/\*\*

\* For this lab, we will be referring to PersonProfile as an

\* object containing a name, age and the time the object was created.

\*

\* Below is an example instance of this object:

\* {

\* name: 'Feliks Zemdegs',

\* age: 28,

\* createdAt: 1715151808,

\* }

\*

\* @typedef {Object} PersonProfile

\* @property {string} name - The full name of the person.

\* @property {number} age - The current age of the person.

\* @property {number} createdAt - Unix timestamp in seconds for creation time

\* The syntax [variable] below denotes that the property is optional. This means that it may

\* or may not exist in the object.

\* @property {number} [updatedAt] - Unix timestamp in seconds for update time (optional)

\*/

/\*\*

\* Create a person's profile for the current time. There should be exactly

\* three properties: name, age and createdAt.

\*

\* The updatedAt property should not be in the returned object.

\*

\* Age is calculated by subtracting the birthYear from the currentYear. You

\* do not need to worry about invalid inputs for birthYear, negative age,

\* months of the year, etc.

\*

\* Below is a sample return value:

\* {

\* name: 'Feliks Zemdegs',

\* age: 28,

\* createdAt: 1715151808,

\* }

\*

\* @param {string} nameFirst

\* @param {string} nameLast

\* @param {string} birthYear

\* @returns {PersonProfile}

\*/

function profileCreate(nameFirst, nameLast, birthYear) {

/\* The current year can be retrieved as follows: \*/

const currentYear = new Date().getFullYear();

/\* A unix timestamp can be created as follows: \*/

const timestamp = Math.floor(Date.now() / 1000);

// TODO: change the return statement below to the correct type

return null;

}

/\*\*

\* Compare the age of two different profiles.

\* If profile1's age is higher than profile2's age, return any positive number (e.g. 1)

\* If profile1's age is less than profile2's age, return any negative number (e.g. -1)

\* Otherwise, profile1's age is the same as profile2's age, return 0.

\*

\* @param {PersonProfile} profile1

\* @param {PersonProfile} profile2

\* @returns {number}

\*/

function profileCompareAge(profile1, profile2) {

// TODO: change the return statement below to be a number >0, <0 or 0.

return null

}

/\*\*

\* Given a profile,

\* 1. Edit the profile's name to be the new name

\* 2. Create (if not exist), or edit the property updatedAt to be a new timestamp

\*

\* For example, if the given profile is:

\* {

\* name: 'Alan Becker',

\* age: 34,

\* createdAt: 1715153483

\* }

\* and we make a function call such as: profileUpdateName('The Chosen One')

\* The new object could look like:

\* {

\* name: 'The Chosen One',

\* age: 34,

\* createdAt: 1715153483

\* updatedAt: 1715153612

\* }

\* where updatedAt would depend on the current time.

\*

\* You do NOT need to worry about the age field

\*

\* You do NOT need to return anything in this function

\* (i.e. the return type is undefined - you should not use the return keyword)

\*

\* @param {PersonProfile} profile

\* @param {string} property

\* @returns {undefined}

\*/

function profileUpdateName(profile, newName) {

// TODO: edit the given profile object in-place for 'name' and 'updatedAt'.

}

/\*\*

\* Returns true if the profile has the property 'updatedAt' and false otherwise

\*

\* @param {PersonProfile} profile

\* @returns {boolean}

\*/

function profileHasUpdate(profile) {

// TODO: return either true or false

return null;

}

/\*\*

\* Convert the given profile into a JSON string

\* HINT: this function can be done in one line of code using a built-in JavaScript feature.

\*

\* @param {PersonProfile} profile

\* @returns {string}

\*/

function profileSerialise(profile) {

// TODO: research online for how to convert a JavaScript object to a JSON string

return null;

}

/\*\*

\* Parse the given serialised JSON string into a profile object (i.e. reverting profileSerialise)

\* HINT: this function can be done in one line of code using a built-in JavaScript feature.

\*

\* @param {string} profileString

\* @returns {PersonProfile}

\*/

function profileDeserialise(profileString) {

// TODO: research online for how to convert a JSON string to a JavaScript object.

return null

}

// DEBUG YOUR CODE - run with "node objects.js"

// NOTE: you should add more console.log statements to debug your code for each function!

console.log(`

\* You will not be able to compare two objects with '==='.

\* For week 1 and week 2, you can simply use console.log() and visually

\* compare the output line by line.

\*

\* NOTE: the output of any console.log statements, e.g. colours/whitespaces

\* does not matter when we mark your code, as we will be assessing the

\* returned objects from your functions directly.

\*

\* This means that if a number appears as brown/yellow, a string appears as

\* green, or some part of your output is on a different line, it is all okay!

\* This is simply how NodeJS format their output :).

\*/

`)

console.log("\n======= DEBUGGING profileCreate ======");

console.log("Calling profileCreate('Michael', 'Jordan', 1963). Check manually if the output is consistent:");

console.log('Received:', profileCreate('Michael', 'Jordan', 1963));

console.log("\n======= DEBUGGING profileCompareAge ======");

console.log('For this test, the console.assert statements below will print nothing if the test pass');

// Hard-code the return values of profileCreate:

const benTen = { name: 'Ben Ten', age: 10, createdAt: 1716191810 };

const benEleven = { name: 'Ben Eleven', age: 11, createdAt: 1716191814 };

const benAlsoTen = { name: 'Ben Also-Ten', age: 10, createdAt: 1716191818 };

console.assert(profileCompareAge(benTen, benEleven) < 0, '10 is less than 11, should return a negative number');

console.assert(profileCompareAge(benEleven, benTen) > 0, '11 is greater than 10, should return a positive number');

console.assert(profileCompareAge(benTen, benAlsoTen) === 0, '10 is equal to 10, should return 0');

console.log("\n======= DEBUGGING profileUpdateName ======");

console.log('For this test, you should see the name changes and the "updatedAt" field being added after edit');

const tennisPlayer = profileCreate('Novak', 'Djokovic', 1987);

console.log('Before edit:', tennisPlayer);

profileUpdateName(tennisPlayer, 'Roger Federer');

console.log('After edit:', tennisPlayer);

console.log("\n======= DEBUGGING profileHasUpdate ======");

console.log('For this test, the console.assert statements below will print nothing if the test pass');

const singer = profileCreate('Taylor', 'Swift', 1989);

console.assert(profileHasUpdate(singer) === false, 'Error: this profile has not been changed, but profileHasUpdate returned true');

profileUpdateName(singer, 'Adele Laurie Blue Adkins');

console.assert(profileHasUpdate(singer) === true, 'Error: this profile been changed, but profileHasUpdate returned false');

console.log("\n======= DEBUGGING profileSerialise ======");

console.log('For this test, you should manually check if the object was converted to a string and then revert back:');

const gamer = profileCreate('Sang-hyeok', 'Lee', 1996);

console.log('Original:', gamer);

const serialisedStringGamer = profileSerialise(gamer);

console.log('Serialised as JSON string:', serialisedStringGamer);

console.log('Deserialised JSON string back to original:', profileDeserialise(serialisedStringGamer));

## Implementation

Open the file [objects.js](file:////COMP1531/24T3/students/z5547184/lab01_objects/-/blob/master/objects.js) in your preferred text editor. The stub code (interface with a fake temporary implementation) for each function has been provided for you.

Before replacing them with your implementation, pay close attention to the returned object and see how it aligns with the [Interface: Functions](#interface-fuctions) and [Interface: Data Types](#interface-data-types).

Complete all listed functions.