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 ECMAscript module named `inventory.mjs`. It looks like this:

```
const 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 */
  export default inventory;
};
```

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 zero or more of the 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`, 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.mjs
```

Note on file extensions. Javascript files normally have the file extension .js. For a long time Node only supported CommonJS modules, but now it also supports ECMAscript modules. However, you can not import a ECMAscript module in a CommonJS module, so both `lab1.mjs` and `inventory.mjs` must be ECMAscript modules. Node defaults to CommonJS modules but treats all files with extension .mjs as ECMAscript modules. Use the .mjs file extension in lab 1 and .js for the rest of the labs.

For this lab, please add additional printouts, to make it clear which text belongs to which task. There is a template file, `lab1.mjs`, in Canvas that contains a skeleton program, so you do not need to copy code from this pdf.Try the following code (you need to download ./inventory.mjs from Canvas or GitHub first):

```
import inventory from './inventory.mjs';
console.log('\n=== beginning of printout ================================')
console.log('inventory:', 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`. VS code 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 optional 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 download the files `lab1.mjs` and `inventory.mjs` from Canvas to this directory. Run the code from VS code or a terminal:

```
node lab1.mjs
```

3. In the `inventory.mjs` file you can find all data for composing a salad. Its structure is good for looking up properties of a known ingredient, e.g. `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(inventory);
names.forEach(name => console.log(name));
```

Note, `forEach` passes three argument: (name, index, array), but we only use the name, so it is enough to declare one parameter in our callback function.
In this case, you will get the same result with:

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

Reflection question 2: When will the two examples above give different outputs, and why is inherited functions, such as `forEach()`, 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. Another advantage is that this that a function call is an expression, so you can do this inline, for example:

```
const myString = 'choose one of: ' + JSON.stringify(
names
.sort((a, b) => a.localeCompare(b, "sv", {sensitivity: 'case'}))
.forEach(name => console.log(name))
) + '. This is all options we offer';
```

This is extremely common in the render function of a web app. Your life as a programmer gets a lot easier when you get familiar with function chaining.

Assignment 1: Write a function that returns a string containing the HTML `<option>` elements for a select box, that has all salad ingredients with a certain property. Example: `makeOptions(inventory, 'foundation')` should return

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

Hint: Use the functions `Array.prototype.filter()`, `Array.prototype.map()` and `Array.prototype.reduce()`.
Note: The `key` attribute is not standard HTML. It is needed by React for tracking changes in the DOM.

4. We need a representation for a salad. Create a JavaScript class named `Salad` for that. You need to store the selected `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 selected 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', inventory['Sallad'])
.add('Kycklingfilé', inventory['Kycklingfilé'])
.add('Bacon', inventory['Bacon'])
.add('Krutonger', inventory['Krutonger'])
.add('Parmesan', inventory['Parmesan'])
.add('Ceasardressing', inventory['Ceasardressing'])
.add('Gurka', 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 4a: Implement this functionality by adding a parameter to the constructor. An empty salad should be created if no argument i passed. If another `Salad` object is passed, copy it.
Assignment 4b: Add a static function, `Salad.parse(json)` that parse a string and return a `Salad` object, or an array of `Salad` objects. The argument must be the JSON representation of a singel salad, or an array of salads.
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, instead `arg` will have the value `undefined`. Check for this or `arguments.length` to see if there is a salad to copy.
Hint 2: use `typeof` to check if a value is a string. `instanceof` can be used to check if an object is a `Salad`.

Hint 3: `Array.isArray()` is the only safe way to check if a value is an array.
Hint 4: `JSON.parse()` will return an object that looks like a `salad`, but is not an instance of `Salad`. All methods in the prototype chain are missing. `singleCopy.getPrice` will be `undefined` resulting in the error: `Uncaught TypeError: undefined is not a function` if you try to call it.
Hint 5: Use the spread operator in combination with object literals to copy objects.

```
const singleText = JSON.stringify(myCaesarSalad);
const arrayText = JSON.stringify([myCaesarSalad, myCaesarSalad]);
const objectCopy = new Salad(myCaesarSalad);
const singleCopy = Salad.parse(singleText);
const arrayCopy = Salad.parse(arrayText);
console.log('original myCaesarSalad\n' + JSON.stringify(myCaesarSalad));
console.log('new(myCaesarSalad)\n' + JSON.stringify(objectCopy));
console.log('Salad.parse(singleText)\n' + JSON.stringify(singleCopy));
console.log('Salad.parse(arrayText)\n' + JSON.stringify(arrayCopy));
singleCopy.add('Gurka', inventory['Gurka']);
console.log('originalet kostar ' + myCaesarSalad.getPrice() + ' kr');
console.log('kopian med gurka kostar ' + singleCopy.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, that defaults to 1. You can add the same ingredient several times to the same salad. For each call to add, the size should increase (newSize = oldSize + addedSize). The size should be stored among the other properties of the ingredient, for example{ `price: 10, protein: true, size: 1.5` }. The price should be the unit price, so `getPrice()` will compute $prize * size$. It is always good to rely on the implementation of the superclass extending a class. `GourmetSalad.prototype.add` should call `Salad.prototype.add` (`super.add(name, propertiesWithSize)`).
Note: Some objects are shared between `inventory` and among salad objects. You must not modify these. Instead make a copy. If you forget this you will get a run-time error since inventory is read only, see deepFreeze() at the bottom of `inventory.mjs`.
Here is a test case:

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

// Min gourmetsallad med mindre 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 view of

a shopping cart must be able to identify the salad object. 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. This will happen 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.mjs` 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:

```
import { v4 as uuidv4 } from '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. A copy should have a new uuid, but when you create salad objects using the parse() function the uuid should be preserved. This is not a copy, just a change of representation.
Use cases 1, different salads:

```
const salad1 = new Salad();
// add ingredients to salad 1
const salad2 = new Salad(salad2)
// salad1.uuid !== salad2.uuid, they are different salads
salad2.add(Bacon', inventory['Bacon']);
order(salad1, salad2);
```

Use cases 2, same salads, you want to update the salad in the database:

```
const salad1 = new Salad(); // add ingredients to salad 1
storeInDatabase(salad1);
// app is reloaded, all JavaScript objects are lost
const text = fetchFromDatabase();
const salad2 = Salad.parse(text);
// salad1.uuid === salad2.uuid, they are the same salad
salad2.add(Bacon', inventory['Bacon']);
storeSaladInDatabase(salad2); // update the existing salad
```

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:
Ahmad Ghaemi
Alfred Åkesson
Anton Risberg Alaküla
Mattias Nordal
Oskar Damkjaer
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`