Exercises for 19.02.2019

Exercise 1: Creating a Form (roughly 15 minutes)

This exercise involves creating an HTML form that POSTs its submitted data to a PHP program on a server. Download `buyagrade.html` to your disk (right-click the link and choose Save Link As...). You need to modify this HTML file by turning it into an HTML form. You will need to give `name` attributes to the form controls so they will be sent to the server as query parameters; the names are up to you. Also, some form controls (such as radio buttons) need `value` attributes. Fill the Section drop-down list with choices MA through MH.

When you're done with this exercise, your form should look roughly like the screenshot at the top of this handout.

Test your form to see that it is submitting the proper parameters by temporarily setting its `action` attribute to:

```
http://webster.cs.washington.edu/params.php
```

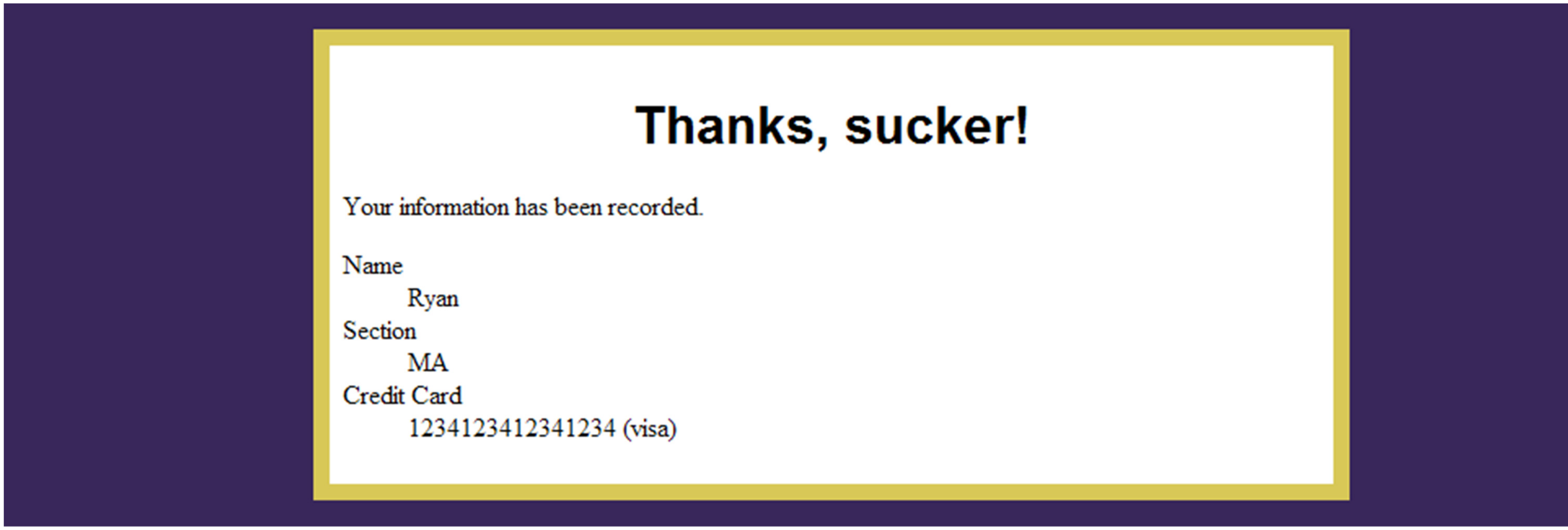
You can try our runnable solution to this exercise (so you can see how your page is supposed to look and work. Don't look at the source code yet, please!)

Exercise 2: Displaying Input Data (roughly 10 minutes)

In this exercise, you will write the PHP page that will handle the submitted form data. Tell you `buyagrade.html` to POST to `sucker.php`. The `sucker.php` page will receive the parameters from `buyagrade.html` and will output an HTML confirmation page. Here is a skeleton version of this page that does not actually display the data submitted by the user. Modify it to display the submitted data.

- sucker.php

Your page should at least display the submitter's name, credit card number, and credit card type (Visa or MasterCard) in the confirmation page. For now, your page doesn't actually need to save this information in any way on the server.


Thanks, sucker!

Your information has been recorded.

Name
    Ryan
Section
    MA
Credit Card
    1234123412341234 (visa)

To keep your code clean, as much as possible you should embed variables' values in HTML using PHP expression blocks such as `<?=` *expression* `?>`.

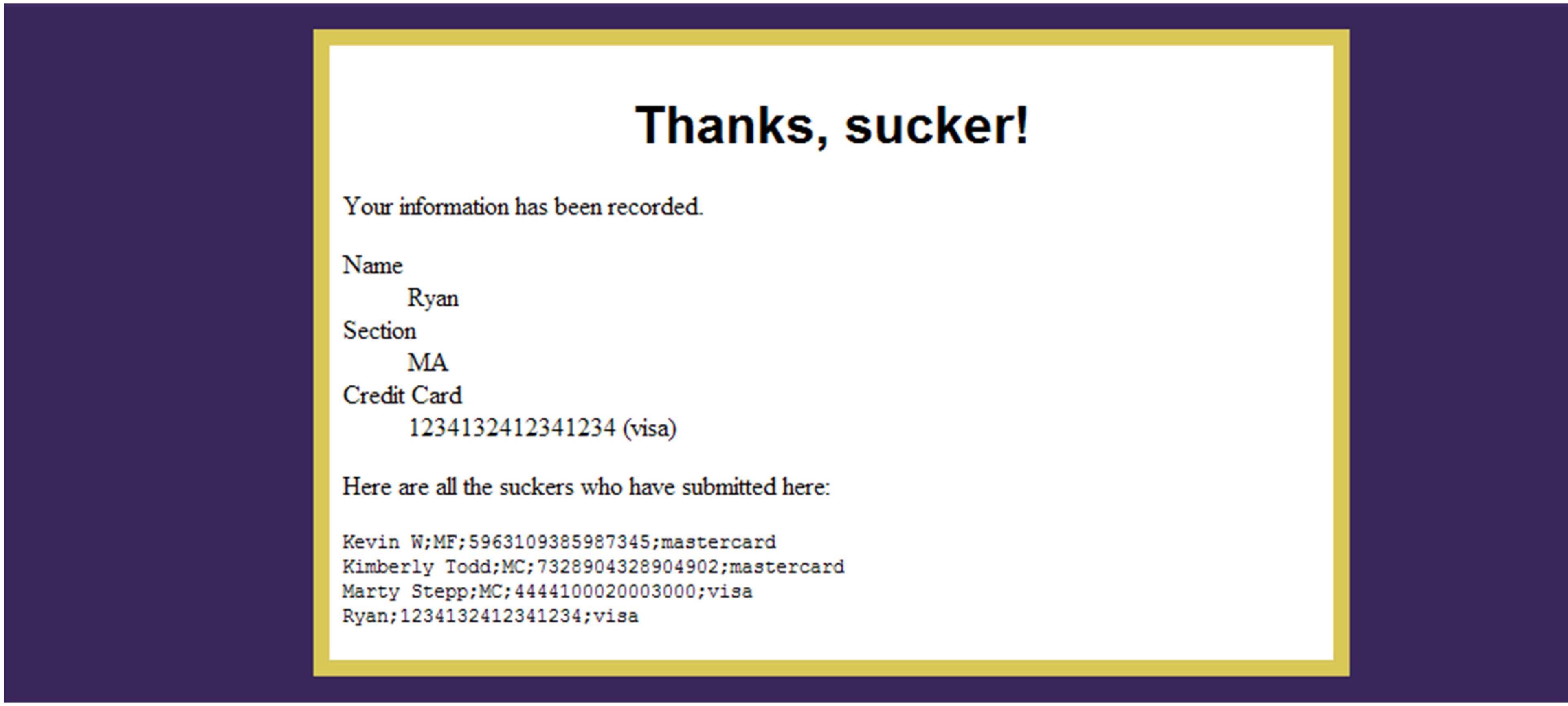
You can try our runnable solution to this exercise (so you can see how your page is supposed to look and work. Don't look at the source code yet, please!)

Exercise 3: Save the Form Data (roughly 10 minutes)

Modify your `sucker.php` page to save the submitted data to the file `suckers.txt`. This file should have a format similar to the following:

```
Ryan;MA;1234123412341234;visa
Kevin W;MF;5963109385987345;mastercard
Kimberly Todd;MC;7328904328904902;mastercard
Marty Stepp;MC;4444100020003000;visa
```

Also change your page's output to show the complete contents of this file to the user. Place the file contents into an HTML `<pre>` element to preserve the whitespace.



Thanks, sucker!

Your information has been recorded.

Name
        Ryan
Section
        MA
Credit Card
        1234132412341234 (visa)

Here are all the suckers who have submitted here:

```
Kevin W;MF;5963109385987345;mastercard
Kimberly Todd;MC;7328904328904902;mastercard
Marty Stepp;MC;4444100020003000;visa
Ryan;1234132412341234;visa
```

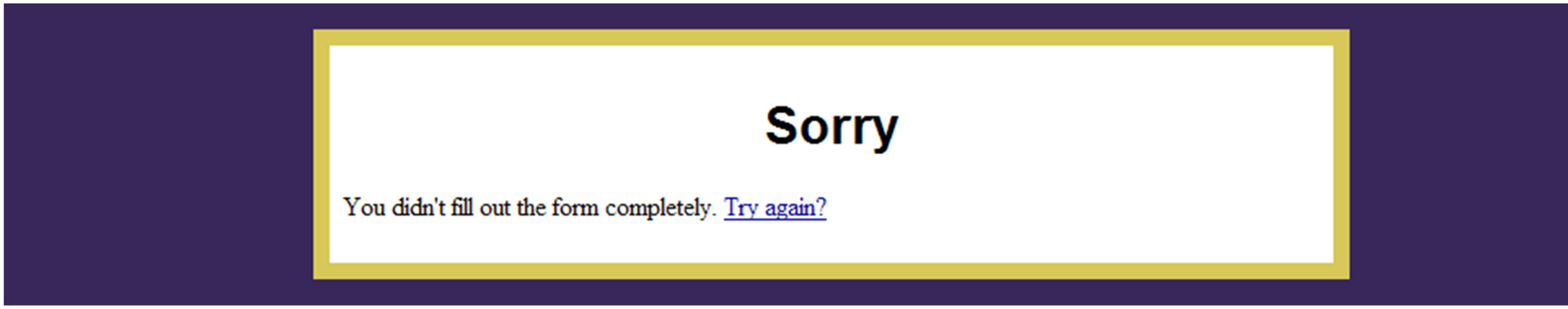
Recall that you can read and write the contents of a file using the PHP `file_get_contents` and `file_put_contents` functions.

Exercise 4: Basic Data Validation (roughly 10 minutes)

Update your `sucker.php` file to verify that the user did not leave any fields blank when submitting the form. You can check whether a particular parameter has been passed using the PHP function `isset`.

However, `isset($_REQUEST["`*name*`"])` will only check if the `$_REQUEST` associative array has `"`*name*`"` as one of its keys; it does not check to see if the value of `$_REQUEST["`*name*`"]` is a non-empty value. Remember that `$_REQUEST["`*name*`"] = ""` would give a falsey value and `$_REQUEST["`*name*`"] = "abc"` would give a truthy value.

If the user has not filled in every field, show an error message like the one below instead of displaying their submitted data:



Sorry

You didn't fill out the form completely. Try again?

Give them the chance to try again by linking back to your HTML page.

You can try our runnable solution to this exercise (so you can see how your page is supposed to look and work. Don't look at the source code yet, please!)

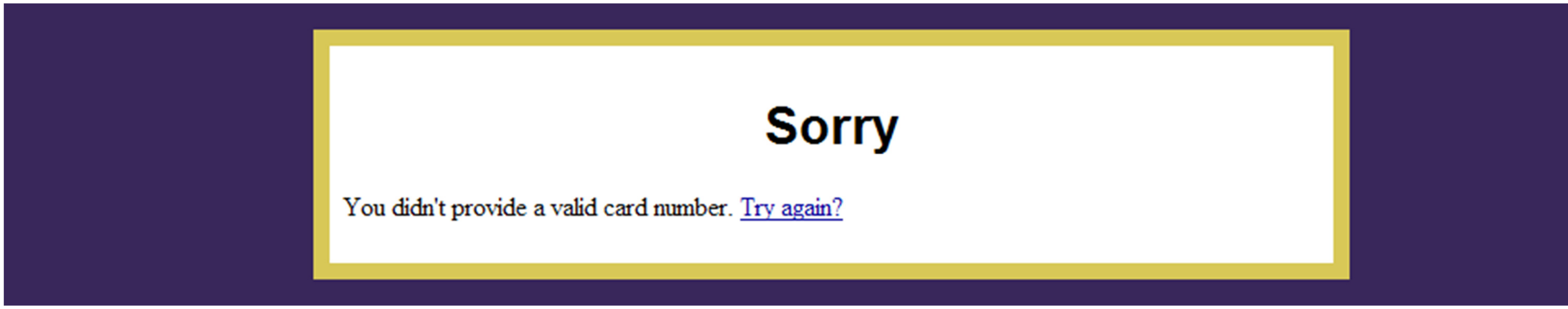
Exercise 5 (l33t h4x0rZ only): Verify Credit Card Numbers (roughly 15-30 minutes)

Update your `sucker.php` file to further check the validity of the credit card number.

- Check to make sure the credit card number is composed of exactly 16 numerical digits.
- Check to make sure that a Visa card number starts with a "4" and a MasterCard number starts with a "5".

- You can check for these things using regular expressions in combination with the php function `preg_match`. A really good regex will allow for an optional - between every grouping of 4 numbers. For example, 4111111111111111 and 4111-1111-1111-1111 would both be valid credit card numbers.

If the user has not supplied a card number with the correct number of digits, show an error message like the one below:



Sorry

You didn't provide a valid card number. Try again?

If you finish the above checks and are **really** l33t, update your `sucker.php` file to check that the supplied card number passes the Luhn Algorithm (wiki). It is a checksum algorithm developed by Hans Peter Luhn often used to protect against accidental errors in typing credit card numbers: any valid credit card number will pass the algorithm, though not every invalid credit card number will fail it. If the submitted credit card number does not pass the Luhn Algorithm, you can display the same error as you did when the card number had the incorrect number of digits.

The Wikipedia page previously linked gives a PHP implementation of the algorithm, but it is poorly coded and hard to understand. If you are running out of time, you may use the code from that page in your php code; otherwise, the text below describes how to code this algorithm and you should use it to try to code it yourself (especially if you're a CSE major!).

From *Building Java Programs*:

You may not know that credit card numbers contain several pieces of information for performing validity tests. For example, a valid credit card number passes a digit-sum test known as the Luhn checksum algorithm. Luhn's algorithm states that if you sum the digits of the number in a certain way, the total sum must be a multiple of 10 for a valid Visa number. Systems that accept credit cards perform a Luhn test before contacting the credit card company for final verification. This lets the company block fake or mistyped credit card numbers.

The algorithm for summing the digits is the following. Consider each digit of the credit card to have a zero-based index: the first is at index 0, and the last is at index 15. Start from the rightmost digit and process each digit one at a time. For digits at odd-numbered indexes (the 15th digit, 13th digit, etc.), simply add that digit to the cumulative sum. For digits at even-numbered indexes (the 14th, 12th, etc), double the digit's value, then if that doubled value is less than 10, add it to the sum. If the doubled number is 10 or greater, add each of its digits separately into the sum.