
Sprint 03
Marathon Full Stack
April 27, 2021
ucode

Contents

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.

Sprint 03 | Marathon Full Stack **> 2**

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 is mixin, and what makes it useful?
- What are JS modules?
- What are encapsulation, polymorphism, and inheritance?
- What is a class parameter, and what is a class method?
- What is a class constructor?

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 here.
- Explore object-oriented design principles for writing clean code here.
- 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.

ucode

Sprint 03 | Marathon Full Stack **> 3**

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.
- All tasks are divided into Act Basic and Act Advanced. You need to complete all basic tasks to validate the Sprint. But to achieve maximum points and more knowledge, consider accomplishing advanced tasks also.
- 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.
- Your work may also be graded by your mentor. So, be ready for that.
- Also, the challenge will pass automatic evaluation which is called `Oracle`.
- If you have any questions or don't understand something, ask other students or just Google it.

Sprint 03 | Marathon Full Stack **> 4**

Act Basic: Task 00

NAME

Hard worker

DIRECTORY

`t00_hard_worker/`

SUBMIT

`index.html, js/hard-worker.js`

ALLOWED

JS

LEGEND

I can't have this, any of this. There is no place on Earth I can go where I'm not a monster.

BEFORE YOU BEGIN

Here starts your great journey in object-oriented programming. Before working on real tasks, it is necessary to understand the concept of classes and objects.

In JS, Classes are templates for JS Objects. Firstly, you "describe" a class and then you can create multiple objects of this class. This scheme helps us to abstract away and think about a huge program as a system of objects, which interact with each other.

The properties of class objects are described by the fields of the class, and objects do their work using methods.

Your 'index.html' must include the 'hard-worker.js' file.

DESCRIPTION

Create a class `HardWorker` with properties `name`, `age`, `salary`. `age` and `salary` mustn't be able to take values outside of the following ranges:

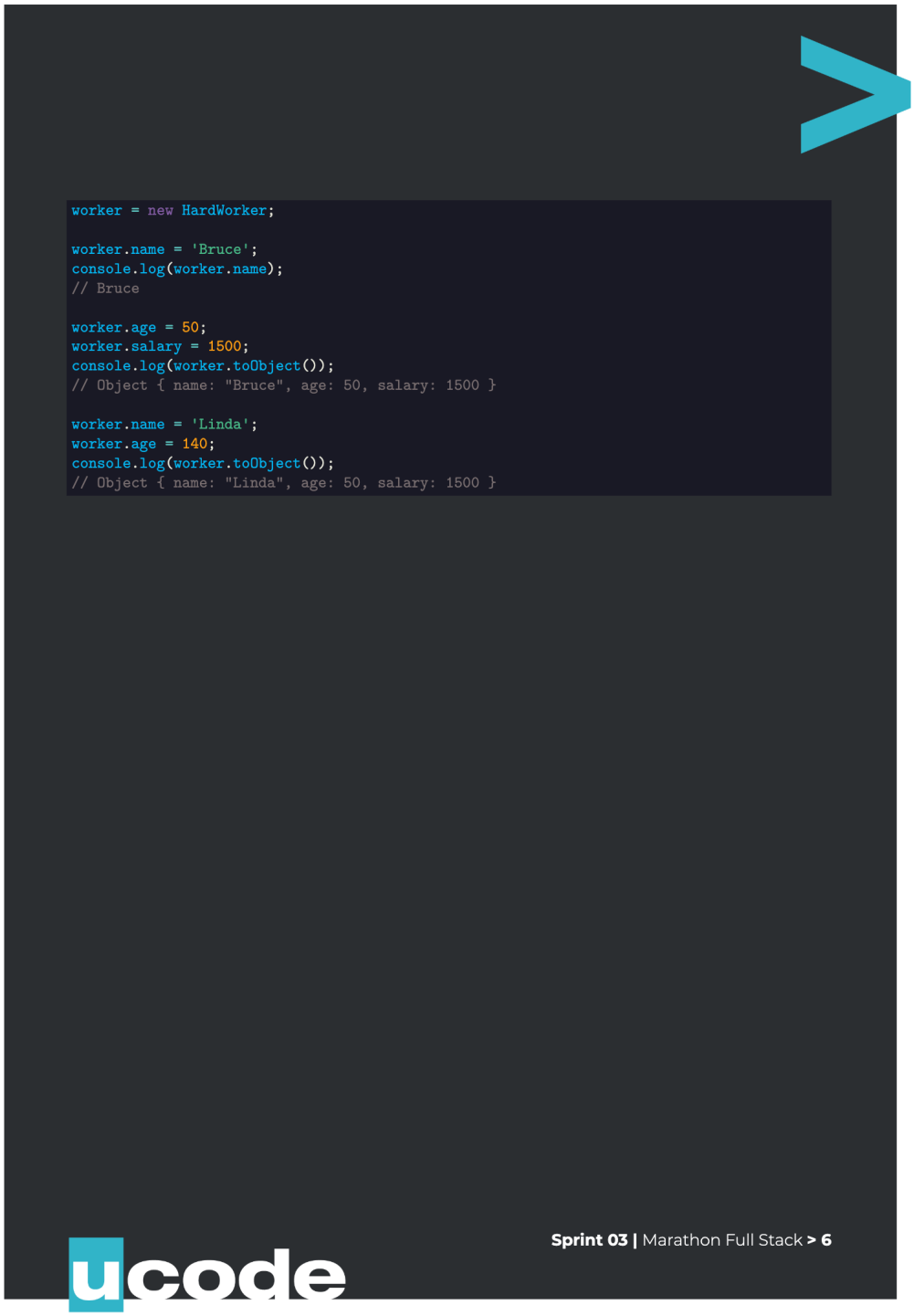
- 1 <= `age` < 100
- 100 <= `salary` < 10000

Add a `toObject()` method that returns all the properties.
See the EXAMPLE section for a script that can test your code.

EXAMPLE

```
/*
  Task name: Hard worker
*/
```

Sprint 03 | Marathon Full Stack **> 5**


worker = new HardWorker;
worker.name = 'Bruce';
console.log(worker.name);
// Bruce
worker.age = 50;
worker.salary = 1500;
console.log(worker.toObject());
// Object { name: "Bruce", age: 50, salary: 1500 }
worker.name = 'Linda';
worker.age = 140;
console.log(worker.toObject());
// Object { name: "Linda", age: 50, salary: 1500 }
ucode
Sprint 03 | Marathon Full Stack > 6

Act Basic: Task 01

NAME

Tower

DIRECTORY

`t01_tower/`

SUBMIT

`index.html, js/tower.js`

ALLOWED

JS

BEFORE YOU BEGIN

It's time to see in practise one cool OOP feature - inheritance. It is a mechanism where you can to derive a class from another class for a hierarchy of classes that share a set of attributes and methods. So if one class extends another one, it gets properties of a parent class.

DESCRIPTION

Create a `Tower` class that inherits from the `Building` class (see in the resources). Add the following properties:

- property `hasElevator`
- property `arcCapacity`
- property `height`
- method `getFloorHeight()`

Don't forget to update the parent `toString()` method.
See the EXAMPLE section for a script that can test your code.

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

EXAMPLE

```
/*
  Task name: Tower
*/

const starkTower = new Tower(93, 'Different', 'Manhattan, NY');
starkTower.hasElevator = true;
starkTower.arcCapacity = 70;
starkTower.height = 1130;
console.log(starkTower.toString());
/*
```

Sprint 03 | Marathon Full Stack **> 7**


Floors: 93
Material: Different
Address: Manhattan, NY
Elevator: +
Arc reactor capacity: 70
Height: 1130
Floor height: 12.150537634408602
*/
ucode
Sprint 03 | Marathon Full Stack > 8

Act Basic: Task 02

NAME

Prototype

DIRECTORY

`t02_prototype/`

SUBMIT

`index.html, js/houseBuilder.js`

ALLOWED

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

BEFORE YOU BEGIN

You need to understand what is Object prototypes.

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 building speed is 0.5 m^2 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.

ucode

Sprint 03 | Marathon Full Stack **> 9**

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._building_speed);
// 0.5

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

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

SEE ALSO

Resources for developers, by developers.
Object prototypes

Sprint 03 | Marathon Full Stack **> 10**

Act Basic: Task 03

NAME

Prototype function

DIRECTORY

`t03_prototype_function/`

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);
// . ?
```

ucode

Sprint 03 | Marathon Full Stack > **11**

Act Basic: Task 04

NAME

Collections

DIRECTORY

`t04_collections/`

SUBMIT

`index.html, js/script.js`

ALLOWED

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

BEFORE YOU BEGIN

You need to understand what are Keyed collections.
In what cases can they be useful?

DESCRIPTION

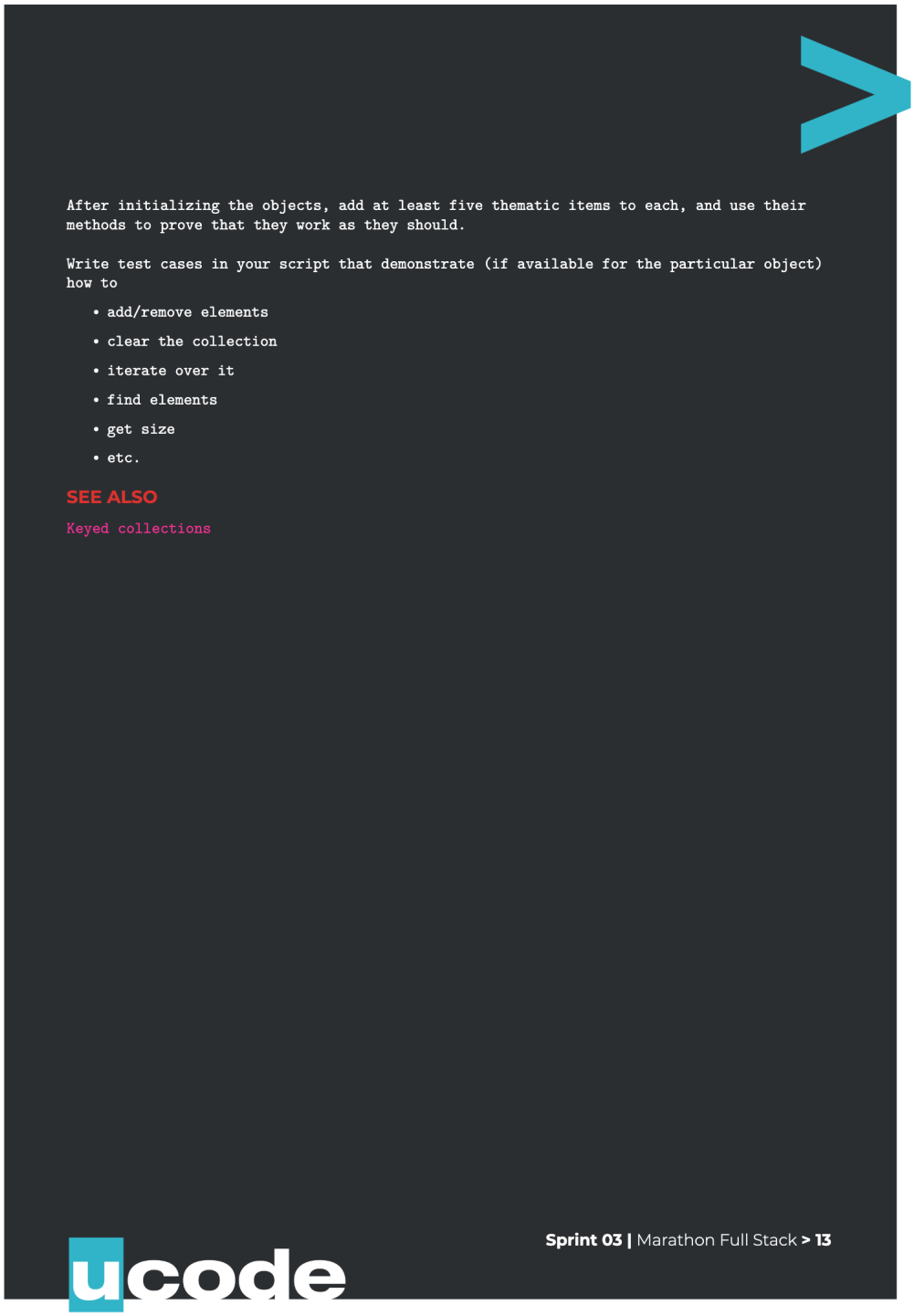
JS has various built-in objects for storing data, and all of them have their own particular use cases. In this task, you will implement four collections, each using one of the following objects: `Set`, `Map`, `WeakSet`, `WeakMap`.

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

ucode

Sprint 03 | Marathon Full Stack **> 12**



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.

SEE ALSO

Keyed collections

Act Basic: Task 05

NAME

Export classes

DIRECTORY

`t05_export_classes/`

SUBMIT

`index.html, js/main.js, js/modules/hard-worker.js`

BEFORE YOU BEGIN

You need to understand what are JavaScript modules.
You can find basic module example here. Figure out how it works.
In what cases using modules can be useful?
Also you can find other module examples here.

DESCRIPTION

Make the class `HardWorker` from the Task 00 as a module.

SEE ALSO

JavaScript modules
basic-modules

Sprint 03 | Marathon Full Stack **> 14**

Act Advanced: Task 06

NAME

Hidden Proxy

DIRECTORY

`t06_hidden_proxy/`

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`.

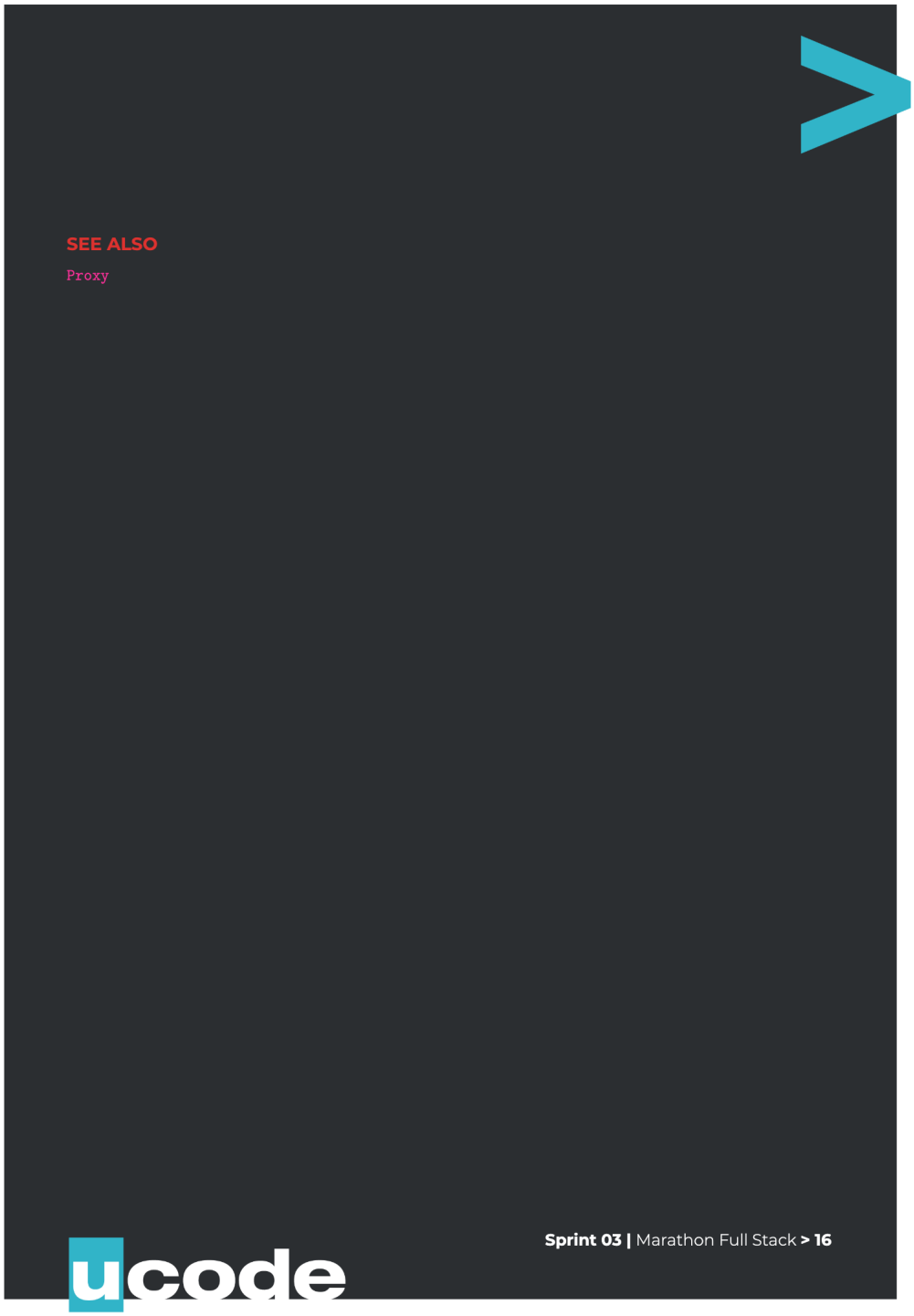
EXAMPLE

```
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
```

ucode

Sprint 03 | Marathon Full Stack **> 15**


SEE ALSO
Proxy
ucode
Sprint 03 | Marathon Full Stack > 16

Act Advanced: Task 07

NAME

Timer

DIRECTORY

`t07_timer/`

SUBMIT

`index.html, js/script.js`

ALLOWED

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

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.

Sprint 03 | Marathon Full Stack **> 17**

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
*/
```

SEE ALSO

Timeouts and intervals

Act Advanced: Task 08

NAME

Mixin

DIRECTORY

`t08_mixin/`

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 02).

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.

Sprint 03 | Marathon Full Stack **> 19**

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.
```

SEE ALSO

Mixins

ucode

Sprint 03 | Marathon Full Stack **> 20**

Act Advanced: Task 09

NAME

Step generator

DIRECTORY

`t09_step_generator/`

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:

The prompt: `Previous result: 1. Enter a new number:`
The user enters 23
The prompt: `Previous result: 24. Enter a new number:`
The user enters 5
The 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*`

ucode

Sprint 03 | Marathon Full Stack **> 21**

Act Advanced: Task 10

NAME

Linked List

DIRECTORY

`t10_linked_list/`

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

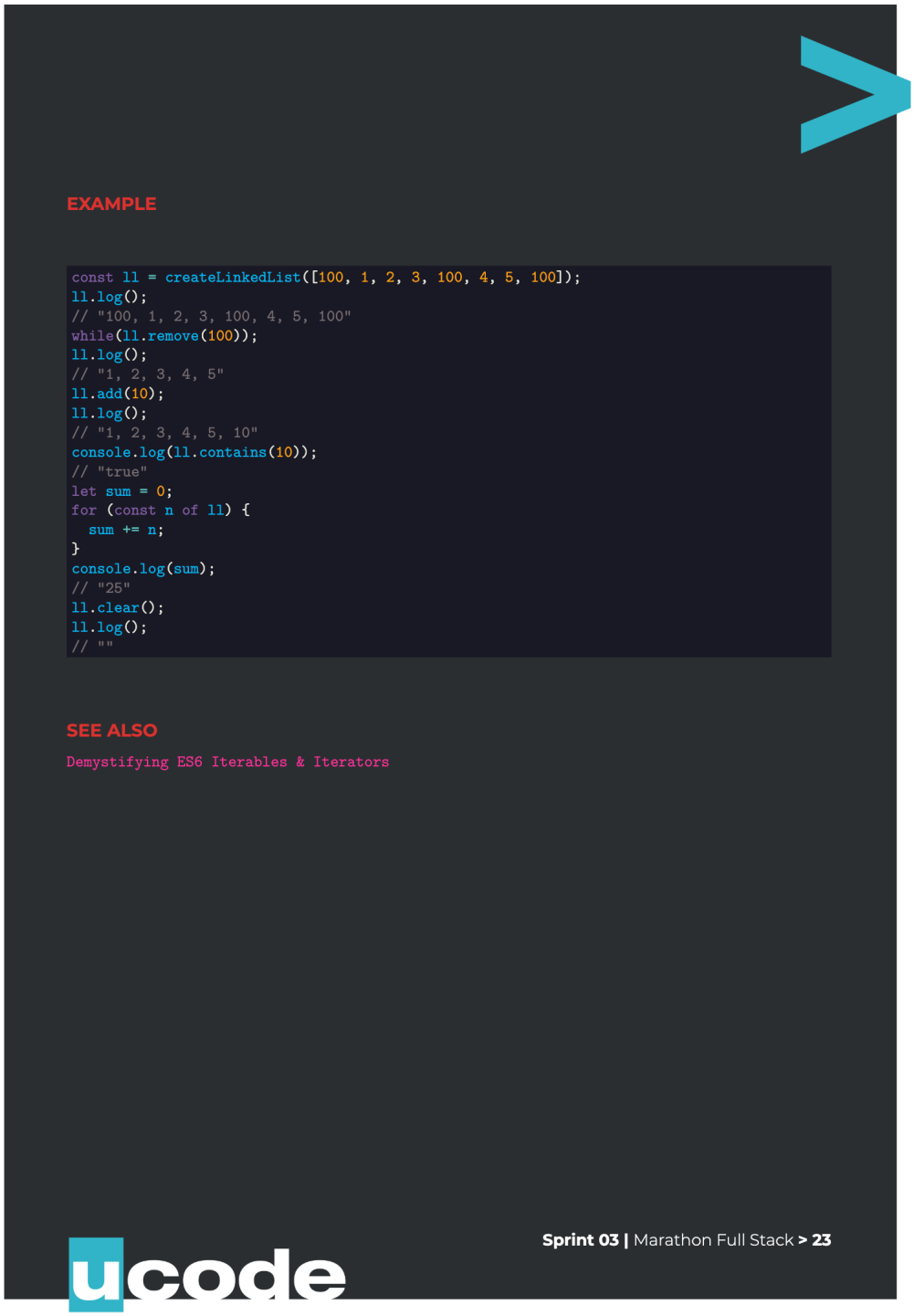
Methods of the LinkedList class:

- `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.

Sprint 03 | Marathon Full Stack **> 22**



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();
// ""
```

SEE ALSO

Demystifying ES6 Iterables & Iterators

Act Advanced: Task 11

NAME

Upgrade Human

DIRECTORY

`t11_upgrade_human/`

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:

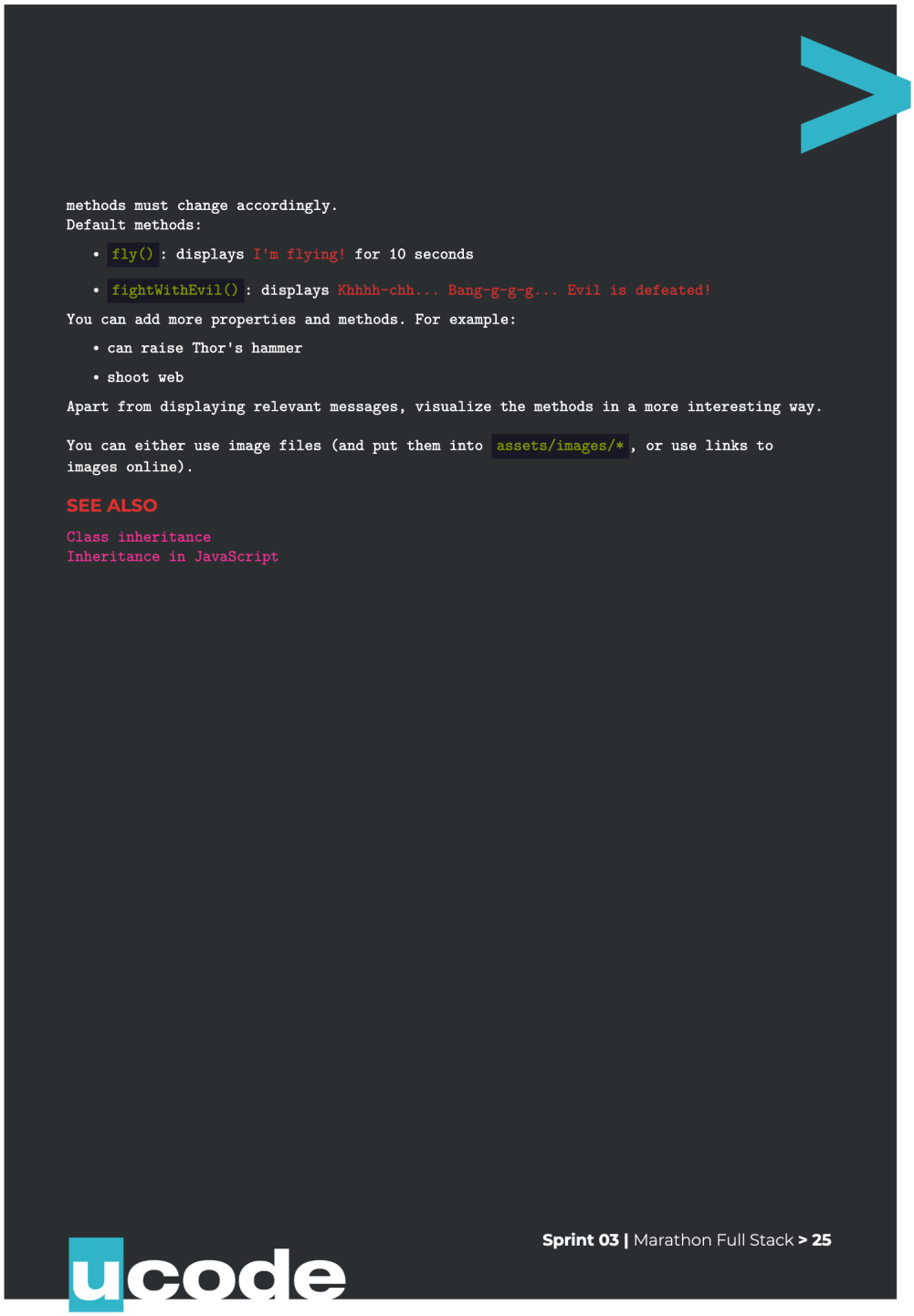
- `sleepFor()`: displays `I'm sleeping` for the number of seconds specified, and then `I'm awake now` after waking up
- `feed()`: displays `Nom nom nom` while eating for 10 seconds, and adds 200 to `calories`. If the user clicks the button when the human has more than 500 `calories`, displays `I'm not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

Sprint 03 | Marathon Full Stack **> 24**



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

ucode

Sprint 03 | Marathon Full Stack **> 25**

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 (macOS)
- ScreenToGif - screen, webcam, and sketchboard recorder with an integrated editor (Windows)

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.

Sprint 03 | Marathon Full Stack **> 26**