JavaScript

this - prototype chain - classi

Giovanni Iovino: giovanni.iovino@intecs.it

Contenuto:

Cosa rende Javascript differente dagli altri linguaggi di programmazione che supportano il paradigma di programmazione orientata agli oggetti.

JS

Tipi

I tipi primitivi in JavaScript sono:

- **String** - Una sequenza di caratteri
- **Number** - interi o float
- **BigInt** (nuovo in ECMAScript 2020) - Interi con precisione arbitraria
- **Boolean** - true e false
- **null** - un oggetto speciale che indica un oggetto nullo
- **undefined** - usato per indicare valori non definiti
- **Object**
- **Symbol** (nuovo in ES6) - Oggetto le cui istanze sono uniche e immutabili

Tutte le variabili che non ricadono in un tipo tra quelli elencati sono di tipo Object.

Un caso particolare sono le **funzioni**

```
// Esempi
const a = 20.05; // Number float

const b = 20; // Number int

const c = "Ciao"; // String

const d = true; // Boolean

var o = {  // Object

   p1: "Prop1", // Property di tipo String
   p2: 20 // Property di tipo Number

};

var ar = [1, 2, 3]; // Object
```

JS

This & That

Nella maggior parte dei linguaggi di programmazione orientati agli oggetti la keyword **this** rappresenta l'istanza dell'oggetto all'interno di un metodo dell'oggetto stesso.

In javascript non è sempre così, in particolare il valore a cui punta la keyword **this** dipende dal contesto in cui viene usata e da se si è in **strict mode o meno.** Nello specifico si distinguono due contesti principali:

- **Contesto globale: this** punta all'oggetto globale a prescindere se si è in strict mode o meno (es. nel browser l'oggetto globale è ***window***)
- **Contesto di funzione:** il valore di **this** dipende da come viene invocata la funzione.

```
// in browser -contesto globale
this === window // true
```

This & That / contesto di funzione

Nel contesto di funzione il valore di this dipende da come viene invocata la funzione:

1. **Function invocation** - o invocazione semplice:
   - Senza strict-mode **this** punta all'oggetto globale se non è impostato all'interno della funzione
   - In strict-mode: **this** se non è impostato all'interno della funzione non è definito

```
// Chiamata semplice - contesto globale
// no strict-mode

function f(){
 return this;
}

const that = f(); // that === window


// Chiamata semplice - contesto globale
// strict-mode

function f2(){

 'use strict'

 return this;

}

const that2 = f2(); // that === undefined
```

This & That / contesto di funzione

Nel contesto di funzione il valore di this dipende da come viene invocata la funzione:

2. **Object method invocation**: Quando una funzione è invocata come metodo di un oggetto **this** viene assegnato all'oggetto in cui è eseguita la funzione

```
// method invocation

const myobj = {
  n: 100,
  stampaN = function(){

    console.log(this.n);

  }
}

myobj.stampaN() // 100
```

This & That / contesto di funzione

Nel contesto di funzione il valore di this dipende da come viene invocata la funzione:

3. **Indirect invocation**: si usa il metodo built-in **call** disponibile in ogni oggetto di tipo **function** per eseguire un'assegnazione esplicita del valore di **this** durante l'esecuzione della funzione

sintassi: **func.call([thisArg[, arg1, arg2, ...argN]])**

```
// Indirect invocation

function stampaN(){

  console.log(this.n);

}

const myobj = {
  n: 100,

}

const myobj2 = {
  n: 200,

}

stampaN() // undefined - chiamata semplice

stampaN.call(myobj) // 100
stampaN.call(myobj2) // 200
```

This & That / contesto di funzione

Nel contesto di funzione il valore di this dipende da come viene invocata la funzione:

4. **Constructor invocation**: quando si usa l'operatore **new** per invocare una funzione come costruttore il valore di **this** all'interno della funzione viene associato all'istanza dell'oggetto creato

Vedremo perché si usano le funzioni come costruttori nelle prossime slides

```
// Constructor invocation

function MyObjConstructor(){

  this.n = 100; // this punta all'istanza
dell'oggetto creato se si usa new

}

const myobj = new MyObjConstructor();

myobj.n; // 100
```

JS

This & That / bind esplicito

In ES5 è stato introdotto il metodo built-in **bind** negli oggetti funzione.

Serve a creare una nuova funzione uguale a quella di origine dove però il valore di this è assegnato esplicitamente

sintassi: **func.bind(thisArg[, arg1[, arg2[, ...argN]]])**

```
function stampaN(){
  console.log(this.n);
}
const myobj = {
  n: 100,
}

const stampaNMyObj = stampaN.bind(myobj);
stampaNMyObj(); //100;
```

This & That / arrow functions

Nella prima parte del corso abbiamo accennato che le funzioni freccia hanno un comportamento diverso a livello di scope.

Infatti le funzioni freccia non creano un proprio contesto di esecuzione, pertanto il valore di **this** all'interno di una funzione freccia dipende dal contesto in cui sono definite. Ereditano il this dal contesto

```
const stampaN = ()=>{
  console.log(this.n);
}
const myobj = {
  n: 100,
}

stampaN() // undefined - chiamata semplice
stampaN.call(myobj) // undefined!!!
```


```
var nome = "Paperino"; //global var

function Persona(nome){
  this.nome = nome;

  this.stampa = function(){
    console.log(this.nome);
  }
  return this;
}

const p1 = new Persona("Topolino"); // constructor
p1.stampa(); //Topolino

const s = p1.stampa; // s punta a stampa di p1
s(); // Paperino perchè s è in contesto globale
s.call(p1); //Topolino - indirect invocation

const p2 = Persona("Pippo"); // chiamata semplice
p2.stampa(); // Pippo

console.log(nome); // Pippo - sovrascritta da
chiamata semplice

s.call(p1); // Topolino - indirect invocation
```

```
//… continua

const o ={
  nome: "Pluto"
}
o.stampa= s; // o.stampa punta a s che punta a
p1.stampa
o.stampa(); // Pluto -  method invocation
```

Prototype

Ogni volta che si dichiara una funzione o si crea un oggetto in Javascript viene **automaticamente creata e associata** la proprietà **prototype** all'oggetto creato

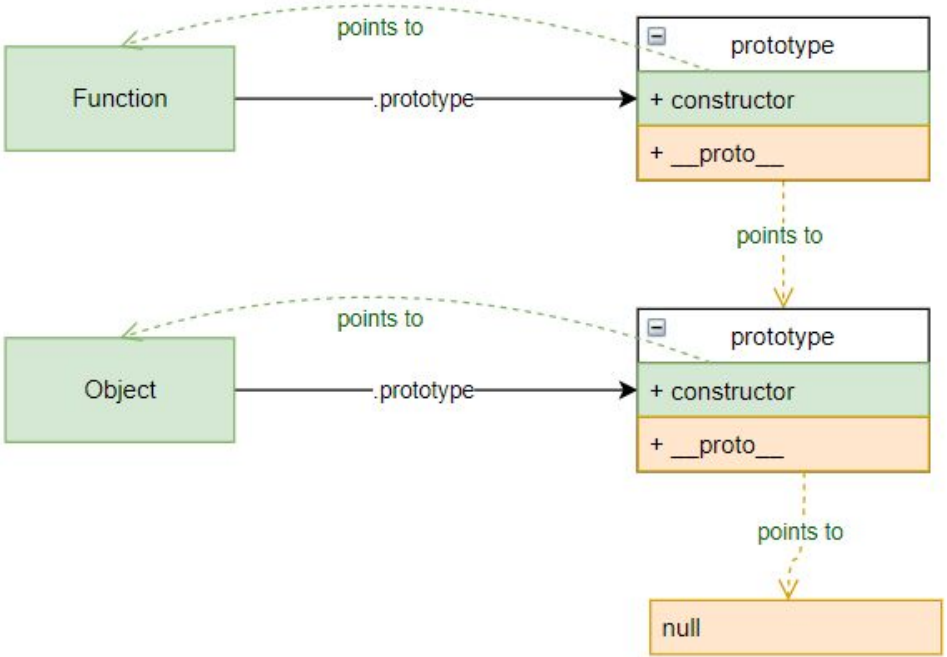
il prototype è composto da:

- un oggetto costruttore - che punta alla funzione che ha generato l'oggetto/funzione
- un oggetto __proto__ - che punta al prototype *interno dell'oggetto a cui è associato il prototype*

Nota: mentre per le funzioni la proprietà prototype è accessibile direttamente, questo non è possibile per gli altri tipi di oggetti. Per questi ultimi si può accedere al prototype tramite la proprietà nascosta **__proto__** o tramite il metodo built-in **Object.getPrototypeOf**

```
function Persona(nome, cognome){

  this.nome = nome;
  this.cognome = cognome;

}

const obj = { // obj is an object
  a: 10;
}

console.log(Persona.prototype); //
Object{constructor:..., __proto__:...}

console.log(obj.prototype); // undefined

console.log(obj.__proto__); // Object{constructor:...,
__proto__:...}
```

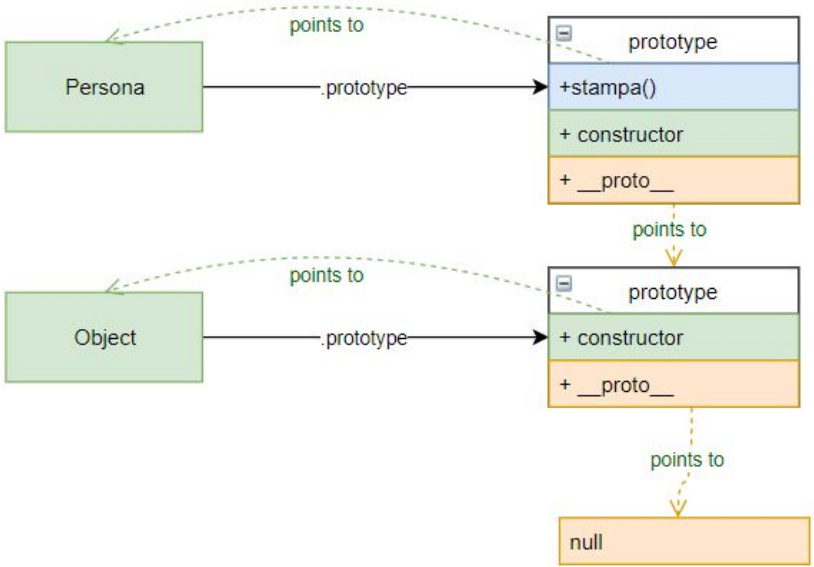
Prototype chain



Prototype

E’ possibile estendere l’oggetto **prototype** per aggiungere nuove proprietà.

```
function Persona(nome, cognome){

  this.nome = nome;
  this.cognome = cognome;

}


console.log(Persona.prototype); //
Object{constructor:..., __proto__:...


Persona.prototype.stampa = function(){
      console.log(this.nome + " "+ this.cognome);
}
```



Operatore new

L'operatore **new** crea un'istanza di un tipo di oggetto definito dall'utente o di uno dei tipi di oggetto nativi che ha una funzione costruttore. Quando si utilizza l'operatore new:

1. Un nuovo oggetto viene creato ed eredita dalla funzione usata come costruttore il prototype.
2. **this** è legato all'oggetto appena creato.
3. L'oggetto ritornato dalla funzione costruttore diventa il risultato dell'intera espressione new. Se la funzione costruttore non ritorna esplicitamente un oggetto, viene invece usato l'oggetto creato nello step 1. (Normalmente i costruttori non ritornano un valore, ma possono scegliere di farlo se vogliono sovrascrivere il processo di creazione di un normale oggetto).

```
function Persona(nome, cognome){

  this.nome = nome;
  this.cognome = cognome;

}


console.log(Persona.prototype); //
Object{constructor:..., __proto__:...


Persona.prototype.stampa = function(){
      console.log(this.nome + " "+ this.cognome);
}

const p1 = new Persona("Gio", "Iovi");

console.log(p1.__proto__ === Persona.prototype) //
true

const p2 = Persona("Gio", "Iovi");

// p2 is undefined
```

Operatore new

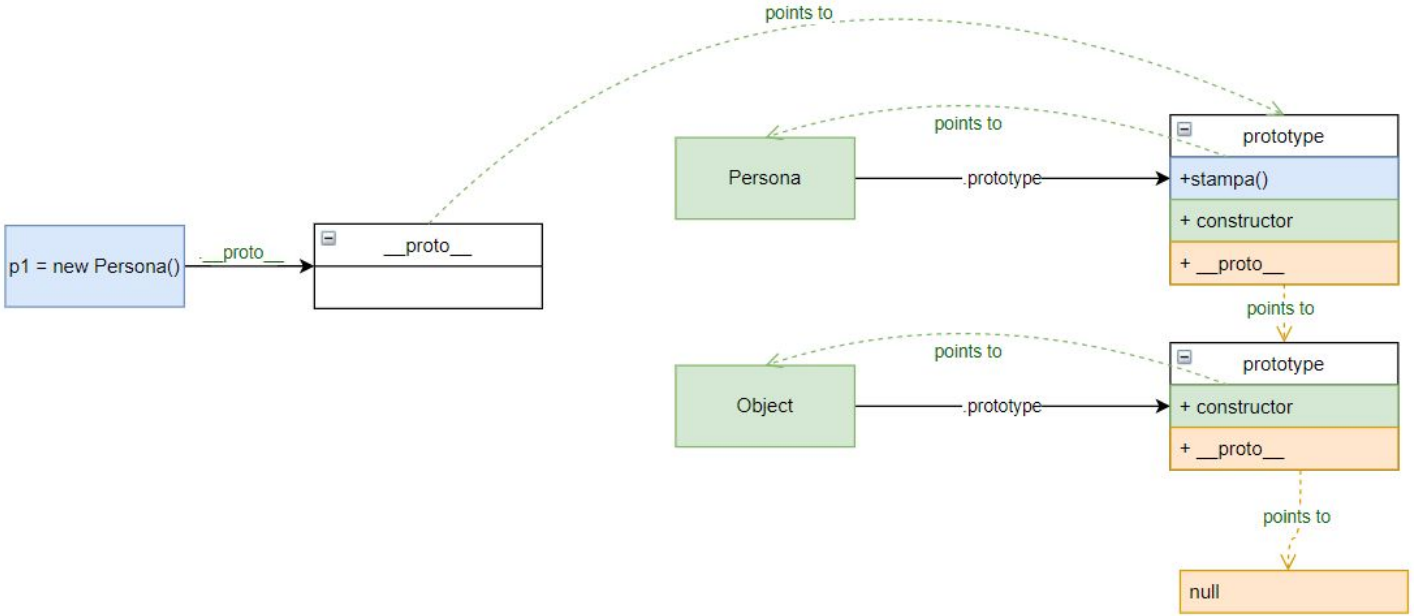
GLi oggetti creati con l'operatore new ereditano il prototype dalla funzione costruttore.

Lookup delle proprietà di un oggetto in Javascript:

1. Controlla se l'oggetto possiede la proprietà
2. In caso l'oggetto non abbia la proprietà cercata, controlla se il prototype dell'oggetto ha la proprietà
3. In caso contrario controlla ciclicamente se il prototype successivo nella catena di prototype possiede la proprietà finché questa non viene trovata o finché il prototype successivo non sia uguale a null

```
function Persona(nome, cognome){

  this.nome = nome;
  this.cognome = cognome;

}

console.log(Persona.prototype); //
Object{constructor:..., __proto__:...


Persona.prototype.stampaNome = function(){
      console.log(this.nome + " "+ this.cognome);
}

const p1 = new Persona("Gio", "Iovi");

p1.stampaNome(); // Gio Iovi

const p2 = new Persona("Super", "Gio");

p2.stampaNome = function(){
      console.log("E' un segreto");
}
p2.stampaNome(); // E' un segreto

p2.toString();  // ??
```

Prototype delle istanze di oggetti



Istanze - operatore instanceof

L'operatore **instanceof** controlla nella catena di prototype di un oggetto se è presente un determinato valore del costruttore restituendo un valore booleano

sintassi: x **instanceof** *Y*

```
function Persona(nome, cognome){

  this.nome = nome;
  this.cognome = cognome;

}

function Animale(nome){

  this.nome = nome;

}

const p1 = new Persona("Gio", "Iovi");

p1 instanceof Persona; // true

p1 instanceof Object; // true Persona -> Object
p1 instanceof Animale; // false Animale -> Object -> null
```

JS

Ereditarietà

A questo punto è intuibile come la creazione di oggetti e l'ereditarietà in javascript si basino interamente sulla prototype chain.

Quanto visto finora è sufficiente per utilizzare Javascript con un paradigma orientato agli oggetti, anche se non è immediato come in altri linguaggi

```
function Persona(nome, cognome){

  this.nome = nome;
  this.cognome = cognome;

}
// Aggiunge un metodo alla classe Persona
Persona.prototype.stampaNome = function(){
  console.log(this.nome+" "+this.cognome);
}

function Studente(nome, cognome, corso){

  Persona.call(this, nome, cognome);
  this.corso = corso;

}

// Assegna al prototype di Studente una copia del prototype di
Persona
Studente.prototype = Object.create(Persona.prototype);

// Cambia il valore del costruttore per farlo puntare alla
funzione Studente
Object.defineProperty(Studente.prototype, 'constructor', {
    value: Studente,
    enumerable: false,
    writable: true });

// Aggiunge un metodo alla classe studente
Studente.prototype.stampaCorso = function(){
  console.log(this.corso);
}

const p1 = new Persona("Gio", "Iovi");
const p2 = new Studente("Gio", "Iovi", "js");

p1 instanceof Persona; // true

p2 instanceof Studente; // true Studente
p2 instanceof Persona; // true Studente -> Persona
```

JS

Classi

In ES5 sono state introdotte delle nuove keyword(**class, extends, constructor, static)** che permettono la scrittura di classi e l'implementazione dell'ereditarietà in modo più semplice ed elegante.

Tuttavia si tratta di **syntactic sugar** ovvero javascript internamente utilizza sempre la prototype chain e la prototype inheritance per la creazione dei modelli!

```
class Persona{

  constructor(nome, cognome){

    this.nome = nome;
    this.cognome = cognome;

  }

  // Aggiunge un metodo alla classe Persona
  this.stampaNome = function(){

    console.log(this.nome+" "+this.cognome);

  }

}

class Studente extends Persona{

  constructor(nome, cognome, corso){

    super(nome,cognome);
    this.corso = corso;

  }

  // Aggiunge un metodo alla classe Studente
  this.stampaCorso = function(){

    console.log(this.corso);

  }

}

const p1 = new Persona("Gio", "Iovi");
const p2 = new Studente("Gio", "Iovi", "js");

p1 instanceof Persona; // true

p2 instanceof Studente; // true Studente
p2 instanceof Persona; // true Studente -> Persona
```