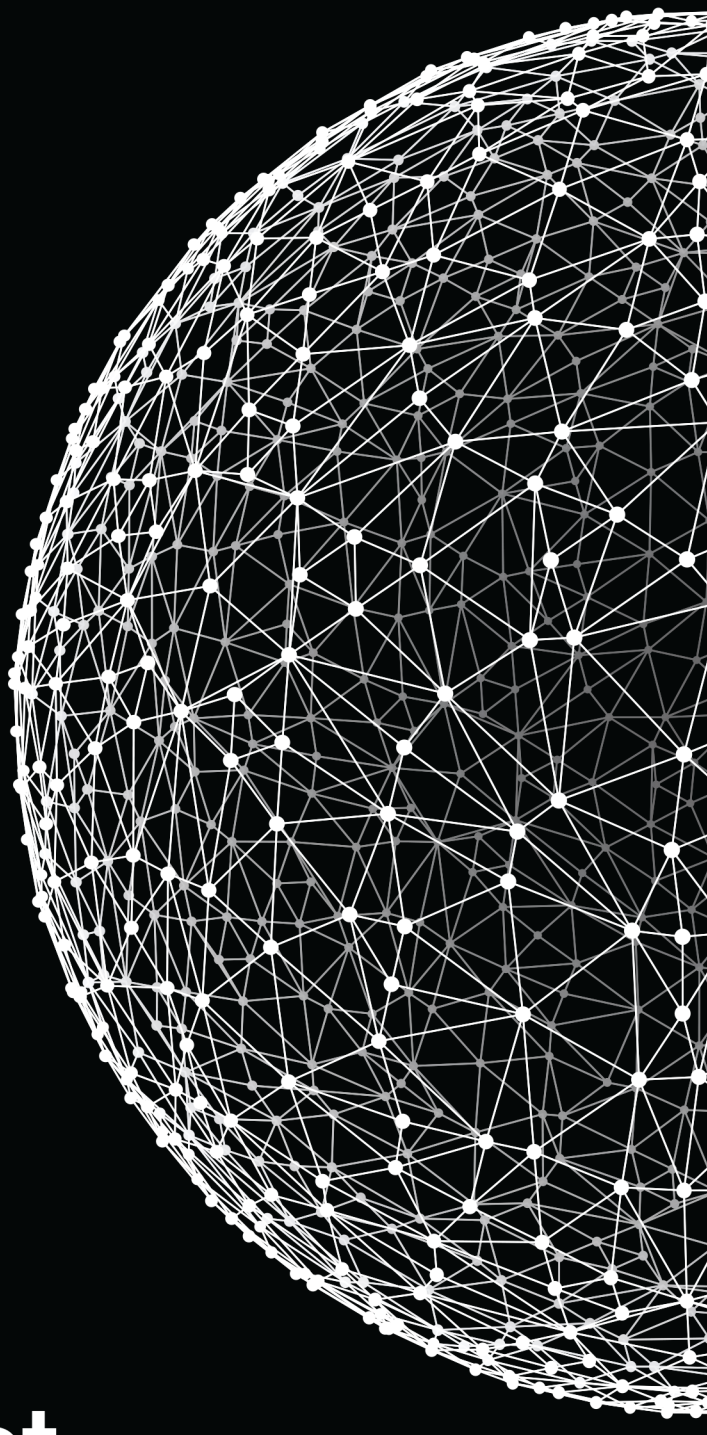
Sprint 03

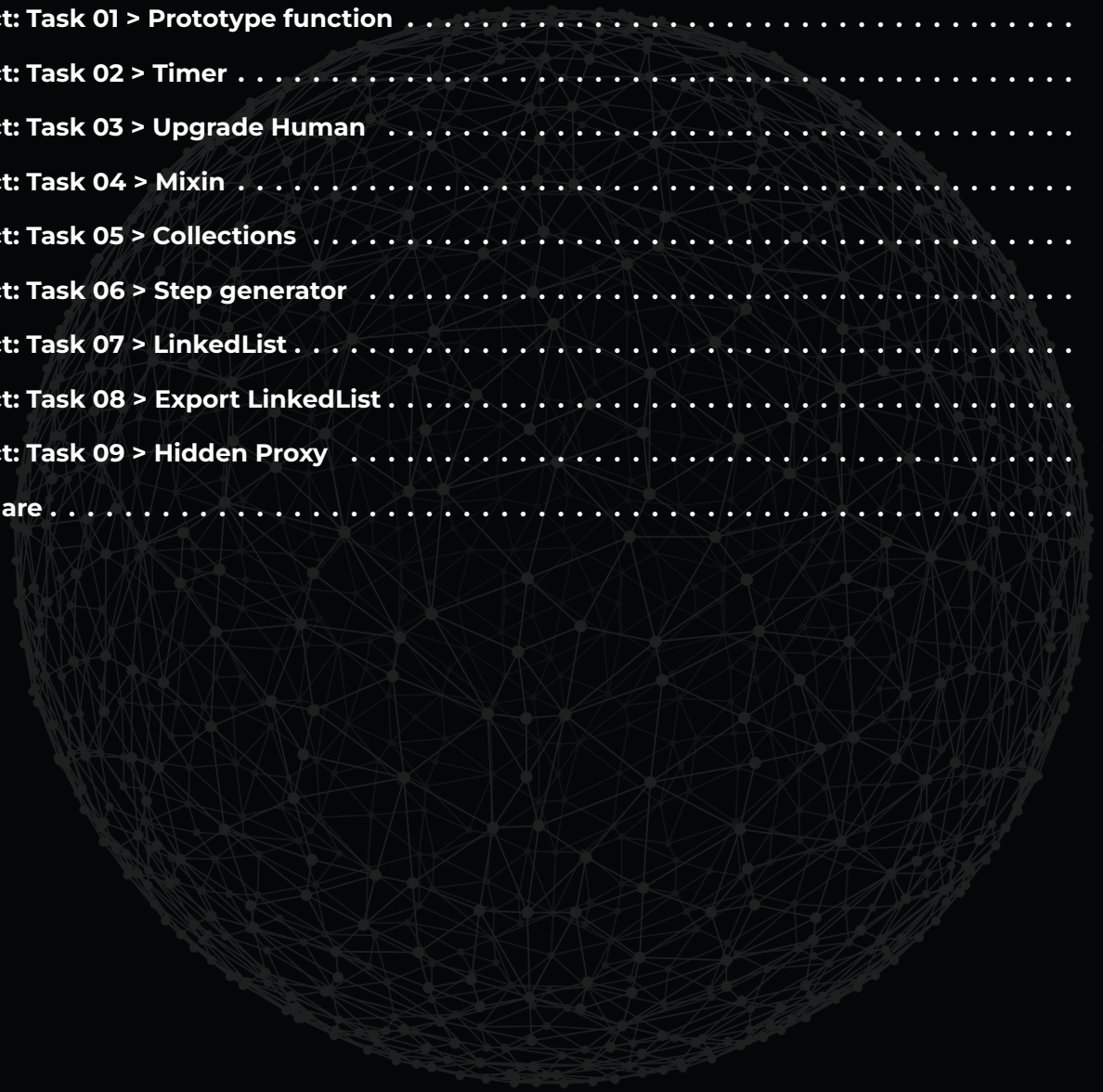
Half Marathon Web

January 15, 2021



ucode connect

Contents



ucode connect

Engage

DESCRIPTION

Hello again!

In the previous Sprint you have started using JavaScript, and, in particular, objects. Today, you will continue this learning path by getting up close and personal with object-oriented programming.

The OOP model of programming is centered around data, rather than functions and procedures. It uses objects, which can be described as containers of data that have unique characteristics and behaviors. When data is organized into objects, code becomes a lot more readable, scalable, and functional.

The key principles of OOP include, but are not limited to,

- encapsulation
- polymorphism
- inheritance

Don't worry if these words mean nothing to you at this point, you will make sense of them once you get some experience using the OOP approach.

Without further ado, let's get started.

BIG IDEA

Basics of OOP programming on JS.

ESSENTIAL QUESTION

How to write code according to OOP principles?

CHALLENGE

Learn to use objects.

Investigate

GUIDING QUESTIONS

We invite you to find answers to the following questions. By researching and answering them, you will gain the knowledge necessary to complete the challenge. To find answers, ask the students around you and search the internet. We encourage you to ask as many questions as possible. Note down your findings and discuss them with your peers.

- What are encapsulation, polymorphism, and inheritance?
- Can you name other OOP principles?
- What is a class parameter, and what is a class method?
- Which rules does JS have for variables names?
- What is an object prototype in JS?
- What is a class constructor?
- What types of constructors exist?
- What is mixin, and what makes it useful?
- What is the difference between WeakSet and Set, and WeakMap and Map?
- How does the `isNaN()` function work?
- What is the main feature of linked lists? Why are they useful?
- What is an iterator in JS?
- What are JS modules? What OOP principles do they implement?
- Why do you need to use `get` and `set` methods in a class definition?

GUIDING ACTIVITIES

Complete the following activities. Don't forget that you have a limited time to overcome the challenge. Use it wisely. Distribute tasks correctly.

- Find definitions for a class, a method, and an object.
- Find several good online resources on OOP.
- Read about OOP in JavaScript and what's special about it.
- Find out what a proxy is.
- Explore object-oriented design principles for writing clean code.
- Read this article on OOP.
- Clone your git repository, issued on the challenge page in the LMS.
- Employ the full power of P2P by brainstorming with other students.

ANALYSIS

Analyze your findings. What conclusions have you made after completing guiding questions and activities? In addition to your thoughts and conclusions, here are some more analysis results.

- Be attentive to all statements of the story. Examine the given examples carefully. They may contain details that are not mentioned in the task.
- Analyze all information you have collected during the preparation stages.
- Perform only those tasks that are given in this document.
- Submit only the specified files in the required directory and nothing else. Garbage shall not pass.
- Pay attention to what is allowed. Use of forbidden stuff is considered a cheat and your challenge will be failed.
- The web page in the browser must open through `index.html`.
- The scripts must be written outside the HTML file - in a separate JS file (script.js).
- You can always use the `Console` panel to test and catch errors.
- Complete tasks according to the rules specified in the following style guides:
  - HTML and CSS: Google HTML/CSS Style Guide. As per section 3.1.7 Optional Tags, it doesn't apply. Do not omit optional tags, such as `<head>` or `<body>`
  - JavaScript:
    * JavaScript Style Guide and Coding Conventions
    * JavaScript Best Practices
- The solution will be checked and graded by students like you. Peer-to-Peer learning.
- If you have any questions or don't understand something, ask other students or just Google it.

Act: Task 00

NAME

Prototype

DIRECTORY

`t00/`

SUBMIT

`index.html, js/script.js`

ALLOWED

`Object.*, Math.*, Date.*`

DESCRIPTION

Let's build some houses.
Create a prototype for a house called `houseBlueprint`, and a constructor `houseBuilder`, which builds houses based on the prototype.

The prototype `houseBlueprint` represents house objects and contains the following properties:

- `address` - a string that describes the house's street address
- `date` - current date (date of creation)
- `description` - a string that describes the house's characteristics
- `owner` - a string that describes the name of the owner of the house
- `size` - size of the house in m^2
- method `getDaysToBuild` - returns number of days needed to build the house (calculated based on size)

The default building speed is 0.5 of a room per day.

The constructor for creating houses, `houseBuilder`, builds houses with the prototype `houseBlueprint` and the property `roomCount`.

Your 'index.html' must include the 'script.js' file.

See the EXAMPLE section for a script that can test your code. Add more test cases of your own.

EXAMPLE

```
const house = new HouseBuilder('88 Crescent Avenue',
    'Spacious town house with wood flooring, 2-car garage, and a back patio.',
    'J. Smith',
    110,
    5);

console.log(house.address);
// '88 Crescent Avenue'

console.log(house.description);
// 'Spacious town house with wood flooring, 2-car garage, and a back patio.'

console.log(house.size);
// 110

console.log(house.date.toDateString());
// [the current date], for example: 'Tue Aug 11 2020'

console.log(house.owner);
// J. Smith

console.log(house._averageBuildSpeed);
// 0.5

console.log(house.getDaysToBuild());
// 220

console.log(house.roomCount);
// 5
```

SEE ALSO

Resources for developers, by developers.

Act: Task 01

NAME

Prototype function

DIRECTORY

`t01/`

SUBMIT

`index.html, js/script.js`

ALLOWED

`String.*, Array.*`

DESCRIPTION

Create a method `removeDuplicates` and add it to the prototype of the global object String.

The method takes a string and removes all the duplicates and extra spaces between and around words. In this task, a word is any number of printable characters separated by whitespace characters or the beginning/end of the string.

Your 'index.html' must include the 'script.js' file.

See the test case in the EXAMPLE for more clues on how your script should work.

EXAMPLE

```
let str = "Giant Spiders?   What's next? Giant Snakes?";
console.log(str);
// Giant Spiders?   What's next? Giant Snakes?
str = str.removeDuplicates();
console.log(str);
// Giant Spiders? What's next? Snakes?
str = str.removeDuplicates();
console.log(str);
// Giant Spiders? What's next? Snakes?

str = ". . . . ? . . . . . . . . . . . . ";
console.log(str);
// . . . . ? . . . . . . . . . . . .
str = str.removeDuplicates();
console.log(str);
// . ?
```

Act: Task 02

NAME

Timer

DIRECTORY

`t02/`

SUBMIT

`index.html, js/script.js`

ALLOWED

`Object.*, setInterval(), Class.*`

DESCRIPTION

Create a class `Timer` for a specialized timer.
The timer

- has the properties:
  - 'title' (String)
  - 'delay' (Number) that describes the duration of one cycle in milliseconds
  - 'stopCount' (Number) - the number of cycles, after which the timer stops
- uses a function-constructor to initialize
- has the following methods in the prototype
  - 'start' - starts the timer
  - 'tick' - prints a 'tick' to a console each time the timer does a cycle
  - 'stop' - stops the timer

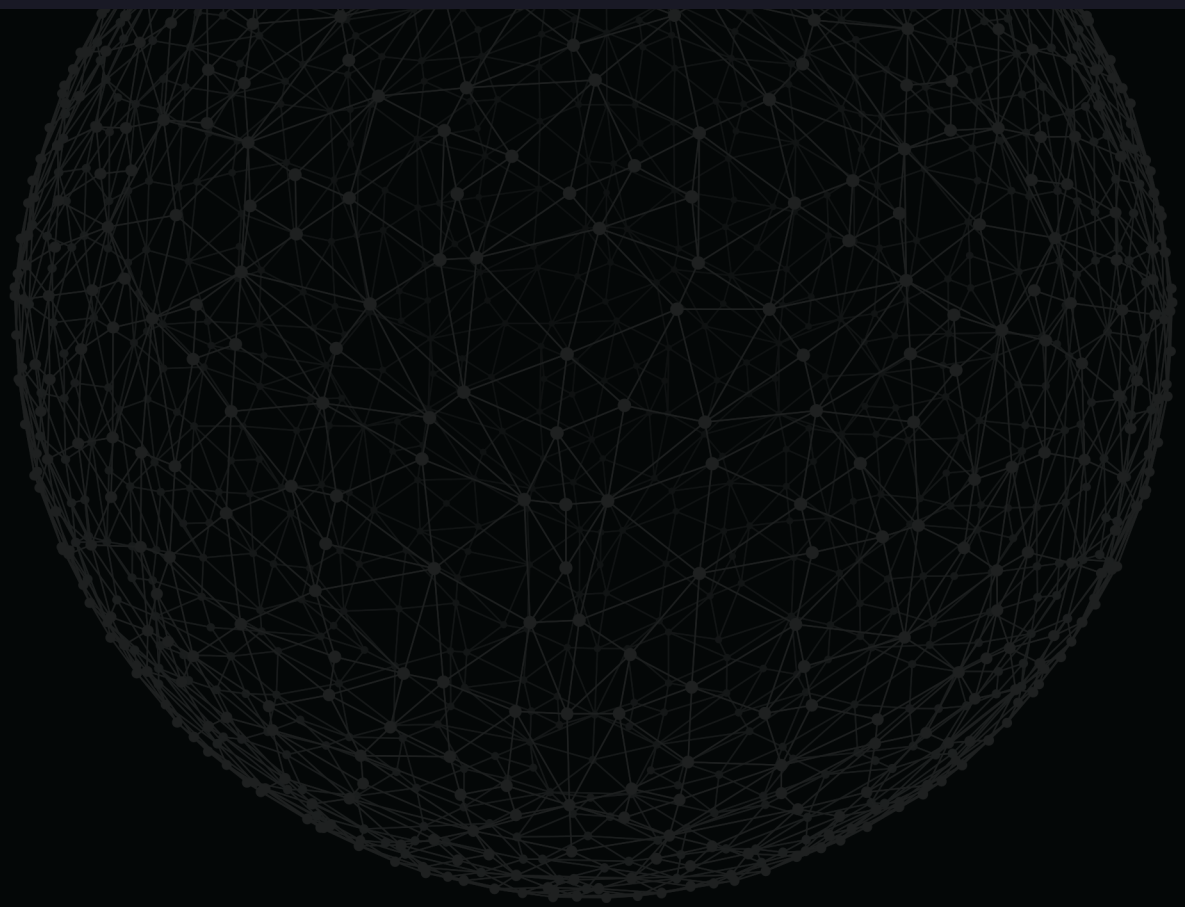
Also, create a function `runTimer(id, delay, counter)`, to initialize and start an object timer.

Your 'index.html' must include the 'script.js' file.

See the EXAMPLE section for a script that can test your code. Add more test cases of your own.

EXAMPLE

```
runTimer("Bleep", 1000, 5);
/*
Console output:

Timer Bleep started (delay=1000,  stopCount=5)
Timer Bleep Tick! | cycles left 4
Timer Bleep Tick! | cycles left 3
Timer Bleep Tick! | cycles left 2
Timer Bleep Tick! | cycles left 1
Timer Bleep Tick! | cycles left 0
Timer Bleep stopped
*/
```



Act: Task 03

NAME

Upgrade Human

DIRECTORY

`t03/`

SUBMIT

`index.html, css/style.css, js/script.js`

ALLOWED

`Object.*, Class.*, String.*, Array.*, Function.*, DOM, prompt()`

DESCRIPTION

Create a web page that visualizes the classes `Human` and `Superhero`. Display an image of a human, and underneath, list the properties of the human. Each method must be a button that calls this method. If a method requires a parameter, find a way to input that parameter (input box, prompt, etc.). Add a button `Turn into superhero`. On click of that button, display, in a similar way, the class `Superhero` instead of `Human`.

The class `Human` has the fields:

- firstName
- lastName
- gender
- age
- calories

and methods:

- `sleepFor()`: displays `I'm sleeping` for the number of seconds specified, and then `I'm awake now` after waking up
- `feed()`: displays `Nom nom nom` while eating for 10 seconds, and adds 200 to `calories`. If the user clicks the button when the human has more than 500 `calories`, displays `I'm not hungry`. If after eating, the amount of calories is less than 500, the page displays `I'm still hungry`

Note that the human loses 200 calories every minute. Make the human get hungry after 5 seconds after being created, and display a corresponding message.

You can add more properties and methods.

Create a class `Superhero` (it could represent Spider-Man/Thor/Iron Man/etc.) that extends the class `Human` and adds superhero-specific methods and properties.
The `Turn into superhero` button turns the human into a superhero, but on the condition that the human has more than 500 calories. After turning, the displayed properties and

methods must change accordingly.
Default methods:

- `fly()` : displays `I'm flying!` for 10 seconds
- `fightWithEvil()` : displays `Khhhh-chh... Bang-g-g-g... Evil is defeated!`

You can add more properties and methods. For example:

- can raise Thor's hammer
- shoot web

Apart from displaying relevant messages, visualize the methods in a more interesting way.

You can either use image files (and put them into `assets/images/*`, or use links to images online).

SEE ALSO

Class inheritance
Inheritance in JavaScript

Act: Task 04

NAME

Mixin

DIRECTORY

`t04/`

SUBMIT

`index.html, js/script.js, js/houseBuilder.js`

ALLOWED

`Object.*, String.*, Array.*`

DESCRIPTION

Create a mixin for the `house` object created by `HouseBuilder` (from the Task 00).

The mixin must manipulate the object's description property using the following methods:

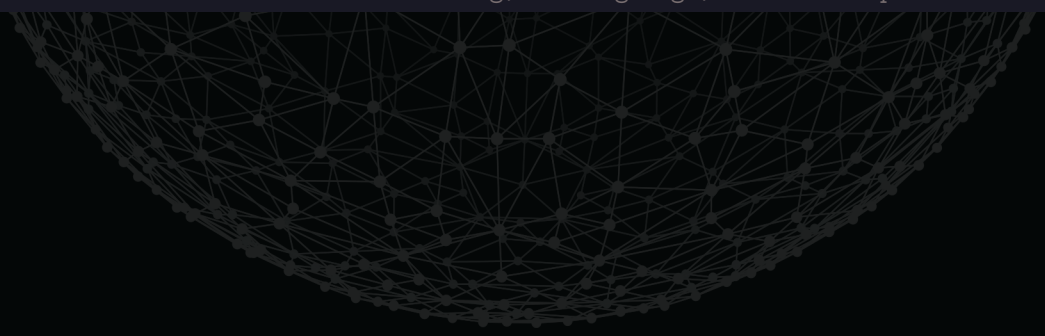
- `wordReplace()` - replace a specified word with another word
- `wordInsertAfter()` - insert a word after a specified word
- `wordDelete()` - delete a specified word
- `wordEncrypt()` - encrypt the text with a rot13 algorithm
- `wordDecrypt()` - decrypt the text with a rot13 algorithm

Your 'index.html' must include both your `houseMixin` script ('script.js'), and the `HouseBuilder` script. However, do not edit the `HouseBuilder` script, it must follow the rules from the Task 00.

See the EXAMPLE for examples of test cases.

EXAMPLE

```
const house = new HouseBuilder('88 Crescent Avenue',
    'Spacious town house with wood flooring, 2-car garage, and a back patio.',
    'J. Smith', 110, 5);

Object.assign(house, houseMixin);

console.log(house.getDaysToBuild());
// 220
console.log(house.description);
// Spacious town house with wood flooring, 2-car garage, and a back patio.

house.wordReplace("wood", "tile");
console.log(house.description);
// Spacious town house with tile flooring, 2-car garage, and a back patio.

house.wordDelete("town ");
console.log(house.description);
// Spacious house with tile flooring, 2-car garage, and a back patio.

house.wordInsertAfter ("with", "marble");
console.log(house.description);
// Spacious house with marble tile flooring, 2-car garage, and a back patio.

house.wordEncrypt();
console.log(house.description);
// Fcnpvbhf ubhfr jvgu zneoyr gvyr sybbevat, 2-pne tnentr, naq n onpx cngvb.

house.wordDecrypt();
console.log(house.description);
// Spacious house with marble tile flooring, 2-car garage, and a back patio.
```



Act: Task 05

NAME

Collections

DIRECTORY

`t05/`

SUBMIT

`index.html, js/script.js`

ALLOWED

`Set.*, Map.*, WeakSet.*, WeakMap.*`

DESCRIPTION

JS has various built-in objects for storing data, and all of them have their own particular use cases. In this task, you will implement four collections, each using one of the following objects: `Set`, `Map`, `WeakSet`, `WeakMap`.

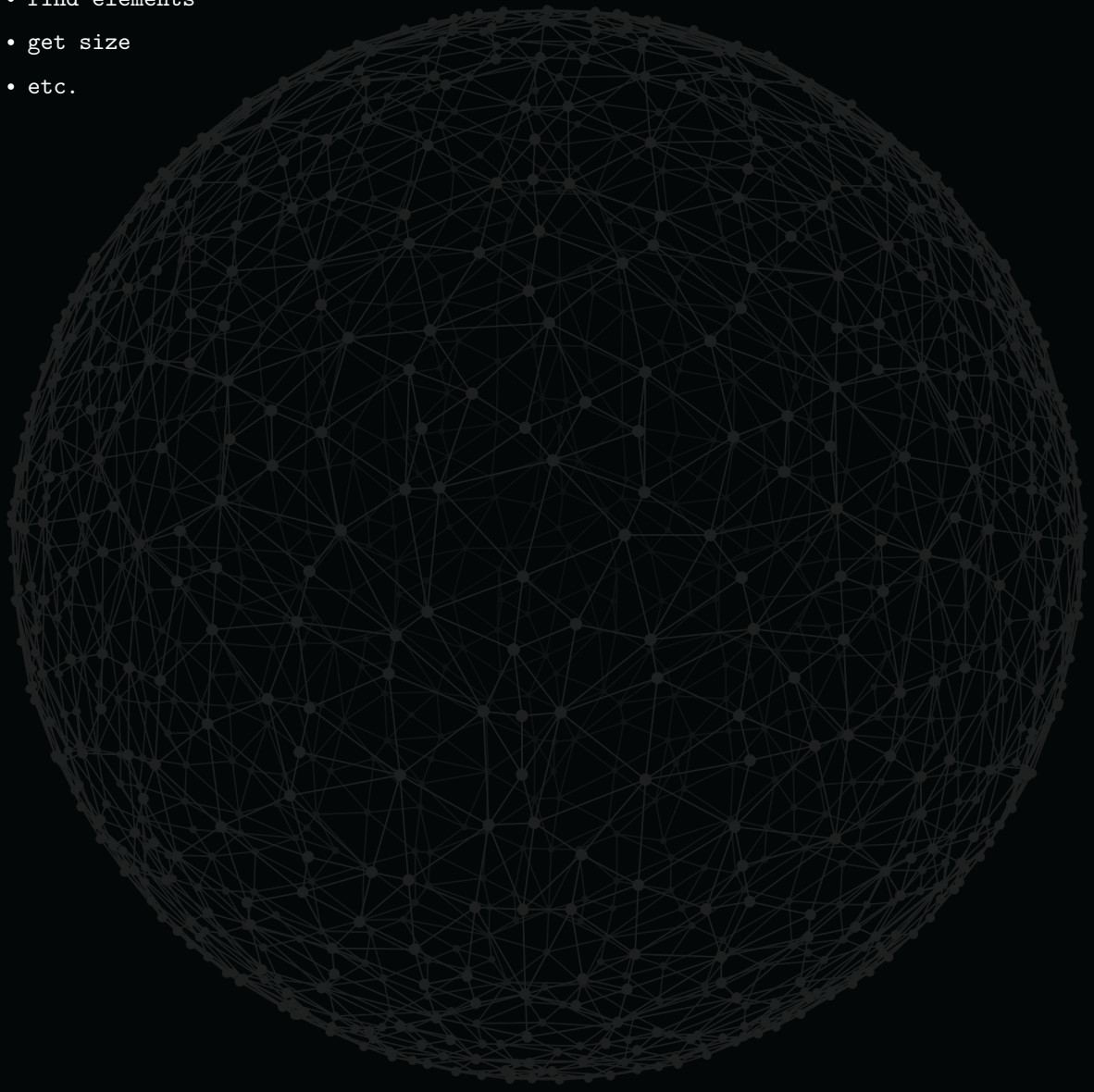
Below you will find 4 descriptions of collections and their functionality. Read the descriptions carefully to understand which object you need to use for which case.
Collections:

- `guestList`
  - contains names of invited guests
  - if you give it your name, it will tell you if you're on the list
  - if you ask who, or how many people are invited, it will not tell you
  - you can ask to remove someone by name off the list, but not everyone at once
- `menu`
  - contains a list of unique dishes and their prices
  - you can ask to name every available dish and its price, one after the other
- `bankVault`
  - contains safety deposit boxes, each with unique credentials and some contents
  - the only way to see the contents of a box, is to provide its correct credentials
- `coinCollection`
  - contains various coins, all unique
  - if you want, you can see the entire collection

After initializing the objects, add at least five thematic items to each, and use their methods to prove that they work as they should.

Write test cases in your script that demonstrate (if available for the particular object) how to

- add/remove elements
- clear the collection
- iterate over it
- find elements
- get size
- etc.



Act: Task 06

NAME

Step generator

DIRECTORY

`t06/`

SUBMIT

`index.html, js/script.js`

ALLOWED FUNCTIONS

`isNaN(), console.error(), console.log(), prompt(), Number.*`

DESCRIPTION

Create a generator. This generator will keep prompting you for input until it meets one of its exit conditions.

The prompt will mention the previous result, and ask for another number.
The previous result will start with the value 1. Once you've entered a second number, the previous result will now add this value to itself.

Here is an example of how the interaction works:

Prompt: `Previous result: 1. Enter a new number:`
User enters 23
Prompt: `Previous result: 24. Enter a new number:`
User enters 5
Prompt: `Previous result: 29. Enter a new number:`

The generator must display `Invalid number!` in the console if the entered value is not a number.

If the value received from the generator is more than 10000, then it starts from 1.

Your 'index.html' must include the 'script.js' file.

SEE ALSO

`function*`

Act: Task 07

NAME

LinkedList

DIRECTORY

`t07/`

SUBMIT

`index.html, js/script.js`

ALLOWED

`Object.*, Class.*, String.*, Array.*`

DESCRIPTION

Create a class for a LinkedList data structure.
Properties of the list elements:

- `data`
- `next` - reference for the next element or null

Methods of the LinkedList class:

- `add(value)` - insert the value to the end of the list
- `remove(value)` - delete the first element equal to the value
- `contains(value)` - returns true if the list contains that value
- `[Symbol.iterator]` - values of elements
- `clear()` - clear the list
- `count()` - return the length of the list
- `log()` - print into the console the values of elements separated by a ","

Write a function `createLinkedList(arr)`, that takes an array of numbers and converts it to a LinkedList with elements from the array. The function returns a LinkedList.

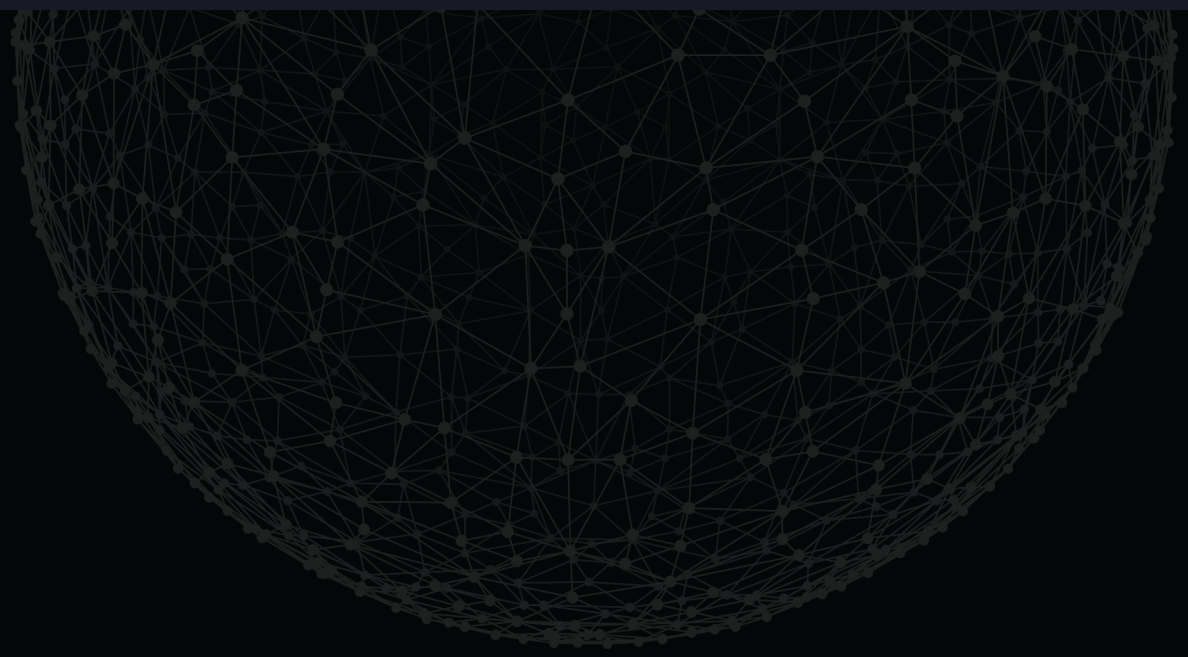
Your 'index.html' must include the 'script.js' file. See the EXAMPLE for a test case.

SEE ALSO

Demystifying ES6 Iterables & Iterators

EXAMPLE

```
const ll = createLinkedList([100, 1, 2, 3, 100, 4, 5, 100]);
ll.log();
// "100, 1, 2, 3, 100, 4, 5, 100"
while(ll.remove(100));
ll.log();
// "1, 2, 3, 4, 5"
ll.add(10);
ll.log();
// "1, 2, 3, 4, 5, 10"
console.log(ll.contains(10));
// "true"
let sum = 0;
for (const n of ll) {
  sum += n;
}
console.log(sum);
// "25"
ll.clear();
ll.log();
// ""
```



Act: Task 08

NAME

Export LinkedList

DIRECTORY

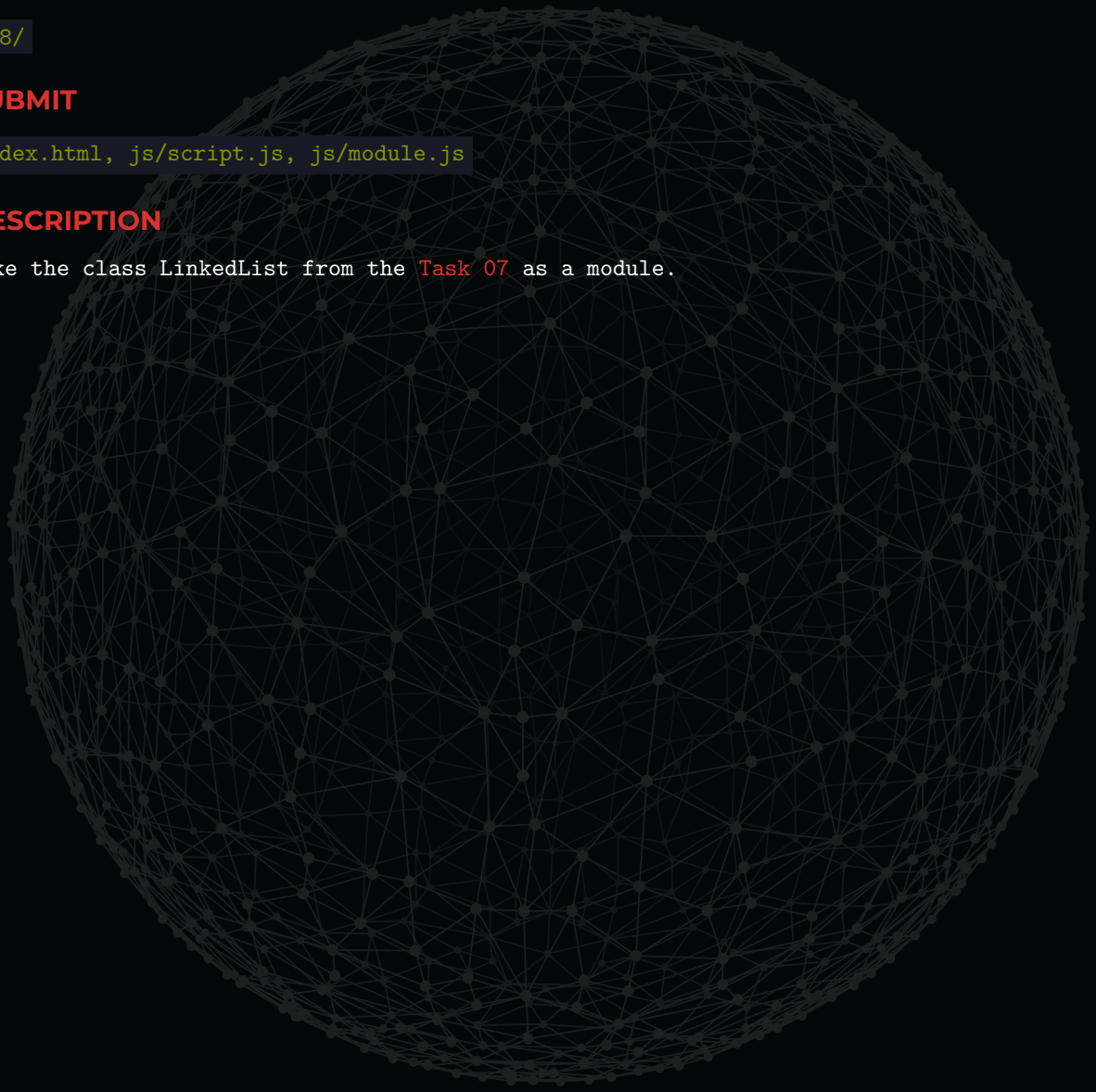
`t08/`

SUBMIT

`index.html, js/script.js, js/module.js`

DESCRIPTION

Make the class LinkedList from the Task 07 as a module.



Act: Task 09

NAME

Hidden Proxy

DIRECTORY

`t09/`

SUBMIT

`index.html, js/script.js`

ALLOWED

`Proxy, Object.*, Number.*`

DESCRIPTION

Create a proxy named `validator` for an object `person`. The validator intercepts the access to the person's properties, and has `get` and `set` traps.

On attempts to get a property's value, the validator displays `Trying to access the property '[property name]'...` in the console.
The validator returns `false` if the property does not exist.

If the user tries to set a new value for a property - the validator displays `Setting value '[value]' to '[property name]'` in the console.

Handle the ability to change the value of the person's age. If the value passed to age is not an integer - throw a corresponding exception and display `The age is not an integer`.

If the age is not in the range `0...200` - throw a corresponding exception and print `The age is invalid`.

EXAMPLE

```
let person = new Proxy({}, validator);

person.age = 100;
// Setting value '100' to 'age'
console.log(person.age);
// Trying to access the property 'age'...
// 100
person.gender = "male";
// Setting value 'male' to 'gender'
person.age = 'young';
// Uncaught TypeError: The age is not an integer
person.age = 300;
// Uncaught RangeError: The age is invalid
```

Share

PUBLISHING

Last but not least, the final stage of your work is to publish it. This allows you to share your challenges, solutions, and reflections with local and global audiences. During this stage, you will discover ways of getting external evaluation and feedback on your work. As a result, you will get the most out of the challenge, and get a better understanding of both your achievements and missteps.

To share your work, you can create:

- a text post, as a summary of your reflection
- charts, infographics or other ways to visualize your information
- a video, either of your work, or a reflection video
- an audio podcast. Record a story about your experience
- a photo report with a small post

Helpful tools:

- Canva - a good way to visualize your data
- QuickTime - an easy way to capture your screen, record video or audio

Examples of ways to share your experience:

- Facebook - create and share a post that will inspire your friends
- YouTube - upload an exciting video
- GitHub - share and describe your solution
- Telegraph - create a post that you can easily share on Telegram
- Instagram - share photos and stories from ucode. Don't forget to tag us :)

Share what you've learned and accomplished with your local community and the world. Use #ucode and #CBLWorld on social media.