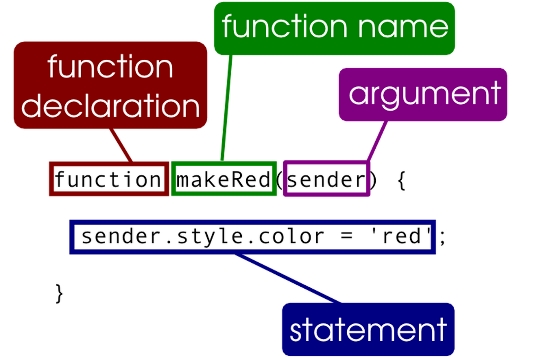
JavaScript Functions

JavaScript functions and Callbacks

**1)** Create a new JavaScript file and add these three functions

**Function Declaration**

       //Observe: no return type, no type on parameters

function add(n1, n2){

   return n1 +n2;

}

**Function Expression**

var sub = function(n1,n2){

  return n1 - n2

}

**Callback example**

var cb = function(n1,n2, callback){

  return "Result from the two numbers: "+n1+"+"+n2+"="+callback(n1,n2);

};

The following questions might seem trivial, but it's extremely important that you can answer (and understand) each, in order to do the JS-stuff we want to do this semester

**2)** Call the functions above as sketched below. It’s not about doing it as fast as you can, but about understanding what's happening, so make sure you understand each line.

1. console.log( add(1,2) )     // What will this print?  
     
   3
2. console.log( add )          // What will it print and what does add represent?  
     
   ƒ add(n1,n2){

return n1+n2;

}  
  
Den funktion der er gemt som add.

1. console.log( add(1,2,3) ) ; // What will it print  
     
   3
2. console.log( add(1) );   // What will it print   
     
   NaN
3. console.log( cb(3,3,add) ); // What will it print  
     
   "result from the two numbers: 3+3=6"
4. console.log( cb(4,3,sub) ); // What will it print  
     
   "result from the two numbers: 4+3=1"
5. console.log(cb(3,3,add())); // What will it print (and what was the problem)  
     
   Uncaught TypeError: callback is not a function

at cb (<anonymous>:2:58)

at <anonymous>:1:1  
  
Problemet er at den ikke får en funktion / forkerte argumenter.

1. console.log(cb(3,"hh",add));// What will it print  
     
   "result from the two numbers: 3+hh=3hh"

3)  Error Handling

7 will fail due to missing/wrong arguments. But it will fail runtime, not as with Java, at compile time.

We can check arguments in JavaScript as sketched below and provide better errors by throwing our own exceptions:

**typeof n1 === "number"** //Will fail if n1 is undefined, or is not a number

**typeof callback === "function"** //Will fail if callback is undefined or is not a function

Rewrite the Callback function expression (cb)  to make a check for all its three required arguments, and throw an Error if any of the arguments do not match as [explained here](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error#Throwing_a_generic_error).

Surround the call to the function with a try-catch block, and provide a more user-friendly error message if the function throws an error

var cb = function(n1,n2,callback){

try{

typeof n1 === "number"

typeof callback === "function"

return "Result from the two numbers: "+n1+"+"+n2+"="+callback(n1,n2);

} catch(e){

console.log("Something went wrong")

}

};

**More Callbacks**

Take another look at the function expression declared in ***cb***, and the provided callbacks in 5+6. What we have in ***cb*** is a very generic function, that can take any callback that can do something with two numbers and return a value.

**4)**  Write a mul(n1, n2) function (inspired by add and sub) and use it as the callback for the **cb** function  
  
var mul = function(n1, n2){

return n1\*n2;

}  
  
cb(3,4,mul)

"Result from the two numbers: 3+4=12"

***5)*** *Call* ***cb****, this time with an anonymous function that divides the first argument with the second  
  
function ergh(n1,n2){return n1/n2;}*

*cb(4,3,ergh)*

*"Result from the two numbers: 4+3=1.3333333333333333"*

Callbacks (with map, filter and forEach)

We saw a simple example of a callback above. Let's get familiar with callbacks, using some of the array-type’s built-in methods.

Getting comfortable with *filter*, *map* and *forEach*:

**1)** Declare a JavaScript array and initialize it with some names (Lars, Jan, Peter, Bo, Frederik etc.). Use the **filter** method to create a new array with only names of length <=3.

Use the forEach method to iterate and print (console.log) both the original and the new array.

var arr = ["lars","jan","peter","bo","frederik"]

var filtered = arr.filter(function(name){return name.length<=3;})

arr.forEach(function(n){console.log(n)})

lars

jan

peter

bo

frederik

`

filtered.forEach(function(n){console.log(n)})

jan

bo

**2)** Use the names-array created above, and, using its **map** method, create a new array with all names uppercased.  
  
ARR = arr.map(function(n){ return n.toUpperCase();})

*We will continue with this exercise tomorrow when we start manipulating the browser's DOM*

**3)** Use map**,** join + just a little bit more to create a function, which given the array of names, for example: ["Lars", "Peter", "Jan", "Ian"] returns a string with the HTML for the names in an <ul> as sketched below:

<ul>

  <li>Lars</li>

  <li>Peter</li>

  <li>Jan</li>

  <li>Ian</li>

<ul>

The output above was shown with newlines for readability, but this is actually what we want (why):

<ul><li>Lars</li><li>Peter</li><li>Jan</li><li>Ian</li><ul>

var toTable = function(array){

return "<ul><li>"+array.join("</li><li>")+"</li><ul>"

}

"<ul><li>Lars</li><li>Peter</li><li>Jan</li><li>Ian</li><ul>"

*Tomorrow we will use DOM manipulation and place this into a “running” web-page.*

**4)**  Given this JavaScript array

var cars = [

  { id: 1, year: 1997, make: 'Ford', model: 'E350', price: 3000 },

  { id: 2, year: 1999, make: 'Chevy', model: 'Venture', price: 4900 },

  { id: 3, year: 2000, make: 'Chevy', model: 'Venture', price: 5000 },

  { id: 4, year: 1996, make: 'Jeep', model: 'Grand Cherokee', price: 4799 },

  { id: 5, year: 2005, make: 'Volvo', model: 'V70', price: 44799 }

];

a) Use the filter **filter** function to get arrays with only:

* Cars newer than 1999
* Al  Volvo’s
* All cars with a price below 5000

1. cars.filter(function(n){ if(n.year > 1999) return n})  
   0: {id: 3, year: 2000, make: "Chevy", model: "Venture", price: 5000}
2. 1: {id: 5, year: 2005, make: "Volvo", model: "V70", price: 44799}
3. cars.filter(function(n){if(n.make ===’Volvo’)return n})  
   0: {id: 5, year: 2005, make: "Volvo", model: "V70", price: 44799}
4. length: 1
5. cars.filter(function(n){if(n.price < 5000 )return n})  
   *(3) [{…}, {…}, {…}]*
   1. 0: {id: 1, year: 1997, make: "Ford", model: "E350", price: 3000}
   2. 1: {id: 2, year: 1999, make: "Chevy", model: "Venture", price: 4900}
   3. 2: {id: 4, year: 1996, make: "Jeep", model: "Grand Cherokee", price: 4799}
   4. length: 3

**4a)**      Use **map,** **join** + just a little bit more to implement a function, that , given the cars array used above, will create, and return a string with valid SQL statements to insert the data into a table with matching column names (id, year, make, model, price) as sketched below:

INSERT INTO cars (id,year,make,model,price) VALUES ( 1, 1997 'Ford','E350', 3000 );

...

var toSQL = function(arr){

var sqlArr = [];

sqlArr = arr.map(function(n){

var str = "INSERT INTO cars(id,year,make,model,price) VALUES ( ";

str+=n.id+","+n.year+","+n.make+","+n.model+","+n.price+" );";

return str;

})

return sqlArr;

}  
  
toSQL(cars).forEach(n=>console.log(n))

INSERT INTO cars(id,year,make,model,price) VALUES ( 1,1997,Ford,E350,3000 );

INSERT INTO cars(id,year,make,model,price) VALUES ( 2,1999,Chevy,Venture,4900 );

INSERT INTO cars(id,year,make,model,price) VALUES ( 3,2000,Chevy,Venture,5000 );

INSERT INTO cars(id,year,make,model,price) VALUES ( 4,1996,Jeep,Grand Cherokee,4799 );

INSERT INTO cars(id,year,make,model,price) VALUES ( 5,2005,Volvo,V70,44799 );

Asynchronous Callbacks

Most of the javascript callbacks you will be using will be *asynchronous*, in contrary to map, *filter* and forEach which are *synchronous* (MAKE SURE you understand the difference)

**1)** Given the code below answer, don’t execute the code, in what order you would expect to see the outputs:

var msgPrinter = function(msg,delay){

**setTimeout**(function(){

    console.log(msg);

  },delay);

};

console.log("aaaaaaaaaa");

msgPrinter ("bbbbbbbbbb",2000);

console.log("dddddddddd");

msgPrinter ("eeeeeeeeee",1000);

console.log("ffffffffff");  
  
aaaaaaaaa  
ddddddddd  
fffffffff  
eeeeeeeee  
bbbbbbbbb

**2)** Add the code to a javascript file, execute and verify whether you answer to 1) was right

I wus

*this* and *constructor functions*

*The value of* ***this*** *passed to all functions, is based on the context in which the function is called at runtime. Pay attention here, because this is one of those quirks you just need to memorize.*

*In the example below, we actually have two versions of* ***this*** *in play (one for the outer function, and one for the asynchronous callback)*

1) Add this code,

function Person(name){

  this.name = name;

  console.log("Name: "+ this.name);

  setTimeout(function(){

    console.log("Hi  "+this.name); //Explain this

  },2000);

}

//call it like this (do it, even if you know it’s silly ;-)

Person("Kurt Wonnegut");  //This calls the function

console.log("I'm global: "+ name);  //Explain this

That was silly. How do we use a function starting with a *capitalized letter?*, and what do we call such a function?

Det er vel fordi this kun refererer til kurt inden for funktionens scope

**2)** Create a Person instance and rerun the example as sketched below*:*

var p = new Person("Kurt Wonnegut");  //Create an instance using the constructor function

console.log("I'm global: "+ name);  //What’s different ?

outputted fra Person funktionen bliver nu gemt i variablen p

*We still need to fix the problem with the callback, not having access to the “outer” this.*

3) Change your code to fix the problem, using both strategies given below.

|  |  |  |
| --- | --- | --- |
| **//Store a reference to the outer this**  function Person(name){    this.name = name;    var self = this;    console.log("Name: "+ this.name);    setTimeout(function(){      console.log("Hi  "+self.name);    },2000);  } |  | **//Using the bind(..) function**  function Person(name){    this.name = name;    console.log("Name: "+ this.name);    setTimeout(function(){      console.log("Hi  "+this.name);    }.bind(this),2000);  } |

4) The *bind* method will be extremely important to understand for our future journey into javascript.

React uses a component-based strategy, for “components” that must be rendered in a browser window. You will often find yourself in situations where you hook up an event handler in one “component”, but actually execute the handler code in another. Here it’s important that we can control which ***this*** are used.

Write, run and UNDERSTAND the example below

var greeter = function(){

  console.log(this.message);

};

var comp1 = { message: "Hello World" };

var comp2 = { message: "Hi" };

var g1 = greeter.bind(comp1 );//We can store a reference, with a specific “this” to use

var g2 = greeter.bind(comp2 );//And here another “this”

setTimeout(g1,500);

setTimeout(g2,1000);

          JavaScript Objects

1) Create an object with four different properties, with values, of your own choice (ex: name, birthday, hobby, email).

Use a for-in loop (as sketched below) to demonstrate that we can iterate over the properties in an object.

for(prop in myObj){  
  console.log(prop,obj[prop]);  
}

Use the delete keyword to demonstrate we can delete existing properties from an object (delete a property, and iterate over the properties again)

Add a new property to your object to demonstrate that we can add new properties to existing objects  
  
for(prop in myObj){

console.log(prop,myObj[prop]);

}

name Navn

birthday Fødselsdag

hobby Hobby

email email  
  
  
  
  
delete myObj.hobby

true

for(prop in myObj){

console.log(prop,myObj[prop]);

}

name Navn

birthday Fødselsdag

email email  
  
  
myObj.newHobby = "NewHobby"

"NewHobby"

for(prop in myObj){

console.log(prop,myObj[prop]);

}

name Navn

birthday Fødselsdag

email email

Hobby NewHobby

2) Create a Constructor function to create new Persons having:

* a firstName, lastName and an age.
* A method to get details about the Person

function Person(firstName, lastName, age){   
 this.firstName = firstName;  
 this.lastName = lastName;  
 this.age = age;  
 this.Details = function(){  
 return this.firstName +" "+ this.lastName+", "+this.age;   
}}

     Reusable Modules with Closures

1) Implement and test the Closure Counter Example from the Slides

2) Implement a reusable function using the Module pattern that should encapsulate information about a person (name, and age) and returns an object with the following methods:

* setAge
* setName
* getInfo (should return a string like Peter, 45)

All of it

Don't do this exercise in the class. Do it someday where you have some spare time and just want to summarize all your previous JavaScript knowledge.

Make yourself a nice cup of coffee, tea or perhaps even grab a beer ;-) and go to:

[http://bonsaiden.github.io/JavaScript-Garden](http://bonsaiden.github.io/JavaScript-Garden/)    Read all the text and execute all examples.