

# JAVASCRIPT

Riccardo Cattaneo

Lezione 7



# Le classi in javascript

Dopo aver visto cosa sono gli oggetti, facciamo un passo avanti e vediamo cosa è successo dalla versione ES6. E' stato «finalmente» introdotto il costrutto **class**.

Abbiamo detto che prima della versione ES6, per «simulare» la creazione di un oggetto in javascript, utilizzavamo la «funzione prototipo», riprendiamo l'ultimo esempio :

esempio1 > JS Persona.js > ...

```
1
2 function Persona(strNome, strCognome){
3     this.nome = strNome;
4     this.cognome = strCognome;
5 }
6
7 //Persona();
8 var persona1 = new Persona('Mario','Rossi');
9
10 console.log(persona1.nome);    // Mario
11 console.log(persona1.cognome); // Rossi
12
13 Persona.prototype.stampa = function(){
14     console.log('il nome è ' + this.nome);
15     console.log('il cognome è ' + this.cognome);
16 }
17
18 persona1.stampa();
19
```

Andiamo a vedere come si usa questo nuovo costrutto (class) e quali sono i suoi vantaggi. La sintassi è molto simile alle nostre funzioni prototipo :

```
esempio1 > JS ObjPersona.js > ...  
1  
2  class ObjPersona{  
3  
4  };  
5  
6  var persona1 = new ObjPersona();  
7  console.log(persona1); // Object  
8
```

# constructor

A differenza della funzione prototipo non gli posso passare dei parametri (non è una funzione e non ha le parentesi tonde) ma devo usare una funzione (o metodo) particolare all'interno della classe che si chiama **constructor** (costruttore) che verrà chiamato ogni volta che si vorrà creare un nuovo oggetto :

esempio1 > JS ObjPersona.js > ...

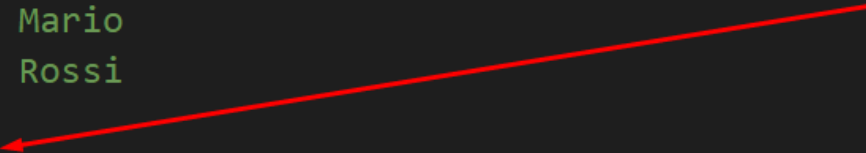
```
1
2  class ObjPersona{
3
4      constructor(strNome, strCognome){
5          this.nome = strNome;
6          this.cognome = strCognome;
7      }
8  };
9
10 var persona1 = new ObjPersona('Mario','Rossi');
11 console.log(persona1.nome);    // Mario
12 console.log(persona1.cognome); // Rossi
13
```

Utilizzare questa nuova sintassi è molto utile per «avvicinarci» a quella che oggi viene definita **programmazione ad oggetti**, ma sostanzialmente abbiamo fatto la stessa cosa delle funzioni prototipo.

Infatti se provo ad aggiungere tramite il prototype una funzione alla «classe», posso verificare che tutto funziona correttamente (l'unica cosa che cambia è che la funzione è **hoisting** mentre la classe non lo è). Per **hoisting** si intende quel meccanismo che muove la dichiarazione di variabili e funzioni all'interno dello scope della funzione in cui si trovano.

esempio1 > JS ObjPersona.js > ...

```
1
2 class ObjPersona{
3
4     constructor(strNome, strCognome){
5         this.nome = strNome;
6         this.cognome = strCognome;
7     }
8 };
9
10 var persona1 = new ObjPersona('Mario','Rossi');
11 console.log(persona1.nome);    // Mario
12 console.log(persona1.cognome); // Rossi
13
14 ObjPersona.prototype.stampaNominativo = function(){
15     console.log('il nome è ' + this.nome + ' ed il cognome è ' + this.cognome )
16 }
17
18 persona1.stampaNominativo();
```





# Ereditarietà

L'ereditarietà è una caratteristica che permette ad una classe di estendere le proprietà di altre classi. Ipotizziamo di avere una classe Persona :

```
esempio1 > JS ObjPersona.js > ...  
1  
2  class ObjPersona{  
3      |  
4      constructor(strNome, strCognome){  
5          |    this.nome = strNome;  
6          |    this.cognome = strCognome;  
7      |    }  
8  };  
9
```

potremmo ora voler creare un'altra classe Dipendente. Poiché il Dipendente è una Persona, la classe Dipendente dovrebbe essere basata su Persona, avendo accesso a tutti i metodi ed alle proprietà di Persona, in questo modo Dipendente può assumere tutti i comportamenti di base di una Persona.

La sintassi utilizzate per estendere un'altra classe è: `class Dipendente extends Persona`. Creiamo quindi la classe Dipendente che eredita da Persona e supponiamo di voler aggiungere la proprietà stipendio :

esempio1 > JS ObjPersona.js > ...

```
1
2  class ObjPersona{
3
4      constructor(strNome, strCognome){
5          this.nome = strNome;
6          this.cognome = strCognome;
7      }
8  };
9
10 class ObjDipendente extends ObjPersona{
11
12     constructor(strStipendio){
13         this.stipendio = strStipendio;
14     }
15 }
16
17 var dipendente1 = new ObjDipendente(2000);
18 console.log(dipendente1.nome);      // errore : ReferenceError: deve chiamare
19 console.log(dipendente1.cognome);   // il costruttore della superclasse
20 console.log(dipendente1.stipendio); // prima di accedere a "this"
21
```

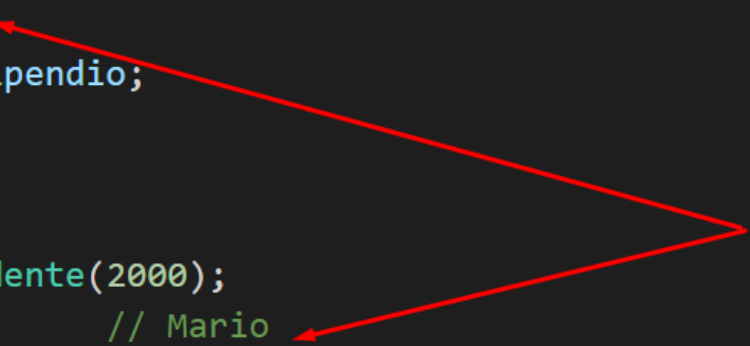
# super

ES6 introduce un'altra parola chiave **super** che fa riferimento alla classe genitore (superclasse) e viene utilizzata per chiamare i metodi corrispondenti della classe genitore all'interno della dichiarazione della sottoclasse.

Importante : JavaScript ci obbliga a chiamare **super** **prima** di utilizzare `this` in un costruttore

esempio1 > JS ObjPersona.js > ...

```
1
2  class ObjPersona{
3
4      constructor(strNome, strCognome){
5          this.nome = strNome;
6          this.cognome = strCognome;
7      }
8  };
9
10 class ObjDipendente extends ObjPersona{
11
12     constructor(strStipendio){
13         super('Mario', 'Rossi');
14         this.stipendio = strStipendio;
15     }
16 }
17
18 var dipendente1 = new ObjDipendente(2000);
19 console.log(dipendente1.nome);      // Mario
20 console.log(dipendente1.cognome);   // Rossi
21 console.log(dipendente1.stipendio); // 2000
```

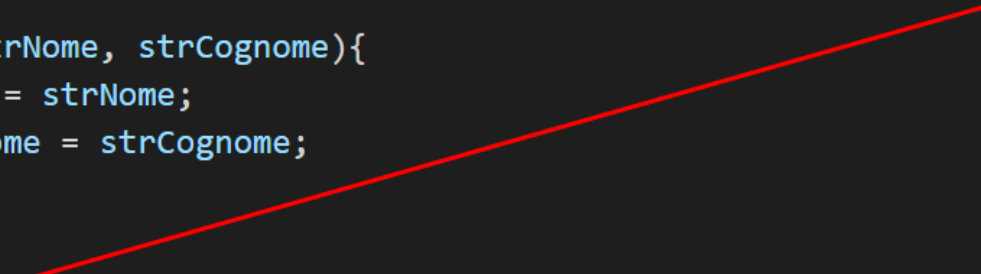


# Metodi

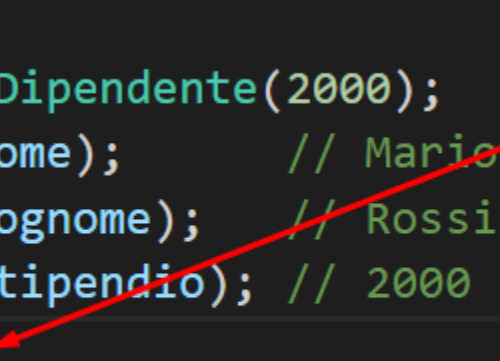
All'interno di una classe, oltre agli attributi posso inserire dei metodi. Supponiamo ad esempio di scrivere nella superclasse un metodo che stampa il nome ed il cognome :

esempio1 > JS ObjPersona.js > ...

```
1
2 class ObjPersona{
3
4     constructor(strNome, strCognome){
5         this.nome = strNome;
6         this.cognome = strCognome;
7     }
8
9     stampaDati(){
10         console.log('il nome è ' + this.nome + ' ed il cognome ' + this.cognome);
11     }
12 };
```



```
14 class ObjDipendente extends ObjPersona{
15
16     constructor(strStipendio){
17         super('Mario','Rossi');
18         this.stipendio = strStipendio;
19     }
20 }
21
22 var dipendente1 = new ObjDipendente(2000);
23 console.log(dipendente1.nome);           // Mario
24 console.log(dipendente1.cognome);        // Rossi
25 console.log(dipendente1.stipendio);      // 2000
26 dipendente1.stampaDati();
27
```




# override

In questo caso ho «ereditato» il metodo «stampaDati» che mi stampa la frase composta dal nome e dal cognome. Nel caso voglio modificare il metodo ereditato, posso «sovrascriverlo» (**override** appunto) ridefinendolo all'interno della sottoclasse in questo modo :

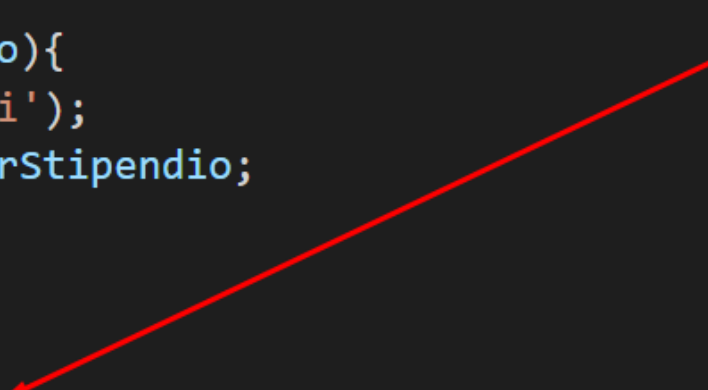


```
14 class ObjDipendente extends ObjPersona{
15
16     constructor(strStipendio){
17         super('Mario','Rossi');
18         this.stipendio = strStipendio;
19     }
20
21     stampaDati(){
22         console.log('il nome è ' + this.nome + ' e lo stipendio è ' + this.stipendio);
23     }
24 }
25
26 var dipendente1 = new ObjDipendente(2000);
27 console.log(dipendente1.nome);      // Mario
28 console.log(dipendente1.cognome);   // Rossi
29 console.log(dipendente1.stipendio); // 2000
30 dipendente1.stampaDati();
31
```



Se invece di sovrascrivere il metodo, voglio aggiungere del codice a quello già scritto, posso sfruttare il comando `super` e richiamare il metodo della superclasse in questo modo :

```
14  class ObjDipendente extends ObjPersona{
15
16      constructor(strStipendio){
17          super('Mario','Rossi');
18          this.stipendio = strStipendio;
19      }
20
21      stampaDati(){
22          super.stampaDati();
23          console.log('e lo stipendio è ' + this.stipendio);
24      }
25  }
```



# Esercizio

- Vedi lampadina.pdf