Basics in JavaScript

Macro index

- Structure
- Object
- Class
- Functions
- Client-Side
- Asynchronous Programming

Index

0. Basics
1. Object
2. Regular Expression
3. Array
4. Object
5. Function
6. DOM Manipulation
7. Class and OOP
8. Asynchronous Programming
9. Testing
10. External libraries

General Structure: Primitive Types and Object types

In JavaScript there are two types of values:

- Primitive Types: String, Number, BigInt, Boolean, Symbol, Undefined, Null
- Object type: list of properties (key-value pair), it's mutable and it is created by literals, classes and functions: Object, Array, Function, Map, Date, RegExp, Set, Custom Class Instance
- primitive types are passed by value, object type are passed by reference

General Structure: Numbers

- Integer Literals
- Floating-Point Literals
- Arithmetic in JavaScript
- Binary Floating-Point and Rounding Errors

General Structure: Dates and Times

General Structure: Text

- String Literals
- Escape Sequences in String Literals
- Pattern Matching

General Structure: Boolean Values

General Structure: null and undefined

General Structure: The Global Object

General Structure: Wrapper Objects

General Structure: Immutable Primitive Values and Mutable Object Refereces

General Structure: Type Conversions

- Conversions and Equality
- Explicit Conversions
- Object to Primitive Conversions

General Structure: Variable Declaration

- Repeated and Omitted Declarations

General Structure: Variable Scope

- global variables has global scope
- variabled declared within a function are defined only within the body of the function
- Function parameters also are local variables
- Local variable hides global variables with the same name

General Structure: Var, Let and Const

- var: Declares variables with function or global scope and allow re-declaration and updates within the same scope (to be avoided)
- let: Declares variable with block scope, allowing updated but not re-declaration within the same block
- const: Declares block-scoped variables that cannot be reassigned after their initial assignment

General Structure: Literals

- A literal is a data value that appears directly in a program

```
// These are all literals
var a = 12;
var s = "hello world";
var o = {x:1, y:2};
var ar = [1,2,3,4]
```

General Structure: Variable Declaration

- var variable declaration
-

Regular Expression in JavaScript

Array in JavaScript

1. forEach, Map, Filter
2. Destructuring Array
3. Rest Operator

Array: Destructuring Array

```
const numbers = [10,20,30];

// Destructuring
const [a,b,c] = numbers;

console.log(a); // 10
console.log(b); // 20
console.log(c); // 30
```

Array: Destructuring Array (2)

```
const numbers = [10,20,30];

// Destructuring
const [first, , third] = numbers;

console.log(first); // 10
console.log(third); // 30
```

Array: Destructuring Array (3)

```
const number = [5];
const [x,y = 99] = number;

console.log(x); // 5
console.log(y); // 99
```

Array: Rest Operator and Spread Operator

Rest operator is used to collect data into arrays: Values -> Array

```
function sumRest (...numbers) {
    console.log(numbers);
}
sumRest(1,2,3);
```

Spread operator is used to spred data from array: Array -> Values

```
function sumSpread (numbers) {
    console.log(...numbers);
}
sumSpread([1, 2, 3]);
```

General Structure: Var, Let or Const?

- var: legacy solution
- let:
- const:

Object in JavaScript

1. Basic feature of an object
2. Creating an Object
3. Destructuring an Object

Object in JavaScript: Spread Operator and Rest Operator

```
const user = {
    name:"john",
    surname:"doe",
    age:30,
};
const user_acces { access_time: [], access_ops: []; }
// copy (deep) w/ spread op
const copy_user = {...user};
// merge w/ spread op
const user_data = {user, user_data};
// overwrite props
const updated_user = {...user, age:31};

// Destructuring with rest op
const {nome, ...data} = user;
```

Object in JavaScript: Constructor vs Function

Function and Constructor are very similar in JavaScript, take a look at this example

```
function Person(name) {
    this.name = name;
    this.sayHello = function() {
        console.log("Hi, I'm " + this.name);
    };
}
const p1 = new Person("Anna");
p1.sayHello();
```

Main differences from a fuction: Capitalized name, this keyword, called with new

Object Section

Object Section: Pseudoclassical Pattern

Object Section: Functional Pattern

Object Section: Durable Object

- A durable object is an object that is created with functional style and all of the methods of the object make no use of this or super class.
- A durable object is simply a collectino of functions that act as capabilities

Object Section: How to Create an Object

- Create it as a Object Literal
- Call a pseudoclassical constructor with the `new` operator
- Call Object.create method on a prototype object
- Call a functional constructor

Object section: How to make private properties and method in Object?

- Use the pseudoclassical pattern

Object section: Parts pattern

- We can compose objects out of sets od parts

```
function canTalk(obj) {
  return {
    talk() {
      console.log(`talk`);
      }};}
function canWalk(obj) {
  return {
    walk() {
      console.log(`walk`);}};}
function canFly(obj) {
  return {
    fly() {
      console.log(`fly`);}};}
```

```
function createRobot(name) {
  const base = { name };
  return {
    ...base,
    ...canTalk(base),
    ...canWalk(base)
  };
}

function createDrone(name) {
  const base = { name };
  return {
    ...base,
    ...canFly(base),
    ...canTalk(base)
  };
}
```

Object: Differential Inheritance

When you define a first initial object and then create another object with the same structure by expressing the differences with the first one

```
var myMammal =  {
    name: 'Herb the Mammal',
    get_name: function () {
        return this.name;
    }
    says: function () {
        return this.saying || '';
    }
}
var myCat = Object.create(myMammal);
myCat.name = 'Henrietta';
myCat.saying = 'meow';
myCat.purr = fuction (n) {/*express the purr function*/}
myCat.get_name = function() {/*overwrite the existing function*/}
```

Class and OOP in JavaScript

1. Class
2. Prototype
3. Constructors
4. Types
5. Subclasses
6. Modules
7. Augmenting Classes

Class and OOP: Class

- Class is not an object, it is a template for object
- Constructors work as in Java

Abstract Example of JavaScript Class

```
class ClassName {
    constructor() {...}
    method_1() {...}
    method_2() {...}
    method_3() {...}
}
```

Concrete Example of JavScript Class

```
class Car {
    constructor(name, year) {
        this.name = name;
        this.year = year;
    }

    age(x) {
        return x - this.year;
    }
}

const myCar = new Car("Ford", 2014);
```

Classes are always executed in "strict mode", never in script mode.

Class and OOP: Inheritance

- Inheritance is based on prototype chain (prototype-based inheritance)
- Every object has its own prototype
- To find the method to be executed on a certain object, explore the prototype chain to the null element
- Class is sintactic sugar for prototype. Behind a class definition there is always a prototype

Class and OOP: Prototype

- actual version of the class definition
- Every object is linked to a prototype object from which it can inherit properties
- All object literals are linked to Object.prototype
- When you make a new object, you can select the object that should be its prototype
- Prototypes create the prototype chain, on which is applied the delegation process

Class and OOP: Reflection and Enumeration

- The `for in` statement can loop over all of the property names in an object
- The `for in` loop will include all of the properties and funcitons of the object and of the prototypes
- to filter out some property and/or function use the hasOwnProperty Method and typeof (=== 'function')

```
const person = {
    name: "Tom",
    age: 30
}
console.log(person.hasOwnProperty("name)); // true
```

Class and OOP: Delete

```
delete another_stooge.nickname;
```

This will remove the nickname only for the object on which is invoked, not on to the prototype

Class and OOP: Global Abatement

Don't use global variables, instead put it into a map

```
var MYAPP = {};
MYAPP.stooge {
    "first-name": "Jow",
    "last-name": "Howard"
};

MYAPP.flight = {
    airline: "Oceanic",
    number: 815,
    departure: {
        IATA: "SYD",
        time: "2004-09-22 14:55",
        city: "Sydney"
    },
    arrival {
        IATA: "LAX",
        time: "2004-09-23 10:42",
        city: "Los Angeles"
```

Class and OOP: Subclass Usage

- Inside the constructor you can call super()
- You can use extends keyword to extend one class
- You can use this keyword to referencing the attribute and the method of the class

Class and OOP: Augmenting Classes

- An object inherits properties from its prototype, even if the prototype changes after the object is created
- Augment JavaScript classes simply by adding new methods to their prototype objects
- The prototype object of built-in JavaScript classes is "open", which means that we can add methods to numbers, strings, arrays, functions, and so on

Class and OOP: Classes and Types

- typeof: operator that allow to distinguish among built-in types (null, undefined, boolean, number, string, function and object)
- classof(): access to the class attribute of Object
- class attribute of an object is not modifiable and for your own custom class is always `'Object'`, so
- classof() doesn't work for own-defined class, in these case use one of the following methods: instanceof, constructor property, constructor function name, duck-typing philosophy

Class and OOP: Classes and Types (instanceof op)

- the expression o instanceof c evaluates to true if o inherits from c.prototype
- it doesn't work with primitive type
- One shortcoming is: by instanceof we can test if an object is instance of a certain class, but we cannot derive the class of the object by itself

Class and OOP: Classes and Types (Constructor)

- Another way to identify the class of an object is to simply use the constructor property, that is the public face of the class
- One shortcoming is: JavaScript does not require that every object have a constructor property, sometimes it is accidentally omit, sometimes intentionally
- Sometimes can be useful get the name of the construct instead of the type but not all object have a constructor name defined and not all object have a constructor function with a nam

Class and OOP: Classes and Type (Duck-Typing)

- not ask "what is the class of this object?", instead try asking "what can this object do?"
- The general idea is: look if a certain object have a certain method or a certain property, without knowing if it is of a certain type
- this is the same concept of implementing an interface (e.g. implementing a functionality) instead of extending a class

Class and OOP: Object-Oriented Techniques in JavaScript

- Encapsulation: private attribute with `#`
- Inheritance: with the extends keyword
- Polimorphism: overloading method
- Abstract: throwing exception

Class and OOP: Object-Oriented Techniques (Polymorphism)

```
class Animal {
    name = undefined;
    constructor(name) {
        this.name = name;
    }
    makeSound() {
        console.log("Need to be implemented!");
    }
}

class Tiger {
    constructor(name) {
        super(name);
    }
    makeSound() {
        console.log("Roar!");
    }
```

Class and OOP: Object-Oriented Techniques (Abstract class and)

```
class Animal {
    constructor() {
        if (new.target === Animal) {
            throw new Error("Animal class is abstract!");
        }
    }

    makeSound() {
        throw new Error("Method to be implemented!");
    }
}

class Tiger {
    constructor() {
    }

    makeSound() {
        console.log("Roar!");
```

Class and OOP: Object-Oriented Techniques (Encapsulation)

```
class Person {
    #name = undefined;
    _surname = undefined;

    constructor (name, surname) {
        this.#name = name;
        this._surname = surname;
    }
}
```

The _underscore notation makes the attribute private by convention, but publicly accessible. The #hashtag notation makes the attribute purely private as other OOP language.

Class and OOP: Object-Oriented Techniques (static)

- Example of static method

```
class MathUtils {
    static somma (a, b) {
        return a + b;
    }
}
console.log(MathUtils.somma(2, 3)); // 5
```

Class and OOP: Object Specifiers

Better way of writing constructors:

```
// One way
var myObject = maker(f,l,m,c,s);
// Better way
var myObject = maker({
    first: f,
    last: l,
    state: s,
    city: c
})
```

Function

Function

1. Function Properties, Method and Constructor
2. Recursion
3. Scope
4. Closure
5. Callbacks
6. IIFE
7. Functional Programming

Function: Hoisting

- Hoisting is the mechanism by which you can referred to the function before its declaration

Function: Function Properties, Method and Constructor

- We say that in JavaScript Function are value, and so they have Constructor, Properties and Method
- Constructor: (not popular)

```
function myFunc(a, b) {
    return a + b;
}

const myFunc = new Function('a','b','return a + b');
```

Function: Function Properties, Method and Constructor (2)

- Properties:
  - .length: number of parameters
  - .name: name of the function (void if anonymous)
  - .arguments: array of the arguments of the function
- Custom properties: you can add properties to a function

```
function greet() {}
greet.customProp = "Hello!";
console.log(greet.customProp); // "Hello!"
```

Function: Function Properties, Method and Constructor (3)

- Method of the function:
    - call: call a function on an object
    - apply: as call, but arguments in the array
    - bind: a function to an object
    - toString: return the body of the function

Function: Function Properties, Method and Constructor (Call and Apply)

```
f.call(o, 1, 2);
f.apply(o, [1,2]);

// does the same thing to:
o.m = f;
o.m([1,2]); // o.m(1,2) in the case of call
delete o.m;
```

Function: Function Properties, Method and Constructor (Bind)

- The bind method bind a function f to an object o
- When you invoke bind on function, it will return a new function f, method of o
- Any arguments you pass to the new function are passed to the original function, but performing partial application

```
function sayHi(name) {
    console.log(`Hi, ${name}`);
}

sayHi.call(null,"Jack"); // "Hi, Jack"
sayHi.apply(null,["John"]); // "Hi, John"

const bound = sayHi.bind(null, "Tom");
bound() // "Hi, Tom"
```

Function: Function Properties, Method and Constructor (4)

- The arrow function are lightweight function:
  - they don't have `this` , `arguments` , `super` nor `new.target`
  - they cannot be use as `constructor`
  - the don't have `prototype`

Function: Function Properties, Method and Constructor (5)

- new.target is an attribute that is `true` if the function has been called with the `new` keyword, false otherwise
- this attribute can be used to ensure a function is called with the `new` keyword, as a constructor. This is the right way of realize a constructor by function
- this works also with class constructor

```
function MyObject() {
    if (!new.target) {
        throw new Error ("This function cannot be called without `new`!");
    }
    return myObject();
}
new MyObject(); // the correct way
MyObject(); // the wrong way
```

Function: Augmenting Types

- JavaScripts allows the basic tyes of the language to be augmented
- Adding a method to Object.prototype makes that method available to all objects
- This also works for functions, arrays, strings, numbers, regular expressions and booleans

Function: Exception

General Structure of the exception

```
try {
  // codice che può causare errori
} catch (err) {
  // gestisci l'errore
} finally {
  // (opzionale) codice che viene eseguito sempre
}
```

Function: Type of Exception

- SyntaxError
- ReferenceError
- TypeError
- RangeError
- ...
- Custom Error

Function: Custom Error

Create a custom error:

```
class MyCustomError extends Error {
  constructor(message) {
    super(message);
    this.name = "MyCustomError";
  }
}
throw new MyCustomError("My Custom Error!");
```

Function: Exception in Asynchronous Programming

- use reject inside Promise
- you can pass an Error object inside the reject

```
const p = new Promise ( (resolve, reject) => {
reject(new Error('promise failed!'));
});
p.catch(
    err => {
        console.log(err);
    }
);
```

- use throw if you wanna create custom error
- you can use throw in a Promise
- you can use throw without Promise

Function: Exception in Reject or Throw (2)

- throw insisde a callback function will not be recognized by a catch block, and in this case use reject
- if throw is encountered, the flow is immediately interrupted, meanwhile the reject end the block and then goes on error
- you cannot create custom error with reject
- if you create a reject error, should always be a catch block of the element

Functional Programming: Function on Arrays

- forEach, Map, Filter, Reduce

Functional Programming: Function are First-Class Citizen

Functional Programming: High-Order Function

Functional Programming: Currying

Functional Programming: Memoization

Asynchronous Programming

Asynchronous Programming: Topics

1. Asynchronous Programming by Events
2. Promise - Then, Catch, Finally
3. Async - Await

Testing

1. Basics
2. Jasmine (Jest, Mocha)

External Libraries

External Libraries: Topics

1. Ajax
2. jQuery
3. Fetch
4. Axios
5. Superagent
6. Prototype
7. Node HTTP