* In JS clasele nu functioneaza ca in Java!
* Ele respecta aceleasi principii ca constructor function, doar ca fac codul mai cititibil
* clasele sunt tot functii defapt, doar ca scrise in alt mod
* Orice clasa trebuie neaparat sa aiba o metoda cu numele **constructor()**, dar care nu necesita keyword function
* In constructor, declaram fields exact ca in constructor functions

class Person{  
 constructor(name, age) {  
 this.name = name;  
 this.age = age;  
 }  
}  
  
const *pers* = new Person('test',20);

* Metodele deja le putem declara inafara la cosntructor, fara function:

class Person{  
 constructor(name, age) {  
 this.name = name;  
 this.age = age;  
 }  
  
 getAge(){  
 return this.age;  
 }  
}

Aceste metode sunt automat puse in property, deci e ca si cum am scrie property.getAge = ....

* Chiar mai putem adauga metode dupa vechiul mod, cu Person.prototype = function() { }
* **classes nu sunt hoisted!**
* **le putem stoca in variabile**
* **clasele sunt automat executate in strict mode**

**Get si Set**

* get si set sunt keyword ce se pun inaintea la metode
* Get ne permite sa primim o valoare, de parca ar fi un field:

class Person{  
 constructor(name, birthyear) {  
 this.name = name;  
 this.birthyear = birthyear;  
 }  
  
 get age(){  
 return 2023-this.birthyear;  
 }  
  
}  
  
const *pers* = new Person('test',2002);  
  
*console*.log(*pers*.age);

problema e ca metoda va adauga un field si gen nu putem avea o metoda de get sau set cu acelasi nume ca un field

class Person{  
 constructor(name, birthyear) {  
 this.name = name;  
 this.birthyear = birthyear;  
 }  
  
 get name(){  
 return this.name.toUpperCase();  
 }  
  
}

this.name va face referinta la field creat de metoda deja, si asa se executa la infinit

class Person{  
 constructor(name, birthyear) {  
 this.name = name;  
 this.birthyear = birthyear;  
 }  
  
 get nameUpperCase(){  
 return this.name.toUpperCase();  
 }  
  
}

asa deja va merge

* Set ne permite sa setam o valoare de parca ar fi un field
* class Person{  
   constructor(name, birthyear) {  
   this.name = name;  
   this.birthyear = birthyear;  
   }  
    
   get age(){  
   return 2023 - this.birthyear;  
   }  
   set age(age){  
   this.birthyear = 2023 - age;  
   }  
    
  }  
    
  const *pers* = new Person('test',2002);  
  *pers*.age = 25;  
    
  *console*.log(*pers*.age);

**get si set method pot avea acelasi nume, dar nu pot avea nume identice cu fieldurile! De altfel this.field va face deja referinta la ele.**

**Daca cream vreun field nou cu set, conventia spune sa punem \_, asa this.\_nouField**

**Static methods**

* **Static method** sunt cele care sunt atasate la functia constructor, nu in prototype
* De ex, Array.from() este o metoda statica, deoarece ea apartine functiei constructor Array, si nu in prototype la care orice object array are acces

*console*.log(*Array*.from(new *Set*([1,2,3,4])))  
*console*.log([1,2,3].from(new *Set*([1,2,3]))) //error!!! from() nu exista in array obj

* **Asa cream o metoda static in function constructor**

function Person(name){  
 this.name = name;  
}   
  
Person.staticMethod = function() { }

* Pentru clase, folosim **static** keyword
* class Person{  
   constructor(name, birthyear) {  
   this.name = name;  
   this.birthyear = birthyear;  
   }  
   static *staticMethod*(){  
     
   }  
  }
* Nu putem acces deci ce este static cu obiecte,caci ele nu mai fac parte din this sau prototype:
* class Person{  
   static *a* = 100;  
  }  
    
  const *pers* = new Person();  
  *console*.log(*pers*.*a*)



* **Nici in clasa nu merge sa le accesam cu this.**
* class Person{  
   static *a* = 100;  
   getA(){  
   return this.*a*;  
   }  
  }  
    
  const *pers* = new Person();  
  *console*.log(*pers*.getA())



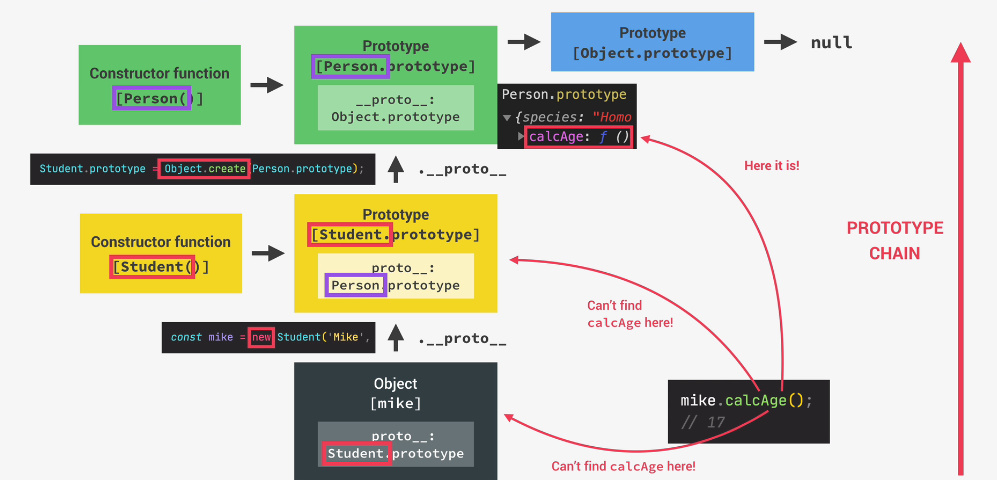
**Object.create**

* Functioneaza diferit de function constructor si class
* **Object.create(prototype)** – creaza un empty object, dar seteaza \_\_proto\_\_ ca obiectul trimis ca argument
* const *personProto* = {  
   hello(){  
   *console*.log('Hi!');  
   }  
  }  
    
  const *person1* = *Object*.create(*personProto*);  
    
  *person1*.hello();
* Defapt, e posibil si sa facem asa: Function.prototype = object,
* const *personProto* = {  
   hello(){  
   *console*.log('Hi!');  
   }  
  }  
    
  function Person(){  
    
  }  
  Person.prototype = *personProto*;  
    
  const *person1* = new Person();  
    
  *person1*.hello();

dar, cu Object.create() e mai usor

**Inheritance with prototype cu constructor functions**

* O metoda de a implementa inheritance este asa:
* function Animal(name){  
   this.name = name;  
  }  
  Animal.prototype.getName = function() {  
   return this.name;  
  }  
    
  function Dog(name,sound){  
   Animal.call(this, name);  
   this.sound = sound;  
  }  
    
  Dog.prototype = *Object*.create(Animal.prototype);  
  Dog.prototype.getSound = function(){  
   return this.sound;  
  }  
    
  const *dog* = new Dog('jack','ham ham');  
    
  *console*.log(*dog*.getName())  
  *console*.log(*dog*.getSound())



**Inheritance between classes**

* Folosim keyword ‘**extends’** si el deja se va ocupa de a face legaturile intre prototype
* extends are grija sa lege function cosntructo cu un nou obiect ce are \_\_proto\_\_ egal cu prototype al parintului
* class Car {  
   constructor(make, speed) {  
   this.make = make;  
   this.speed = speed;  
   }  
    
   accelerate() {  
   this.speed += 10;  
   *console*.log('Accelerating: ' + this.speed);  
   }  
   brake() {  
   this.speed -= 5;  
   *console*.log('Stopping: ' + this.speed);  
   }  
  }  
    
  class EV extends Car {  
   constructor(make, speed, charge) {  
   super(make, speed);  
   this.charge = charge;  
   }  
    
   chargeBattery(chargeTo) {  
   this.charge = chargeTo;  
   }  
   accelerate () {  
   this.speed += 20;  
   this.charge -= 1;  
    
   *console*.log(`${this.make} going at ${this.speed} km/h, with a charge of ${this.charge}%`);  
   }  
  }  
    
  const *tesla* = new EV('Tesla', 130, 50);  
    
  *tesla*.accelerate();  
  *tesla*.brake();

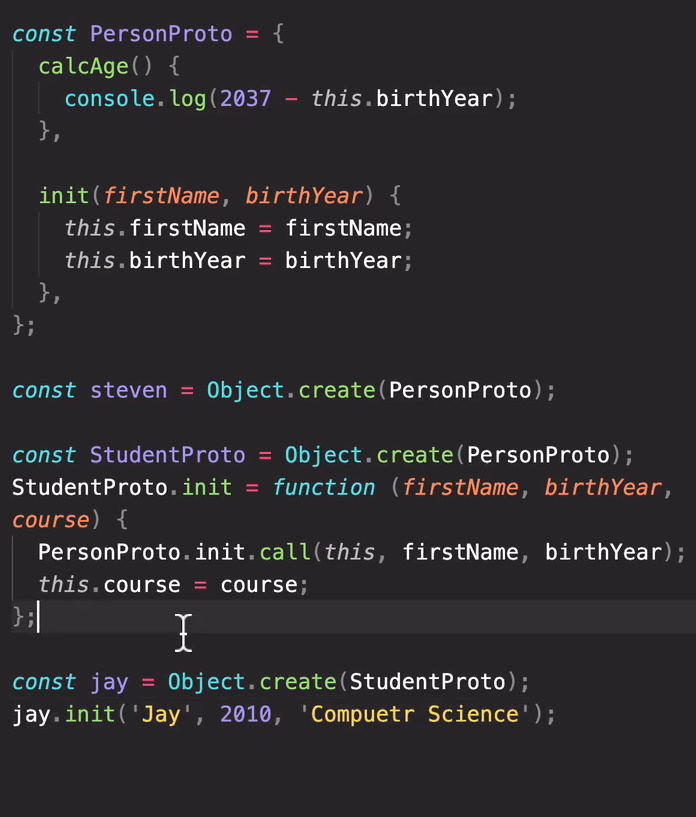
class EV extends Car e echivalent cu

**EV.prototype = Object.create(Car.prototype)**

utilizarea lui super() e obligatorie mereu daca avem extends!

* Daca nu folosim constructor in functia parinte, toate argumentele ce le trimitem la crearea obiectului vor fi trimisi automati in super()

**Inheritance cu Object.create cu chain objects**





const *PersonProto* = {  
 countBirthyear(){  
 return 2023 - this.age;  
 },  
 init(name, age){  
 this.name = name;  
 this.age = age;  
 }  
}  
  
const *StudentProto* = *Object*.create(*PersonProto*);  
*StudentProto*.init = function(name, age, school){  
 *PersonProto*.init.call(this,name, age);  
 this.school = school;  
}  
*StudentProto*.getInfo = function(){  
 *console*.log(`I'm ${this.name} and I'm ${this.age} and I study at ${this.school}`)  
}  
  
const *student1* = *Object*.create(*StudentProto*);  
*student1*.init('Eduard', 21, 'ASEM');  
*student1*.getInfo();

**PersonProto e ca o clasa, dar defapt e prototype pe care il vor avea copiii**

**StudentProto e ca si cum clasa Student ce mosteneste de la Person**

**Folosim init ca metoda constructor**

**Apoi, cand cream obiecte concrete, cream un obiect nou, doar ca StudentProto va fi prototype a lor, si folosim init() ca sa initializam dateles**

**Public fields in class**

* JS permite sa declaram fields nu doar in constructor, ci si in afara lui:
* Le declaram exact ca in Java, doar ca fara get sau const sau var, si le separam prin ;
* class Person{  
   age = 0;  
   nam= 100;  
   school;  
   getAge(){  
   return this.age;  
   }  
  }

tot folosim this pentru a le accesa.

* Ele nu sunt in prototype, ci in this!
* Putem declara un field ca static ca el sa faca parte din prototype
* class Person{  
   static *age* = 0;  
   nam= 100;  
   school;  
    
  }  
    
  person1 = new Person();  
  *console*.log(Person.*age*)



* Putem declara field in afara si sa il setam in constructor:
* class Person{  
   age = 0;  
   constructor(age){

*console*.log(this.age)  
 this.age = age;  
 }  
 getAge(){  
 return this.age;  
 }  
}  
  
const *person* = new Person(21);

cand constructorul il va accesa, el deja va fi definit

**Access modifiers**

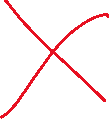
* In JS, nu exista propriu zis modificatori de acces
* Totul este public
* Totusi, exista o conventie ca fieldurile care incep cu \_ sunt **protected**, desi e doar conventie, nu le face protected la propriu
* **Private –** folosim # inaintea la numele fieldului. Fieldul chiar devine private

class Person{  
 #age = 100;  
 getAge(){  
 return this.#age;  
 }  
  
}  
  
person1 = new Person();  
*console*.log(person1.#age)



**SyntaxError: Private field '#age' must be declared in an enclosing class**

* **Atentie! Nu putem declara un private field cu this! this.#field face ca JS sa creada ca acest field deja e definit si noi il accesam:**
* class Person{  
   constructor(age){  
   this.#age = age;  
   }  
   getAge(){  
   return this.#age;  
   }  
  }  
    
  const *person* = new Person(21);  
    
  *console*.log(*person*.getAge())



Declararea unui private field mereu se face in afara constructorului

* Pentru a crea un private method, tot folosim #, desi inca multe browsere nu suporta asta