

# Web Developer

Programmazione - Javascript e Typescript

Docente: Shadi Lahham

# Object oriented

Advanced Objects

Shadi Lahham - Web development

# Object Oriented

JavaScript in origine non era completamente orientato agli oggetti come Java.

Prima di ES6, usava un approccio basato su prototipi, che permetteva di riutilizzare il comportamento copiando oggetti esistenti.

Dopo ES6, JavaScript ha introdotto classi, moduli e altre caratteristiche orientate agli oggetti più tradizionali, rendendo il linguaggio più compatibile con la OOP.

# Object Oriented Javascript

- JavaScript used to not be fully object-oriented like Java
- It supported object-oriented concepts through prototype-based inheritance
- As the language evolved, especially after ES6, it added more traditional object-oriented features like classes and modules
- This made JavaScript better for object-oriented programming
- Examples of both these approaches will be demonstrated

# Prototype based language

- Before ES6, JavaScript didn't have classes, only objects
- JavaScript used to have a unique approach compared to C# or C++ for object-oriented programming
- It was prototype-based, meaning behavior could be reused by copying existing objects
- In JavaScript, every object inherits from a prototype, defining its functions and members
- Objects could serve as prototypes for creating other objects

# Prototype example

*// Canine is called a **Constructor Function***

*// typeof Canine is 'function'*

```
let Canine = function (latinName) {  
  this.genus = 'Canis';  
  this.latinName = latinName;  
};
```

Canine --> è una FUNZIONE COSTRUTTRICE!. quando la chiami con new, crea un nuovo oggetto.

this --> in questo caso, si riferisce all'oggetto che stiamo CREANDO!!!

*// Use the new keyword to create new instances of this "class"*

```
let dog = new Canine('Canis familiaris'); // { genus: 'Canis', latinName: 'Canis familiaris' }
```

```
let greyWolf = new Canine('Canis lupus'); // { genus: 'Canis', latinName: 'Canis lupus' }
```

normalmente senza NEW

# Prototype example

*// add methods and properties to the prototype of the Constructor Function  
// able to use them on all instances of the "class"*

```
Canine.prototype.howl = function () {  
  console.log('AAAAWWWOOOOOO');  
};
```

```
dog.howl(); // AAAAWWWOOOOOO  
greyWolf.howl(); // AAAAWWWOOOOOO
```

# Prototype example

*// adding methods and properties to an instance does not apply them to all instances*

```
dog.fetch = function () {  
  console.log('dog wants to play fetch!');  
};  
greyWolf.hunt = function () {  
  console.log('grey wolf is hunting its prey');  
};
```

```
dog.fetch(); // dog wants to play fetch!  
dog.hunt(); // Error: dog.hunt is not a function
```

```
greyWolf.fetch(); // Error: greyWolf.fetch is not a function  
greyWolf.hunt(); // grey wolf is hunting its prey
```



PROTOTYPE ESISTE IN TUTTE LE FUNZIONI (funzioni normali + funzioni costruttrici) ma entra in gioco solamente quando si UTILIZZA NEW!

# Prototype chains

Ogni oggetto in JavaScript ha una proprietà interna nascosta, chiamata `[[prototype]]`. Questa proprietà non è direttamente visibile, ma è fondamentale per il funzionamento della catena dei prototipi.

# Prototype chains

- Every object in Javascript has a **[[prototype]]**
- Technically this is a “hidden” property added to the object when it is defined or instantiated
- **note:**
  - **\_\_proto\_\_** is an accessor property that allows us to access the prototype
  - **never** use **\_\_proto\_\_** because it is risky and might be deprecated

**\_\_proto\_\_** --> È una proprietà accessoria che ti permette di accedere al prototipo di un oggetto.

Tuttavia, non è consigliato utilizzarla perché potrebbe essere deprecata in futuro e non è standardizzata in tutte le implementazioni di JavaScript

# Prototype chains

- When a message reaches an object
  - JavaScript will attempt to find a property in that object first
  - If it cannot find it then the message will be sent to the object's prototype and so on
- This works just like single parent inheritance in a class based language

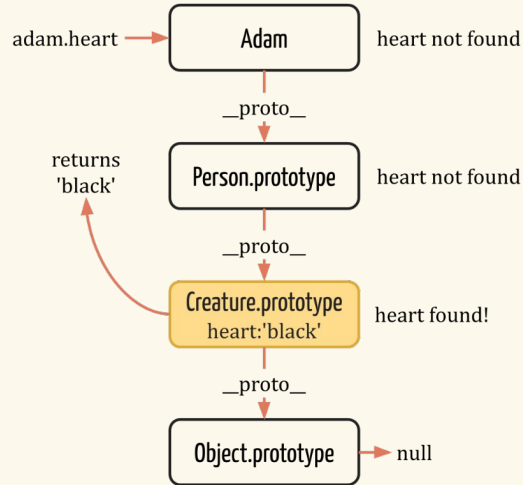
# Prototype chains

JAVASCRIPT SE NON TROVA UNA PROPRIETA IN UN OGGETTO, L'OGGETTO VA A CHIDERLO AL SUO PROTOTYPE

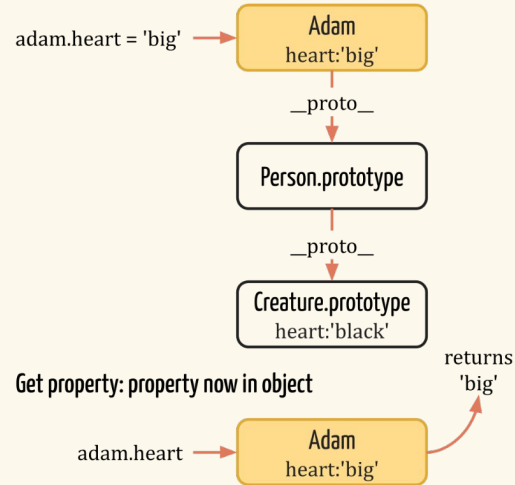
## Prototype chains

adam non ha la prop.  
prototype, perchè un oggetto!!

Get property: property not in object



Set property: new property on object



Get property: property now in object



© Shadi Lahham

Creating without constructors

# Creating objects without a constructor

```
let person = {  
  heart: 'black'  
};
```

```
let adam = Object.create(person);  
let sam = Object.create(person);
```

```
console.log(adam.heart); // black  
console.log(sam.heart); // black  
adam.heart = 'big';  
console.log(adam.heart); // big  
console.log(sam.heart); // black
```

Il metodo statico `Object.create()` crea un nuovo oggetto, utilizzando un oggetto esistente come prototipo dell'oggetto appena creato.

NEL MOMENTO IN CUI VERRA MODIFICATO, L'OGGETTO APPENA CREATO VERRA SOVRASCRITTO!!!

## References:

[`Object.create\(\)`](#)

# Never use `__proto__` directly

*// dog is a simple javascript object, not a constructor function*

```
let dog = {  
  happy: true  
};
```

*// create buck setting dog as the prototype*

*// Object.create() works a bit differently than using new*

```
let buck = Object.create(dog);
```

```
console.log(buck.happy); // true
```

```
console.log(buck.__proto__); // { happy: true }
```

*// \_\_proto\_\_ is an accessor property that allows us to access `[[prototype]]`*

*// it is unofficial, risky and might get deprecated*

*// used here only used here to explain how prototype inheritance works*

# Getting and setting the prototype

*// Object.getPrototypeOf() and Object.setPrototypeOf() are safer to use than \_\_proto\_\_*

```
console.log(Object.getPrototypeOf(buck) === dog); // true
Object.setPrototypeOf(buck, {}); // change the prototype of buck
```

## References:

[The JavaScript Object Paradigm and Prototypes Explained Simply](#)

[Object.prototype.\\_\\_proto\\_\\_](#)

[Object.getPrototypeOf\(\)](#)

[Object.setPrototypeOf\(\)](#)



Creating objects

# Creating Objects

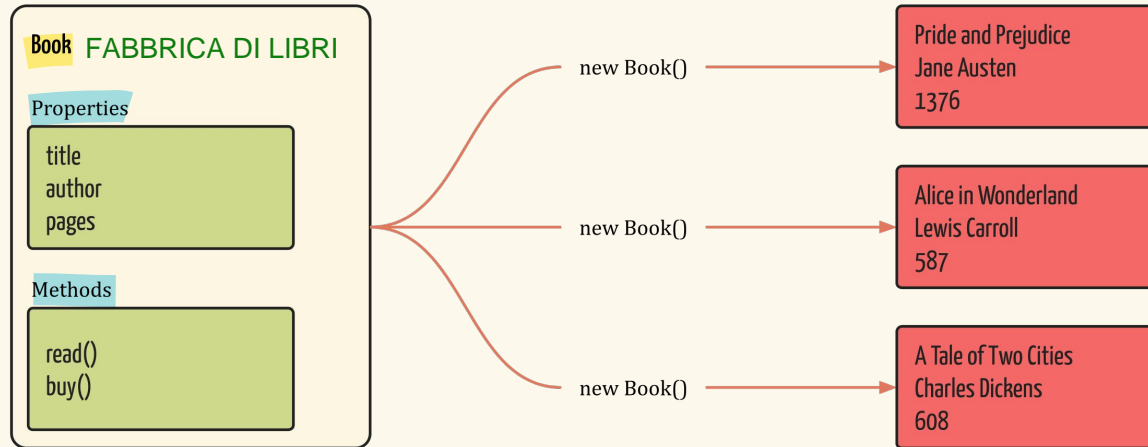
obbiettivo --> ci serve un meccanismo che ha già: proprietà metodi necessari (ex di un libro)

## Objects

### Instances

Book is defined once

Each instance has the same properties and methods



© Shadi Lahham

# Defining constructor functions and methods

PRIMA DI ES6

```
// constructor function
function Book(title, author, numPages) {
  // The properties of this object
  this.title = title;
  this.author = author;
  this.numPages = numPages;
  this.currentPage = 0;
}

// adding a method to the prototype object
Book.prototype.read = function () {
  this.currentPage = this.numPages;
  console.log('You read ' + this.numPages + ' pages!');
};

// instantiating a new Book object
let book = new Book('Robot Dreams', 'Isaac Asimov', 320);
book.read();
```

# How the reserved word new works

1. Creates a new Object
2. Creates and binds a new “this” to the object
3. Sets this new object's `[[prototype]]` to the value in the constructor function's prototype property
4. Executes the constructor function
5. Returns the newly created object

# Cleaner Constructors

*// better to pass a config object if many properties need to be set*

```
function Book(config) {  
  this.title = config.title;  
  this.author = config.author;  
  this.numPages = config.numPages;  
  this.currentPage = 0;  
}
```

```
let book = new Book({  
  title: 'Robot Dreams',  
  author: 'Isaac Asimov',  
  numPages: 320  
});
```

# Optional properties

*// some properties can be made optional by assigning default values*

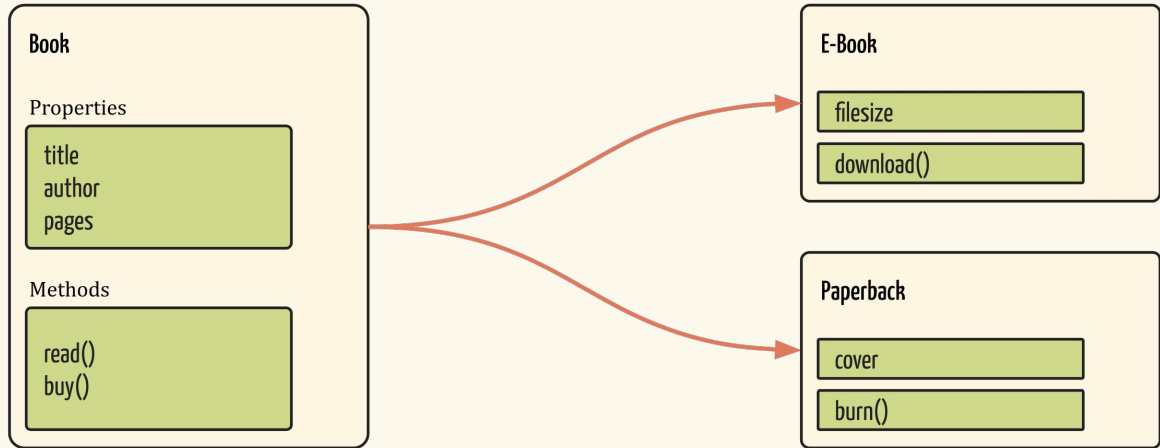
```
function Book(config) {  
  this.title = config.title || 'Untitled';  
  this.author = config.author || 'Unknown';  
  this.numPages = config.numPages || 100;  
  this.currentPage = 0;  
}
```

```
let book = new Book({  
  title: 'Robot Dreams',  
  numPages: 320  
});
```

# Extending Objects

## Inheritance

Objects can inherit properties and methods, implement parent methods differently and add new methods or properties



avranno le classiche proprietà di ogni libro (titolo, autore...) (comprare, leggere)

ma si distinguono in libri fisici con la copertina oppure e-book da leggere online

# Extending Objects

```
// constructor function
function PaperBack(title, author, numPages, cover) {
  Book.call(this, title, author, numPages); this è un valore (di solito L'OGGETTO DI RIFERIMENTO!!)
  this.cover = cover;
}

// extending the Book prototype object
PaperBack.prototype = Object.create(Book.prototype);

// adding a method to PaperBack's prototype object
PaperBack.prototype.burn = function () {
  console.log('Omg, you burnt all ' + this.numPages + ' pages');
  this.numPages = 0;
};

// instantiating a new PaperBack object
let paperback = new PaperBack('1984', 'George Orwell', 250, 'cover.jpg');
paperback.read();
paperback.burn();
```



# Operator instanceof

è un operatore BINARIO

```
// book is a book, also paperback is a Book
```

```
console.log(book instanceof Book); // true
```

```
console.log(paperback instanceof Book); // true
```

è true perché il book --> è stato creato dalla sua fabbrica Book (function constructor)

```
// but book is not a PaperBack
```

```
console.log(paperback instanceof PaperBack); // true
```

```
console.log(book instanceof PaperBack); // false
```

```
// both are instances of Object because of prototype inheritance
```

```
console.log(book instanceof Object); // true
```

```
console.log(paperback instanceof Object); // true
```

## Note

instanceof also works with Class inheritance

DOPO ES6!! (moderno JS)

# Modern Javascript classes

Una classe è essenzialmente una funzione costruttrice che definisce proprietà e metodi per gli oggetti creati con essa.

Si utilizza la parola chiave class.

# A modern Javascript class

```
class Person {  
  constructor(name) {    il costruttore si occupa di creare le istanze (un oggetto e settare le sue proprietà / metodi)  
    this.name = name;  
  }  
  speak() {  
    return 'My name is ' + this.name;  definizione di un metodo alla classe Persons, che ritorna una stringa con il nome  
  }  
}
```

```
class Teacher extends Person {  La classe Teacher estende (extends) --> la classe Person, quindi eredita le proprietà e i metodi della  
  speak() {                    classe base, ma aggiunge o modifica alcuni comportamenti.  
    return super.speak() + ', I am a teacher';  SUPER per ereditare dalla classe (Person) da cui sto ereditando!!  
  }  
}
```

```
let guy = new Teacher('James');  
console.log(guy.speak()); // output: My name is james, i am a teacher
```

# Using class

```
// Book class
```

```
class Book {  
  constructor(title, author, numPages) {  
    this.title = title;  
    this.author = author;  
    this.numPages = numPages;  
    this.currentPage = 0;  
  }  
}
```

```
  read() {  
    this.currentPage = this.numPages;  
    console.log('You read ' + this.numPages + ' pages!');  
  }  
}
```

```
// instantiating a new Book object
```

```
let book = new Book('Robot Dreams', 'Isaac Asimov', 320);  
book.read();
```

read() è un metodo della classe Book:

Quando viene chiamato, aggiorna la proprietà currentPage al valore totale delle pagine (numPages), simulando che si è letto l'intero libro.

# Extending with classes

*// Paperback class*

```
class Paperback extends Book {  
  constructor(title, author, numPages, cover) {  
    super(title, author, numPages);  
    this.cover = cover;  
  }  
  burn() {  
    console.log('Omg, you burnt all ' + this.numPages + ' pages');  
    this.numPages = 0;  
  }  
}
```

*// instantiating a new Paperback object*

```
let paperback = new Paperback('1984', 'George Orwell', 250, 'cover.jpg');  
paperback.read();  
paperback.burn();
```

# Classes in modern Javascript

- Using classes makes the code simpler and similar to other programming languages
- Classes are not just syntactic sugar but add a new features that were not possible with prototype inheritance in ES5

# Static properties and methods

*// static methods and properties are assigned to the class function itself, not to its "prototype"*

```
class Car {  
  constructor(color) {  
    this.color = color; // this refers to the new instance  
    Car.instances += 1;  
  }  
  
  static instances = 0;  
  
  static getInstances = function () {  
    return this.instances; // this refers to Car  
    // return Car.instances; // could have used Car instead of this  
  };  
}  
  
const cars = [new Car('red'), new Car('green'), new Car('orange')];  
console.log(Car.getInstances()); // 3
```

# Private properties and methods

le proprietà ed i metodi PRIVATI sono accessibili e utilizzabili solo all'interno della classe in cui sono state creati!!

```
class BankAccount {  
  #widthdrawLimit = 500; non usabile fuori dalla classe!  
  
  #limitedWithdraw(amount) { non usabile fuori dalla classe!  
    if (amount < 0) return 0;  
    if (amount > this.#widthdrawLimit) return this.#widthdrawLimit;  
    return amount;  
  }  
  
  withdraw(amount) { usabile anche fuori dalla classe!  
    return this.#limitedWithdraw(amount);  
  }  
}
```



# Private properties and methods

DIMOSTRAZIONE DELLA CLASSE PRIMA  
USANDO PROPRIETA E METODI PRIVATI!

```
let account = new BankAccount();
```

```
// can't access privates from outside of the class
```

```
account.#limitedWithdraw(123); // Error
```

```
account.#withdrawLimit = 1000; // Error
```

```
console.log(account.withdraw(-40));
```

```
console.log(account.withdraw(40));
```

```
console.log(account.withdraw(600));
```

References:

[Private properties](#)

# Shorthand property syntax

```
const name = 'james';  
const person = {  
  name, shorthand qui  
  age: 26  
};
```

```
// same as  
const name = 'james';  
const person = {  
  name: name,  
  age: 26  
};
```

```
// usage example  
function createPerson(name, age) {  
  return { name, age };  
}
```

# Dynamic properties

```
const getStatus = function () {  
  return 'employee';  
};  
  
const person = {  
  name: 'adam',  
  [getStatus() + '_' + 'number']: 13324  
};  
  
console.log(person.employee_number); // 13324
```

# Get & set

```
class Bank {  
  #money;  
  
  constructor(money) {  
    this.#money = money;  
  }  
  
  get money() {  
    return this.#money < 1000 ? this.#money : 'you are rich!';  
  }  
  
  set money(newMoney) {  
    this.#money = newMoney < 0 ? 0 : newMoney;  
  }  
}
```

# Get & set

```
const bank = new Bank(80);  
console.log(bank.money); // 80  
bank.money = 30;  
console.log(bank.money); // 30  
bank.money = -100;  
console.log(bank.money); // 0  
bank.money = 1300;  
console.log(bank.money); // 'you are rich!'
```

[get - JavaScript | MDN](#)

[set - JavaScript | MDN](#)

Your turn

# 1.DogSpeak

Add a method to the String prototype called `dogSpeak()` that works as follows:

```
let s = 'We like to learn';  
s.dogSpeak();
```

```
'Dogs are smart'.dogSpeak();
```

```
// Console output  
// We Like to Learn Woof!  
// Dogs are smart Woof!
```

**Think about the following question**

Is it a good idea to extend prototypes of built-in Javascript objects such as String, Array, etc?

## 2.Digital Age

- A Video has the following methods and properties
  - title (a string)
  - seconds (a number)
  - watch(x seconds [optional]) prints "You watched **X** seconds of '**TITLE**'" e.g. "You watched 120 seconds of 'Lord of the rings'". If x is missing prints "You watched all **SECONDS** seconds of '**TITLE**'" e.g. "You watched all 160 seconds of 'Lord of the rings'"
- A MusicVideo extends Video and has these extra methods and properties
  - artist (a string)
  - play() prints "You played '**TITLE**' by '**ARTIST**'" e.g. "You played 'Another Brick in the Wall' by 'Pink Floyd'"

Continues on next page >>>



## 2.Digital Age

- Use the prototype method, not classes, to write a constructors for Video and MusicVideo
  - The constructor functions accept a single config object
  - All arguments are optional, use defaults if missing
- Create an array that contains a mix of Video and MusicVideo instances
- Loop on the Array and for each item
  - call the watch() method
  - call the play() method only if it's a MusicVideo. Hint: Use [instanceof](#)
- Optional:
  - in a new folder, repeat the exercise using the class syntax rather than the prototype method
  - All behaviors should be identical

## 3.Strange Kebab

Add a method to the String prototype called toStrangeKebab() that transforms strings to kebab-case

```
// Given the following array
const source = [
  'MyNameIsMyPassportVerifyMe',
  'My Name    Is    My Passport Verify Me MMM',
  '  -- -My?Name&*is**my$$Passport???p??',
  'mY--name---  is- - 2023---',
  'mynameismypassport',
  '2022 my name is',
  '2024-my-name-is'
];

source.forEach(item => console.log(item.toStrangeKebab()));
```

Continues on next page >>>

# 3.Strange Kebab

The output should exactly match this:

```
my-name-is-my-passport-verify-me  
my-name-is-my-passport-verify-me-m-m-m  
my-name-is-my-passport-p  
m-y-name-is-2023  
mynameismypassport  
my-name-is  
my-name-is
```

## Note:

This implementation of kebab-case is not standard. It was invented for this exercise  
You might want to use regular expressions in your solution

[Regular expressions in JavaScript](#)

[Regex101](#)

# Group work

## 4. Do we know 'this'?

Create a short clear presentation to explain the 'this' keyword in Javascript

Refer to the following articles:

[Understanding the "this" keyword in JavaScript](#)

[Gentle Explanation of "this" in JavaScript](#)

Additional article:

[Understand JavaScript's "this" With Clarity, and Master It](#)

You can skip the case of Call & Apply until you do the next exercise

Once you do the next exercise, complete this one

Notes:

- English is preferred but Italian is also accepted
- You may do this exercise in groups of 2 people
- You may create a single presentation for this exercise and the next one

# 5.Call, Apply and Bind

Create a short clear presentation to explain Call, Apply and Bind in Javascript

Refer to the following articles:

[JavaScript .call\(\) .apply\(\) and .bind\(\) – explained to a total noob](#)

[What is the difference between call and apply in JavaScript ?](#)

And method documentation:

[Function.prototype.call\(\)](#)

[Function.prototype.apply\(\)](#)

[Function.prototype.bind\(\)](#)

Notes:

- English is preferred but Italian is also accepted
- You may do this exercise in groups of 2 people
- You may create a single presentation for this exercise and the previous one

# References

[JavaScript Constructor Function](#)

[JavaScript Prototype guide](#)

[JavaScript Prototypes](#)

[Object.create\(\)](#)

[Classes](#)

[static](#)

[Private properties](#)

# References

Older Prototype references

[JavaScript Prototype in Plain Language](#)

[A Plain English Guide to JavaScript Prototypes](#)

[Simple Inheritance with JavaScript](#)

[Prototypes in JavaScript](#)