1. Write classes: *Person*, *Student*, *Staff*.
   Person should have:
   - firstName,
   - lastName,
   - gender,
   - age.

   It should have appropriate getters and setters.
   It should have a method:
   - toString().

   Student is inherited from *Person*. It should have
   - program(array of strings),
   - year,
   - fee.

   It should have appropriate getters and setters.
   It should have a methods:
   - passExam(program, grade),
   - isAllPassed()
   - toString()

   Student should contain the data about their programs and exams. passExam will update that data, so if a student passed all the exams(grade is great or equal to 50), its year should be increased by one.

   Teacher is inherited from Person. It should have
   - program (string),
   - pay.

   It should have appropriate getters and setters.
   It should have a method:
   - toString()

2. Create a *CustomConsole* class with following methods:
   - log() function that takes user arguments and returns them as a string,
   - history() function that takes an optional range as an argument,
   - clearHistory() function to remove the history memory.

   The log function has no limit of arguments.

```
const myConsole = new Console('Regular');

const fancyConsole = new Console('Fancy');
```

```
myConsole.log([0, 1, 2, 3]) // "Regular: [0,1,2,3]"

myConsole.log({ a:1, b:2 }) // "Fancy: {a:1, b:2}"

myConsole.log("ok : ", 1, 2, 3) → "ok : 1, 2, 3"

myConsole.clearHistory() // true

myConsole.history() // ""
```

3. Write a class called *CoffeeShop*, which has three instance variables:

    1. name: a string (basically, of the shop)
    2. 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.
    3. orders : an empty array

    and seven methods:

    1. 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!"`
    2. 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!"`
    3. listOrders: returns the list of orders taken, otherwise, an empty array.
    4. dueAmount: returns the total amount due for the orders taken.
    5. cheapestItem: returns the name of the cheapest item on the menu.
    6. drinksOnly: returns only the *item* names of *type* drink from the menu.
    7. foodOnly: returns only the *item* names of *type* food from the menu.

    IMPORTANT: Orders are fulfilled in a FIFO (first-in, first-out) order.

```
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"]
```

4. Shiritori Game

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

1. The first character of the next word must match the last character of the previous word.
2. The word must not have already been saying.

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 saying.

- 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.

5. Describe a model of a library. For that define classes: *Library*, *Reader*, *Book*.

To create correct hierarchies and connections, you should have a subclass of *Book* such as *LibraryBookBase*, *LibraryBook*, *ReaderBook*.

Book should have
- fields
  - title - string
  - author - string
- methods
  - getters for fields
  - toString()
  - isTheSameBook(book) - which returns true if the book title and author is the same with the current instance, false, otherwise.

LibraryBookBase should have
- fields
  - title - string
  - author - string
  - bookId - number
- methods
  - getters for fields
  - toString()

LibraryBook should have
- fields
  - title - string
  - author - string
  - bookId - number
  - quantity - number
- methods
  - getters for fields
  - setters for appropriate fields
  - toString()
  - increaseQuantityBy(amount - number) - increases the quantity of the book by the given amount.
  - decreaseQuantityBy(amount - number) - decrease the quantity of the book by the given amount.

ReaderBook should have
- fields
  - title - string
  - author - string

bookId - number
expirationDate - string
isReturned - boolean

methods

getters for fields
setters for appropriate fields
toString()

Reader should have

fields

firstName - string
lastName - string
readerId - number
books - Array of ReaderBook

methods

getters for fields
setters for appropriate fields
toString()
borrowBook(book - Book, library - Library) - requests a book from the library. If returned book is not a null and is a type of **ReaderBook**, pushes it to the books.

Library should have

fields

books - Array of LibraryBook
readers - Array of Readers

methods

getters for fields
doHaveBook(requestedBook - Book) - returns true if library has the book, false otherwise.
addBook(newBook - Book) - add new book to the library. If the book already exists, increases its quantity, otherwise adds new book of type **LibraryBook**.
addBooks(newBooks) - add new books to the library with the same logic as the addBook. Returns changed array of the books.
checkReaderId(readerId) - returns true if there exist a reader with the given id, otherwise returns false.
lendBook(book - Book, readeId) - checks whether the book exists and there is at least one at the library. Checks whether library has a reader with the given id. If the both are true, returns a book of type **ReaderBook**.