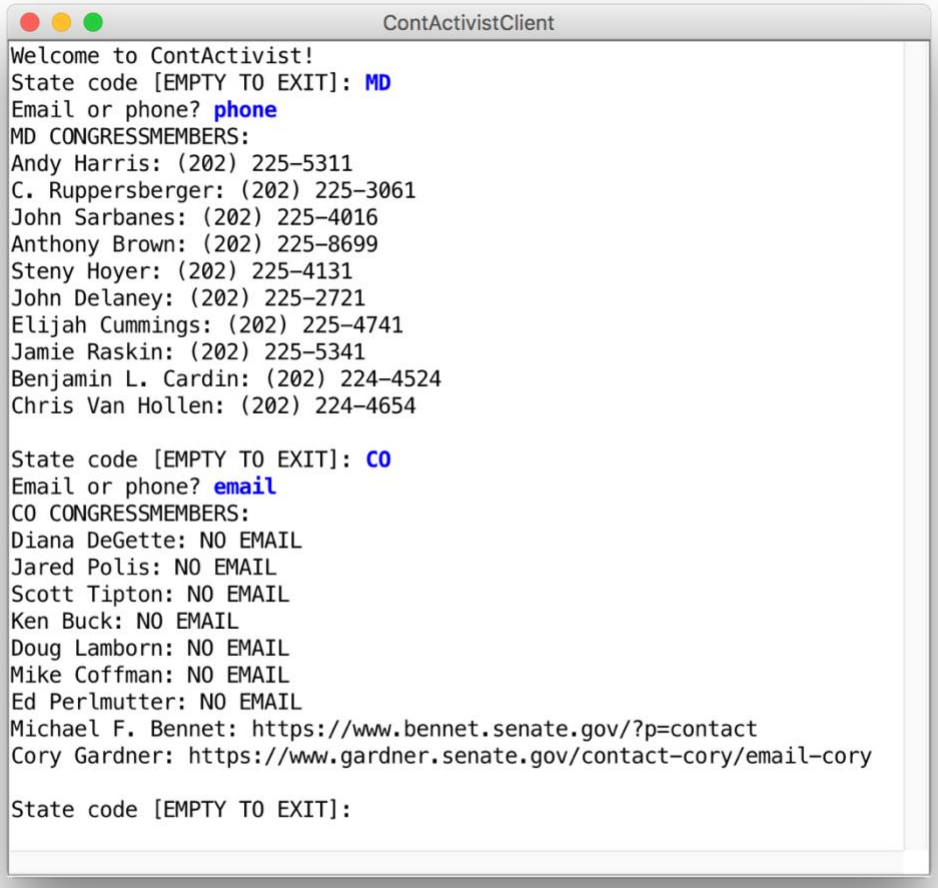
Chris Piech
CS 106A

Section #8
May 30, 2018

Section Handout #8: Data Structures and The Internet

Based on problems by Brandon Burr, Patrick Young, and Nick Troccoli

Your task for this week is to write server and client programs for an application called **ContActivist** that allows the user to easily look up information about congress members representing different states, and how to contact them. Here is what a sample run of the client program might look like:



You will need to apply your knowledge of file reading, data structures, string parsing, internet applications, and more to create this program. We have included specific details about the client and server programs below.

1. ContActivistServer

There are two types of requests that the server needs to handle:

Command	Parameters	Response
`getCongressPhonesForState`	`stateCode (String)`	Contains information about all the state's congress members. Each line contains information about a single congress member in the specified state, in the format `"[NAME]: [PHONE]"`
`getCongressEmailsForState`	`stateCode (String)`	Contains information about all the state's congress members. Each line contains information about a single congress member, in the format `"[NAME]: [EMAIL]"` If a member has no email, include the text "NO EMAIL" instead of the email.

The data comes from a file named **`congress.txt`**, which has the following format:

```
NAME
STATE CODE
PHONE
EMAIL (blank if no email)

NAME
STATE CODE
PHONE
EMAIL (blank if no email)
...
```

To provide a concrete example, here is an excerpt of the provided data file:

```
...
Liz Cheney
WY
(202) 225-2311


Lamar Alexander
TN
(202) 224-4944
http://www.alexander.senate.gov/public/index.cfm?p=Email

Tammy Baldwin
WI
(202) 224-5653
https://www.baldwin.senate.gov/feedback
...
```

In particular, notice how Liz Cheney does not have an email address specified, and thus the line underneath their phone number is blank. You may assume all fields other than email will always be provided.

Your server should:

- Read in the congress member information from the file **`congress.txt`** and store it appropriately when it begins running.
- Respond to **`getCongressEmailsForState`** and **`getCongressPhonesForState`** requests as described above.
- If a provided state code is invalid, return an error message. You may assume that if the state is not in the data file, then it is invalid.

How you implement this server, including any data structures you may choose to use, is up to you. As part of this implementation, you should also consider how designing an additional class, something like **`CongressMember`**, might help you in storing the required information.

2. ContActivistClient

The client should behave as in the screenshot above. Specifically, in a loop, it should:

1) Prompt the user for the state code they would like to look up
2) Ask which of either emails or phone numbers they would like to view
3) Print out the information for all representatives of the inputted state, including either phone numbers or emails (whichever the user specified).

The loop should terminate when the user enters the empty string (“”) as the state code.

Hint: an easy way to prompt the user for a response to an “a or b” question is to write something like the following:

```
// true if they entered "email", false if they entered "phone"
boolean email = readBoolean("Email or phone? ", "email", "phone");
```

The **`readBoolean`** method takes as parameters a prompt to display to the user, followed by the “yes” answer and the “no” answer. **`readBoolean`** will reprompt the user until they enter one of those answers. If the user picks the “yes” answer, **`readBoolean`** returns **`true`**. If the user picks the “no” answer, **`readBoolean`** returns **`false`**.

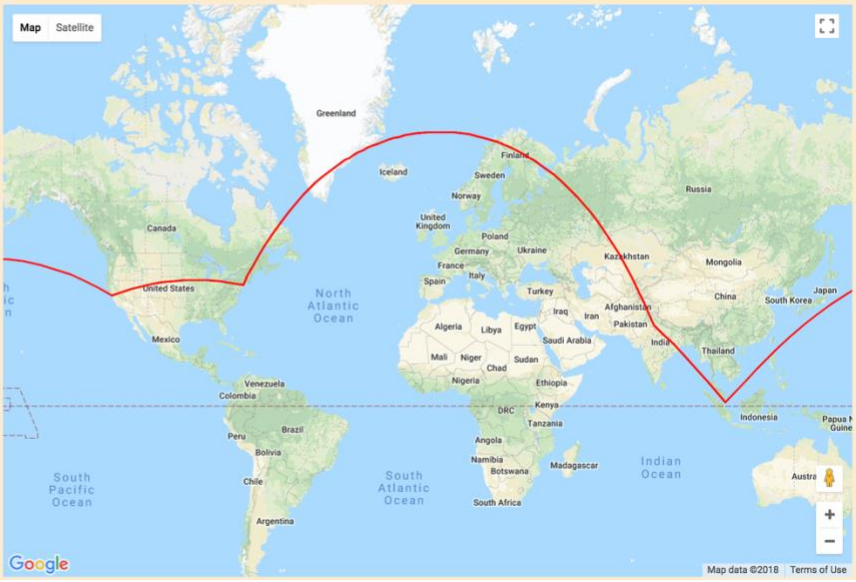
Bonus: Web Demos

In order to see how this might be used to build a real application, SL Brahm Capoor has written an in-browser demonstration of a graphics-based client. You are certainly not expected to implement it, but it gives a more familiar feel to the problem and is fun to play around with. In order to access it, open the section code in Eclipse as you normally would, and start the server running (either your implementation or the solution version). Next, go to the folder in your computer's filesystem that contains the Eclipse project. Open the `webpage` folder and open `index.html` in your browser. It should look like this:

The Flight Planner problem below also has a web demo of a graphical client. Download the extra section problem code and follow the same instructions as above. The client should look like this:

My itinerary

- Singapore
- New Delhi
- New York
- San Francisco
- Singapore



Extra Practice: Flight Planner

These server and client programs allow the user to plan a round-trip flight route. Note that these are long programs and given as extra practice to supplement the section problem above. First, here's what a sample run of the client program might look like:

A critical issue in building these programs is designing appropriate data structures to keep track of the information you'll need in order to produce flight plans. You'll need to both have a way of keeping track of information on available flights that you read in from the **`flights.txt`** file, as described below, as well as a means for keeping track of the flight

routes that the user is choosing in constructing their flight plan. Consider how both **`ArrayList`**s and **`HashMap`**s might be useful to store the information you care about.

1. FlightPlannerServer

There are two types of requests that the server needs to handle from the client in order for the client to do its job:

Command	Parameters	Response
`getAllCities`	N/A	Returns the list of all cities, as a string, that the user can visit.
`getDestinations`	**`city (String)`**	Returns a list of all cities that the user can travel to *from the given city.*

The flight data come from a file named **`flights.txt`**, which has the following format:

- Each line consists of a pair of cities separated by an arrow indicated by the two-character combination **`->`**, as in

```
New York -> Anchorage
```

- The file may contain blank lines for readability (you should just ignore these).

The entire data file used to produce this sample run appears below.

```
San Jose -> San Francisco
San Jose -> Anchorage

New York -> Anchorage
New York -> San Jose
New York -> San Francisco
New York -> Honolulu

Anchorage -> New York
Anchorage -> San Jose

Honolulu -> New York
Honolulu -> San Francisco

Denver -> San Jose

San Francisco -> New York
San Francisco -> Honolulu
San Francisco -> Denver
```

Your server should:

- Read in the flight information from the file **`flights.txt`** and store it in an appropriate data structure when it begins running.
- Respond to **`getAllCities`** and **`getDestinations`** requests as described above.

Hint: when a server receives a request, it must respond with a String. If you need to respond with a *list* (e.g. an **`ArrayList`**), one method is to return the value of that list's **`toString`** method. For instance:

```
ArrayList<String> myList = ...
String stringToSend = myList.toString();
```

2. FlightPlannerClient

The client should behave as in the screenshot above. Specifically, it should:

- Display the complete list of cities.
- Allow the user to select a city from which to start.
- In a loop, print out all the destinations that the user may reach directly from the current city, and prompt the user to select the next city.
- Once the user has selected a round-trip route (i.e., once the user has selected a flight that returns them to the starting city), exit from the loop and print out the route that was chosen.

As mentioned in the server description, the client will need to contact the server to get a list of all cities, and to get the list of cities that can be traveled to from a particular city. As an example, if you want to make the **`getDestinations`** request to the server, you would write something like the following:

```
String city = ...
try {
   Request request = new Request("getDestinations");

   // specify which city to get possible destinations for
   request.addParam("city", city);

   String result = SimpleClient.makeRequest(HOST, request);
   ArrayList<String> cities = makeListFromString(result);
} catch (IOException e) {
   // an error occurred, do something...
}
```

You'll notice the use of the **`makeListFromString`** method in the above example. When a server sends back a response that is the **`toString`** of an **`ArrayList`**, you need to first convert that string to an **`ArrayList`** in order to use it. We have provided a method **`makeListFromString`** that does just that: it takes a string in the format of an **`ArrayList toString`** and returns the **`ArrayList`** it represents.