Exercise Thursday

Introduction to Programming (/introduction-to-programming) / Arrays and Looping (/introduction-to-programming/arrays-and-looping) / Practice: Credit Card Validator, Roman Numerals, or Cryptosquare

Text

Goal: For these exercises (and all exercises going forward) Focus on breaking your project down into small pieces of functionality, and tackling them one at a time with TDD. Write your first test and then write the code necessary to get the test passing. Repeat the process of writing tests and getting code passing until you've completed your business logic. If you are interested, try experimenting with different kinds of loops.

Warm Up

- Explain what test-driven development is. Why is it so beneficial?
- Review the tests for the Pig Latin project that each partner wrote and discuss the following:
  - Is each individual test listed as specific as possible? Does each represent *one* piece of functionality or multiple?
  - Is the simplest possible test listed first? If so, how do you know it's the simplest? If not, which one should come first?

- How did you ensure you wrote the *least* amount of code possible to make each pass? What did that look like?

Code

Meet the goal by completing one of the projects listed below. If you only have a half day to work on these practice prompts, we recommend working on the Credit Card Validator prompt with the goal of completing step 1.

Credit Card Validator

Write a program that checks if a credit card number is valid. There are multiple ways to check the validity of a credit card number, and in this practice prompt, we'll explore just a few ways for only a few credit card companies.

For this prompt, the **input** will always be a string of numbers. Something like this:

```
"0998445533334452"
```

The **output** of the method should be a string with one of two messages depending on whether or not the card number is valid:

```
"This card number is not valid."
```

```
"This card number is valid."
```

As you write your method(s), pay attention to your data types and implement at least one loop and one array. Think about the built-in JavaScript methods and operators you can use. Write tests for all of your business logic functions. Only create a UI after you have completed your business logic (all 3 steps) and worked through the further exploration options.

Step 1: Apply the Luhn Algorithm

Luhn's Algorithm checks the digits of a number in order to validate it. It is widely used today in creating and validating credit card, account, and ID numbers.

Work through the instructions below to learn how to use Luhn's Algorithm. Alternatively, you can visit the Luhn Algorithm Wikipedia page (https://en.wikipedia.org/wiki/Luhn_algorithm) or watch this video from Concerning Reality (https://www.youtube.com/watch?v=Yr9s5NjsVAo) to learn how to use the Luhn Algorithm.

How to apply Luhn's Algorithm:

- Using the inputted card number, create a new set of numbers by transforming each of the digits in the inputted credit card number. Let's use this number `4102 0808 8043 5620` as an example input. After following the steps below the inputted number becomes `4204 0707 8046 5320`.
    - Starting on the right and moving left, double every other digit. For example the digit `3` becomes `6`.
    - If the result of the doubled digit is a double digit number, add together each digit of the double digit number. For example, the digit `7` doubled becomes `14`, adding those together becomes `1 + 4`, which results in `5`.
    - Using our example credit card number `4102 0808 8043 5620`, here are the transformations:

Original number	4	1	0	2	0	8	0	8	8	0	4	3	5	6	2	0

Doubling every other digit	4	**1x2**	0	**2x2**	0	**8x2**	0	**8x2**	8	**0x2**	4	**3x2**	5	**6x2**	2	**0x2**
Adding digits of double-digit numbers	4	**2**	0	**4**	0	**1+6**	0	**1+6**	8	**0**	4	**6**	5	**1+2**	2	**0**
New set of transformed numbers	4	**2**	0	**4**	0	**7**	0	**7**	8	**0**	4	**6**	5	**3**	2	**0**

- Next, sum all of the digits in the new transformed set of numbers. Using our example `4204 0707 8046 5320` becomes the addition of `4 + 2 + 0 + 4 + 0 + 7 + 0 + 7 + 8 + 0 + 4 + 6 + 5 + 3 + 2 + 0`, which equals `52`.
- If the resulting number ends in a zero, the card number is valid. If the number does NOT end in a zero, the card number is NOT valid. In our case, the credit card number is NOT valid, because it ends in a 2.

Here are two credit card numbers for you to use to test your code:

- A valid number: `4102080860435620`
- An invalid number: `4102080880435620`

Step 2: Validate the First Digits of the Credit Card Number

The first digits in a credit card number is used to identify the company that issued the credit card. We can use that information to validate a credit card. Update your method or create a new one that uses the rules below to determine whether the credit card number was issued by an accredited company.

```
American Express cards always begin with the numbers 34 or
37.
Visa cards begin with the number 4.
Mastercards start with the number 5.
Discover Cards begin with the number 6.
```

Step 3: Validate the Length of the Credit Card Number

You can also validate a credit card by its length. Use the rules below to update your method(s) to determine if a card number is valid based on its length. Taking Visa for an example, Visa cards have a length of 16 digits, so a card that has 15 digits or less, or 17 digits or more would be invalid.

```
Visa: 16
Mastercard: 16
Discover: 16
American Express: 15
```

Further Exploration

- Watch the video from Concerning Reality (https://www.youtube.com/watch?v=Yr9s5NjsVAo) about the Luhn Algorithm. Use the instructions in the second half of the video to refactor your method(s) to use the check digit number to validate the credit card number. You can also find this information in the Luhn Algorithm Wikipedia page (https://en.wikipedia.org/wiki/Luhn_algorithm).
- Refactor your application so that it tells the user which company issued the credit card. For example, the program could return "This Discover card number is valid."
- Refactor your application so that it tells you why the credit card is not valid (meaning, the length, initial digits, or not passing the Luhn Algorithm). For example, the program could return "This VISA card number is not valid. It does not have the

correct length." or "This VISA card number is not valid. It does not pass the Luhn Algorithm."

- Create a user interface for your application.

Challenging: Roman Numerals

Write a method to convert numbers into Roman numerals. Roman numerals are based on seven symbols:

Symbol	Value
I	1
V	5
X	10
L	50
C	100
D	500
M	1,000

The most basic rule is that you add the value of all the symbols: so II is 2, LXVI is 66, etc.

The exception is that there may not be more than three of the same characters in a row. Instead, you switch to subtraction. So instead of writing IIII for 4, you write IV (for 5 minus 1); and instead of writing LXXXX for 90, you write XC.

You also have to separate ones, tens, hundreds, and thousands. In other words, 99 is XCIX, not IC. You cannot count higher than 3,999 in Roman numerals.

Do not add any UI logic until you've completed your business logic (and included testing).

Challenging: Cryptosquare

A classic method for composing secret messages is called a *square code*.

The spaces and punctuation are removed from the English text and the characters are written into a square (or rectangle) and the entire message is downcased. For example, the sentence *"don't compare yourself to others, compare yourself to the person you were yesterday"* is 69 characters long, so it is written into a rectangle with 9 rows and 8 columns.

d	o	n	t	c	o	m	p
a	r	e	y	o	u	r	s
e	l	f	t	o	o	t	h
e	r	s	c	o	m	p	a
r	e	y	o	u	r	s	e
l	f	t	o	t	h	e	p
e	r	s	o	n	y	o	u
w	e	r	e	y	e	s	t
e	r	d	a	y			

The coded message is obtained by reading down the columns going left to right, outputting encoded text in groups of five letters. For example, the message above is coded as:

"daeer leweo rlref rerne fsyts rdtyt coooe acooo utnyy ouomr hyemr tpseo spsha eput"

Write a program that outputs the encoded version of a given block of text. Again, identify each individual behavior this application should demonstrate, and write a test for each. Tackle writing code for *one* behavior at a time. Manually test the code described in each test in the console before moving onto the next one. All tests should be included in the project README.

The size of the square (number of columns) should be decided by the length of the message. If the message is a length that creates a perfect square (e.g. 4, 9, 16, 25, 36, etc), use that number of columns. If the message doesn't fit neatly into a square, choose the number of columns that corresponds to the smallest square that is larger than the number of characters in the message.

```
> encrypt("Have a nice day. Feed the dog & chill out!");
"hifei acedl v..."
```

Further Exploration

Go back and tackle any *Further Exploration* exercises from previous classwork in this section that you have not yet completed.

Peer Code Review

- Are tests included for all functionality?
- Are all tests passing?
- Are variable names descriptive and in lower camel case?
- Is code indented properly throughout?
- If a user interface is included, is the business and user interface logic well-separated?
- Does the application work as expected?
- Is the code clean, well-refactored, and generally easy to follow?

Previous (/introduction-to-programming/arrays-and-looping/practice-array-mapping)
Next (/introduction-to-programming/arrays-and-looping/further-exploration-looping-with-for-of)

Lesson 45 of 50
Last updated March 24, 2023

disable dark mode

(http://www.epicodus.com)