Objects in JavaScript

In a nutshell: Objects in JavaScript are similar to what in other languages (e.g. Python) are called dictionaries or *named* arrays.

Example:

```
var rabbit1 = {name: "tommy",
               age: 4,
               greet: function(){
                          console.log("Hi from rrrabbit1");
                      }
              };

console.log(rabbit1.name) // 'tommy'
console.log(rabbit1["name"]) // 'tommy'
rabbit1.greet() //Hi from rrrabbit1
```

The *dot* syntax, e.g. `rabbit1.greet()` is a characteristic signal that we may be dealing with an object (if this were about the C language, then it signals the use of a `struct` structure instead, as C lacks objects),

Analogies:

- `struct` in C
- `namespace` in C++, Scala, Java
- `class/module` in Python

Prototyping, aka. OO-Programming in JS

Using the HTML Live Editor:

```
function sing(){
  msg = "I'm "+this.name  ;
  document.body.innerHTML+= msg;
  console.log(msg) ;
}
var rabbit1 = {name: "tommy", sing:sing} ;
var rabbit2 = {name: "tammy", sing:sing} ;

rabbit2.sing();

//What about rabbit3? another sing??

//Ans: Prototyping (aka. OO-programming a la JS)

//Constructor
```

```
function Rabbit(name="none-yet"){
  this.name = name ;
}
// name is an instance variable, but sign shall be a prototype method
Rabbit.prototype.sing = function(){
    msg = "Hi, I'm "+this.name  ;
  document.body.innerHTML+= msg;      //append msg to page
  console.log(msg) ;                 // print msg to console
}

var tommy = new Rabbit("tommy") ;
var tammy = new Rabbit("tammy") ;
tommy.sing() ;
tammy.sing();
```

Exercises

1. **Matrices**: We can think of a matrix as a particular 2-dimensional arrangement of numbers. An $n \times m$ matrix has n rows and m columns.

   Examples:

   - A 3×1 matrix
   $$\begin{pmatrix} 3.3 \\ 1 \\ 7.01 \end{pmatrix}$$
   - A 2×3 matrix
   $$\begin{pmatrix} 3.3 & 5 & 7 \\ 1 & 2.0 & 9 \end{pmatrix}$$
   - A 3×3 matrix
   $$\begin{pmatrix} 3.3 & 1 & 2 \\ 5 & 81 & 746 \\ 0.1 & 2 & 2 \end{pmatrix}$$

   Implement matrices in Javascript as objects. The constructor should accept as input the dimensions and a list or array of numbers as the data.

   Example: `var a3x1matrix = new Matrix([3,1], [3.3, 1, 7.01])` should produce the first matrix. Here, the dimensions have been given as an array of 2 integers and the data is internally organized *as needed*.

   The following methods should be implemented:

   1. `a3x1matrix.dim //returns the dimensions as an array of integers for rows and columns`
   2. `a3x1matrix.print() //print to console and on the page the matrix reflecting its shape`. Thus it should print like above, although you may ignore the parentheses.

2. **Matrix Multiplication**: We can define a multiplication of two matrices A and B and write $A * B$, or simply AB if there is no confusion, **iff** *the number of columns of A is the same as the number of rows of B*!

   That is, A is of dimension say $n \times k$ then B must be of dimensions $k \times m$, where n and m are arbitrary. The result will be a new matrix of dimensions $n \times m$.

   Examples:

   - $A = \begin{pmatrix} 3 \\ 1.5 \\ 2 \end{pmatrix}$ is 3×1, while $B = \begin{pmatrix} 4.5 & 52 & 0 \end{pmatrix}$ is a 1×3 matrix. Then it makes sense the multiplication
   $$AB = \begin{pmatrix} 13.5 & 156 & 0 \\ 27/4 & 78 & 0 \\ 9 & 104 & 0 \end{pmatrix}$$
   .
   - It makes no sense, however, to try multiplying the above matrices in reverse order, that is, BA **is not well defined**!

   How does Matrix Multiplication work?: Consider the following two matrices
   $$A = \begin{pmatrix} 3 & 8 & 5 \\ 1 & -2 & 6 \end{pmatrix} \quad B = \begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 1 & 2 \end{pmatrix}$$

   Their product AB will be a 2×3 matrix given by
   $$AB = \overrightarrow{\begin{pmatrix} 3 & 8 & 5 \\ 1 & -2 & 6 \end{pmatrix}} \begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 1 & 2 \end{pmatrix} \Bigg\downarrow =$$
   $$= \begin{pmatrix} 3 \cdot 9 + 8 \cdot 6 + 5 \cdot 3 & 3 \cdot 8 + 8 \cdot 5 + 5 \cdot 1 & 3 \cdot 7 + 8 \cdot 4 + 5 \cdot 2 \\ 1 \cdot 9 + (-2) \cdot 6 + 6 \cdot 3 & 1 \cdot 8 + (-2) \cdot 5 + 6 \cdot 1 & 1 \cdot 7 + (-2) \cdot 4 + 6 \cdot 2 \end{pmatrix} =$$
   $$= \begin{pmatrix} 27 + 48 + 15 & 24 + 40 + 5 & 21 + 32 + 10 \\ 9 - 12 + 18 & 8 - 10 + 6 & 7 - 8 + 12 \end{pmatrix} =$$
   $$= \begin{pmatrix} 90 & 69 & 63 \\ 15 & 4 & 11 \end{pmatrix}$$

   The arrows hint you at the way those products are organized.

   General case: Let's convey on a way to denote the elements of any matrix by using the row and column as *coordinates*. We will denote the element of matrix A at row r and column c is A_{rc}.

   Say A's dimensions are $n \times c$ and B's $c \times m$.

The product AB is as well a matrix. What is its element at row i and column j? That is, what will be the value of $(AB)_{ij}$? Ans:

$$(AB)_{ij} = \sum_{k=1}^{k=c} A_{ik} \cdot B_{kj} = A_{i1} \cdot B_{1j} + A_{i2} \cdot B_{2j} + \ldots + A_{ic} \cdot B_{cj}$$

Implement matrix multiplication by extending the code developed in question 1 before. Make it so that the previous product could be coded as `var AB = A.mult(B)` where A and B are matrices previously instantiated. It should **not** work in the reverse order $B.mult(A)$!

3. **Laying Out a Table**: (pp 108, Ch.6, "Eloquent...")
   - **Input**: `[{mountain}]` and 'mountain ~ {name: "mnt", height: 1, country: "mountania"}, i.e., an array of "mountain" objects
   - **Output**:
     1. `[[cell]]`, i.e., an array of array of cells, and...
     2. string w/ nicely layed-out table to print.

name	height	country
Kilimanjaro	5895	Tanzania
Everest	8848	Nepal
Mount Fuji	3776	Japan
Mont Blanc	4808	Italy/France
Vaalserberg	323	Netherlands
Denali	6168	United States
Popocatepetl	5465	Mexico

The input data we are given is an array `MOUNTAINS` of "mountains", each of these being an object as, e.g., `{name: "Kilimanjaro, height: 5895, country: "Tanzania"}`. In other words, and according to the table above, the following call to `console.log` `console.log(MOUNTAINS.slice(0,2))` would output the array of only two elements given by `[{name: "Kilimanjaro", height: 5895, country: Tanzania},{name: "Everest",height: 8848, country: "Nepal"}]`

Furthermore, we want the following methods to be implemented (below you may find examples of their usage that clarify their signature -or *type*): `rowHeights`, `colWidths`, `drawTable {drawLine, drawRow}`, `TextCell`, `dataTable`, `UnderlinedCell`

The drawTable function uses the internal helper function drawRow to draw all rows and then joins them together with newline characters.

The drawRow function itself first converts the cell objects in the row to

blocks, which are arrays of strings representing the content of the cells, split by line. A single cell containing simply the number 3776 might be repre- sented by a single-element array like [“3776”], whereas an underlined cell might take up two lines and be represented by the array [“name”, “—”].

The blocks for a row, which all have the same height, should appear next to each other in the final output. The second call to map in drawRow builds up this output line by line by mapping over the lines in the leftmost block and, for each of those, collecting a line that spans the full width of the table. These lines are then joined with newline characters to provide the whole row as drawRow’s return value.

The function drawLine extracts lines that should appear next to each other from an array of blocks and joins them with a space character to create a one-character gap between the table’s columns.

The constructor for cells that contain text, implements the interface for table cells. The constructor splits a string into an array of lines using the string method split, which cuts up a string at every occurrence of its argument and returns an array of the pieces. The minWidth method finds the maximum line width in this array.

An underlined cell contains another cell. It reports its minimum size as being the same as that of its inner cell (by calling through to that cell’s minWidth and minHeight methods) but adds one to the height to account for the space taken up by the underline.

Drawing such a cell is quite simple—we take the content of the inner cell and concatenate a single line full of dashes to it.

Examples of usage of the methods:

```
console.log(drawTable(dataTable(MOUNTAINS)));
/* This prints: name,  height and country are underlined cells
   name         height country
   ------------ ------ -------------
   Kilimanjaro  5895   Tanzania
     ... etcetera
*/

var rows = [];
for (var i = 0; i < 5; i++) {
   var row = [];
   for (var j = 0; j < 5; j++) {
     if ((j + i) % 2 == 0)
       row.push(new TextCell("##"));
     else
       row.push(new TextCell("  "));
}
```

```
    rows.push(row);
}
console.log(drawTable(rows));
/* This prints:
   ##  ##  ##
     ##  ##
   ##  ##  ##
     ##  ##
  ##   ##  ##
*/
```

4. **Library**: The advantage of the paradigm of OO-programming is twofold:

    1. it entails the idea of modularity, where we isolate part of the code in "chunks" that we can plug and reuse wherever we need. This saves us time, is less error prone, is easier to debug, the code is easier to read.

    2. it provides a new abstraction level that increases our expresivity as programmers: Consider the following paragraph:

        "*A perceptron constitutes a simplistic model of how neurons manage signal flow in order to interact with each other and thereby give rise to an information flow.*

        A perceptron is characterized by the type of ***aggregatpor*** *(also* ***synaptic potential****) function and the type of non-linear,* ***activation*** *function it uses.*

        A neuron interacts with other neurons by establishing connections (called ***synapses****) at its two ends, the* ***axon*** *(or tail) and the nueron body (called* ***soma****). The usual signal flow is* ***from*** *the neuron's body* ***to*** *its axon terminal*[1]*. At the neuron's body it accumulates multiple signals comming from different synapses at the soma's dendrites' end. This compound electrical signal is gets "proccessed" in the body. If the signal (voltage) is high enough, the neuron fires on through its synapses at the axon's terminal otherwise, it doesn't, and keeps "silent" until next input signal gets processed.*"

        Implementing such a conceptual picture of neurons connected to each other in a programming language can be quite a challenge *unless the language offers enough expresivity.* How could one code the expressions "this neuron", "that neuron", "neuron A is connected with neuron B", "neuron A fires, neuron B gets actived and neuron C is connected to both",etc., such that the very same code captures the idea of a tangible *thing* as those expressions suggest in english? That is, the ideal languagge would allow us to write an english text the way we usually write it and have the computer proccess and calculate the

[1] This is the so called *neuron doctrine* put forward by the Spanish neurologist and Nobel prize-winner Santiago Ramón y Cajal.

answer we want. Of course, the only computer than works this way is our brother or sister when we convince them to do some work for us. Alas, no programming language allows such a thing, and all are but a compromise between the *think-as-a-cpu-when-programming* and the *think-as-a-human-when-programming*, differing in the difficulty of implementing different things.

If a programming language allows to express those concepts in an easier way than another one, then we say the first language offers or allows for a higher abstraction level or more expressivity. The advantage is tremendous: the programmer can concentrate more on the **ideas and concepts she wants to model instead of on how to model them**!

But not all the burden of abstraction, expressivity and modularity lies on the syntax of a language.

Organizing our code into different parts **stored in different files** is another, explicit way of increasing our modularity, and whence our abstraction and expressivity levels. This is the idea of **modules** or **libraries**: code already written and tested that we can **use** in our own program without even the need to copy&paste it, but simply ***importing* and calling it**.

In JavaScript we can do this in two different ways: (1) if we are building the *front-end*, aka user-interface, of a web app, we can write our code in a file say `matrix.js` and "import" it in the `HTML` code using the `<script src="url_of_js_file"></script>` tag.

(2) if we are building the *back-end*, or server-side of a web app using `node.js`, then we can "import" the desired library by using the keyword `require(url_of_js_file)`.

Example: Here is how method (1) would be implemented.

```
<!doctype html>
<html lang="en">
  <head>
    <title>Matrix Algebra</title>
    <meta charset="utf-8">
    <script>

      //Helper function
      function print(x=""){
        document.body.innerHTML += x + "<br>";
      }
    </script>
    <script src="file:///Users/msantos/Dragon/Course/G12/matrix.js">
    </script>
```

```
    <style>
        body {
            font-family: monospace;
            background: #444444;
            color: aliceblue;
        }
    </style>
  </head>

  <body>



    <script>

      print("Testing Matrix.js:");
      var beta = Math.PI/3. ;
      var cb = Math.cos(beta) ; var sb = Math.sin(beta);
      var x = new Matrix([3,1],[cb,sb,0],"x")
      var y = new Matrix([3,1],[-sb,cb,0],"y")
      var z = new Matrix([3,1],[0,0,1],"z")
      var Rx = x.mult(x.t())
      var Ry = y.mult(y.t())
      var Rz = z.mult(z.t())
      print(x+y+z)
      print(Rx)
      print(Ry)
      print(Rz)
      sv = x.add(y).add(z);
      print(sv +"=sv")
      sv.name="sv"
      print(sv+sv.t().mult(sv))

    </script>
  </body>
</html>
```

The `url` can be reduced to simply the name of the file (`matrix.js`) in case both, the `HTML` and the JS files lie in the same directory.

Exercise: Modify the code for `matrix.js` such that the output of this testbed is exactly as follows:

```
Testing Matrix.js:
x[ 0.5000000000000001 ]
 [ 0.8660254037844386 ]
 [ 0 ]
```

```
y[ -0.8660254037844386 ]
 [ 0.5000000000000001 ]
 [ 0 ]
z[ 0 ]
 [ 0 ]
 [ 1 ]

xx†[ 0.2500000000000001 0.4330127018922194 0 ]
   [ 0.4330127018922194 0.7499999999999999 0 ]
   [ 0 0 0 ]

yy†[ 0.7499999999999999 -0.4330127018922194 0 ]
   [ -0.4330127018922194 0.2500000000000001 0 ]
   [ 0 0 0 ]

zz†[ 0 0 0 ]
   [ 0 0 0 ]
   [ 0 0 1 ]

((x+y)+z)[ -0.3660254037844385 ]
         [ 1.3660254037844388 ]
         [ 1 ]
=sv
sv[ -0.3660254037844385 ]
  [ 1.3660254037844388 ]
  [ 1 ]
sv†sv[ 3.0000000000000004 ]
```

5. **Complex (or Imaginary) Numbers**: Following the example of the Matrix algebra, implement a library for dealing with *complex (imaginary) numbers*. In particular, implement a constructor that handles as well name labels, and methods for adding, multiplying complex numbers, for taking the conjuagate, for calculating their norm (or magnitude), their angle with respect to the x-axis, and for calculating its inverse.

   In particular, the following tests code

```
<html>
  <!-- HTML template page: complex-test.html
       http://msantos.sdf.org/G12/Term2/complex-test.html
  -->
  <head>
    <title>Complex test bed</title>
    <meta charset="utf-8">
    <script>
      //Helper function
      function print(x){
```

```
        document.body.innerHTML += x + "<br>";
      }
    </script>
    <style>
      body { font-family: monospace ; }
    </style>
    <script src="complex.js"></script>
  </head>

  <body>
    <script>

      var z = new Complex([2,2],"z")
      var v = new Complex([-2,1],"v")
      var w = new Complex([-7,4],"w")
      var i = new Complex([0,1],"v")
      print(z)     //prints z(2,2)
      print(v)     //      v(-2,1)
      print(w)     //      w(-7,4)
      print(i)     //      i

      var zt = z.t()       // z†(2,-2)
      var vt = v.t()       // v†(-2,-1)
      var it = i.t()       // i†

      print("|z|="+ z.norm() )    // |z| = 2.8284
      print("|v|="+ v.norm() )    // |v| = sqrt(5) ~ 2.2361
      print( "|i|="+i.norm() )    // |i| = 1

      var z_v = z.add(v)   // (z+v)
      var wxz_v = w.mult(z.add(v)) // w(z+v)

      print( z.mult(i) )   //  zi(-2,2)
      print( i.mult(i) )   //  -1
      print( z.inv() )    // 1/z = z†/|z|^2 = (0.25, -0.25)
      print( z.inv().mult(z) ) // 1

      print( w.phase() )   // angle with x-axis
    </script>
  </body>
</html>
```

outputs

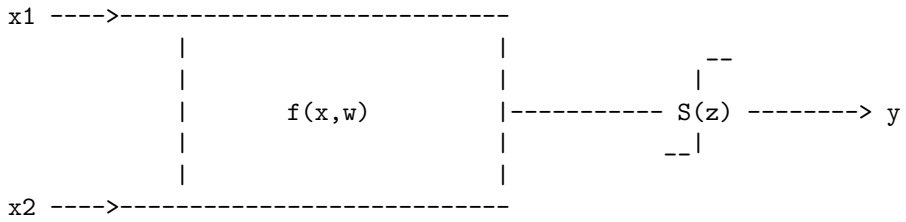
```
z(2,2)
v(-2,1)
w(-7,4)
```

```
i
|z|=2.8284271247461903
|v|=2.23606797749979
|i|=1
zi(-2,2)
-1
z'(0.24999999999999994,-0.24999999999999994)
0.9999999999999998
-0.5191461142465229
```

6. **Perceptron**: The actual functioning of a neuron involves many complicated processes, not the least because not all neurons have the same structure nor do all interact with other neurons in the same way.

   The main function of a neuron is basically that of a *signal receiver-transmitter*, or singal transducer: it receives input signals at one end and outputs signals at another end. Simplifying further, in its simplest form, this could be receiving one input signal and providing one ouput signal, or receiving two input signals and outputting one.

   It is this function that we would like to model in as simple as possible a way. The perceptron is such a model invented in 1957 by Frank Rosenblatt.



   The input signals are processed in two stages: $z = f(x, w)$ and $y = S(z)$, where $x = [x_1, x_2]$ are the input values, $w = [w_1, w_2]$ are *weights* characteristic of each perceptron, $f(x, w) = x_1\, w_1 + x_2\, w_2$ is the aggregation or integration function, which in this case is a weighted sum of the input values, and `S`, the activation function, is a step function: $S(z) = 0$ if $z \leq 0$ and $S(z) = 1$ otherwise. Whence, composing both functions, we can write $y = S(f(x, w))$.

   Question: What does that description and diagram of a neuron and a perceptron reminds you of? Have we seen before abstractions that resemble these?

   Exercise: Write a library that implements a perceptron with an arbitrary number of input values (input neurons).

More on Objects in JavaScript

- Overriding derived properties
- Check for property: `in` Vs `hasOwnProperty`
- Loop: for/in Vs in
- (Non-)Enumerable Properties
- Object.creat(null) : bare-metal object
- Polymorphism
- get/set (optional)
- Inheritance
- Private properties and methods: The Bank.

Overriding Derived Properties

Any instance can modify its inherited properties, i.e., its member variables and methods. See `rabbit2` in the following example.

```
function Rabbit(name="none",greeting="ugh"){
  this.name = name ;
  this.greeting = greeting ;
}

Rabbit.prototype.greet = function(){
  return (this.greeting+" "+this.name) ;
}

var rabbit1 = new Rabbit("tommy","Hi, I'm ")

alert(rabbit1.greet()) ;


var rabbit2 = new Rabbit("tammy","Welcome to my world!") ;

rabbit2.greet = function(){
  return "Hi I'm "+this.name+". "+this.greeting ;
}

alert(rabbit2.greet())
```

in Vs. hasOwnProperty

```
for(p in rabbit1){
  console.log(p);   //name, greeting, greet
}
```

```
//Add property to ancestor object 'Object' thus changing time and evolution
Object.prototype.absurd = "uh!?";

for(p in rabbit1){
  console.log(p);   //name, greeting, greet, absurd
}
```

We cna check whether an object has or not a given property or method with the keyword `in`:

```
console.log("absurd" in rabbit2)    //true
console.log("toString" in rabbit2)  //true
```

There is a difference between our defined method `greet` and the method `toString`. **The latter doesn't show up in the `for-in` loop***!

We call the latter a *nonenumerable* property, while the former would be an *enumerable* one.

```
//all enumerable properties (whether instance's or ancestor's ones)
console.log("greet in rabbit1: "+ ("greet" in rabbit1))
//all instance properties (whether enumerable or not)
console.log("rabbit1.hasOwnProperty(greet): "+rabbit1.hasOwnProperty("greet"))
```

Bare Metal Object

```
var bareRabbit = Object.create(null)
for(p in bareRabbit) console.log(p) //nothing
```

Polymorphism

All instance objects *inherit* the .toString() method from the ancestor Object object.

```
alert(rabbit2.toString())   // [object Object]
```

But *we can change that method* at both the prototype and the instance level. Let's first see at the prototype level

```
Rabbit.prototype.toString = function(){
    return "(prototype.toString) Rabbit: name="+this.name+", this.greet.name="+
           this.greet.name  ;
}

// (prototype.toString) Rabbit: name=tammy, this.greet.name=greet
alert(rabbit2.toString())
```

After this example, it has to come as no surprise that we can as well modify it at the instance level, thereby overriding prototype definitions:

```
rabbit2.toString = function(){
  var msg = "object Rabbit2 = {"+
                      "name:'"+this.name+"'"+
                      ", greeting:'"+this.greeting+"'}" ;
  return msg;
}

alert(rabbit2.toString());
```

This is a form of what's called **polymorphism** (from greek, 'many forms'), whereby the same method/function `toString` can be used with different objects.

Another, apparently different, form of polymorphism is where the same *call to a funciton/method* can be made providing different, *explicit* type of arguments:

```
function add(x,y){ return x+y ; }

add(3,7) // 10
add('three','seven') // threeseven
```

In this example, *we have used the fact that the* ***operator + is polymorphic****, thereby making our function* ***add*** *also polymorphic*: it works both, with numbers as well as with strings!

This apparent two distinct examples of polymorphism are really but the same type: when we write `rabbit1.toString()` we are implicitly passing the argument `this` to the method `toString`; it's just a different flavor of syntax that kind of hides this fact -that is, the difference is just *eye-candy*.

Inheritance

But how can we define new objects that *inherit* properties and methods from other, previously defined objects?

```
function Human(text) {
  Primate.call(this, text);
}
Human.prototype = Object.create(Primate.prototype);
Human.prototype.talk = function(width, height) {
  var result = [];
  for (var i = 0; i < height; i++) {
    var line = this.text[i] || "";
    result.push(repeat(" ", width - line.length) + line);
  }
  return result;
};
```

Now a `Human` shares all properties and methods of a `Primate`, except that it has its own version of `talk` method.

Question: Use the code of `matrix.js` as a library and define a new object, called `Vector`, that inherits from `Matrix` all properties. Instantiate one such a vector, called it `v`, print it and print its transpose.

Answer: We assume that the testbed html file is located in the same directory as the `matrix-ng.js` library. We can load the latter by adding the line `<script src="matrix-ng.js"></script>`. Now, inside a `<script>_write_here_code_that_follows</script>`, we write the following code

```
function Vector(data,name=""){
    Matrix.call(this,[data.length,1],data,name)
}
Matrix.prototype = Object.create(Matrix.prototype)

var v = new Vector([3,7,2], "v");
print(v)
print(v.t() )
```

where we assume that the function `print` is available and writes its argument as a string on the page.

Notice, how we have used the concept of Object-Oriented Programming (OOP) in order to

1. **Abstract away details**: The dimensions of a vector are always n-rows by 1 column, e.g., 2×1, $3 \times 1, \ldots$. Whence, we don't need to specify the number of columns when instantiating one such a new object `Vector`: We just need to say how many rows it has! But this information is already contained in the data array that we are passing: it's its length, i.e., the number of elements it contains!
2. **Simplify code developing**: For the same previous reason, it helps create an easier "language".
3. **Increases modularity, thus, code reuse**: We do not touch our `matrix-ng.js` code developed and *tested* earlier! We just use it here by importing it through a `script` tag.
4. **Decreases programming mistakes**: The new code we need to write is much simplier, as we don't need to rewrite from scratch all the code for a Matrix, but just use it.
5. **Decreases the chance of accidental code overwrite**: If we were to add this new code to the file containing the definition for Matrix, there is a higher chance that we corrupt the code of Matrix just by accident.
6. **Helps troubleshooting**: As `matrix-ng.js` is already tested (it should be!), and we know all ins and outs of what it does, if we get problems with the new code, the likelihood that the issue lies in a part of the

`matrix-ng.js` code is extremely smaller than the possibility it lies on the new code we are writing.

Private properties and methods: The Bank

See the second solution to question 7 of Term 2 test 3 below.

Abstraction, Modularity and Factories: Towards a network of neurons

Building a perceptron as defined above is straightforward. Of course, a perceptron is pretty much useless on its own. The importance of perceptrons lies as building blocks of what's called *Neural Networks* or *Deep Learning*.

Whence, the key feature of a perceptron that really must be included in its implementation is that of connecting to other neurons and form a network. But how do we do this? and how can we do this efficiently?

We will see how answering these questions will force us to change our initial, trivial implementation of a perceptron!

Piping perceptrons

The testbed shows how we would like for now to connect two perceptrons together. The actual code is further below.

```
<!doctype html>
<html>
  <!-- HTML template page: perceptron.html
       http://msantos.sdf.org/G12/Term2/perceptron.html
  -->
  <head>
      <title>Perceptron test bed</title>
      <meta charset="utf-8">
      <script>
        //Helper function
        function print(){
          var endl="<br>"; if(arguments[-1] === "") {endl=""; arguments.splice(-1,1)}
          var msg=""
          for(var i = 0 ; i<arguments.length ; i++) msg += arguments[i]
          document.body.innerHTML += msg + endl;
        }
      </script>
```

```
    <style>
      body { font-family: monospace ; }
    </style>
  </head>

  <body>


    <script src="perceptron.js"></script>
    <script>

        var p1 = new Perceptron([1,1],"p1") ;
        print(p1," ",p1.w);
        var ix = [0.5,-.25] ;
        print( p1.name+"("+ix+")="+p1.s(ix) ) ;

        var p2 = new Perceptron([-1,0.5],"p2") ;
        print(p2," ",p2.w);

        var pp = p2.pipe(p1,0) ;
        print(pp," ",pp.w);

        var ixx = [-0.25,-.25,-.25] ;
        print( pp.name+"("+ixx+")="+pp.s(ixx) ) ;




    </script>
  </body>
</html>
```

And here is the JavaScript code. The first two objects we build are called *factories*. Their sole role is to provide specific type of functions at the request of the programmer and in a well controlled way.

We could have build all their code right into that of a perceptron, of course.

Question: What do you think is the advantage in not doing so, i.e., in using a factory?

Question: Each of the factories, in turn, could have contained all their code in their constructors. Yet we don't do so. Explain the advantage of doing the way we do.

```
function AggregatorFactory(perceptron,order="weighted"){
    this.i = AggregatorFactory.prototype.process(perceptron,order);
    this.name = order ;
```

```
}
AggregatorFactory.prototype.process = function(perceptron,order){
    var aggregator ;
    switch(order){
        case "weighted":
        case "average":
            aggregator = function(x) {
                if( x.length != perceptron.w.length ) {
                    msg = perceptron.name +
                          " ERROR : #input neurons "+
                          x.length+" != #weights "+perceptron.w.length ;
                    console.log(msg);
                    alert(msg);
                    return msg
                }
                var z = 0;
                for(var i = 0 ; i<x.length ; i++) z += x[i]*perceptron.w[i];
                return z;
            }
            break;
        default:
            msgs = "ERROR : Ordered an unknown aggregator function"
            console.log(msg);
            alert(msg);
            aggregator = msg
    }
    return aggregator
}

function ActivationFactory(perceptron,order="step"){
    this.a = ActivationFactory.prototype.process(perceptron,order) ;
    this.name = order ;
}

ActivationFactory.prototype.process = function(perceptron,order){
    var activation;
    switch(order){
        case "step":
            activation = function(z) { y=0 ; if(z>0) y=1; return y}
            break
        default:
            msg = perceptron.name +
                  " ERROR : Ordered an unknown activation function"
            console.log(msg);
            alert(msg);
            activation = msg
```

```
    }
    return activation;
}


function Perceptron(weights,name="ptron",aggregator="weighted",activation="step"){
    this.w = weights ;
    this.aggregator = new  AggregatorFactory(this,aggregator)  //
    this.activation = new  ActivationFactory(this,activation)
    this.name=name ;
}

Perceptron.prototype.s = function(x) {
    return  this.activation.a( this.aggregator.i(x) ) ;
}

Perceptron.prototype.toString = function(){
    return this.name+"::"+
           this.w.length+
           ">-|"+this.aggregator.name+"--"+this.activation.name+"|->" ;
}

Perceptron.prototype.pipe = function (ptron,idendrite){
    if( idendrite < 0 || idendrite >= this.w.length){
        msg = this.name +
              " ERROR: can't plug a neuron into input drendrite "+
              idendrite
        console.log(msg)
        alert(msg)
        return msg
    }
    var isig1 = ptron.w.length
    var w = this.w.concat(ptron.w) ; //join this.w+ptron.w in this order
    w.splice(idendrite,1) //remove dendrite idendrite
    var ptmp = new Perceptron(w,this.name+ptron.name);
    var oldai = this.aggregator.i
    ptmp.aggregator.i = function(x){
            return oldai(x.slice(0,-isig1).concat([ptron.s(x.slice(-isig1))]))
    }
    ptmp.aggregator.name = isig1+p1.name+">"+p2.name ; //means compound
    return ptmp
}
```

Solutions to Exercises

Matrix and Matrix multiplication

We will write he `Javascript` code in a file called `matrix.js` and we will load and test it in an html file called `matrix.html`.

The `matrix.html` file:

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Matrices in JS</title>
    <script src="matrix.js"></script>
<!-- matrix.js and this present html file must lie in the same folder
     for this to work
-->
</head>
<body>
<h1>Matrices in JS</h1>
<script>
var M = new Matrix( [2,2], [0, 1, 1, 0]) ;
var v = new Matrix( [2,1], [1, -1] ) ;

document.body.innerHTML += "<p> "+ "Matrix M =" ;
M.print();
document.body.innerHTML += "</p>" ;
document.body.innerHTML += "<p> Matrix v =" ;
v.print();
document.body.innerHTML += "<p> "+ "M v =" ;
var Mv = M.mult(v) ;
Mv.print() ;
document.body.innerHTML += "</p>" ;

document.body.innerHTML += "<p> "+ "Matrix A =" ;
var A = new Matrix( [2,3] , [3,8,5,1,-2,6]) ;
A.print() ;
document.body.innerHTML += "</p>" ;

document.body.innerHTML += "<p> "+ "Matrix B =" ;
var B = new Matrix( [3,3] , [9, 8, 7, 6, 5, 4, 3, 1, 2]) ;
B.print() ;
document.body.innerHTML += "</p>" ;

document.body.innerHTML += "<p> "+ "AB =" ;
```

```
var AB = A.mult(B) ;
AB.print() ;
document.body.innerHTML += "</p>" ;

document.body.innerHTML += "<p> "+ "BA =" ;
var BA = B.mult(A) ;
BA.print() ;
document.body.innerHTML += "</p>" ;


</script>
</body>
</html>
```

There are two ways to solve this problem, given the stated requirements.

1. The constructor stores the actual matrix elements as an array of arrays of rows. Example: `[[1,2],[3,4],[5,6]]` represents a matrix with 3 rows and 2 columns.
2. The constructor stores the matrix as a 1-dimensional array with the actual elements sorted from left->right and top->bottom.

Both cases will require different implmentation details, of course. However, *the way a user would call our matrix constructor and its methods* **will be the same**!.

1. **Implementation as array of arrays**:

```
//The matrix constructor
/*
Input:
    dim :: array of 2 integers
    data:: array of all matrix elements from left->right and top->bottom

"Output": (not really a return value, mind you!)
    Object with properties:
        dim :: an array of 2 integers
        data:: an array of arrays, the latter each containing
               one row of the matrix from top->bottom.
*/
function Matrix(dim=[0,0], data=[]){
    // In `var m = new Matrix([2,1],[-1,-7])` this is a reference to `m` !!
    this.dim = dim ;
    this.data = [] ;

    var x = [];
    for(var i=0 ; i < data.length ; i++){
        x.push( data[i] ) ;
```

```
        if ( (i+1) % dim[1] == 0 ) {
            this.data.push(x);
            x=[];
        }

    }

}

Matrix.prototype.print = function(){
    var line="" ;
    for(var i =0 ; i < this.data.length ; i++) {
        for( var j=0 ; j  < this.data[i].length ; j++ ){
            line += this.data[i][j] + " " ;
        }
        line += "<br>" ;
    }
    document.body.innerHTML += line ;
}


Matrix.prototype.mult = function(B){
    if( this.dim[1] != B.dim[0] ) {
        var msg = "<p>ERROR: Can't multiply. #columns doesn't match #rows! </p>" ;
        document.body.innerHTML +=  msg ;
        console.log(msg);
        return -1;
    }

    var dim = [this.dim[0] , B.dim[1] ] ;
    var data = [] ;

    //loop over rows of caller matrix ('A', aka. the 'this' matrix)
    for(var i = 0 ; i < this.dim[0] ; i++ ) {

        // loop over columns of callee matrix ('B')
        for(var j = 0 ; j < B.dim[1] ; j++) {
            var sum=0;

            //multiply row i of 'this' with column j of 'B'
            for(var k = 0 ; k < this.dim[1] ; k++ ) {
                sum += this.data[i][k] * B.data[k][j] ;
            }
            data.push( sum ) ;
        }
```

```
        }
        return new Matrix( dim, data ) ;
    }
```

2. **Implementation as a 1-dimensional array**: The main changes affect only the implementation of matrix printing and, most importantly, the calculation of `sum` in the matrix multiplication which now requires a different way of referencing the row and column elements: `sum += this.data[i*this.dim[1] + k] * B.data[k*B.dim[1] + j] ;`

```
//The matrix constructor
/*
Input:
    dim :: array of 2 integers
    data:: array of all matrix elements from left->right and top->bottom

"Output": (not really a return value, mind you!)
    Object with properties:
        dim :: an array of 2 integers
        data:: a 1-dimensional array w/ actual elements sorted left->right,top->bottom
*/
function Matrix(dim=[0,0], data=[]){
    // In `var m = new Matrix([2,1],[-1,-7])` this is a reference to `m` !!
    this.dim = dim ;
    this.data = data ;
}

Matrix.prototype.print = function(){
    var line="" ;
    for(var i =0 ; i < this.data.length ; i++) {
        line += this.data[i] + " " ;
        if( (i+1) % this.dim[1] == 0 ) line += "<br>" ; //new after at end of row
    }
    document.body.innerHTML += line ;
}


Matrix.prototype.mult = function(B){
    if( this.dim[1] != B.dim[0] ) {
        var msg = "<p>ERROR: Can't multiply. #columns doesn't match #rows! </p>" ;
        document.body.innerHTML +=  msg ;
        console.log(msg);
        return -1;
    }

    var dim = [this.dim[0] , B.dim[1] ] ;
    var data = [] ;
```

```
        //loop over rows of caller matrix ('A', aka. the 'this' matrix)
        for(var i = 0 ; i < this.dim[0] ; i++ ) {
            // loop over columns of callee matrix ('B')
            for(var j = 0 ; j < B.dim[1] ; j++) {
                var sum=0;
                //multiply row i of 'this' with column j of 'B'
                for(var k = 0 ; k < this.dim[1] ; k++ ) {
                    sum += this.data[i*this.dim[1] + k] * B.data[k*B.dim[1] + j] ;
                }
                data.push( sum ) ;
            }
        }
        return new Matrix( dim, data ) ;
    }
```

Library: The Matrix implementation using named objects.

We want to use an independent `HTML` testbed separate from the code that implements the matrix algebra, i.e, separate from `matrix.js`. In addition, this implementation of the matrix object will include a name label.

```
//Matrix Library

/*The matrix constructor

Input:
    dim :: array of 2 integers
    data:: array of all matrix elements from left->right and top->bottom

"Output": (not really a return value, mind you!)
    Object with properties:
        dim :: an array of 2 integers
        data:: a 1-dimensional array w/ actual elements sorted left->right,top->bottom
*/
function Matrix(dim=[0,0], data=[], name="noname"){
    // In `var m = new Matrix([2,1],[-1,-7])` this is a reference to `m` !!
    this.dim = dim ;
    this.data = data ;
        this.name = name;
    //consistency test
        if( this.dim[0]*this.dim[1] != this.data.length){
        var msg="ERROR:"+
                " matrix "+this.name+" :"+
                        " Incompatible data length ("+this.data.length+")"+
```

```
                        " w/ matrix dimensions ("+this.dim[0]+"x"+this.dim[1]+") ";
        console.log(msg);
        alert(msg); //only in browser
    }
}

Matrix.prototype.t = function() {
    var tdata = [];
    for(var j = 0 ; j < this.dim[1] ; j++){
        for(var i=0 ; i < this.dim[0]; i++){
            tdata.push(this.data[j+this.dim[1]*i] );
        }
    }
    return new Matrix([this.dim[1],this.dim[0]], tdata,this.name+"†") ;
}

Matrix.prototype.toString = function(){
        var nl = "<br>";
        var om = "[ ";
        var cm = " ]";
    var line=this.name+om ;
        for(var i=0 ; i<this.name.length ; i++){om = " "+om}

    for(var i =0 ; i < this.data.length ; i++) {
        line += this.data[i] + " " ;
        if( (i+1) % this.dim[1] == 0 ) {
        //add new line after at end of row
            if( (i+1) == this.data.length ) line += cm+nl ;
            else line += cm+nl+om ;
                }
    }
    return line ;
}


Matrix.prototype.mult = function(B){
    if( this.dim[1] != B.dim[0] ) {
        var msg = "<p>ERROR: Can't multiply. #columns doesn't match #rows! </p>" ;
        document.body.innerHTML +=  msg ;
        console.log(msg);
        return -1;
    }

    var dim = [this.dim[0] , B.dim[1] ] ;
    var data = [] ;
```

```
    //
    //loop over rows of caller matrix ('A', aka. the 'this' matrix)
    for(var i = 0 ; i < this.dim[0] ; i++ ) {
        // loop over columns of callee matrix ('B')
        for(var j = 0 ; j < B.dim[1] ; j++) {
            var sum=0;
        //multiply row i of 'this' with column j of 'B'
            for(var k = 0 ; k < this.dim[1] ; k++ ) {
                sum += this.data[i*this.dim[1] + k] * B.data[k*B.dim[1] + j] ;
            }
            data.push( sum ) ;
        }
    }
    return new Matrix( dim, data , this.name+B.name) ;
}

Matrix.prototype.add = function(B){
    if ( this.dim[0] != B.dim[0] || this.dim[1] != B.dim[1] ) {
        var msg = "<p>ERROR: Can't add matrices "+
              this.name+" and "+B.name+
              ". Dimensions must be the same! </p>"
        document.body.innerHTML +=  msg ;
        console.log(msg);
        return -1;
    }

    var dim = this.dim ;
    var data = [] ;

    for(var i = 0 ; i < this.data.length ; i++)
            data.push( this.data[i] + B.data[i] ) ;

    return new Matrix( dim, data, "("+this.name+"+"+B.name+")")

}
```

Complex Numbers

In analogy of the Matrix algebra, we will implement a library for dealing with complex (aka imaginary) numnbers.

```
/*The complex number constructor

Input:
        vec:: array of 2 numbers
```

```
        name :: string

"Output": (not really a return value, mind you!)
        Object with properties:
            x, y :: real & imaginary components
            name :: string, label of our complex number
                If the input vec variable is {0,1], the
                name **must** be `i`
                If not given any input name, it needs to get a default one of `z`
.*/

function Complex(vec,name="z"){
    this.x = vec[0] ;
    this.y = vec[1] ;
    if ( this.x == 0 ) name="i";
    this.name = name ;
}

Complex.prototype.toString = function(){
    var str  = "" ;
    if ( this.x == 0 ) {
        str = "i"
        if ( this.y != 1 )  str = this.y+"i" ;
    }
    else if ( this.y == 0 ){
        str = this.x ;
    }
    else str = this.name+"("+this.x+","+this.y+")"

    return str ;
}

Complex.prototype.add = function(v) {
    return new Complex( [ this.x+v.x, this.y+v.y],
                "("+this.name+"+"+v.name+")" );
}


Complex.prototype.mult = function(v){
    return new Complex( [ this.x*v.x - this.y*v.y, this.x*v.y+this.y*v.x] ,
              this.name+v.name ) ;
}

Complex.prototype.t = function(){ //...the conjugate
    return new Complex( [ this.x, -this.y ],
              this.name+"†" );
```

```
}

Complex.prototype.norm = function(){ //...or magnitude of a complex number
    return Math.sqrt( this.x*this.x+this.y*this.y);
}

Complex.prototype.inv = function() {
        var iv = new Complex([1/Math.pow(this.norm(),2),0],"") //1/norm^2
        var t = this.t().mult(iv)
        return new Complex([t.x,t.y],this.name+"'") ;
}

Complex.prototype.phase = function(){
    return Math.atan( this.y/this.x);
}
```

Fizz Quizz

Problem statement

Write a constructor Vector that represents a vector in two-dimensional space. It takes x and y parameters (numbers), which it should save to properties of the same name.

Give the Vector prototype two methods, neg, plus and minus. The first returns the opposite of a vector (x,y), i.e., it returns (-x,-y). The other two take another vector as a parameter and return a new vector that has the sum or difference of the two vectors' (the one in this and the parameter) x and y values.

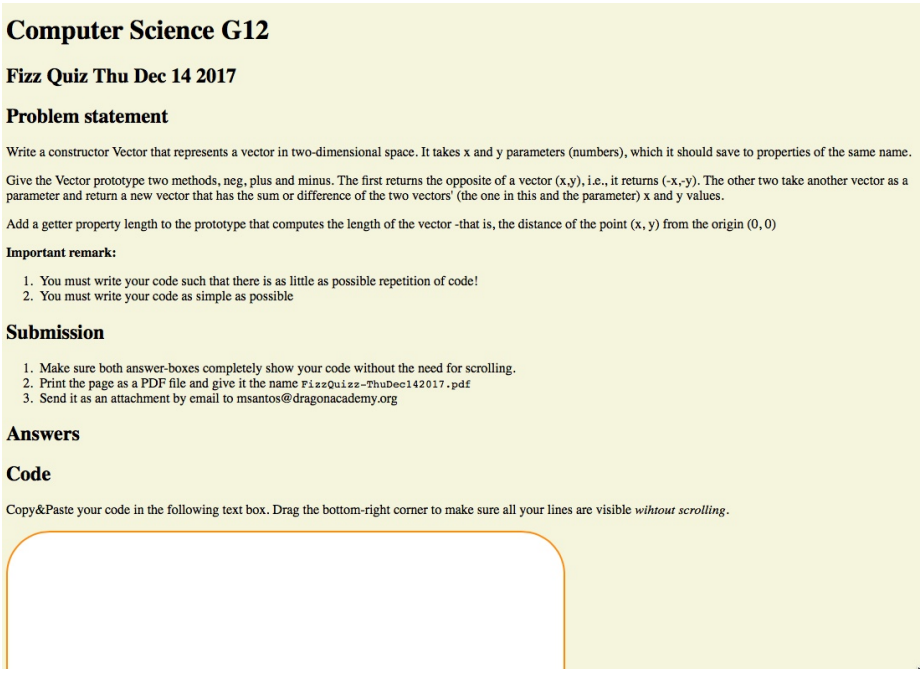
Add a getter property length to the prototype that computes the length of the vector -that is, the distance of the point (x, y) from the origin (0, 0)

Important remark:

- You must write your code such that there is as little as possible repetition of code!
- You must write your code as simple as possible.

The quiz HTML file

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Fizz Quizz: Thu Dec 14</title>
```



Computer Science G12

Fizz Quiz Thu Dec 14 2017

Problem statement

Write a constructor Vector that represents a vector in two-dimensional space. It takes x and y parameters (numbers), which it should save to properties of the same name.

Give the Vector prototype two methods, neg, plus and minus. The first returns the opposite of a vector (x,y), i.e., it returns (-x,-y). The other two take another vector as a parameter and return a new vector that has the sum or difference of the two vectors' (the one in this and the parameter) x and y values.

Add a getter property length to the prototype that computes the length of the vector -that is, the distance of the point (x, y) from the origin (0, 0)

Important remark:

1. You must write your code such that there is as little as possible repetition of code!
2. You must write your code as simple as possible

Submission

1. Make sure both answer-boxes completely show your code without the need for scrolling.
2. Print the page as a PDF file and give it the name `FizzQuizz-ThuDec142017.pdf`
3. Send it as an attachment by email to msantos@dragonacademy.org

Answers

Code

Copy&Paste your code in the following text box. Drag the bottom-right corner to make sure all your lines are visible *wihtout scrolling*.

Figure 1: FizzQuizzThuDec14

```
    <style>
        body { background: beige; width: 80%; margin-bottom: 15em;}
        textarea { width: 60em; height: 40em; border: 2px solid darkorange;
                   border-radius: 52px;
        }
    </style>
</head>
<body>

<h1>Computer Science G12</h1>
<h2>Fizz Quiz Thu Dec 14 2017</h2>
<section>
<h1>Problem statement</h1>
<p>
Write a constructor Vector that represents a vector in two-dimensional space.
It takes x and y parameters (numbers), which it should save to properties of the same name.
</p>
<p>
Give the Vector prototype two methods, neg, plus and minus.  The first returns
the opposite of a vector (x,y), i.e., it returns (-x,-y).
The other two take another vector as a parameter
```

```
and return a new vector that has the sum or difference of the two vectors'
(the one in this and the parameter) x and y values.
</p>
<p>Add a getter property length to the prototype that computes the length of
the vector -that is, the distance of the point (x, y) from the origin (0, 0)
</p>
<p>
<strong>Important remark:</strong>
<ol>
 <li>You must write your code such that there is as little as possible
     repetition of code!
 </li>
 <li>You must write your code as simple as possible</li>
</ol>
</p>
<h1>Submission</h1>
<ol>
    <li>Make sure both answer-boxes completely show your code without the need
        for scrolling.
    </li>
    <li>Print the page as a PDF file and give it the name
        <code>FizzQuizz-ThuDec142017.pdf</code>
    </li>
    <li>Send it as an attachment by email to
        <email>msantos@dragonacademy.org</email>
    </li>
</ol>
</section>
<section>
<h1>Answers</h1>
<h2>Code</h2>
<p>Copy&amp;amp;Paste your code in the following text box. Drag the bottom-right
corner to make sure all your lines are visible <em>wihtout scrolling</em>.
</p>
<textarea id="ans_code" name="ans_code" ></textarea>
<h2>Test file</h2>
<p>Copy&amp;amp;Paste your HTML code in the following text box. It must load
your previous code and must contain the necessary test code.
Drag the bottom-right corner to make sure all your lines are visible
<em>wihtout scrolling</em>.
</p>
<textarea id="ans_test" name="ans_test" ></textarea>
</section>

</body>
</html>
```

Solution

```
/*
 We will save this JS code into a file called Vectors.js
*/

function Vector(x,y,name="v"){
   this.x=x ;
   this.y=y ;
   this.name = name ;
}

Vector.prototype.plus = function(v){
   return new Vector(this.x+v.x, this.y+v.y, this.name+"+"+v.name) ;
}
Vector.prototype.neg = function(){
   return new Vector(-this.x, -this.y, "(-"+this.name+")" );
}
Vector.prototype.minus = function(v){
   //in order not to repeat code we use neg()
   return this.plus( v.neg() )
}
Vector.prototype.toString = function(){
   return this.name+"("+this.x+", "+this.y+")" ;
}
Vector.prototype.length = function(){
   return Math.sqrt( this.x*this.x + this.y*this.y ) ;
}
```

with a testbed

```
<!doctype html>
<html>
  <head>
      <title>Test of Vectors</title>
      <meta charset="utf-8">
      <script>
      /* Introduction to Programming
            with
         JavaScript
      */
        //Helper function
        function print(x){
          document.body.innerHTML += x + "<br>";
        }
      </script>
```

```
    <script src="Vectors.js" >
    </script>
  </head>
  <body>
    <script >
      var v1 = new Vector(1,2,"v1");
      var v2 = new Vector(3,4,"v2");
      print(v1)
      print(v1.length() )
      print(v2)
      print(v2.length() )
      print(v2.neg())
      print(v2.neg().length())
      print(v1.plus(v2))
      print(v1.plus(v2).length() )
      print(v1.minus(v2))
      print(v1.minus(v2).length())
    </script>
  </body>
</html>
```

Assignment 4

Due date: Mon Dec 18 2017

Problem

Extend the code for vectors such that the following **test code**:

```
var v1 = new Vector(1,2,"v1");
var v2 = new Vector(3,4,"v2");
print(v1)
print(" |v1|="+v1.length() +"<br><br>")
print(v2)
print(" |v2|="+v2.length() +"<br><br>")
print(v2.neg())
print(" |-v2|="+v2.neg().length()+"<br><br>")
print(v1.plus(v2))
print(" |v1+v2|="+v1.plus(v2).length() +"<br><br>")
print(v1.minus(v2))
print(" |v1-v2|="+v1.minus(v2).length()+"<br><br>")

print("Angle of v1: v1.angle()="); print(v1.angle() +" deg<br><br>")
```

```
print("dot product of v1 and v2: v1.dot(v2)="); print(v1.dot(v2) +"<br><br>") ;
print("Angle between v1 and v2: v1.angle(v2)="); print(v1.angle(v2) +" deg<br><br>") ;
```

prints on the page the **results**:

```
v1(1, 2) |v1|=2.23606797749979

v2(3, 4) |v2|=5

(-v2)(-3, -4) |-v2|=5

v1+v2(4, 6) |v1+v2|=7.211102550927978

v1+(-v2)(-2, -2) |v1-v2|=2.8284271247461903

Angle of v1: v1.angle()=63.43494882292201 deg

dot product of v1 and v2: v1.dot(v2)=11

Angle between v1 and v2: v1.angle(v2)=10.304846468766044 deg
```

Note: You'll need to modify slightly the print function

Submission

Submit as attachment the augmented Vectors.js file as well as the corresponding Vectors.html one.

Term 2, Test 1. Tue Dec 19 2017

Name:

Note: Write your full name in capitals.

Submission

Write your code in a text file (extension `txt`) and submitted as attachment via email.

Problems

Note: The reference to the `print` function is the one available in the HTML Live Editor, which you can use.

1. Implement a counter object in Javascript such that the following code prints `32`. **Constraints**: The only instance properties are the methods `inc()` and getCount()‘.

```
var i = new Counter(30)
i.inc()
i.inc()
print("i:"+i)
```

2. You are provided a `matrix.js` script file and a `matrix-t2-test1.html` HTML test file. You'll need to modify them to satisfy the constraints stated below.

   Both files can be downloaded from http://msantos.sdf.org/G12/Term2 and are also attached in the appendix section below in this document.

   1. Implement the `toString` method such that the test file prints

```
0 1
1 0

3
3
```

   2. Modify the `toString` method you just wrote, as well as the Matrix constructor such that the following code

```
var M = new Matrix([2,2],[0,1,1,0],"M")
var v = new Matrix([2,1],[3,3],"v")
var nona = new Matrix([2,1],[5,7])

print(M)
print(v)
print(nona)
```

      prints

```
M[ 0 1 ]
 [ 1 0 ]

v[ 3 ]
 [ 3 ]

noname[ 5 ]
      [ 7 ]
```

   3. **Consistency test**: Implement a *consistency test* such that we get an error message printed in an alert and in the console saying `ERROR: matrix A : Incompatible data length (4) w/ matrix dimensions (2x3)` if we try to instantiate a matrix where the dimensions doesn't fit with the data length as, e.g.,

```
var M = new Matrix([2,3],[0,1,1,0],"M")
```

4. **Transpose**: Implement the method `t` (lowercase 't') that returns the transpose of a matrix. Follow these examples:

   This code

   ```
   var A = new Matrix([2,3],[11,12,13,21,22,23],"A")
   var w = new Matrix([2,1],[3,3],"w")
   print(A)
   print(A.t())
   print(w)
   print(w.t())
   ```

   prints

   ```
   A[ 11 12 13 ]
    [ 21 22 23 ]

   A†[ 11 21 ]
     [ 12 22 ]
     [ 13 23 ]

   w[ 3 ]
    [ 3 ]

   w†[ 3 3 ]
   ```

   where the symbol † (*dagger*) denotes the transposed matrix.

Appendix

Matrix HTML template file: matrix-t2-test1.html

```
<!doctype html>
<html>
  <!-- HTML template page: matrix-t2-test1.html
       http://msantos.sdf.org/G12/Term2/matrix2-t2-test1.html
  -->
  <head>
      <title>Matrix test bed</title>
      <meta charset="utf-8">
      <script>
        //Helper function
        function print(x){
          document.body.innerHTML += x + "<br>";
        }
      </script>
```

```
        <style>
          body { font-family: monospace ; }
        </style>
    </head>

    <body>


<script src="file://PATH/matrix.js"></script>
    <script>

var M = new Matrix([2,2],[0, 1, 1, 0])
var v = new Matrix([2,1],[3,3])
print(M)
print(v)



    </script>
    </body>
</html>
```

where `PATH` depends on wehter you run on Windows or Mac:

a. **Mac**: `PATH=/Users/'your-use-name'/Desktop`
b. **Windows**: `PATH=/C:/Users/'your-use-name'/Desktop`

and where the script file `matrix.js`, which can be downloaded from http://msantos.sdf.org/G12/Term2/, contains the matrix template library that can be found in the appendix below.

Matrix Template Library: matrix.js

```
//Matrix Library

/*The matrix constructor

Input:
        dim :: array of 2 integers
        data:: array of all matrix elements from left->right and top->bottom

"Output": (not really a return value, mind you!)
        Object with properties:
            dim :: an array of 2 integers
            data:: a 1-dimensional array w/ actual elements sorted left->right,top->bottom
*/
```

```
function Matrix(dim=[0,0], data=[]){
        this.dim = dim ;
        this.data = data ;
}

Matrix.prototype.mult = function(B){
        if( this.dim[1] != B.dim[0] ) {
                var msg = "<p>ERROR: Can't multiply. #columns doesn't match #rows! </p>" ;
                document.body.innerHTML +=  msg ;
                console.log(msg);
                return -1;
        }

        var dim = [this.dim[0] , B.dim[1] ] ;
        var data = [] ;

        //
        for(var i = 0 ; i < this.dim[0] ; i++ ) {
                for(var j = 0 ; j < B.dim[1] ; j++) {
                        var sum=0;
                        for(var k = 0 ; k < this.dim[1] ; k++ ) {
                            sum += this.data[i*this.dim[1] + k] * B.data[k*B.dim[1] + j] ;
                        }
                        data.push( sum ) ;
                }
        }
        return new Matrix( dim, data ) ;
}

Matrix.prototype.toString = function(){

}
```

Solution

You may as well download the solutions following the links:

- http://msantos.sdf.org/G12/Term2/Test-T2-1-G12-OOP-solution.html
- http://msantos.sdf.org/G12/Term2/Test-T2-1-G12-OOP-solution.js.txt

```
//Problem 1
function Counter(c=0){
    var count = c ;
    this.inc = function() { count++ ;}
    this.getCount = function(){ return count; }
}
```

```
//to make sure that we can call print on an instance of Counter objects we need
// to implement a specific code for the 'toString' method that it inherents from the
// ancestor object 'Object'
Counter.prototype.toString = function(){ return this.getCount(); }

//Problem 2
/* Question 2.1
Matrix.prototype.toString() {
        var line="" ;
        for(var i =0 ; i < this.data.length ; i++) {
                line += this.data[i] + " " ;
                if( (i+1) % this.dim[1] == 0 )  line += "<br>" ; //new line at end of row
        }
        return line ;
}
*/


// The following code contains the answers to questions 2.2, 2.3 and 2.4

//Matrix Library

/*The matrix constructor

Input:
    dim :: array of 2 integers
    data:: array of all matrix elements from left->right and top->bottom

"Output": (not really a return value, mind you!)
    Object with properties:
        dim :: an array of 2 integers
        data:: a 1-dimensional array w/ actual elements sorted left->right,top->bottom
*/
function Matrix(dim=[0,0], data=[], name="noname"){
    this.dim = dim ;
    this.data = data ;
        this.name = name;
    //consistency test    v--- Question 2.3
        if( this.dim[0]*this.dim[1] != this.data.length){
        var msg="ERROR:"+
                " matrix "+this.name+" :"+
                        " Incompatible data length ("+this.data.length+")"+
                        " w/ matrix dimensions ("+this.dim[0]+"x"+this.dim[1]+") ";
        console.log(msg);
        alert(msg); //only in browser
```

```
    }
}

// Question 2.2
Matrix.prototype.toString = function(){
        var nl = "<br>";
        var om = "[ ";
        var cm = " ]";
    var line=this.name+om ;
        for(var i=0 ; i<this.name.length ; i++){om = " "+om}

    for(var i =0 ; i < this.data.length ; i++) {
        line += this.data[i] + " " ;
        if( (i+1) % this.dim[1] == 0 ) {
            //new after at end of row
            if( (i+1) == this.data.length ) line += cm+nl ;
            else line += cm+nl+om ;
                }
    }
    return line ;
}

// Question 2.4
Matrix.prototype.t = function() {
    var tdata = [];
    //these 2 loops very similar to the mult method
    for(var j = 0 ; j < this.dim[1] ; j++){
        for(var i=0 ; i < this.dim[0]; i++){
            tdata.push(this.data[j+this.dim[1]*i] );
        }
    }
    return new Matrix([this.dim[1],this.dim[0]], tdata,this.name+"†") ;
}

// We were not ask to do so, but for consistency we would need
// to modify 'mult' in order
// to deal properly with the name of the matrix, as was done in the vector case.
Matrix.prototype.mult = function(B){
    if( this.dim[1] != B.dim[0] ) {
        var msg = "<p>ERROR: Can't multiply. "+
                  "#columns doesn't match #rows! </p>" ;
        document.body.innerHTML +=  msg ;
        console.log(msg);
        return -1;
    }
```

```
    var dim = [this.dim[0] , B.dim[1] ] ;
    var data = [] ;

    //
    //loop over rows of caller matrix ('A', aka. the 'this' matrix)
    for(var i = 0 ; i < this.dim[0] ; i++ ) {
        // loop over columns of callee matrix ('B')
        for(var j = 0 ; j < B.dim[1] ; j++) {
            var sum=0;
            //multiply row i of 'this' with column j of 'B'
            for(var k = 0 ; k < this.dim[1] ; k++ ) {
                sum += this.data[i*this.dim[1] + k] * B.data[k*B.dim[1] + j] ;
            }
            data.push( sum ) ;
        }
    }
        var nname = "("+this.name+")("+B.name+")"
    return new Matrix( dim, data , nname) ; //dealing with the new name
}
```

Testbed HTML

```
<!doctype html>
<html lang="en">
  <head>
      <title>Title of the document</title>
      <meta charset="utf-8">
      <script>
      /* Introduction to Programming
            with
         JavaScript
      */
        //Helper function
        function print(x=""){
          document.body.innerHTML += x + "<br>";
        }
      </script>
<style> body { font-family: monospace } </style>
  </head>

  <body>


<!-- long, full path version
<script src="file:///Users/msantos/Dragon/Course/G12/
```

```
Test-T2-1-G12-OOP-solution.js.txt">
</script>
Remark: there cannot be any white spaces between G12/ and Test-T2...
-->
<script src="Test-T2-1-G12-OOP-solution.js.txt"> </script>
    <script>
print(" Computer Science G12")
print(" Term 2, Test 1.")
print("Date: Tue Dec 19 2017")
print()

print("Problem 1")
var i = new Counter(30)
i.inc() ; i.inc()
print( "i:"+i )
print()

print("Problem 2")
var A = new Matrix([2,3],[11,12,13,21,22,23],"A")
print( "Question 2.1/2.2")
print(A)
print()
print( "Question 2.3" )
var B = new Matrix([2,1], [1,2,3],"B")
print("You should have seen at least the altert window with the error message popping up.")
print()
print( "Question 2.4")
print(A.t() )
print()
print("Extra")
print( A.mult(A.t() ) )
print( A.t().mult(A) )
    </script>
  </body>
</html>
```

Term 2, Test 2. Wed Jan 10 2018

Name:

Note: Write your full name in capitals.

Submission

Write your code in a text file (extension `txt`) and submitted as attachment via email.

Problems

Note: The reference to the `print` function is the one available in the HTML Live Editor, which you can use.

1. Implement a counter object in Javascript such that the following code prints `32`. **Constraints**: The only instance properties are the methods `inc()` and getCount()‘.

   ```
   var i = new Counter(30)
   i.inc()
   i.inc()
   print("i:"+i)
   ```

2. You are provided a `complex.js` script file and a `complex-test.html` HTML test file. You'll need to modify them to satisfy the constraints stated below.

   Both files can be downloaded from http://msantos.sdf.org/G12/Term2 and are also attached in the appendix section below in this document.

   Implement the necessary code so that the test code runs as expected.

Appendix

Complex HTML template file: complex-test.html

```
<!doctype html>
<html>
  <!-- HTML template page: complex-test.html
       http://msantos.sdf.org/G12/Term2/complex-test.html
  -->
  <head>
      <title>Complex test bed</title>
      <meta charset="utf-8">
      <script>
        //Helper function
        function print(x){
          document.body.innerHTML += x + "<br>";
        }
      </script>
```

```
      <style>
        body { font-family: monospace ; }
      </style>
  </head>

  <body>


<script src="file://PATH/complex.js"></script>
    <script>

var z = new Complex([2,2],"z")
var v = new Complex([-2,1],"v")
var w = new Complex([-7,4],"w")
var i = new Complex([0,1],"v")
print(z)    //prints z(2,2)
print(v)    //    v(-2,1)
print(w)    //    w(-7,4)
print(i)    //    i

var zt = z.t()     // z†(2,-2)
var vt = v.t()    // v†(-2,-1)
var it = i.t()    // i†

print( z.norm() )     // 2 sqrt(2) ~ 2.8284
print( v.norm() )    // sqrt(5) ~ 2.2361
print( i.norm() )    // 1

var z_v = z.add(v)    // (z+v)
var wxz_v = w.mult(z.add(v))    // w(z+v)

print( z.mult(i) )     //  zi(-2,2)
print( i.mult(i) )    //  -1

print( w.phase() )    // angle with x-axis
    </script>
  </body>
</html>
```

where `PATH` depends on wehter you run on Windows or Mac:

a. **Mac**: `PATH=/Users/'your-use-name'/Desktop`
b. **Windows**: `PATH=/C:/Users/'your-use-name'/Desktop`

and where the script file `complex.js`, which can be downloaded from http://msantos.sdf.org/G12/Term2/, contains the complex template library that can be found in the appendix below.

Complex Template Library: complex.js

```
//Complex Numbers Library

/*The complex number constructor

Input:
        vec:: array of 2 numbers
    name :: string

"Output": (not really a return value, mind you!)
        Object with properties:
            x, y :: real & imaginary components
        name :: string, label of our complex number
        If the input vec variable is [0,1], the
                name **must** be `i`
            If not given any input name, it needs to get a default one of `z`
*/
function Complex(){
}

Complex.prototype.toString = function(){

}

Complex.prototype.mult = function(B){

}
```

Term 2, Test 3 Wed. Jan. 24 2018

Name:

All questions have the same weight towards the final mark of this test.

Questions:

1. Suppose that `School` is the name of an object type. What is the meaning of the statement `var shs = new School()`? That is, what does the computer do when it executes that statement?
2. What is meant by the terms *instance variable* and *instance method*? Give an example of each.

3. In JavaScript we define a *constructor* in order to define a new object type, e.g., `function Voice(msg=""){ this.msg=msg}`. In other languages like `C++` or `Python` that would be called *defining a class*. We could called it that as well in JS. Whence OOP languages are said to use *classes* and *objects*. Explain what you think is the distinction between these two terms and how are they related? How did we called all this during the course? What distinctions did we make?
4. What is a constructor in JS? What is its purpose?
5. What is in essence an object in JS? Give an example of an object in JS. Try to come up with one that is as simple as possible.
6. You need to write an as simple as possible, but complete object The class represens a counter that counts 0,1,2,... The name of the constructor should be `Counter`. It should work as `var count = new Counter(); count.increment() ; count.getValue()` so that the last statement would return the value of *that particular* counter at that moment.
7. Design an object in JS to represent a bank account. Include the following properties: Name of depositor, type of account, account number and balance. In addition it should have methods for: assigning initial values, depositing an amount, withdrawing an amount *after* checking balance, displaying the name and balance.

Solution:

1. It *instantiates an object of type* `School`. This entails reserving memory for all instance properties and methods. Prototype properties and methods have already stored in memory at the time of their definition.
2. Any instance variable has a copy in each instance of the an object type, whence the expression "instance variable (property) and instance method". Example: `var shs1 = new School() ; var sh2 = new School(); shs1.name="Newt School" ; shs2.name = "Dragon school".` The property `name` of type string appears in both instances of `School` object, `shs1` and `shs2`, yet their values are unique for each instance and there is no clash between both.
3. Here we are talking about the distinction between an ***instance*** *of an object of a given type* and that *type definition*. In our course, we talked about *object definition* and *instance of an object*.
4. A constructor is a function that either defines instance properties or methods in its body, or object properties and methods through its prototype. In this respect, a constructor, as everything else in JS, is an object in itself.
5. An object in JS can be seen simply as a dictionary of properties-values pairs. i Example: `var myob = {name: "silly object", price: 1000, getPrice = function(){ return this.price}`
6. It will be accepted if you define an instance property containing the current value of the counter. However, the neatest solution is hiding the counter value by way of using a **closure** (remember the example of `adder(n)` and

```
var add3 = adder(3)):
```

```
function Counter(c=0){
    var a=c
    this.increment(){ ++a }
    this.getValue(){ return a }
}
```

7. We will present here two approaches. The first one is a completely acceptable one given the context of the test. The second serves to showcases how we can implement *private* properties and methods that are only accessible through a very specific interface (set of methods). In other OO-languages like `C++` `Scala` or `C#` this is achieved through the use of the keywords `public` and `private`.

```
function Bank(){
    Bank.prototype.init("")
}

Bank.prototype.init = function(name,balance=0,type="checking"){
    this.name = name
    this.acn = Math.floor(100000*Math.random()+0.5)
    this.type = "checking"
    this.balance = balance
}

Bank.prototype.toString = function(){
    return this.name+" : "+this.acn+" : "+this.balance
}

Bank.prototype.withdraw = function(x=0){
    if ( this.balance >= x ) {
        this.balance -= x
        return x
    }
}
Bank.prototype.deposit = function(d=0){ this.balance += d }
```

A more complete method is as follows. We will call this "bank" design as `BankSec` for "secure banking". What is meant is that all allowed access to the account is achieved through a very specific and well defined set of methods (the **interface of the object**) and *no direct access to key properties by the user of this code would be allowed*. In addition we add a *fancy* feature of keeping a history of all transactions and error messages -maybe a better name would be a `log` of all transactions.

```
//Secure bank version
```

```
function BankSec(name="noname"){
    var acn=-1
    var type = -1
    var balance = -1
    var history = ""
    BankSec.prototype.init(name,acn,type,balance,history="")
}
BankSec.prototype.init = function(name="noname",balance=-1,
                                  type=-1,acn=-1,history=""){
    this.getHistory = function(){ return history}
    function updateHistory(transaction){ history += "<br>" + transaction}
    this.name = function(){ return name }
    this.acn = function(){
                    if( acn == -1) {
                       acn = Math.floor( Math.random()*100000 + 0.5 )
                    }
                    return acn
    }
    this.balance = function(){
        if( balance == -1 ) balance = 0
        return balance
    }
    this.type = function(){
        if ( type == -1 ) type="checking"
        return type
    }
    this.withdraw = function(x){
        if( x < balance ) balance -= x
        else {
                        trans = "Operation not allowed! (widthdraw "+
                                 x+") Insufficient balance "+
                                 this.balance()
                        console.log( trans )
                        updateHistory( trans )
        }
        trans = "Withdrowal of $"+x
        updateHistory( trans )
        return balance
    }
    this.deposit = function(x){
        if( x >= 0 ) balance += x
      else {
                        trans = "Operation not allowed! depositing "+
                                "negative amount "+x
                        console.log( trans )
                        updateHistory( trans )
```

```
        }
        trans = "Deposit of $"+x
        updateHistory( trans )
        return balance
    }
    this.toString = function(){
                    return this.name()+" : "+this.acn()+" : "+
                           this.type() + " :: "+ this.balance()
    }
    this.setType = function(ntype){
           if( ntype != "checking" && ntype != "savings"){
                               trans = "Operation not allowed! "+
                                       "Account type unknown "+ntype
                               console.log( trans )
                               updateHistory( trans )
                               return this.type()
           }
           trans = "Changed account type from "+type+" to "+ntype
           updateHistory(  trans )
           type = ntype
           return this.type()
    }
}
```

As a testbed for this code we could use the following HTML file

```
<!doctype html>
<html>
    <head>
        <title>Bank js</title>
        <meta charset="utf-8">
        <script>
            function print(){
                var  msg = ""
                for(var i=0 ; i<arguments.length ; i++)
                   msg += arguments[i]
                if ( arguments[-1] != "") msg += "<br>"
                document.body.innerHTML += msg
            }
        </script>
        <style>
            body {
                background-color: black;
                color: gold;
            }
        </style>
        <script src="bank.js"></script>
```

```
        <script src="bankSec.js"></script>
    </head>
    <body>

    <script>
        /*
print("Testing bank.js":)
        var acc = new Bank()
        acc.init("Nic")
        acc.deposit(100)
        print(acc)
        acc.name="666"
        print(acc)
        // Direct access to properties is still possible though! Bad!!
        acc.balance += 100
        print(acc)
        acc.acn="000"
        print(acc)
        acc.init("Nic")
        print(acc)
        var sam = new Bank()
        sam.init("sam",100)
        print(sam)
        */
        //bank 2 : secure
print("Testing bankSec.js":)
        var nicSec = new BankSec()
        nicSec.init("nicSec")
        print(nicSec)
        nicSec.setType("savings")
        nicSec.deposit(1000)
        print(nicSec)
        nicSec.withdraw(2000)
        nicSec.withdraw(200)
        print(nicSec)
        print(nicSec.getHistory() )
        //nicSec.updateHistory( "hacked" )
        nicSec.history = "Hacked again!"
        print(nicSec.getHistory() )
        print(nicSec.history)
    </script>
    </body>
</html>
```

All code can be downloaded from http://msantos.sdf.org/G12/Term2 The output of this test of `BankSec` is the following:

```
nicSec : 47216 : checking :: 0
nicSec : 47216 : savings :: 1000
nicSec : 47216 : savings :: 800

Changed account type from checking to savings
Deposit of $1000
Operation not allowed! (widthdraw 2000) Insufficient balance 1000
Withdrowal of $200

Changed account type from checking to savings
Deposit of $1000
Operation not allowed! (widthdraw 2000) Insufficient balance 1000
Withdrowal of $200
Hacked again!
```