# Homework: Functions and Function Expressions

This document defines the homework assignments from the ["Advanced JavaScript" Course @ Software University](https://softuni.bg/trainings/1099/Advanced-JavaScript-March-2015). Please submit as homework a single zip / rar / 7z archive holding the solutions (source code) of all below described problems.

## Function Arguments

Create a function **printArgsInfo()** with no parameters. For each argument passed to it, the function should return its type and its value.

* Call the function with different number and type of arguments

|  |  |
| --- | --- |
| **Sample Code** | **Sample Output** |
| printArgsInfo(2, 3, 2.5, -110.5564, false) | 2 (number)  3 (number)  2.5 (number)  -110.5564 (number)  false (boolean) |
| printArgsInfo(null, undefined, "", 0, [], {})  // Note that [].toString() returns "" | null (object)  undefined (undefined)  (string)  0 (number)  (array)  [object Object] (object) |
| printArgsInfo([1, 2], ["string", "array"], ["single value"]) | 1,2 (array)  string,array (array)  single value (array) |
| printArgsInfo("some string", [1, 2], ["string", "array"], ["mixed", 2, false, "array"], {name: "Peter", age: 20}) | some string (string)  1,2 (array)  string,array (array)  mixed,2,false,array (array)  [object Object] (object) |
| printArgsInfo([[1, [2, [3, [4, 5]]]], ["string", "array"]]) | 1,2,3,4,5,string,array (array) |

## call() and apply()

Call the function **printArgsInfo()** using **call()** and **apply()** as follows:

* Using **call()** without arguments
* Using **call()** with arguments
* Using **apply()** without arguments
* Using **apply()** with arguments

## Test the Value of this

Create a function **testContext()** with no parameters. The function should print the **this** object. Call the function three times:

* From the global scope
* Inside another function
* As an object (for example, using **new testContext()**)

Write a comment inside your code explaining the different behaviour.

## Adding Numbers Using Functions

Write a function **add()** which adds numbers in a functional manner. It should work as follows:

* add(1) // returns 1
* add(2)(3) // returns 5
* add(1)(1)(1)(1)(1) // returns 5
* add(1)(0)(-1)(-1) // returns -1

We should also be able to store the result and reuse it:

var addTwo = add(2);  
console.log(addTwo); // 2  
console.log(addTwo(3)); // 5

|  |  |
| --- | --- |
| **Sample Code** | **Sample Output** |
| add(1) | 1 |
| add(2)(3) | 5 |
| add(1)(1)(1)(1)(1) | 5 |
| add(1)(0)(-1)(-1) | -1 |
| var addTwo = add(2); console.log(addTwo); | 2 |
| var addTwo = add(2);  console.log(addTwo(3)); | 5 |
| var addTwo = add(2);  console.log(addTwo(3)(5)); | 10 |

## \* Function Composition

**Function composition** ([article](http://www.mathsisfun.com/sets/functions-composition.html)) of two functions f() and g() is defined (mathematically) as g(f(x)), where x is the input to the function f(). For example, if **f(x) = sin(x)** and **g(x) = x^2**, their composition will be **sin(x^2)**.

Write a function **compose(f, g)** which returns the function composition of f() and g(). **compose(f, g)** (where f and g are functions) should return another function, and **compose(f, g)(x)** should return the value of the composition applied to the argument **x**. For simplicity, assume that f() can take only one argument.

For the sample output, assume these functions are defined:

function add(x, y) {  
 return x + y;  
}  
function addOne(x) {  
 return x + 1;  
}  
function square(x) {  
 return x \* x;  
}

|  |  |
| --- | --- |
| **Sample Code** | **Sample Output** |
| compose(addOne, square)(5) | 26 // addOne(square(5)) = 5 \* 5 + 1 = 26 |
| compose(addOne, add)(5, 6) | 12 |
| compose(Math.cos, addOne)(-1) | 1 // cos(0) = 1 |
| compose(addOne, Math.cos)(-1) | 1.5403023058681398 |
| var compositeFunction = compose(Math.sqrt, Math.cos);  console.log(compositeFunction(0.5));  console.log(compositeFunction(1));  console.log(compositeFunction(-1)); | 0.9367937670001721  0.7350525871447157  0.7350525871447157 |

## DOM Traversal

You are given an HTML file **index.html**. Write a function that traverses all **child elements** of an **element** by a **given CSS selector** and prints all found elements in the format:

***<Element name>***: id="***<id>***", class="***<class>***"

Print each element on a new line. **Indent** child elements.

|  |  |
| --- | --- |
| **Sample Code** | **Sample Output** |
| var selector = ".birds";  traverse(selector); | h2: class="birds-heading"  ul: class="birds-list"  li: class="bird"  li: class="bird"  ul:  li: class="sub-bird"  li: class="sub-bird"  li: id="eagle" class="bird"  li: class="bird"  li: class="bird"  li: class="bird" |

## DOM Manipulation

Create an **IIFE module** for working with the **DOM** **tree**. The module should support the following operations:

* **appendChild(element, child) – adds** а DOM **element** to a **parent element**
* **removeChild(element, child) – removes** a **child element** from a **parent element**
* **addHandler(element, eventType, eventHandler) – attaches** an **event** to a given **selector** by given **event type** (string) and **event hander**
* **retrieveElements(selector) – retrieves** elements by a given CSS **selector**

**element** and **child** can be selectors (strings) or DOM elements. If any selector is passed to the module, it should find (and manipulate) all elements the selector corresponds to.

The module should reveal only its **public methods** (i.e. everything else should be encapsulated).

|  |
| --- |
| **Sample Code** |
| var liElement = document.createElement("li");  // Appends a list item to ul.birds-list  domModule.**appendChild**(liElement,".birds-list");  // Removes the first li child from the bird list  domModule.**removeChild**("ul.birds-list","li:first-child");  // Adds a click event to all bird list items  domModule.**addHandler**("li.birds", 'click', function(){ alert("I'm a bird!") });  // Retrives all elements of class "bird"  var elements = domModule.**retrieveElements**(".bird"); |

## Console Module

Create a **module** for working with the console object. The module should support the following functionality:

* Writing a line to the console
* Writing a line to the console using **formatting** (with placeholders, like in C#). The first placeholder is {0}, the second is {1} and so on
  + For simplicity, assume that no message will need to contain the string "{0}", i. e. all numbers between { and } should be considered placeholders
  + Throw an error if there are not enough arguments passed to the function. For example, specialConsole.writeLine("Message: {1}", hello"); should not work
* Writing **errors**, **warnings** and **messages** to the console with and without format
  + Check the documentation for console.error(), console.warn() and console.info()
* If there are some objects passed to any of the functions, their **toString()** method must be called

|  |
| --- |
| **Sample Code** |
| specialConsole.**writeLine**("Message: hello");  specialConsole.**writeLine**("Message: {0}", "hello");  specialConsole.**writeLine**("Object: {0}", { name: "Gosho", toString: function() { return this.name });  specialConsole.**writeError**("Error: {0}", "A fatal error has occurred.");  specialConsole.**writeWarning**("Warning: {0}", "You are not allowed to do that!");  specialConsole.**writeInfo**("Info: {0}", "Hi there! Here is some info for you.");  specialConsole.**writeError**("Error object: {0}", { msg: "An error happened", toString: function() { return this.msg }); |