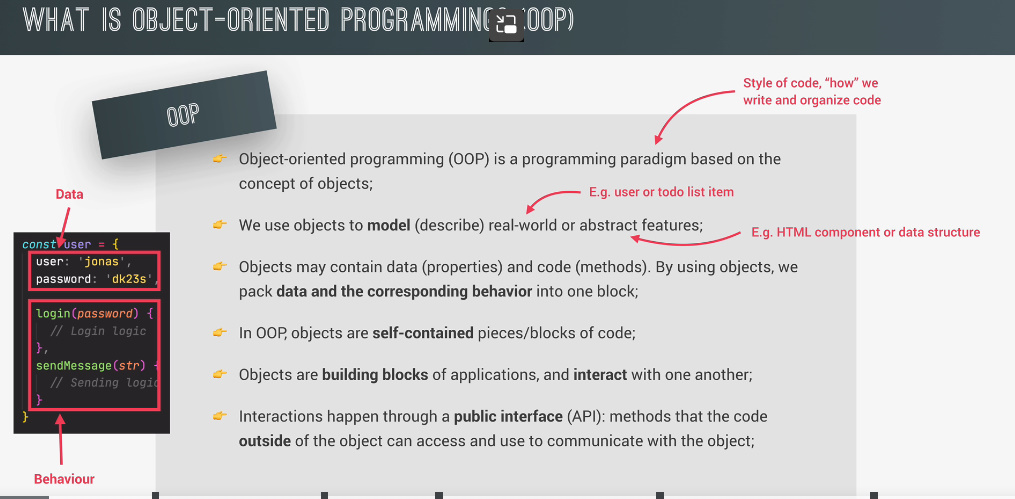
* **OOP** – paradigma de programare bazata pe obiecte



**Prototype**

* **Prototype** – obiect la care au referinta toate instantele unei functii constructor
* Fiecare obiect creat de noi, si deci si metoda(care tot e obiect) au fieldurile prototype si \_\_proto\_\_
* Insa, ele se comporta diferit in functie daca obiectul e de functie sau instanta a functiei. Asta e din cauza ca si functia are \_\_proto\_\_ al ei, creat de Object, si trebuie cumva sa avem acces la prototype impartasit de instante
* **prototype** – in cadrul functiilor, au referinta la obiectul comun la care au referinta toate instale functiei, iar in cazul instantelor functiei, e obiectul parinte, dar nu e accesibil prin instante!Nu il putem acces cu .prototype la instante!

**\_\_proto\_\_** - la functii e un object creat pentru toate functiile,creat de Object, cel care a creat functia constructor, la instante e obiectul comun impartasit de instante

* Metodele au fieldul numit prototype, obiectele \_\_proto\_\_

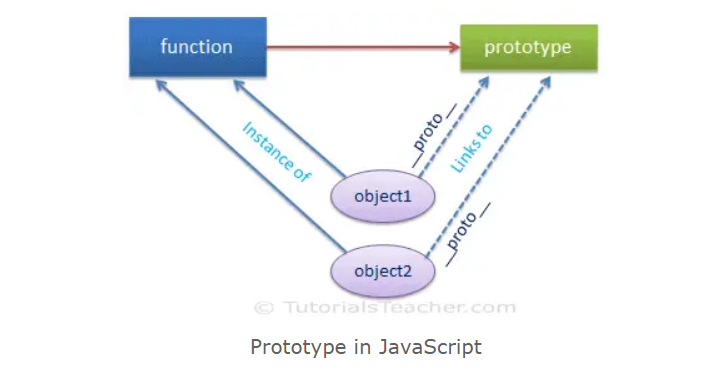
function Student(name){  
 this.name = name;  
}  
  
Student.prototype.age = 21;  
  
const *student1* = new Student('test1');  
const *student2* = new Student('test2');  
  
  
*console*.log(*student1*.prototype); //undefined  
*console*.log(*student2*.\_\_proto\_\_); //{ age: 21 }  
*console*.log(Student.prototype); //{ age: 21 }

Obiectele au fieldul prototype, dar nu e accesibil

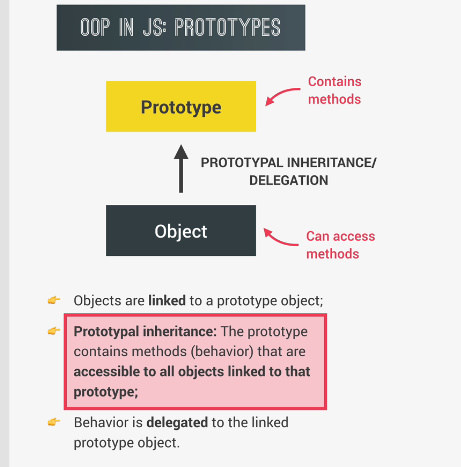
Insa, prototype de la functia constructor si \_\_proto\_\_ de la instante au referinte identice

*console*.log(*student1*.\_\_proto\_\_ === Student.prototype); //true

* Prototype este folosit pentru a permite inheritance in JS
* In js, putem defini noi fields oricand intr-un obiect, dar evident ele vor apartine doar lui
* function Student(name){  
   this.name = name;  
  }  
    
  const *student1* = new Student('test1');  
  const *student2* = new Student('test2');  
    
  *student1*.age = 21;  
    
  *console*.log(*student1*.age); //21  
  *console*.log(*student2*.age); //undefined
* Dar, poate noi vrem sa definim un nou field pentru toate obiectele, nu doar pentru unul.
* Orice function constructor object are grija sa creeze in noul obiect si field \_\_proto\_\_ pentru a oferi referinta la obiectul prototype la care are si ea referinta
* Scopul lui \_\_proto\_\_ este de a oferi fiecarui nou obiect creat pe baza la function constructor un field \_\_proto\_\_ cu un obiect la care toate noile instante vor avea acces.
* De ex, daca vrem sa adaugam un field la care sa aiba acces toate instantele, doar il declaram folosind obiectul functiei in prototype sau a unei instante cu \_\_proto\_\_, e tot aia
* function Student(name){  
   this.name = name;  
  }  
    
  const *student1* = new Student('test1');  
  const *student2* = new Student('test2');  
    
  Student.prototype.age = 21;  
  *student1*.\_\_proto\_\_.sex = 'm'  
    
  *console*.log(*student1*.\_\_proto\_\_.age); //21  
  *console*.log(*student2*.\_\_proto\_\_.sex); // m  
  *console*.log(Student.prototype.sex); // m
* Acum, fiecare obiect, daca nu gaseste un field in corpul sau, foloseste mereu \_\_proto\_\_ pentru a cauta fieldul negasit, ca referinta de rezerva oarecum, si daca nici acolo nu gaseste, va folosi \_\_proto\_\_ din \_\_proto\_\_ si tot asa
* function Student(name){  
   this.name = name;  
  }  
    
  const *student1* = new Student('test1');  
  const *student2* = new Student('test2');  
    
  Student.prototype.age = 21;  
    
  *console*.log(*student1*.age); //21  
  *console*.log(*student2*.age); // 21



* In loc de a folosi \_\_proto\_\_, putem folosi **Object.getPropertyOf(obj)**
* *console*.log(*Object*.getPrototypeOf(*student1*));
* **prototype field al functiei nu este propriu zis prototype al ei, ci este prototype al viitoarelo obiecte, de asta el nu e \_\_proto\_\_, cu prototype un functions, ca nu e al ei/**
* **Functia are si ea \_\_proto\_\_ al ei deja**
* Daca la un moment dat decidem sa modificam prototype al functiei, noile obiecte e logic ca vor avea noua referinta la prototype, dar nu si vechile. Functia constructor nu are de unde sti ce obiecte au fost deja create cu ea pentru a le modifica referinta la \_\_proto\_\_
* function Student(name){  
   this.name = name;  
  }  
    
  const *student1* = new Student('test1');  
    
  Student.prototype = {test: 'test'};  
    
  const *student2* = new Student('test2');  
    
    
  *console*.log(*student1*.test); //test  
  *console*.log(*student2*.test); //undefined  
  *console*.log(*student1*.\_\_proto\_\_ === *student2*.\_\_proto\_\_) //false



In JS inheritance nu functioneaza asa ca in alte limbaje de programare. O clasa nu copie propriu zis metodele si fields ale alte clase pe care o mosteneste, ci foloseste acel prototype field pentru a accesa obiecte parinti.



vedem ca orice array are metoda map(), dar metoda asta nu e propriu zis definita in array object,ci in obiectul prototype creat de functia constructor Array() si orice array va avea acces deci la aceste metode din acelasi obiect prototype

*console*.log([1,2,3].\_\_proto\_\_ === *Array*.prototype)

true

Nu intamplator metodele din array se acceseaza asa: **Array.prototype.filter()**

* Prototypes ne ajuta mult si la crearea metodelor

De ex:

function Student(name){  
 this.name = name;  
 this.getName = function(){  
 return this.name  
 }  
}

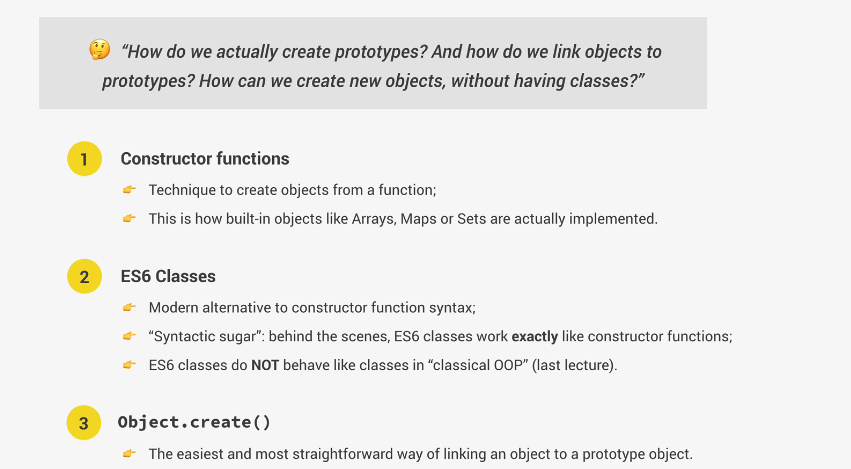
asta e o idee foarte rea de a defini metode. Fiecare obiect crea va avea o instanta noua a acestei metode, si deci 1000 de obiecte vor avea aceeasi metoda, dar creata de 1000 de ori, si asta va ocupa memorie enorma

function Student(name){  
 this.name = name;  
}  
  
Student.prototype.getAge = function(){  
 return this.name  
}

Acum e mult mai bine, toate obiectele vor aceeasi functie, doar ca deja this va diferentia

**Metode de a crea obiecte in JS**

1. Function constructor
2. Classes
3. Object.create()



Desi in noile versiuni de JS sunt clase, ele se comporta intern exact ca function constructor, doar ca fac codul mai frumos si usor de inteles.

**Constructor functions**

* Constructor function e exact ca orice function, doar e folosita cu new pentru a crea obiecte
* Conventia e de a incepe unei constructor function cu litera mare
* Operatorul new face urmatoarele chestii:

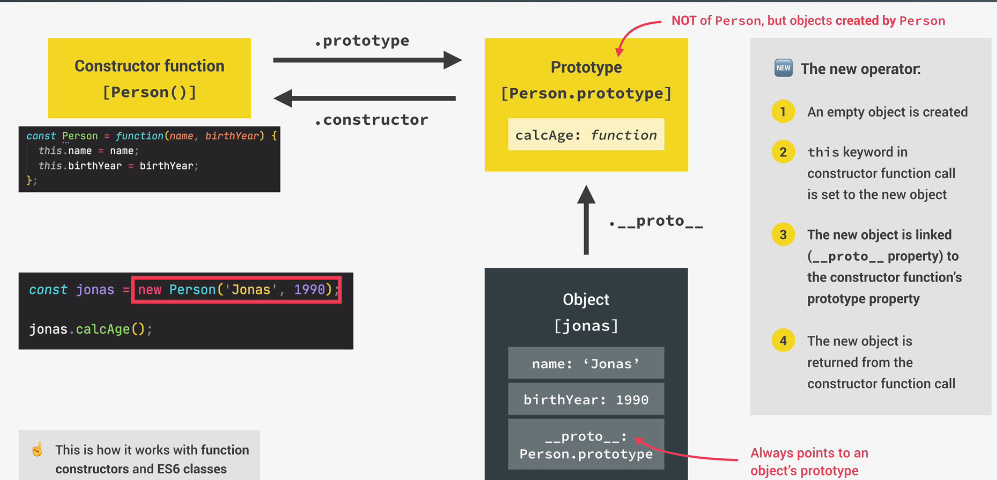
1. Creaza un empty object { }
2. functia constructor este apelata, insa this primeste referinta la obiectul nou creat
3. obiectul creat e legat de prototype, anume aici e creat fieldul \_\_proto\_\_
4. obiectul creat e returnat

* Deci, functia ia pur si simplu referinta la noul obiect si o stocheaza in this, adauga ce setam noi si returneaza si gata

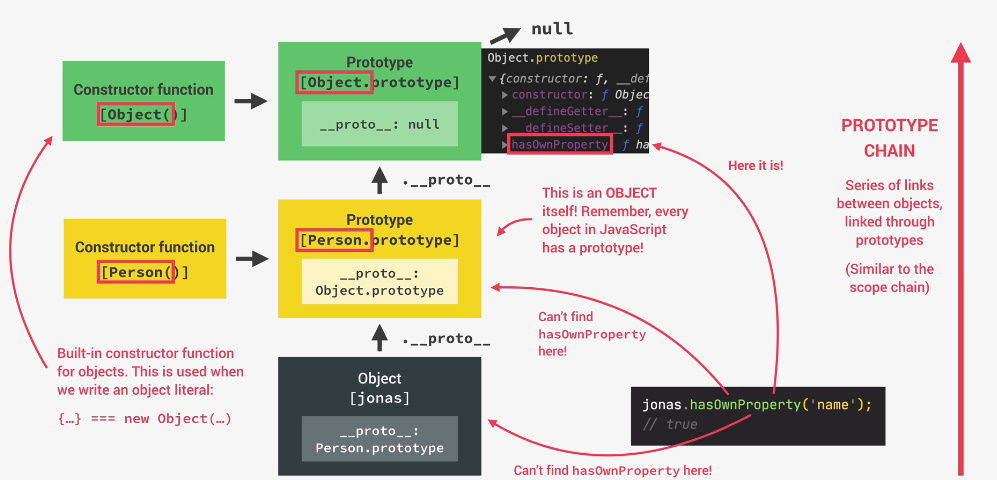
**Prototypal inheritance and prototype chain**

* prototype object al are un field .constructor, face face referinta propriu zis la functia constructor:
* function Student(name){  
   this.name = name;  
  }  
    
  *console*.log(Student === Student.prototype.constructor)

true



* Cand JS nu gaseste o metoda sau field in obiect, ea cauta automat in \_\_proto\_\_ si asta si se numeste **Prototype inheritance**
* **Prototype Chain** inseamna ca fiecare obiect are un field prototype, care contine obiectul care l-a creat. Nu doar functiile au prototype field, si obiectele au, doar ca noi nu avem acces la el! Anume asta si e chain



Functiile au referinta la prototype al instantelor care are si el prototype cu referinta la Object, si Object deja are prototype cu referinta la null, si aici si se incheie prtotype chain

Orice obiect are metoda hasOwnProperty(‘field’) pentru a verifica daca acest field apartine unui obiect, si aceasta metoda e definita un Object, deci se va apela \_\_proto\_\_ din Person, apoi se va ajunge in prototype object, si in el se va apela \_\_proto\_\_ si se va ajunge in Object si acolo se va gasi metoda, impreuna cu altele, gen toString(). Chiar si obiectele prototype au si ele \_\_protot\_\_, identic pentru toate obiectele prototype, ca sa ofere acces la metodele lui Object, ca toString()

* Deci, orice property object o sa aiba si el un \_\_proto\_\_ cu referinta la Object.prototype
* Orice obiect prototype, adica cel \_\_proto\_\_, are si el un \_\_proto\_\_ in el care are referinta la prototype creat de Object, si el e cel final, caci la el \_\_proto\_\_ e null.