Classes

Tasks

CoffeeShop

Properties:

- `name`: a string (basically, of the shop)
- `menu`: an array of items (of object type), with each item containing the item (name of the item), type (whether food or a drink) and price.
- `orders`: an empty array

Methods:

- `addOrder`: adds the name of the item to the end of the orders array if it exists on the menu. Otherwise, return "This item is currently unavailable!"
- `fulfillOrder`: if the orders array is not empty, return "The {item} is ready!". If the orders array is empty, return "All orders have been fulfilled!" listOrders: returns the list of orders taken, otherwise, an empty array.
- `dueAmount`: returns the total amount due for the orders taken.
- `cheapestItem`: returns the name of the cheapest item on the menu.
- `drinksOnly`: returns only the item names of type drink from the menu.
- `foodOnly`: returns only the item names of type food from the menu.
- **IMPORTANT**: Orders are fulfilled in a FIFO (first-in, first-out) order.

Examples:

```
tcs.addOrder("hot cocoa"); // "This item is currently unavailable!"
// Tesha's coffee shop does not sell hot cocoa
tcs.addOrder("iced tea"); // "This item is currently unavailable!"
// specifying the variant of "iced tea" will help the process

tcs.addOrder("cinnamon roll"); // "Order added!"
tcs.addOrder("iced coffee"); // "Order added!"
tcs.listOrders; // ["cinnamon roll", "iced coffee"]
// the list of all the items in the current order

tcs.dueAmount(); // 2.17

tcs.fulfillOrder(); // "The cinnamon roll is ready!"
tcs.fulfillOrder(); // "The iced coffee is ready!"
tcs.fulfillOrder(); // "All orders have been fulfilled!"
// all orders have been presumably served

tcs.listOrders(); // []
// an empty array is returned if all orders have been exhausted

tcs.dueAmount(); // 0.0
```

```
// no new orders taken, expect a zero payable

tcs.cheapestItem(); // "lemonade"
tcs.drinksOnly(); // ["orange juice", "lemonade", "cranberry juice",
"pineapple juice", "lemon iced tea", "vanilla chai latte", "hot
chocolate", "iced coffee"]
tcs.foodOnly(); // ["tuna sandwich", "ham and cheese sandwich", "bacon and
egg", "steak", "hamburger", "cinnamon roll"]
```

Shiritori

This challenge is an English twist on the Japanese word game Shiritori. The basic premise is to follow two rules:

- First character of next word must match last character of previous word.
- The word must not have already been said.

Below is an example of a Shiritori game:

```
["word", "dowry", "yodel", "leader", "righteous", "serpent"]; // valid!
["motive", "beach"]; // invalid! - beach should start with "e"
["hive", "eh", "hive"]; // invalid! - "hive" has already been said
```

Write a Shiritori class that has two instance properties:

- `words`: an array of words already said.
- `game_over`: a boolean that is true if the game is over.

  Methods:

- `play`: a method that takes in a word as an argument and checks if it is valid (the word should follow rules #1 and #2 above).
  - If it is valid, it adds the word to the words array, and returns the words array.
  - If it is invalid (either rule is broken), it returns "game over" and sets the `game_over` boolean to true.
  - restart: a method that sets the words array to an empty one [] and sets the game_over boolean to false. It should return "game restarted".

Examples:

```
myShiritori = new Shiritory();

myShiritori.play("apple"); // ["apple"]
myShiritori.play("ear"); // ["apple", "ear"]
myShiritori.play("rhino"); // ["apple", "ear", "rhino"]
myShiritori.play("corn"); // "game over"

// Corn does not start with an "o".

myShiritori.words; // ["apple", "ear", "rhino"]

// Words should be accessible.

myShiritori.restart(); // "game restarted"
myShiritori.words; // []

// Words array should be set back to empty.

myShiritori.play("hostess"); // ["hostess"]
myShiritori.play("stash"); // ["hostess", "stash"]
myShiritori.play("hostess"); // "game over"
```

IMPORTANT Words cannot have already been said.

- The play method should not add an invalid word to the words array.
- You don't need to worry about capitalization or white spaces for the inputs for the play method. There will only be single inputs for the play method.

Account

Create an Account class that have.

- Properties:
    - id: it's should be a uniq
    - name: it's should be a string
    - balance: it's should be hidden property and should have get and set methods
- Methods:
    - `get` and `set` methods for the `balance`
    - `credit` which should increase a new amount on the `balance`
    - `debit` which should decrease an amount from the `balance`
    - `transferTo` which takes other account and amount and transfer from current balance to the balance of the given account
    - `identifyAccounts`, this should be a static method for identify accounts by id of them

```
const saving = new Account("saving", 1000);
const current = new Account("current", 8000);

saving.credit(5000);
saving.debit(1000);
saving.debit(2000);
saving.transferTo(current, 1000);
console.log(saving.balance);
console.log(current.balance);

const res = Account.identifyAccounts(current, saving);

console.log(saving.balance);
saving.balance = "hello";

saving.submitBalance("hello");
console.log(saving);
```