General information

- The course has four compulsory laboratory exercises.
- You are to work in groups of two people. Sign up for the labs at `http://sam.cs.lth.se/Labs`
- The labs are mostly homework. Before each lab session, you must have done all the assignments in the lab, written and tested the programs, and so on. Contact a teacher if you have problems solving the assignments.
- Extra labs are organized only for students who cannot attend a lab because of illness. Notify Per Andersson (`Per.Andersson@cs.lth.se`) if you fall ill, before the lab.
- To pass the labs you need to show that you have achieved the learning outcomes of the lab. A correct and complete program is less important, but if you have many bugs in your program you will have a hard time convincing the TA that you have achieved the learning outcomes.

Laboratory Exercise 1

The first lab is about the JavaScript language, learning outcomes:

1. Get familiar with JavaScript.
2. Understanding how prototype based object orientation in JavaScript works.
3. Get some experience using functional style of programming.
4. Get familiar with Node.js.
5. Develop data structures and functions to be used in later labs.

Background

Later in the course you will develop a web application for orders in a salad bar, similar to Grönt o' Gott at the LTH campus. The customer composes their own salads from a selection of ingredients. Each salad is composed of one foundation, one protein, a selection of extras, and one dressing. For example, a Caesar salad is composed of: lettuce, chicken breast, bacon, croutons, Parmesan cheese, and Caesar dressing.

In addition to handling salad composition, the application should also provide additional information about the salad, for example the price and if it contains ingredients that could cause an allergic reaction.
All ingredients will be imported from a CommonJS module named `inventory.js`. It looks like this:

```
exports.inventory = {
  Sallad: {price: 10, foundation: true, vegan: true},
  'Norsk fjordlax': {price: 30, protein: true},
  Krutonger: {price: 5, extra: true},
  Caesardressing: {price: 5, dressing: true},
  /* more ingredients */
};
```

The properties `foundation`, `protein`, `extra`, and `dressing` indicate which part of the salad the ingredient is to be used for. All ingredients also have a `price` and may also have properties `vegan`, `gluten` and `lactose`.
Reflection question 1: In most programming languages a complete record for each ingredient would be used, for example: `Sallad: {price: 10, foundation: true, protein: false, extra: false, dressing: false, vegan: true, gluten: false, lactose: false}` This is not the case in `inventory.js`, which is common when writing JavaScript code. Why don't we need to store properties with the value `false` in the JavaScript objects?

Node.js

In this lab you will use Node.js as execution environment. The tool is installed on the Linux computers at LTH. You can also install it on your own computer, see `https://nodejs.org/`. You start Node.js from the terminal with the command: `node`. If you do not provide any arguments, you will start the REPL (Read-Eval-Print-Loop). Write `.exit` to quit the REPL, see `https://nodejs.org/api/repl.html`. This is great for testing stuff, but it is a good idea to save the code for the labs in a file. To execute the JavaScript code in a file, you simply give the file name as argument to `node`:

```
node lab1.js
```

Please add additional printouts, to make it clear which text belongs to which task. There is a template file, `lab1.js`, in Canvas that does this for you, so you do not need to copy code from this pdf. Node.js does not support ECMAScript modules, so we will use CommonJS modules instead. Try the following code (you need to download ./ingredients.js from Canvas or GitHub first):

```
const imported = require("./inventory.js");
console.log(imported.inventory);
```

Have you forgotten about the terminal? Check out the introduction from LTH `https://www.lth.se/fileadmin/ddg/text/unix-x.pdf`.

IDE

Do you want to use an IDE when writing code? I recommend Visual Studio Code, see `https://code.visualstudio.com`. Check out their tutorial on running and debugging JavaScript programs using node.js (skip the "An Express application" part), see `https://code.visualstudio.com/docs/nodejs/nodejs-tutorial`. There is also a video showing how to debug JavaScript code here `https://www.youtube.com/watch?v=2oFKNL7vYV8`. It has great support for JavaScript and TypeScript. We will use TypeScript later in the course which Eclipse has poor support for. TypeScript is JavaScript extended with static typing.

Assignments

1. Study the relevant material for lecture 1-2, see the reading instructions for lecture-1-2 in Canvas.

2. Set up the project: Create a directory and add a new file named `lab1.js` containing the code below (or download it from Canvas). You also need to download `inventory.js`.

   ```
   'use strict';
   const imported = require("./inventory.js");
   console.log(imported.inventory['Sallad']);
   ```

3. In the `inventory.js` file you can find all data for composing a salad. Its structure is good for looking up properties of the ingredients, e.g. `imported.inventory['Krutonger']` for looking up properties of krutonger. However, it might not be ideal for presenting the options to the customers, where you want to present foundations, proteins, extras and dressings in separate boxes. Fortunately, using functional programming style, function chaining and the functions from `Array.prototype`, you can easily transform the data structure to match your needs. The documentation of the functions can be found at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array`. To get started, let's print the names of all ingredients:

```
const names = Object.keys(imported.inventory);
names.forEach(name => console.log(name));
```

In this case, you will get the same result with:

```
for (const name in imported.inventory) {
  console.log(name);
}
```

Reflection question 2: When will the two examples above give different outputs, and why is inherited functions, such as `sort()`, not printed? Hint: read about enumerable properties and own properties.

The for loop might seem to be simpler code, but using arrays have advantages. One advantage is the ease of add additional data transformations. Let's sort the array before printing:

```
names
.sort((a, b) => a.localeCompare(b, "sv", {sensitivity: 'case'}))
.forEach(name => console.log(name));
```

This is an example of function chaining. Each function in the chain returns a collection, and you can easily add additional functions to the chain without storing the intermediate result in a local variable. This is the same principle as used with streams, which we will use later in this course. This is very convenient when generating data-dependent content of web pages.

Assignment 1: Write a function that returns a string containing the HTML code for the options of a select box, that has all salad ingredients with a certain property. Example: `makeOptions(imported.inventory, 'foundation')` returns

```
<option value="Pasta"> Pasta, 10 kr</option>
<option value="Sallad"> Sallad, 10 kr</option> ...
```

Hint: Use the functions `Array.prototype.filter()`, `Array.prototype.map()` and `Array.prototype.reduce()`.

4. We need a representation for a salad. Create a JavaScript class named `Salad` for that. You need to store the `foundation`, `protein`, `extras`, and `dressing`. Later on, salad objects will be passed to different components in the web app and to avoid having to pass along the inventory to all components, the salad object should itself also store copies of the properties of the ingredients in the salad. Use one object as a dictionary to store the ingredients, see the printout bellow.

Assignment 2: Create a Salad class. You may use the ECMAScript 2015 `class` syntax, or the backwards compatible constructor function for this and the remaining assignments

(except in assignment 3). The class must contain the following methods:

```
class Salad {
  constructor();
  add(name, properties);  // return this object to make it chainable
  remove(name);           // return this object to make it chainable
}
```

Create an object for a Caesar salad:

```
let myCaesarSalad = new Salad()
.add('Sallad', imported.inventory['Sallad'])
.add('Kycklingfilé', imported.inventory['Kycklingfilé'])
.add('Bacon', imported.inventory['Bacon'])
.add('Krutonger', imported.inventory['Krutonger'])
.add('Parmesan', imported.inventory['Parmesan'])
.add('Ceasardressing', imported.inventory['Ceasardressing'])
.add('Gurka', imported.inventory['Gurka']);
console.log(JSON.stringify(myCaesarSalad) + '\n');
myCaesarSalad.remove('Gurka');
console.log(JSON.stringify(myCaesarSalad) + '\n');
```

This is my printout for the final salad:

```
{"ingridients":{
    "Sallad" : {"price" : 10, "foundation" : true, "vegan" : true},
    "Kycklingfilé": {"price" : 10, "protein" : true},
    "Bacon" : {"price" : 10, "extra" : true},
    "Krutonger" : {"price" : 5, "extra" : true , "gluten" : true},
    "Parmesan" : {"price" : 5,"extra" : true, "lactose" : true},
    "Ceasardressing" : {"price" : 5, "dressing" : true, "lactose" : true},
}
```

5. Assignment 3: Next task is to update the `Salad` class with two more functions. For this exercise, you must do this by modifying the existing class. That is, add the functions to the existing `Salad`'s prototype object, and not by modifying the code inside `class Salad{ ... }`.

   - Add a function, `getPrice()`, to calculate the price. The price is simply the sum of the prices of all ingredients. The computation should be done using functional style, i.e. using `Array.prototype.reduce` etc.
   - Also, add a second function, `count(property)`, that counts the number of ingredients with a given property. This can be used to check if a salad is well-formed (for example, that it has exactly one foundation and at least three extras).
     Hint: `Object.values()`.

   Test your code:

   ```
   console.log('En ceasarsallad kostar ' + myCaesarSalad.getPrice() + ' kr');
   console.log('En ceasarsallad har ' + myCaesarSalad.count('lactose') +
               ' ingredienser med laktos');
   console.log('En ceasarsallad har ' + myCaesarSalad.count('extra') + ' tillbehör');

   // En ceasarsallad kostar 45kr
   // En ceasarsallad har 2 ingredienser med laktos
   // En ceasarsallad har 3 tillbehör
   ```

   Reflection question 3: How are classes and inherited properties represented in JavaScript?

Let's explore this by checking some types. What is the type and value of: `Salad`, `Salad.prototype`, `Salad.prototype.prototype`, `myCaesarSalad` and `myCaesarSalad.prototype`?
Hint: `console.log('typeof Salad: ' + typeof Salad);`

What is the difference between an object's prototype chain and having a `prototype` property? Which objects have a `prototype` property? How do you get the next object in the prototype chain? Also try:

```
console.log('check 1: ' +
  (Salad.prototype === Object.getPrototypeOf(Salad)));
console.log('check 2: ' +
  (Salad.prototype === Object.getPrototypeOf(myCaesarSalad)));
console.log('check 3: ' +
  (Object.prototype === Object.getPrototypeOf(Salad.prototype)));
```

6. The Salad class currently creates an empty salad. However, sometimes you want to copy another salad.
Assignment 4: Implement this functionality by adding a parameter to the constructor. There are two representations of a salad you must support: a `Salad` object, and a string containing a JSON representation of a `Salad`.
Hint 1: JavaScript do not support function overloading, you can not have both `constructor()` and `constructor(arg)` in the same class. This is not a problem since `constructor(arg)` can be called without arguments. `new Salad()` will not generate any error, `arg` will have the value `undefined`.
Hint 2: use `typeof` to distinguish between sting and Salad object values. `instanceof` is also relevant.
Hint 3: `JSON.parse()` will return an object which is not an instance of `Salad`. All methods are missing.
Hint 4: Use the spread operator in combination with object literals to copy objects.

```
const objectCopy = new Salad(myCaesarSalad);
const json = JSON.stringify(myCaesarSalad);
const jsonCopy = new Salad(json);
console.log('original object\n' + JSON.stringify(myCaesarSalad));
console.log('copy of object\n' + JSON.stringify(objectCopy));
console.log('copy from json\n' + JSON.stringify(jsonCopy));
jsonCopy.add('Gurka', imported.inventory['Gurka']);
console.log('originalet kostar kostar ' + myCaesarSalad.getPrice() + ' kr');
console.log('med gurka kostar den ' + jsonCopy.getPrice() + ' kr');
```

7. One limitation with the Salad class is that you can only have a fixed amount of each ingredient. What happens if a customer wants extra Parmesan?
Assignment 5: Create a new class, `GourmetSalad`, which extends Salad to support this. In a `GourmetSalad` the customer can specify the size of each ingredient when adding it to the salad as an optional third parameter. You can add the same ingredient several times to the same salad. For each addition, add the amount to any previous amount already in the salad. The price is adjusted linearly, so if you add 1.5 portion Parmesan it costs 1.5 times the price of Parmesan. The size should be stored among the other properties of the ingredient. `GourmetSalad.add(ingredient, props, size)` must add a size property to `props` and use the `Salad.prototype.add` function (`super.add(name, propertiesWithSize)`).
Note: You should not modify the `imported.inventory` object. Make sure you copy any object before modifying it. If you forget this you will get a run-time error since inventory is

read only, see deepFreeze() at the bottom of `inventory.js`.
Here is a test case:

```
let myGourmetSalad = new GourmetSalad()
.add('Sallad', imported.inventory['Sallad'], 0.5)
.add('Kycklingfilé', imported.inventory['Kycklingfilé'], 2)
.add('Bacon', imported.inventory['Bacon'], 0.5)
.add('Krutonger', imported.inventory['Krutonger'])
.add('Parmesan', imported.inventory['Parmesan'], 2)
.add('Ceasardressing', imported.inventory['Ceasardressing']);
console.log('Min gourmetsallad med lite bacon kostar '
    + myGourmetSalad.getPrice() + ' kr');
myGourmetSalad.add('Bacon', imported.inventory['Bacon'], 1)
console.log('Med extra bacon kostar den '
    + myGourmetSalad.getPrice() + ' kr');

// Min gourmetsallad med lite bacon kostar 50 kr
// Med extra bacon kostar den 60 kr
```

8. In the coming labs you will use `Salad` objects in a web application. Sometimes you want to refer to the object from the HTML code; For example, a remove button on the shopping cart page must be able to identify. HTML is text only, so for this purpose you want a string identifier for the object. One way to accomplish this is to add a unique identifier to each `Salad` object. Then you can refer to a specific object from the HTML code using this id.
Assignment 6: Use a static instance counter and add the following to the `Salad` constructor:

```
this.id = 'salad_' + Salad.instanceCounter++;
```

Test it:

```
console.log('Min sallad har id: ' + myGourmetSalad.id);
\\ Min gourmetsallad har id: salad_1
```

Reflection question 4: In which object are static properties stored?
Reflection question 5: Can you make the `id` property read only?
Reflection question 6: Can properties be private?

9. The use of a static counter to generate identifiers only works during a session. Every time the program is restarted the counter is reset. The identifiers might not be unique if you use a persistent storage, for example a database, for salad objects. You will experience in lab 4. A better way to generate identifiers is to use universally unique identifiers (UUID) as specified in RFC4122. There is a npm package that implements the RFC: `https://www.npmjs.com/package/uuid`, making it easy to use UUIDs in your program. In the terminal (make sure you are in the same directory as your `lab1.js` file):

```
npm install uuid
```

This will download the source code and place it in the directory `node_modules`. Now you can use it in your program:

```
const { v4: uuidv4 } = require('uuid');
const uuid = uuidv4();  // use this in the constructor
```

Assignment 7: Add a UUID to the Salad class. Make sure the copy constructor still works.

If you copy an object you want a new UUID, so you do not end up with two different salads with the same UUID. However, when parsing a JSON representation of a salad, you can assume that it comes from a persistent storage, and you want to keep the UUID.

Extra assignments, if you have time.

1. Create an object to manage an order. Example of functions needed: create an empty shopping basket, add and remove a salad, calculate the total price for all salads in the shopping basket.

This concludes all assignments for lab 1. In Lab 2 you will develop your first React application. Lab 2 will take significantly longer time compared to lab 1.

Editor: Per Andersson

Contributors in alphabetical order:
Alfred Åkesson
Kevlanche
Oscar Ammkjaer
Per Andersson

Home: `https://cs.lth.se/edaf90`

Repo: `https://github.com/lunduniversity/webprog`

This compendium is on-going work.
Contributions are welcome!
Contact: `per.andersson@cs.lth.se`