# WEEK 1: CRITICAL CONCEPTS OF JAVASCRIPT FUNCTIONS AND ARRAYS

The Whole Is Greater than the Sum of the Parts

# Global variables (avoid)

- Variables declared outside of any function are called global.
  - visible from any function
  - Avoid globals

```
let userName = 'John';

function showMessage() {
  userName = "Bob"; // (1) changed the outer variable
  let message = 'Hello, ' + userName;
  alert(message);
}
```

# Shadowing (avoid)

- a local variable is given the same name as a more global one
  - global hidden or shadowed by the local one

```
function showMessage(from, text) {
  from = '*' + from + '*'; // make "from" look nicer
  console.log(from + ': ' + text);
}
let from = "Ann";
showMessage(from, "Hello"); // *Ann*: Hello
```

# Returning a value

- return can be in any place of the function. When the execution reaches it, the function stops, and the value is returned to the calling code (assigned to result above).
  - return without a value. That causes the function to exit immediately.
  - A function with an empty return or without it returns undefined

# Functions as values

- Functions are first-class objects in JavaScript
  - stored in a variable, object, or array.
  - passed as an argument to a function.
  - returned from a function

  - ( ) after function name is an important syntactic construct in an expression context
    - means that there is a function and it should be executed

# Exercise

Refactor the code by adding appropriate ()’s so that “Hello” appears twice in the console.

```
function sayHi() {
 console.log( "Hello" );
}
const myHi = sayHi;
console.log( sayHi ); // shows the function code
function higherOrder() { return sayHi; }
higherOrder();
```

# Callback functions

- arguments showOk and showCancel of ask are called callback functions or callbacks
- pass a function and expect it to be “called back” later

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}
function showOk() {
  console.log( "You agreed." );
}
function showCancel() {
 console.log( "You canceled the execution." );
}
ask("Do you agree?", showOk, showCancel);
```

```
//more succinct function expression version (anonymous functions)
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

ask( "Do you agree?",
  function() { alert("You agreed."); },
  function() { alert("You canceled the execution."); }
```

# Exercise

```
"use strict";
const assert = require("assert");
/* eslint-disable */
/* write functions executor, add, and mult as defined by the test below */

describe("executor", function(){
    it ("tests add", function(){
        assert.strictEqual(executor(add, 5, 10), 15);
    });
    it("tests mult", function(){
        assert.strictEqual(executor(mult, 5, 10), 50);
    });
});
```

# Arrow functions

- used in the same way as function expressions

```
//function expression
let sum = function(a, b) {
  return a + b;
};

//equivalent arrow function
let sum = (a, b) => a + b;

//only one argument, then parentheses around parameters can be omitted
let double = x => 2 * x;

//no arguments, parentheses should be empty (but they should be present):
let sayHi = () => alert("Hello!");

//if body has { } brackets then must use return
let sum = (a, b) => { return a + b; }
```

# Loops

- Favor for..of if do not need indices
  - Less opportunities for bugs
  - Problems with beginning or end of array indices, etc
  - code is simpler and cleaner without indices

# Multi-dimensional arrays

Arrays can have items that are also arrays. We can use it for multidimensional arrays, for example to store matrices

```
const matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];
alert( matrix[1][1] ); // 5, the central element
```

# LESSONS 6 & 7
# OBJECTS

Knowledge is the Wholeness of Knower, Known, and Process of Knowing

# Main Points

1. Objects and properties
    1. Object literals
    2. Dot and square bracket notations
2. Objects are references:
3. Object methods, this :
4. Constructor functions, new operator

# Exercise

- Create 3 objects, student1, student2, student3
  - property studentId :  s101, s102, s103 respectively
  - property quiz answers:  [1, 1, 2,4], [2, 1, 2,2], [3, 1, 3,4] respectively

- push the students into an array, quiz

- write a function, gradeQuiz, that takes the quiz array and an array of correct answers, e.g., [3,1,2,4] and returns an object that has properties with the names of each studentId and the value of the property will be their score on the quiz (one point for each correct answer)
  - expect  { s101: 3, s102: 2, s103: 3 }

- write a function that takes the grade report object and returns the average score of all the students

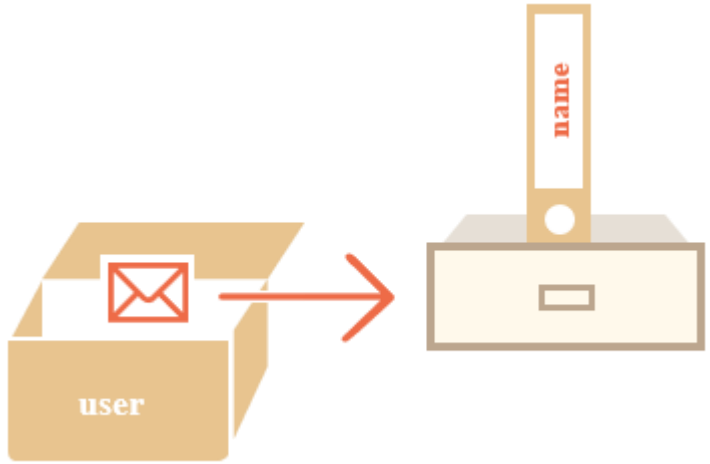
# Access by reference

- fundamental differences of objects vs primitives is that they are stored and copied “by reference”
- Primitive values: strings, numbers, booleans – are assigned/copied “as a whole value”.

```
let message = "Hello!";
let phrase = message;
```

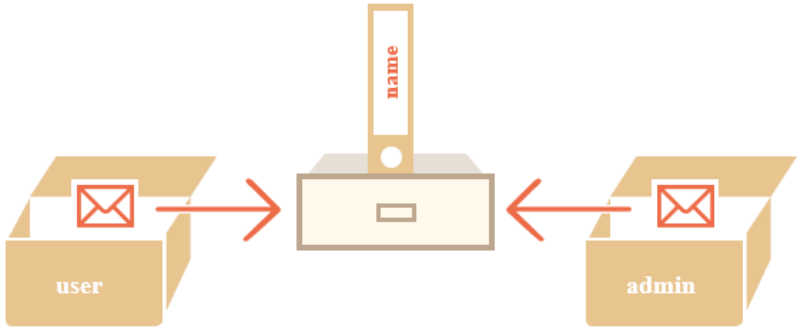

- An object variable stores not the object itself, but its “address in memory”, in other words “a reference” to it.

```
let user = {
  name: "John"
};
```




- When an object variable is copied – the reference is copied, the object is not duplicated.

```
let user = { name: "John" };
let admin = user; // copy the reference
```

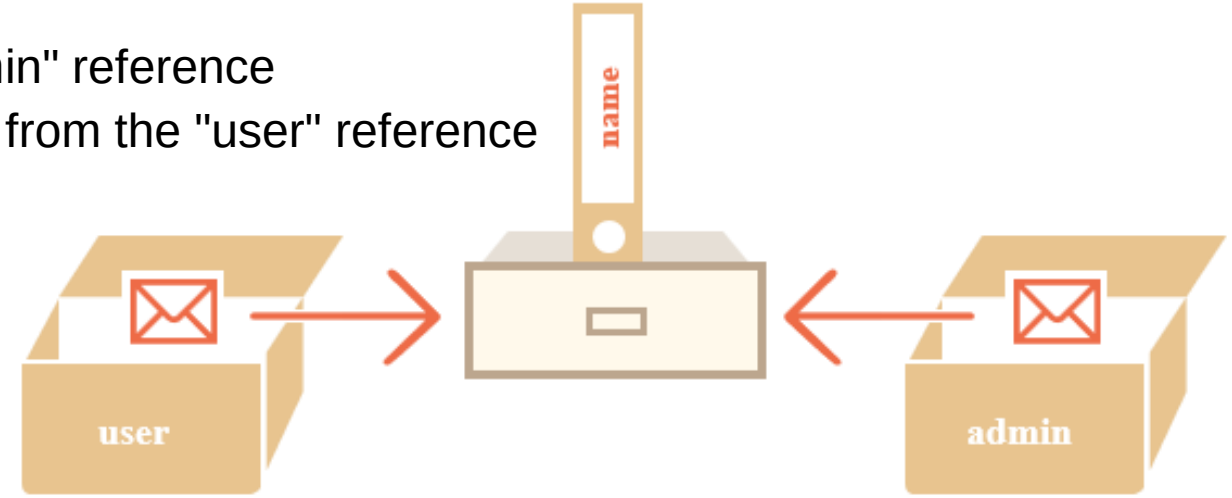


# Access by reference

- can use any variable to access the cabinet and modify its contents

- **there is only one object**
  - If have two keys and use one of them (admin) to get into it.
  - later use the other key (user) we will  see changes

```
let user = { name: 'John' };
let admin = user;
admin.name = 'Pete'; // changed by the "admin" reference
alert(user.name); // 'Pete', changes are seen from the "user" reference
```



# Methods

- Objects are usually created to represent entities of the real world, like users, orders and so on
- in the real world, a user can act
  - actions are represented in JavaScript by functions in properties.

```
let user = {
  name: "John",
  age: 30
};

user.sayHi = function() {
  alert("Hello!");
};

user.sayHi(); // Hello!
```

- A function that is the property of an object is called its *method*.

# “this” in methods

- It’s common that an object method needs to access the information stored in the object to do its job.
  - the code inside user.sayHi() may need the name of the user.
  - To access the object, a method can use the this keyword.
  - The value of this is the object “before dot”, the one used to call the method.

```
let user = {
  name: "John",
  age: 30,
  sayHi:  function() {
    // "this" is the "current object"
    alert(this.name);
  }
};

user.sayHi(); // John
```

# Calling without an object: this == undefined

- We can even call the function without an object at all:

```
"use strict";
function sayHi() {
    this.lat = 41.00;
    this.long = -92.96;
    console.log(this);
  }
sayHi(); // undefined
```

- In this case this is undefined in strict mode. If we try to access this.name, there will be an error.

- In non-strict mode the value of this in such case will be the global object
  - historical behavior that "use strict" fixes.

# Exercise

Create a calculator

Create an object calculator with three methods:

getValues(operand1, operand2) takes two values and saves them as object properties.
sum() returns the sum of saved values.
mul() multiplies saved values and returns the result.

```
let calculator = {
  // ... your code ...
};

calculator.getValues(5, 10);
console.log( “expect 15 : “, calculator.sum() );
console.log(“expect 50 : “, calculator.mul() );
```

# Constructor functions, operator “new”

- Constructor functions technically are regular functions.

- two conventions:
  - start with capital letter
  - executed only with "new" operator

```
function User(name) {
  this.name = name;
  this.isAdmin = false;
}

let user = new User("Jack");
```

# new User(...) does the following steps:

1. A new empty object is created and assigned to this.
2. The function body executes. Usually it modifies this, adds new properties to it.
3. The value of this is returned.

➢In other words, new User(...) does something like:

```
function User(name) {
  // this = {};  (implicitly)

  // add properties to this
  this.name = name;
  this.isAdmin = false;

  // return this;  (implicitly)}
```

# Homework exercise

```
describe("calculator", function() {
  let calculator;
  before(function() {

    calculator = new Calculator();
    calculator.setValues(2, 3);
  });

  it("when 2 and 3 are entered, the sum is 5", function() {
    assert.equal(calculator.sum(), 5);
  });

  it("when 2 and 3 are entered, the product is 6", function() {
    assert.equal(calculator.mul(), 6);
  });


});
```

# LESSONS 8 & 9
# DATA TYPES

Knowledge Has Organizing Power

# Numeric conversion using a plus + or Number()

- Numeric conversion using a plus + or Number() is strict. If a value is not exactly a number, it fails:

- alert( +"100px" ); // NaN

# Iterate: forEach

- run a function for every element of the array.
  - result of the function (if it returns any) is thrown away and ignored
  - Intended for some side effect on each element of the array
    - print or alert or post to database

```
// for each element call alert
["Bilbo", "Gandalf", "Nazgul"].forEach(function(character){console.log(character)} );

["Bilbo", "Gandalf", "Nazgul"].forEach((item, index, array) => {
  console.log (`${item} is at index ${index} in ${array}`);
});
```

# Exercise

- use forEach to log all the even elements of an array to the console

[1,5,16,3, 108]

# indexOf/lastIndexOf and includes

- arr.indexOf, arr.lastIndexOf and arr.includes have same syntax and do essentially same as string counterparts
  - operate on items instead of characters:
  - arr.indexOf(item, from) – looks for item starting from index from, and returns the index where it was found, otherwise -1.
  - arr.lastIndexOf(item, from) – same, but looks for from right to left.
  - arr.includes(item, from) – looks for item starting from index from, returns true if found.

```
let arr = [1, 0, false];

alert( arr.indexOf(0) ); // 1
alert( arr.indexOf(false) ); // 2
alert( arr.indexOf(null) ); // -1

alert( arr.includes(1) ); // true
```

# filter

- Apply function to each item in array and return new array of all that pass the filter

```
let results = arr.filter(function(item, index, array) {
  // if true item is pushed to results and the iteration continues
});
```

```
let users = [
  {id: 1, name: "John"},
  {id: 2, name: "Pete"},
  {id: 3, name: "Mary"}
];
// returns array of the first two users
let someUsers = users.filter(item => item.id < 3);
```

# find and findIndex

- find first element that satisfies a specific condition

```
arr.find(function(item, index, array)
  // if true is returned by passed function, item is returned and iteration is stopped
  // for falsy scenario returns undefined
```

- The function is called for elements of the array, one after another:
  - item is the element.
  - index is its index.
  - array is the array itself.

```
//Let’s find the one with id === 1:
let users = [
  {id: 1, name: "John"},
  {id: 2, name: "Pete"},
  {id: 3, name: "Mary"}
];
let user = users.find(item => item.id ===1);
alert(user.name); // John
```

- .

# Exercise

const numbers = [1, 5, 18, 2, 77, 108];

- use filter, find, and findIndex to find
  - all the even numbers
  - the first even number
  - the index of the first even number

# transform an array

- map
- sort
- reverse
- reduce
- split / join

pure

destructive

# split/join

- **split()** divides a String into an array of substrings,
  - division done by searching for a pattern; provided as the first parameter

- **join()** creates and returns a new string by concatenating all the elements in an array
  - separated by commas or a specified separator string.

# map

- calls function for each element and returns new array of results
- map the passed function onto each element of the array

```
let result = arr.map(function(item, index, array) {
  // returns the new value instead of item
});

let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);
console.log(lengths); // 5,7,6
```

# Exercise

```
let result = arr.map(function(item, index, array) {
  // returns the new value instead of item
});

let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);
console.log(lengths); // 5,7,6

//modify so that it logs array with index: item.length instead of just item.length
console.log("expect 0: 5, 1: 7, 2: 6", lengths);
```

# sort(fn)

- sorts the array in place, changing its element order.

- Default sort converts all arguments to strings

```
let arr = [ 1, 2, 15 ];
// the method reorders the content of arr
arr.sort();
alert( arr );  // 1, 15, 2
```

To use our own sorting order, we need to supply a function as the argument of arr.sort().

```
function compareNumeric(a, b) {
  if (a > b) return 1;
  if (a == b) return 0;
  if (a < b) return -1;
}
let arr = [ 1, 2, 15 ];
arr.sort(compareNumeric);
alert(arr);
```

# Exercise

change comparator function to sort in descending order, then change it to sort in lexicographic descending order

```
function compareNumeric(a, b) {
  if (a > b) return 1;
  if (a == b) return 0;
  if (a < b) return -1;
}
let arr = [ 1, 15, 2 ];
arr.sort(compareNumeric);
console.log(arr);
```

# sort(fn) [2]

- comparison function is only required to return
  - positive number to say “greater” and a
  - negative number to say “less”.
  - “greater goes to the right”
    - “increasing”
- That allows to write shorter functions:

```
let arr = [ 1, 2, 15 ];
arr.sort(function(a, b) { return a - b; });
alert(arr);  // 1, 2, 15
```

# reduce

calculate a single value based on the array.

```
let value = arr.reduce(function(previousValue, item, index, array) {
  // ...
}, [initial]);
```

The function is applied to all array elements one after another and “carries on” its result to the next call.

previousValue – is the result of the previous function call, equals initial the first time (if initial is provided).
item – is the current array item.
index – is its position.
array – is the array.

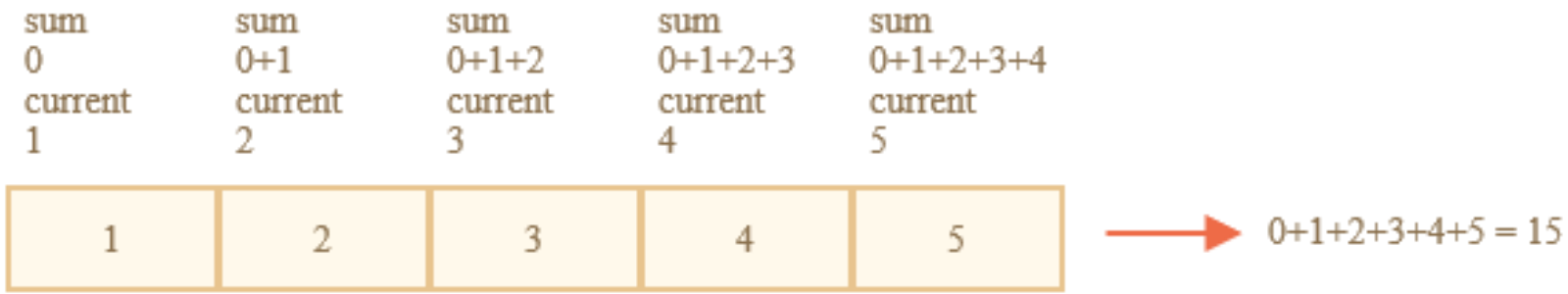
- first argument is the “accumulator” that stores the combined result of all previous execution.
  - at the end it becomes the result of reduce.
- CS303 convention:  always include an initial value for clarity

# reduce [2]

Here we get a sum of an array in one line:

```
let arr = [1, 2, 3, 4, 5];
let result = arr.reduce(function (sum, current) { return sum + current; }, 0);
let result2 = arr.reduce((sum, current) => sum + current, 0);
console.log(result); // 15
console.log(result2); // 15
```

- On the first run, sum is the initial value = 0, and current is first array element = 1
- On the second run, sum = 1, we add the second array element (2) to it and return.
- On the 3rd run, sum = 3 and we add one more element to it, and so on…



# Exercise

- reduce the array to the product of the numbers (“expect 120”)
- reduce the array to the max of the numbers (“expect 5”)

```
let arr = [1, 2, 3, 4, 5];
```

# array methods

- To add/remove elements:
  - push(...items) – adds items to the end,
  - pop() – extracts an item from the end,
  - shift() – extracts an item from the beginning,
  - unshift(...items) – adds items to the beginning.
  - splice(pos, deleteCount, ...items) – at index pos delete deleteCount elements and insert items.
  - slice(start, end) – creates a new array, copies elements from position start till end (not inclusive) into it.
  - concat(...items) – returns a new array: copies all members of the current one and adds items to it. If any of items is an array, then its elements are taken.

- To search among elements:
  - indexOf/lastIndexOf(item, pos) – look for item starting from position pos, return the index or -1 if not found.
  - includes(value) – returns true if the array has value, otherwise false.
  - find/filter(func) – filter elements through the function, return first/all values that make it return true.
  - findIndex is like find, but returns the index instead of a value.

- To iterate over elements:
  - forEach(func) – calls func for every element, does not return anything.

- To transform the array:
  - map(func) – creates a new array from results of calling func for every element.
  - sort(func) – sorts the array in-place, then returns it.
  - reverse() – reverses the array in-place, then returns it.
  - split/join – convert a string to array and back.
  - reduce(func, initial) – calculate a single value over the array by calling func for each element and passing an intermediate result between the calls.

# Summary ‘for’ loops

- ‘for’ is the basic for loop in JavaScript for looping
  - Almost exactly like Java for loop
  - Use this if you need the loop index
- ‘for in’ is useful for iterating through the properties of objects
  - can also be used to go through the indices of an array (unusual use case)
- ‘for of’ is new convenience (ES6) method for ‘iterable’ collections
  - Array, Map, Set, String
  - Use this if usage involves a side effect and do not need loop index
- ‘forEach’ like for .. of but executes a provided function for each element.
  - forEach returns undefined rather than a new array
  - Intended use is for side effects, e.g., writing to output, etc.
- Best practice to use convenience methods when possible
  - Avoids bugs associated with indices at end points
  - map, filter, find, reduce best practice when appropriate