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.

Laboratory Exercise 1

The first lab is about the JavaScript language, *objectives*:

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 ordering 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 or more proteins, a selection of extras, and one dressing. For example, a Caesar salad is composed of: chicken breast, bacon, croutons, lettuce, 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`, `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 fale 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, but you need to run initcs to add it to the path. 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
```

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 the course website or github first):

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

Have you forgotten about the terminal? Check out the introduction from LTH http://www.ddg.lth.se/perf/unix/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. If you are using the linux system at LTH, remember to run initcs to add node to the path.

3. Set up the project: Create a directory and add a new file named lab1.js containing the following code:

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

4. 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, i.e. imported.inventory['Krutonger']. You might think that this structure is not good for presenting the options for the customers,

where you want to present foundations, proteins extras, and dressings in separate boxes. However, 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/forEach`. To get started, let's print the names of all ingredients:

```
let 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: enumerable, nonenumerable, and inherited 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. Lets 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 the 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 the options HTML code of a select box for a part of a salad. Example: `makeOptions(imported.inventory, 'foundation')` returns

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

Hint: Use the functions `map()` and `reduce()`.
Note: You will use this code in lab 2 to render a compose salad form. However then we will use JSX, reacts domain specific language for rendering pages. The code will be the same with the exception for the expression that generate the text.

5. We need a representation for a salad. Create a JavaScript class named `Salad` for that. You need to store the `foundation`, `proteins`, `extras`, and `dressing`. Salad objects will be passed to different components in the web app and to avoid passing the inventory to all components the salad object should also store the properties of the ingredients in the salad. Use one or more object as maps to store the ingrediens, same principle as in inventory.
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.

```
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']);
console.log(JSON.stringify(myCaesarSalad) + '\n');
```

Optional assignment: The class above creates an empty salad. Sometimes you want to copy another salad. Implement this functionality in the constructor. There are two important cases: another `Salad` object, and a string containing a JSON representation of a `Salad`.
Hint 1: assume `class Salad{ constructor(arg) ...}`, you do not need to pass an argument when calling the function: `new Salad()` is ok, the value of `arg` will be `undefined`.
Hint 2: use `typeof` on the constructor argument.
Hint 3: `JSON.parse()`

6. *Assignment 3:* Next task is to add additional functionality to the `Salad` class. This must be done by adding the functions to the `Salad`'s prototype chain, not 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. no loops (`for/while`).

   Also, add a function, `count(property)`, that counts the number of ingredients with a property. This can be used to check if a salad is well formed (one foundation and at least three extras...)
   hint: `Object.values()`.

```
console.log('En ceasarsallad kostar ' + myCaesarSalad.getPrice() + ' kr\n');
console.log('En ceasarsallad kostar ' + myCaesarSalad.getPrice() + 'kr');
console.log('En ceasarsallad har ' + myCaesarSalad.count('extra') + ' tillbehör');

// En ceasarsallad kostar 45kr
// 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);`
   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(myCaesarSalad)));
console.log('check 2: ' +
  (Object.prototype === Object.getPrototypeOf(Salad.prototype)));
```

7. One limitation with the Salad class is that you only can have one portion of each ingredient. For example a customer might want two portions of parmesan. 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. The amount should be added 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.
Assignment 4: In `GourmetSalad.add()`, set a size property in the second argument and re-use the (add) function from `Salad` (super.add(name, propertiesWithSize)) to add the ingredient.
Note: The `imported.inventory` object and its properties are read only, see deepFreeze() at the bottom of `inventory.js`.
Hint: if you need to copy an object and add extra properties, use the spread operator in combination with object literals.
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. HTML i text only, so sometimes you want a string identifier for an object. One way to deal with this is to add a unique id to each `Salad` object. Then you can refer to a specific object from the HTML code using this id.
Assignment 5: Use a static instance counter and add the following to the `Salad` constructor:

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

Test it:

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

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

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.

There are all assignments for lab 1. In Lab 2 you will develop your first react application. Lab 2 will take significantly loger time compared to lab 1.

Editor: Per Andersson

Contributors in alphabetical order:
Alfred Åkesson
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`