least the design check portion of all CS 18 projects. Your partner for this project will be the same as from Sparkzilla!

For the design check, you must do the following:

For the server:

- Explain what your server has to remember between requests, and what kind of data structures would be appropriate for storing this information.
- Implement a test page class that uses dynamic content (just the page, not the code used to serve it).
- Figure out how to present the results of interacting with the search server in the HTML, so that when a user clicks on a result (i.e., a link) the proper page name is sent to the page server.

For the GUI:

- Make sure you understand the general structure of the FXML file we provide to you.
- Explain how you will re-use some, if not most, of the code you wrote for Sparkzilla. Come prepared to describe what classes and methods must be changed, and how, to render HTML graphically instead of textually.
- Explain how your browser will handle interactions within the page that entail a change of state in the browser. For example, clicking a link requires loading a new page, while editing a form field only requires modifying the current page. What methods will be called on the browser, and from where will they be called?

10.2 Final Handin

The final handin is due by April 29. For the final handin, your scalaproject should contain the packages containing your code for the GUI and server. Finally, your handin 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/cs0180/workspace/scalaproject` directory, and run the command `cs018_handin guizilla`. 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.

### 10.3 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 description of how the session map and unique IDs will be implemented
- An explanation of the code for your simple dynamic web page
- A discussion of the control flow of your GUI

Functionality counts for 70% of your grade, including:

- Server interactions with clients
- Server handling of pages
- Server dynamic demo page
- Server search page
- GUI

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.
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